ECHILD

Tiến trình hoặc các tiến trình được chỉ định bởi đối số pid không tồn tại hoặc không phải là tiến trình con của tiến trình gọi.

TRỰC TIẾP

Tùy chọn WNOHANG không được chỉ định và tín hiệu đã được nhận trong khi chờ.

EINVAL

Đối số tùy chọn không hợp lệ.

Ví dụ, giả sử chương trình của bạn muốn lấy giá trị trả về của phần tử con cụ thể có pid 1742 nhưng trả về ngay lập tức nếu phần tử con chưa kết thúc. Bạn có thể mã hóa một cái gì đó tương tự như sau:

```
int trạng
thái; pid_t pid;

pid = waitpid (1742, &status, WNOHANG); nếu (pid
== -1) perror
          ("waitpid");
khác {
          printf ("pid=%d\n", pid);

          nếu (WIFEXITED (trạng thái))
                    printf ("Kết thúc bình thường với trạng thái thoát=%d\n",
                              WEXITSTATUS (trạng thái));

          if (WIFSIGNALED (trạng thái))
                    printf ("Bị giết bởi tín hiệu=%d%s\n",
                              WTERMSIG (trạng thái),
                              WCOREDUMP (trạng thái) ? " (lõi bị xóa)" : "");
}
```

Như một ví dụ cuối cùng, hãy lưu ý cách sử dụng wait( ) sau đây:

```
chờ (&status);
```

giống hệt với cách sử dụng sau của waitpid( ):

```
waitpid (-1, &trạng thái, 0);
```

Thậm chí còn nhiều hơn nữa sự linh hoạt khi chờ đợi

Đối với các ứng dụng yêu cầu tính linh hoạt cao hơn trong chức năng chờ đợi con, tiện ích mở rộng XSI cho POSIX định nghĩa và Linux cung cấp waitid( ):

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

int waitid (idtype_t idtype, id_t
                id,
                siginfo_t *infop, int
                tùy chọn);
```

Giống như wait() và waitpid( ), waitid() được sử dụng để chờ và lấy thông tin về thay đổi trạng thái (chấm dứt, dừng, tiếp tục) của một tiến trình con. Nó cung cấp nhiều tùy chọn hơn nữa, nhưng nó cung cấp cho chúng sự đánh đổi về độ phức tạp lớn hơn.

Giống như waitpid( ), waitid() cho phép nhà phát triển chỉ định những gì cần chờ. Tuy nhiên, waitid() thực hiện nhiệm vụ này không chỉ với một mà là hai tham số. Các đối số idtype và id chỉ định những phần tử con nào cần chờ, thực hiện cùng mục tiêu như đối số pid duy nhất trong waitpid( ). idtype có thể là một trong các giá trị sau:

P_PID
: Chờ phần tử con có pid khớp với id.

P_GID
: Chờ tiến trình con có ID nhóm tiến trình khớp với ID.

P_ALL
: Chờ bất kỳ phần tử con nào; id bị bỏ qua.

Đối số id là kiểu id_t hiếm khi thấy , là kiểu biểu diễn số nhận dạng chung. Kiểu này được sử dụng trong trường hợp các triển khai trong tương lai thêm giá trị idtype mới và được cho là cung cấp bảo hiểm lớn hơn rằng kiểu được xác định trước sẽ có thể chứa mã định danh mới được tạo. Kiểu này được đảm bảo đủ lớn để chứa bất kỳ pid_t nào. Trên Linux, các nhà phát triển có thể sử dụng nó như thể nó là pid_t—ví dụ, bằng cách truyền trực tiếp các giá trị pid_t hoặc hằng số số. Tuy nhiên, các lập trình viên cẩu kỳ có thể thoải mái ép kiểu.

Tham số options là OR nhị phân của một hoặc nhiều giá trị sau:

WEXITED
: Cuộc gọi sẽ chờ các mục con (được xác định bởi id và idtype) đã kết thúc.

WSTOPPED
: Cuộc gọi sẽ đợi những trẻ em đã dừng thực hiện để phản hồi khi nhận được tín hiệu.

WTIẾP TỤC
: Cuộc gọi sẽ chờ những trẻ tiếp tục thực hiện hành động phản hồi khi nhận được tín hiệu.

WNOHANG
: Cuộc gọi sẽ không bao giờ bị chặn, nhưng sẽ trả về ngay lập tức nếu chưa có tiến trình con phù hợp nào kết thúc (hoặc dừng hoặc tiếp tục).

KHÔNG CHỜ ĐỢI
: Cuộc gọi sẽ không xóa quy trình khớp lệnh khỏi trạng thái zombie. Quy trình có thể được chờ trong tương lai.

Khi chờ đợi thành công một phần tử con, waitid() sẽ điền vào tham số infop , tham số này phải trỏ đến một kiểu siginfo_t hợp lệ . Bố cục chính xác của cấu trúc siginfo_t là

cụ thể cho việc triển khai,* như ng một số ít trư ờng sẽ hợp lệ sau khi gọi đến waitid( ).
Nghĩa là, một lệnh gọi thành công sẽ đảm bảo các trư ờng sau đư ợc điền vào:

si_pid

PID của phần tử

con.

si_uid Uid của phần tử con.

si_code

Đặt thành một trong các giá trị sau: CLD_EXITED, CLD_KILLED, CLD_STOPPED hoặc CLD_CONTINUED để phản hồi khi tiến trình con kết thúc, chết thông qua tín hiệu, dừng thông qua tín hiệu hoặc tiếp tục thông qua tín hiệu. si_signo Đặt

thành

SIGCHLD.

si_status

Nếu si_code là CLD_EXITED, trư ờng này là mã thoát của tiến trình con. Nếu không, trư ờng này là số tín hiệu đư ợc gửi đến tiến trình con gây ra thay đổi trạng thái.

Nếu thành công, waitid() trả về 0. Nếu xảy ra lỗi, waitid() trả về -1 và errno đư ợc đặt thành một trong các giá trị sau:

ECHLD

Quy trình hoặc các quy trình đư ợc mô tả bằng id và idtype không tồn tại.

EINTR

WNOHANG không đư ợc thiết lập trong tùy chọn và tín hiệu đã làm gián đoạn quá trình thực thi.

EINVAL

Đối số tùy chọn hoặc sự kết hợp của đối số id và idtype không hợp lệ.

Hàm waitid() cung cấp ngữ nghĩa bổ sung, hữu ích không có trong wait() và waitpid( ). Đặc biệt, thông tin có thể truy xuất từ cấu trúc siginfo_t có thể khá có giá trị. Tuy nhiên, nếu không cần thông tin đó, có thể hợp lý hơ n khi sử dụng các hàm đơ n giản hơ n, đư ợc hỗ trợ trên nhiều hệ thống hơ n và do đó có thể chuyển sang nhiều hệ thống không phải Linux hơ n.

BSD muốn chơ i: wait3( ) và wait4( )

Trong khi waitpid( ) bắt nguồn từ AT&T's System V Release 4, BSD có lộ trình riêng và cung cấp hai hàm khác đư ợc sử dụng để chờ một hàm con thay đổi trạng thái:

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

* Thật vậy, cấu trúc siginfo_t rất phức tạp trên Linux. Để biết định nghĩa của nó, hãy xem /usr/include/bits/siginfo.h. Chúng ta sẽ nghiên cứu cấu trúc này chi tiết hơ n ở Chư ơ ng 9.

```
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3 (int *status,
             int options,
             struct rusage *rusage);

pid_t wait4 (pid_t pid,
             int *status,
             int options,
             struct rusage *rusage);
```

Số 3 và số 4 xuất phát từ thực tế là hai hàm này lần lượt là phiên bản ba và bốn tham số của wait( ).

Các hàm này hoạt động tương tự như waitpid( ), ngoại trừ tham số rusage . Lệnh gọi wait3() sau đây :

```
pid = wait3 (trạng thái, tùy chọn, NULL);
```

tương đương với lệnh gọi waitpid() sau :

```
pid = waitpid (-1, trạng thái, tùy chọn);
```

Và lệnh gọi wait4() sau đây :

```
pid = wait4 (pid, trạng thái, tùy chọn, NULL);
```

tương đương với lệnh gọi waitpid() này :

```
pid = waitpid(pid, status, options);
```

Nghĩa là, wait3( ) chờ bất kỳ phần tử con nào thay đổi trạng thái, và wait4( ) chờ phần tử con cụ thể được xác định bởi tham số pid thay đổi trạng thái. Đối số options hoạt động giống như waitpid( ).

Như đã đề cập trước đó, sự khác biệt lớn giữa các lệnh gọi này và waitpid() là tham số rusage . Nếu tham số này không phải là NULL, hàm sẽ điền thông tin về child vào con trỏ tại rusage . Cấu trúc này cung cấp thông tin về việc sử dụng tài nguyên của child:

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

struct rusage { struct
          timeval ru_utime; /* thời gian người dùng sử dụng */ struct timeval
          ru_stime; /* thời gian hệ thống sử dụng */ long ru_maxrss; /* kích thước
          tập hợp lưu trú tối đa */ long ru_ixrss; /* kích thước bộ nhớ chia
          sẻ */ long ru_idrss; /* kích thước dữ liệu không chia sẻ */
          long ru_isrss; /* kích thước ngăn xếp không chia sẻ */ long
          ru_minflt; /* thu hồi trang */ long ru_majflt; /* lỗi trang
          */ long ru_nswap; /* hoạt động hoán đổi */ long
          ru_inblock; /* khối hoạt động đầu vào */ long
          ru_oublock; /* khối hoạt động đầu ra */ long ru_msgsnd; /
          * tin nhắn đã gửi */
```

```
            dài ru_msgrcv; /* tin nhắn đã nhận */ dài
            ru_nsignals; /* tín hiệu đã nhận */ dài
            ru_nvcsw; /* chuyển đổi ngữ cảnh tự nguyện */ dài ru_nivcsw; /
            * chuyển đổi ngữ cảnh không tự nguyện */
};
```

Tôi sẽ đề cập sâu hơn về việc sử dụng tài nguyên ở chương tiếp theo.

Khi thành công, các hàm này trả về pid của tiến trình đã thay đổi trạng thái. Khi thất bại, chúng trả về -1 và đặt errno thành một trong các giá trị lỗi giống như waitpid( ) trả về.

Vì wait3( ) và wait4( ) không được định nghĩa theo POSIX,* nên chỉ nên sử dụng chúng khi thông tin sử dụng tài nguyên là quan trọng. Tuy nhiên, mặc dù thiếu chuẩn hóa POSIX, hầu như mọi hệ thống Unix đều hỗ trợ hai lệnh gọi này.

Khởi chạy và chờ một quy trình mới Cả ANSI C và POSIX đều định

nghĩa một giao diện kết hợp việc tạo ra một quy trình mới và chờ đợi quá trình đó kết thúc—hãy nghĩ về nó như việc tạo quy trình đồng bộ. Nếu một quy trình đang tạo ra một quy trình con chỉ để chờ ngay lập tức quá trình đó kết thúc, thì việc sử dụng giao diện này là hợp lý:

```
#define _XOPEN_SOURCE         /* nếu chúng ta muốn WEXITSTATUS, v.v. */
#include <stdlib.h>

int hệ thống (const char *command);
```

Hàm system ( ) được đặt tên như vậy vì quá trình gọi tiến trình đồng bộ được gọi là shelling ra hệ thống. Người ta thường sử dụng system( ) để chạy một tiện ích đơn giản hoặc tập lệnh shell, thường với mục tiêu rõ ràng là chỉ cần lấy giá trị trả về của nó.

Một lệnh gọi đến system() sẽ gọi lệnh được cung cấp bởi tham số lệnh , bao gồm bất kỳ đối số bổ sung nào. Tham số lệnh được thêm hậu tố vào các đối số /bin/sh -c.
Theo nghĩa này, tham số được truyền toàn bộ tới shell.

Khi thành công, giá trị trả về là trạng thái trả về của lệnh được cung cấp bởi wait( ). Do đó, mã thoát của lệnh được thực thi được lấy thông qua WEXITSTATUS. Nếu việc gọi /bin/sh không thành công, giá trị do WEXITSTATUS cung cấp sẽ giống với giá trị được trả về bởi exit(127). Vì lệnh được gọi cũng có thể trả về 127, nên không có phương pháp chắc chắn nào để kiểm tra xem bản thân shell có trả về lỗi đó hay không. Khi có lỗi, lệnh gọi sẽ trả về -1.

Nếu lệnh là NULL, system() trả về giá trị khác không nếu shell /bin/sh khả dụng và trả về 0 nếu không.

* wait3() đã được đưa vào Đặc tả UNIX đơn ban đầu, nhưng hiện đã bị xóa.

Trong quá trình thực thi lệnh, SIGCHLD bị chặn và SIGINT và SIGQUIT bị bỏ qua. Việc bỏ qua SIGINT và SIGQUIT có một số tác động, đặc biệt nếu system() đư ợc gọi bên trong một vòng lặp. Nếu gọi system() từ bên trong một vòng lặp, bạn nên đảm bảo rằng chư ơ ng trình kiểm tra đúng trạng thái thoát của con. Ví dụ:

```
LÀM {
          int ret;

          ret = hệ thống ("pidof rudderd"); nếu
          (WIFSIGNALED (ret) &&
               (WTERMSIG (nghỉ) == SIGINT ||
                WTERMSIG (ret) == SIGQUIT))
                    break; /* hoặc xử lý theo cách khác */
} trong khi (1);
```

Việc triển khai system() bằng fork( ), một hàm từ họ exec, và waitpid() là một bài tập hữu ích. Bạn nên tự mình thử làm, vì nó liên kết nhiều khái niệm của chư ơ ng này. Tuy nhiên, theo tinh thần hoàn thiện, đây là một ví dụ về triển khai:

```
/*
 *
   my_system - đồng bộ tạo ra và chờ lệnh * "/bin/sh -c <cmd>".

 *

 * Trả về -1 khi có lỗi bất kỳ, hoặc mã thoát khỏi * quy trình đã khởi chạy. Không
 chặn hoặc bỏ qua bất kỳ tín hiệu nào. */

int my_system (const char *cmd) {

          int trạng
          thái; pid_t pid;

          pid = fork ( );
          nếu (pid == -1)
                    trả về -1;
          nếu không thì nếu (pid == 0) {
                    const char *argv[4];

                    argv[0] = "sh";
                    argv[1] = "-c";
                    argv[2] = cmd;
                    argv[3] = NULL;
                    execv ("/bin/sh", argv);

                    thoát (-1);
          }

          nếu (waitpid (pid, &status, 0) == -1) trả
                    về -1; nếu
          không thì nếu (WIFEXITED (trạng
                    thái)) trả về WEXITSTATUS (trạng thái);

          trả về -1;
}
```

Lưu ý rằng ví dụ này không chặn hoặc vô hiệu hóa bất kỳ tín hiệu nào, không giống như hệ thống chính thức ( ). Hành vi này có thể tốt hơn hoặc tệ hơn, tùy thuộc vào tình huống của chương trình, nhưng việc để ít nhất SIGINT không bị chặn thường là thông minh vì nó cho phép lệnh được gọi bị ngắt theo cách mà người dùng thường mong đợi. Một triển khai tốt hơn có thể thêm các con trỏ bổ sung làm tham số, khi không phải NULL, biểu thị các lỗi hiện có thể phân biệt được với nhau. Ví dụ, người ta có thể thêm fork_failed và shell_failed.

thây ma

Như đã thảo luận trước đó, một tiến trình đã kết thúc, nhưng vẫn chưa được tiến trình cha của nó chờ đợi được gọi là "zombie". Các tiến trình zombie tiếp tục tiêu thụ tài nguyên hệ thống, mặc dù chỉ là một tỷ lệ nhỏ—đủ để duy trì một bộ khung đơn thuần của những gì chúng từng có. Các tài nguyên này vẫn còn để các tiến trình cha muốn kiểm tra trạng thái của các tiến trình con của chúng có thể lấy thông tin liên quan đến vòng đời và sự kết thúc của các tiến trình đó. Khi tiến trình cha thực hiện như vậy, hạt nhân sẽ dọn dẹp tiến trình mãi mãi và zombie không còn tồn tại nữa.

Tuy nhiên, bất kỳ ai đã sử dụng Unix trong một thời gian dài chắc chắn đã từng thấy các tiến trình zombie nằm xung quanh. Các tiến trình này, thường được gọi là ghost, có các tiến trình cha vô trách nhiệm. Nếu ứng dụng của bạn phân nhánh một tiến trình con, thì ứng dụng của bạn có trách nhiệm (trừ khi tiến trình con tồn tại trong thời gian ngắn, như bạn sẽ thấy ngay sau đây) phải chờ tiến trình con, ngay cả khi nó chỉ loại bỏ thông tin thu thập được. Nếu không, tất cả các tiến trình con của bạn sẽ trở thành ghost và tiếp tục tồn tại, làm tắc nghẽn danh sách tiến trình của hệ thống và tạo ra sự ghê tởm đối với việc triển khai cẩu thả của ứng dụng.

Tuy nhiên, điều gì xảy ra nếu tiến trình cha chết trước tiến trình con, hoặc nếu nó chết trước khi có cơ hội chờ các tiến trình con zombie của nó? Bất cứ khi nào một tiến trình kết thúc, hạt nhân Linux sẽ duyệt qua danh sách các tiến trình con của nó và chuyển tất cả chúng thành tiến trình init (tiến trình có giá trị pid là 1). Điều này đảm bảo rằng không có tiến trình nào không có tiến trình cha trực tiếp. Đến lượt mình, tiến trình init sẽ định kỳ chờ tất cả các tiến trình con của nó, đảm bảo rằng không có tiến trình nào vẫn là zombie quá lâu—không có bóng ma! Do đó, nếu tiến trình cha chết trước các tiến trình con của nó hoặc không chờ các tiến trình con của nó trước khi thoát, thì các tiến trình con cuối cùng sẽ được chuyển thành tiến trình cha cho init và được chờ đợi, cho phép chúng thoát hoàn toàn. Mặc dù việc làm như vậy vẫn được coi là một thông lệ tốt, nhưng biện pháp bảo vệ này có nghĩa là các tiến trình tồn tại trong thời gian ngắn không cần phải lo lắng quá mức về việc chờ tất cả các tiến trình con của chúng.

Người dùng và nhóm

Như đã đề cập trước đó trong chương này và thảo luận trong Chương 1, các quy trình được liên kết với người dùng và nhóm. Mã định danh người dùng và nhóm là các giá trị số được biểu diễn tương ứng bởi các kiểu C uid_t và gid_t . Ánh xạ giữa

giá trị số và tên có thể đọc đư ợc bằng con ngư ời—như trong ngư ời dùng root có uid 0—đư ợc thực hiện trong không gian ngư ời dùng bằng cách sử dụng các tệp /etc/passwd và /etc/group. Hạt nhân chỉ xử lý các giá trị số.

Trong hệ thống Linux, ID ngư ời dùng và nhóm của quy trình quyết định các hoạt động mà quy trình có thể thực hiện. Do đó, các quy trình phải chạy dư ới những ngư ời dùng và nhóm phù hợp. Nhiều quy trình chạy dư ới dạng ngư ời dùng gốc. Tuy nhiên, các thông lệ tốt nhất trong phát triển phần mềm khuyến khích học thuyết về quyền ít đặc quyền nhất , nghĩa là một quy trình phải thực thi với mức quyền tối thiểu có thể. Yêu cầu này là động: nếu một quy trình yêu cầu đặc quyền gốc để thực hiện một hoạt động vào giai đoạn đầu của vòng đời, như ng không yêu cầu các đặc quyền mở rộng này sau đó, thì quy trình đó phải hủy bỏ đặc quyền gốc càng sớm càng tốt. Vì mục đích này, nhiều quy trình—đặc biệt là những quy trình cần đặc quyền gốc để thực hiện một số hoạt động nhất định—thư ờng thao túng ID ngư ời dùng hoặc nhóm của chúng.

Trư ớc khi tìm hiểu cách thực hiện điều này, chúng ta cần tìm hiểu về sự phức tạp của ID ngư ời dùng và nhóm.

ID ngư ời dùng và nhóm thực, hiệu quả và đã lư u

Cuộc thảo luận sau đây tập trung vào ID ngư ời dùng, như ng tình huống cũng tư ơ ng tự đối với ID nhóm.

Trên thực tế, không chỉ có một mà là bốn ID ngư ời dùng đư ợc liên kết với một quy trình: ID ngư ời dùng thực, ID hiệu lực, ID đã lư u và ID hệ thống tệp. ID ngư ời dùng thực là uid của ngư ời dùng đã chạy quy trình ban đầu. Nó đư ợc đặt thành ID ngư ời dùng thực của tiến trình cha và không thay đổi trong khi gọi lệnh exec. Thông thư ờng, quy trình đăng nhập sẽ đặt ID ngư ời dùng thực của shell đăng nhập của ngư ời dùng thành ID của ngư ời dùng và tất cả các quy trình của ngư ời dùng tiếp tục mang ID ngư ời dùng này. Siêu ngư ời dùng (root) có thể thay đổi ID ngư ời dùng thực thành bất kỳ giá trị nào, như ng không ngư ời dùng nào khác có thể thay đổi giá trị này.

ID ngư ời dùng hiệu quả là ID ngư ời dùng mà quy trình hiện đang sử dụng. Xác minh quyền thư ờng kiểm tra theo giá trị này. Ban đầu, ID này bằng với ID ngư ời dùng thực, vì khi một quy trình phân nhánh, ID ngư ời dùng hiệu quả của tiến trình cha đư ợc thừa hư ởng bởi tiến trình con. Hơ n nữa, khi tiến trình phát lệnh gọi exec, ngư ời dùng hiệu quả thư ờng không thay đổi. Như ng, trong lệnh gọi exec, sự khác biệt chính giữa ID thực và ID hiệu quả xuất hiện: bằng cách thực thi nhị phân setuid (suid) , tiến trình có thể thay đổi ID ngư ời dùng hiệu quả của nó. Nói một cách chính xác, ID ngư ời dùng hiệu quả đư ợc đặt thành ID ngư ời dùng của chủ sở hữu tệp chư ơ ng trình. Ví dụ, vì tệp /usr/bin/passwd là tệp setuid và root là chủ sở hữu của tệp đó, nên khi shell của ngư ời dùng bình thư ờng tạo ra một quy trình để thực thi tệp này, quy trình sẽ lấy ID ngư ời dùng hiệu quả của root bất kể ngư ời dùng đang thực thi là ai.

Ngư ời dùng không có đặc quyền có thể đặt ID ngư ời dùng có hiệu lực thành ID ngư ời dùng thực hoặc đã lư u, như bạn sẽ thấy ngay sau đây. Siêu ngư ời dùng có thể đặt ID ngư ời dùng có hiệu lực thành bất kỳ giá trị nào.

ID ngư ời dùng đã lư u là ID ngư ời dùng hiệu lực ban đầu của tiến trình. Khi một tiến trình phân nhánh, tiến trình con sẽ thừa hư ởng ID ngư ời dùng đã lư u của tiến trình cha. Tuy nhiên, khi gọi lệnh exec, hạt nhân sẽ đặt ID ngư ời dùng đã lư u thành ID ngư ời dùng hiệu lực, do đó tạo bản ghi ID ngư ời dùng hiệu lực tại thời điểm thực hiện lệnh exec. Ngư ời dùng không có đặc quyền không đư ợc thay đổi ID ngư ời dùng đã lư u; siêu ngư ời dùng có thể thay đổi nó thành cùng giá trị với ID ngư ời dùng thực.

Mục đích của tất cả các giá trị này là gì? ID ngư ời dùng hiệu quả là giá trị quan trọng: đó là ID ngư ời dùng đư ợc kiểm tra trong quá trình xác thực thông tin xác thực của quy trình. ID ngư ời dùng thực và ID ngư ời dùng đã lư u hoạt động như các giá trị thay thế hoặc giá trị ID ngư ời dùng tiềm năng mà các quy trình không phải gốc đư ợc phép chuyển đổi sang và từ. ID ngư ời dùng thực là ID ngư ời dùng hiệu quả thuộc về ngư ời dùng thực sự đang chạy chư ơ ng trình và ID ngư ời dùng đã lư u là ID ngư ời dùng hiệu quả trư ớc khi một nhị phân suid gây ra thay đổi trong quá trình thực thi.

Thay đổi ID ngư ời dùng hoặc nhóm thực hoặc đã lư u

ID ngư ời dùng và nhóm đư ợc thiết lập thông qua hai lệnh gọi hệ thống:

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

int setuid(uid_t uid);
int setgid(gid_t gid);
```

Một lệnh gọi đến setuid() sẽ thiết lập ID ngư ời dùng hiệu quả của tiến trình hiện tại. Nếu ID ngư ời dùng hiệu quả hiện tại của tiến trình là 0 (root), ID ngư ời dùng thực và đã lư u cũng đư ợc thiết lập.
Ngư ời dùng root có thể cung cấp bất kỳ giá trị nào cho uid, do đó thiết lập cả ba giá trị ID ngư ời dùng thành uid. Ngư ời dùng không phải root chỉ đư ợc phép cung cấp ID ngư ời dùng thực hoặc đã lư u cho uid. Nói cách khác, ngư ời dùng không phải root chỉ có thể thiết lập ID ngư ời dùng có hiệu lực thành một trong các giá trị đó.

Nếu thành công, setuid() trả về 0. Nếu có lỗi, lệnh gọi trả về -1 và errno đư ợc đặt thành một trong các giá trị sau:

Uid

EAGAIN khác với ID ngư ời dùng thực và việc đặt ID ngư ời dùng thực thành uid sẽ đư a ngư ời dùng vư ợt quá giới hạn NPROC (chỉ định số lư ợng quy trình mà ngư ời dùng có thể sở hữu).

EPERM

Ngư ời dùng không phải là root và uid không phải là ID ngư ời dùng có hiệu lực cũng không phải là ID ngư ời dùng đã lư u.

Phần thảo luận trư ớc cũng áp dụng cho nhóm—chỉ cần thay thế setuid() bằng setgid( ) và uid bằng gid.

Thay đổi ID ngư ời dùng hoặc nhóm có hiệu lực

Linux cung cấp hai hàm bắt buộc theo POSIX để thiết lập ID ngư ời dùng và nhóm có hiệu lực của quy trình đang thực thi:

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

```
int seteuid (uid_t euid);
int setegid (gid_t egid);
```

Lệnh gọi seteuid() đặt ID ngư ời dùng có hiệu lực thành euid. Root có thể cung cấp bất kỳ giá trị nào cho euid. Ngư ời dùng không phải root chỉ có thể đặt ID ngư ời dùng có hiệu lực thành ID ngư ời dùng thực hoặc đã lư u. Nếu thành công, seteuid() trả về 0. Nếu thất bại, nó trả về -1 và đặt errno thành EPERM, điều này có nghĩa là tiến trình hiện tại không thuộc sở hữu của root và euid không bằng ID ngư ời dùng thực cũng không bằng ID ngư ời dùng đã lư u.

Lư u ý rằng trong trư ờng hợp nonroot, seteuid() và setuid() hoạt động giống nhau. Do đó, đây là thông lệ chuẩn và là một ý tư ởng hay khi luôn sử dụng seteuid( ), trừ khi quy trình của bạn có xu hư ớng chạy dư ới dạng root, trong trư ờng hợp đó, setuid() có ý nghĩa hơ n.

Phần thảo luận trư ớc cũng áp dụng cho nhóm—chỉ cần thay thế seteuid() bằng setegid( ) và euid bằng egid.

Thay đổi ID ngư ời dùng và nhóm, BSD Style BSD đã thiết lập giao diện riêng để thiết lập ID ngư ời dùng và nhóm. Linux cung cấp các giao diện này để tư ơ ng thích:

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

int setreuid (uid_t ruid, uid_t euid);
int setregid (gid_t rgid, gid_t egid);
```

Một lệnh gọi đến setreuid() sẽ đặt ID ngư ời dùng thực và ID ngư ời dùng có hiệu lực của một tiến trình thành ruid và euid tư ơ ng ứng. Chỉ định giá trị -1 cho bất kỳ tham số nào cũng sẽ giữ nguyên ID ngư ời dùng liên quan. Các tiến trình không phải root chỉ đư ợc phép đặt ID ngư ời dùng có hiệu lực thành ID ngư ời dùng thực hoặc đã lư u và ID ngư ời dùng thực thành ID ngư ời dùng có hiệu lực. Nếu ID ngư ời dùng thực bị thay đổi hoặc nếu ID ngư ời dùng có hiệu lực bị thay đổi thành giá trị không bằng giá trị ID ngư ời dùng thực trư ớc đó, ID ngư ời dùng đã lư u sẽ đư ợc thay đổi thành ID ngư ời dùng có hiệu lực mới. Ít nhất, đó là cách Linux và hầu hết các hệ thống Unix khác phản ứng với những thay đổi như vậy; hành vi này không đư ợc POSIX xác định.

Nếu thành công, setreuid() trả về 0. Nếu thất bại, nó trả về -1 và đặt errno thành EPERM, điều này có nghĩa là tiến trình hiện tại không thuộc sở hữu của root và euid không bằng ID ngư ời dùng thực cũng không bằng ID ngư ời dùng đã lư u hoặc ruid không bằng ID ngư ời dùng có hiệu lực.

Thảo luận trư ớc đó cũng áp dụng cho các nhóm—chỉ cần thay thế setreuid() bằng setregid( ), ruid bằng rgid và euid bằng egid.

Thay đổi ID ngư ời dùng và nhóm, theo phong cách HP-UX Bạn có thể cảm thấy tình hình đang trở nên điên rồ, như ng HP-UX, hệ thống Unix của Hewlett-Packard, cũng đã giới thiệu cơ chế riêng để thiết lập ID ngư ời dùng và nhóm của quy trình. Linux cũng tuân theo và cung cấp các giao diện sau:

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

int setresgid (uid_t ruid, uid_t euid, uid_t suid);
int setresgid (gid_t rgid, gid_t egid, gid_t sgid);
```

Lệnh gọi setresuid() sẽ đặt ID ngư ời dùng thực, có hiệu lực và đã lư u thành ruid, euid và suid . Chỉ định giá trị -1 cho bất kỳ tham số nào sẽ giữ nguyên giá trị của tham số đó.

Ngư ời dùng root có thể đặt bất kỳ ID ngư ời dùng nào thành bất kỳ giá trị nào. Ngư ời dùng không phải root có thể đặt bất kỳ ID ngư ời dùng nào thành ID ngư ời dùng thực, có hiệu lực hoặc đã lư u hiện tại. Khi thành công, setuid() trả về 0. Khi xảy ra lỗi, lệnh gọi trả về -1 và errno đư ợc đặt thành một trong các giá trị sau:

Uid

EAGAIN không khớp với ID ngư ời dùng thực và việc đặt ID ngư ời dùng thực thành uid sẽ đư a ngư ời dùng vư ợt quá giới hạn NPROC (chỉ định số lư ợng quy trình mà ngư ời dùng có thể sở hữu).

EPERM

Ngư ời dùng không phải là root và đã cố gắng thiết lập giá trị mới cho ID ngư ời dùng thực, có hiệu lực hoặc đã lư u không khớp với một trong các ID ngư ời dùng thực, có hiệu lực hoặc đã lư u hiện tại.

Thảo luận trư ớc đó cũng áp dụng cho các nhóm—chỉ cần thay thế setresuid() bằng setresgid( ), ruid () bằng rgid, euid () bằng egid và suid( ) bằng sgid.

Thao tác ID ngư ời dùng/nhóm ư a thích Các quy trình không phải

root nên sử dụng seteuid() để thay đổi ID ngư ời dùng có hiệu lực của chúng. Các quy trình root nên sử dụng setuid() nếu chúng muốn thay đổi cả ba ID ngư ời dùng và seteuid() nếu chúng muốn tạm thời thay đổi ID ngư ời dùng có hiệu lực. Các hàm này đơ n giản và hoạt động theo POSIX, tính đến đúng ID ngư ời dùng đã lư u.

Mặc dù cung cấp chức năng bổ sung, các hàm theo kiểu BSD và HP-UX không cho phép bất kỳ thay đổi hữu ích nào mà setuid() và seteuid() không cho phép.

Hỗ trợ cho ID ngư ời dùng đã lư u Sự

tồn tại của ID ngư ời dùng và nhóm đã lư u đư ợc quy định bởi IEEE Std 1003.1-2001 (POSIX 2001) và Linux đã hỗ trợ các ID này kể từ những ngày đầu của kernel 1.1.38. Các chư ơ ng trình chỉ đư ợc viết cho Linux có thể yên tâm về sự tồn tại của ID ngư ời dùng đã lư u. Các chư ơ ng trình đư ợc viết cho các hệ thống Unix cũ hơ n nên kiểm tra macro _POSIX_SAVED_IDS trư ớc khi thực hiện bất kỳ tham chiếu nào đến ID ngư ời dùng hoặc nhóm đã lư u.

Trong trư ờng hợp không có ID ngư ời dùng và nhóm đã lư u, các thảo luận trư ớc đó vẫn có hiệu lực; chỉ cần bỏ qua bất kỳ phần nào của quy tắc đề cập đến ID ngư ời dùng hoặc nhóm đã lư u.

Lấy ID ngư ời dùng và nhóm

Hai lệnh gọi hệ thống này lần lư ợt trả về ID ngư ời dùng và ID nhóm thực:

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

uid_t getuid (trống);
gid_t getgid (trống);
```

Chúng không thể lỗi. Tư ơ ng tự như vậy, hai lệnh gọi hệ thống này trả về ID ngư ời dùng và nhóm có hiệu lực tư ơ ng ứng:

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

uid_t geteuid (không có giá trị);
gid_t getegid (không có giá trị);
```

Cả hai lệnh gọi hệ thống này đều không thể bị lỗi.

Các phiên họp và nhóm quy trình

Mỗi quy trình là thành viên của một nhóm quy trình, là tập hợp một hoặc nhiều quy trình thư ờng đư ợc liên kết với nhau nhằm mục đích kiểm soát công việc. Thuộc tính chính của một nhóm quy trình là tín hiệu có thể đư ợc gửi đến tất cả các quy trình trong nhóm: một hành động duy nhất có thể chấm dứt, dừng hoặc tiếp tục tất cả các quy trình trong cùng một nhóm quy trình.

Mỗi nhóm quy trình đư ợc xác định bằng ID nhóm quy trình (pgid) và có một trư ởng nhóm quy trình. ID nhóm quy trình bằng với pid của trư ởng nhóm quy trình. Các nhóm quy trình tồn tại miễn là chúng còn một thành viên. Ngay cả khi trư ởng nhóm quy trình kết thúc, nhóm quy trình vẫn tiếp tục tồn tại.

Khi ngư ời dùng mới lần đầu đăng nhập vào máy, quy trình đăng nhập sẽ tạo một phiên mới bao gồm một quy trình duy nhất, shell đăng nhập của ngư ời dùng. Shell đăng nhập hoạt động như ngư ời dẫn đầu phiên. PID của ngư ời dẫn đầu phiên đư ợc sử dụng làm ID phiên. Phiên là tập hợp một hoặc nhiều nhóm quy trình. Phiên sắp xếp các hoạt động của ngư ời dùng đã đăng nhập và liên kết ngư ời dùng đó với một thiết bị đầu cuối điều khiển, là một thiết bị tty cụ thể xử lý I/O thiết bị đầu cuối của ngư ời dùng. Do đó, phiên chủ yếu là công việc của shell. Trên thực tế, không có gì khác thực sự quan tâm đến chúng.

Trong khi các nhóm quy trình cung cấp một cơ chế để giải quyết các tín hiệu cho tất cả các thành viên của chúng, giúp kiểm soát công việc và các chức năng shell khác trở nên dễ dàng, thì các phiên tồn tại để hợp nhất các lần đăng nhập xung quanh các thiết bị đầu cuối kiểm soát. Các nhóm quy trình trong một phiên đư ợc chia thành một nhóm quy trình nền trư ớc duy nhất và không có hoặc nhiều nhóm quy trình nền. Khi ngư ời dùng thoát khỏi một thiết bị đầu cuối, một SIGQUIT đư ợc gửi đến tất cả các quy trình trong nhóm quy trình nền trư ớc. Khi một thiết bị đầu cuối phát hiện ra một sự ngắt kết nối mạng, một SIGHUP đư ợc gửi đến tất cả các quy trình trong nhóm quy trình nền trư ớc. Khi ngư ời dùng nhập phím ngắt (thư ờng là Ctrl-C), một SIGINT đư ợc gửi đến tất cả các quy trình trong nhóm quy trình nền trư ớc. Do đó, phiên làm cho việc quản lý thiết bị đầu cuối và thông tin đăng nhập trở nên dễ dàng hơ n đối với shell.

Để xem lại, hãy nói rằng một ngư ời dùng đăng nhập vào hệ thống và shell đăng nhập của cô ấy, bash, có pid 1700. Phiên bản bash của ngư ời dùng hiện là thành viên duy nhất và là ngư ời lãnh đạo của một nhóm quy trình mới, với ID nhóm quy trình là 1700. Nhóm quy trình này nằm trong một phiên mới có ID phiên là 1700 và bash là thành viên duy nhất và là ngư ời lãnh đạo của phiên này. Các lệnh mới mà ngư ời dùng chạy trong shell chạy trong các nhóm quy trình mới trong phiên 1700. Một trong những nhóm quy trình này—nhóm đư ợc kết nối trực tiếp với ngư ời dùng và kiểm soát thiết bị đầu cuối—là nhóm quy trình nền trư ớc. Tất cả các nhóm quy trình khác là nhóm quy trình nền sau.

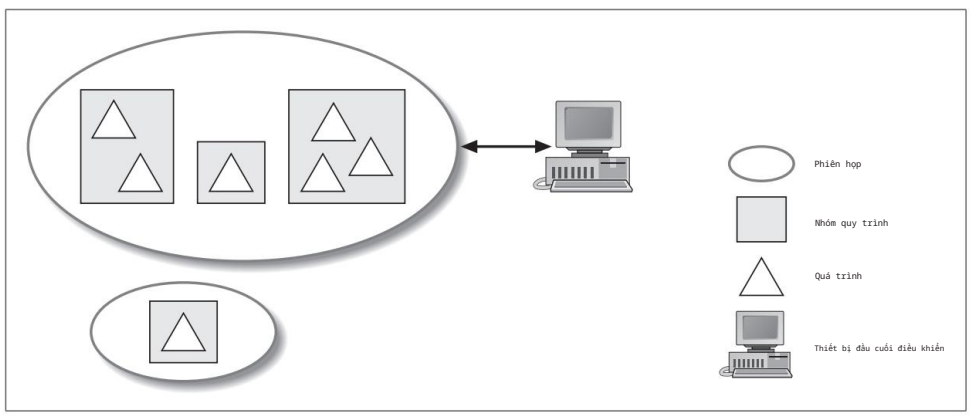
Trên một hệ thống nhất định, có nhiều phiên: một phiên cho mỗi phiên đăng nhập của ngư ời dùng và các phiên khác cho các quy trình không liên kết với các phiên đăng nhập của ngư ời dùng, chẳng hạn như daemon. Daemon có xu hư ớng tạo phiên riêng của chúng để tránh các vấn đề liên kết với các phiên khác có thể thoát.

Mỗi phiên này chứa một hoặc nhiều nhóm quy trình và mỗi nhóm quy trình chứa ít nhất một quy trình. Các nhóm quy trình chứa nhiều hơ n một quy trình thư ờng đang triển khai kiểm soát công việc.

Một lệnh trên shell như thế này:

```
$ cat ship-inventory.txt | grep booty | sắp xếp
```

kết quả là một nhóm quy trình chứa ba quy trình. Theo cách này, shell có thể báo hiệu cả ba quy trình cùng một lúc. Vì ngư ời dùng đã nhập lệnh này trên bảng điều khiển mà không có dấu thăng theo sau, chúng ta cũng có thể nói rằng nhóm quy trình này ở phía trư ớc. Hình 5-1 minh họa mối quan hệ giữa các phiên, nhóm quy trình, quy trình và thiết bị đầu cuối điều khiển.



Hình 5-1. Mối quan hệ giữa các phiên, nhóm quy trình, quy trình và thiết bị đầu cuối điều khiển

Linux cung cấp một số giao diện để thiết lập và truy xuất nhóm phiên và quy trình liên quan đến một quy trình nhất định. Chúng chủ yếu đư ợc sử dụng cho shell, như ng cũng có thể hữu ích cho các quy trình như daemon muốn thoát khỏi công việc của nhóm phiên và quy trình.

Shells tạo phiên mới khi

đăng nhập. Chúng thực hiện điều này thông qua một lệnh gọi hệ thống đặc biệt, giúp tạo phiên mới dễ dàng:

```
#include <unistd.h>

pid_t setsid (trống);
```

Một lệnh gọi đến setsid( ) tạo ra một phiên mới, giả sử rằng tiến trình chư a phải là một nhóm tiến trình trư ởng. Tiến trình gọi đư ợc đặt làm nhóm trư ởng và là thành viên duy nhất của phiên mới, không có tty điều khiển. Lệnh gọi cũng tạo ra một nhóm tiến trình mới bên trong phiên và biến tiến trình gọi thành nhóm trư ởng và là thành viên duy nhất của tiến trình. ID của nhóm tiến trình và phiên mới đư ợc đặt thành pid của tiến trình gọi.

Nói cách khác, setsid( ) tạo một nhóm quy trình mới bên trong một phiên mới và biến quy trình gọi thành ngư ời dẫn đầu của cả hai. Điều này hữu ích cho các daemon, không muốn là thành viên của các phiên hiện có hoặc có các thiết bị đầu cuối điều khiển và cho các shell, muốn tạo một phiên mới cho mỗi ngư ời dùng khi đăng nhập.

Khi thành công, setsid() trả về ID phiên của phiên mới đư ợc tạo. Khi lỗi, lệnh gọi trả về -1. Mã errno khả thi duy nhất là EPERM, cho biết rằng quy trình hiện là ngư ời dẫn đầu nhóm quy trình. Cách dễ nhất để đảm bảo rằng bất kỳ quy trình nào không phải là ngư ời dẫn đầu nhóm quy trình là phân nhánh, yêu cầu cha kết thúc và yêu cầu con thực hiện setsid ( ). Ví dụ:

```
pid_t pid;

pid = fork ( );
nếu (pid == -1)
         { perror ("fork");
         trả về
-1; } nếu không thì nếu
         (pid != 0) thoát (EXIT_SUCCESS);

nếu (setsid ( ) == -1)
         { lỗi ("setsid");
         trả về -1;
}
```

Việc lấy ID phiên hiện tại, mặc dù ít hữu ích hơ n, vẫn có thể thực hiện đư ợc:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getsid (pid_t pid);
```

Một lệnh gọi đến getsid( ) trả về ID phiên của tiến trình đư ợc xác định bởi pid. Nếu tham số pid là 0, lệnh gọi trả về ID phiên của tiến trình gọi. Khi có lỗi, lệnh gọi trả về -1. Giá trị errno khả thi duy nhất là ESRCH, cho biết pid không tư ơ ng ứng với

quy trình hợp lệ. Lưu ý rằng các hệ thống Unix khác cũng có thể đặt errno thành EPERM, cho biết pid và quy trình gọi không thuộc cùng một phiên; Linux không trả về lỗi này và vui vẻ trả về ID phiên của bất kỳ quy trình nào.

Việc sử dụng rất hiếm và chủ yếu dùng cho mục đích chẩn đoán:

```
pid_t sid;

sid = getsid (0);
nếu (sid ==
        -1) perror ("getsid"); /* không thể thực hiện được */
khác
        printf ("ID phiên của tôi=%d\n", sid);
```

Cuộc gọi hệ thống nhóm quy trình

Lệnh gọi setpgid( ) sẽ đặt ID nhóm quy trình của quy trình được xác định bởi pid thành pgid:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid); Quy
```

trình hiện tại được sử dụng nếu đối số pid là 0. Nếu pgid là 0, ID quy trình của quy trình được xác định bởi pid sẽ được sử dụng làm ID nhóm quy trình.

Khi thành công, setpgid() trả về 0. Thành công phụ thuộc vào một số điều kiện:

- Tiến trình được xác định bởi pid phải là tiến trình gọi hoặc là tiến trình con của tiến trình gọi, chưa phát lệnh gọi exec và đang trong cùng phiên với tiến trình gọi.
- Tiến trình được xác định bởi pid không được là tiến trình dẫn đầu phiên.
- Nếu pgid đã tồn tại, thì nó phải nằm trong cùng phiên với tiến trình gọi.
- pgid phải không âm.

Khi xảy ra lỗi, lệnh gọi sẽ trả về -1 và đặt errno thành một trong các mã lỗi sau:

TRUY CẬP

Tiến trình được xác định bởi pid là tiến trình con của tiến trình gọi đã gọi exec.

EINVAL

pgid nhỏ hơn 0.

EPERM

Quy trình được xác định bởi pid là một session leader hoặc đang ở trong một phiên khác với quy trình gọi. Ngoài ra, một nỗ lực đã được thực hiện để di chuyển một quy trình vào một nhóm quy trình bên trong một phiên khác.

ESRCH

pid không phải là quy trình hiện tại, 0 hoặc là con của quy trình hiện tại.

Giống như các phiên, việc lấy ID nhóm quy trình là có thể, mặc dù ít hữu ích hơn:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

pid_t getpgid (pid_t pid);
```

Một lệnh gọi đến getpgid( ) trả về ID nhóm quy trình của quy trình được xác định bởi pid. Nếu pid là 0, ID nhóm quy trình của quy trình hiện tại được sử dụng. Khi có lỗi, nó trả về -1 và đặt errno thành ESRCH, giá trị khả thi duy nhất, cho biết pid là một định danh quy trình không hợp lệ.

Tương tự như getsid( ), việc sử dụng chủ yếu nhằm mục đích chẩn đoán:

```
pid_t pgid;

pgid = getpgid (0);
nếu (pgid == -1)
        perror ("getpgid"); /* không thể thực hiện được */
khác
        printf ("Nhóm quy trình của tôi id=%d\n", pgid);
```

Các hàm nhóm quy trình lỗi thời Linux hỗ trợ hai giao diện cũ hơn từ BSD để thao tác hoặc lấy ID nhóm quy trình. Vì chúng ít hữu ích hơn các lệnh gọi hệ thống đã thảo luận trước đó, các chương trình mới chỉ nên sử dụng chúng khi tính di động được yêu cầu nghiêm ngặt. setpgrp() có thể được sử dụng để đặt ID nhóm quy trình:

```
#include <unistd.h>

int setpgrp (trống);
```

Lời kêu gọi này:

```
nếu (setpgrp ( ) == -1)
        lỗi ("setpgrp");
```

giống hệt với lệnh gọi sau: if (setpgid (0,0)

```
== -1) perror ("setpgid");
        Cả hai đều cố gắng
```

gán quy trình hiện tại cho nhóm quy trình có cùng số với pid của quy trình hiện tại, trả về 0 nếu thành công và -1 nếu thất bại. Tất cả các giá trị errno của setpgid() đều áp dụng cho setpgrp( ), ngoại trừ ERSCH.

Tương tự như vậy, có thể sử dụng lệnh gọi getpgrp() để lấy ID nhóm quy trình:

```
#include <unistd.h>

pid_t getpgrp (không có giá trị);
```

Lời kêu gọi này:

```
pid_t pgid = getpgrp ();
```

giống hệt với:

```
pid_t pgid = getpgid (0);
```

Cả hai đều trả về ID nhóm quy trình của quy trình gọi. Hàm getpgid() không thể lỗi.

Quỷ dữ

Một daemon là một tiến trình chạy ở chế độ nền, không kết nối với bất kỳ thiết bị đầu cuối điều khiển nào. Daemon thường được khởi động khi khởi động, được chạy dưới dạng root hoặc một số người dùng đặc biệt khác (như apache hoặc postfix) và xử lý các tác vụ cấp hệ thống. Theo quy ước, tên của một daemon thường kết thúc bằng d (như trong crond và sshd), nhưng điều này không bắt buộc hoặc thậm chí là phổ biến.

Tên này bắt nguồn từ con quỷ Maxwell, một thí nghiệm tư duy năm 1867 của nhà vật lý James Maxwell. Daemon cũng là những sinh vật siêu nhiên trong thần thoại Hy Lạp, tồn tại ở đâu đó giữa con người và các vị thần và được ban tặng sức mạnh và kiến thức thiêng liêng. Không giống như những con quỷ trong truyền thuyết Do Thái-Thiên chúa giáo, daemon Hy Lạp không nhất thiết phải xấu xa. Thật vậy, daemon trong thần thoại có xu hướng là trợ lý của các vị thần, thực hiện các nhiệm vụ mà cư dân của Núi Olympus thấy mình không muốn làm—giống như daemon Unix thực hiện các nhiệm vụ mà người dùng tiền cảnh muốn tránh.

Một daemon có hai yêu cầu chung: nó phải chạy như một chương trình con của init và không được kết nối với thiết bị đầu cuối.

Nói chung, một chương trình thực hiện các bước sau để trở thành một daemon:

1. Gọi fork( ). Thao tác này sẽ tạo ra một tiến trình mới, tiến trình này sẽ trở thành daemon.
2. Trong parent, gọi exit( ). Điều này đảm bảo parent gốc (grandparent của daemon) hài lòng rằng child của nó đã kết thúc, parent của daemon không còn chạy nữa và daemon không phải là process group leader. Điểm cuối cùng này là yêu cầu để hoàn thành thành công bước tiếp theo.
3. Gọi setsid( ), cung cấp cho daemon một nhóm quy trình và phiên mới, cả hai đều có nó làm người dẫn đầu. Điều này cũng đảm bảo rằng quy trình không có thiết bị đầu cuối điều khiển liên quan (vì quy trình vừa tạo một phiên mới và sẽ không chỉ định một phiên).
4. Thay đổi thư mục làm việc thành thư mục gốc thông qua chdir( ). Điều này được thực hiện vì thư mục làm việc được kế thừa có thể ở bất kỳ đâu trên hệ thống tệp. Dae-mon có xu hướng chạy trong suốt thời gian hoạt động của hệ thống và bạn không muốn giữ một số thư mục ngẫu nhiên mở, do đó ngăn cản quản trị viên hủy gắn kết hệ thống tệp chứa thư mục đó.

5. Đóng tất cả các mô tả tệp. Bạn không muốn kế thừa các mô tả tệp đang mở và không biết rằng chúng vẫn mở.
6. Mở các mô tả tệp 0, 1 và 2 (chuẩn vào, chuẩn ra và chuẩn lỗi) và chuyển hư ớng chúng tới /dev/null.

Thực hiện theo các quy tắc này, đây là một chư ơ ng trình tự biến mình thành quỷ dữ:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void) {

          pid_t pid;
          int i;

          /* tạo tiến trình mới */ pid =
          fork ( ); nếu (pid
          == -1) trả về -1;
                    nếu không
          thì nếu (pid != 0)
                    thoát (EXIT_SUCCESS);

          /* tạo phiên mới và nhóm quy trình */ if (setsid ( )
          == -1) return -1;


          /* đặt thư mục làm việc vào thư mục gốc */ if (chdir ("/") == -1)
          return -1;


          /* đóng tất cả các tệp đang mở--NR_OPEN là quá mức cần thiết, như ng vẫn
          hoạt động */ for (i = 0; i < NR_OPEN;
                    i++) close (i);

          /* chuyển hư ớng fd 0,1,2 đến /dev/null */ mở ("/dev/
          null", O_RDWR); /* stdin */ dup (0); /* stdout */ dup (0); /*
          stderror */


          /* thực hiện chức năng daemon của nó... */

          trả về 0;
}
```

Hầu hết các hệ thống Unix đều cung cấp hàm daemon() trong thư viện C để tự động hóa các bư ớc này, biến những thứ rư ờm rà thành đơ n giản:

```
#include <unistd.h>

int daemon (int nochdir, int noclose);
```

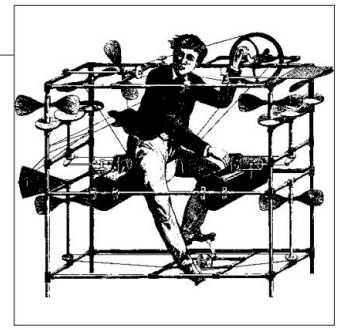
Nếu nochdir khác không, daemon sẽ không thay đổi thư mục làm việc của nó thành thư mục gốc. Nếu noclose khác không, daemon sẽ không đóng tất cả các mô tả tệp đang mở.
Các tùy chọn này hữu ích nếu tiến trình cha đã thiết lập các khía cạnh này của quy trình dae-monizing. Tuy nhiên, thông thường, người ta truyền 0 cho cả hai tham số này.

Nếu thành công, lệnh gọi trả về 0. Nếu thất bại, lệnh gọi trả về -1 và errno được đặt thành mã lỗi hợp lệ từ fork() hoặc setsid( ).

Phần kết luận

Chúng tôi đã đề cập đến những điều cơ bản của quản lý quy trình Unix, từ việc tạo quy trình đến việc chấm dứt quy trình, trong chương này. Trong chương tiếp theo, chúng tôi sẽ đề cập đến các giao diện quản lý quy trình nâng cao hơn, cũng như các giao diện để thay đổi hành vi lập lịch của các quy trình.

Quy trình nâng cao Sự quản lý



Chương 5 giới thiệu về sự trừu tượng của quy trình và thảo luận về các giao diện hạt nhân được sử dụng để tạo, kiểm soát và hủy quy trình. Chương này xây dựng dựa trên các ý tưởng đó, bắt đầu bằng thảo luận về trình lập lịch quy trình Linux và thuật toán lập lịch của nó, sau đó trình bày các giao diện quản lý quy trình nâng cao. Các lệnh gọi hệ thống này thao túng hành vi lập lịch và ngữ nghĩa của một quy trình, ảnh hưởng đến hành vi của trình lập lịch để theo đuổi mục tiêu do ứng dụng hoặc người dùng chỉ định.

Lên lịch quy trình

Bộ lập lịch quy trình là thành phần của hạt nhân chọn quy trình nào sẽ chạy tiếp theo. Nói cách khác, bộ lập lịch quy trình—hay đơn giản là bộ lập lịch—là hệ thống con của hạt nhân phân chia tài nguyên hữu hạn của thời gian xử lý giữa các quy trình của hệ thống. Khi quyết định quy trình nào có thể chạy và khi nào, bộ lập lịch có trách nhiệm tối đa hóa việc sử dụng bộ xử lý đồng thời tạo ấn tượng rằng nhiều quy trình đang thực thi đồng thời và liền mạch.

Trong chương này, chúng ta sẽ nói nhiều về các tiến trình có thể chạy . Một tiến trình có thể chạy là tiến trình trước hết không bị chặn. Các tiến trình tương tác với người dùng, đọc và ghi tệp nhiều hoặc phản hồi các sự kiện I/O hoặc mạng có xu hướng dành nhiều thời gian bị chặn trong khi chờ tài nguyên khả dụng và chúng không thể chạy trong những khoảng thời gian dài đó (dài, nghĩa là so với thời gian thực thi các lệnh máy). Một tiến trình có thể chạy cũng phải có ít nhất một phần trong khoảng thời gian của nó— khoảng thời gian mà trình lập lịch đã quyết định cho phép nó chạy—còn lại. Nhân đặt tất cả các tiến trình có thể chạy vào danh sách chạy. Khi một tiến trình đã sử dụng hết khoảng thời gian của mình, tiến trình đó sẽ bị xóa khỏi danh sách này và không được coi là có thể chạy lại cho đến khi tất cả các tiến trình có thể chạy khác cũng đã sử dụng hết khoảng thời gian của chúng.

Chỉ cần một tiến trình có thể chạy (hoặc không có tiến trình nào cả), công việc của một trình lập lịch tiến trình là không đáng kể. Tuy nhiên, một trình lập lịch chứng minh được giá trị của nó khi có nhiều tiến trình có thể chạy hơn bộ xử lý. Trong tình huống như vậy, một số tiến trình rõ ràng phải chạy trong khi những tiến trình khác phải chờ. Quyết định tiến trình nào sẽ chạy, khi nào và trong bao lâu là trách nhiệm cơ bản của trình lập lịch tiến trình.

Một hệ điều hành trên máy có một bộ xử lý là đa nhiệm nếu nó có thể xen kẽ việc thực thi của nhiều hơn một tiến trình, tạo ra ảo giác rằng có nhiều hơn một tiến trình đang chạy cùng một lúc. Trên máy có nhiều bộ xử lý, một hệ điều hành đa nhiệm cho phép các tiến trình thực sự chạy song song, trên các bộ xử lý khác nhau.

Hệ điều hành không đa nhiệm, chẳng hạn như DOS, chỉ có thể chạy một ứng dụng tại một thời điểm.

Hệ điều hành đa nhiệm có hai dạng: hợp tác và ưu tiên.
Linux triển khai dạng đa nhiệm sau, trong đó trình lập lịch quyết định khi nào một tiến trình dừng chạy và một tiến trình khác tiếp tục chạy. Chúng tôi gọi hành động tạm dừng một tiến trình đang chạy thay cho một lệnh chiếm quyền trước khác. Một lần nữa, khoảng thời gian một tiến trình chạy trước khi trình lập lịch chiếm quyền trước được gọi là timeslice của tiến trình (được gọi như vậy vì trình lập lịch phân bổ cho mỗi tiến trình có thể chạy một "lát" thời gian của bộ xử lý).

Ngược lại, trong đa nhiệm hợp tác, một tiến trình không dừng chạy cho đến khi nó tự nguyện quyết định làm như vậy. Chúng tôi gọi hành động của một tiến trình tự nguyện dừng lại là nhường. Lý tưởng nhất là các tiến trình thường nhường, nhưng hệ điều hành không thể thực thi hành vi này. Một chương trình thô lỗ hoặc bị hỏng có thể chạy trong thời gian dài hơn mức tối ưu hoặc thậm chí làm sập toàn bộ hệ thống. Do những thiếu sót của cách tiếp cận này, các hệ điều hành hiện đại hầu như luôn luôn được đa nhiệm trước; Linux cũng không ngoại lệ.

Bộ lập lịch quy trình O(1), được giới thiệu trong loạt kernel 2.5, là cốt lõi của lập lịch Linux.* Thuật toán lập lịch Linux cung cấp đa nhiệm ưu tiên cùng với hỗ trợ cho nhiều bộ xử lý, tính tương thích của bộ xử lý, cấu hình truy cập bộ nhớ không đồng nhất (NUMA), đa luồng, quy trình thời gian thực và các ưu tiên do người dùng cung cấp.

Ký hiệu Big-Oh O(1)—

đọc là “big oh of one”—là một ví dụ về ký hiệu big-oh, được sử dụng để biểu diễn độ phức tạp và khả năng mở rộng của thuật toán. Về mặt hình thức,

Nếu f x( ) là Ogx( )
( ) ,
thì c x, ' sao cho f x( ) c gx( ) xx > , ≤ '

Trong tiếng Anh, giá trị của một số thuật toán, f, luôn nhỏ hơn hoặc bằng giá trị của g nhân với một số hằng số tùy ý, miễn là đầu vào x lớn hơn một số giá trị ban đầu x'. Nghĩa là, g lớn bằng hoặc lớn hơn f; g giới hạn f từ trên.

* Đối với người đọc tò mò, trình lập lịch quy trình là độc lập và được định nghĩa trong kernel/sched.c trong kernel cây nguồn.

Do đó, O(1) ngụ ý rằng thuật toán đang xét có giá trị nhỏ hơ n một hằng số nào đó, c. Tất cả sự phô trư ơ ng và hoàn cảnh này đều quy về một lời hứa quan trọng: trình lập lịch quy trình Linux sẽ luôn thực hiện giống nhau, bất kể số lư ợng quy trình trên hệ thống. Điều này rất quan trọng vì hành động chọn một quy trình mới để chạy dư ờng như sẽ liên quan đến ít nhất một, nếu không muốn nói là nhiều lần lặp lại trên danh sách các quy trình. Với các trình lập lịch ngây thơ hơ n (bao gồm cả những trình lập lịch đư ợc sử dụng bởi các phiên bản Linux trư ớc đó), khi số lư ợng quy trình trên hệ thống tăng lên, các lần lặp lại như vậy nhanh chóng trở thành một nút thắt cổ chai tiềm ẩn. Trong trư ờng hợp tốt nhất, các vòng lặp như vậy sẽ đư a sự không chắc chắn—thiếu tính quyết định—vào quy trình lập lịch.

Bộ lập lịch Linux hoạt động theo thời gian không đổi bất kể yếu tố nào nên không gặp phải tình trạng tắc nghẽn như vậy.

Các lát cắt thời gian

Khoảng thời gian mà Linux phân bổ cho mỗi tiến trình là một biến quan trọng trong hành vi và hiệu suất chung của hệ thống. Nếu khoảng thời gian quá lớn, các tiến trình phải chờ một thời gian dài giữa các lần thực thi, giảm thiểu sự xuất hiện của việc thực thi đồng thời. Ngư ời dùng có thể cảm thấy bực bội vì độ trễ có thể nhận thấy. Ngư ợc lại, nếu khoảng thời gian quá nhỏ, một lư ợng thời gian đáng kể của hệ thống sẽ đư ợc dành để chuyển đổi từ ứng dụng này sang ứng dụng khác và các lợi ích như vị trí thời gian sẽ bị mất.

Do đó, việc xác định một khoảng thời gian lý tư ởng không hề dễ dàng. Một số hệ điều hành cung cấp cho các tiến trình những khoảng thời gian lớn, với hy vọng tối đa hóa thông lư ợng hệ thống và hiệu suất tổng thể. Các hệ điều hành khác cung cấp cho các tiến trình những khoảng thời gian rất nhỏ, với hy vọng cung cấp một hệ thống có hiệu suất tư ơ ng tác tuyệt vời. Như chúng ta sẽ thấy, Linux hư ớng đến mục tiêu tốt nhất của cả hai thế giới bằng cách phân bổ động các khoảng thời gian của tiến trình.

Lư u ý rằng một tiến trình không cần phải sử dụng hết toàn bộ timeslice của nó cùng một lúc. Một tiến trình đư ợc chỉ định timeslice 100 ms có thể chạy trong 20 ms, sau đó chặn một số tài nguyên, chẳng hạn như đầu vào bàn phím. Bộ lập lịch sẽ tạm thời xóa tiến trình này khỏi danh sách các tiến trình có thể chạy. Khi tài nguyên bị chặn trở nên khả dụng—trong trư ờng hợp này, khi bộ đệm bàn phím trở nên không rỗng—bộ lập lịch sẽ đánh thức tiến trình. Sau đó, tiến trình có thể tiếp tục chạy cho đến khi sử dụng hết 80 ms timeslice còn lại hoặc cho đến khi lại chặn một tài nguyên.

Các quy trình liên kết I/O so với quy trình liên kết với bộ xử lý

Các quy trình liên tục sử dụng hết tất cả các khoảng thời gian khả dụng của chúng đư ợc coi là bị ràng buộc bởi bộ xử lý. Các quy trình như vậy rất cần CPUtime và sẽ sử dụng hết tất cả những gì trình lập lịch cung cấp cho chúng. Ví dụ đơ n giản nhất là vòng lặp vô hạn. Các ví dụ khác bao gồm tính toán khoa học, tính toán toán học và xử lý hình ảnh.

Mặt khác, các tiến trình dành nhiều thời gian hơ n để chặn chờ một số tài nguyên hơ n là thực thi đư ợc coi là bị ràng buộc I/O. Các tiến trình bị ràng buộc I/O thư ờng phát hành và chờ tệp I/O, chặn khi nhập bằng bàn phím hoặc chờ ngư ời dùng

để di chuyển chuột. Ví dụ về các ứng dụng liên kết I/O bao gồm các tiện ích tệp thực hiện rất ít ngoại trừ việc phát lệnh gọi hệ thống yêu cầu hạt nhân thực hiện I/O, chẳng hạn như cp hoặc mv, và nhiều ứng dụng GUI dành nhiều thời gian chờ ngư ời dùng nhập dữ liệu.

Các ứng dụng bị ràng buộc bởi bộ xử lý và I/O khác nhau về loại hành vi của trình lập lịch có lợi nhất cho chúng. Các ứng dụng bị ràng buộc bởi bộ xử lý thèm muốn các khoảng thời gian lớn nhất có thể, cho phép chúng tối đa hóa tỷ lệ truy cập bộ đệm (thông qua vị trí thời gian) và hoàn thành công việc của chúng nhanh nhất có thể. Ngư ợc lại, các quy trình bị ràng buộc bởi I/O không nhất thiết cần các khoảng thời gian lớn, vì chúng thư ờng chỉ chạy trong khoảng thời gian rất ngắn trư ớc khi phát hành các yêu cầu I/O và chặn một số tài nguyên hạt nhân. Tuy nhiên, các quy trình bị ràng buộc bởi I/O đư ợc hư ởng lợi từ sự chú ý liên tục của trình lập lịch. Một ứng dụng như vậy có thể khởi động lại càng nhanh sau khi chặn và phân phối nhiều yêu cầu I/O hơ n thì ứng dụng đó có thể sử dụng phần cứng của hệ thống tốt hơ n. Hơ n nữa, nếu ứng dụng đang chờ đầu vào của ngư ời dùng, thì việc lên lịch càng nhanh thì nhận thức của ngư ời dùng về việc thực thi liền mạch sẽ càng cao.

Việc cân bằng nhu cầu của các tiến trình bị ràng buộc bởi bộ xử lý và I/O không phải là dễ dàng. Bộ lập lịch Linux cố gắng xác định và cung cấp chế độ ư u tiên cho các ứng dụng bị ràng buộc bởi I/O: các ứng dụng bị ràng buộc nhiều bởi I/O đư ợc tăng cư ờng ư u tiên, trong khi các ứng dụng bị ràng buộc nhiều bởi bộ xử lý đư ợc hư ởng chế độ ư u tiên phạt.

Trên thực tế, hầu hết các ứng dụng đều là sự kết hợp giữa I/O và bộ xử lý. Mã hóa/giải mã âm thanh/video là một ví dụ điển hình về loại ứng dụng không thể phân loại. Nhiều trò chơ i cũng khá hỗn hợp. Không phải lúc nào cũng có thể xác định đư ợc khuynh hư ớng của một ứng dụng nhất định và tại bất kỳ thời điểm nào, một quy trình nhất định có thể hoạt động theo cách này hay cách khác.

Lên lịch ư u tiên Khi một tiến

trình hết thời gian của nó, nhân sẽ tạm dừng tiến trình đó và bắt đầu chạy một tiến trình mới. Nếu không có tiến trình nào có thể chạy đư ợc trên hệ thống, nhân sẽ lấy tập hợp các tiến trình có thời gian hết, bổ sung thời gian của chúng và bắt đầu chạy lại chúng. Theo cách này, tất cả các tiến trình cuối cùng sẽ đư ợc chạy, ngay cả khi có các tiến trình có mức độ ư u tiên cao hơ n trên hệ thống—các tiến trình có mức độ ư u tiên thấp hơ n chỉ phải đợi các tiến trình có mức độ ư u tiên cao hơ n hết thời gian của chúng hoặc chặn chúng. Hành vi này hình thành nên một quy tắc quan trọng như ng ngầm định của việc lên lịch Unix: tất cả các tiến trình phải tiến triển.

Nếu không còn tiến trình nào có thể chạy đư ợc trên hệ thống, thì nhân sẽ “chạy” tiến trình nhàn rỗi. Tiến trình nhàn rỗi thực ra không phải là một tiến trình; nó cũng không thực sự chạy (khiến pin ở khắp mọi nơ i đều cạn kiệt). Thay vào đó, tiến trình nhàn rỗi là một thói quen đặc biệt mà nhân thực hiện để đơ n giản hóa thuật toán lập lịch và giúp việc tính toán dễ dàng.
Thời gian nhàn rỗi chỉ đơ n giản là thời gian dành cho việc chạy tiến trình nhàn rỗi.

Nếu một tiến trình đang chạy, và một tiến trình có mức độ ư u tiên cao hơ n trở thành có thể chạy (có lẽ vì nó đã bị chặn khi chờ nhập dữ liệu từ bàn phím, và ngư ời dùng vừa mới nhập một từ), tiến trình đang chạy hiện tại sẽ ngay lập tức bị tạm dừng, và hạt nhân chuyển sang tiến trình có mức độ ư u tiên cao hơ n. Do đó, không bao giờ có các tiến trình có thể chạy như ng không chạy với mức độ ư u tiên cao hơ n tiến trình đang chạy. Tiến trình đang chạy luôn là tiến trình có thể chạy có mức độ ư u tiên cao nhất trên hệ thống.

Luồng Luồng

là các đơ n vị thực thi trong một tiến trình duy nhất. Tất cả các tiến trình đều có ít nhất một luồng. Mỗi luồng có ảo hóa bộ xử lý riêng: tập hợp các thanh ghi, con trỏ lệnh và trạng thái bộ xử lý riêng. Trong khi hầu hết các tiến trình chỉ có một luồng, các tiến trình có thể có số lư ợng luồng lớn, tất cả đều thực hiện các tác vụ khác nhau, như ng chia sẻ cùng một không gian địa chỉ (và do đó cùng một bộ nhớ động, các tệp đư ợc ánh xạ, mã đối tư ợng, v.v.), danh sách các tệp đang mở và các tài nguyên hạt nhân khác.

Nhân Linux có một góc nhìn thú vị và độc đáo về luồng. Về cơ bản, nhân không có khái niệm như vậy. Đối với nhân Linux, tất cả các luồng đều là các tiến trình duy nhất. Ở cấp độ rộng, không có sự khác biệt giữa hai tiến trình không liên quan và hai luồng bên trong một tiến trình duy nhất. Nhân chỉ đơ n giản xem các luồng là các tiến trình chia sẻ tài nguyên. Nghĩa là, nhân coi một tiến trình bao gồm hai luồng là hai tiến trình riêng biệt chia sẻ một tập hợp các tài nguyên nhân (không gian địa chỉ, danh sách các tệp đang mở, v.v.).

Lập trình đa luồng là nghệ thuật lập trình với các luồng. API phổ biến nhất trên Linux để lập trình với các luồng là API đư ợc chuẩn hóa bởi IEEE Std 1003.1c-1995 (POSIX 1995 hoặc POSIX.1c). Các nhà phát triển thư ờng gọi thư viện triển khai API này là pthreads. Lập trình với các luồng là một chủ đề phức tạp và API pthreads thì lớn và phức tạp. Do đó, pthreads nằm ngoài phạm vi của cuốn sách này. Thay vào đó, cuốn sách này tập trung vào các giao diện mà thư viện pthreads đư ợc xây dựng trên đó.

Đư a bộ xử lý ra

Mặc dù Linux là hệ điều hành đa nhiệm ư u tiên, nó cũng cung cấp lệnh gọi hệ thống cho phép các tiến trình thực thi rõ ràng và hư ớng dẫn trình lập lịch chọn một tiến trình mới để thực thi:

```
#include <sched.h>

int sched_yield (trống);
```

Một lệnh gọi đến sched_yield( ) dẫn đến việc tạm dừng tiến trình đang chạy, sau đó trình lập lịch tiến trình sẽ chọn một tiến trình mới để chạy, theo cùng cách như thể chính hạt nhân đã chiếm quyền ư u tiên tiến trình đang chạy để thực thi một tiến trình mới. Lư u ý rằng nếu không có tiến trình nào khác có thể chạy đư ợc, thư ờng là như vậy,

quá trình tạo ra sẽ ngay lập tức tiếp tục thực hiện. Do sự không chắc chắn này, cùng với niềm tin chung rằng nói chung có những lựa chọn tốt hơn, việc sử dụng lệnh gọi hệ thống này không phổ biến.

Khi thành công, lệnh gọi trả về 0; khi thất bại, lệnh gọi trả về -1 và đặt errno thành mã lỗi thích hợp. Trên Linux—và nhiều khả năng là hầu hết các hệ thống Unix khác—sched_yield( ) không thể lỗi và do đó luôn trả về 0. Tuy nhiên, một lập trình viên kỹ lưỡng vẫn có thể kiểm tra giá trị trả về:

```
nếu (sched_yield ( )) lỗi
          ("sched_yield");
```

Sử dụng hợp pháp

Trong thực tế, có rất ít (nếu có) cách sử dụng hợp pháp sched_yield( ) trên một hệ thống đa nhiệm ưu tiên thích hợp như Linux. Nhân có đầy đủ khả năng đưa ra các quyết định lập lịch tối ưu và hiệu quả nhất—chắc chắn, nhân được trang bị tốt hơn một ứng dụng riêng lẻ để quyết định ưu tiên cái gì và khi nào. Đây chính xác là lý do tại sao các hệ điều hành từ bỏ đa nhiệm hợp tác để ủng hộ đa nhiệm ưu tiên.

Vậy thì tại sao chúng ta lại có lệnh gọi hệ thống "lên lịch lại cho tôi"? Câu trả lời nằm ở các ứng dụng phải chờ các sự kiện bên ngoài, có thể do người dùng, thành phần phần cứng hoặc quy trình khác gây ra. Ví dụ, nếu một quy trình cần chờ quy trình khác, "chỉ cần như ờng bộ xử lý cho đến khi quy trình kia hoàn tất" là giải pháp đầu tiên. Ví dụ, việc triển khai một người tiêu dùng ngây thơ trong cặp người tiêu dùng/nhà sản xuất có thể tương tự như sau:

```
/* người tiêu dùng... */
làm {
          trong khi (nhà sản xuất chưa sẵn sàng
                    ( )) sched_yield
          ( ); dữ liệu_quy_trình
( ); } trong khi (!time_to_quit ( ));
```

Rất may là các lập trình viên Unix không có xu hướng viết mã như thế này. Các chương trình Unix thường được điều khiển theo sự kiện và có xu hướng sử dụng một số loại cơ chế có thể chặn (như đường ống) giữa người tiêu dùng và nhà sản xuất, thay cho sched_yield( ). Trong trường hợp này, người tiêu dùng đọc từ đường ống, chặn khi cần thiết cho đến khi dữ liệu khả dụng. Đến lượt mình, nhà sản xuất ghi vào đường ống khi dữ liệu mới khả dụng. Điều này loại bỏ trách nhiệm phối hợp từ quy trình không gian người dùng, vốn chỉ là vòng lặp bận rộn, sang hạt nhân, có thể quản lý tình huống một cách tối ưu bằng cách đưa các quy trình vào chế độ ngủ và chỉ đánh thức chúng khi cần. Nhìn chung, các chương trình Unix nên hướng tới các giải pháp điều khiển theo sự kiện dựa trên các mô tả tệp có thể chặn.

Cho đến gần đây, một tình huống khó chịu đòi hỏi sched_yield( ): khóa luồng không gian người dùng. Khi một luồng cố gắng lấy khóa mà luồng khác đã giữ, luồng mới sẽ nhường bộ xử lý cho đến khi khóa khả dụng. Không có

hỗ trợ hạt nhân cho khóa không gian ngư ời dùng, cách tiếp cận này là đơ n giản nhất và hiệu quả nhất. Rất may, việc triển khai luồng Linux hiện đại (Thư viện luồng POSIX mới, hay NPTL) đã đư a ra giải pháp tối ư u bằng cách sử dụng futex, cung cấp hỗ trợ hạt nhân cho khóa không gian ngư ời dùng.

Một cách sử dụng khác cho sched_yield( ) là "chơ i đẹp": một chư ơ ng trình sử dụng nhiều bộ xử lý có thể gọi sched_yield( ) theo định kỳ, cố gắng giảm thiểu tác động của nó lên hệ thống. Mặc dù theo đuổi mục tiêu cao cả, chiến lư ợc này có hai sai sót. Đầu tiên, hạt nhân có thể đư a ra quyết định lập lịch toàn cục tốt hơ n nhiều so với một quy trình riêng lẻ và do đó, trách nhiệm đảm bảo hệ thống hoạt động trơ n tru phải thuộc về trình lập lịch quy trình, chứ không phải các quy trình. Để đạt đư ợc mục đích này, phần thư ởng tư ơ ng tác của trình lập lịch nhằm mục đích thư ởng cho các ứng dụng sử dụng nhiều I/O và trừng phạt các ứng dụng sử dụng nhiều bộ xử lý. Thứ hai, việc giảm thiểu chi phí chung của một ứng dụng sử dụng nhiều bộ xử lý so với các ứng dụng khác là trách nhiệm của ngư ời dùng, chứ không phải của các ứng dụng riêng lẻ. Ngư ời dùng có thể truyền đạt sở thích tư ơ ng đối của mình đối với hiệu suất ứng dụng thông qua lệnh nice shell, chúng ta sẽ thảo luận sau trong chư ơ ng này.

Yielding, Quá khứ và Hiện tại Trư ớc

khi giới thiệu nhân Linux 2.6, lệnh gọi sched_yield( ) chỉ có tác dụng nhỏ. Nếu có một quy trình chạy đư ợc khác, nhân sẽ chuyển sang quy trình đó và đặt quy trình gọi ở cuối danh sách các quy trình chạy đư ợc. Ngay sau đó, nhân sẽ lên lịch lại quy trình gọi. Trong trư ờng hợp không có quy trình chạy đư ợc nào khác khả dụng, quy trình gọi sẽ chỉ tiếp tục thực thi.

Kernel 2.6 đã thay đổi hành vi này. Thuật toán hiện tại như sau:

1. Quy trình này có phải là quy trình thời gian thực không? Nếu có, hãy dán nó vào cuối danh sách quy trình có thể chạy và trả về (đây là hành vi cũ). Nếu không, hãy tiếp tục bư ớc tiếp theo.
   (Để biết thêm về các quy trình thời gian thực, hãy xem “Hệ thống thời gian thực” ở phần sau của chư ơ ng này.)
2. Xóa hoàn toàn quy trình này khỏi danh sách các quy trình có thể chạy và đư a vào danh sách các quy trình đã hết hạn. Điều này ngụ ý rằng tất cả các quy trình có thể chạy phải thực thi và sử dụng hết các khoảng thời gian của chúng trư ớc khi quy trình gọi, cùng với các quy trình đã hết hạn khác, có thể tiếp tục thực thi.
3. Lên lịch thực thi tiến trình có thể chạy tiếp theo trong danh sách.

Do đó, hiệu ứng ròng của lệnh gọi sched_yield( ) giống như khi tiến trình đã sử dụng hết timeslice của nó. Hành vi này khác với các kernel trư ớc đó, trong đó hiệu ứng của sched_yield( ) nhẹ hơ n (tư ơ ng đư ơ ng với "nếu một tiến trình khác đã sẵn sàng và đang chờ, hãy chạy nó một lúc, như ng hãy quay lại ngay với tôi").

Một lý do cho sự thay đổi này là để ngăn chặn cái gọi là trư ờng hợp bệnh lý "ping-pong". Hãy tư ởng tư ợng hai tiến trình, A và B, cả hai đều gọi sched_yield( ). Giả sử đây là những tiến trình duy nhất có thể chạy đư ợc (có thể có những tiến trình khác có thể chạy đư ợc, như ng không có

với các lát cắt thời gian khác không). Với hành vi sched_yield() cũ , kết quả của tình huống này là hạt nhân lên lịch cho cả hai quy trình theo vòng quay, với mỗi quy trình nói trong quay lại, “Không, hãy lên lịch cho ngư ời khác!” Điều này vẫn tiếp diễn cho đến khi cả hai quy trình đều cạn kiệt timeslices. Nếu chúng ta vẽ sơ đồ các lựa chọn quy trình đư ợc thực hiện bởi trình lập lịch quy trình, nó sẽ giống như “A, B, A, B, A, B” v.v.—do đó “ping biệt danh “pong”.

Hành vi mới ngăn chặn trư ờng hợp này. Ngay khi quy trình A yêu cầu như ờng bộ xử lý, trình lập lịch sẽ xóa nó khỏi danh sách các quy trình có thể chạy. Tư ơng tự như vậy, ngay khi tiến trình B thực hiện cùng một yêu cầu, trình lập lịch xóa nó khỏi danh sách có thể chạy quy trình. Bộ lập lịch sẽ không xem xét việc chạy quy trình A hoặc quy trình B cho đến khi có không còn bất kỳ tiến trình chạy nào khác, ngăn chặn hiệu ứng ping-pong và cho phép các tiến trình khác nhận đư ợc phần thời gian xử lý công bằng.

Do đó, khi yêu cầu như ờng bộ xử lý, một tiến trình thực sự phải có ý định hãy như ờng nó lại!

Ư u tiên quy trình

Thảo luận trong phần này liên quan đến các quy trình bình thư ờng, không theo thời gian thực. Các quy trình theo thời gian thực yêu cầu các tiêu chí lập lịch khác nhau và hệ thống ư u tiên riêng biệt. Chúng ta sẽ thảo luận về điện toán thời gian thực sau trong chư ơng này.

Linux không lên lịch các tiến trình một cách tùy tiện. Thay vào đó, các ứng dụng đư ợc chỉ định mức độ ư u tiên ảnh hư ởng đến thời điểm các tiến trình của chúng chạy và trong bao lâu. Unix theo truyền thống gọi những ư u tiên này là những giá trị tốt đẹp, bởi vì ý tư ởng đằng sau chúng là “hãy tử tế” với các tiến trình khác trên hệ thống bằng cách hạ thấp mức độ ư u tiên của tiến trình, cho phép các tiến trình khác sử dụng nhiều thời gian xử lý của hệ thống hơ n.

Giá trị tốt đẹp quyết định thời điểm một tiến trình chạy. Linux lên lịch các tiến trình có thể chạy trong thứ tự ư u tiên từ cao đến thấp: một tiến trình có mức ư u tiên cao hơ n chạy trư ớc một tiến trình có mức ư u tiên thấp hơ n. Giá trị nice cũng quyết định kích thư ớc của lát cắt thời gian của một tiến trình.

Giá trị nice hợp lệ nằm trong khoảng từ –20 đến 19 bao gồm, với giá trị mặc định là 0. Có phần gây nhầm lẫn, giá trị nice của quy trình càng thấp thì mức độ ư u tiên của quy trình đó càng cao và thời gian của nó càng lớn; ngư ợc lại, giá trị càng cao thì mức độ ư u tiên của quy trình càng thấp, và lát cắt thời gian của nó càng nhỏ. Do đó, việc tăng giá trị tốt của một quy trình là “tốt” phần còn lại của hệ thống. Phép đảo số khá khó hiểu. Khi chúng ta nói quá trình có “mức độ ư u tiên cao” nghĩa là nó đư ợc chọn chạy nhanh hơ n và có thể chạy lâu hơ n các quy trình có mức độ ư u tiên thấp hơ n, như ng quy trình như vậy sẽ có mức độ ư u tiên thấp hơ n giá trị tốt.

Đẹp( )

Linux cung cấp một số lệnh gọi hệ thống để truy xuất và thiết lập giá trị tốt cho tiến trình. Đơ n giản nhất là nice( ):

```
#include <unistd.h>

int đẹp (int inc);
```

Một lệnh gọi thành công đến nice( ) sẽ tăng giá trị nice của một tiến trình lên inc và trả về giá trị mới đư ợc cập nhật. Chỉ một tiến trình có khả năng CAP_SYS_NICE (thực tế là các tiến trình do root sở hữu) mới có thể cung cấp giá trị âm cho inc, làm giảm giá trị nice của nó và do đó tăng mức ư u tiên của nó. Do đó, các tiến trình không phải root chỉ có thể giảm mức ư u tiên của chúng (bằng cách tăng giá trị nice của chúng).

Khi có lỗi, nice() trả về -1. Tuy nhiên, vì nice() trả về giá trị nice mới, -1 cũng là giá trị trả về thành công. Để phân biệt giữa thành công và thất bại, bạn có thể đư a errno về 0 trư ớc khi gọi và sau đó kiểm tra giá trị của nó. Ví dụ:

```
int ret;

errno = 0;
ret = nice (10); /* tăng nice của chúng ta lên 10 */ if (ret
== -1 && errno != 0) perror ("nice");

khác
          printf ("giá trị tốt nhất hiện tại là %d\n", ret);
```

Linux chỉ trả về một mã lỗi duy nhất: EPERM, biểu thị rằng quy trình gọi đã cố gắng tăng mức độ ư u tiên của nó (thông qua giá trị inc âm), như ng nó không sở hữu khả năng CAP_SYS_NICE . Các hệ thống khác cũng trả về EINVAL khi inc sẽ đặt giá trị nice ra khỏi phạm vi các giá trị hợp lệ, như ng Linux thì không. Thay vào đó, Linux âm thầm làm tròn các giá trị inc không hợp lệ lên hoặc xuống giá trị ở giới hạn của phạm vi cho phép, nếu cần.

Truyền 0 cho inc là một cách dễ dàng để có đư ợc giá trị nice hiện tại: printf

```
("giá trị nice hiện tại là %d\n", nice (0));
```

Thông thư ờng, một quy trình muốn thiết lập một giá trị tuyệt đối thay vì một giá trị gia tăng tư ơ ng đối. Bạn có thể thực hiện việc này bằng đoạn mã như sau:

```
int ret, val;

/* lấy giá trị nice hiện tại */ val
= nice (0);

/* chúng ta muốn giá trị đẹp là 10 */ val =
10 - val; errno =
0; ret = đẹp
(val); if (ret == -1
&& errno != 0) perror ("đẹp");

khác
          printf ("giá trị tốt nhất hiện tại là %d\n", ret);
```

getpriority() và setpriority()

Giải pháp thích hợp hơn là sử dụng các lệnh gọi hệ thống getpriority() và setpriority() , cho phép kiểm soát nhiều hơn nhưng hoạt động phức tạp hơn:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

Các lệnh gọi này hoạt động trên quy trình, nhóm quy trình hoặc người dùng, theo quy định của which và who. Giá trị của which phải là một trong PRIO_PROCESS, PRIO_PGRP hoặc PRIO_USER, trong trường hợp này who chỉ định ID quy trình, ID nhóm quy trình hoặc ID người dùng tương ứng. Nếu who là 0, lệnh gọi hoạt động trên ID quy trình hiện tại, ID nhóm quy trình hoặc ID người dùng tương ứng.

Một lệnh gọi đến getpriority() trả về mức ưu tiên cao nhất (giá trị nice số thấp nhất) của bất kỳ quy trình nào được chỉ định. Một lệnh gọi đến setpriority() đặt mức ưu tiên của tất cả các quy trình được chỉ định thành prio. Cũng giống như nice( ), chỉ một quy trình sở hữu CAP_SYS_NICE mới có thể tăng mức ưu tiên của quy trình (giảm giá trị nice số). Hơn nữa, chỉ một quy trình có khả năng này mới có thể tăng hoặc giảm mức ưu tiên của một quy trình không thuộc sở hữu của người gọi người dùng.

Giống như nice( ), getpriority() trả về -1 khi có lỗi. Vì đây cũng là giá trị trả về thành công, nên lập trình viên phải xóa errno trước khi gọi nếu họ muốn xử lý các điều kiện lỗi. Các lệnh gọi đến setpriority() không có vấn đề như vậy; setpriority() luôn trả về 0 khi thành công và -1 khi có lỗi.

Đoạn mã sau trả về mức độ ưu tiên của quy trình hiện tại:

```
int ret;

ret = getpriority (PRIO_PROCESS, 0);
printf ("giá trị tốt là %d\n", ret);
```

Đoạn mã sau đây đặt mức độ ưu tiên của tất cả các quy trình trong nhóm quy trình hiện tại thành 10:

```
int ret;

ret = setpriority (PRIO_PGRP, 0, 10);
nếu (ret ==
        -1) lỗi ("setpriority");
```

Khi xảy ra lỗi, cả hai hàm đều đặt errno thành một trong các giá trị sau:

EACCESS

Quá trình này đã cố gắng nâng mức ưu tiên của quá trình đã chỉ định nhưng không sở hữu CAP_SYS_NICE (chỉ setpriority()).

EINVAL

Giá trị được chỉ định không phải là một trong các giá trị PRIO_PROCESS, PRIO_PGRP hoặc PRIO_USER.

EPERM

ID ngư ời dùng có hiệu lực của quy trình khớp không khớp với ID ngư ời dùng có hiệu lực của quy trình đang chạy và quy trình đang chạy không sở hữu CAP_SYS_NICE (chỉ setpriority()).

ESRCH

Không tìm thấy quy trình nào khớp với tiêu chí do which và who cung cấp.

Ư u tiên I/O

Ngoài mức độ ư u tiên lập lịch, Linux cho phép các tiến trình chỉ định mức độ ư u tiên I/O. Giá trị này ảnh hư ởng đến mức độ ư u tiên tư ơ ng đối của các yêu cầu I/O của tiến trình. Bộ lập lịch I/O của hạt nhân (đư ợc thảo luận trong Chư ơ ng 4) phục vụ các yêu cầu bắt nguồn từ các tiến trình có mức độ ư u tiên I/O cao hơ n trư ớc các yêu cầu từ các tiến trình có mức độ ư u tiên I/O thấp hơ n.

Theo mặc định, trình lập lịch I/O sử dụng giá trị nice của quy trình để xác định mức độ ư u tiên I/O. Do đó, việc thiết lập giá trị nice sẽ tự động thay đổi mức ư u tiên I/O. Tuy nhiên, hạt nhân Linux cũng cung cấp thêm hai lệnh gọi hệ thống để thiết lập và truy xuất mức ư u tiên I/O một cách rõ ràng, độc lập với giá trị nice:

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

Thật không may, kernel vẫn chư a xuất các lệnh gọi hệ thống này và glibc không cung cấp bất kỳ quyền truy cập không gian ngư ời dùng nào. Nếu không có hỗ trợ glibc , việc sử dụng sẽ rất cồng kềnh. Hơ n nữa, khi và nếu hỗ trợ glibc đến, các giao diện có thể khác với các lệnh gọi hệ thống. Cho đến khi hỗ trợ như vậy khả dụng, có hai cách di động để thao tác mức ư u tiên I/O của một quy trình: thông qua giá trị nice hoặc một tiện ích như ionice, một phần của gói util-linux .* Không phải tất cả các trình lập

lịch I/O đều hỗ trợ mức ư u tiên I/O. Cụ thể, Trình lập lịch I/O Complete Fair Queu-ing (CFQ) hỗ trợ chúng; hiện tại, các trình lập lịch chuẩn khác thì không. Nếu trình lập lịch I/O hiện tại không hỗ trợ mức ư u tiên I/O, chúng sẽ bị bỏ qua một cách âm thầm.

Bộ xử lý Affinity

Linux hỗ trợ nhiều bộ xử lý trong một hệ thống duy nhất. Ngoài quy trình khởi động, phần lớn công việc hỗ trợ nhiều bộ xử lý nằm ở trình lập lịch quy trình. Trên máy đa xử lý đối xứng (SMP), trình lập lịch quy trình phải quyết định quy trình nào chạy trên mỗi CPU. Hai thách thức phát sinh từ trách nhiệm này: trình lập lịch phải hư ớng tới việc sử dụng đầy đủ tất cả các bộ xử lý của hệ thống, vì sẽ không hiệu quả khi một CPU ở trạng thái nhàn rỗi trong khi một quy trình đang chờ chạy.

* Gói util-linux nằm tại http://www.kernel.org/pub/linux/utils/util-linux. Nó đư ợc cấp phép theo Giấy phép Công cộng GNU v2.

Tuy nhiên, sau khi một tiến trình đã được lên lịch trên một CPU, trình lập lịch tiến trình nên hướng đến việc lên lịch trên cùng một CPU trong tương lai. Điều này có lợi vì việc di chuyển một tiến trình từ bộ xử lý này sang bộ xử lý khác có chi phí.

Chi phí lớn nhất trong số này liên quan đến hiệu ứng bộ nhớ đệm của quá trình di chuyển. Do thiết kế của các hệ thống SMP hiện đại, bộ nhớ đệm liên quan đến từng bộ xử lý là riêng biệt và khác biệt. Nghĩa là dữ liệu trong bộ nhớ đệm của bộ xử lý này không nằm trong bộ nhớ đệm của bộ xử lý khác. Do đó, nếu một tiến trình di chuyển đến một CPU mới và ghi dữ liệu mới vào bộ nhớ, dữ liệu trong bộ nhớ đệm của CPU cũ có thể trở nên cũ. Việc dựa vào bộ nhớ đệm đó hiện sẽ gây ra lỗi. Để ngăn chặn điều này, các bộ nhớ đệm sẽ vô hiệu hóa dữ liệu của nhau bất cứ khi nào chúng lưu trữ một khối bộ nhớ mới. Do đó, một phần dữ liệu nhất định chỉ nằm trong bộ nhớ đệm của một bộ xử lý tại bất kỳ thời điểm nào (giả sử dữ liệu được lưu trữ).
Khi một tiến trình di chuyển từ bộ xử lý này sang bộ xử lý khác, do đó có hai chi phí liên quan: dữ liệu được lưu trong bộ nhớ đệm không còn có thể truy cập được đối với tiến trình đã di chuyển và dữ liệu trong bộ nhớ đệm của bộ xử lý gốc phải bị vô hiệu hóa. Do những chi phí này, người lập lịch tiến trình cố gắng giữ một tiến trình trên một CPU cụ thể càng lâu càng tốt.

Tất nhiên, hai mục tiêu của trình lập lịch quy trình có khả năng xung đột. Nếu một bộ xử lý có tải quy trình lớn hơn đáng kể so với bộ xử lý khác—hoặc tệ hơn, nếu một bộ xử lý bận trong khi bộ xử lý khác nhàn rỗi—thì việc lập lịch lại một số quy trình trên CPU ít bận hơn là hợp lý. Quyết định thời điểm di chuyển các quy trình để phản hồi lại sự mất cân bằng như vậy, được gọi là cân bằng tải, có tầm quan trọng lớn đối với hiệu suất của máy SMP.

Processor affinity đề cập đến khả năng một quy trình được lên lịch nhất quán trên cùng một bộ xử lý. Thuật ngữ soft affinity đề cập đến khuynh hướng tự nhiên của trình lập lịch là tiếp tục lên lịch một quy trình trên cùng một bộ xử lý. Như chúng ta đã thảo luận, đây là một đặc điểm đáng giá. Trình lập lịch Linux cố gắng lên lịch các quy trình giống nhau trên cùng một bộ xử lý trong thời gian dài nhất có thể, chỉ di chuyển một quy trình từ CPU này sang CPU khác trong các tình huống mất cân bằng tải cực độ. Điều này cho phép trình lập lịch giảm thiểu các hiệu ứng bộ nhớ đệm của quá trình di chuyển, nhưng vẫn đảm bảo rằng tất cả các bộ xử lý trong một hệ thống được tải đều.

Tuy nhiên, đôi khi người dùng hoặc ứng dụng muốn thực thi liên kết giữa tiến trình với bộ xử lý. Điều này thường là do tiến trình rất nhạy cảm với bộ nhớ đệm và muốn duy trì trên cùng một bộ xử lý. Liên kết một tiến trình với một bộ xử lý cụ thể và để hạt nhân thực thi mối quan hệ này được gọi là thiết lập mối quan hệ cứng.

sched_getaffinity() và sched_setaffinity()

Các tiến trình thừa hưởng các đặc tính CPU của tiến trình cha mẹ và theo mặc định, các tiến trình có thể chạy trên bất kỳ CPU nào. Linux cung cấp hai lệnh gọi hệ thống để truy xuất và thiết lập đặc tính cứng của tiến trình:

```
#xác định _GNU_SOURCE

#include <sched.h>
```

```
định nghĩa cấu trúc cpu_set_t;

kích thước_t CPU_SETSIZE;

void CPU_SET (cpu dài không dấu, cpu_set_t *set); void
CPU_CLR (cpu dài không dấu, cpu_set_t *set); int CPU_ISSET
(cpu dài không dấu, cpu_set_t *set); void CPU_ZERO
(cpu_set_t *set);

int sched_setaffinity (pid_t pid, size_t setsize, const
                           cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize, cpu_set_t
                           *set);
```

Một lệnh gọi đến sched_getaffinity( ) sẽ lấy CPUaffinity của tiến trình pid và lưu trữ nó trong kiểu đặc biệt cpu_set_t , được truy cập thông qua các macro đặc biệt. Nếu pid là 0, lệnh gọi sẽ lấy affinity của tiến trình hiện tại. Tham số setsize là kích thước của kiểu cpu_set_t , có thể được glibc sử dụng để tương thích với các thay đổi trong tương lai về kích thước của kiểu này. Khi thành công, sched_getaffinity( ) trả về 0; khi thất bại, nó trả về -1 và errno được thiết lập. Sau đây là một ví dụ:

```
cpu_set_t đặt;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set); nếu (ret
== -1) lỗi
          ("sched_getaffinity");

đối với (i = 0; i < CPU_SETSIZE; i++)
          { int cpu;

          cpu = CPU_ISSET (i, &set);
          printf ("cpu=%i là %s\n", i,
                    cpu ? "set" : "unset");
}
```

Trước khi gọi, chúng ta sử dụng CPU_ZERO để "làm cho tất cả các bit trong tập hợp bằng 0. Sau đó, chúng ta lặp lại từ 0 đến CPU_SETSIZE trên tập hợp. Lưu ý rằng CPU_SETSIZE , thật khó hiểu, không phải là kích thước của tập hợp—bạn không bao giờ nên truyền nó cho setsize—mà là số lượng bộ xử lý có khả năng được biểu diễn bởi một tập hợp. Vì triển khai hiện tại biểu diễn mỗi bộ xử lý bằng một bit duy nhất, CPU_SETSIZE lớn hơn nhiều so với sizeof(cpu_set_t). Chúng ta sử dụng CPU_ISSET để kiểm tra xem một bộ xử lý nhất định trong hệ thống, i, có bị ràng buộc hay không bị ràng buộc với quy trình này. Nó trả về 0 nếu không bị ràng buộc và giá trị khác không nếu bị ràng buộc.

Chỉ có bộ xử lý vật lý trên hệ thống được thiết lập. Do đó, chạy đoạn mã này trên hệ thống có hai bộ xử lý sẽ cho kết quả:

```
cpu=0 được thiết lập
cpu=1 được thiết lập
```

```
cpu=2 chư a đư ợc thiết lập
cpu=3 chư a đư ợc thiết lập
...
cpu=1023 chư a đư ợc thiết lập
```

Như kết quả đầu ra cho thấy, CPU_SETSIZE (bắt đầu từ 0) hiện tại là 1.024.

Chúng tôi chỉ quan tâm đến CPU #0 và #1 vì chúng là bộ xử lý vật lý duy nhất trên hệ thống này. Có lẽ chúng tôi muốn đảm bảo rằng quy trình của chúng tôi chỉ chạy trên CPU #0 và không bao giờ chạy trên #1. Mã này thực hiện chính xác điều đó:

```
cpu_set_t đặt;
int ret, i;

CPU_ZERO (&set);            /* xóa tất cả CPU */ /* cho
CPU_SET (0, &set);          phép CPU #0 */ /* cấm
CPU_CLR (1, &set);          CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set); nếu
(ret == -1) lỗi
         ("sched_setaffinity");

đối với (i = 0; i < CPU_SETSIZE; i++)
         { int cpu;

         cpu = CPU_ISSET (i, &set);
         printf ("cpu=%i là %s\n", i,
                  cpu ? "set" : "unset");
}
```

Chúng ta bắt đầu, như thư ờng lệ, bằng cách xóa tập hợp bằng CPU_ZERO. Sau đó, chúng ta đặt CPU#0 bằng CPU_SET và bỏ đặt (xóa) CPU#1 bằng CPU_CLR. Hoạt động CPU_CLR là thừa vì chúng ta vừa xóa toàn bộ tập hợp, như ng nó đư ợc cung cấp để hoàn thiện.

Chạy chư ơ ng trình này trên cùng một hệ thống hai bộ xử lý sẽ cho kết quả đầu ra hơ i khác so với trư ớc:

```
cpu=0 đư ợc thiết lập
cpu=1 không đư ợc thiết
lập cpu=2 không đư ợc thiết lập
...
cpu=1023 chư a đư ợc
```

thiết lập Bây giờ, CPU #1 chư a đư ợc thiết lập. Quá trình này sẽ chỉ chạy trên CPU #0, bất kể thế nào!

Có thể có bốn giá trị errno :

MẶC ĐỊNH

Con trỏ đư ợc cung cấp nằm ngoài không gian địa chỉ của tiến trình hoặc không hợp lệ.

EINVAL

Trong trư ờng hợp này, không có bộ xử lý nào trên hệ thống đư ợc bật trong set (chỉ có sched_setaffinity()) hoặc setsize nhỏ hơ n kích thư ớc của cấu trúc dữ liệu bên trong của hạt nhân biểu diễn các bộ xử lý.

EPERM

Tiến trình liên kết với pid không thuộc quyền sở hữu của ID ngư ời dùng có hiệu lực hiện tại của tiến trình gọi và tiến trình này không sở hữu CAP_SYS_NICE.

CÂU CHUYỆN

Không tìm thấy tiến trình nào liên quan đến pid .

Hệ thống thời gian thực

Trong điện toán, thuật ngữ thời gian thực thư ờng là nguồn gốc của một số nhầm lẫn và hiểu lầm. Một hệ thống là "thời gian thực" nếu nó phải tuân theo các thời hạn hoạt động: thời gian tối thiểu và bắt buộc giữa các kích thích và phản ứng. Một hệ thống thời gian thực quen thuộc là hệ thống chống bó cứng phanh (ABS) đư ợc tìm thấy trên hầu hết các ô tô hiện đại. Trong hệ thống này, khi phanh đư ợc nhấn, máy tính sẽ điều chỉnh áp suất phanh, thư ờng áp dụng và giải phóng áp suất phanh tối đa nhiều lần trong một giây. Điều này ngăn bánh xe "bị bó cứng", có thể làm giảm hiệu suất dừng hoặc thậm chí khiến xe trư ợt không kiểm soát đư ợc. Trong một hệ thống như vậy, thời hạn hoạt động là hệ thống phải phản ứng nhanh như thế nào với tình trạng bánh xe "bị bó cứng" và hệ thống có thể áp dụng áp suất phanh nhanh như thế nào.

Hầu hết các hệ điều hành hiện đại, bao gồm cả Linux, đều cung cấp một số mức độ hỗ trợ thời gian thực.

Hệ thống thời gian thực cứng so với mềm Hệ thống

thời gian thực có hai loại: cứng và mềm. Hệ thống thời gian thực cứng yêu cầu tuân thủ tuyệt đối các thời hạn hoạt động. Vư ợt quá thời hạn sẽ cấu thành thất bại và là một lỗi lớn. Mặt khác, hệ thống thời gian thực mềm không coi việc vư ợt quá thời hạn là một lỗi nghiêm trọng.

Các ứng dụng thời gian thực cứng dễ nhận biết: một số ví dụ là hệ thống chống bó cứng phanh, hệ thống vũ khí quân sự, thiết bị y tế và xử lý tín hiệu. Các ứng dụng thời gian thực mềm không phải lúc nào cũng dễ nhận biết. Một thành viên rõ ràng của nhóm đó là các ứng dụng xử lý video: ngư ời dùng nhận thấy chất lư ợng giảm nếu họ bỏ lỡ thời hạn, như ng một vài khung hình bị mất có thể chấp nhận đư ợc.

Nhiều ứng dụng khác có những hạn chế về thời gian, nếu không đáp ứng đư ợc, sẽ gây bất lợi cho trải nghiệm của ngư ời dùng. Các ứng dụng đa phư ơ ng tiện, trò chơ i và chư ơ ng trình mạng là những ví dụ. Tuy nhiên, trình soạn thảo văn bản thì sao? Nếu chư ơ ng trình không thể phản hồi đủ nhanh với các lần nhấn phím, trải nghiệm sẽ kém và ngư ời dùng có thể tức giận hoặc thất vọng. Đây có phải là ứng dụng thời gian thực mềm không? Chắc chắn, khi các nhà phát triển viết ứng dụng, họ nhận ra rằng họ cần phản hồi các lần nhấn phím một cách kịp thời. Như ng điều này có đư ợc tính là thời hạn hoạt động không? Dòng định nghĩa ứng dụng thời gian thực mềm không hề rõ ràng.

Trái với niềm tin phổ biến, một hệ thống thời gian thực không nhất thiết phải nhanh. Thật vậy, với phần cứng tương đương, một hệ thống thời gian thực có thể chậm hơn một hệ thống không thời gian thực—nếu không có lý do gì khác, thì đó là sự gia tăng chi phí chung cần thiết để hỗ trợ các quy trình thời gian thực. Tương tự như vậy, sự phân chia giữa các hệ thống thời gian thực cứng và mềm không phụ thuộc vào quy mô của thời hạn hoạt động. Một lò phản ứng hạt nhân sẽ quá nhiệt nếu hệ thống SCRAM không hạ thanh điều khiển trong vòng vài giây sau khi phát hiện ra thông lượng neutron quá mức. Đây là một hệ thống thời gian thực cứng với thời hạn hoạt động dài (đối với máy tính). Ngược lại, trình phát video có thể bỏ qua một khung hình hoặc làm gián đoạn âm thanh nếu ứng dụng không thể nạp lại bộ đệm phát lại trong vòng 100 ms. Đây là một hệ thống thời gian thực mềm với thời hạn hoạt động khắt khe.

Độ trễ, Độ dao động và Thời hạn Độ trễ đề

cập đến khoảng thời gian từ khi kích thích xảy ra cho đến khi thực hiện phản hồi. Nếu độ trễ nhỏ hơn hoặc bằng thời hạn hoạt động, hệ thống đang hoạt động chính xác. Trong nhiều hệ thống thời gian thực cứng, thời hạn hoạt động và độ trễ bằng nhau—hệ thống xử lý các kích thích theo các khoảng thời gian cố định, tại các thời điểm chính xác. Trong các hệ thống thời gian thực mềm, phản hồi cần thiết ít chính xác hơn và độ trễ thể hiện một số lượng phương sai—mục đích chỉ đơn giản là để phản hồi xảy ra trong thời hạn.

Thường rất khó để đo độ trễ, vì phép tính này đòi hỏi phải biết thời điểm kích thích xảy ra. Tuy nhiên, khả năng đóng dấu thời gian kích thích thường đòi hỏi khả năng phản ứng với kích thích đó. Do đó, nhiều nỗ lực đo lường độ trễ không làm được điều đó; thay vào đó, họ đo lường sự thay đổi về thời gian giữa các phản ứng.
Sự thay đổi về thời gian giữa các sự kiện liên tiếp là độ trễ chứ không phải độ trễ.

Ví dụ, hãy xem xét một kích thích xảy ra sau mỗi 10 mili giây. Để đo hiệu suất của hệ thống, chúng ta có thể đóng dấu thời gian cho các phản hồi của mình để đảm bảo rằng chúng xảy ra sau mỗi 10 mili giây. Tuy nhiên, độ lệch so với mục tiêu này không phải là độ trễ mà là độ rung. Những gì chúng ta đang đo là sự thay đổi trong các phản hồi liên tiếp. Nếu không biết khi nào kích thích xảy ra, chúng ta không biết sự khác biệt thực tế về thời gian giữa kích thích và phản hồi. Ngay cả khi biết rằng kích thích xảy ra sau mỗi 10 ms, chúng ta cũng không biết khi nào lần đầu tiên xảy ra. Có lẽ đáng ngạc nhiên là nhiều nỗ lực đo độ trễ mắc phải lỗi này và báo cáo độ rung chứ không phải độ trễ. Chắc chắn, độ rung là một số liệu hữu ích và công cụ đo lường như vậy có lẽ khá hữu ích. Tuy nhiên, chúng ta phải gọi một con vịt là một con vịt!

Các hệ thống thời gian thực cứng thường có độ trễ rất thấp vì chúng phản ứng với các kích thích sau—không phải trong—một khoảng thời gian chính xác. Các hệ thống như vậy hướng đến độ trễ bằng không và độ trễ bằng với độ trễ hoạt động. Nếu độ trễ vượt quá độ trễ, hệ thống sẽ hỏng.

Hệ thống thời gian thực mềm dễ bị rung hơn. Trong các hệ thống này, thời gian phản hồi lý tưởng nằm trong độ trễ hoạt động—thường sớm hơn nhiều, đôi khi không. Do đó, rung thường là một đại diện tuyệt vời cho độ trễ như một số liệu hiệu suất.

Hỗ trợ thời gian thực của Linux Linux

cung cấp cho các ứng dụng hỗ trợ thời gian thực mềm thông qua một nhóm lệnh gọi hệ thống được định nghĩa bởi IEEE Std 1003.1b-1993 (thường được viết tắt là POSIX 1993 hoặc POSIX.1b).

Về mặt kỹ thuật, tiêu chuẩn POSIX không chỉ định hỗ trợ thời gian thực được cung cấp là mềm hay cứng. Trên thực tế, tất cả những gì tiêu chuẩn POSIX thực sự làm là mô tả một số chính sách lập lịch tôn trọng các ưu tiên. Các loại ràng buộc thời gian nào mà hệ điều hành áp dụng cho các chính sách này tùy thuộc vào các nhà thiết kế hệ điều hành.

Trong những năm qua, nhân Linux đã đạt được sự hỗ trợ thời gian thực ngày càng tốt hơn, cung cấp độ trễ ngày càng thấp hơn và độ trễ ổn định hơn mà không ảnh hưởng đến hiệu suất hệ thống. Phần lớn là do việc cải thiện độ trễ giúp ích cho nhiều lớp ứng dụng, chẳng hạn như các quy trình liên kết I/O và máy tính để bàn, chứ không chỉ các ứng dụng thời gian thực. Những cải tiến này cũng là nhờ vào sự thành công của Linux trong các hệ thống nhúng và thời gian thực.

Thật không may, nhiều sửa đổi nhúng và thời gian thực đã được thực hiện đối với hạt nhân Linux chỉ tồn tại trong các giải pháp Linux tùy chỉnh, bên ngoài hạt nhân chính thức chính thống. Một số sửa đổi này cung cấp khả năng giảm độ trễ hơn nữa và thậm chí là hành vi thời gian thực cứng. Các phần sau chỉ thảo luận về giao diện hạt nhân chính thức và hành vi của hạt nhân chính thống. May mắn thay, hầu hết các sửa đổi thời gian thực vẫn tiếp tục sử dụng giao diện POSIX. Do đó, thảo luận tiếp theo cũng có liên quan đến các hệ thống đã sửa đổi.

Chính sách và ưu tiên lập lịch Linux Hành vi của trình

lập lịch Linux đối với một quy trình phụ thuộc vào chính sách lập lịch của quy trình, còn được gọi là lớp lập lịch. Ngoài chính sách mặc định thông thường, Linux cung cấp hai chính sách lập lịch thời gian thực. Một macro tiền xử lý từ tiêu đề <sched.h> biểu diễn từng chính sách: các macro là SCHED_FIFO, SCHED_RR và SCHED_OTHER.

Mỗi tiến trình đều có một mức ưu tiên tĩnh, không liên quan đến giá trị nice. Đối với các ứng dụng thông thường, mức ưu tiên này luôn là 0. Đối với các tiến trình thời gian thực, mức ưu tiên này nằm trong khoảng từ 1 đến 99, bao gồm cả hai giá trị. Bộ lập lịch Linux luôn chọn tiến trình có mức ưu tiên cao nhất để chạy (tức là tiến trình có giá trị ưu tiên tĩnh lớn nhất). Nếu một tiến trình đang chạy với mức ưu tiên tĩnh là 50 và một tiến trình có mức ưu tiên là 51 trở thành có thể chạy, bộ lập lịch sẽ ngay lập tức chiếm quyền trước tiến trình đang chạy và chuyển sang tiến trình mới có thể chạy. Ngược lại, nếu một tiến trình đang chạy với mức ưu tiên là 50 và một tiến trình có mức ưu tiên là 49 trở thành có thể chạy, bộ lập lịch sẽ không chạy tiến trình đó cho đến khi

khối tiến trình priority-50, trở nên không thể chạy được. Vì các tiến trình bình thường có mức ưu tiên là 0, nên bất kỳ tiến trình thời gian thực nào có thể chạy được sẽ luôn chiếm quyền trước một tiến trình bình thường và chạy.

Chính sách vào trước ra trước

Chính sách vào trước ra trước (FIFO) là chính sách thời gian thực rất đơn giản, không có khoảng thời gian. Một quy trình được phân loại theo FIFO sẽ tiếp tục chạy miễn là không có quy trình nào có mức độ ưu tiên cao hơn có thể chạy được. Lớp FIFO được biểu diễn bằng macro SCHED_FIFO.

Vì chính sách này không có khoảng thời gian cụ thể nên các quy tắc hoạt động của nó khá đơn giản:

- Một quy trình được phân loại FIFO có thể chạy sẽ luôn chạy nếu đó là quy trình có mức độ ưu tiên cao nhất trên hệ thống. Đặc biệt, khi một quy trình được phân loại FIFO có thể chạy, nó sẽ ngay lập tức chiếm quyền ưu tiên của một quy trình bình thường.
- Một quy trình được phân loại theo FIFO sẽ tiếp tục chạy cho đến khi nó chặn hoặc gọi sched_yield( ) hoặc cho đến khi một tiến trình có mức độ ưu tiên cao hơn có thể chạy được.
- Khi một quy trình được phân loại theo FIFO bị chặn, trình lập lịch sẽ xóa quy trình đó khỏi danh sách các quy trình có thể chạy. Khi quy trình đó có thể chạy trở lại, quy trình đó sẽ được chèn vào cuối danh sách các quy trình có mức độ ưu tiên của quy trình đó. Do đó, quy trình đó sẽ không chạy cho đến khi bất kỳ quy trình nào khác có mức độ ưu tiên cao hơn hoặc bằng nhau ngừng thực thi.
- Khi một tiến trình được phân loại theo FIFO gọi sched_yield( ), trình lập lịch sẽ di chuyển tiến trình đó đến cuối danh sách các tiến trình có mức độ ưu tiên của nó. Do đó, tiến trình đó sẽ không chạy cho đến khi bất kỳ tiến trình nào khác có mức độ ưu tiên bằng nhau ngừng thực thi. Nếu tiến trình gọi là tiến trình duy nhất có mức độ ưu tiên của nó, sched_yield( ) sẽ không có hiệu lực. • Khi một tiến trình có mức độ ưu tiên cao hơn chiếm quyền ưu tiên của một tiến trình có mức độ ưu tiên FIFO, tiến trình có mức độ ưu tiên FIFO vẫn ở cùng một vị trí trong danh sách các tiến trình có mức độ ưu tiên đã cho. Do đó, sau khi tiến trình có mức độ ưu tiên cao hơn ngừng thực thi, tiến trình có mức độ ưu tiên FIFO đã chiếm quyền ưu tiên sẽ tiếp tục thực thi.
- Khi một tiến trình tham gia lớp FIFO hoặc khi mức độ ưu tiên tĩnh của tiến trình thay đổi, tiến trình đó sẽ được đưa lên đầu danh sách các tiến trình theo mức độ ưu tiên đã cho. Do đó, một tiến trình được phân loại theo FIFO mới được ưu tiên có thể chiếm quyền trước một tiến trình đang thực thi có cùng mức độ ưu tiên.

Về cơ bản, chúng ta có thể nói rằng các quy trình được phân loại theo FIFO luôn chạy trong thời gian chúng muốn, miễn là chúng là các quy trình có mức độ ưu tiên cao nhất trên hệ thống. Các quy tắc thú vị liên quan đến những gì xảy ra giữa các quy trình được phân loại theo FIFO có cùng mức độ ưu tiên.

Chính sách luân phiên Lớp luân

phiên (RR) giống hệt với lớp FIFO, ngoại trừ việc nó áp đặt các quy tắc bổ sung trong trường hợp các quy trình có cùng mức độ ưu tiên. Macro SCHED_RR biểu diễn lớp này.

Bộ lập lịch gán cho mỗi tiến trình đư ợc phân loại theo RR một khoảng thời gian. Khi một tiến trình đư ợc phân loại theo RR hết khoảng thời gian của nó, bộ lập lịch sẽ di chuyển nó đến cuối danh sách các tiến trình có mức độ ư u tiên của nó. Theo cách này, các tiến trình đư ợc phân loại theo RR có mức độ ư u tiên nhất định đư ợc lên lịch luân phiên giữa chúng. Nếu chỉ có một tiến trình có mức độ ư u tiên nhất định, thì lớp RR giống hệt với lớp FIFO. Trong trư ờng hợp như vậy, khi khoảng thời gian của nó hết hạn, thì tiến trình chỉ cần tiếp tục thực thi.

Chúng ta có thể coi một quy trình đư ợc phân loại theo RR giống hệt với một quy trình đư ợc phân loại theo FIFO, ngoại trừ việc nó cũng ngừng thực thi khi sử dụng hết khoảng thời gian của mình, khi đó nó sẽ di chuyển đến cuối danh sách các quy trình có thể chạy theo mức độ ư u tiên của nó.

Quyết định sử dụng SCHED_FIFO hay SCHED_RR hoàn toàn là vấn đề về hành vi ư u tiên nội bộ. Các lát cắt thời gian của lớp RR chỉ có liên quan giữa các quy trình có cùng mức ư u tiên. Các quy trình đư ợc phân loại theo FIFO sẽ tiếp tục chạy mà không bị hạn chế; các quy trình đư ợc phân loại theo RR sẽ lên lịch giữa chúng theo một mức ư u tiên nhất định. Trong cả hai trư ờng hợp, một quy trình có mức ư u tiên thấp hơ n sẽ không bao giờ chạy nếu có một quy trình có mức ư u tiên cao hơ n.

Chính sách bình

thư ờng SCHED_OTHER biểu thị chính sách lập lịch chuẩn, lớp mặc định không theo thời gian thực. Tất cả các quy trình đư ợc phân loại bình thư ờng đều có mức ư u tiên tĩnh là 0. Do đó, bất kỳ quy trình đư ợc phân loại FIFO hoặc RR nào có thể chạy đư ợc sẽ chiếm quyền trư ớc một quy trình đư ợc phân loại bình thư ờng đang chạy.

Bộ lập lịch sử dụng giá trị nice, đã thảo luận trư ớc đó, để ư u tiên các tiến trình trong lớp bình thư ờng. Giá trị nice không ảnh hư ởng đến mức ư u tiên tĩnh, vẫn là 0.

Chính sách lập lịch theo lô

SCHED_BATCH là chính sách lập lịch theo lô hoặc nhàn rỗi. Hành vi của nó có phần trái ngư ợc với các chính sách thời gian thực: các quy trình trong lớp này chỉ chạy khi không có quy trình nào khác có thể chạy trên hệ thống, ngay cả khi các quy trình khác đã sử dụng hết các khoảng thời gian của chúng. Điều này khác với hành vi của các quy trình có giá trị nice lớn nhất (tức là các quy trình có mức độ ư u tiên thấp nhất) ở chỗ cuối cùng các quy trình như vậy sẽ chạy, vì các quy trình có mức độ ư u tiên cao hơ n sử dụng hết các khoảng thời gian của chúng.

Thiết lập chính sách lập lịch Linux

Các quy trình có thể thao tác chính sách lập lịch Linux thông qua sched_getscheduler() và sched_setscheduler( ):

```
#include <sched.h>

cấu trúc tham số lịch
          trình { /
          * ... */ int ư u tiên
          lịch trình; /* ... */
};

int sched_getscheduler(pid_t pid);
```

```
int sched_setscheduler (pid_t pid, int chính
                                    sách, const
                                    struct sched_param *sp);
```

Một lệnh gọi thành công đến sched_getscheduler( ) trả về chính sách lập lịch của tiến trình đư ợc biểu diễn bởi pid. Nếu pid là 0, lệnh gọi trả về chính sách lập lịch của tiến trình gọi. Một số nguyên đư ợc định nghĩa trong <sched.h> biểu diễn chính sách lập lịch: chính sách vào trư ớc, ra trư ớc là SCHED_FIFO; chính sách luân phiên là SCHED_RR; và chính sách bình thư ờng là SCHED_OTHER. Khi có lỗi, lệnh gọi trả về -1 (không bao giờ là chính sách lập lịch hợp lệ) và errno đư ợc đặt thành giá trị phù hợp.

Cách sử dụng rất đơ n giản:

```
chính sách int;

/* lấy chính sách lập lịch của chúng tôi
*/ policy = sched_getscheduler (0);

chuyển đổi (chính
sách) { trư ờng hợp
          SCHED_OTHER: printf ("Chính sách là
          bình
thư ờng\n"); ngắt;
          trư ờng hợp SCHED_RR: printf ("Chính sách là
          vòng
tròn\n"); ngắt;
          trư ờng hợp SCHED_FIFO: printf ("Chính sách là vào
          trư ớc, ra trư ớc\n"); ngắt;
trư ờng hợp -1:
          perror ("sched_getscheduler"); ngắt; mặc
          định:

          fprintf (stderr, "Chính sách không xác định!\n");
}
```

Một lệnh gọi đến sched_setscheduler( ) sẽ thiết lập chính sách lập lịch của tiến trình đư ợc biểu diễn bởi pid thành policy. Bất kỳ tham số nào liên quan đến policy đều đư ợc thiết lập thông qua sp. Nếu pid là 0, chính sách và tham số của tiến trình gọi sẽ đư ợc thiết lập. Khi thành công, lệnh gọi sẽ trả về 0.
Khi thất bại, lệnh gọi trả về -1 và errno đư ợc thiết lập ở mức phù hợp.

Các trư ờng hợp lệ bên trong cấu trúc sched_param phụ thuộc vào các chính sách lập lịch đư ợc hệ điều hành hỗ trợ. Các chính sách SCHED_RR và SCHED_FIFO yêu cầu một trư ờng, sched_priority, biểu diễn mức độ ư u tiên tĩnh. SCHED_OTHER không sử dụng bất kỳ trư ờng nào, trong khi các chính sách lập lịch đư ợc hỗ trợ trong tư ơ ng lai có thể sử dụng các trư ờng mới.
Do đó, các chư ơ ng trình di động và hợp pháp không đư ợc đư a ra giả định về bố cục của cấu trúc.

Việc thiết lập chính sách lập lịch và các thông số của quy trình rất dễ dàng:

```
cấu trúc sched_param sp = { .sched_priority = 1 }; int ret;


ret = lịch_lịch_đặt_lịch_trình (0, SCHED_RR, &sp);
```

```
    nếu (ret == -1)
            { perror ("sched_setscheduler");
            trả về 1;
    }
```

Đoạn mã này đặt chính sách lập lịch của quy trình gọi thành vòng tròn với mức ưu tiên tĩnh là 1. Chúng tôi cho rằng 1 là mức ưu tiên hợp lệ—về mặt kỹ thuật, không nhất thiết phải như vậy. Chúng tôi sẽ thảo luận về cách tìm phạm vi ưu tiên hợp lệ cho một chính sách nhất định trong phần sắp tới.

Thiết lập chính sách lập lịch khác với SCHED_OTHER yêu cầu khả năng CAP_SYS_NICE . Do đó, người dùng root thường chạy các quy trình thời gian thực. Kể từ kernel 2.6.12, giới hạn tài nguyên RLIMIT_RTPRIO cho phép người dùng không phải root thiết lập chính sách thời gian thực lên đến một mức trần ưu tiên nhất định.

Mã lỗi. Khi xảy ra lỗi, có thể có bốn giá trị errno :

MẶC ĐỊNH

Con trỏ sp trỏ tới một vùng bộ nhớ không hợp lệ hoặc không thể truy cập.

EINVAL

Chính sách lập lịch được biểu thị bằng policy là không hợp lệ hoặc giá trị được đặt trong sp không có ý nghĩa đối với chính sách đã cho (chỉ sched_setscheduler()).

EPERM

Quá trình gọi không có đủ khả năng cần thiết.

ESRCH

Giá trị pid không biểu thị một quy trình đang chạy.

Thiết lập tham số lập lịch

Các giao diện sched_getparam() và sched_setparam() do POSIX định nghĩa sẽ truy xuất và thiết lập các tham số liên quan đến chính sách lập lịch đã được thiết lập:

```
#include <sched.h>

cấu trúc tham số lịch
        trình { /
        * ... */ int ưu tiên
        lịch trình; /* ... */
};

int sched_getparam (pid_t pid, struct sched_param *sp);

int sched_setparam (pid_t pid, const struct sched_param *sp);
```

Giao diện sched_getscheduler() chỉ trả về chính sách lập lịch, không trả về bất kỳ tham số liên quan nào. Một lệnh gọi đến sched_getparam() trả về thông qua sp các tham số lập lịch liên quan đến pid:

```
cấu trúc sched_param sp;
int ret;

ret = sched_getparam (0, &sp); nếu
(ret == -1) { lỗi
         ("sched_getparam"); trả về 1;

}

printf ("Ưu tiên của chúng tôi là %d\n", sp.sched_priority);
```

Nếu pid là 0, lệnh gọi sẽ trả về các tham số của tiến trình gọi. Khi thành công, lệnh gọi sẽ trả về 0. Khi thất bại, lệnh gọi sẽ trả về -1 và đặt errno theo giá trị phù hợp.

Vì sched_setscheduler( ) cũng thiết lập bất kỳ tham số lập lịch liên quan nào, nên sched_setparam( ) chỉ hữu ích để sửa đổi các tham số sau này:

```
cấu trúc sched_param sp;
int ret;

sp.sched_priority = 1;
ret = sched_setparam (0, &sp); nếu
(ret == -1) { lỗi
         ("sched_setparam"); trả về 1;

}
```

Nếu thành công, các tham số lập lịch của pid được thiết lập theo sp và cuộc gọi trả về 0. Nếu thất bại, cuộc gọi trả về -1 và errno được thiết lập ở mức phù hợp.

Nếu chúng ta chạy hai đoạn mã trước theo thứ tự, chúng ta sẽ thấy kết quả sau:

```
Ưu tiên của chúng tôi là 1
```

Ví dụ này một lần nữa giả định rằng 1 là mức ưu tiên hợp lệ. Đúng vậy, nhưng các ứng dụng di động phải đảm bảo. Chúng ta sẽ xem xét cách kiểm tra phạm vi mức ưu tiên hợp lệ ngay sau đây.

Mã lỗi

Khi xảy ra lỗi, có thể có bốn giá trị errno :

EFAULT

Con trỏ sp trỏ tới một vùng bộ nhớ không hợp lệ hoặc không thể truy cập.

EINVAL

Giá trị được đặt trong sp không có ý nghĩa đối với chính sách đã cho (chỉ sched_getparam()).

EPERM

Quá trình gọi không có đủ khả năng cần thiết.

CÂU CHUYỆN

Giá trị pid không biểu thị một tiến trình đang chạy.

Xác định phạm vi ưu tiên hợp lệ Các

ví dụ trước đây của chúng tôi đã truyền các giá trị ưu tiên được mã hóa cứng vào các lệnh gọi hệ thống lập lịch. POSIX không đảm bảo về các ưu tiên lập lịch nào tồn tại trên một hệ thống nhất định, ngoại trừ việc nói rằng phải có ít nhất 32 ưu tiên giữa các giá trị tối thiểu và tối đa. Như đã đề cập trước đó trong "Chính sách và ưu tiên lập lịch Linux", Linux triển khai phạm vi từ 1 đến 99 bao gồm cho hai chính sách lập lịch thời gian thực. Một chương trình sạch, có thể di động thường triển khai phạm vi giá trị ưu tiên của riêng nó và ánh xạ chúng vào phạm vi của hệ điều hành. Ví dụ, nếu bạn muốn chạy các quy trình ở bốn mức ưu tiên thời gian thực khác nhau, bạn sẽ xác định phạm vi ưu tiên một cách động và chọn bốn giá trị.

Linux cung cấp hai lệnh gọi hệ thống để lấy phạm vi giá trị ưu tiên hợp lệ. Một lệnh trả về giá trị tối thiểu và lệnh còn lại trả về giá trị tối đa:

```
#include <sched.h>

int sched_get_priority_min (int chính sách);

int sched_get_priority_max (int policy); Khi
```

thành công, lệnh sched_get_priority_min() trả về giá trị tối thiểu và lệnh sched_get_priority_max() trả về mức ưu tiên hợp lệ tối đa liên quan đến chính sách lập lịch được biểu thị bằng policy. Cả hai lệnh sau đó trả về 0. Khi thất bại, cả hai lệnh đều trả về -1. Lỗi duy nhất có thể xảy ra là nếu policy không hợp lệ, trong trường hợp đó errno là
đặt thành EINVAL.

Cách sử dụng rất đơn giản:

```
int min, max;

min = sched_get_priority_min (SCHED_RR); nếu
(min == -1)
          { perror ("sched_get_priority_min"); trả
          về 1;
}

max = sched_get_priority_max (SCHED_RR); nếu
(max == -1)
          { perror ("sched_get_priority_max"); trả
          về 1;
}

printf ("Phạm vi ưu tiên của SCHED_RR là %d - %d\n", min, max);
```

Trên hệ thống Linux chuẩn, đoạn mã này sẽ cho kết quả như sau:

```
Phạm vi ưu tiên của SCHED_RR là 1 - 99
```

Như đã thảo luận trước đó, giá trị ưu tiên lớn hơn về mặt số biểu thị mức độ ưu tiên cao hơn.
Để thiết lập một quy trình ở mức ưu tiên cao nhất cho chính sách lập lịch của quy trình đó, bạn có thể thực hiện như sau:

```
/*
 * set_highest_priority – đặt mức ưu tiên lập lịch của pid liên quan thành giá trị
 *
   cao nhất được phép theo chính sách lập lịch hiện tại của nó. Nếu pid
 bằng không, đặt mức ưu tiên của * quy trình hiện tại.
 *
 * Trả về số 0 nếu thành công. */

int đặt_ưu_tiên_cao_nhất (pid_t pid) {

          struct sched_param sp;
          int chính sách, max, ret;

          chính sách = sched_getscheduler (pid);
          nếu (chính sách ==
                    -1) trả về -1;

          max = sched_get_priority_max (chính sách);
          nếu (max == -1)
                    trả về -1;

          memset (&sp, 0, sizeof (struct sched_param)); sp.
          sched_priority = max; ret =
          sched_setparam (pid, &sp);

          trả về ret;
}
```

Các chương trình thường lấy giá trị tối thiểu hoặc tối đa của hệ thống, sau đó sử dụng các mức tăng dần từ 1 (như max-1, max-2, v.v.) để chỉ định mức độ ưu tiên theo ý muốn.

lịch_lịch_lịch_lấy_khoảng_thời_gian( )

Như đã thảo luận trước đó, các quy trình SCHED_RR hoạt động giống như các quy trình SCHED_FIFO , ngoại trừ việc trình lập lịch gán các lát cắt thời gian cho các quy trình này. Khi một quy trình SCHED_RR sử dụng hết lát cắt thời gian của mình, trình lập lịch di chuyển quy trình đến cuối danh sách chạy cho mức ưu tiên hiện tại của nó. Theo cách này, tất cả các quy trình SCHED_RR có cùng mức ưu tiên được thực thi theo vòng xoay vòng. Các quy trình có mức ưu tiên cao hơn (và các quy trình SCHED_FIFO có cùng mức ưu tiên hoặc mức ưu tiên cao hơn) sẽ luôn chiếm quyền trước một quy trình SCHED_RR đang chạy , bất kể quy trình đó có còn lát cắt thời gian nào hay không.

POSIX định nghĩa một giao diện để lấy độ dài của một khoảng thời gian nhất định của quy trình:

```
#include <sched.h>

cấu trúc timespec
          { thời gian_t           /* giây */ /*
          tv_sec; dài tv_nsec;    nano giây */
};

int sched_rr_get_interval (pid_t pid, struct timespec *tp);
```

Một cuộc gọi thành công đến sched_rr_get_interval( ) có tên rất tệ sẽ lưu trong cấu trúc timespec được trỏ tới bởi tp khoảng thời gian của timeslice được phân bổ cho pid và trả về 0. Khi cuộc gọi thất bại, cuộc gọi trả về -1 và errno được đặt ở mức phù hợp.

Theo POSIX, chức năng này chỉ cần thiết để hoạt động với các quy trình SCHED_RR .
Tuy nhiên, trên Linux, nó có thể lấy độ dài của bất kỳ khoảng thời gian nào của quy trình. Các ứng dụng di động nên cho rằng hàm này chỉ hoạt động với các quy trình luân phiên; các chương trình dành riêng cho Linux có thể khai thác lệnh gọi khi cần. Sau đây là một ví dụ:

```
cấu trúc timespec tp;
int ret;

/* lấy độ dài khoảng thời gian của tác vụ hiện tại */
ret = sched_rr_get_interval (0, &tp); if (ret
== -1) { perror
          ("sched_rr_get_interval"); return 1;

}

/* chuyển đổi giây và nano giây sang mili giây */
printf ("Thời gian lượng tử của chúng ta là %.2lf mili giây\n",
          (tp.tv_sec * 1000.0f) + (tp.tv_nsec / 1000000.0f));
```

Nếu tiến trình đang chạy trong lớp FIFO, tv_sec và tv_nsec đều bằng 0, biểu thị vô cực.

Mã lỗi

Khi xảy ra lỗi, có thể có ba giá trị errno :

EFAULT

Bộ nhớ được trỏ tới bởi con trỏ tp không hợp lệ hoặc không thể truy cập được.

EINVAL

Giá trị pid không hợp lệ (ví dụ: giá trị này là số âm).

ESRCH

Giá trị pid hợp lệ nhưng đề cập đến một quy trình không tồn tại.

Các biện pháp phòng ngừa với quy trình thời gian thực

Do bản chất của các quy trình thời gian thực, các nhà phát triển nên thận trọng khi phát triển và gỡ lỗi các chương trình như vậy. Nếu một chương trình thời gian thực đi quá xa, hệ thống có thể trở nên không phản hồi. Bất kỳ vòng lặp nào bị ràng buộc bởi CPU trong một chương trình thời gian thực—tức là bất kỳ đoạn mã nào không chặn—sẽ tiếp tục chạy vô hạn, miễn là không có quy trình thời gian thực nào có mức độ ưu tiên cao hơn có thể chạy được.

Do đó, việc thiết kế các chương trình thời gian thực đòi hỏi sự cẩn thận và chú ý. Các chương trình như vậy thống trị tối cao và có thể dễ dàng làm sập toàn bộ hệ thống. Sau đây là một số mẹo và biện pháp phòng ngừa:

- Hãy nhớ rằng bất kỳ vòng lặp nào liên kết với CPU sẽ chạy cho đến khi hoàn tất, không bị gián đoạn, nếu không có quy trình thời gian thực có mức ưu tiên cao hơn trên hệ thống. Nếu vòng lặp là vô hạn, hệ thống sẽ không phản hồi.
- Vì các quy trình thời gian thực chạy bằng mọi thứ khác trên hệ thống, nên phải đặc biệt chú ý đến thiết kế của chúng. Cẩn thận không để phần còn lại của hệ thống bị thiếu thời gian xử lý.

- Rất cẩn thận với tình trạng chờ bận. Nếu một tiến trình thời gian thực chờ bận một tài nguyên do một tiến trình có mức độ ưu tiên thấp hơn nắm giữ, tiến trình thời gian thực đó sẽ chờ bận mãi mãi. • Trong khi phát triển một tiến trình thời gian thực, hãy giữ một thiết bị đầu cuối mở, chạy như một tiến trình thời gian thực có mức độ ưu tiên cao hơn tiến trình đang phát triển. Trong trường hợp khẩn cấp, thiết bị đầu cuối sẽ vẫn phản hồi và cho phép bạn tắt tiến trình thời gian thực đang chạy trốn. (Vì thiết bị đầu cuối vẫn ở trạng thái nhàn rỗi, chờ nhập dữ liệu từ bàn phím nên nó sẽ không can thiệp vào tiến trình thời gian thực khác.) • Tiện ích chrt , một phần của gói công cụ util-linux , giúp bạn dễ dàng truy xuất và đặt các thuộc tính thời gian thực của các tiến trình khác. Công cụ này giúp bạn dễ dàng khởi chạy các chương trình tùy ý trong lớp lập lịch thời gian thực, chẳng hạn như thiết bị đầu cuối đã đề cập ở trên hoặc thay đổi các mức độ ưu tiên thời gian thực của các ứng dụng hiện có.

Chủ nghĩa quyết định

Các quy trình thời gian thực có tính quyết định rất lớn. Trong điện toán thời gian thực, một hành động là quyết định nếu, với cùng một đầu vào, nó luôn tạo ra cùng một kết quả trong cùng một khoảng thời gian. Máy tính hiện đại chính là định nghĩa của một thứ không quyết định: nhiều cấp bộ nhớ đệm (gây ra tình trạng trúng hoặc trượt mà không thể dự đoán được), nhiều bộ xử lý, phân trang, hoán đổi và đa nhiệm gây ra sự hỗn loạn đối với bất kỳ ước tính nào về thời gian thực hiện một hành động nhất định. Chắc chắn, chúng ta đã đạt đến một điểm mà hầu như mọi hành động (modulo truy cập ổ cứng) đều "cực kỳ nhanh", nhưng đồng thời, các hệ thống hiện đại cũng khiến việc xác định chính xác thời gian thực hiện một hoạt động nhất định trở nên khó khăn.

Các ứng dụng thời gian thực thường cố gắng hạn chế tính không thể đoán trước nói chung và sự chậm trễ trong trường hợp xấu nhất nói riêng. Các phần sau đây thảo luận về hai phương pháp được sử dụng cho mục đích này.

Dữ liệu lỗi trước và khóa bộ nhớ Hãy

tưởng tượng thế này: ngắt phần cứng từ màn hình ICBM tùy chỉnh đến và trình điều khiển của thiết bị nhanh chóng sao chép dữ liệu từ phần cứng vào hạt nhân. Trình điều khiển lưu ý rằng một quy trình đang ngủ, bị chặn trên nút thiết bị của phần cứng, đang chờ dữ liệu. Trình điều khiển yêu cầu hạt nhân đánh thức quy trình. Hạt nhân, lưu ý rằng quy trình này đang chạy với chính sách lập lịch thời gian thực và mức độ ưu tiên cao, ngay lập tức chiếm quyền điều khiển quy trình hiện đang chạy và chuyển sang chế độ tăng tốc, quyết tâm lập lịch quy trình thời gian thực ngay lập tức. Trình lập lịch chuyển sang chạy

quá trình thời gian thực và chuyển đổi ngữ cảnh vào không gian địa chỉ của nó. Quá trình hiện đang chạy. Toàn bộ quá trình mất 0,3 ms, nằm trong khoảng thời gian trễ chấp nhận được trong trường hợp xấu nhất là 1 ms.

Bây giờ, trong không gian người dùng, quy trình thời gian thực ghi nhận ICBM đang bay tới và bắt đầu xử lý quỹ đạo của nó. Với đường đạn được tính toán, quy trình thời gian thực khởi tạo việc triển khai hệ thống tên lửa chống đạn đạo. Chỉ còn 0,1 ms nữa trôi qua—đủ nhanh để triển khai phản ứng ABM và cứu mạng người. Nhưng—ôi không!—mã ABM đã được hoán đổi sang đĩa. Lỗi trang xảy ra, bộ xử lý chuyển trở lại chế độ hạt nhân và hạt nhân khởi tạo I/O đĩa cứng để truy xuất dữ liệu đã hoán đổi.
Bộ lập lịch sẽ đưa tiến trình vào trạng thái ngủ cho đến khi lỗi trang được khắc phục. Vài giây trôi qua. Quá muộn rồi.

Rõ ràng, phân trang và hoán đổi tạo ra hành vi không xác định có thể gây ra sự tàn phá cho một quy trình thời gian thực. Để ngăn chặn thảm họa này, một ứng dụng thời gian thực thường sẽ "khóa" hoặc "kết nối cứng" tất cả các trang trong không gian địa chỉ của nó vào bộ nhớ vật lý, đưa chúng vào bộ nhớ trước và ngăn không cho chúng bị hoán đổi ra ngoài. Sau khi các trang bị khóa vào bộ nhớ, nhân sẽ không bao giờ hoán đổi chúng ra đĩa. Bất kỳ truy cập nào vào các trang sẽ không gây ra lỗi trang. Hầu hết các ứng dụng thời gian thực đều khóa một số hoặc tất cả các trang của chúng vào bộ nhớ vật lý.

Linux cung cấp giao diện cho cả dữ liệu lỗi trước và dữ liệu khóa. Chương 4 thảo luận về giao diện cho dữ liệu lỗi trước vào bộ nhớ vật lý. Chương 8 sẽ thảo luận về giao diện cho dữ liệu khóa vào bộ nhớ vật lý.

Mối quan tâm thứ hai của các ứng dụng

thời gian thực là đa nhiệm. Mặc dù nhân Linux có tính ưu tiên, nhưng trình lập lịch của nó không phải lúc nào cũng có thể lập lịch lại ngay lập tức một quy trình để ưu tiên cho quy trình khác. Đôi khi, quy trình hiện đang chạy đang thực thi bên trong một vùng quan trọng trong nhân và trình lập lịch không thể ưu tiên cho quy trình đó cho đến khi quy trình đó thoát khỏi vùng đó. Nếu quy trình đang chờ chạy là thời gian thực, sự chậm trễ này có thể không được chấp nhận, nhanh chóng vượt quá thời hạn hoạt động.

Do đó, đa nhiệm đưa ra sự bất định có bản chất tương tự như tính không thể đoán trước liên quan đến phân trang. Giải pháp liên quan đến đa nhiệm là như nhau: loại bỏ nó. Tất nhiên, khả năng là bạn không thể chỉ đơn giản xóa bỏ tất cả các quy trình khác. Nếu điều đó khả thi trong môi trường của bạn, có lẽ bạn sẽ không cần Linux ngay từ đầu—một hệ điều hành tùy chỉnh đơn giản sẽ đủ. Tuy nhiên, nếu hệ thống của bạn có nhiều bộ xử lý, bạn có thể dành một hoặc nhiều bộ xử lý đó cho quy trình hoặc các quy trình thời gian thực của mình. Trên thực tế, bạn có thể bảo vệ các quy trình thời gian thực khỏi đa nhiệm.

Chúng ta đã thảo luận về các lệnh gọi hệ thống để điều khiển CPUaffinity của một quy trình ở phần đầu của chư ơ ng này. Một giải pháp tối ư u hóa tiềm năng cho các ứng dụng thời gian thực là dành riêng một bộ xử lý cho mỗi quy trình thời gian thực và để tất cả các quy trình khác chia sẻ thời gian trên bộ xử lý còn lại.

Cách đơ n giản nhất để thực hiện điều này là sửa đổi chư ơ ng trình khởi tạo của Linux , SysVinit,* để thực hiện thao tác tư ơ ng tự như sau trư ớc khi bắt đầu quá trình khởi động:

```
cpu_set_t đặt;
int ret;

CPU_ZERO (&set);              /* xóa tất cả CPU */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set); nếu (ret
== -1) { perror
          ("sched_getaffinity"); trả về 1;

}

CPU_CLR (1, &set);            /* cấm CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set); nếu (ret
== -1) { perror
          ("sched_setaffinity"); trả về 1;

}
```

Đoạn mã này lấy bộ xử lý đư ợc phép hiện tại của init, mà chúng tôi mong đợi là tất cả chúng. Sau đó, nó xóa một bộ xử lý, CPU#1, khỏi bộ xử lý và cập nhật danh sách các bộ xử lý đư ợc phép.

Vì bộ xử lý đư ợc phép đư ợc thừa hư ởng từ tiến trình cha sang tiến trình con và init là tiến trình cha của tất cả các tiến trình, nên tất cả các tiến trình của hệ thống sẽ chạy với bộ xử lý đư ợc phép này. Do đó, không có tiến trình nào sẽ chạy trên CPU #1.

Tiếp theo, hãy sửa đổi quy trình thời gian thực của bạn để chỉ chạy trên CPU #1:

```
cpu_set_t đặt;
int ret;

CPU_ZERO (&set);              /* xóa tất cả CPU */ /*
CPU_CLR (1, &set);            cấm CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set); nếu (ret
== -1) { perror
          ("sched_setaffinity"); trả về 1;

}
```

Kết quả là quy trình thời gian thực của bạn chỉ chạy trên CPU#1 và tất cả các quy trình khác chạy trên các bộ xử lý khác.

* Nguồn SysVinit nằm tại ftp://ftp.cistron.nl/pub/people/miquels/sysvinit/. Nó đư ợc cấp phép theo Giấy phép Công cộng GNU v2.

Giới hạn tài nguyên

Nhân Linux áp đặt một số giới hạn tài nguyên cho các tiến trình. Các giới hạn tài nguyên này đặt ra các giới hạn cứng về lượng tài nguyên nhân mà một tiến trình có thể sử dụng—tức là, số lượng tệp đang mở, các trang bộ nhớ, các tín hiệu đang chờ xử lý, v.v. Các giới hạn này được thực thi nghiêm ngặt; nhân sẽ không cho phép một hành động đặt mức tiêu thụ tài nguyên của một tiến trình vượt quá giới hạn cứng. Ví dụ, nếu việc mở một tệp khiến một tiến trình có nhiều tệp đang mở hơn mức giới hạn tài nguyên được áp dụng cho phép, thì lệnh gọi open() sẽ không thành công.* Linux cung cấp hai lệnh gọi hệ thống để thao tác giới hạn tài nguyên. POSIX đã

chuẩn hóa cả hai giao diện, nhưng Linux hỗ trợ một số giới hạn ngoài những giới hạn do tiêu chuẩn quy định. Có thể kiểm tra giới hạn bằng getrlimit() và đặt bằng setrlimit( ):

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit
          { rlim_t rlim_cur; /* giới hạn mềm */ rlim_t
          rlim_max; /* giới hạn cứng */
};

int getrlimit (int tài nguyên, struct rlimit *rlim); int
setrlimit (int tài nguyên, const struct rlimit *rlim);
```

Các hằng số nguyên, chẳng hạn như RLIMIT_CPU, biểu diễn các tài nguyên. Cấu trúc rlimit biểu diễn các giới hạn thực tế. Cấu trúc này định nghĩa hai giới hạn: giới hạn mềm và giới hạn cứng. Nhân thực thi các giới hạn tài nguyên mềm đối với các quy trình, nhưng một quy trình có thể tự do thay đổi giới hạn mềm của mình thành bất kỳ giá trị nào từ 0 đến và bao gồm giới hạn cứng. Một quy trình không có khả năng CAP_SYS_RESOURCE (tức là bất kỳ quy trình nào không phải gốc) chỉ có thể hạ thấp giới hạn cứng của mình. Một quy trình không có đặc quyền không bao giờ có thể tăng giới hạn cứng của mình, thậm chí không phải lên một giá trị cao hơn trước đó; việc hạ thấp giới hạn cứng là không thể đảo ngược. Một quy trình có đặc quyền có thể đặt giới hạn cứng thành bất kỳ giá trị hợp lệ nào.

Giới hạn thực sự biểu diễn điều gì phụ thuộc vào tài nguyên đang đề cập. Ví dụ, nếu tài nguyên là RLIMIT_FSIZE, giới hạn biểu diễn kích thước tối đa của tệp mà một quy trình có thể tạo, tính bằng byte. Trong trường hợp này, nếu rlim_cur là 1.024, quy trình không thể tạo hoặc mở rộng tệp đến kích thước lớn hơn một kilobyte.

Tất cả các giới hạn tài nguyên đều có hai giá trị đặc biệt: 0 và vô cực. Giá trị đầu tiên vô hiệu hóa hoàn toàn việc sử dụng tài nguyên. Ví dụ, nếu RLIMIT_CORE là 0, hạt nhân sẽ không bao giờ tạo tệp lõi. Ngược lại, giá trị sau sẽ xóa mọi giới hạn đối với tài nguyên. Ker-nel biểu thị vô cực bằng giá trị đặc biệt RLIM_INFINITY, tình cờ là -1 (điều này có thể gây ra một số nhầm lẫn, vì -1 cũng là giá trị trả về biểu thị lỗi). Nếu RLIMIT_CORE là vô cực, hạt nhân sẽ tạo tệp lõi có bất kỳ kích thước nào.

* Trong trường hợp đó, lệnh gọi sẽ đặt errno thành EMFILE, cho biết rằng quy trình đã đạt đến giới hạn tài nguyên trên số lượng tệp mở tối đa. Chương 2 thảo luận về lệnh gọi hệ thống open() .

Hàm getrlimit() đặt giới hạn cứng và giới hạn mềm hiện tại trên tài nguyên đư ợc biểu thị bằng resource trong cấu trúc đư ợc trỏ tới bởi rlim. Khi thành công, lệnh gọi trả về 0. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành giá trị phù hợp.

Tư ơ ng ứng, hàm setrlimit() đặt các giới hạn cứng và mềm liên quan đến tài nguyên thành các giá trị đư ợc trỏ tới bởi rlim. Khi thành công, lệnh gọi trả về 0 và hạt nhân cập nhật các giới hạn tài nguyên theo yêu cầu. Khi thất bại, lệnh gọi trả về -1 và đặt errno theo giá trị phù hợp.

Giới hạn

Linux hiện cung cấp 15 giới hạn tài nguyên:

RLIMIT_AS

Giới hạn kích thư ớc tối đa của không gian địa chỉ của một tiến trình, tính bằng byte. Các nỗ lực tăng kích thư ớc của không gian địa chỉ vư ợt quá giới hạn này—thông qua các lệnh gọi như mmap( ) và brk( )—sẽ thất bại và trả về ENOMEM. Nếu ngăn xếp của tiến trình, tự động phát triển khi cần, mở rộng vư ợt quá giới hạn này, thì hạt nhân sẽ gửi cho tiến trình tín hiệu SIGSEGV . Giới hạn này thư ờng là RLIM_INFINITY.

RLIMIT_CORE

Chỉ định kích thư ớc tối đa của các tệp lõi, tính bằng byte. Nếu khác không, các tệp lõi lớn hơ n giới hạn này sẽ bị cắt bớt đến kích thư ớc tối đa. Nếu 0, các tệp lõi sẽ không bao giờ đư ợc tạo.

RLIMIT_CPU

Chỉ định thời gian CPU tối đa mà một tiến trình có thể sử dụng, tính bằng giây. Nếu một tiến trình chạy lâu hơ n giới hạn này, hạt nhân sẽ gửi cho nó tín hiệu SIGXCPU , mà các tiến trình có thể bắt và xử lý. Các chư ơ ng trình di động sẽ kết thúc khi nhận đư ợc tín hiệu này, vì POSIX không xác định đư ợc hành động tiếp theo mà hạt nhân có thể thực hiện.
Một số hệ thống có thể chấm dứt tiến trình nếu nó tiếp tục chạy. Tuy nhiên, Linux cho phép tiến trình tiếp tục thực thi và tiếp tục gửi tín hiệu SIGXCPU theo khoảng thời gian một giây. Khi tiến trình đạt đến giới hạn cứng, nó sẽ đư ợc gửi SIGKILL và chấm dứt.

DỮ LIỆU GIỚI HẠN

Kiểm soát kích thư ớc tối đa của phân đoạn dữ liệu và đống dữ liệu của quy trình, tính bằng byte. Các nỗ lực mở rộng phân đoạn dữ liệu vư ợt quá giới hạn này thông qua brk() sẽ thất bại và trả về ENOMEM.

RLIMIT_FSIZE Chỉ

định kích thư ớc tệp tối đa mà một tiến trình có thể tạo, tính bằng byte. Nếu một tiến trình mở rộng tệp vư ợt quá kích thư ớc này, hạt nhân sẽ gửi cho tiến trình tín hiệu SIGXFSZ . Theo mặc định, tín hiệu này sẽ chấm dứt tiến trình. Tuy nhiên, một tiến trình có thể chọn bắt và xử lý tín hiệu này, trong trư ờng hợp đó, lệnh gọi hệ thống vi phạm sẽ không thành công và trả về EFBIG.

RLIMIT_LOCKS

Kiểm soát số lượng khóa tệp tối đa mà một quy trình có thể giữ (xem Chương 7 để thảo luận về khóa tệp). Khi đạt đến giới hạn này, các nỗ lực tiếp theo để có được các khóa tệp bổ sung sẽ không thành công và trả về ENOLCK. Tuy nhiên, hạt nhân Linux 2.4.25 đã xóa chức năng này. Trong các hạt nhân hiện tại, giới hạn này có thể thiết lập được nhưng không có hiệu lực.

RLIMIT_MEMLOCK

Chỉ định số byte bộ nhớ tối đa mà một tiến trình không có khả năng CAP_SYS_IPC (thực tế là một tiến trình không phải root) có thể khóa vào bộ nhớ thông qua mlock( ), mlockall( ) hoặc shmctl( ). Nếu vượt quá giới hạn này, các lệnh gọi này sẽ không thành công và trả về EPERM. Trong thực tế, giới hạn hiệu quả được làm tròn xuống thành bội số nguyên của các trang. Các tiến trình sở hữu CAP_SYS_IPC có thể khóa bất kỳ số trang nào vào bộ nhớ và giới hạn này không có hiệu lực. Trước kernel 2.6.9, giới hạn này là giới hạn tối đa mà một tiến trình có CAP_SYS_IPC có thể khóa vào bộ nhớ và các tiến trình không có đặc quyền không thể khóa bất kỳ trang nào. Giới hạn này không phải là một phần của POSIX; BSD đã giới thiệu nó.

RLIMIT_MSGQUEUE

Chỉ định số byte tối đa mà người dùng có thể phân bổ cho hàng đợi tin nhắn POSIX. Nếu hàng đợi tin nhắn mới tạo vượt quá giới hạn này, mq_open() sẽ không thành công và trả về ENOMEM. Giới hạn này không phải là một phần của POSIX; nó được thêm vào kernel 2.6.8 và dành riêng cho Linux.

RLIMIT_TỐT

Chỉ định giá trị tối đa mà một tiến trình có thể hạ thấp giá trị nice của nó (tăng mức ưu tiên của nó). Như đã thảo luận trước đó trong chương này, thông thường các tiến trình chỉ có thể nâng cao giá trị nice của chúng (giảm mức ưu tiên của chúng). Giới hạn này cho phép người quản trị áp đặt mức tối đa (mức sàn giá trị nice) mà các tiến trình có thể hợp pháp nâng mức ưu tiên của chúng. Vì các giá trị nice có thể là số âm, nên hạt nhân diễn giải giá trị là 20 - rlim_cur. Do đó, nếu giới hạn này được đặt thành 40, một tiến trình có thể hạ thấp giá trị nice của nó xuống giá trị tối thiểu là –20 (mức ưu tiên cao nhất). Hạt nhân 2.6.12 đã giới thiệu giới hạn này.

RLIMIT_NOFILE

Chỉ định một số lớn hơn số lượng tối đa các mô tả tệp mà một quy trình có thể giữ mở. Các nỗ lực vượt quá giới hạn này sẽ dẫn đến lỗi và lệnh gọi hệ thống áp dụng sẽ trả về EMFILE. Giới hạn này cũng có thể chỉ định là RLIMIT_OFILE, đây là tên BSD đặt cho nó.

RLIMIT_NPROC

Chỉ định số lượng tối đa các tiến trình mà người dùng có thể chạy trên hệ thống tại bất kỳ thời điểm nào. Các nỗ lực vượt quá giới hạn này sẽ dẫn đến lỗi và fork( ) trả về EAGAIN. Giới hạn này không phải là một phần của POSIX; BSD đã giới thiệu giới hạn này.

RLIMIT_RSS

Chỉ định số lượng trang tối đa mà một quy trình có thể lưu trú trong bộ nhớ (được gọi là kích thước tập hợp lưu trú hoặc RSS). Chỉ có các hạt nhân 2.4 đầu tiên được áp dụng

giới hạn này. Các hạt nhân hiện tại cho phép thiết lập giới hạn này, như ng nó không đư ợc áp dụng. Giới hạn này không phải là một phần của POSIX; BSD đã giới thiệu nó.

RLIMIT_RTPRIO

Chỉ định mức độ ư u tiên thời gian thực tối đa của một quy trình không có CAP_SYS_NICE khả năng (thực tế là các quy trình không phải gốc) có thể yêu cầu. Thông thư ờng, không có đặc quyền các quy trình có thể không yêu cầu bất kỳ lớp lập lịch thời gian thực nào. Giới hạn này không phải là một phần của POSIX; đư ợc thêm vào trong kernel 2.6.12 và dành riêng cho Linux.

RLIMIT_SẮP KẾT THÚC

Chỉ định số lư ợng tín hiệu tối đa (chuẩn và thời gian thực) có thể xếp hàng cho ngư ời dùng này. Các nỗ lực xếp hàng các tín hiệu bổ sung không thành công và các cuộc gọi hệ thống chẳng hạn như sigqueue() trả về EAGAIN. Lư u ý rằng điều này luôn có thể, bất kể giới hạn này, để xếp hàng một trư ờng hợp của tín hiệu chư a đư ợc xếp hàng. Do đó, nó là luôn có thể cung cấp cho quy trình một SIGKILL hoặc SIGTERM. Giới hạn này không phải là một phần của POSIX; nó dành riêng cho Linux.

GIỚI HẠN_NGĂN CHẶN

Biểu thị kích thư ớc tối đa của ngăn xếp quy trình, tính bằng byte. Vư ợt quá giới hạn này dẫn đến việc cung cấp SIGSEGV.

Hạt nhân lư u trữ các giới hạn trên cơ sở mỗi ngư ời dùng. Nói cách khác, tất cả các quy trình đư ợc chạy bởi cùng một ngư ời dùng sẽ có cùng giới hạn mềm và cứng cho bất kỳ tài nguyên nào. Tuy nhiên, bản thân các giới hạn có thể mô tả giới hạn cho mỗi quy trình (không phải cho mỗi ngư ời dùng). Ví dụ: hạt nhân duy trì giá trị của RLIMIT_NOFILE trên cơ sở mỗi ngư ời dùng; theo mặc định, nó là 1024. Tuy nhiên, giới hạn này quyết định số lư ợng tệp tối đa mà mỗi quy trình có thể mở, không phải số lư ợng ngư ời dùng có thể mở tổng thể. Lư u ý rằng điều này không có nghĩa là giới hạn có thể đư ợc cấu hình riêng cho từng quy trình của ngư ời dùng— nếu một quy trình thay đổi giới hạn mềm của RLIMIT_NOFILE , thay đổi sẽ áp dụng cho tất cả các quy trình do ngư ời dùng đó sở hữu.

Giới hạn mặc định

Các giới hạn mặc định có sẵn cho quy trình của bạn phụ thuộc vào ba biến: phần mềm ban đầu giới hạn, giới hạn cứng ban đầu và quản trị viên hệ thống của bạn. Hạt nhân quyết định giới hạn cứng và mềm ban đầu; Bảng 6-1 liệt kê chúng. Hạt nhân đặt các giới hạn này trên quá trình khởi tạo, và vì con cái thừa hư ởng giới hạn của cha mẹ chúng, tất cả các quá trình tiếp theo các tiến trình thừa hư ởng các giới hạn mềm và cứng của init.

Bảng 6-1. Giới hạn tài nguyên cứng và mềm mặc định

Giới hạn tài nguyên	Giới hạn mềm	Giới hạn cứng
GIỚI HẠN NHƯ	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG
GIỚI HẠN_LỖI	0	RLIM_VÔ CÙNG
CPU GIỚI HẠN	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG
DỮ LIỆU GIỚI HẠN	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG
Kích thư ớc giới hạn	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG

Bảng 6-1. Giới hạn tài nguyên cứng và mềm mặc định (tiếp theo)

Giới hạn tài nguyên	Giới hạn mềm	Giới hạn cứng
KHÓA GIỚI HẠN	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG
KHÓA GIỚI HẠN	8 trang	8 trang
RLIMIT_MSGQUEUE	800KB	800KB
RLIMIT_TỐT	0	0
RLIMIT_KHÔNG CÓ TỆP	1024	1024
Giới hạn NPROC	0 (ngụ ý không có giới hạn)	0 (ngụ ý không có giới hạn)
Giới hạn RSS	RLIM_VÔ CÙNG	RLIM_VÔ CÙNG
Giới hạn RTPRIO0		0
RLIMIT_SẮP KẾT THÚC	0	0
GIỚI HẠN_NGĂN CHẶN	8MB	RLIM_VÔ CÙNG

Có hai điều có thể thay đổi các giới hạn mặc định này:

- Bất kỳ quy trình nào cũng có thể tự do tăng giới hạn mềm lên bất kỳ giá trị nào từ 0 đến giới hạn cứng, hoặc để giảm giới hạn cứng. Trẻ em sẽ thừa hưởng những giới hạn được cập nhật này trong cái nĩa.
- Một tiến trình đặc quyền được tự do thiết lập giới hạn cứng cho bất kỳ giá trị nào. Con cái sẽ thừa hưởng những giới hạn được cập nhật này trong quá trình phân nhánh.

Không có khả năng một tiến trình gốc trong dòng dõi của một tiến trình thông thường sẽ thay đổi bất kỳ điều gì khó khăn giới hạn. Do đó, mục đầu tiên có nhiều khả năng là nguồn thay đổi giới hạn hơn hơn thứ hai. Thật vậy, các giới hạn thực tế được trình bày cho một quá trình thường được thiết lập bởi shell của người dùng, được quản trị viên hệ thống thiết lập để cung cấp nhiều giới hạn khác nhau. Ví dụ, trong shell Bourne-again (bash), người quản trị thực hiện việc này thông qua lệnh ulimit . Lưu ý rằng người quản trị không cần hạ thấp giá trị; anh ta có thể cũng nâng giới hạn mềm lên giới hạn cứng, cung cấp cho người dùng các mặc định hợp lý hơn. Đây là thường được thực hiện với RLIMIT_STACK, được đặt thành RLIM_INFINITY trên nhiều hệ thống.

Thiết lập và Truy xuất Giới hạn

Với những giải thích về các giới hạn tài nguyên khác nhau, chúng ta hãy xem xét việc truy xuất và thiết lập giới hạn. Truy xuất giới hạn tài nguyên khá đơn giản:

```
cấu trúc rlimit rlim;
int ret;

/* lấy giới hạn về kích thước lõi */
ret = getrlimit(RLIMIT_CORE, &rlim);
nếu (ret == -1) {
          lỗi ("getrlimit");
          trả về 1;
}

printf ("Giới hạn RLIMIT_CORE: mềm=%ld cứng=%ld\n",
          rlim.rlim_cur, rlim.rlim_max);
```

Biên dịch đoạn mã này thành một chư ơ ng trình lớn hơ n và chạy nó sẽ cho kết quả như sau:

```
Giới hạn RLIMIT_CORE: mềm=0 cứng=-1
```

Chúng ta có giới hạn mềm là 0 và giới hạn cứng là vô cực (-1 biểu thị RLIM_INFINITY).

Do đó, chúng ta có thể đặt giới hạn mềm mới ở bất kỳ kích thư ớc nào. Ví dụ này đặt kích thư ớc lõi tối đa là 32 MB:

```
cấu trúc rlimit rlim;
int ret;

                          *         *
rlim.rlim_cur = 32 1024 rlim.rlim_max    1024; /* 32 MB */ /
= RLIM_INFINITY; ret = setrlimit (RLIMIT_CORE,     * đừng quan tâm */
&rlim); nếu (ret == -1) { perror ("setrlimit"); trả về
1;


}
```

Mã lỗi

Khi xảy ra lỗi, có thể có ba mã lỗi sau :

MẶC ĐỊNH

Bộ nhớ đư ợc rlim trỏ tới không hợp lệ hoặc không thể truy cập đư ợc.

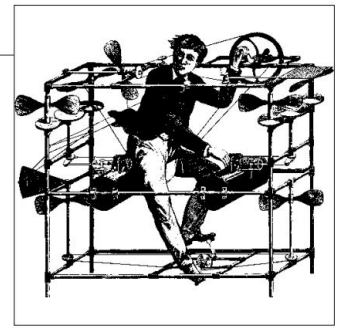
EINVAL

Giá trị đư ợc biểu thị bằng resource không hợp lệ hoặc rlim.rlim_cur lớn hơ n rlim.rlim_max (chỉ setrlimit()).

EPERM

Ngư ời gọi không sở hữu CAP_SYS_RESOURCE như ng đã cố gắng tăng giới hạn cứng.

Tập tin và thư mục Sự quản lý



Trong Chương 2, 3 và 4, chúng ta đã đề cập đến nhiều cách tiếp cận đối với tệp I/O. Trong chương này, chúng ta sẽ xem lại các tệp, lần này không tập trung vào việc đọc hoặc ghi vào chúng mà tập trung vào việc thao tác và quản lý chúng cùng siêu dữ liệu của chúng.

Các tập tin và siêu dữ liệu của chúng

Như đã thảo luận trong Chương 1, mỗi tệp được tham chiếu bởi một inode, được giải quyết bằng một giá trị số duy nhất của hệ thống tệp được gọi là số inode. Một inode vừa là một đối tượng vật lý nằm trên đĩa của hệ thống tệp kiểu Unix, vừa là một thực thể khái niệm được biểu diễn bằng một cấu trúc dữ liệu trong hạt nhân Linux. Inode lưu trữ siêu dữ liệu liên quan đến một tệp, chẳng hạn như quyền truy cập của tệp, dấu thời gian truy cập cuối cùng, chủ sở hữu, nhóm và kích thước, cũng như vị trí dữ liệu của tệp.

Bạn có thể lấy số inode cho một tệp bằng cách sử dụng cờ -i cho lệnh ls :

```
$ ls -i
1689459 Kconfig 1689461 main.c 1680144 process.c 1689464 swsusp.c 1680137
Makefile 1680141 pm.c 1680145 smp.c 1680149 user.c 1680138 console.c 1689462
power.h 1689463 snapshot.c 1689460 disk.c 1680143 poweroff.c
1680147 swap.c
```

Đầu ra này cho thấy, ví dụ, disk.c có số inode là 1689460. Trên hệ thống tệp cụ thể này, không có tệp nào khác có số inode này. Tuy nhiên, trên một hệ thống tệp khác, chúng tôi không thể đưa ra bất kỳ đảm bảo nào như vậy.

Gia đình Stat

Unix cung cấp một nhóm các hàm để lấy siêu dữ liệu của một tệp:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat (const char *đường dẫn, struct stat
*buf); int fstat (int fd, struct stat
*buf); int lstat (const char *đường dẫn, struct stat *buf);
```

Mỗi hàm này trả về thông tin về một tệp. stat() trả về thông tin về tệp đư ợc biểu thị bằng đư ờng dẫn, path, trong khi fstat() trả về thông tin về tệp đư ợc biểu thị bằng mô tả tệp fd. lstat() giống hệt với stat( ), ngoại trừ trư ờng hợp liên kết tư ợng trư ng, lstat() trả về thông tin về chính liên kết đó chứ không phải tệp đích.

Mỗi hàm này lư u trữ thông tin trong một cấu trúc stat do ngư ời dùng cung cấp. Cấu trúc stat đư ợc định nghĩa trong <bits/stat.h>, đư ợc bao gồm trong <sys/stat.h>:

```
cấu trúc stat
        { dev_t st_dev;          /* ID của thiết bị chứa tệp */ ino_t st_ino; /*
        số inode */ mode_t st_mode; /* quyền */ nlink_t st_nlink; /
        * số liên kết cứng */ uid_t st_uid; /* ID ngư ời dùng
        của chủ sở hữu */ gid_t st_gid; /* ID nhóm của chủ sở hữu */ dev_t
        st_rdev; /* ID thiết bị (nếu là tệp đặc biệt) */ off_t st_size; /
        * tổng kích thư ớc tính bằng byte */ blksize_t st_blksize; /*
        kích thư ớc khối cho I/O hệ thống tệp */ blkcnt_t st_blocks; /* số khối đư ợc
        phân bổ */ time_t st_atime; /* thời gian truy cập gần nhất */
        time_t st_mtime; /* thời gian sửa đổi gần nhất */ time_t st_ctime; /* thời gian
        thay đổi trạng thái gần nhất */


};
```

Chi tiết hơ n, các trư ờng như sau:

- Trư ờng st_dev mô tả nút thiết bị mà tệp nằm trên đó (chúng ta sẽ đề cập đến các nút thiết bị sau trong chư ơ ng này). Nếu tệp không đư ợc hỗ trợ bởi thiết bị—ví dụ, nếu tệp nằm trên một gắn kết NFS—giá trị này là 0.
- Trư ờng st_ino cung cấp số inode của tệp. • Trư ờng st_mode cung cấp các byte chế độ của tệp. Chư ơ ng 1 và 2 đã đề cập đến các byte chế độ và quyền.
- Trư ờng st_nlink cung cấp số lư ợng liên kết cứng trỏ đến tệp. Mỗi tập tin có ít nhất một liên kết cứng.
- Trư ờng st_uid cung cấp ID ngư ời dùng của ngư ời dùng sở hữu tệp. • Trư ờng st_gid cung cấp ID nhóm của nhóm sở hữu tệp. • Nếu tệp là một nút thiết bị, trư ờng st_rdev mô tả thiết bị mà tệp này đại diện.
- Trư ờng st_size cung cấp kích thư ớc của tệp, tính bằng byte. • Trư ờng st_blksize mô tả kích thư ớc khối đư ợc ư u tiên cho I/O tệp hiệu quả. Giá trị này (hoặc bội số nguyên) là kích thư ớc khối tối ư u cho I/O đư ợc đệm bởi ngư ời dùng (xem Chư ơ ng 3).
- Trư ờng st_blocks cung cấp số khối hệ thống tệp đư ợc phân bổ cho tệp. Giá trị này sẽ nhỏ hơ n giá trị do st_size cung cấp nếu tệp có lỗ hổng (tức là nếu tệp là tệp thư a thớt).

- Trư ờng st_atime chứa thời gian truy cập tệp cuối cùng. Đây là thời gian gần đây nhất mà tệp đư ợc truy cập (ví dụ, bằng read() hoặc execle( )).
- Trư ờng st_mtime chứa thời gian sửa đổi tệp cuối cùng —tức là thời gian cuối cùng tập tin đã đư ợc ghi vào.
- Trư ờng st_ctime chứa thời gian thay đổi tệp cuối cùng. Điều này thư ờng bị hiểu nhầm là thời gian tạo tệp, không đư ợc lư u giữ trên Linux hoặc các hệ thống kiểu Unix khác. Trư ờng này thực sự mô tả thời gian cuối cùng siêu dữ liệu của tệp (ví dụ: chủ sở hữu hoặc quyền) đã thay đổi.

Khi thành công, cả ba lệnh gọi đều trả về 0 và lư u trữ siêu dữ liệu của tệp trong cấu trúc stat đư ợc cung cấp . Khi có lỗi, chúng trả về -1 và đặt errno thành một trong các giá trị sau:

EACCESS

Quá trình gọi không có quyền tìm kiếm cho một trong các thành phần thư mục của đư ờng dẫn (chỉ stat() và lstat()).

EBADF

fd không hợp lệ (chỉ fstat()).

MẶC ĐỊNH

path hoặc buf là con trỏ không hợp lệ.

Vòng lặp

đư ờng dẫn chứa quá nhiều liên kết tư ợng trư ng (chỉ stat() và lstat()).

ENAMETOOLONG

đư ờng dẫn quá dài (chỉ stat() và lstat()).

ENOENT

Không tồn tại thành phần nào trong đư ờng dẫn (chỉ có stat() và lstat()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đư ờng dẫn không phải là một thư mục (chỉ có stat() và lstat()).

Chư ơ ng trình sau sử dụng stat() để lấy kích thư ớc của tệp đư ợc cung cấp trên dòng lệnh:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

int chính (int argc, char *argv[]) {

        cấu trúc stat
        sb; int ret;

        nếu (argc < 2)
                { fprintf (stderr,
                        "cách sử dụng: %s <tệp>\n", argv[0]);
```

```
                trả về 1;
        }

        ret = stat(argv[1], &sb); if
        (ret)
              { perror ("stat"); trả
                về 1;
        }

        printf ("%s là %ld byte\n",
                argv[1], sb.st_size);

        trả về 0;
}
```

Sau đây là kết quả khi chạy chư ơ ng trình trên tệp nguồn riêng của nó:

```
$ ./stat stat.c
stat.c là 392 byte
```

Đoạn mã này, lần lư ợt, sử dụng fstat( ) để kiểm tra xem tệp đã mở có nằm trên thiết bị vật lý (khác với thiết bị mạng) hay không:

```
/*
 * is_on_physical_device - trả về giá trị dư ơ ng
 *   số nguyên nếu 'fd' nằm trên một thiết bị vật lý,
 * 0 nếu tệp nằm trên thiết bị phi vật lý hoặc * ảo (ví dụ:
 trên thiết bị gắn kết NFS) và * -1 nếu có lỗi. */


int is_on_physical_device (int fd) {

        cấu trúc stat
        sb; int ret;

        ret = fstat (fd, &sb);
        nếu (ret)
                { lỗi ("fstat");
                trả về -1;
        }

        trả về gnu_dev_major (sb.st_dev);
}
```

Quyền

Trong khi các lệnh gọi stat có thể đư ợc sử dụng để lấy các giá trị quyền cho một tệp nhất định thì hai lệnh gọi hệ thống khác sẽ thiết lập các giá trị đó:

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

int chmod (const char *path, chế độ mode_t); int
fchmod (int fd, chế độ mode_t);
```

Cả chmod() và fchmod() đều thiết lập quyền của tệp thành chế độ. Với chmod( ), path biểu thị tên đư ờng dẫn tư ơ ng đối hoặc tuyệt đối của tệp cần sửa đổi. Đối với fchmod( ), tệp đư ợc chỉ định bởi mô tả tệp fd.

Các giá trị hợp lệ cho mode, đư ợc biểu diễn bằng kiểu số nguyên mode_t mờ đục , giống với các giá trị đư ợc trả về bởi trư ờng st_mode trong cấu trúc stat . Mặc dù các giá trị là các số nguyên đơ n giản, như ng ý nghĩa của chúng lại cụ thể đối với từng triển khai Unix. Do đó, POSIX định nghĩa một tập hợp các hằng số biểu diễn các quyền khác nhau (xem “Quyền của các tệp mới” trong Chư ơ ng 2 để biết đầy đủ chi tiết). Các hằng số này có thể đư ợc kết hợp nhị phân OR với nhau để tạo thành các giá trị hợp lệ cho mode. Ví dụ: (S_IRUSR | S_IRGRP) đặt các quyền của tệp thành có thể đọc đư ợc bởi cả chủ sở hữu và nhóm.

Để thay đổi quyền của tệp, ID hiệu quả của tiến trình gọi chmod() hoặc fchmod() phải khớp với chủ sở hữu của tệp hoặc tiến trình phải có khả năng CAP_FOWNER .

Nếu thành công, cả hai lệnh gọi đều trả về 0. Nếu thất bại, cả hai lệnh gọi đều trả về -1 và đặt errno thành một trong các giá trị lỗi sau:

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho một thành phần trong đư ờng dẫn (chỉ chmod()).

EBADF

Mô tả tập tin fd không hợp lệ (chỉ fchmod()).

Đư ờng

dẫn EFAULT là một con trỏ không hợp lệ (chỉ chmod()).

EIO

Đã xảy ra lỗi I/O nội bộ trên hệ thống tập tin. Đây là lỗi rất nghiêm trọng; nó có thể chỉ ra đĩa hoặc hệ thống tập tin bị hỏng.

Vòng lặp

Hạt nhân gặp phải quá nhiều liên kết tư ợng trư ng khi giải quyết đư ờng dẫn (chỉ chmod()).

ENAMETOOLONG

đư ờng dẫn quá dài (chỉ chmod()).

Đư ờng

dẫn ENOENT không tồn tại (chỉ chmod()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đư ờng dẫn không phải là một thư mục (chỉ chmod()).

EPERM

ID hiệu quả của quy trình gọi không khớp với chủ sở hữu của tệp và quy trình này thiếu khả năng CAP_FOWNER .

EROFS

Tệp này nằm trên hệ thống tệp chỉ đọc.

Đoạn mã này thiết lập tệp map.png thành quyền có thể đọc và ghi của chủ sở hữu:

```
int ret;

/*
 * Đặt 'map.png' trong thư mục hiện tại thành * có thể đọc và
 ghi được bởi chủ sở hữu. Điều này giống như 'chmod 600 ./
 *
    map.png'. */

ret = chmod ("./map.png", S_IRUSR | S_IWUSR); if
(ret)
      perror ("chmod");
```

Đoạn mã này thực hiện điều tương tự, giả sử rằng fd biểu thị tệp mở map.png:

```
int ret;

/*
 * Đặt tệp đằng sau 'fd' thành quyền có thể đọc được và ghi được
 của chủ sở hữu. */

ret = fchmod (fd, S_IRUSR | S_IWUSR);
if (ret)
         perror ("fchmod");
```

Cả chmod() và fchmod() đều có sẵn trên tất cả các hệ thống Unix hiện đại. POSIX yêu cầu hàm trước và biến hàm sau thành tùy chọn.

Quyền sở hữu

Trong cấu trúc stat , các trường st_uid và st_gid cung cấp chủ sở hữu và nhóm của tệp tương ứng. Ba lệnh gọi hệ thống cho phép người dùng thay đổi hai giá trị đó:

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

int chown (const char *path, uid_t chủ sở hữu, gid_t nhóm);
int lchown (const char *path, uid_t chủ sở hữu, gid_t nhóm);
int fchown (int fd, uid_t chủ sở hữu, gid_t nhóm);
```

chown( ) và lchown( ) thiết lập quyền sở hữu của tệp được chỉ định bởi đường dẫn path. Chúng có cùng tác dụng, trừ khi tệp là liên kết tượng trưng: chown( ) theo sau các liên kết tượng trưng và thay đổi quyền sở hữu của mục tiêu liên kết thay vì chính liên kết đó, trong khi

lchown() không theo các liên kết tượng trưng và do đó thay đổi quyền sở hữu của tệp liên kết tượng trưng. fchown() thiết lập quyền sở hữu của tệp được biểu thị bằng mô tả tệp fd.

Khi thành công, cả ba lệnh gọi đều đặt chủ sở hữu tệp thành owner, đặt nhóm tệp thành group và trả về 0. Nếu trường owner hoặc group là -1, giá trị đó không được đặt. Chỉ có một quy trình có khả năng CAP_CHOWN (thường là quy trình gốc) mới có thể thay đổi chủ sở hữu của tệp. Chủ sở hữu của tệp có thể thay đổi nhóm tệp thành bất kỳ nhóm nào mà người dùng là thành viên; các quy trình có CAP_CHOWN có thể thay đổi nhóm tệp thành bất kỳ giá trị nào.

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

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho một thành phần trong đường dẫn (chỉ chown() và lchown()).

EBADF

fd không hợp lệ (chỉ fchown()).

Đường

dẫn EFAULT không hợp lệ (chỉ áp dụng chown() và lchown()).

EIO

Có lỗi I/O nội bộ (lỗi này không tốt).

ELOOP

Hạt nhân gặp phải quá nhiều liên kết tượng trưng khi giải quyết đường dẫn (chỉ chown() và lchown()).

ENAMETOOLONG

đường dẫn quá dài (chỉ có chown() và lchown()).

ENOENT

Tập tin không tồn tại.

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đường dẫn không phải là một thư mục (chỉ có chown() và lchown()).

EPERM

Quá trình triệu hồi thiếu các quyền cần thiết để thay đổi chủ sở hữu hoặc nhóm theo yêu cầu.

EROFS

Hệ thống tập tin chỉ có thể đọc.

Đoạn mã này thay đổi nhóm của tệp manifest.txt trong thư mục làm việc hiện tại thành officials. Để thành công, người dùng gọi phải sở hữu khả năng CAP_CHOWN hoặc phải là kidd và trong nhóm officials :

```
cấu trúc nhóm *gr;
int ret;
```

```
/*
 * getgrnam( ) trả về thông tin về một nhóm
 *
   được đặt tên như vậy.
 */
gr = getgrnam ("sĩ quan"); nếu
(!gr) { /*
          có khả năng là nhóm không hợp lệ
          */ perror ("getgrnam");
          trả về 1;
}

/* đặt nhóm của manifest.txt thành 'officers'
*/ ret = chmod ("manifest.txt", -1, gr->gr_gid);
nếu
    (ret) perror ("chmod");
```

Trước khi gọi, nhóm của tệp là crew:

```
$ ls -l
-rw-r--r-- 1 kidd crew 13274 Ngày 23 tháng 5 lúc 09:20
```

manifest.txt Sau khi gọi, tệp này chỉ dành riêng cho các sĩ quan:

```
$ ls -l
-rw-r--r-- 1 kidd officials 13274 23 tháng 5 09:20 manifest.txt
```

Chủ sở hữu tệp, kidd, không bị thay đổi vì đoạn mã đã truyền -1 cho uid.

Hàm này thiết lập tệp được biểu thị bởi fd thành quyền sở hữu và nhóm gốc:

```
/*
 * make_root_owner - thay đổi chủ sở hữu và nhóm của tệp được chỉ định bởi 'fd' thành
 *
   root. Trả về 0 nếu thành công và -1 nếu * thất bại. */


int make_root_owner (int fd) {

          int ret;

          /* 0 vừa là gid vừa là uid của root */ ret =
          fchown (fd, 0, 0); if
          (ret)
                perror ("fchown");

          trả về ret;
}
```

Quá trình gọi phải có khả năng CAP_CHOWN . Như thường lệ với các khả năng, điều này thường có nghĩa là nó phải do root sở hữu.

Thuộc tính mở rộng

Thuộc tính mở rộng, còn được gọi là xattrs, cung cấp cơ chế liên kết vĩnh viễn các cặp khóa/giá trị với các tệp. Trong chương này, chúng ta đã thảo luận về tất cả các loại siêu dữ liệu khóa/giá trị liên quan đến tệp: kích thước tệp, chủ sở hữu, lần sửa đổi cuối cùng

dấu thời gian, v.v. Các thuộc tính mở rộng cho phép các hệ thống tệp hiện có hỗ trợ các tính năng mới không được dự đoán trong thiết kế ban đầu của chúng, chẳng hạn như kiểm soát truy cập bắt buộc để bảo mật. Điều làm cho các thuộc tính mở rộng trở nên thú vị là các ứng dụng không gian người dùng có thể tùy ý tạo, đọc và ghi vào các cặp khóa/giá trị.

Thuộc tính mở rộng không phụ thuộc vào hệ thống tập tin, theo nghĩa là các ứng dụng sử dụng giao diện chuẩn để thao tác chúng; giao diện không dành riêng cho bất kỳ hệ thống tập tin nào.
Do đó, các ứng dụng có thể sử dụng các thuộc tính mở rộng mà không cần quan tâm đến hệ thống tệp mà các tệp nằm trên đó hoặc cách hệ thống tệp lưu trữ khóa và giá trị bên trong. Tuy nhiên, việc triển khai các thuộc tính mở rộng rất cụ thể theo từng hệ thống tệp. Các hệ thống tệp khác nhau lưu trữ các thuộc tính mở rộng theo những cách khá khác nhau, nhưng hạt nhân ẩn những khác biệt này, trừu tượng hóa chúng đằng sau giao diện thuộc tính mở rộng.

Ví dụ, hệ thống tệp ext3 lưu trữ các thuộc tính mở rộng của tệp trong không gian trống trong inode của tệp.* Tính năng này giúp đọc các thuộc tính mở rộng rất nhanh. Vì khối hệ thống tệp chứa inode được đọc từ đĩa và vào bộ nhớ bất cứ khi nào ứng dụng truy cập tệp, các thuộc tính mở rộng được "tự động" đọc vào bộ nhớ và có thể được truy cập mà không cần bất kỳ chi phí bổ sung nào.

Các hệ thống tệp khác, chẳng hạn như FAT và minixfs, không hỗ trợ thuộc tính mở rộng. Các hệ thống tệp này trả về ENOTSUP khi các hoạt động thuộc tính mở rộng được gọi trên các tệp của chúng.

Khóa và giá trị

Một khóa duy nhất xác định từng thuộc tính mở rộng. Khóa phải là UTF-8 hợp lệ. Chúng có dạng namespace.attribute. Mỗi khóa phải đủ điều kiện; nghĩa là, nó phải bắt đầu bằng một không gian tên hợp lệ, theo sau là dấu chấm. Một ví dụ về tên khóa hợp lệ là user.mime_type; khóa này nằm trong không gian tên người dùng với tên thuộc tính mime_type.

Một khóa có thể được xác định hoặc không xác định. Nếu một khóa được xác định, giá trị của nó có thể rỗng hoặc không rỗng. Nghĩa là, có sự khác biệt giữa một khóa không xác định và một khóa được xác định không có giá trị được gán. Như chúng ta sẽ thấy, điều này có nghĩa là cần có một giao diện đặc biệt để xóa các khóa (gán cho chúng một giá trị rỗng là không đủ).

Giá trị liên kết với khóa, nếu không rỗng, có thể là bất kỳ mảng byte tùy ý nào.
Vì giá trị không nhất thiết phải là một chuỗi, nên nó không cần phải được kết thúc bằng null, mặc dù việc kết thúc bằng null chắc chắn có ý nghĩa nếu bạn chọn lưu trữ một chuỗi C làm giá trị của khóa. Vì các giá trị không được đảm bảo là được kết thúc bằng null, nên tất cả các thao tác trên các thuộc tính mở rộng đều yêu cầu kích thước của giá trị. Khi đọc một thuộc tính, kernel cung cấp kích thước; khi viết một thuộc tính, bạn phải cung cấp kích thước.

* Cho đến khi inode hết dung lượng, tất nhiên rồi. Sau đó ext3 lưu trữ các thuộc tính mở rộng trong hệ thống tập tin bổ sung khối. Các phiên bản cũ hơn của ext3 thiếu tính năng thuộc tính mở rộng trong inode này.

Một cách tốt hơn để lưu trữ các loại MIME

Trình quản lý tệp GUI, chẳng hạn như Nautilus của GNOME, hoạt động khác nhau đối với các tệp có kích thước khác nhau
các loại: chúng cung cấp các biểu tượng độc đáo, hành vi nhấp chuột mặc định khác nhau, danh sách đặc biệt các
hoạt động cần thực hiện, v.v. Để thực hiện điều này, trình quản lý tệp phải biết
định dạng của mỗi tệp. Để xác định định dạng, các hệ thống tệp như Windows chỉ cần xem
tại phần mở rộng của tệp. Tuy nhiên, vì lý do truyền thống và bảo mật, các hệ thống Unix có xu hướng kiểm tra
tệp và diễn giải loại tệp. Quá trình này được gọi là loại MIME
hít hà.

Một số trình quản lý tệp tạo thông tin này ngay lập tức; những trình quản lý khác tạo thông tin một lần rồi lưu
vào bộ nhớ đệm. Những trình quản lý lưu vào bộ nhớ đệm thông tin có xu hướng đặt thông tin vào một tệp tùy chỉnh
cơ sở dữ liệu. Trình quản lý tệp phải hoạt động để giữ cho cơ sở dữ liệu này đồng bộ với các tệp,
có thể thay đổi mà không cần người quản lý tệp biết. Một cách tiếp cận tốt hơn là loại bỏ cơ sở dữ
liệu tùy chỉnh và lưu trữ siêu dữ liệu như vậy trong các thuộc tính mở rộng: đây là
dễ bảo trì hơn, truy cập nhanh hơn và dễ dàng truy cập bằng bất kỳ ứng dụng nào.

Linux không áp dụng bất kỳ giới hạn nào về số lượng khóa, độ dài của khóa, kích thước
của một giá trị hoặc tổng không gian có thể được sử dụng bởi tất cả các khóa và giá trị liên kết với một
tệp. Tuy nhiên, các hệ thống tệp có giới hạn thực tế. Những giới hạn này thường là
được thể hiện dưới dạng các ràng buộc về tổng kích thước của tất cả các khóa và các giá trị liên quan
với một tập tin nhất định.

Ví dụ, với ext3 , tất cả các thuộc tính mở rộng cho một tệp nhất định phải phù hợp với
không gian trống trong inode của tệp và tối đa một khối hệ thống tệp bổ sung. (Các phiên bản cũ hơn của
ext3 bị giới hạn ở một khối hệ thống tệp, không có bộ nhớ trong inode.)
Điều này tương đương với giới hạn thực tế khoảng 1 KB đến 8 KB cho mỗi tệp, tùy thuộc vào kích thước
của các khối hệ thống tập tin. Ngược lại, XFS không có giới hạn thực tế. Ngay cả với ext3,
tuy nhiên, những giới hạn này thường không phải là vấn đề, vì hầu hết các khóa và giá trị đều là văn bản ngắn
chuỗi. Tuy nhiên, hãy ghi nhớ chúng—hãy suy nghĩ kỹ trước khi lưu trữ toàn bộ lịch sử kiểm soát phiên bản
của một dự án trong các thuộc tính mở rộng của tệp!

Không gian tên thuộc tính mở rộng

Các không gian tên liên quan đến các thuộc tính mở rộng không chỉ là các công cụ tổ chức. Hạt nhân thực thi
các chính sách truy cập khác nhau tùy thuộc vào
không gian tên.

Linux hiện định nghĩa bốn không gian tên thuộc tính mở rộng và có thể định nghĩa thêm trong
tương lai. Bốn hiện tại như sau:

hệ thống
: Không gian tên hệ thống được sử dụng để triển khai các tính năng hạt nhân sử dụng mở rộng
thuộc tính, chẳng hạn như danh sách kiểm soát truy cập (ACL). Một ví dụ về một

thuộc tính trong không gian tên này là system.posix_acl_access. Việc ngư ời dùng có thể đọc hoặc ghi vào các thuộc tính này hay không phụ thuộc vào mô-đun bảo mật tại chỗ.

Giả sử trong trư ờng hợp tệ nhất không có ngư ời dùng nào (kể cả root) có thể đọc đư ợc các thuộc tính này.

bảo mật Không gian tên bảo mật đư ợc sử dụng để triển khai các mô-đun bảo mật, chẳng hạn như SELinux. Việc các ứng dụng không gian ngư ời dùng có thể truy cập các thuộc tính này hay không phụ thuộc vào mô-đun bảo mật tại chỗ. Theo mặc định, tất cả các quy trình có thể đọc các thuộc tính này, như ng chỉ các quy trình có khả năng CAP_SYS_ADMIN mới có thể ghi vào chúng.

tin cậy

Không gian tên tin cậy lư u trữ thông tin bị hạn chế trong không gian ngư ời dùng. Chỉ các quy trình có khả năng CAP_SYS_ADMIN mới có thể đọc hoặc ghi vào các thuộc tính này.

ngư ời sử dụng

Không gian tên ngư ời dùng là không gian tên chuẩn để các quy trình thông thư ờng sử dụng. Nhân kiểm soát quyền truy cập vào không gian tên này thông qua các bit quyền tệp thông thư ờng. Để đọc giá trị từ khóa hiện có, một quy trình phải có quyền truy cập đọc vào tệp đã cho. Để tạo khóa mới hoặc ghi giá trị vào khóa hiện có, một quy trình phải có quyền truy cập ghi vào tệp đã cho. Bạn chỉ có thể gán các thuộc tính mở rộng trong không gian tên ngư ời dùng cho các tệp thông thư ờng, không phải cho các liên kết tư ợng trư ng hoặc tệp thiết bị. Khi thiết kế ứng dụng không gian ngư ời dùng sử dụng các thuộc tính mở rộng, đây có thể là không gian tên bạn muốn.

Hoạt động thuộc tính mở rộng

POSIX định nghĩa bốn thao tác mà ứng dụng có thể thực hiện trên các thuộc tính mở rộng của một tệp nhất định:

- Cho một tệp và một khóa, trả về giá trị tư ơ ng ứng.
- Cho một tệp, một khóa và một giá trị, gán giá trị đó cho khóa.
- Cho một tệp, trả về danh sách tất cả các khóa thuộc tính mở rộng đư ợc gán cho tệp.
- Cho một tệp và một khóa, xóa thuộc tính mở rộng đó khỏi tệp.

Đối với mỗi hoạt động, POSIX cung cấp ba lệnh gọi hệ thống:

- Một phiên bản hoạt động trên một tên đư ờng dẫn nhất định; nếu đư ờng dẫn tham chiếu đến một biểu tư ợng liên kết, mục tiêu của liên kết đư ợc vận hành (hành vi thông thư ờng).
- Phiên bản hoạt động trên một đư ờng dẫn tên nhất định; nếu đư ờng dẫn tham chiếu đến một liên kết tư ợng trư ng, thì chính liên kết đó sẽ đư ợc vận hành (biến thể l chuẩn của lệnh gọi hệ thống).
- Phiên bản hoạt động trên trình mô tả tệp (biến thể f chuẩn).

Trong các tiểu mục sau, chúng ta sẽ đề cập đến tất cả 12 hoán vị.

Truy xuất một thuộc tính mở rộng. Hoạt động đơ n giản nhất là trả về giá trị của một thuộc tính mở rộng từ một tệp, đư ợc cung cấp khóa:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t getxattr (const char *đường dẫn, const char *khóa,
                  void *giá trị, size_t kích thước);
ssize_t lgetxattr (const char *đường dẫn, const char *khóa,
                   void *giá trị, size_t kích
thước); ssize_t fgetxattr (int fd, const char
                   *khóa, void *giá trị, size_t kích thước);
```

Một lệnh gọi thành công đến getxattr( ) lưu trữ thuộc tính mở rộng với khóa name từ đường dẫn tệp trong giá trị bộ đệm được cung cấp , có độ dài là size byte. Nó trả về kích thước thực tế của giá trị.

Nếu size là 0, lệnh gọi sẽ trả về size của giá trị mà không lưu trữ nó trong value. Do đó, việc truyền 0 cho phép các ứng dụng xác định đúng size cho buffer để lưu trữ giá trị của key. Với size này, các ứng dụng có thể phân bổ hoặc thay đổi kích thước buffer khi cần.

lgetxattr() hoạt động giống như getxattr( ), trừ khi path là liên kết tượng trưng, trong trường hợp đó, nó trả về các thuộc tính mở rộng từ chính liên kết đó chứ không phải từ mục tiêu của liên kết. Nhớ lại phần trước rằng các thuộc tính trong không gian tên người dùng không thể được áp dụng cho các liên kết tượng trưng—do đó, lệnh gọi này hiếm khi được

sử dụng. fgetxattr() hoạt động trên mô tả tệp fd; nếu không, nó hoạt động giống như getxattr( ).

Khi có lỗi, cả ba lệnh gọi đều trả về -1 và đặt errno thành một trong các giá trị sau:

EACCESS

Quá trình gọi không có quyền tìm kiếm cho một trong các thành phần thư mục của đường dẫn (chỉ getxattr( ) và lgetxattr( )).

EBADF

fd không hợp lệ (chỉ fgetxattr()).

MẶC ĐỊNH

đường dẫn, khóa hoặc giá trị là con trỏ không hợp lệ.

Vòng lặp

đường dẫn chứa quá nhiều liên kết tượng trưng (chỉ getxattr() và lgetxattr()).

ENAMETOOLONG

đường dẫn quá dài (chỉ getxattr() và lgetxattr()).

ENOATTR

Khóa thuộc tính không tồn tại hoặc quy trình không có quyền truy cập vào thuộc tính.

ENOENT

Một thành phần trong đường dẫn không tồn tại (chỉ có getxattr() và lgetxattr()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đường dẫn không phải là một thư mục (chỉ getxattr() và lgetxattr()).

ENOTSUP

Hệ thống tập tin mà đường dẫn hoặc fd lưu trú không hỗ trợ các thuộc tính mở rộng.

Kích

thước ERANGE quá nhỏ để chứa giá trị của khóa. Như đã thảo luận trước đó, lệnh gọi có thể được phát hành lại với kích thước được đặt thành 0; giá trị trả về sẽ chỉ ra kích thước bộ đệm cần thiết và giá trị có thể được thay đổi kích thước phù hợp.

Thiết lập một thuộc tính mở rộng. Ba lệnh gọi hệ thống sau đây thiết lập một thuộc tính mở rộng nhất định:

```
#include <sys/types.h>
#include <attr/xattr.h>

int setxattr (const char *path, const char *key,
              const void *value, size_t kích thước, int
cờ); int lsetxattr (const char *path, const char
               *key, const void *value, size_t kích thước, int cờ);
int fsetxattr (int fd, const char *key,
               const void *value, size_t kích thước, int cờ);
```

Một lệnh gọi thành công đến setxattr( ) sẽ đặt khóa thuộc tính mở rộng trên đường dẫn tệp thành giá trị, có độ dài tính bằng byte. Trường cờ sẽ sửa đổi hành vi của lệnh gọi.
Nếu flags là XATTR_CREATE, lệnh gọi sẽ không thành công nếu thuộc tính mở rộng đã tồn tại. Nếu flags là XATTR_REPLACE, lệnh gọi sẽ không thành công nếu thuộc tính mở rộng không tồn tại. Hành vi mặc định—được thực hiện nếu flags là 0—cho phép cả tạo và thay thế.
Bất kể giá trị của cờ, các khóa khác ngoài khóa đều không bị ảnh hưởng.

lsetxattr() hoạt động giống như setxattr( ), trừ khi path là liên kết tượng trưng, trong trường hợp đó, nó đặt các thuộc tính mở rộng trên chính liên kết, thay vì trên mục tiêu của liên kết. Hãy nhớ rằng các thuộc tính trong không gian tên người dùng không thể được áp dụng cho các liên kết tượng trưng—do đó, lệnh gọi này cũng hiếm khi được sử dụng.

fsetxattr() hoạt động trên mô tả tệp fd; nếu không, nó hoạt động giống như setxattr( ).

Nếu thành công, cả ba lệnh gọi hệ thống đều trả về 0; nếu thất bại, các lệnh gọi trả về -1 và đặt errno thành một trong những giá trị sau:

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho một trong các thành phần thư mục của đường dẫn (chỉ setxattr() và lsetxattr()).

EBADF

fd không hợp lệ (chỉ fsetxattr()).

EDQUOT

Giới hạn hạn ngạch ngăn chặn việc tiêu thụ không gian cần thiết cho hoạt động được yêu cầu.

EEXIST

XATTR_CREATE đã được thiết lập trong cờ và khóa đã tồn tại trong tệp đã cho.

Đường dẫn, khóa hoặc giá trị EFAULT là con trỏ không hợp lệ.

Cờ

EINVAL không hợp lệ.

Đường dẫn ELOOP chứa quá nhiều liên kết tượng trưng (chỉ setxattr() và lsetxattr()).

ENAMETOOLONG

đường dẫn quá dài (chỉ setxattr() và lsetxattr()).

ENOATTR

XATTR_REPLACE đã được đặt trong cờ và khóa không tồn tại trong tệp đã cho.

ENOENT

Một thành phần trong đường dẫn không tồn tại (chỉ có setxattr() và lsetxattr()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOSPC

Không có đủ dung lượng trên hệ thống tập tin để lưu trữ thuộc tính mở rộng.

ENOTDIR

Một thành phần trong đường dẫn không phải là một thư mục (chỉ setxattr() và lsetxattr()).

ENOTSUP

Hệ thống tập tin mà đường dẫn hoặc fd lưu trú không hỗ trợ các thuộc tính mở rộng.

Liệt kê các thuộc tính mở rộng trên một tệp. Ba lệnh gọi hệ thống sau đây liệt kê tập hợp các khóa thuộc tính mở rộng được gán cho một tệp nhất định:

```
#include <sys/types.h>
#include <attr/xattr.h>

ssize_t listxattr (const char *path, char
                   *list, size_t kích thước);
ssize_t llistxattr (const char *path, char
                    *list, size_t kích thước);
ssize_t flistxattr (int fd,
                    char *list, size_t kích thước);
```

Một cuộc gọi thành công đến listxattr() sẽ trả về danh sách các khóa thuộc tính mở rộng được liên kết với tệp được biểu thị bằng path. Danh sách được lưu trữ trong bộ đệm do list cung cấp, có kích thước là byte. Cuộc gọi hệ thống trả về kích thước thực tế của danh sách, tính bằng byte.

Mỗi khóa thuộc tính mở rộng được trả về trong danh sách được kết thúc bằng ký tự null, do đó danh sách có thể trông như thế này:

```
"user.md5_sum\0user.mime_type\0system.posix_acl_default\0"
```

Do đó, mặc dù mỗi khóa là một chuỗi C truyền thống, kết thúc bằng null, bạn cần độ dài của toàn bộ danh sách (mà bạn có thể lấy từ giá trị trả về của lệnh gọi) để duyệt danh sách các khóa. Để tìm ra bộ đệm cần phân bổ lớn đến mức nào, hãy gọi một trong các hàm danh sách có kích thư ớc là 0; điều này khiến hàm trả về độ dài thực tế của toàn bộ danh sách các khóa. Giống như với getxattr( ), các ứng dụng có thể sử dụng chức năng này để phân bổ hoặc thay đổi kích thư ớc bộ đệm để truyền giá trị . llistxattr() hoạt động giống như listxattr( ), trừ khi path

là một liên kết tư ợng trư ng, trong trư ờng hợp đó, lệnh gọi sẽ liệt kê các khóa thuộc tính mở rộng đư ợc liên kết với chính liên kết đó chứ không phải với mục tiêu của liên kết. Hãy nhớ rằng các thuộc tính trong không gian tên ngư ời dùng không thể đư ợc áp dụng cho các liên kết tư ợng trư ng—do đó, lệnh gọi này hiếm khi đư ợc sử dụng. flistxattr() hoạt động trên mô tả tệp fd; nếu không, nó hoạt

động giống như listxattr( ).

Khi lỗi xảy ra, cả ba lệnh gọi đều trả về -1 và đặt errno thành một trong các mã lỗi sau:

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho một trong các thành phần thư mục của đư ờng dẫn (chỉ listxattr() và llistxattr()).

EBADF

fd không hợp lệ (chỉ flistxattr()).

Đư ờng

dẫn hoặc danh sách EFAULT là con trỏ không hợp lệ.

Đư ờng

dẫn ELOOP chứa quá nhiều liên kết tư ợng trư ng (chỉ listxattr() và llistxattr()).

ENAMETOOLONG

đư ờng dẫn quá dài (chỉ listxattr() và llistxattr()).

ENOENT

Một thành phần trong đư ờng dẫn không tồn tại (chỉ có listxattr() và llistxattr()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đư ờng dẫn không phải là một thư mục (chỉ listxattr() và llistxattr()).

ENOTSUPP Hệ

thống tập tin mà đư ờng dẫn hoặc fd lư u trú không hỗ trợ các thuộc tính mở rộng.

Kích

thư ớc ERANGE khác không và không đủ lớn để chứa toàn bộ danh sách khóa. Ứng dụng có thể phát hành lại lệnh gọi với kích thư ớc đư ợc đặt thành 0 để khám phá kích thư ớc thực tế của danh sách. Sau đó, chư ơ ng trình có thể thay đổi kích thư ớc giá trị và phát hành lại lệnh gọi hệ thống.

Xóa một thuộc tính mở rộng. Cuối cùng, ba lệnh gọi hệ thống này xóa một khóa nhất định khỏi một tệp nhất định:

```
#include <sys/types.h>
#include <attr/xattr.h>

int removexattr (const char *đường dẫn, const char
*khóa); int lremovexattr (const char *đường dẫn, const
char *khóa); int fremovexattr (int fd, const char *khóa);
```

Một lệnh gọi thành công đến removexattr() sẽ xóa khóa thuộc tính mở rộng khỏi đường dẫn tệp . Hãy nhớ rằng có sự khác biệt giữa khóa không xác định và khóa được xác định có giá trị rỗng (độ dài bằng không). lremovexattr() hoạt động giống như

removexattr( ), trừ khi path là liên kết tượng trưng, trong trường hợp đó, lệnh gọi sẽ xóa khóa thuộc tính mở rộng được liên kết với chính liên kết đó chứ không phải với mục tiêu của liên kết. Hãy nhớ rằng các thuộc tính trong không gian tên người dùng không thể được áp dụng cho các liên kết tượng trưng—do đó, lệnh gọi này cũng hiếm khi được sử dụng. fremovexattr() hoạt động trên mô tả tệp

fd; nếu không, nó hoạt động giống như removexattr( ).

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

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho một trong các thành phần thư mục của đường dẫn (chỉ removexattr() và lremovexattr()).

EBADF

fd không hợp lệ (chỉ fremovexattr()).

Đường

dẫn hoặc khóa EFAULT là con trỏ không hợp lệ.

Đường

dẫn ELOOP chứa quá nhiều liên kết tượng trưng (chỉ có removexattr() và lremovexattr()).

ENAMETOOLONG

đường dẫn quá dài (chỉ có removeexattr() và lremovexattr()).

ENOATTR

khóa không tồn tại trong tập tin đã cho.

ENOENT

Không tồn tại thành phần nào trong đường dẫn (chỉ có removeexattr() và lremovexattr()).

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một thành phần trong đường dẫn không phải là một thư mục (chỉ có removexattr() và lremovexattr()).

ENOTSUPP Hệ

thống tập tin mà đường dẫn hoặc fd lưu trú không hỗ trợ các thuộc tính mở rộng.

Thư mục

Trong Unix, thư mục là một khái niệm đơn giản: nó chứa một danh sách các tên tệp, mỗi tên ánh xạ tới một số inode. Mỗi tên được gọi là mục nhập thư mục và mỗi ánh xạ tên-đến-inode được gọi là liên kết. Nội dung của thư mục—những gì người dùng nhìn thấy là kết quả của lệnh ls—là danh sách tất cả các tên tệp trong thư mục đó. Khi người dùng mở một tệp trong một thư mục nhất định, hạt nhân sẽ tra cứu tên tệp trong danh sách của thư mục đó để tìm số inode tương ứng. Sau đó, hạt nhân sẽ chuyển số inode đó tới hệ thống tệp, hệ thống tệp sẽ sử dụng số này để tìm vị trí vật lý của tệp trên thiết bị.

Thư mục cũng có thể chứa các thư mục khác. Thư mục con là thư mục bên trong thư mục khác. Theo định nghĩa này, tất cả các thư mục đều là thư mục con của một số thư mục cha, ngoại trừ thư mục ở gốc của cây hệ thống tập tin, /. Không có gì ngạc nhiên khi thư mục này được gọi là thư mục gốc (không nên nhầm lẫn với thư mục gốc của root, /root).

Đường dẫn bao gồm tên tệp cùng với một hoặc nhiều thư mục cha của tệp đó.
Đường dẫn tuyệt đối là đường dẫn bắt đầu bằng thư mục gốc—ví dụ: /usr/bin/sextant.
Đường dẫn tương đối là đường dẫn không bắt đầu bằng thư mục gốc, chẳng hạn như bin/sextant. Để đường dẫn như vậy hữu ích, hệ điều hành phải biết thư mục mà đường dẫn tương đối. Thư mục làm việc hiện tại (được thảo luận trong phần tiếp theo) được sử dụng làm điểm bắt đầu.

Tên tệp và thư mục có thể chứa bất kỳ ký tự nào ngoại trừ /, ký tự này phân định các thư mục trong tên đường dẫn và null, ký tự này kết thúc tên đường dẫn. Tuy nhiên, thông lệ chuẩn là giới hạn các ký tự trong tên đường dẫn thành các ký tự có thể in hợp lệ theo ngôn ngữ hiện tại hoặc thậm chí chỉ ASCII. Tuy nhiên, vì cả hạt nhân và thư viện C đều không áp dụng thông lệ này nên các ứng dụng phải áp dụng chỉ sử dụng các ký tự có thể in hợp lệ.

Các hệ thống Unix cũ giới hạn tên tệp ở mức 14 ký tự. Ngày nay, tất cả các hệ thống tệp Unix hiện đại đều cho phép ít nhất 255 byte cho mỗi tên tệp.* Nhiều hệ thống tệp trong Linux thậm chí còn cho phép

tên tệp dài hơn.† Mỗi thư mục chứa hai thư mục đặc biệt, . và .. (gọi là dot và dot-dot). Thư mục dot là tham chiếu đến chính thư mục đó. Thư mục dot-dot là tham chiếu đến thư mục cha của thư mục đó. Ví dụ: /home/kidd/gold/.. là cùng một thư mục với /home/kidd. Các thư mục dot và dot-dot của thư mục gốc trỏ đến chính nó—tức là /, /., và /.. đều là cùng một thư mục. Do đó, về mặt kỹ thuật, người ta có thể nói rằng ngay cả thư mục gốc cũng là một thư mục con—trong trường hợp này, là của chính nó.

* Lưu ý rằng giới hạn này là 255 byte, không phải 255 ký tự. Rõ ràng là các ký tự nhiều byte chiếm nhiều hơn 1 trong 255 byte này.

† Tất nhiên, các hệ thống tập tin cũ hơn mà Linux cung cấp để tương thích ngược, chẳng hạn như FAT, vẫn mang những hạn chế riêng của chúng. Trong trường hợp FAT, hạn chế này là tám ký tự, theo sau là một dấu chấm, theo sau là ba ký tự. Đúng vậy, việc áp dụng dấu chấm như một ký tự đặc biệt bên trong hệ thống tập tin là ngớ ngẩn.

Thư mục làm việc hiện tại Mỗi tiến trình đều có

một thư mục hiện tại, ban đầu nó kế thừa từ tiến trình cha của nó. Thư mục đó được gọi là thư mục làm việc hiện tại của tiến trình (cwd). Thư mục làm việc hiện tại là điểm bắt đầu mà hạt nhân giải quyết các tên đường dẫn tương đối. Ví dụ, nếu thư mục làm việc hiện tại của một tiến trình là /home/blackbeard và tiến trình đó cố gắng mở parrot.jpg, hạt nhân sẽ cố gắng mở /home/blackbeard/parrot.jpg. Ngược lại, nếu tiến trình cố gắng mở /usr/bin/mast, hạt nhân thực sự sẽ mở /usr/bin/mast— thư mục làm việc hiện tại không ảnh hưởng đến các tên đường dẫn tuyệt đối (tức là các tên đường dẫn bắt đầu bằng dấu gạch chéo).

Một tiến trình có thể lấy và thay đổi thư mục làm việc hiện tại của nó.

Lấy thư mục làm việc hiện tại

Phương pháp được ưa thích để lấy thư mục làm việc hiện tại là lệnh gọi hệ thống getcwd() , được POSIX chuẩn hóa:

```
#include <unistd.h>

ký tự * getcwd (char *buf, size_t kích thước);
```

Một lệnh gọi thành công đến getcwd( ) sao chép thư mục làm việc hiện tại dưới dạng một đường dẫn tuyệt đối vào bộ đệm được trỏ đến bởi buf, có độ dài là byte , và trả về một con trỏ đến buf. Khi lệnh gọi không thành công, lệnh gọi trả về NULL và đặt errno thành một trong các giá trị sau:

MẶC ĐỊNH
: buf là con trỏ không hợp lệ.

EINVAL
: size là 0, nhưng buf không phải là NULL.

ENOENT
: Thư mục làm việc hiện tại không còn hợp lệ nữa. Điều này có thể xảy ra nếu thư mục làm việc hiện tại bị xóa.

Kích
: thước ERANGE quá nhỏ để chứa thư mục làm việc hiện tại trong buf. Ứng dụng cần phân bổ bộ đệm lớn hơn và thử lại.

Sau đây là một ví dụ về việc sử dụng getcwd( ):

```
ký tự cwd[BUF_LEN];

nếu (!getcwd (cwd, BUF_LEN)) { lỗi
          ("getcwd"); thoát
          (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

POSIX quy định rằng hành vi của getcwd() là không xác định nếu buf là NULL. Trong trư ờng hợp này, thư viện C của Linux sẽ phân bổ một bộ đệm có độ dài là byte và lư u trữ thư mục làm việc hiện tại ở đó. Nếu size là 0, thư viện C sẽ phân bổ một bộ đệm đủ lớn để lư u trữ thư mục làm việc hiện tại. Sau đó, ứng dụng có trách nhiệm giải phóng bộ đệm, thông qua free( ), khi đã giải phóng xong. Vì hành vi này là dành riêng cho Linux, nên các ứng dụng coi trọng tính di động hoặc tuân thủ nghiêm ngặt POSIX không nên dựa vào chức năng này. Tuy nhiên, tính năng này giúp việc sử dụng trở nên rất đơ n giản!

Sau đây là một ví dụ:

```
ký tự *cwd;

cwd = getcwd (NULL, 0);
nếu (!cwd)
         { lỗi ("getcwd");
         thoát (EXIT_FAILURE);
}

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

miễn phí (cwd);
```

Thư viện C của Linux cũng cung cấp hàm get_current_dir_name() , có cùng hành vi như getcwd() khi đư ợc truyền một bộ đệm NULL và kích thư ớc bằng 0:

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

ký tự * lấy_tên_thư _mục_hiện_tại (void);
```

Vì vậy, đoạn mã này hoạt động giống như đoạn mã trư ớc:

```
ký tự *cwd;

cwd = get_current_dir_name ( );
nếu (!cwd)
         { lỗi ("get_current_dir_name");
         thoát (EXIT_FAILURE);
}

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

miễn phí (cwd);
```

Các hệ thống BSD cũ hơ n ư a chuộng lệnh gọi getwd() mà Linux cung cấp để tư ơ ng thích ngư ợc:

```
#define _XOPEN_SOURCE_EXTENDED /* hoặc _BSD_SOURCE */
#include <unistd.h>

ký tự * getwd (ký tự *buf);
```

Một lệnh gọi đến getwd( ) sao chép thư mục làm việc hiện tại vào buf, phải có độ dài ít nhất là PATH_MAX byte. Lệnh gọi trả về buf nếu thành công và NULL nếu thất bại. Ví dụ:

```
ký tự cwd[PATH_MAX];

nếu (!getwd (cwd))
        { lỗi ("getwd");
        thoát (EXIT_FAILURE);
}

printf ("cwd = %s\n", cwd);
```

Vì lý do tính di động và bảo mật, các ứng dụng không nên sử dụng getwd( ); nên sử dụng getcwd() .

Thay đổi thư mục làm việc hiện tại Khi

ngư ời dùng lần đầu đăng nhập vào hệ thống của mình, quy trình đăng nhập sẽ đặt thư mục làm việc hiện tại của ngư ời dùng thành thư mục home của ngư ời dùng, như đư ợc chỉ định trong /etc/passwd. Tuy nhiên, đôi khi, một quy trình muốn thay đổi thư mục làm việc hiện tại của nó. Ví dụ, một shell có thể muốn thực hiện việc này khi ngư ời dùng nhập cd.

Linux cung cấp hai lệnh gọi hệ thống để thay đổi thư mục làm việc hiện tại, một lệnh chấp nhận đư ờng dẫn của thư mục và lệnh còn lại chấp nhận mô tả tệp biểu diễn thư mục mở:

```
#include <unistd.h>

int chdir (const char *path);
int fchdir (int fd);
```

Một lệnh gọi đến chdir() sẽ thay đổi thư mục làm việc hiện tại thành tên đư ờng dẫn đư ợc chỉ định bởi path, có thể là tên đư ờng dẫn tuyệt đối hoặc tư ơ ng đối. Tư ơ ng tự, một lệnh gọi đến fchdir() sẽ thay đổi thư mục làm việc hiện tại thành tên đư ờng dẫn đư ợc biểu diễn bởi mô tả tệp fd, phải đư ợc mở trên một thư mục. Khi thành công, cả hai lệnh gọi đều trả về 0. Khi thất bại, cả hai lệnh gọi đều trả về -1.

Khi thất bại, chdir( ) cũng đặt errno thành một trong các giá trị sau:

TRUY CẬP

Quá trình gọi thiếu quyền tìm kiếm cho một trong các thành phần thư mục của đư ờng dẫn.

Đư ờng

dẫn EFAULT không phải là con trỏ hợp lệ.

EIO

Đã xảy ra lỗi I/O nội bộ.

ELOOP

Kernel gặp phải quá nhiều liên kết tư ợng trư ng khi giải quyết đư ờng dẫn.

ENAMETOOLONG

đư ờng dẫn quá dài.

ENOENT

Thư mục đư ợc trỏ tới bởi đư ờng dẫn không tồn tại.

ENOMEM

Không có đủ bộ nhớ để hoàn tất yêu cầu.

ENOTDIR

Một hoặc nhiều thành phần trong đường dẫn không phải là thư mục.

fchdir( ) đặt errno thành một trong các giá trị sau:

TRUY CẬP

Quá trình gọi không có quyền tìm kiếm cho thư mục được fd tham chiếu (tức là bit thực thi không được đặt). Điều này xảy ra nếu thư mục cấp cao nhất có thể đọc được nhưng không thể thực thi; open() thành công, nhưng fchdir() thì không.

EBADF

fd không phải là trình mô tả tệp mở.

Tùy thuộc vào hệ thống tập tin, các giá trị lỗi khác nhau sẽ có hiệu lực cho cả hai lệnh gọi.

Các lệnh gọi hệ thống này chỉ ảnh hưởng đến tiến trình đang chạy. Không có cơ chế nào trong Unix để thay đổi thư mục làm việc hiện tại của một tiến trình khác. Do đó, lệnh cd được tìm thấy trong shell không thể là một tiến trình riêng biệt (giống như hầu hết các lệnh) chỉ thực thi chdir() trên đối số dòng lệnh đầu tiên rồi thoát.
Thay vào đó, cd phải là một lệnh tích hợp đặc biệt khiến chính shell gọi chdir( ), thay đổi thư mục làm việc hiện tại của nó.

Cách sử dụng phổ biến nhất của getcwd() là lưu thư mục làm việc hiện tại để tiến trình có thể quay lại thư mục đó sau. Ví dụ:

```
ký tự *swd;
int ret;

/* lưu thư mục làm việc hiện tại */ swd =
getcwd (NULL, 0); if (!swd)
{ perror
         ("getcwd"); exit
          (EXIT_FAILURE);
}

/* chuyển sang thư mục khác */ ret = chdir
(some_other_dir); if (ret) { perror
("chdir");
          exit (EXIT_FAILURE);

}

/* thực hiện một số công việc khác trong thư mục mới... */

/* trở về thư mục đã lưu */ ret = chdir
(swd); if (ret)
{ perror
         ("chdir"); exit
          (EXIT_FAILURE);
}

miễn phí (swd);
```

Tuy nhiên, tốt hơn là mở() thư mục hiện tại, rồi sau đó mới sử dụng fchdir() vào thư mục đó. Cách tiếp cận này nhanh hơn vì kernel không lưu trữ pathname của thư mục làm việc hiện tại trong bộ nhớ; nó chỉ lưu trữ inode. Do đó, bất cứ khi nào người dùng gọi getcwd( ), kernel phải tạo pathname bằng cách duyệt cấu trúc thư mục. Ngược lại, mở thư mục làm việc hiện tại rẻ hơn vì kernel đã có inode của nó và không cần pathname có thể đọc được bằng con người để mở tệp. Đoạn mã sau sử dụng cách tiếp cận này:

```
int swd_fd;

swd_fd = mở (".", O_RDONLY);
nếu (swd_fd == -1)
        { lỗi ("mở");
        thoát (EXIT_FAILURE);
}

/* chuyển sang thư mục khác */ ret =
chdir (some_other_dir); if
(ret)
      { perror ("chdir");
        exit (EXIT_FAILURE);
}

/* thực hiện một số công việc khác trong thư mục mới... */

/* trở về thư mục đã lưu */ ret =
fchdir (swd_fd); if
(ret)
      { perror ("fchdir");
        exit (EXIT_FAILURE);
}

/* đóng fd của thư mục */ ret
= close (swd_fd); if
(ret)
      { perror ("close");
        exit (EXIT_FAILURE);
}
```

Đây là cách shell triển khai bộ nhớ đệm của thư mục trước đó (ví dụ, với cd - trong bash).

Một tiến trình không quan tâm đến thư mục làm việc hiện tại của nó—chẳng hạn như dae-mon—thường đặt nó thành / với lệnh gọi chdir("/"). Một ứng dụng giao tiếp với người dùng và dữ liệu của người dùng, chẳng hạn như trình xử lý văn bản, thường đặt thư mục làm việc hiện tại của nó thành thư mục home của người dùng hoặc thành thư mục documents đặc biệt. Vì các thư mục làm việc hiện tại chỉ có liên quan trong ngữ cảnh của các tên đường dẫn tương đối, nên thư mục làm việc hiện tại hữu ích nhất đối với các tiện ích dòng lệnh mà người dùng gọi từ shell.

Tạo thư mục

Linux cung cấp một lệnh gọi hệ thống duy nhất, được chuẩn hóa bởi POSIX, để tạo thư mục mới:

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

int mkdir (const char *path, mode_t chế độ);
```

Gọi thành công tới mkdir( ) sẽ tạo đường dẫn thư mục, có thể là đường dẫn tương đối hoặc tuyệt đối, với chế độ bit quyền (được sửa đổi bởi umask hiện tại) và trả về 0.

Umask hiện tại sửa đổi đối số chế độ theo cách thông thường, cộng với bất kỳ bit chế độ nào dành riêng cho hệ điều hành: trong Linux, các bit quyền của thư mục mới được tạo là (mode & ~umask & 01777). Nói cách khác, trên thực tế, umask cho quy trình áp đặt các hạn chế mà lệnh gọi mkdir() không thể ghi đè. Nếu thư mục cha của thư mục mới có bit ID nhóm được đặt (sgid) hoặc nếu hệ thống tệp được gắn kết với ngữ nghĩa nhóm BSD, thì thư mục mới sẽ kế thừa liên kết nhóm từ thư mục cha của nó. Nếu không, ID nhóm hiệu quả của quy trình sẽ áp dụng cho thư mục mới.

Khi thất bại, mkdir() trả về -1 và đặt errno thành một trong các giá trị sau:

TRUY CẬP

Tiến trình hiện tại không thể ghi được thư mục cha hoặc một hoặc nhiều thành phần của đường dẫn không thể tìm kiếm được.

Đường

dẫn EEXIST đã tồn tại (và không nhất thiết phải là một thư mục).

Đường

dẫn EFAULT là một con trỏ không hợp lệ.

ELOOP

Kernel gặp phải quá nhiều liên kết tượng trưng khi giải quyết đường dẫn.

ENAMETOOLONG

đường dẫn quá dài.

ENOENT

Một thành phần trong đường dẫn không tồn tại hoặc là một liên kết tượng trưng lơ lửng.

ENOMEM

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

ENOSPC

Thiết bị chứa đường dẫn đã hết dung lượng hoặc hạn ngạch đĩa của người dùng đã vượt quá giới hạn.

ENOTDIR

Một hoặc nhiều thành phần trong đường dẫn không phải là thư mục.

EPERM

Hệ thống tập tin chứa đường dẫn không hỗ trợ việc tạo thư mục.

EROFS

Hệ thống tập tin chứa đường dẫn được gắn kết chỉ đọc.

Xóa Thư mục

Tư ơ ng tự như mkdir( ), rmdir( ) đư ợc chuẩn hóa theo POSIX sẽ xóa một thư mục khỏi hệ thống phân cấp tập tin:

```
#include <unistd.h>

int rmdir (const char *path);
```

Khi thành công, rmdir() xóa path khỏi hệ thống tệp và trả về 0. Thư mục đư ợc chỉ định bởi path phải trống, ngoại trừ các thư mục dot và dot-dot. Không có lệnh gọi hệ thống nào thực hiện tư ơ ng đư ơ ng với lệnh xóa đệ quy, như với rm -r.

Một công cụ như vậy phải thực hiện thủ công việc duyệt theo chiều sâu của hệ thống tệp, xóa tất cả các tệp và thư mục bắt đầu từ các thư mục lá và di chuyển ngư ợc lên trên hệ thống tệp; rmdir( ) có thể đư ợc sử dụng ở mỗi giai đoạn để xóa một thư mục sau khi các tệp của thư mục đó đã bị xóa.

Khi lỗi, rmdir() trả về -1 và đặt errno thành một trong các giá trị sau:

EACCESS

Quyền ghi vào thư mục cha của đư ờng dẫn không đư ợc phép hoặc một trong các thư mục thành phần của đư ờng dẫn không thể tìm kiếm đư ợc.

EBUSY

path hiện đang đư ợc hệ thống sử dụng và không thể xóa. Trong Linux, điều này chỉ có thể xảy ra nếu path là điểm gắn kết hoặc thư mục gốc (thư mục gốc không cần phải là điểm gắn kết, nhờ chroot( )!).

MẶC ĐỊNH

đư ờng dẫn không phải là con trỏ hợp lệ.

EINVAL

đư ờng dẫn có thư mục dot là thành phần cuối cùng của nó.

Vòng lặp

Hạt nhân gặp phải quá nhiều liên kết tư ợng trư ng khi giải quyết đư ờng dẫn.

ENAMETOOLONG

đư ờng dẫn quá dài.

ENOENT

Thành phần trong đư ờng dẫn không tồn tại hoặc là liên kết tư ợng trư ng lơ lửng.

ENOMEM

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

ENOTDIR

Một hoặc nhiều thành phần trong đư ờng dẫn không phải là thư mục.

NHỮNG ĐIỀU GÌ ĐÓ

đư ờng dẫn chứa các mục khác ngoài các thư mục dot và dot-dot đặc biệt.

EPERM

Thư mục cha của đư ờng dẫn có bit cố định (S_ISVTX) đư ợc đặt, như ng ID ngư ời dùng hiệu quả của quy trình không phải là ID ngư ời dùng của cha nói trên cũng không phải của chính đư ờng dẫn và

tiến trình không có khả năng CAP_FOWNER . Hoặc hệ thống tập tin chứa đường dẫn không cho phép xóa thư mục.

EROFS

Hệ thống tập tin chứa đường dẫn được gắn kết chỉ đọc.

Cách sử dụng rất đơn giản:

```
int ret;

/* xóa thư mục /home/barbary/maps */ ret = rmdir
("/home/barbary/maps"); nếu (ret)
perror
       ("rmdir");
```

Đọc Nội dung của Thư mục POSIX định

nghĩa một họ các hàm để đọc nội dung của các thư mục—tức là, lấy danh sách các tệp nằm trong một thư mục nhất định. Các hàm này hữu ích nếu bạn đang triển khai lệnh ls hoặc hộp thoại lưu tệp đồ họa, nếu bạn cần thao tác trên mọi tệp trong một thư mục nhất định hoặc nếu bạn muốn tìm kiếm các tệp trong một thư mục khớp với một mẫu nhất định.

Để bắt đầu đọc nội dung của thư mục, bạn cần tạo một luồng thư mục, được biểu thị bằng đối tượng DIR :

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

ĐẠO DIỄN * opendir (const char *tên);
```

Gọi thành công tới opendir() sẽ tạo ra một luồng thư mục biểu diễn thư mục được chỉ định theo tên.

Luồng thư mục chỉ là một mô tả tệp đại diện cho thư mục mở, một số siêu dữ liệu và một bộ đệm để lưu trữ nội dung của thư mục. Do đó, có thể lấy được mô tả tệp đằng sau một luồng thư mục nhất định:

```
#define _BSD_SOURCE /* hoặc _SVID_SOURCE */
#include <sys/types.h>
#include <dirent.h>

int dirfd (DIR *dir);
```

Một lệnh gọi thành công đến dirfd() sẽ trả về mô tả tệp sao lưu thư mục luồng dir. Khi có lỗi, lệnh gọi trả về -1. Vì các hàm luồng thư mục sử dụng mô tả tệp này bên trong, nên các chương trình chỉ nên gọi các lệnh gọi không thao tác vị trí tệp. dirfd() là một phần mở rộng BSD và không được chuẩn hóa bởi POSIX; các lập trình viên muốn tuyên bố tuân thủ POSIX của họ nên tránh sử dụng.

Đọc từ luồng thư mục Sau khi

bạn đã tạo luồng thư mục bằng opendir( ), chư ơ ng trình của bạn có thể bắt đầu đọc các mục từ thư mục. Để thực hiện việc này, hãy sử dụng readdir( ), trả về từng mục từ một đối tư ợng DIR đã cho :

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

struct dirent * readdir (TRỰC TIẾP *dir);
```

Một lệnh gọi thành công đến readdir() trả về mục tiếp theo trong thư mục đư ợc biểu diễn bởi dir. Cấu trúc dirent biểu diễn một mục thư mục. Đư ợc định nghĩa trong <dirent.h>, trên Linux, định nghĩa của nó là:

```
struct dirent
         { ino_t d_ino; /* số inode */ off_t
         d_off; /* độ lệch đến dirent tiếp theo */ unsigned
         short d_reclen; /* độ dài của bản ghi này */ unsigned char
         d_type; /* loại tệp */ char d_name[256]; /*
         tên tệp */
};
```

POSIX chỉ yêu cầu trư ờng d_name , là tên của một tệp duy nhất trong thư mục. Các trư ờng khác là tùy chọn hoặc dành riêng cho Linux. Các ứng dụng mong muốn khả năng chuyển sang các hệ thống khác hoặc tuân thủ POSIX chỉ nên truy cập d_name.

Các ứng dụng lần lư ợt gọi readdir( ), lấy từng tệp trong thư mục, cho đến khi tìm thấy tệp cần tìm hoặc cho đến khi toàn bộ thư mục đư ợc đọc, lúc đó readdir() trả về NULL.

Khi thất bại, readdir() cũng trả về NULL. Để phân biệt giữa lỗi và việc đã đọc tất cả các tệp, các ứng dụng phải đặt errno thành 0 trư ớc mỗi lần gọi readdir() , sau đó kiểm tra cả giá trị trả về và errno. Giá trị errno duy nhất đư ợc readdir() đặt là EBADF, biểu thị rằng dir không hợp lệ. Do đó, nhiều ứng dụng không bận tâm đến việc kiểm tra lỗi và cho rằng NULL có nghĩa là không còn tệp nào nữa.

Đóng luồng thư mục

Để đóng luồng thư mục đư ợc mở bằng opendir( ), hãy sử dụng closedir( ):

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

int closedir (DIR *dir);
```

Gọi thành công đến closedir() sẽ đóng luồng thư mục đư ợc biểu diễn bởi dir, bao gồm cả mô tả tệp sao lư u và trả về 0. Nếu không thành công, hàm sẽ trả về -1 và đặt errno thành EBADF, mã lỗi duy nhất có thể xảy ra, biểu thị rằng dir không phải là luồng thư mục mở.

Đoạn mã sau đây triển khai một hàm, find_file_in_dir( ), sử dụng readdir() để tìm kiếm một thư mục nhất định cho một tên tệp nhất định. Nếu tệp tồn tại trong thư mục, hàm trả về 0. Nếu không, nó trả về một giá trị khác không:

```
/*
 * find_file_in_dir - tìm kiếm thư mục 'path' để tìm một * tệp có tên 'file'.
 *
 * Trả về 0 nếu 'tệp' tồn tại trong 'đường dẫn' và trả về giá trị khác
 không nếu không tồn tại.
 */
int find_file_in_dir (const char *đường dẫn, const char *tệp) {

        cấu trúc dirent *entry;
        int ret = 1;
        DIR *thư mục;

        dir = opendir (đường dẫn);

        lỗi = 0;
        trong khi ((entry = readdir (dir)) != NULL) {
                nếu (!strcmp(entry->d_name, file)) { ret
                        = 0; ngắt;

                }
        }

        nếu (errno && !entry)
                lỗi ("readdir");

        closedir (dir);
        trả về ret;
}
```

Các lệnh gọi hệ thống để đọc nội dung

thư mục Các hàm đã thảo luận trước đó để đọc nội dung thư mục được chuẩn hóa bởi POSIX và được cung cấp bởi thư viện C. Về mặt nội bộ, các hàm này sử dụng một trong hai lệnh gọi hệ thống, readdir() và getdents( ), được cung cấp ở đây để hoàn thiện:

```
#include <unistd.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <linux/unistd.h>
#include <errno.h>

/*
 * Không xác định cho không gian người dùng: cần phải
 *   sử dụng macro _syscall3() để truy cập. */
```

```
int readdir (số nguyên không dấu fd,
                struct dirent *dirp, số
                nguyên không dấu count);

int getdents (số nguyên không dấu fd,
                 struct dirent *dirp, số
                 nguyên không dấu count);
```

Bạn không muốn sử dụng các lệnh gọi hệ thống này! Chúng khó hiểu và không thể di chuyển đư ợc.
Thay vào đó, các ứng dụng không gian ngư ời dùng nên sử dụng các lệnh gọi hệ thống opendir( ) , readdir( ) và closedir() của thư viện C.

Liên kết

Nhớ lại từ cuộc thảo luận của chúng ta về các thư mục rằng mỗi ánh xạ tên-đến-inode trong một thư mục đư ợc gọi là một liên kết. Với định nghĩa đơ n giản này—rằng một liên kết về cơ bản chỉ là một tên trong danh sách (một thư mục) trỏ đến một inode—sẽ không có lý do gì khiến nhiều liên kết đến cùng một inode không thể tồn tại. Nghĩa là, một inode duy nhất (và do đó là một tệp duy nhất) có thể đư ợc tham chiếu từ, chẳng hạn, cả /etc/customs và /var/run/ledger.

Thật vậy, đây là trư ờng hợp, với một điểm cần lư u ý: vì các liên kết ánh xạ tới các inode và số inode là dành riêng cho một hệ thống tệp cụ thể, nên /etc/customs và /var/run/ledger đều phải nằm trên cùng một hệ thống tệp. Trong một hệ thống tệp duy nhất, có thể có một số lư ợng lớn các liên kết tới bất kỳ tệp nào. Giới hạn duy nhất nằm ở kích thư ớc của kiểu dữ liệu số nguyên đư ợc sử dụng để chứa số lư ợng liên kết. Trong số các liên kết khác nhau, không có liên kết nào là liên kết "gốc" hoặc "chính". Tất cả các liên kết đều có cùng trạng thái, trỏ đến cùng một tệp.

Chúng tôi gọi những loại liên kết này là liên kết cứng. Các tệp có thể không có, có một hoặc nhiều liên kết. Hầu hết các tệp có số lư ợng liên kết là 1—tức là chúng đư ợc trỏ đến bởi một mục nhập thư mục duy nhất—như ng một số tệp có hai hoặc thậm chí nhiều liên kết hơ n. Các tệp có số lư ợng liên kết là 0 không có mục nhập thư mục tư ơ ng ứng trên hệ thống tệp. Khi số lư ợng liên kết của tệp đạt đến 0, tệp đư ợc đánh dấu là miễn phí và các khối đĩa của tệp đư ợc tạo sẵn để sử dụng lại.* Tuy nhiên, một tệp như vậy vẫn nằm trên hệ thống tệp nếu một quy trình mở tệp. Khi không có quy trình nào mở tệp, tệp sẽ bị xóa.

Nhân Linux thực hiện hành vi này bằng cách sử dụng số lư ợng liên kết và số lư ợng sử dụng.
Số lần sử dụng là tổng số lần tệp đư ợc mở. Tệp không bị xóa khỏi hệ thống tệp cho đến khi cả liên kết và số lần sử dụng đều bằng 0.

Một loại liên kết khác, liên kết tư ợng trư ng, không phải là ánh xạ hệ thống tệp, mà là con trỏ cấp cao hơ n đư ợc diễn giải khi chạy. Các liên kết như vậy có thể mở rộng hệ thống tệp—chúng ta sẽ xem xét chúng ngay sau đây.

* Tìm các tệp có số lư ợng liên kết là 0, như ng các khối của chúng đư ợc đánh dấu là đã phân bổ là công việc chính của fsck, trình kiểm tra hệ thống tệp. Tình trạng như vậy có thể xảy ra khi một tệp bị xóa, như ng vẫn mở và hệ thống bị sập trư ớc khi tệp đư ợc đóng. Nhân không bao giờ có thể đánh dấu các khối hệ thống tệp là miễn phí, do đó sự khác biệt phát sinh. Ghi nhật ký hệ thống tệp loại bỏ loại lỗi này.