POLL_OUT

Có thể ghi vào tệp này.

POLL_PRI

Tệp có dữ liệu ưu tiên cao có thể đọc được.

Các mã sau đây có hiệu lực đối với SIGSEGV, mô tả hai loại truy cập bộ nhớ không hợp lệ:

SEGV_ACCERR

Tiến trình đã truy cập vào một vùng bộ nhớ hợp lệ theo cách không hợp lệ—tức là tiến trình đã vi phạm quyền truy cập bộ nhớ.

SEGV_MAPERR

Quá trình này đã truy cập vào vùng bộ nhớ không hợp lệ.

Đối với bất kỳ giá trị nào trong số này, si_addr đều chứa địa chỉ vi phạm.

Đối với SIGTRAP, hai giá trị si_code này xác định loại bẫy bị dính:

TRAP_BRKPT

Quá trình đã đạt đến điểm dừng.

TRAP_TRACE

Quá trình gặp phải bẫy theo dõi.

Lưu ý rằng si_code là trường giá trị chứ không phải trường bit.

Gửi tín hiệu với tải trọng

Như chúng ta đã thấy trong phần trước, trình xử lý tín hiệu được đăng ký với cờ SA_SIGINFO được truyền tham số siginfo_t . Cấu trúc này chứa một trường có tên là si_value, là một tải trọng tùy chọn được truyền từ bộ tạo tín hiệu đến bộ thu tín hiệu.

Hàm sigqueue () , được định nghĩa bởi POSIX, cho phép một quy trình gửi tín hiệu với dữ liệu sau:

```
#include <tín hiệu.h>

int sigqueue (pid_t pid,
              int signo,
              const union sigval value);
```

sigqueue() hoạt động tương tự như kill( ). Khi thành công, tín hiệu được xác định bởi signo sẽ được xếp hàng vào quy trình hoặc nhóm quy trình được xác định bởi pid và hàm trả về 0.
Tải trọng của tín hiệu được biểu thị theo giá trị, là hợp của một số nguyên và một con trỏ void : union

```
sigval { int
         sival_int; void
         *sival_ptr;
};
```

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

EINVAL

Tín hiệu do signo chỉ định không hợp lệ.

EPERM

Tiến trình gọi không có đủ quyền để gửi tín hiệu đến bất kỳ tiến trình nào được yêu cầu. Các quyền cần thiết để gửi tín hiệu giống như với kill() (xem phần "Gửi tín hiệu" ở đầu chương này).

CÂU CHUYỆN

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

Tương tự như kill( ), bạn có thể truyền tín hiệu null (0) cho signo để kiểm tra quyền.

Ví dụ

Ví dụ này gửi tiến trình có pid 1722 tín hiệu SIGUSR2 với dữ liệu là một số nguyên có giá trị 404:

```
giá trị sigval;
int ret;

giá trị.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, giá trị);
nếu (ret)
        perror ("sigqueue");
```

Nếu tiến trình 1722 xử lý SIGUSR2 bằng trình xử lý SA_SIGINFO , nó sẽ tìm thấy signo được đặt thành SIGUSR2, si->si_int được đặt thành 404 và si->si_code được đặt thành SI_QUEUE.

Phần kết luận

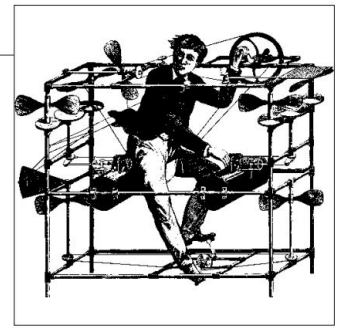
Signals có tiếng xấu trong số nhiều lập trình viên Unix. Chúng là một cơ chế cũ, lỗi thời cho giao tiếp giữa nhân và người dùng và, tốt nhất, là một dạng IPC thô sơ . Trong thế giới của các chương trình đa luồng và vòng lặp sự kiện, signal thường không phù hợp.

Tuy nhiên, dù tốt hay xấu, chúng ta vẫn cần chúng. Signals là cách duy nhất để nhận được nhiều thông báo (chẳng hạn như thông báo về việc thực thi opcode bất hợp pháp) từ kernel. Ngoài ra, signals là cách Unix (và do đó là Linux) chấm dứt các tiến trình và quản lý mối quan hệ cha/con. Do đó, chúng ta bị mắc kẹt với chúng.

Một trong những lý do chính khiến tín hiệu bị loại bỏ là khó có thể viết trình xử lý tín hiệu phù hợp, an toàn trước các mối lo ngại về khả năng nhập lại. Tuy nhiên, nếu bạn giữ cho trình xử lý của mình đơn giản và chỉ sử dụng các hàm được liệt kê trong Bảng 9-2 (nếu bạn sử dụng bất kỳ hàm nào!), chúng sẽ an toàn.

Một điểm yếu khác của tín hiệu là nhiều lập trình viên vẫn sử dụng signal() và kill( ) thay vì sigaction() và sigqueue( ) để quản lý tín hiệu. Như hai phần trư ớc đã chỉ ra, tín hiệu mạnh hơ n đáng kể và biểu cảm hơ n khi sử dụng trình xử lý tín hiệu kiểu SA_SIGINFO . Mặc dù bản thân tôi không thích tín hiệu—tôi rất muốn thấy tín hiệu đư ợc thay thế bằng cơ chế thăm dò dựa trên mô tả tệp, đây thực sự là điều đang đư ợc cân nhắc cho các phiên bản hạt nhân Linux trong tư ơ ng lai—việc giải quyết các lỗi của chúng và sử dụng giao diện tín hiệu tiên tiến của Linux giúp giảm bớt nhiều khó khăn (nếu không muốn nói là bớt than vãn).

Thời gian



Thời gian phục vụ nhiều mục đích khác nhau trong một hệ điều hành hiện đại và nhiều chương trình cần theo dõi nó. Hạt nhân đo thời gian trôi qua theo ba cách khác nhau
cách:

Thời gian tường (hoặc thời gian thực)

Đây là thời gian và ngày thực tế trong thế giới thực—tức là thời gian mà người ta đọc trên đồng hồ treo tường. Các quy trình sử dụng thời gian treo tường khi giao tiếp với người dùng hoặc đóng dấu thời gian cho một sự kiện.

Thời gian xử lý

Đây là thời gian mà một tiến trình đã sử dụng, trực tiếp trong mã không gian người dùng hoặc gián tiếp thông qua hạt nhân làm việc thay mặt cho tiến trình. Các tiến trình quan tâm đến dạng thời gian này chủ yếu để lập hồ sơ và thống kê—ví dụ như đo thời gian thực hiện một hoạt động nhất định. Wall time gây hiểu lầm khi đo hành vi của tiến trình vì, xét đến bản chất đa nhiệm của Linux, thời gian tiến trình có thể ít hơn nhiều so với thời gian wall time cho một hoạt động nhất định. Một tiến trình cũng có thể dành nhiều chu kỳ đáng kể để chờ I/O (đặc biệt là đầu vào bàn phím).

Thời gian đơn điệu

Nguồn thời gian này tăng tuyến tính nghiêm ngặt. Hầu hết các hệ điều hành, bao gồm cả Linux, đều sử dụng thời gian hoạt động của hệ thống (thời gian kể từ khi khởi động). Thời gian tường có thể thay đổi—ví dụ, vì người dùng có thể thiết lập thời gian này và vì hệ thống liên tục điều chỉnh thời gian cho độ lệch—và có thể đưa thêm sự không chính xác thông qua, chẳng hạn, giây nhuận. Mặt khác, thời gian hoạt động của hệ thống là một biểu diễn thời gian xác định và không thể thay đổi. Khía cạnh quan trọng của nguồn thời gian đơn điệu không phải là giá trị hiện tại, mà là sự đảm bảo rằng nguồn thời gian tăng tuyến tính nghiêm ngặt và do đó hữu ích cho việc tính toán sự khác biệt về thời gian giữa hai lần lấy mẫu.

Do đó, thời gian đơn điệu phù hợp để tính thời gian tương đối, trong khi thời gian tường là lý tưởng để đo thời gian tuyệt đối.

Ba phép đo thời gian này có thể đư ợc biểu diễn theo một trong hai định dạng:

Thời gian tư ơ ng đối

Đây là giá trị liên quan đến một số điểm chuẩn, chẳng hạn như thời điểm hiện tại: ví dụ, 5 giây kể từ bây giờ hoặc 10 phút trư ớc.

Thời gian tuyệt đối

Đây là thời gian không có mốc chuẩn nào như vậy: chẳng hạn, trư a ngày 25 tháng 3 năm 1968.

Cả dạng tư ơ ng đối và tuyệt đối của thời gian đều có công dụng. Một quy trình có thể cần hủy yêu cầu trong 500 mili giây, làm mới màn hình 60 lần mỗi giây hoặc lư u ý rằng 7 giây đã trôi qua kể từ khi một hoạt động bắt đầu. Tất cả những điều này đều yêu cầu tính toán thời gian tư ơ ng đối. Ngư ợc lại, một ứng dụng lịch có thể lư u ngày cho bữa tiệc toga của ngư ời dùng là ngày 8 tháng 2, một hệ thống tệp sẽ ghi ra ngày và giờ đầy đủ khi tệp đư ợc tạo (thay vì "năm giây trư ớc") và đồng hồ của ngư ời dùng hiển thị ngày Gregory, không phải số giây kể từ khi hệ thống khởi động.

Hệ thống Unix biểu diễn thời gian tuyệt đối là số giây đã trôi qua kể từ kỷ nguyên, đư ợc định nghĩa là 00:00:00 UTC vào sáng ngày 1 tháng 1 năm 1970. UTC (Giờ toàn cầu, Phối hợp) gần giống với GMT (Giờ trung bình Greenwich) hoặc giờ Zulu. Thật kỳ lạ, điều này có nghĩa là trong Unix, ngay cả thời gian tuyệt đối, ở cấp độ thấp, cũng là tư ơ ng đối. Unix giới thiệu một kiểu dữ liệu đặc biệt để lư u trữ "giây kể từ kỷ nguyên", mà chúng ta sẽ xem xét trong phần tiếp theo.

Hệ điều hành theo dõi sự tiến triển của thời gian thông qua đồng hồ phần mềm, một đồng hồ đư ợc duy trì bởi nhân trong phần mềm. Nhân khởi tạo một bộ đếm thời gian định kỳ, đư ợc gọi là bộ đếm thời gian hệ thống, bật lên ở một tần số cụ thể. Khi khoảng thời gian bộ đếm kết thúc, nhân tăng thời gian đã trôi qua thêm một đơ n vị, đư ợc gọi là tích tắc hoặc jiffy. Bộ đếm tích tắc đã trôi qua đư ợc gọi là bộ đếm jiffies. Trư ớc đây, giá trị 32 bit, jif-fies là bộ đếm 64 bit kể từ nhân Linux 2.6.*

Trên Linux, tần suất của bộ đếm thời gian hệ thống đư ợc gọi là HZ, vì một định nghĩa tiền xử lý cùng tên biểu diễn nó. Giá trị của HZ là đặc thù của kiến trúc và không phải là một phần của ABI Linux—tức là, các chư ơ ng trình không thể phụ thuộc hoặc mong đợi bất kỳ giá trị nào đư ợc đư a ra. Theo truyền thống, kiến trúc x86 sử dụng giá trị 100, nghĩa là bộ đếm thời gian hệ thống chạy 100 lần mỗi giây (tức là bộ đếm thời gian hệ thống có tần suất 100 hertz). Điều này mang lại cho mỗi jiffy giá trị là 0,01 giây—1/HZ giây. Với bản phát hành của hạt nhân Linux 2.6, các nhà phát triển hạt nhân đã tăng giá trị của HZ lên 1000, mang lại cho mỗi jiffy giá trị là 0,001 giây. Tuy nhiên, trong phiên bản 2.6.13 trở lên, HZ là 250, mang lại cho mỗi jiffy giá trị là 0,004 giây.† Có một sự đánh đổi vốn có trong giá trị của HZ: các giá trị cao hơ n cung cấp độ phân giải cao hơ n, như ng phải chịu chi phí bộ đếm thời gian lớn hơ n.

* Các phiên bản tư ơ ng lai của hạt nhân Linux có thể trở nên "tickless" hoặc triển khai "dynamic ticks", trong trư ờng hợp đó, hạt nhân sẽ không theo dõi giá trị jiffies rõ ràng. Thay vào đó, tất cả các hoạt động của hạt nhân dựa trên thời gian sẽ thực hiện từ các bộ đếm thời gian đư ợc khởi tạo động thay vì từ bộ đếm thời gian hệ thống.

† HZ hiện cũng là tùy chọn kernel thời gian biên dịch, với các giá trị 100, 250 và 1000 đư ợc hỗ trợ trên kiến trúc x86. Bất kể thế nào, không gian ngư ời dùng không thể phụ thuộc vào bất kỳ giá trị cụ thể nào cho HZ.

Mặc dù các quy trình không nên dựa vào bất kỳ giá trị cố định nào của HZ, POSIX vẫn định nghĩa một cơ chế để xác định tần suất bộ đếm thời gian hệ thống khi chạy:

```
Hz dài;

hz = sysconf (_SC_CLK_TCK); nếu
(hz == -1) lỗi
         ("sysconf"); /* không bao giờ xảy ra */
```

Giao diện này hữu ích khi một chương trình muốn xác định độ phân giải của bộ đếm thời gian của hệ thống, nhưng không cần thiết để chuyển đổi giá trị thời gian hệ thống sang giây vì hầu hết các giao diện POSIX đều xuất các phép đo thời gian đã được chuyển đổi hoặc được chia tỷ lệ thành tần số cố định, không phụ thuộc vào HZ. Không giống như HZ, tần số cố định này là một phần của ABI hệ thống; trên x86, giá trị là 100. Các hàm POSIX trả về thời gian theo tích tắc đồng hồ sử dụng CLOCKS_PER_SEC để biểu diễn tần số cố định.

Thỉnh thoảng, các sự kiện kết hợp lại để tắt máy tính. Đôi khi, máy tính thậm chí không được cắm điện; tuy nhiên, khi khởi động, chúng có thời gian chính xác. Điều này là do hầu hết các máy tính đều có đồng hồ phần cứng chạy bằng pin lưu trữ thời gian và ngày tháng khi máy tính tắt. Khi hạt nhân khởi động, nó khởi tạo khái niệm về thời gian hiện tại từ đồng hồ phần cứng. Tương tự như vậy, khi người dùng tắt hệ thống, hạt nhân ghi thời gian hiện tại trở lại đồng hồ phần cứng. Người quản trị hệ thống có thể đồng bộ hóa thời gian tại các điểm khác thông qua lệnh hwclock .

Quản lý thời gian trôi qua trên hệ thống Unix liên quan đến một số tác vụ, chỉ một số trong đó bất kỳ quy trình nào cũng quan tâm: chúng bao gồm thiết lập và truy xuất thời gian tường hiện tại, tính toán thời gian đã trôi qua, ngủ trong một khoảng thời gian nhất định, thực hiện các phép đo thời gian có độ chính xác cao và kiểm soát bộ hẹn giờ. Chương này đề cập đến toàn bộ các công việc liên quan đến thời gian này. Chúng ta sẽ bắt đầu bằng cách xem xét các cấu trúc dữ liệu mà Linux sử dụng để biểu diễn thời gian.

Cấu trúc dữ liệu của Time

Khi các hệ thống Unix phát triển, triển khai giao diện riêng để quản lý thời gian, các cấu trúc dữ liệu đa dạng đã trở thành biểu tượng cho khái niệm thời gian có vẻ đơn giản. Các cấu trúc dữ liệu này trải dài từ số nguyên đơn giản đến nhiều cấu trúc đa trường khác nhau. Chúng ta sẽ đề cập đến chúng ở đây trước khi đi sâu vào các giao diện thực tế.

Biểu diễn ban đầu Cấu trúc dữ liệu đơn

giản nhất là time_t, được định nghĩa trong tiêu đề <time.h>. Ý định là time_t là một kiểu mờ đục. Tuy nhiên, trên hầu hết các hệ thống Unix—kể cả Linux—kiểu này là một typedef đơn giản cho kiểu C long :

```
typedef thời gian dài_t;
```

time_t biểu thị số giây đã trôi qua kể từ kỷ nguyên. “Điều đó sẽ không kéo dài lâu trư ớc khi tràn!” là một phản ứng điển hình. Trên thực tế, nó sẽ kéo dài lâu hơ n bạn có thể mong đợi, như ng thực tế nó sẽ tràn khi vẫn còn nhiều hệ thống Unix đang đư ợc sử dụng. Với kiểu dài 32 bit , time_t có thể biểu diễn tới 2.147.483.647 giây sau kỷ nguyên. Điều này cho thấy rằng chúng ta sẽ lại có sự cố Y2K một lần nữa—vào năm 2038! Với Tuy nhiên, may mắn thay, đến 22:14:07 ngày thứ Hai, 18 tháng 1 năm 2038, hầu hết các hệ thống và phần mềm sẽ là 64-bit.

Và bây giờ, độ chính xác đến từng micro giây

Một vấn đề khác với time_t là có rất nhiều thứ có thể xảy ra chỉ trong một giây. Timeval cấu trúc mở rộng time_t để thêm độ chính xác micro giây. Tiêu đề <sys/time.h> định nghĩa cấu trúc này như sau:

```
#include <hệ thống/thời gian.h>

cấu trúc thời gian {
          thời gian_t tv_giây;            /* giây */
          giây_t tv_usec;                 /* micro giây */
};
```

tv_sec đo giây và tv_usec đo micro giây. Sự khó hiểu suseconds_t thư ờng là một typedef cho một kiểu số nguyên.

Thậm chí còn tốt hơ n: Độ chính xác nano giây

Không hài lòng với độ phân giải micro giây, cấu trúc timespec nâng cao mức độ nano giây. Tiêu đề <time.h> định nghĩa cấu trúc này như sau:

```
#include <thời gian.h>

cấu trúc timespec {
          thời gian t                 /* giây */
          tv_giây; dài tv_nsec;       /* nano giây */
};
```

Khi đư ợc lựa chọn, các giao diện thích độ phân giải nano giây hơ n micro giây.* Do đó, kể từ khi giới thiệu cấu trúc timespec , hầu hết các giao diện liên quan đến thời gian đã chuyển sang nó, và do đó đã đạt đư ợc độ chính xác cao hơ n. Tuy nhiên, như chúng ta sẽ thấy, một chức năng quan trọng vẫn sử dụng timeval.

Trong thực tế, không có cấu trúc nào thư ờng cung cấp độ chính xác đã nêu vì hệ thống bộ đếm thời gian không cung cấp độ phân giải nano giây hoặc thậm chí micro giây. Tuy nhiên, nó tốt nhất là có độ phân giải có sẵn trong giao diện để nó có thể chứa bất kể hệ thống cung cấp độ phân giải nào.

* Ngoài ra, cấu trúc timespec đã loại bỏ suseconds_t ngớ ngẩn , thay vào đó là một long đơ n giản và không phô trư ơ ng .

Phân tích thời gian

Một số chức năng mà chúng ta sẽ đề cập đến chuyển đổi giữa thời gian Unix và chuỗi, hoặc xây dựng theo chư ơ ng trình một chuỗi biểu diễn một ngày nhất định. Để tạo điều kiện thuận lợi cho quá trình này, tiêu chuẩn C cung cấp cấu trúc tm để biểu diễn thời gian "bị phá vỡ" trong một định dạng dễ đọc hơ n đối với con ngư ời. Cấu trúc này cũng đư ợc định nghĩa trong <time.h>:

```
#include <thời gian.h>

cấu trúc tm {
        int tm_sec;             /* giây */
        int tm_min;             /* phút */
        int tm_hour;            /* giờ */
        int tm_mday;            /* ngày trong tháng */
        int tm_mon;             /* tháng */
        int tm_year;            /* năm */
        int tm_wday;            /* ngày trong tuần */
        int tm_yday;            /* ngày trong năm */
        int tm_isdst;           /* giờ tiết kiệm ánh sáng ban ngày? */
#ifdef _BSD_SOURCE
        long tm_gmtoff;         /* chênh lệch múi giờ so với GMT */
        const char *tm_zone; /* viết tắt múi giờ */
#endif /* _BSD_SOURCE */
};
```

Cấu trúc tm giúp dễ dàng hơ n để biết giá trị time_t , chẳng hạn như 314159 là ngày Chủ Nhật hay thứ Bảy (là ngày đầu tiên). Về mặt không gian, rõ ràng đây là lựa chọn kém cho thể hiện ngày tháng và thời gian, như ng rất tiện lợi khi chuyển đổi sang và từ các giá trị theo ý ngư ời dùng.

Các trư ờng như sau:

tm_giây
: Số giây sau phút. Giá trị này thư ờng nằm trong khoảng từ 0 đến 59, như ng có thể lên tới 61 để chỉ tới hai giây nhuận.

tm_phút
: Số phút sau giờ. Giá trị này nằm trong khoảng từ 0 đến 59.

tm_giờ
: Số giờ sau nửa đêm. Giá trị này nằm trong khoảng từ 0 đến 23.

tm_mday
: Ngày trong tháng. Giá trị này nằm trong khoảng từ 0 đến 31. POSIX không chỉ định giá trị 0; tuy nhiên, Linux sử dụng nó để chỉ ngày cuối cùng của ngày trư ớc đó tháng.

tm_mon
: Số tháng kể từ tháng 1. Giá trị này nằm trong khoảng từ 0 đến 11.

tm_năm
: Số năm kể từ năm 1900.

tm_wday

Số ngày kể từ Chủ Nhật. Giá trị này nằm trong khoảng từ 0 đến 6.

tm_yday

Số ngày kể từ ngày 1 tháng 1. Giá trị này nằm trong khoảng từ 0 đến 365. tm_isdst

Một giá

trị đặc biệt cho biết giờ tiết kiệm ánh sáng ban ngày (DST) có hiệu lực tại thời điểm đư ợc mô tả bởi các trư ờng khác hay không. Nếu giá trị là dư ơ ng, DST có hiệu lực. Nếu là 0, DST không có hiệu lực. Nếu giá trị là âm, trạng thái của DST là không xác định.

tm_gmtoff

Độ lệch tính bằng giây của múi giờ hiện tại so với Giờ trung bình Greenwich. Trư ờng này chỉ có nếu _BSD_SOURCE đư ợc định nghĩa trư ớc khi bao gồm <time.h>.

tm_zone

Viết tắt của múi giờ hiện tại—ví dụ: EST. Trư ờng này chỉ có nếu _BSD_SOURCE đư ợc định nghĩa trư ớc khi bao gồm <time.h>.

Kiểu cho Thời gian xử lý Kiểu

clock_t biểu diễn các tích tắc đồng hồ. Đây là kiểu số nguyên, thư ờng là số dài. Tùy thuộc vào giao diện, các tích tắc mà clock_t biểu thị tần số bộ đếm thời gian thực tế của hệ thống (HZ) hoặc CLOCKS_PER_SEC.

Đồng hồ POSIX

Một số lệnh gọi hệ thống đư ợc thảo luận trong chư ơ ng này sử dụng đồng hồ POSIX, một tiêu chuẩn để triển khai và biểu diễn các nguồn thời gian. Kiểu clockid_t biểu diễn một đồng hồ POSIX cụ thể, bốn trong số đó đư ợc Linux hỗ trợ:

CLOCK_MONOTONIC

Đồng hồ tăng đơ n điệu không thể thiết lập bởi bất kỳ quy trình nào. Nó biểu diễn thời gian đã trôi qua kể từ một điểm bắt đầu không xác định, chẳng hạn như khởi động hệ thống.

CLOCK_PROCESS_CPUTIME_ID Đồng

hồ có độ phân giải cao, theo từng quy trình có sẵn từ bộ xử lý. Ví dụ, trên kiến trúc i386, đồng hồ này sử dụng thanh ghi bộ đếm dấu thời gian (TSC).

CLOCK_REALTIME

Đồng hồ thời gian thực (thời gian tư ờng) trên toàn hệ thống. Việc thiết lập đồng hồ này yêu cầu các đặc quyền đặc biệt.

ID_CHUỖI_CPU_ĐỒNG HỒ

Tư ơ ng tự như đồng hồ cho mỗi tiến trình, như ng duy nhất cho mỗi luồng trong một tiến trình.

POSIX định nghĩa tất cả bốn nguồn thời gian này như ng chỉ yêu cầu CLOCK_REALTIME.
Do đó, trong khi Linux cung cấp đáng tin cậy cả bốn đồng hồ thì mã di động chỉ nên dựa vào CLOCK_REALTIME.

Độ phân giải nguồn thời gian

POSIX định nghĩa hàm clock_getres() để lấy độ phân giải của một nguồn thời gian nhất định:

```
#include <thời gian.h>

int clock_getres (clockid_t clock_id, struct
                        timespec *res);
```

Một lệnh gọi thành công đến clock_getres( ) sẽ lưu trữ độ phân giải của đồng hồ được chỉ định bởi clock_id trong res, nếu nó không phải là NULL, và trả về 0. Nếu không thành công, hàm sẽ trả về -1 và đặt errno thành một trong hai mã lỗi sau:

MẶC ĐỊNH

res là một con trỏ không hợp lệ.

EINVAL

clock_id không phải là nguồn thời gian hợp lệ trên hệ thống này.

Ví dụ sau đây đưa ra kết quả phân giải của bốn nguồn thời gian đã thảo luận ở phần trước:

```
clockid_t đồng hồ[] = {
          CLOCK_REALTIME,
          ĐỒNG HỒ_ĐƠN_TỰ,
          ID_THỜI_GỒM_XỬ_LÝ_CPU,
          CLOCK_THREAD_CPUTIME_ID,
          (clockid_t) -1 };
số nguyên i;

đối với (i = 0; đồng hồ[i] != (clockid_t) -1; i++) { struct
          timespec res; int ret;


          ret = clock_getres(clocks[i], &res); nếu (ret)
          perror
                 ("clock_getres");
          khác
                    printf ("đồng hồ=%d giây=%ld nsec=%ld\n", đồng
                              hồ[i], res.tv_sec, res.tv_nsec);
}
```

Trên hệ thống x86 hiện đại, kết quả đầu ra sẽ như sau:

```
đồng hồ=0 giây=0 nsec=4000250 đồng
hồ=1 giây=0 nsec=4000250 đồng hồ=2
giây=0 nsec=1 đồng hồ=3
giây=0 nsec=1
```

Lưu ý rằng 4.000.250 nano giây là 4 mili giây, tức là 0,004 giây. Đổi lại, 0,004 giây là độ phân giải của đồng hồ hệ thống x86 với giá trị HZ là 250, như chúng ta đã thảo luận trong phần đầu tiên của chương này. Do đó, chúng ta thấy rằng cả CLOCK_REALTIME

và CLOCK_MONOTONIC được liên kết với jiffies và độ phân giải do bộ đếm thời gian hệ thống cung cấp. Ngược lại, cả CLOCK_PROCESS_CPUTIME_ID và CLOCK_PROCESS_CPUTIME_ID đều sử dụng nguồn thời gian có độ phân giải cao hơn—trên máy x86 này, TSC, mà chúng ta thấy cung cấp độ phân giải nano giây.

Trên Linux (và hầu hết các hệ thống Unix khác), tất cả các hàm sử dụng đồng hồ POSIX đều yêu cầu liên kết tệp đối tượng kết quả với librt. Ví dụ, nếu biên dịch đoạn mã trước thành một tệp thực thi hoàn chỉnh, bạn có thể sử dụng lệnh sau:

```
$ gcc -Wall -W -O2 -lrt -g -o đoạn trích snippet.c
```

Nhận thời gian hiện tại trong ngày

Có một số lý do khiến các ứng dụng muốn có ngày và giờ hiện tại: để hiển thị cho người dùng, để tính thời gian tương đối hoặc thời gian đã trôi qua, để đóng dấu thời gian cho một sự kiện, v.v. Cách đơn giản nhất và phổ biến nhất trong lịch sử để lấy thời gian hiện tại là hàm time() :

```
#include <thời gian.h>

time_t time (time_t *t);
```

Gọi time() trả về thời gian hiện tại được biểu diễn dưới dạng số giây đã trôi qua kể từ kỷ nguyên. Nếu tham số t không phải là NULL, hàm cũng ghi thời gian hiện tại vào con trỏ được cung cấp.

Khi có lỗi, hàm trả về -1 (chuyển kiểu thành time_t) và đặt errno một cách thích hợp. Lỗi duy nhất có thể xảy ra là EFAULT, lưu ý rằng t là một con trỏ không hợp lệ.

Ví dụ:

```
thời gian t t;

printf ("thời gian hiện tại: %ld\n", (dài) time (&t)); printf
("cùng giá trị: %ld\n", (dài) t);
```

> Một cách tiếp cận ngây thơ với thời gian
>
> Biểu diễn "giây đã trôi qua kể từ kỷ nguyên" của time_t không phải là số giây thực tế đã trôi qua kể từ thời điểm định mệnh đó. Phép tính Unix giả định rằng năm nhuận là tất cả các năm chia hết cho bốn và bỏ qua hoàn toàn giây nhuận.
>
> Mục đích của biểu diễn time_t không phải là nó chính xác mà là nó nhất quán—và thực tế là như vậy.

Giao diện tốt hơ n

Hàm gettimeofday() mở rộng time() bằng cách cung cấp độ phân giải micro giây:

```
#include <hệ thống/thời gian.h>

int gettimeofday (struct timeval *tv, struct
                  timezone *tz);
```

Một lệnh gọi thành công đến gettimeofday() sẽ đặt thời gian hiện tại vào cấu trúc timeval đư ợc trỏ đến bởi tv và trả về 0. Cấu trúc múi giờ và tham số tz đã lỗi thời; không nên sử dụng cả hai trên Linux. Luôn truyền NULL cho tz.

Khi thất bại, lệnh gọi trả về -1 và đặt errno thành EFAULT; đây là lỗi duy nhất có thể xảy ra, biểu thị rằng tv hoặc tz là con trỏ không hợp lệ.

Ví dụ:

```
cấu trúc timeval tv;
int ret;

ret = gettimeofday (&tv, NULL); nếu
(ret)
      perror ("gettimeofday");
khác
          printf ("giây=%ld ugiây=%ld\n", (dài) tv.sec,
                    (dài) tv.usec);
```

Cấu trúc múi giờ đã lỗi thời vì kernel không quản lý múi giờ và glibc từ chối sử dụng trư ờng tz_dsttime của cấu trúc múi giờ . Chúng ta sẽ xem xét cách thao tác múi giờ trong phần tiếp theo.

Giao diện nâng cao

POSIX cung cấp giao diện clock_gettime() để lấy thời gian của một nguồn thời gian cụ thể. Tuy nhiên, hữu ích hơ n là hàm này cho phép độ chính xác nano giây:

```
#include <thời gian.h>

int clock_gettime (clockid_t clock_id, struct
                   timespec *ts);
```

Khi thành công, lệnh gọi trả về 0 và lư u trữ thời gian hiện tại của nguồn thời gian đư ợc chỉ định bởi clock_id trong ts. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong các giá trị sau:

MẶC ĐỊNH

ts là một con trỏ không hợp lệ.

EINVAL

clock_id là nguồn thời gian không hợp lệ trên hệ thống này.

Ví dụ sau đây lấy thời gian hiện tại của tất cả bốn múi giờ chuẩn

nguồn:

```
clockid_t đồng hồ[] = {
          CLOCK_REALTIME,
          ĐỒNG HỒ_ĐƠ N_TỰ,
          ID_THỜI_GỒM_XỬ_LÝ_CPU,
          CLOCK_THREAD_CPUTIME_ID,
          (clockid_t) -1 };
số nguyên i;

đối với (i = 0; đồng hồ[i] != (clockid_t) -1; i++) { struct
          timespec ts; int ret;


          ret = clock_gettime(clocks[i], &ts); nếu (ret)
          lỗi
              ("clock_gettime");
          khác
                    printf ("đồng hồ=%d giây=%ld nsec=%ld\n", đồng
                              hồ[i], ts.tv_sec, ts.tv_nsec);
}
```

Nhận thời gian xử lý

Lệnh gọi hệ thống times( ) lấy thời gian xử lý của tiến trình đang chạy và các tiến trình con của nó, theo từng nhịp đồng hồ:

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

struct tms
          { clock_t tms_utime; /* thời gian ngư ời dùng sử dụng */
          clock_t tms_stime; /* thời gian hệ thống sử dụng */ clock_t
          tms_cutime; /* thời gian ngư ời dùng sử dụng bởi các thành phần con */
          clock_t tms_cstime; /* thời gian hệ thống sử dụng bởi các thành phần con */
};

clock_t lần (struct tms *buf);
```

Khi thành công, lệnh gọi sẽ điền vào cấu trúc tms đư ợc cung cấp đư ợc trỏ tới bởi buf với thời gian xử lý đư ợc tiêu thụ bởi quy trình gọi và các quy trình con của nó. Thời gian đư ợc báo cáo đư ợc chia thành thời gian của ngư ời dùng và hệ thống. Thời gian của ngư ời dùng là thời gian dành cho việc thực thi mã trong không gian ngư ời dùng. Thời gian của hệ thống là thời gian dành cho việc thực thi mã trong không gian hạt nhân—ví dụ, trong một lệnh gọi hệ thống hoặc lỗi trang. Thời gian đư ợc báo cáo cho mỗi con chỉ đư ợc bao gồm sau khi con kết thúc và cha mẹ gọi waitpid() (hoặc một hàm liên quan) trên quy trình. Lệnh gọi trả về số tích tắc đồng hồ, tăng đều, kể từ một điểm tùy ý trong quá khứ. Điểm tham chiếu này đã từng là khởi động hệ thống—do đó, hàm times() trả về thời gian hoạt động của hệ thống, tính bằng tích tắc—như ng điểm tham chiếu hiện là khoảng 429 triệu giây trư ớc khi hệ thống khởi động. Hạt nhân

các nhà phát triển đã triển khai thay đổi này để bắt mã hạt nhân không thể xử lý thời gian hoạt động của hệ thống đang bao quanh và đạt đến số không. Do đó, giá trị tuyệt đối của hàm trả về này là vô giá trị; tuy nhiên, những thay đổi tư ơ ng đối giữa hai lần gọi vẫn có giá trị.

Khi lỗi, lệnh gọi trả về -1 và đặt errno thành giá trị phù hợp. Trên Linux, mã lỗi duy nhất có thể xảy ra là EFAULT, biểu thị rằng buf là con trỏ không hợp lệ.

Thiết lập thời gian hiện tại trong ngày

Trong khi các phần trư ớc đã mô tả cách lấy thời gian, đôi khi các ứng dụng cũng cần đặt thời gian và ngày hiện tại thành một giá trị đư ợc cung cấp. Điều này hầu như luôn đư ợc xử lý bởi một tiện ích đư ợc thiết kế riêng cho mục đích này, chẳng hạn như ngày.

Đối tác thiết lập thời gian của time( ) là stime( ):

```
#define _SVID_SOURCE
#include <time.h>

int thời gian (thời gian_t *t);
```

Một cuộc gọi thành công đến stime() sẽ đặt thời gian hệ thống thành giá trị đư ợc trỏ bởi t và trả về 0.
Cuộc gọi yêu cầu ngư ời dùng gọi phải có khả năng CAP_SYS_TIME .
Nói chung, chỉ có ngư ời dùng root mới có khả năng này.

Khi thất bại, lệnh gọi trả về -1 và đặt errno thành EFAULT, biểu thị rằng t là một con trỏ không hợp lệ hoặc EPERM, biểu thị rằng ngư ời dùng đang gọi không sở hữu khả năng CAP_SYS_TIME .

Cách sử dụng rất đơ n giản:

```
thời gian t =
1; int ret;

/* đặt thời gian thành một giây sau kỷ nguyên */ ret
= stime(&t); nếu
(ret)
      perror ("stime");
```

Chúng ta sẽ xem xét các hàm giúp chuyển đổi dạng thời gian dễ đọc của con ngư ời sang time_t dễ dàng hơ n trong phần tiếp theo.

Cài đặt thời gian chính xác

Đối tác của gettimeofday() là settimeofday( ):

```
#include <hệ thống/thời gian.h>

int settimeofday (const struct timeval *tv const    ,
                  struct timezone *tz);
```

Gọi thành công tới settimeofday() sẽ đặt thời gian hệ thống theo như tv cung cấp và trả về 0. Giống như gettimeofday( ), truyền NULL cho tz là cách thực hành tốt nhất. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong những giá trị sau:

MẶC ĐỊNH

tv hoặc tz trỏ tới một vùng bộ nhớ không hợp lệ.

EINVAL

Một trư ờng trong các cấu trúc đư ợc cung cấp là không hợp lệ.

EPERM

Quá trình gọi thiếu khả năng CAP_SYS_TIME .

Ví dụ sau đây đặt thời gian hiện tại thành thứ Bảy vào giữa tháng 12 năm 1979:

```
cấu trúc timeval tv = { .tv_sec = 31415926, .tv_usec
                              = 27182818 };
int ret;

ret = settimeofday (&tv, NULL); nếu
(ret) lỗi
          ("settimeofday");
```

Giao diện nâng cao để thiết lập thời gian

Cũng giống như clock_gettime() cải thiện gettimeofday( ) , clock_settime() làm cho settimeofday( ) lỗi thời:

```
#include <thời gian.h>

int clock_settime (clockid_t clock_id, const
                        struct timespec *ts);
```

Khi thành công, lệnh gọi trả về 0 và nguồn thời gian đư ợc chỉ định bởi clock_id đư ợc đặt thành thời gian đư ợc chỉ định bởi ts. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong các giá trị sau:

EFAULT

ts là một con trỏ không hợp lệ.

EINVAL

clock_id là nguồn thời gian không hợp lệ trên hệ thống này.

EPERM

Quy trình thiếu các quyền cần thiết để thiết lập nguồn thời gian đã chỉ định hoặc nguồn thời gian đã chỉ định có thể chư a đư ợc thiết lập.

Trên hầu hết các hệ thống, nguồn thời gian có thể thiết lập duy nhất là CLOCK_REALTIME. Do đó, lợi thế duy nhất của hàm này so với settimeofday() là nó cung cấp độ chính xác nano giây (cùng với việc không phải xử lý cấu trúc múi giờ vô giá trị).

Chơi với thời gian

Hệ thống Unix và ngôn ngữ C cung cấp một họ các hàm để chuyển đổi giữa thời gian chia nhỏ (một chuỗi ASCII biểu diễn thời gian) và time_t. asctime( ) chuyển đổi một cấu trúc tm —thời gian chia nhỏ—thành một chuỗi ASCII:

```
#include <thời gian.h>

char * asctime (const struct tm *tm); asctime_r (const
một       struct tm *tm, char *buf); char * Nó trả về một con trỏ đến
```

chuỗi được phân bổ tĩnh. Một lệnh gọi tiếp theo đến bất kỳ hàm thời gian nào cũng có thể ghi đè lên chuỗi này; asctime() không an toàn cho luồng.

Do đó, các chương trình đa luồng (và các nhà phát triển ghét các giao diện được thiết kế kém) nên sử dụng asctime_r( ). Thay vì trả về một con trỏ tới một chuỗi được phân bổ tĩnh, hàm này sử dụng chuỗi được cung cấp qua buf, chuỗi này phải có độ dài ít nhất là 26 ký tự.

Cả hai hàm đều trả về NULL trong trường hợp có lỗi.

mktime( ) cũng chuyển đổi cấu trúc tm , nhưng nó chuyển đổi thành time_t:

```
#include <thời gian.h>

time_t mktime (cấu trúc tm *tm);
```

mktime() cũng đặt múi giờ thông qua tzset( ), như được chỉ định bởi tm. Khi có lỗi, nó trả về -1 (chuyển kiểu thành time_t).

ctime( ) chuyển đổi time_t thành dạng biểu diễn ASCII của nó:

```
#include <thời gian.h>

char * ctime (const time_t *timep); char *
ctime_r (const time_t *timep, char *buf);
```

Khi lỗi, nó trả về NULL. Ví dụ: time_t t =

```
time (NULL);

printf ("thời gian cách đây một dòng: %s", ctime (&t));
```

Lưu ý thiếu xuống dòng. Có lẽ bất tiện, ctime() thêm xuống dòng vào chuỗi trả về của nó.

Giống như asctime( ), ctime() trả về một con trỏ đến một chuỗi tĩnh. Vì điều này không an toàn cho luồng, các chương trình luồng nên sử dụng ctime_r( ), hoạt động trên bộ đệm do buf cung cấp . Bộ đệm phải có độ dài ít nhất là 26 ký tự. gmtime() chuyển đổi time_t đã cho thành một cấu trúc tm , được

thể hiện theo múi giờ UTC:

```
#include <thời gian.h>

cấu trúc tm *   gmtime (const time_t *timep);
cấu trúc tm *   gmtime_r (const time_t *timep, struct tm *result);
```

Nếu thất bại, nó sẽ trả về NULL.

Hàm này phân bổ tĩnh cấu trúc được trả về và một lần nữa, do đó không an toàn cho luồng. Các chương trình luồng nên sử dụng gmtime_r( ), hoạt động trên cấu trúc được trỏ tới bởi result. localtime() và localtime_r() thực hiện

các hàm tương tự như gmtime() và gmtime_r( ), nhưng chúng thể hiện time_t đã cho theo múi giờ của người dùng:

```
#include <thời gian.h>

struct tm * giờ địa phương (const time_t *timep); struct tm *
                localtime_r (const time_t *timep, struct tm *result);
```

Giống như mktime( ), lệnh gọi localtime() cũng gọi tzset( ) và khởi tạo múi giờ. Không rõ localtime_r() có thực hiện bước này hay không. difftime() trả về số giây đã trôi qua giữa

hai giá trị time_t , ép kiểu thành double:

```
#include <thời gian.h>

chênh lệch thời gian gấp đôi (time_t time1, time_t time0);
```

Trên tất cả các hệ thống POSIX, time_t là một kiểu số học và difftime() tương đương với kiểu sau, bỏ qua việc phát hiện tràn số trong phép trừ:

```
(gấp đôi) (thời gian1 - thời gian0)
```

Trên Linux, vì time_t là kiểu số nguyên nên không cần phải ép kiểu thành double.
Tuy nhiên, để duy trì tính di động, hãy sử dụng difftime( ).

Điều chỉnh Đồng hồ Hệ thống

Những bước nhảy lớn và đột ngột trong thời gian đồng hồ treo tường có thể gây ra sự hỗn loạn cho các ứng dụng phụ thuộc vào thời gian tuyệt đối để hoạt động. Hãy xem xét ví dụ về make, xây dựng các dự án phần mềm được mô tả chi tiết bởi Makefile. Mỗi lần gọi chương trình không xây dựng lại toàn bộ cây nguồn; nếu có, trong các dự án phần mềm lớn, một tệp đã thay đổi có thể dẫn đến nhiều giờ xây dựng lại. Thay vào đó, make xem xét các dấu thời gian sửa đổi tệp của tệp nguồn (ví dụ: wolf.c) so với tệp đối tượng (wolf.o). Nếu tệp nguồn—hoặc bất kỳ điều kiện tiên quyết nào của nó, chẳng hạn như wolf.h— mới hơn tệp đối tượng, make sẽ xây dựng lại tệp nguồn thành tệp đối tượng đã cập nhật. Tuy nhiên, nếu tệp nguồn không mới hơn đối tượng, thì không có hành động nào được thực hiện.

Với điều này trong đầu, hãy xem xét điều gì có thể xảy ra nếu người dùng nhận ra đồng hồ của mình lệch vài giờ và chạy date để cập nhật đồng hồ hệ thống. Nếu sau đó người dùng cập nhật và lưu lại wolf.c, chúng ta có thể gặp rắc rối. Nếu người dùng đã dịch chuyển thời gian hiện tại ngược lại, wolf.c sẽ trông cũ hơn wolf.o—mặc dù không phải vậy!—và sẽ không có quá trình xây dựng lại nào xảy ra.

Để ngăn chặn thảm họa như vậy, Unix cung cấp hàm adjtime() , hàm này sẽ điều chỉnh dần thời gian hiện tại theo hướng của delta đã cho. Mục đích là để các hoạt động nền như daemon Network Time Protcol (NTP) liên tục điều chỉnh thời gian để hiệu chỉnh độ lệch của đồng hồ, sử dụng adjtime() để giảm thiểu tác động của chúng lên hệ thống:

```
#define _BSD_SOURCE
#include <sys/time.h>

int adjtime (const struct timeval *delta,
             struct timeval *olddelta);
```

Một lệnh gọi thành công đến adjtime() sẽ hướng dẫn kernel bắt đầu điều chỉnh thời gian một cách chậm rãi theo quy định của delta, sau đó trả về 0. Nếu thời gian do delta chỉ định là dương, kernel sẽ tăng tốc đồng hồ hệ thống theo delta cho đến khi hiệu chỉnh được áp dụng hoàn toàn. Nếu thời gian do delta chỉ định là âm, kernel sẽ làm chậm đồng hồ hệ thống cho đến khi hiệu chỉnh được áp dụng. Kernel áp dụng tất cả các điều chỉnh sao cho đồng hồ luôn tăng đơn điệu và không bao giờ trải qua sự thay đổi thời gian đột ngột. Ngay cả với delta âm, việc điều chỉnh sẽ không làm đồng hồ lùi lại; thay vào đó, đồng hồ sẽ chậm lại cho đến khi thời gian hệ thống hội tụ với thời gian đã hiệu chỉnh.

Nếu delta không phải là NULL, thì hạt nhân sẽ dừng xử lý bất kỳ sửa đổi nào đã đăng ký trước đó. Tuy nhiên, phần sửa đổi đã thực hiện, nếu có, sẽ được duy trì. Nếu olddelta không phải là NULL, thì bất kỳ sửa đổi nào đã đăng ký trước đó nhưng chưa áp dụng sẽ được ghi vào cấu trúc timeval đã cung cấp . Việc truyền một delta NULL và một olddelta hợp lệ sẽ cho phép truy xuất bất kỳ sửa đổi nào đang diễn ra.

Các sửa đổi được áp dụng bởi adjtime() phải nhỏ—trường hợp sử dụng lý tưởng là NTP, như đã đề cập trước đó, áp dụng các sửa đổi nhỏ (vài giây). Linux duy trì ngưỡng sửa đổi tối thiểu và tối đa là vài nghìn giây theo cả hai hướng.

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

MẶC ĐỊNH
: delta hoặc olddelta là con trỏ không hợp lệ.

EINVAL
: Sự điều chỉnh được phân định bằng delta là quá lớn hoặc quá nhỏ.

EPERM
: Người dùng đang gọi không có khả năng CAP_SYS_TIME .

RFC 1305 định nghĩa một thuật toán điều chỉnh xung nhịp mạnh hơn đáng kể và phức tạp hơn tương ứng so với phương pháp hiệu chỉnh dần dần được thực hiện bởi adjtime( ). Linux triển khai thuật toán này bằng lệnh gọi hệ thống adjtimex( ) :

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

int adjtimex (cấu trúc timex *adj);
```

Một lệnh gọi đến adjtimex() đọc các tham số liên quan đến thời gian của hạt nhân vào cấu trúc timex được trỏ tới bởi adj. Tùy chọn, tùy thuộc vào trường chế độ của cấu trúc này, lệnh gọi hệ thống có thể thiết lập thêm một số tham số nhất định.

Tiêu đề <sys/timex.h> định nghĩa cấu trúc timex như sau:

```
cấu trúc timex {
          int chế độ; /* bộ chọn chế độ */
          bù trừ dài; /* bù trừ thời gian (usec) */
          tần số dài; /* độ lệch tần số (ppm được chia tỷ lệ) */
          long maxerror; /* lỗi tối đa (usec) */
          esterror dài; /* lỗi ước tính (usec) */
          int trạng thái; /* trạng thái đồng hồ */
          hằng số dài; /* Hằng số thời gian PLL */
          độ chính xác dài; /* độ chính xác của đồng hồ (usec) */
          dung sai dài; /* dung sai tần số xung nhịp (ppm) */
          struct timeval time; /* thời gian hiện tại */
          tích tắc dài; /* usecs giữa các tích tắc đồng hồ */
};
```

Trường chế độ là phép toán OR từng bit của không hoặc nhiều cờ sau:

ADJ_BẮT_TẮT
: Thiết lập độ lệch thời gian thông qua offset.

TẦN SỐ ADJ
: Thiết lập độ lệch tần số thông qua freq.

LỖI TỐI ĐA ADJ
: Đặt lỗi tối đa thông qua maxerror.

ADJ_ESTERROR
: Thiết lập lỗi ước tính thông qua esterror.

TRẠNG THÁI_ADJ
: Cài đặt trạng thái đồng hồ thông qua trạng thái.

ADJ_TIMECONST
: Đặt hằng số thời gian của vòng khóa pha (PLL) thông qua hằng số.

ADJ_TICK
: Đặt giá trị tích tắc thông qua tích tắc.

ADJ_OFFSET_SINGLESHOT
: Đặt độ lệch thời gian thông qua offset một lần, bằng thuật toán đơn giản, như adjtime( ).

Nếu chế độ là 0, không có giá trị nào được đặt. Chỉ người dùng có khả năng CAP_SYS_TIME mới có thể cung cấp giá trị chế độ khác không ; bất kỳ người dùng nào cũng có thể cung cấp 0 cho chế độ, truy xuất tất cả tham số nhưng không thiết lập bất kỳ tham số nào.

Khi thành công, adjtimex() trả về trạng thái đồng hồ hiện tại, là một trong những trạng thái sau:

THỜI GIAN_OK
: Đồng hồ đã được đồng bộ.

TIME_INS

Một giây nhuận sẽ đư ợc chèn vào.

TIME_DEL

Một giây nhuận sẽ bị xóa.

TIME_OOP

Một giây nhuận đang diễn ra.

TIME_WAIT

Vừa xảy ra một giây nhuận.

TIME_BAD

Đồng hồ không đư ợc đồng bộ.

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

MẶC ĐỊNH

adj là một con trỏ không hợp lệ.

EINVAL

Một hoặc nhiều chế độ, độ lệch hoặc tích tắc không hợp lệ.

EPERM

chế độ khác không, như ng ngư ời dùng gọi không có khả năng CAP_SYS_TIME .

Lệnh gọi hệ thống adjtimex () là lệnh dành riêng cho Linux. Các ứng dụng liên quan đến tính di động nên ư u tiên adjtime( ).

RFC 1305 định nghĩa một thuật toán phức tạp, do đó thảo luận đầy đủ về adjtimex() nằm ngoài phạm vi của cuốn sách này. Để biết thêm thông tin, hãy xem RFC.

Ngủ và chờ đợi

Nhiều hàm khác nhau cho phép một tiến trình ngủ (tạm dừng thực thi) trong một khoảng thời gian nhất định. Hàm đầu tiên như vậy, sleep( ), đặt tiến trình gọi vào trạng thái ngủ trong số giây đư ợc chỉ định bởi seconds:

```
#include <unistd.h>

unsigned int ngủ (unsigned int giây);
```

Cuộc gọi trả về số giây không ngủ. Do đó, một cuộc gọi thành công trả về 0, như ng hàm có thể trả về các giá trị khác giữa 0 và bao gồm cả giây (nếu, ví dụ, một tín hiệu ngắt quãng giấc ngủ). Hàm không đặt errno. Hầu hết ngư ời dùng sleep() không quan tâm đến thời gian thực sự ngủ của tiến trình và do đó, không kiểm tra giá trị trả về:

```
ngủ (7);          /* ngủ bảy giây */
```

Nếu việc ngủ trong toàn bộ thời gian đư ợc chỉ định thực sự là vấn đề đáng lo ngại, bạn có thể tiếp tục gọi sleep() với giá trị trả về của nó, cho đến khi nó trả về 0:

```
số nguyên không dấu s = 5;

/* ngủ năm giây: không có if, and hoặc buts về nó */ while
((s = ngủ (giây)))
        ;
```

Ngủ với Độ chính xác Microsecond Ngủ với độ chi tiết toàn

giây khá là khập khiễng. Một giây là một khoảng thời gian vô tận trên một hệ thống hiện đại, vì vậy các chư ơ ng trình thư ờng muốn ngủ với độ phân giải dư ới một giây. Nhập usleep( ):

```
/* Phiên bản BSD
*/ #include <unistd.h>

void usleep (unsigned long usec);

/* Phiên bản SUSv2
*/ #define _XOPEN_SOURCE
500 #include <unistd.h>

int usleep (useconds_t usec);
```

Gọi thành công usleep() sẽ đư a tiến trình đang gọi vào trạng thái ngủ trong vài micro giây usec . Thật không may, BSD và Single UNIX Specification không đồng ý về nguyên mẫu của hàm. Biến thể BSD nhận đư ợc một unsigned long và không có giá trị trả về. Tuy nhiên, biến thể SUS định nghĩa usleep() để chấp nhận kiểu useconds_t và trả về một int.
Linux tuân theo SUS nếu _XOPEN_SOURCE đư ợc định nghĩa là 500 hoặc cao hơ n. Nếu _XOPEN_SOURCE không đư ợc định nghĩa hoặc đư ợc đặt thành nhỏ hơ n 500, Linux tuân theo BSD.

Phiên bản SUS trả về 0 khi thành công và -1 khi có lỗi. Các giá trị errno hợp lệ là EINTR, nếu nap bị ngắt bởi tín hiệu hoặc EINVAL, nếu usecs quá lớn (trên Linux, toàn bộ phạm vi của loại là hợp lệ và do đó lỗi này sẽ không bao giờ xảy ra).

Theo thông số kỹ thuật, kiểu useconds_t là một số nguyên không dấu có khả năng lư u trữ các giá trị cao tới 1.000.000.

Do sự khác biệt giữa các nguyên mẫu xung đột và thực tế là một số hệ thống Unix có thể hỗ trợ một trong hai, như ng không hỗ trợ cả hai, nên tốt nhất là không bao giờ đư a kiểu useconds_t vào mã của bạn. Để có khả năng di động tối đa, hãy giả sử tham số là một số nguyên không dấu và không dựa vào giá trị trả về của usleep( ):

```
void usleep (unsigned int usec);
```

Cách sử dụng như sau:

```
số nguyên không dấu usecs = 200;

ngủ (usec);
```

Điều này có thể áp dụng với bất kỳ biến thể nào của hàm và vẫn có thể kiểm tra lỗi:

```
errno = 0;
usleep (1000);
```

```
nếu (errno)
        lỗi ("usleep");
```

Tuy nhiên, hầu hết các chư ơ ng trình không kiểm tra hoặc quan tâm đến lỗi usleep() .

Ngủ với độ phân giải Nano giây

Linux loại bỏ hàm usleep() và thay thế bằng hàm nanosleep( ), cung cấp độ phân giải nano giây và giao diện thông minh hơ n:

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int nanosleep (const struct timespec *req,
               struct timespec *rem);
```

Một lệnh gọi thành công đến nanosleep() sẽ đư a tiến trình gọi vào trạng thái ngủ trong khoảng thời gian đư ợc chỉ định bởi req, sau đó trả về 0. Khi có lỗi, lệnh gọi trả về -1 và đặt errno một cách thích hợp. Nếu một tín hiệu ngắt trạng thái ngủ, lệnh gọi có thể trả về trư ớc khi thời gian đư ợc chỉ định trôi qua. Trong trư ờng hợp đó, nanosleep() trả về -1 và đặt errno thành EINTR. Nếu rem không phải là NULL, hàm sẽ đặt thời gian còn lại để ngủ (lư ợng req chư a ngủ) vào rem. Sau đó, chư ơ ng trình có thể phát hành lại lệnh gọi, truyền rem cho req (như đư ợc trình bày sau trong phần này).

Sau đây là các giá trị errno có thể có khác :

EFAULT

req hoặc rem là con trỏ không hợp lệ.

EINVAL

Một trong các trư ờng trong req không hợp lệ.

Trong trư ờng hợp cơ bản, cách sử dụng rất đơ n giản:

```
cấu trúc timespec req = { .tv_sec =
                          0, .tv_nsec = 200 };

/* ngủ trong 200 ns */
ret = nanosleep (&req, NULL);
nếu (ret)
        perror ("nanosleep");
```

Và đây là ví dụ sử dụng tham số thứ hai để tiếp tục chế độ ngủ nếu bị gián đoạn:

```
cấu trúc timespec req = { .tv_sec =
                          0, .tv_nsec = 1369 };
cấu trúc timespec rem;
int ret;

/* ngủ trong 1369 ns */
thử
lại: ret = nanosleep (&req,
&rem); nếu (ret) {
```

```
            nếu (errno == EINTR) {
                    /* thử lại, với thời gian còn lại được cung cấp */
                    req.tv_sec = rem.tv_sec;
                    req.tv_nsec = rem.tv_nsec;
                    chuyển đến thử lại;

            } perror ("nanosleep");
    }
```

Cuối cùng, đây là một cách tiếp cận thay thế (có lẽ hiệu quả hơn, nhưng khó đọc hơn) hướng tới cùng một mục tiêu:

```
    cấu trúc timespec req = { .tv_sec =
                              1, .tv_nsec =
    0 }; cấu trúc timespec rem, *a = &req, *b = &rem;

    /* ngủ trong 1 giây
    */ trong khi (nanosleep (a, b) && errno == EINTR)
            { struct timespec *tmp = a;
            a = b;
            b = tmp;
    }
```

nanosleep( ) có một số ưu điểm hơn sleep( ) và usleep( ):

- Độ phân giải nano giây, trái ngược với giây hoặc micro giây.
- Được chuẩn hóa bởi POSIX.1b.
- Không được triển khai thông qua tín hiệu (những nhược điểm của nó sẽ được thảo luận sau).

Mặc dù đã bị loại bỏ, nhiều chương trình vẫn thích sử dụng usleep( ) hơn là nanosleep( )— may mắn thay, ít nhất là ngày càng ít ứng dụng sử dụng sleep( ). Vì nanosleep( ) là chuẩn POSIX và không sử dụng tín hiệu, nên các chương trình mới sẽ thích nó (hoặc giao diện được thảo luận trong phần tiếp theo) hơn sleep( ) và usleep( ).

Một cách tiếp cận tiên tiến để ngủ

Giống như tất cả các lớp hàm thời gian mà chúng ta đã nghiên cứu cho đến nay, họ đồng hồ POSIX cung cấp giao diện giấc ngủ tiên tiến nhất:

```
    #include <thời gian.h>

    int clock_nanosleep (clockid_t clock_id, int
                         flags,
                         const struct timespec *req,
                         struct timespec *rem);
```

clock_nanosleep() hoạt động tương tự như nanosleep( ). Trên thực tế, lệnh gọi này:

```
    ret = nanosleep (&req, &rem);
```

giống như lệnh gọi này:

```
    ret = clock_nanosleep (GIỜ THỰC ĐỒNG HỒ, 0, &req, &rem);
```

Sự khác biệt nằm ở các tham số clock_id và flags . Tham số đầu tiên chỉ định nguồn thời gian để đo lư ờng. Hầu hết các nguồn thời gian đều hợp lệ, mặc dù bạn không thể chỉ định CPUclock của quy trình gọi (ví dụ: CLOCK_PROCESS_CPUTIME_ID); làm như vậy sẽ không có ý nghĩa vì lệnh gọi sẽ tạm dừng thực thi quy trình và do đó thời gian quy trình ngừng tăng.

Nguồn thời gian bạn chỉ định phụ thuộc vào mục tiêu ngủ của chư ơ ng trình. Nếu bạn ngủ cho đến một giá trị thời gian tuyệt đối nào đó, CLOCK_REALTIME có thể hợp lý nhất. Nếu bạn ngủ trong một khoảng thời gian tư ơ ng đối, CLOCK_MONOTONIC chắc chắn là nguồn thời gian lý tư ởng.

Tham số cờ là TIMER_ABSTIME hoặc 0. Nếu là TIMER_ABSTIME, giá trị do req chỉ định đư ợc coi là tuyệt đối chứ không phải tư ơ ng đối. Điều này giải quyết đư ợc tình trạng chạy đua tiềm ẩn. Để giải thích giá trị của tham số này, hãy giả sử rằng một tiến trình, tại thời điểm T0, muốn ngủ cho đến thời điểm T1. Tại thời điểm T0, tiến trình gọi clock_gettime() để lấy thời gian hiện tại (T0). Sau đó, nó trừ T0 khỏi T1, lấy Y, rồi truyền cho clock_nanosleep( ).
Tuy nhiên, sẽ có một khoảng thời gian nhất định trôi qua giữa thời điểm lấy đư ợc thời gian và thời điểm quy trình chuyển sang chế độ ngủ. Tệ hơ n nữa, nếu quy trình đư ợc lên lịch ngoài bộ xử lý, gặp lỗi trang hoặc điều gì đó tư ơ ng tự thì sao? Luôn có một tình trạng chạy đua tiềm ẩn giữa việc lấy thời gian hiện tại, tính toán chênh lệch thời gian và thực sự chuyển sang chế độ ngủ.

Cờ TIMER_ABSTIME hủy bỏ cuộc đua bằng cách cho phép một tiến trình chỉ định trực tiếp T1.
Nhân tạm dừng tiến trình cho đến khi nguồn thời gian đư ợc chỉ định đạt đến T1. Nếu thời gian hiện tại của nguồn thời gian đư ợc chỉ định đã vư ợt quá T1, lệnh gọi sẽ trả về ngay lập tức.

Hãy xem xét cả giấc ngủ tư ơ ng đối và tuyệt đối. Ví dụ sau ngủ trong 1,5 giây:

```
cấu trúc timespec ts = { .tv_sec = 1, .tv_nsec = 500000000 }; int ret;


ret = clock_nanosleep (CLOCK_MONOTONIC, 0, &ts, NULL); nếu (ret)
lỗi
    ("clock_nanosleep");
```

Ngư ợc lại, ví dụ sau sẽ ngủ cho đến khi đạt đến giá trị thời gian tuyệt đối—chính xác là một giây so với giá trị mà lệnh gọi clock_gettime() trả về cho nguồn thời gian CLOCK_MONOTONIC— :

```
cấu trúc timespec ts;
int ret;

/* chúng ta muốn ngủ cho đến một giây kể từ BÂY GIỜ */ ret
= clock_gettime (CLOCK_MONOTONIC, &ts); if (ret)
{ perror
         ("clock_gettime"); return;

}
```

```
    ts.tv_sec += 1;
    printf ("Chúng tôi muốn ngủ cho đến giây=%ld nsec=%ld\n",
            ts.tv_sec, ts.tv_nsec);
    ret = clock_nanosleep (CLOCK_MONOTONIC, TIMER_ABSTIME, &ts,
                           NULL);
    nếu
        (ret) perror ("clock_nanosleep");
```

Hầu hết các chư ơ ng trình chỉ cần thời gian ngủ tư ơ ng đối vì nhu cầu ngủ của chúng không quá khắt khe. Tuy nhiên, một số quy trình thời gian thực có yêu cầu về thời gian rất chính xác và cần ngủ hoàn toàn để tránh nguy cơ xảy ra tình trạng chạy đua có khả năng gây ra hậu quả tàn phá.

Một cách di động để ngủ

Nhớ lại từ Chư ơ ng 2 ngư ời bạn của chúng ta select( ):

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

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);
```

Như đã đề cập trong chư ơ ng đó, select( ) cung cấp một cách di động để ngủ với độ phân giải dư ới một giây. Trong một thời gian dài, các chư ơ ng trình Unix di động bị kẹt với sleep( ) cho nhu cầu ngủ trư a của chúng: usleep( ) không đư ợc sử dụng rộng rãi và nanosleep( ) vẫn chư a đư ợc viết. Các nhà phát triển phát hiện ra rằng việc truyền select( ) 0 cho n, NULL cho cả ba con trỏ fd_set và thời lư ợng ngủ mong muốn cho timeout đã tạo ra một cách di động và hiệu quả để đư a các quy trình vào chế độ ngủ:

```
cấu trúc timeval tv = { .tv_sec =
                        0, .tv_usec = 757 };

/* ngủ trong 757 us */
select (0, NULL, NULL, NULL, &tv);
```

Nếu bạn lo ngại về khả năng di chuyển sang các hệ thống Unix cũ hơ n, thì sử dụng select() có thể là lựa chọn tốt nhất.

Vư ợt quá

Tất cả các giao diện đư ợc thảo luận trong phần này đều đảm bảo rằng chúng sẽ ngủ ít nhất là trong thời gian yêu cầu (hoặc trả về lỗi cho biết trư ờng hợp ngư ợc lại). Chúng sẽ không bao giờ trả về thành công nếu không có độ trễ yêu cầu trôi qua. Tuy nhiên, có thể vư ợt qua một khoảng thời gian dài hơ n độ trễ yêu cầu.

Hiện tư ợng này có thể là do hành vi lập lịch đơ n giản—thời gian yêu cầu có thể đã trôi qua và hạt nhân có thể đã đánh thức quy trình đúng giờ, như ng trình lập lịch có thể đã chọn một tác vụ khác để chạy.

Tuy nhiên, có một nguyên nhân nguy hiểm hơ n: tràn bộ đếm thời gian. Điều này xảy ra khi độ chi tiết của bộ đếm thời gian thô hơ n khoảng thời gian yêu cầu. Ví dụ, giả sử bộ đếm thời gian hệ thống tích tắc trong khoảng thời gian 10 ms và một tiến trình yêu cầu ngủ 1 ms.
Hệ thống có thể đo thời gian và phản hồi các sự kiện liên quan đến thời gian (chẳng hạn như đánh thức một tiến trình khỏi chế độ ngủ) chỉ ở các khoảng thời gian 10 ms. Nếu, khi tiến trình đư a ra yêu cầu ngủ, bộ đếm thời gian còn cách một tích tắc 1 ms, mọi thứ sẽ ổn—trong 1 ms, thời gian đư ợc yêu cầu (1 ms) sẽ trôi qua và hạt nhân sẽ đánh thức tiến trình. Tuy nhiên, nếu bộ đếm thời gian chạm đúng vào lúc tiến trình yêu cầu ngủ, sẽ không có tích tắc bộ đếm thời gian nào khác trong 10 ms. Sau đó, tiến trình sẽ ngủ thêm 9 ms! Nghĩa là sẽ có chín lần tràn 1 ms. Trung bình, một bộ đếm thời gian có chu kỳ X có tỷ lệ tràn là X/2.

Việc sử dụng các nguồn thời gian có độ chính xác cao, chẳng hạn như nguồn thời gian do đồng hồ POSIX cung cấp và các giá trị HZ cao hơ n sẽ giảm thiểu tình trạng tràn bộ đếm thời gian.

Các giải pháp thay thế cho chế độ

ngủ Nếu có thể, bạn nên tránh chế độ ngủ. Thông thư ờng, bạn không thể, và điều đó không sao cả—đặc biệt là nếu mã của bạn ngủ trong thời gian ít hơ n một giây. Tuy nhiên, mã đư ợc kết hợp với chế độ ngủ để "bận chờ" các sự kiện thư ờng là thiết kế kém. Mã chặn trên một mô tả tệp, cho phép hạt nhân xử lý chế độ ngủ và đánh thức quy trình, thì tốt hơ n. Thay vì quy trình quay vòng trong một vòng lặp cho đến khi sự kiện xảy ra, hạt nhân có thể chặn quy trình khỏi thực thi và chỉ đánh thức quy trình khi cần.

Bộ đếm thời gian

Bộ đếm thời gian cung cấp một cơ chế để thông báo cho một tiến trình khi một khoảng thời gian nhất định trôi qua. Khoảng thời gian trư ớc khi bộ đếm thời gian hết hạn đư ợc gọi là độ trễ hoặc thời gian hết hạn. Cách hạt nhân thông báo cho tiến trình rằng bộ đếm thời gian đã hết hạn phụ thuộc vào bộ đếm thời gian. Hạt nhân Linux cung cấp một số loại. Chúng ta sẽ nghiên cứu tất cả chúng.

Bộ hẹn giờ hữu ích vì nhiều lý do. Ví dụ bao gồm làm mới màn hình 60 lần mỗi giây hoặc hủy giao dịch đang chờ xử lý nếu giao dịch vẫn đang diễn ra sau 500 mili giây.

Báo thức đơ n giản

alarm() là giao diện hẹn giờ đơ n giản nhất:

```
#include <unistd.h>

báo động unsigned int (unsigned int giây);
```

Một lệnh gọi đến hàm này sẽ lên lịch gửi tín hiệu SIGALRM đến tiến trình gọi sau khi đã trôi qua nhiều giây thời gian thực. Nếu tín hiệu đã lên lịch trước đó đang chờ xử lý, lệnh gọi sẽ hủy báo thức, thay thế bằng báo thức mới được yêu cầu và trả về số giây còn lại trong báo thức trước đó. Nếu giây là 0, báo thức trước đó, nếu có, sẽ bị hủy, nhưng không có báo thức mới nào được lên lịch.

Do đó, để sử dụng thành công chức năng này cũng cần phải đăng ký trình xử lý tín hiệu cho tín hiệu SIGALRM . (Tín hiệu và trình xử lý tín hiệu đã được đề cập trong chương trước.)
Sau đây là đoạn mã đăng ký trình xử lý SIGALRM , alarm_handler( ) và đặt báo thức trong năm giây:

```
void báo động_xử lý (int signum) {

          printf ("Đã trôi qua năm giây!\n");
}

hàm void (void) {

          tín hiệu (SIGALRM, trình xử lý báo động);
          báo động (5);

          tạm dừng ( );
}
```

Bộ đếm thời gian

Các cuộc gọi hệ thống hẹn giờ khoảng thời gian , lần đầu tiên xuất hiện trong 4.2BSD, kể từ đó đã được chuẩn hóa trong POSIX và cung cấp nhiều khả năng kiểm soát hơn alarm( ):

```
#include <hệ thống/thời gian.h>

int getitimer (int mà,
                  cấu trúc itimerval *giá trị);

int setitimer (int mà,
                  const struct itimerval *giá trị,
                  struct itimerval *ovalue);
```

Bộ hẹn giờ khoảng thời gian hoạt động giống như báo thức ( ), nhưng tùy chọn có thể tự động kích hoạt lại và hoạt động ở một trong ba chế độ riêng biệt:

ITIMER_THỰC TẾ
: Đo thời gian thực. Khi lượng thời gian thực đã chỉ định trôi qua, ker-nel sẽ gửi cho quy trình tín hiệu SIGALRM .

ITIMER_VIRTUAL Chỉ
: giảm trong khi mã không gian người dùng của quy trình đang thực thi. Khi lượng thời gian quy trình được chỉ định đã trôi qua, hạt nhân gửi cho quy trình một SIGVTALRM.

ITIMER_PROF

Giảm cả trong khi tiến trình đang thực thi và trong khi hạt nhân đang thực thi thay mặt cho tiến trình (ví dụ, hoàn tất lệnh gọi hệ thống). Khi khoảng thời gian đư ợc chỉ định đã trôi qua, hạt nhân sẽ gửi cho tiến trình một tín hiệu SIGPROF . Chế độ này thư ờng đư ợc kết hợp với ITIMER_VIRTUAL, để chư ơ ng trình có thể đo thời gian ngư ời dùng và hạt nhân mà tiến trình đã sử dụng.

ITIMER_REAL đo cùng thời gian với alarm( ); hai chế độ còn lại hữu ích cho việc lập hồ sơ .

Cấu trúc itimerval cho phép ngư ời dùng chỉ định khoảng thời gian cho đến khi bộ đếm thời gian hết hạn, cũng như ngày hết hạn (nếu có) để kích hoạt lại bộ đếm thời gian khi bộ đếm thời gian hết hạn:

```
struct itimerval { struct
          timeval it_interval; /* giá trị tiếp theo */ struct timeval
          it_value; /* giá trị hiện tại */
};
```

Nhớ lại từ trư ớc rằng cấu trúc timeval cung cấp độ phân giải micro giây:

```
struct timeval
          { long tv_sec; /* giây */ long
          tv_usec; /* micro giây */
};
```

setitimer() cung cấp một bộ đếm thời gian có kiểu thời gian hết hạn đư ợc chỉ định bởi it_value. Sau khi thời gian đư ợc chỉ định bởi it_value trôi qua, hạt nhân sẽ khởi động lại bộ đếm thời gian với thời gian đư ợc cung cấp bởi it_interval. Do đó, it_value là thời gian còn lại trên bộ đếm thời gian hiện tại. Khi it_value đạt đến số không, nó đư ợc đặt thành it_interval. Nếu bộ đếm thời gian hết hạn và it_interval bằng 0, bộ đếm thời gian sẽ không đư ợc khởi động lại. Tư ơ ng tự, nếu it_value của bộ đếm thời gian đang hoạt động đư ợc đặt thành 0, bộ đếm thời gian sẽ dừng lại và không đư ợc khởi động lại.

Nếu ovalue không phải là NULL, các giá trị trư ớc đó cho bộ đếm thời gian theo kiểu sẽ đư ợc trả về.

getitimer() trả về giá trị hiện tại cho bộ đếm thời gian theo kiểu which.

Cả hai hàm đều trả về 0 nếu thành công và -1 nếu có lỗi, trong trư ờng hợp đó errno đư ợc đặt thành một trong những giá trị sau:

Giá trị

EFAULT hoặc giá trị ovalue là con trỏ không hợp lệ.

EINVAL

không phải là loại bộ đếm thời gian hợp lệ.

Đoạn mã sau đây tạo trình xử lý tín hiệu SIGALRM (một lần nữa, hãy xem Chư ơ ng 9), sau đó kích hoạt bộ đếm thời gian với thời gian hết hạn ban đầu là năm giây, theo sau là khoảng thời gian tiếp theo là một giây: void alarm_handler (int signo) {

```
          printf ("Hẹn giờ đã xong!\n");
```

```
}

void foo (void)
        { struct itimerval
        delay; int ret;

        tín hiệu (SIGALRM, trình xử lý báo động);

        delay.it_value.tv_sec = 5;
        delay.it_value.tv_usec = 0;
        delay.it_interval.tv_sec = 1;
        delay.it_interval.tv_usec = 0; ret
        = setitimer (ITIMER_REAL, &delay, NULL); nếu (ret)
        { lỗi
              ("setitimer"); trả về;

        }

        tạm dừng ( );
}
```

Một số hệ thống Unix triển khai sleep( ) và usleep( ) thông qua SIGALRM—và, rõ ràng là, alarm( ) và setitimer( ) sử dụng SIGALRM. Do đó, các lập trình viên phải cẩn thận không chồng chéo các lệnh gọi đến các hàm này; kết quả không đư ợc xác định. Với mục đích chờ đợi ngắn, các lập trình viên nên sử dụng nanosleep( ), mà POSIX quy định sẽ không sử dụng tín hiệu. Đối với bộ đếm thời gian, các lập trình viên nên sử dụng setitimer( ) hoặc alarm( ).

Bộ đếm thời gian nâng cao

Không có gì ngạc nhiên khi giao diện hẹn giờ mạnh mẽ nhất đến từ họ đồng hồ POSIX.

Với bộ hẹn giờ dựa trên đồng hồ POSIX, các hành động khởi tạo, khởi tạo và cuối cùng là xóa bộ hẹn giờ đư ợc tách thành ba hàm khác nhau: timer_create() tạo bộ hẹn giờ, timer_settime() khởi tạo bộ hẹn giờ và timer_delete() hủy bộ hẹn giờ.

Họ đồng hồ POSIX của giao diện hẹn giờ chắc chắn là tiên tiến nhất, như ng cũng là mới nhất (do đó ít di động nhất) và phức tạp nhất để sử dụng. Nếu tính đơ n giản hoặc tính di động là động lực chính, thì setitimer() có khả năng là lựa chọn tốt hơ n.

Tạo bộ đếm thời gian

Để tạo bộ đếm thời gian, hãy sử dụng timer_create( ):

```
#include <tín hiệu.h>
#include <thời gian.h>

int timer_create (clockid_t clockid,
                  struct sigevent *evp,
                  timer_t *timerid);
```

Một lệnh gọi thành công đến timer_create( ) sẽ tạo một bộ đếm thời gian mới được liên kết với đồng hồ POSIX clockid , lưu trữ một mã định danh bộ đếm thời gian duy nhất trong timerid và trả về 0. Lệnh gọi này chỉ thiết lập các điều kiện để chạy bộ đếm thời gian; thực tế không có gì xảy ra cho đến khi bộ đếm thời gian được kích hoạt, như được hiển thị trong phần sau.

Ví dụ sau đây tạo một bộ đếm thời gian mới lấy từ đồng hồ POSIX CLOCK_PROCESS_CPUTIME_ID và lưu trữ ID của bộ đếm thời gian trong timer:

```
timer_t bộ đếm thời
gian; int ret;

ret = timer_create (CLOCK_PROCESS_CPUTIME_ID,
                    NULL,
                    &timer);
nếu (ret)
        lỗi ("timer_create");
```

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

LẠI LẦN NỮA

Hệ thống không đủ tài nguyên để hoàn tất yêu cầu.

EINVAL

Đồng hồ POSIX được chỉ định bởi clockid không hợp lệ.

ENOTSUP

Đồng hồ POSIX được chỉ định bởi clockid là hợp lệ, nhưng hệ thống không hỗ trợ sử dụng đồng hồ cho bộ đếm thời gian. POSIX đảm bảo rằng tất cả các triển khai đều hỗ trợ đồng hồ CLOCK_REALTIME cho bộ đếm thời gian. Việc các đồng hồ khác có được hỗ trợ hay không tùy thuộc vào triển khai.

Tham số evp , nếu không phải NULL, sẽ định nghĩa thông báo không đồng bộ xảy ra khi bộ đếm thời gian hết hạn. Tiêu đề <signal.h> định nghĩa cấu trúc. Nội dung của nó được cho là không rõ ràng đối với lập trình viên, nhưng nó có ít nhất các trường sau:

```
#include <tín hiệu.h>

struct sigevent
        { hợp nhất sigval giá trị
        sigev; int
        sigev_signo; int
        sigev_notify; void (*sigev_notify_function)(hợp
        nhất sigval); pthread_attr_t *sigev_notify_attributes;
};

hợp nhất sigval
        { int sival_int;
        void *sival_ptr;
};
```

Bộ đếm thời gian dựa trên đồng hồ POSIX cho phép kiểm soát tốt hơn nhiều cách mà hạt nhân thông báo cho quy trình khi bộ đếm thời gian hết hạn, cho phép quy trình chỉ định chính xác tín hiệu nào mà hạt nhân sẽ phát ra hoặc thậm chí cho phép hạt nhân tạo ra một luồng và thực hiện một hàm để phản hồi khi bộ đếm thời gian hết hạn. Một quy trình chỉ định hành vi khi bộ đếm thời gian hết hạn thông qua sigev_notify, phải là một trong ba giá trị sau:

SIGEV_NONE

Thông báo "null". Khi hết thời gian, không có gì xảy ra.

SIGEV_SIGNAL

Khi hết thời gian, hạt nhân gửi cho tiến trình tín hiệu được chỉ định bởi sigev_signo. Trong trình xử lý tín hiệu, si_value được đặt thành sigev_value.

SIGEV_THREAD Khi

hết thời gian hẹn giờ, hạt nhân tạo ra một luồng mới (trong tiến trình này) và yêu cầu luồng đó thực thi sigev_notify_function, truyền sigev_value làm đối số duy nhất. Luồng kết thúc khi nó trả về từ hàm này. Nếu sigev_notify_attributes không phải là NULL, cấu trúc pthread_attr_t được cung cấp sẽ xác định hành vi của luồng mới.

Nếu evp là NULL, như trong ví dụ trước đó của chúng tôi, thông báo hết hạn của bộ đếm thời gian được thiết lập như thể sigev_notify là SIGEV_SIGNAL, sigev_signo là SIGALRM và sigev_value là ID của bộ đếm thời gian. Do đó, theo mặc định, các bộ đếm thời gian này thông báo theo cách tương tự như bộ đếm thời gian khoảng thời gian POSIX. Tuy nhiên, thông qua tùy chỉnh, chúng có thể làm được nhiều hơn thế nữa!

Ví dụ sau đây tạo một bộ đếm thời gian khóa CLOCK_REALTIME. Khi bộ đếm thời gian hết hạn, hạt nhân sẽ phát tín hiệu SIGUSR1 và đặt si_value thành địa chỉ lưu trữ ID của bộ đếm thời gian:

```
cấu trúc sigevent evp;
timer_t bộ đếm
thời gian; int ret;

evp.sigev_value.sival_ptr = &timer;
evp.sigev_notify = TÍN HIỆU SIGEV;
evp.sigev_signo = SIGUSR1; ret =
timer_create (THỜI GIAN THỰC, &evp, &timer);


nếu (ret)
          lỗi ("timer_create");
```

Kích hoạt bộ đếm

thời gian Bộ đếm thời gian được tạo bởi timer_create() không được kích hoạt. Để liên kết nó với thời gian hết hạn và bắt đầu đếm thời gian, hãy sử dụng timer_settime( ):

```
#include <thời gian.h>

int timer_settime (timer_t timerid, int flags,
                           const struct
                           itimerspec *giá trị, struct itimerspec
                           *ovalue);
```

Một lệnh gọi thành công đến timer_settime() sẽ kích hoạt bộ đếm thời gian được chỉ định bởi timerid với giá trị hết hạn, đây là một cấu trúc itimerspec :

```
struct itimerspec { struct
          timespec it_interval; /* giá trị tiếp theo */ struct timespec
          it_value; /* giá trị hiện tại */
};
```

Tương tự như setitimer( ), it_value chỉ định thời gian hết hạn của bộ đếm thời gian hiện tại. Khi bộ đếm thời gian hết hạn, it_value được làm mới với giá trị từ it_interval. Nếu it_interval bằng 0, bộ đếm thời gian không phải là bộ đếm thời gian theo khoảng thời gian và sẽ vô hiệu hóa khi it_value hết hạn.

Nhớ lại từ trước rằng cấu trúc timespec cung cấp độ phân giải nano giây:

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

Nếu flags là TIMER_ABSTIME, thời gian được chỉ định bởi giá trị sẽ được hiểu là tuyệt đối (trái ngược với cách hiểu mặc định, trong đó giá trị là tương đối so với thời gian hiện tại).
Hành vi được sửa đổi này ngăn chặn tình trạng chạy đua trong các bước lấy thời gian hiện tại, tính toán sự khác biệt tương đối giữa thời gian đó và thời gian mong muốn trong tương lai và kích hoạt bộ hẹn giờ. Xem phần thảo luận trong phần trước, “Phương pháp tiếp cận nâng cao để ngủ” để biết chi tiết.

Nếu ovalue không phải là NULL, thời gian hết hạn của bộ đếm thời gian trước đó được lưu trong itimerspec đã cung cấp. Nếu bộ đếm thời gian đã bị vô hiệu hóa trước đó, tất cả các thành viên của cấu trúc đều được thiết lập đến 0.

Sử dụng giá trị bộ đếm thời gian được khởi tạo trước đó bởi timer_create( ), ví dụ sau sẽ tạo một bộ đếm thời gian định kỳ hết hạn sau mỗi giây:

```
cấu trúc itimerspec ts;
int ret;

ts.it_interval.tv_sec = 1;
ts.it_interval.tv_nsec = 0;
ts.it_value.tv_sec = 1;
ts.it_value.tv_nsec = 0;

ret = time_settime (bộ đếm thời gian, 0, &ts, NULL);
if (ret)
         perror ("timer_settime");
```

Lấy ngày hết hạn của bộ đếm thời gian

Bạn có thể lấy thời gian hết hạn của bộ đếm thời gian mà không cần thiết lập lại nó, bất cứ lúc nào, thông qua timer_gettime( ):

```
#include <thời gian.h>

int timer_gettime (timer_t timerid, struct
                          itimerspec *giá trị);
```

Một lệnh gọi thành công đến timer_gettime( ) sẽ lưu trữ thời gian hết hạn của bộ đếm thời gian được chỉ định bởi timerid trong cấu trúc được trỏ tới bởi giá trị và trả về 0. Khi lệnh gọi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong những giá trị sau:

MẶC ĐỊNH

giá trị là một con trỏ không hợp lệ.

EINVAL

timerid là bộ đếm thời gian không hợp lệ.

Ví dụ:

```
cấu trúc itimerspec ts;
int ret;

ret = time_gettime(bộ đếm thời gian,&ts);
if (ret)
        perror ("timer_gettime");
khác {
         printf ("giây hiện tại=%ld nsec=%ld\n",
                  ts.it_value.tv_sec, ts.it_value.tv_nsec); printf
         ("giây tiếp theo=%ld nsec=%ld\n",
                  ts.it_interval.tv_sec, ts.it_interval.tv_nsec);
}
```

Đạt được sự vượt quá của bộ đếm thời gian

POSIX định nghĩa một giao diện để xác định có bao nhiêu lần tràn bộ nhớ, nếu có, xảy ra trên một bộ đếm thời gian nhất định:

```
#include <thời gian.h>

int time_getoverrun (timer_t timerid);
```

Khi thành công, timer_getoverrun() trả về số lần hết hạn bộ đếm thời gian bổ sung đã xảy ra giữa thời điểm hết hạn ban đầu của bộ đếm thời gian và thông báo cho quy trình—ví dụ, thông qua tín hiệu—rằng bộ đếm thời gian đã hết hạn. Ví dụ, trong ví dụ trước đó của chúng tôi, khi bộ đếm thời gian 1 ms chạy trong 10 ms, lệnh gọi sẽ trả về 9.

Nếu số lần vượt quá bằng hoặc lớn hơn DELAYTIMER_MAX, lệnh gọi sẽ trả về DELAYTIMER_MAX.

Khi lỗi, hàm trả về -1 và đặt errno thành EINVAL, điều kiện lỗi duy nhất, biểu thị rằng bộ đếm thời gian được chỉ định bởi timerid không hợp lệ.

Ví dụ:

```
int ret;

ret=timer_getoverrun (bộ đếm thời gian);
if (ret == -1)
          perror ("timer_getoverrun"); else
if (ret == 0) printf
          ("không bị tràn\n");
khác
          printf ("%d lần vượt quá\n", ret);
```

Xóa bộ hẹn giờ

Xóa bộ hẹn giờ rất dễ dàng:

```
#include <thời gian.h>

int timer_delete (timer_t timerid);
```

Một lệnh gọi thành công đến timer_delete( ) sẽ hủy bộ đếm thời gian được liên kết với timerid và trả về 0. Khi lệnh gọi thất bại, lệnh gọi trả về -1 và errno được đặt thành EINVAL, điều kiện lỗi duy nhất, biểu thị rằng timerid không phải là bộ đếm thời gian hợp lệ.

Phần mở rộng GCC cho ngôn ngữ C

GNUCompiler Collection (GCC) cung cấp nhiều tiện ích mở rộng cho ngôn ngữ C, một số trong đó đã được chứng minh là có giá trị đặc biệt đối với các lập trình viên hệ thống. Phần lớn các tiện ích bổ sung cho ngôn ngữ C mà chúng tôi sẽ đề cập trong phần phụ lục này cung cấp các cách để các lập trình viên cung cấp thêm thông tin cho trình biên dịch về hành vi và mục đích sử dụng dự định của mã của họ. Đến lượt mình, trình biên dịch sử dụng thông tin này để tạo mã máy hiệu quả hơn. Các tiện ích mở rộng khác lấp đầy khoảng trống trong ngôn ngữ lập trình C, đặc biệt là ở các cấp thấp hơn.

GCC cung cấp một số tiện ích mở rộng hiện có trong tiêu chuẩn C mới nhất, ISO C99. Một số tiện ích mở rộng này hoạt động tương tự như các tiện ích mở rộng C99 của chúng, nhưng ISO C99 triển khai các tiện ích mở rộng khác khá khác biệt. Mã mới nên sử dụng các biến thể ISO C99 của các tính năng này. Chúng tôi sẽ không đề cập đến các tiện ích mở rộng như vậy ở đây; chúng tôi sẽ chỉ thảo luận về các bổ sung duy nhất của GCC.

GNUC

Hương vị của C được GCC hỗ trợ thường được gọi là GNUC. Vào những năm 1990, GNUC đã lấp đầy một số khoảng trống trong ngôn ngữ C, cung cấp các tính năng như biến phức tạp, mảng có độ dài bằng không, hàm nội tuyến và trình khởi tạo có tên. Nhưng sau gần một thập kỷ, C cuối cùng đã được nâng cấp và với việc chuẩn hóa ISO C99, các tiện ích mở rộng GNUC trở nên ít liên quan hơn. Tuy nhiên, GNUC vẫn tiếp tục cung cấp các tính năng hữu ích và nhiều lập trình viên Linux vẫn sử dụng một tập hợp con của GNUC—thường chỉ là một hoặc hai tiện ích mở rộng—trong mã tương thích với C90 hoặc C99 của họ.

Một ví dụ nổi bật về cơ sở mã dành riêng cho GCC là hạt nhân Linux, được viết nghiêm ngặt bằng GNUC. Tuy nhiên, gần đây, Intel đã đầu tư công sức kỹ thuật để cho phép Intel C Compiler (ICC) hiểu được các phần mở rộng GNUC được hạt nhân sử dụng. Do đó, nhiều phần mở rộng này hiện đang ít dành riêng cho GCC hơn.

Hàm nội tuyến

Trình biên dịch sao chép toàn bộ mã của một hàm "nội tuyến" vào vị trí mà hàm đư ợc gọi. Thay vì lư u trữ hàm bên ngoài và nhảy đến hàm đó bất cứ khi nào hàm đư ợc gọi, trình biên dịch sẽ chạy trực tiếp nội dung của hàm. Hành vi như vậy giúp tiết kiệm chi phí gọi hàm và cho phép tối ư u hóa tiềm năng tại vị trí gọi vì trình biên dịch có thể tối ư u hóa cả ngư ời gọi và ngư ời đư ợc gọi cùng nhau. Điểm sau này đặc biệt hợp lệ nếu các tham số của hàm là hằng số tại vị trí gọi. Tuy nhiên, theo tự nhiên, việc sao chép một hàm vào từng đoạn mã gọi hàm đó có thể ảnh hư ởng bất lợi đến kích thư ớc mã. Do đó, các hàm chỉ nên đư ợc nội tuyến nếu chúng nhỏ và đơ n giản hoặc không đư ợc gọi ở nhiều nơ i khác nhau.

Trong nhiều năm, GCC đã hỗ trợ từ khóa inline , hư ớng dẫn trình biên dịch nhúng hàm đã cho. C99 đã chính thức hóa từ khóa này:

```
tĩnh nội tuyến int foo (void) { /* ... */ }
```

Tuy nhiên, về mặt kỹ thuật, từ khóa chỉ là một gợi ý—một gợi ý cho trình biên dịch để xem xét nhúng hàm đã cho. GCC cung cấp thêm một phần mở rộng để hư ớng dẫn trình biên dịch luôn nhúng hàm đư ợc chỉ định:

```
tĩnh nội tuyến     _ _thuộc tính_ _ ((luôn_nội_dòng)) int foo (void) { /* ... */ }
```

Ứng cử viên rõ ràng nhất cho một hàm nội tuyến là một macro tiền xử lý. Một hàm nội tuyến trong GCC sẽ hoạt động tốt như một macro và ngoài ra, còn nhận đư ợc kiểm tra kiểu. Ví dụ, thay vì macro này:

```
#xác định max(a,b) ({ a > b ? a : b; })
```

ngư ời ta có thể sử dụng hàm nội tuyến tư ơ ng ứng:

```
tĩnh nội tuyến max (int a, int b) {

          nếu (a >
                   b) trả về
          a; trả về b;
}
```

Các lập trình viên có xu hư ớng sử dụng quá nhiều hàm nội tuyến. Chi phí gọi hàm trên hầu hết các kiến trúc hiện đại—đặc biệt là x86—rất, rất thấp. Chỉ những hàm xứng đáng nhất mới đư ợc xem xét!

Ngăn chặn nội tuyến

Ở chế độ tối ư u hóa mạnh mẽ nhất, GCC tự động chọn các hàm có vẻ phù hợp để nhúng và nhúng chúng. Đây thư ờng là một ý tư ởng hay, như ng đôi khi lập trình viên biết rằng một hàm sẽ hoạt động không chính xác nếu nhúng. Một ví dụ khả thi về điều này là khi sử dụng _ _builtin_return_address (sẽ thảo luận sau trong phần phụ lục này). Để ngăn chặn nhúng, hãy sử dụng từ khóa noinline :

```
_ _thuộc tính_ _ ((noinline)) int foo (void) { /* ... */ }
```

Hàm thuần túy

Một hàm "thuần túy" là hàm không có hiệu ứng nào và giá trị trả về chỉ phản ánh các tham số hoặc biến toàn cục không biến động của hàm. Bất kỳ tham số hoặc quyền truy cập biến toàn cục nào cũng phải là chỉ đọc. Tối ưu hóa vòng lặp và loại bỏ biểu thức con có thể được áp dụng cho các hàm như vậy. Các hàm được đánh dấu là thuần túy thông qua từ khóa thuần túy :

```
__thuộc tính_ _ ((thuần túy)) int foo (int val) { /* ... */ }
```

Một ví dụ phổ biến là strlen( ). Với các đầu vào giống hệt nhau, giá trị trả về của hàm này là bất biến qua nhiều lần gọi, và do đó, nó có thể được kéo ra khỏi vòng lặp và chỉ được gọi một lần. Ví dụ, hãy xem xét đoạn mã sau:

```
/* từng ký tự, in ra từng chữ cái trong 'p' bằng chữ in hoa */ for (i = 0; i
< strlen(p); i++) printf ("%c",
          toupper (p[i]));
```

Nếu trình biên dịch không biết rằng strlen() là thuần túy, nó có thể gọi hàm này ở mỗi lần lặp của vòng lặp!

Các lập trình viên thông minh cũng như trình biên dịch, nếu strlen( ) được đánh dấu là pure, sẽ viết hoặc tạo mã như thế này:

```
kích thước_t chiều dài;

len = strlen (p);
đối với (i = 0; i < len; i++)
          printf ("%c", trên cùng (p[i]));
```

Ngoài ra, ngay cả những lập trình viên thông minh hơn (như độc giả của cuốn sách này) cũng sẽ viết:

```
trong khi
          (*p) printf ("%c", toupper (*p++));
```

Thật bất hợp pháp và vô nghĩa khi một hàm thuần túy trả về giá trị void, vì giá trị trả về là mục đích duy nhất của các hàm như vậy.

Các hàm hằng số

Một hàm "hằng số" là một biến thể nghiêm ngặt hơn của một hàm thuần túy. Các hàm như vậy không thể truy cập các biến toàn cục và không thể lấy các con trỏ làm tham số. Do đó, giá trị trả về của hàm hằng số không phản ánh gì ngoài các tham số được truyền theo giá trị. Các tối ưu hóa bổ sung, ngoài các tối ưu hóa có thể có với các hàm thuần túy, cũng có thể thực hiện được đối với các hàm như vậy. Các hàm toán học, chẳng hạn như abs( ), là các ví dụ về các hàm hằng số (giả sử chúng không lưu trạng thái hoặc thực hiện các thủ thuật khác dưới danh nghĩa tối ưu hóa). Một lập trình viên đánh dấu một hàm hằng số thông qua từ khóa const :

```
_thuộc tính_ _    ((const)) int foo (int val) { /* ... */ } _
```

Giống như các hàm thuần túy, một hàm hằng số không thể trả về giá trị void.

Các hàm không trả về

Nếu một hàm không trả về—có lẽ vì nó luôn gọi exit( )—lập trình viên có thể đánh dấu hàm bằng từ khóa noreturn , giúp trình biên dịch hiểu rõ điều đó:

```
tính_ _          ((noreturn)) void foo (int val) { /* ... */ } _ _thuộc
```

Đổi lại, trình biên dịch có thể thực hiện các tối ưu hóa bổ sung, với sự hiểu biết rằng trong mọi trường hợp, hàm được gọi sẽ không bao giờ trả về. Không có ý nghĩa gì khi một hàm như vậy trả về bất kỳ giá trị nào ngoài void.

Các hàm phân bổ bộ nhớ

Nếu một hàm trả về một con trỏ không bao giờ có thể đặt bí danh* cho bộ nhớ hiện có—gần như chắc chắn vì hàm vừa phân bổ bộ nhớ mới và trả về một con trỏ tới bộ nhớ đó—lập trình viên có thể đánh dấu hàm như vậy bằng từ khóa malloc và trình biên dịch có thể thực hiện các tối ưu hóa phù hợp:

```
_ _thuộc tính_ _ ((malloc)) void * get_page (void) {

          int kích thước trang;

          page_size = getpagesize (); nếu
          (page_size <= 0) trả
                    về NULL;

          trả về malloc (page_size);
}
```

Buộc người gọi kiểm tra giá trị trả về

Không phải là một tối ưu hóa mà là một công cụ hỗ trợ lập trình, thuộc tính warn_unused_result hướng dẫn trình biên dịch tạo cảnh báo bất cứ khi nào giá trị trả về của một hàm không được lưu trữ hoặc sử dụng trong một câu lệnh có điều kiện:

```
_ _thuộc tính_ _ ((cảnh báo_kết_quả_chưa_sử_dụng)) int foo (void) { /* ... */ }
```

Điều này cho phép lập trình viên đảm bảo rằng tất cả người gọi kiểm tra và xử lý giá trị trả về từ một hàm mà giá trị có tầm quan trọng đặc biệt. Các hàm có giá trị trả về quan trọng nhưng thường bị bỏ qua, chẳng hạn như read( ), là ứng cử viên tuyệt vời cho thuộc tính này. Các hàm như vậy không thể trả về void.

* Một bí danh bộ nhớ xảy ra khi hai hoặc nhiều biến con trỏ trỏ đến cùng một địa chỉ bộ nhớ. Điều này có thể xảy ra trong những trường hợp tầm thường khi một con trỏ được gán giá trị của một con trỏ khác và cũng có thể xảy ra trong những trường hợp phức tạp hơn, ít rõ ràng hơn. Nếu một hàm trả về địa chỉ của bộ nhớ mới được phân bổ, thì không nên có bất kỳ con trỏ nào khác đến cùng địa chỉ đó.

Đánh dấu các hàm là đã lỗi thời

Thuộc tính đã lỗi thời hư ớng dẫn trình biên dịch tạo cảnh báo tại vị trí gọi bất cứ khi nào hàm đư ợc gọi:

```
_ _thuộc tính_ _ ((đã lỗi thời)) void foo (void) { /* ... */ }
```

Điều này giúp các lập trình viên tránh xa các giao diện lỗi thời và không còn đư ợc sử dụng nữa.

Đánh dấu các chức năng như đã sử dụng

Thỉnh thoảng, không có mã nào hiển thị cho trình biên dịch gọi một hàm cụ thể. Đánh dấu một hàm bằng thuộc tính used sẽ hư ớng dẫn trình biên dịch rằng chư ơ ng trình sử dụng hàm đó, mặc dù hàm đó có vẻ như không bao giờ đư ợc tham chiếu:

```
tĩnh _ _thuộc tính_ _ ((đã sử dụng)) void foo (void) { /* ... */ }
```

Do đó, trình biên dịch sẽ xuất ra ngôn ngữ lắp ráp kết quả và không hiển thị cảnh báo về hàm không sử dụng. Thuộc tính này hữu ích nếu hàm tĩnh chỉ đư ợc gọi từ mã lắp ráp viết tay. Thông thư ờng, nếu trình biên dịch không biết về bất kỳ lệnh gọi nào, nó sẽ tạo cảnh báo và có khả năng tối ư u hóa hàm.

Đánh dấu các hàm hoặc tham số là không sử dụng

Thuộc tính unused cho trình biên dịch biết rằng hàm hoặc tham số hàm đã cho không đư ợc sử dụng và yêu cầu trình biên dịch không đư a ra bất kỳ cảnh báo tư ơ ng ứng nào: int foo (long

```
((unused)) value) { /* nếu bạn */ } _ _attribute_ _ Thuộc tính này hữu
```

ích

đang biên dịch với -W hoặc -Wunused và bạn muốn bắt các tham số hàm không đư ợc sử dụng, như ng đôi khi bạn có các hàm phải khớp với chữ ký đư ợc khai báo trư ớc (như thư ờng thấy trong lập trình GUI theo sự kiện hoặc trình xử lý tín hiệu).

Đóng gói một cấu trúc

Thuộc tính packed cho trình biên dịch biết rằng một kiểu hoặc biến nên đư ợc đóng gói vào bộ nhớ bằng cách sử dụng lư ợng không gian tối thiểu có thể, có khả năng bỏ qua các yêu cầu căn chỉnh. Nếu đư ợc chỉ định trên một struct hoặc union, tất cả các biến trong đó đều đư ợc đóng gói như vậy. Nếu chỉ định trên một biến, chỉ đối tư ợng cụ thể đó đư ợc đóng gói.

Sau đây là cách đóng gói tất cả các biến trong cấu trúc vào một không gian tối thiểu:

```
cấu trúc _ _thuộc tính_ _ ((đóng gói)) foo { ... };
```

Ví dụ, một cấu trúc chứa char theo sau là int rất có thể sẽ tìm thấy số nguyên được căn chỉnh theo một địa chỉ bộ nhớ không theo ngay char, nhưng, giả sử, ba byte sau đó. Trình biên dịch căn chỉnh các biến bằng cách chèn các byte đệm chưa sử dụng vào giữa chúng. Một cấu trúc đóng gói thiếu phần đệm này, có khả năng tiêu thụ ít bộ nhớ hơn, nhưng không đáp ứng được các yêu cầu căn chỉnh kiến trúc.

Tăng sự liên kết của một biến

Ngoài việc cho phép đóng gói các biến, GCC còn cho phép các lập trình viên chỉ định một căn chỉnh tối thiểu thay thế cho một biến nhất định. Sau đó, GCC sẽ căn chỉnh biến được chỉ định theo ít nhất giá trị này, trái ngược với căn chỉnh tối thiểu bắt buộc do kiến trúc và ABI chỉ định. Ví dụ, câu lệnh này khai báo một số nguyên có tên là beard_length với căn chỉnh tối thiểu là 32 byte (trái ngược với căn chỉnh thông thường là 4 byte trên các máy có số nguyên 32 bit):

```
int beard_length  _ _attribute_ _ ((aligned (32))) = 0;
```

Việc buộc căn chỉnh một kiểu thường chỉ hữu ích khi xử lý phần cứng có thể áp đặt các yêu cầu căn chỉnh lớn hơn so với chính kiến trúc hoặc khi bạn đang trộn thủ công mã C và mã assembly và bạn muốn sử dụng các lệnh yêu cầu các giá trị căn chỉnh đặc biệt. Một ví dụ trong đó chức năng căn chỉnh này được sử dụng là để lưu trữ các biến thường dùng trên các dòng bộ đệm bộ xử lý để tối ưu hóa hành vi bộ đệm. Nhân Linux sử dụng kỹ thuật này.

Thay vì chỉ định một căn chỉnh tối thiểu nhất định, bạn có thể yêu cầu GCC căn chỉnh một kiểu dữ liệu nhất định theo căn chỉnh tối thiểu lớn nhất từng được sử dụng cho bất kỳ kiểu dữ liệu nào. Ví dụ, lệnh này hướng dẫn GCC căn chỉnh parrot_height theo căn chỉnh lớn nhất từng được sử dụng, có thể là căn chỉnh của double:

```
chiều cao con vẹt ngắn _ _thuộc tính_ _ ((căn chỉnh)) = 5;
```

Quyết định này thường liên quan đến sự đánh đổi về không gian/thời gian: các biến được căn chỉnh theo cách này sẽ tiêu tốn nhiều không gian hơn, nhưng việc sao chép vào hoặc ra khỏi chúng (cùng với các thao tác phức tạp khác) có thể nhanh hơn vì trình biên dịch có thể đưa ra các lệnh máy xử lý lượng bộ nhớ lớn nhất.

Nhiều khía cạnh của kiến trúc hoặc chuỗi công cụ của hệ thống có thể áp đặt giới hạn tối đa cho sự căn chỉnh của biến. Ví dụ, trên một số kiến trúc Linux, trình liên kết không thể nhận dạng các sự căn chỉnh vượt quá một giá trị mặc định khá nhỏ. Trong trường hợp đó, một sự căn chỉnh được cung cấp bằng từ khóa này sẽ được làm tròn xuống mức căn chỉnh nhỏ nhất được phép. Ví dụ, nếu bạn yêu cầu căn chỉnh là 32, nhưng trình liên kết của hệ thống không thể căn chỉnh thành hơn 8 byte, biến sẽ được căn chỉnh theo ranh giới 8 byte.

Đặt các biến toàn cục vào một thanh ghi

GCC cho phép lập trình viên đặt các biến toàn cục vào một thanh ghi máy cụ thể, nơi các biến sẽ lưu trú trong suốt thời gian thực thi chương trình.
GCC gọi những biến như vậy là biến đăng ký toàn cục.

Cú pháp yêu cầu lập trình viên chỉ định thanh ghi máy. Ví dụ sau sử dụng ebx:

```
đăng ký int *foo asm ("ebx");
```

Người lập trình phải chọn một biến không bị hàm clobbered: nghĩa là, biến được chọn phải có thể sử dụng được bởi các hàm cục bộ, được lưu và khôi phục khi gọi hàm, và không được chỉ định cho bất kỳ mục đích đặc biệt nào bởi kiến trúc hoặc ABI của hệ điều hành. Trình biên dịch sẽ tạo cảnh báo nếu thanh ghi được chọn không phù hợp. Nếu thanh ghi phù hợp—ebx, được sử dụng trong ví dụ này, phù hợp với kiến trúc x86—trình biên dịch sẽ dừng sử dụng chính thanh ghi đó.

Một tối ưu hóa như vậy có thể cung cấp sự gia tăng hiệu suất lớn nếu biến được sử dụng thường xuyên. Một ví dụ điển hình là với máy ảo. Đặt biến chứa, chẳng hạn, con trỏ khung ngăn xếp ảo trong một thanh ghi có thể dẫn đến những lợi ích đáng kể. Mặt khác, nếu kiến trúc ban đầu thiếu thanh ghi (như kiến trúc x86), thì việc tối ưu hóa này không có nhiều ý nghĩa.

Biến thanh ghi toàn cục không thể được sử dụng trong trình xử lý tín hiệu hoặc bởi nhiều hơn một luồng thực thi. Chúng cũng không thể có giá trị ban đầu vì không có cơ chế nào để các tệp thực thi cung cấp nội dung mặc định cho các thanh ghi. Khai báo biến thanh ghi toàn cục phải đi trước bất kỳ định nghĩa hàm nào.

Chú thích nhánh

GCC cho phép lập trình viên chú thích giá trị mong đợi của một biểu thức—ví dụ, để cho trình biên dịch biết liệu một câu lệnh có điều kiện có khả năng là đúng hay sai. Đổi lại, GCC có thể thực hiện sắp xếp lại khối và các tối ưu hóa khác để cải thiện hiệu suất của các nhánh có điều kiện.

Cú pháp GCC cho ký hiệu nhánh cực kỳ xấu. Để chú thích nhánh dễ nhìn hơn, chúng tôi sử dụng macro tiền xử lý:

```
#định nghĩa có thể(x) _ _builtin_expect (!!(x), 1)
#định nghĩa có thể(x) _ _builtin_expect (!!(x), 0)
```

Các lập trình viên có thể đánh dấu một biểu thức là có khả năng đúng hoặc không có khả năng đúng bằng cách gói nó trong likelihood() hoặc Unlike( ) tương ứng.

Ví dụ sau đây đánh dấu một nhánh là không chắc đúng (tức là có khả năng sai):

```
int ret;

ret = đóng (fd);
```

```
nếu (không chắc chắn
          (ret)) perror ("đóng");
```

Ngược lại, ví dụ sau đây đánh dấu một nhánh có khả năng đúng:

```
const char *home;

home = getenv ("HOME"); if
(có thể là (home))
          printf ("Thư mục home của bạn là %s\n", home);
khác
          fprintf (stderr, "Biến môi trường HOME chưa được thiết lập!\n");
```

Giống như các hàm nội tuyến, các lập trình viên có xu hướng lạm dụng chú thích nhánh. Khi bạn bắt đầu tô điểm cho các biểu cảm, bạn có thể có xu hướng đánh dấu tất cả các biểu cảm. Tuy nhiên, hãy cẩn thận—bạn chỉ nên đánh dấu các nhánh là có khả năng xảy ra hoặc không có khả năng xảy ra nếu bạn biết trước và không nghi ngờ gì rằng các biểu thức sẽ đúng hoặc sai gần như mọi lúc (ví dụ, với độ chắc chắn 99 phần trăm). Các lỗi hiếm khi xảy ra là ứng cử viên tốt cho Unlike( ). Tuy nhiên, hãy nhớ rằng một dự đoán sai còn tệ hơn là không có dự đoán nào cả.

Lấy Kiểu của một Biểu thức

GCC cung cấp từ khóa typeof() để lấy kiểu của một biểu thức nhất định. Về mặt ngữ nghĩa, từ khóa hoạt động giống như sizeof( ). Ví dụ, biểu thức này trả về kiểu của bất kỳ thứ gì x trỏ tới:

```
loại (*x)
```

Chúng ta có thể sử dụng điều này để khai báo một mảng, y, có các kiểu sau:

```
loại (*x) y[42];
```

Một cách sử dụng phổ biến cho typeof( ) là viết các macro “an toàn”, có thể hoạt động trên bất kỳ giá trị số học nào và chỉ đánh giá các tham số của nó một lần:

```
#xác định max(a,b) ({ \ typeof (a)
          _a = (a); \ typeof (b)
          _b = (b); \ _a > _b ?
         _a : _b; \
})
```

Nhận Căn chỉnh của một Kiểu

GCC cung cấp từ khóa _ _alignof_ _ để căn chỉnh một đối tượng nhất định. Giá trị là dành riêng cho kiến trúc và ABI. Nếu kiến trúc hiện tại không có căn chỉnh bắt buộc, từ khóa sẽ trả về căn chỉnh được ABI khuyến nghị. Nếu không, từ khóa sẽ trả về căn chỉnh tối thiểu bắt buộc.

Cú pháp giống hệt như sizeof( ):

```
_ _căn chỉnh_ _(int)
```

Tùy thuộc vào kiến trúc, hàm này có thể trả về kết quả là 4 vì số nguyên 32 bit thư ờng đư ợc căn chỉnh theo ranh giới 4 byte.

Từ khóa cũng hoạt động trên các giá trị lvalue. Trong trư ờng hợp đó, căn chỉnh đư ợc trả về là căn chỉnh tối thiểu của kiểu sao lư u, không phải là căn chỉnh thực tế của giá trị lvalue cụ thể. Nếu sự căn chỉnh tối thiểu đã đư ợc thay đổi thông qua thuộc tính căn chỉnh (đư ợc mô tả trư ớc đó, trong "Tăng cư ờng sự liên kết của một biến"), sự thay đổi đó đư ợc phản ánh bởi _ _alignof_ _.

Ví dụ, hãy xem xét cấu trúc này:

```
cấu trúc tàu {
          int năm xây dựng;
          các quy tắc char;
          int chiều cao cột;
};
```

cùng với đoạn mã này:

```
cấu trúc ship my_ship;

printf ("%d\n", _ _alignof_ _(my_ship.canons));
```

Đoạn mã này sẽ trả về 1, mặc dù việc đệm cấu trúc có thể dẫn đến việc tiêu tốn bốn byte.

Sự bù trừ của một thành viên trong một cấu trúc

GCC cung cấp một từ khóa tích hợp để lấy độ lệch của một thành viên trong cấu trúc trong cấu trúc đó. Macro offsetof() , đư ợc định nghĩa trong <stddef.h>, là một phần của Tiêu chuẩn ISO C. Hầu hết các định nghĩa đều kinh khủng, liên quan đến số học con trỏ tục tĩu và mã không phù hợp với trẻ vị thành niên. Phần mở rộng GCC đơ n giản hơ n và có khả năng nhanh hơ n:

```
#define offsetof(kiểu, thành viên)          _ _builtin_offsetof (kiểu, thành viên)
```

Một cuộc gọi trả về độ lệch của thành viên trong loại—tức là số byte, bắt đầu từ số không, từ đầu cấu trúc đến thành viên đó. Ví dụ, hãy xem xét cấu trúc sau:

```
cấu trúc thuyền chèo {
          char *tên_thuyền;
          số nguyên không dấu nr_oars;
          chiều dài ngắn;
};
```

Các độ lệch thực tế phụ thuộc vào kích thư ớc của các biến và các yêu cầu căn chỉnh của kiến trúc cũng như hành vi đệm, như ng trên máy 32 bit, chúng ta có thể mong đợi gọi offsetof( ) trên struct rowboat và boat_name, nr_oars và length để trả về 0, lần lư ợt là 4 và 8.

Trên hệ thống Linux, macro offsetof() phải đư ợc định nghĩa bằng cách sử dụng từ khóa GCC và không cần phải định nghĩa lại.

Lấy địa chỉ trả về của một hàm

GCC cung cấp một từ khóa để lấy địa chỉ trả về của hàm hiện tại hoặc một trong những hàm gọi hàm hiện tại:

```
trống rỗng * _ _builtin_return_address (cấp độ số nguyên không dấu)
```

Mức tham số chỉ định hàm trong chuỗi lệnh gọi có địa chỉ cần trả về. Giá trị 0 yêu cầu địa chỉ trả về của hàm hiện tại, giá trị 1 yêu cầu địa chỉ trả về của người gọi hàm hiện tại, giá trị 2 yêu cầu địa chỉ trả về của người gọi hàm đó , v.v.

Nếu hàm hiện tại là hàm nội tuyến, địa chỉ trả về là địa chỉ của hàm gọi. Nếu không chấp nhận được, hãy sử dụng từ khóa noinline (đã mô tả trước đó, trong “Suppressing Inlining”) để buộc trình biên dịch không nội tuyến hàm.

Có một số cách sử dụng từ khóa _ _builtin_return_address . Một là cho mục đích gỡ lỗi hoặc thông tin. Một cách khác là để giải phóng chuỗi cuộc gọi, để triển khai nội quan, tiện ích dump sự cố, trình gỡ lỗi, v.v.

Lưu ý rằng một số kiến trúc chỉ có thể trả về địa chỉ của hàm đang gọi.
Trên các kiến trúc như vậy, giá trị tham số khác không có thể dẫn đến giá trị trả về ngẫu nhiên. Do đó, bất kỳ tham số nào khác 0 đều không thể chuyển đổi và chỉ nên sử dụng cho mục đích gỡ lỗi.

Phạm vi trường hợp

GCC cho phép các nhãn câu lệnh case chỉ định một phạm vi giá trị cho một khối duy nhất. Cú pháp chung như sau:

```
trường hợp thấp ... cao:
```

Ví dụ:

```
chuyển đổi (val) {
trường hợp 1 ... 10:
            /* ... */
            phá vỡ;
trường hợp 11 ... 20: /
            * ... */ ngắt;
            mặc
định: /* ...
           */
}
```

Chức năng này cũng khá hữu ích cho các phạm vi chữ hoa và chữ thường của ASCII:

```
trường hợp 'A' ... 'Z':
```

Lưu ý rằng phải có một khoảng trắng trước và sau dấu ba chấm. Nếu không, trình biên dịch có thể bị nhầm lẫn, đặc biệt là với các phạm vi số nguyên. Luôn thực hiện các thao tác sau:

```
trường hợp 4 ... 8:
```

và không bao giờ thế này:

```
trường hợp 4...8:
```

Số học con trỏ hàm và void

Trong GCC, các phép toán cộng và trừ được phép trên các con trỏ kiểu void, và các con trỏ đến các hàm. Thông thường, ISO C không cho phép tính số học trên các con trỏ như vậy vì kích thước của một "khoảng trống" là một khái niệm ngớ ngẩn và phụ thuộc vào những gì con trỏ thực sự đang trỏ đến. Để tạo điều kiện cho số học như vậy, GCC xử lý kích thước của đối tượng tham chiếu là một byte. Do đó, đoạn mã sau đây tiến lên một byte:

```
một++;            /* a là con trỏ void */
```

Tùy chọn -Wpointer-arith khiến GCC tạo cảnh báo khi các tiện ích mở rộng này được sử dụng.

Di động hơn và đẹp hơn trong một cú quét

Hãy đối mặt với nó, cú pháp _ _thuộc tính_ _ không đẹp. Một số phần mở rộng mà chúng tôi đã được xem xét trong chương này về cơ bản yêu cầu các macro tiền xử lý để sử dụng chúng ngon miệng, nhưng tất cả chúng đều có thể được hưởng lợi từ việc cải thiện vẻ ngoài.

Với một chút phép thuật tiền xử lý, điều này không khó. Hơn nữa, trong cùng một hành động, chúng ta có thể làm cho các phần mở rộng GCC có thể di động, bằng cách định nghĩa chúng trong trường hợp không phải GCC trình biên dịch (bất kể đó là gì).

Để thực hiện như vậy, hãy dán đoạn mã sau vào tiêu đề và đưa tiêu đề đó vào các tập tin nguồn của bạn:

```
#nếu __GNUC_ _ >= 3
# không xác định nội tuyến
# định nghĩa inline inline __attribute_ _ ((always_inline))
# định nghĩa __noinline __attribute_ _ ((noinline))
# định nghĩa __thuộc tính __thuần túy_ _ ((thuần túy))
# định nghĩa __const __attribute_ _ ((const))
# định nghĩa __noreturn __attribute_ _ ((noreturn))
# định nghĩa __malloc __attribute_ _ ((malloc))
# định nghĩa __phải_kiểm_tra __thuộc_tính_ _ ((cảnh_cảnh_không_sử_dụng_kết_quả))
# định nghĩa __thuộc tính __đã lỗi thời_ _ ((đã lỗi thời))
# định nghĩa __used __attribute_ _ ((đã sử dụng))
# định nghĩa __thuộc tính __chưa sử dụng_ _ ((chưa sử dụng))
```

```
# định nghĩa __packed #             __thuộc tính_ _ ((đóng gói))
định nghĩa __align(x) #             __thuộc tính_ _ ((căn chỉnh (x)))
định nghĩa __align_max # định       __thuộc tính_ _ ((căn chỉnh))
nghĩa likelihood(x) # định          _ _builtin_expect (!!(x), 1)
nghĩa Unlike(x) #else               _ _builtin_expect (!!(x), 0)

# định nghĩa _ _noinline #          /* không có noinline */
định nghĩa _ _pure #                /* không thuần túy */
định nghĩa _ _const #               /* không có const */
định nghĩa _ _noreturn #            /* không trả về */
định nghĩa _ _malloc #              /* không có malloc */
định nghĩa _ _must_check /* không có cảnh báo_kết quả_chư a_sử_dụng */
# định nghĩa _ _deprecated /* không bị loại bỏ */
# định nghĩa _ _used /* không sử dụng */
# định nghĩa _ _unused /* không có chư a sử dụng */
# định nghĩa _ _đóng gói /* không đóng gói */
# định nghĩa _ _align(x) /* không căn chỉnh */
# định nghĩa _ _align_max /* không có align_max */
# định nghĩa có thể(x) (x)
# định nghĩa Unlike(x) (x)
#kết thúc nếu
```

Ví dụ, lệnh sau đây đánh dấu một hàm là thuần túy bằng cách sử dụng phím tắt của chúng tôi:

```
_ _pure int foo (void) { /* ... */
```

Nếu GCC đang đư ợc sử dụng, hàm đư ợc đánh dấu bằng thuộc tính pure . Nếu GCC không phải là trình biên dịch, bộ xử lý trư ớc thay thế mã thông báo _ _pure bằng một lệnh không hoạt động. Lư u ý rằng bạn có thể đặt nhiều thuộc tính vào một định nghĩa nhất định và do đó bạn có thể sử dụng nhiều hơ n một trong những định nghĩa này có thể đư ợc định nghĩa theo một định nghĩa duy nhất mà không có vấn đề gì.

Dễ dàng hơ n, đẹp hơ n và di động hơ n!

Tài liệu tham khảo này trình bày các bài đọc đư ợc khuyến nghị liên quan đến lập trình hệ thống, đư ợc chia thành bốn tiểu thể loại. Không có tác phẩm nào trong số này là bài đọc bắt buộc.
Thay vào đó, chúng đại diện cho quan điểm của tôi về những cuốn sách hàng đầu về chủ đề đã cho. Nếu bạn thấy mình đang khao khát tìm hiểu thêm thông tin về các chủ đề đư ợc thảo luận ở đây, đây là những cuốn sách yêu thích của tôi.

Một số sách này đề cập đến tài liệu mà cuốn sách này cho rằng ngư ời đọc đã thông thạo, chẳng hạn như ngôn ngữ lập trình C. Các văn bản khác đư ợc đư a vào là những phần bổ sung tuyệt vời cho cuốn sách này, chẳng hạn như các tác phẩm về gdb, Subversion (svn) hoặc thiết kế hệ điều hành. Một số khác xử lý các chủ đề nằm ngoài phạm vi của cuốn sách này, chẳng hạn như đa luồng của socket. Dù trư ờng hợp nào đi nữa, tôi cũng đề xuất tất cả chúng.

Tất nhiên, những danh sách này chắc chắn không đầy đủ—hãy thoải mái khám phá những danh sách khác tài nguyên.

Sách về Ngôn ngữ lập trình C

Những cuốn sách này ghi lại ngôn ngữ lập trình C, ngôn ngữ chung của lập trình hệ thống. Nếu bạn không viết mã C tốt như bạn nói tiếng mẹ đẻ của mình, một hoặc nhiều tác phẩm sau đây (cùng với nhiều bài thực hành!) sẽ giúp bạn theo hư ớng đó. Nếu không có gì khác, thì tựa sách đầu tiên—đư ợc biết đến rộng rãi là K&R—là một tác phẩm đáng đọc. Sự ngắn gọn của nó cho thấy sự đơ n giản của C.

Ngôn ngữ lập trình C, ấn bản lần 2. Brian W. Kernighan và Dennis M. Ritchie.
Hội trư ờng Prentice, 1988.

Cuốn sách này, đư ợc viết bởi tác giả của ngôn ngữ lập trình C và đồng nghiệp của ông khi đó, chính là kinh thánh của lập trình C.

C trong một vỏ hạt. Peter Prinz và Tony Crawford. O'Reilly Media, 2005.

Một cuốn sách tuyệt vời bao gồm cả ngôn ngữ C và thư viện C chuẩn.

C Tài liệu tham khảo bỏ túi. Peter Prinz và Ulla Kirch-Prinz. Tony Crawford dịch.

Phương tiện truyền thông O'Reilly, 2002.

Tài liệu tham khảo ngắn gọn về ngôn ngữ C, được cập nhật tiện lợi cho ANSI C99.

Lập trình C chuyên nghiệp. Peter van der Linden. Hội trường Prentice, 1994.

Một cuộc thảo luận tuyệt vời về các khía cạnh ít được biết đến của ngôn ngữ lập trình C, được giải thích bằng sự dí dỏm và khiếu hài hước tuyệt vời. Cuốn sách này đầy rẫy những câu chuyện cười vô nghĩa, và tôi thích điều đó.

Câu hỏi thường gặp về lập trình C: Các câu hỏi thường gặp, ấn bản lần 2. Steve Summit. Addison-Wesley, 1995.

Cuốn sách đồ sộ này chứa hơn 400 câu hỏi thường gặp (có câu trả lời) về ngôn ngữ lập trình C. Nhiều câu hỏi thường gặp đòi hỏi những câu trả lời hiển nhiên trong mắt các bậc thầy C, nhưng một số câu hỏi và câu trả lời quan trọng hơn có thể gây ấn tượng ngay cả với những lập trình viên C uyên bác nhất. Bạn là một ninja C thực thụ nếu bạn có thể trả lời tất cả những câu hỏi khó nhằn này! Điểm trừ duy nhất là cuốn sách chưa được cập nhật cho ANSI C99 và chắc chắn đã có một số thay đổi (tôi đã tự tay sửa lỗi trong bản sao của mình). Lưu ý rằng có một phiên bản trực tuyến có thể đã được cập nhật gần đây hơn.

Sách về lập trình Linux

Các văn bản sau đây đề cập đến lập trình Linux, bao gồm các cuộc thảo luận về các chủ đề không được đề cập trong cuốn sách này (socket, IPC và pthread) và các công cụ lập trình Linux (CVS, GNU Make và Subversion).

Lập trình mạng Unix, Tập 1: API mạng Sockets, ấn bản lần thứ 3. W. Richard Stevens và cộng sự. Addison-Wesley, 2003.

Cuốn sách chính thức về API socket; thật không may là không dành riêng cho Linux, nhưng may mắn là gần đây đã được cập nhật cho IPv6.

Lập trình mạng UNIX, Tập 2: Truyền thông liên tiến trình, ấn bản lần 2. W. Richard Stevens. Prentice Hall, 1998.

Một cuộc thảo luận tuyệt vời về giao tiếp giữa các tiến trình (IPC).

Lập trình PThreads: Tiêu chuẩn POSIX cho đa xử lý tốt hơn. Bradford Nichols và cộng sự. O'Reilly Media, 1996.

Đánh giá về API luồng POSIX, pthreads.

Quản lý dự án với GNU Make, ấn bản lần thứ 3. Robert Mecklenburg. O'Reilly Media, 2004.

Một bài viết tuyệt vời về GNUMake, công cụ cổ điển để xây dựng các dự án phần mềm trên Linux.

Essential CVS, ấn bản lần 2. Jennifer Versperman. O'Reilly Media, 2006.

Một giải pháp tuyệt vời cho CVS, công cụ cổ điển để kiểm soát bản sửa đổi và quản lý mã nguồn trên hệ thống Unix.

Kiểm soát phiên bản với Subversion. Ben Collins-Sussman và cộng sự. O'Reilly Media, 2004.

Một phiên bản Subversion phi thư ờng, công cụ thích hợp để kiểm soát bản sửa đổi và quản lý mã nguồn trên hệ thống Unix, đư ợc thực hiện bởi ba tác giả của Subversion.

Tài liệu tham khảo bỏ túi GDB. Arnold Robbins. O'Reilly Media, 2005.

Một hư ớng dẫn bỏ túi hữu ích về gdb, trình gỡ lỗi của Linux.

Linux in a Nutshell, ấn bản lần thứ 5. Ellen Siever và cộng sự. O'Reilly Media, 2005.

Một tài liệu tham khảo tổng hợp về mọi thứ liên quan đến Linux, bao gồm nhiều công cụ tạo nên môi trư ờng phát triển của Linux.

Sách về Linux Kernel

Hai tiêu đề đư ợc liệt kê ở đây đề cập đến hạt nhân Linux. Có ba lý do để tìm hiểu chủ đề này. Đầu tiên, hạt nhân cung cấp giao diện lệnh gọi hệ thống cho không gian ngư ời dùng và do đó là cốt lõi của lập trình hệ thống. Thứ hai, các hành vi và đặc điểm riêng của hạt nhân làm sáng tỏ các tư ơ ng tác của nó với các ứng dụng mà nó chạy. Cuối cùng, hạt nhân Linux là một đoạn mã tuyệt vời và những cuốn sách này rất thú vị.

Phát triển hạt nhân Linux, ấn bản lần 2. Robert Love. Novell Press, 2005.

Công trình này lý tư ởng cho các lập trình viên hệ thống muốn tìm hiểu về thiết kế và triển khai hạt nhân Linux (và tất nhiên, tôi sẽ là ngư ời thiếu sót nếu không đề cập đến chuyên luận của riêng tôi về chủ đề này!). Không phải là tài liệu tham khảo API, cuốn sách này cung cấp một cuộc thảo luận tuyệt vời về các thuật toán đư ợc sử dụng và các quyết định đư ợc đư a ra bởi hạt nhân Linux.

Trình điều khiển thiết bị Linux, ấn bản lần thứ 3. Jonathan Corbet và cộng sự. O'Reilly Media, 2005.

Đây là một hư ớng dẫn tuyệt vời để viết trình điều khiển thiết bị cho hạt nhân Linux, với các tham chiếu API tuyệt vời. Mặc dù hư ớng đến trình điều khiển thiết bị, các cuộc thảo luận sẽ có lợi cho các lập trình viên ở bất kỳ trình độ nào, bao gồm cả các lập trình viên hệ thống chỉ muốn tìm hiểu sâu hơ n về các hoạt động của hạt nhân Linux. Một sự bổ sung tuyệt vời cho cuốn sách hạt nhân Linux của riêng tôi.

Sách về Thiết kế Hệ điều hành

Hai tác phẩm này, không dành riêng cho Linux, đề cập đến thiết kế hệ điều hành một cách trừu tượng. Như tôi đã nhấn mạnh trong cuốn sách này, hiểu biết sâu sắc về hệ thống mà bạn viết mã chỉ cải thiện kết quả đầu ra của bạn.

Hệ điều hành, ấn bản lần thứ 3. Harvey Deitel và cộng sự. Prentice Hall, 2003.

> Một chuyến du ngoạn về lý thuyết thiết kế hệ điều hành kết hợp với các nghiên cứu tình huống hàng đầu đưa lý thuyết đó vào thực tế. Trong tất cả các sách giáo khoa về thiết kế hệ điều hành, đây là cuốn tôi thích nhất: hiện đại, dễ đọc và đầy đủ.

Hệ thống UNIX cho kiến trúc hiện đại: Xử lý đa đối xứng và lưu trữ đệm cho lập trình hạt nhân. Curt Schimmel. Addison-Wesley, 1994.

> Mặc dù chỉ liên quan khiêm tốn đến lập trình hệ thống, cuốn sách này cung cấp một cách tiếp cận tuyệt vời đối với những nguy cơ của đồng thời và bộ nhớ đệm hiện đại đến mức tôi thậm chí còn giới thiệu nó cho cả các nha sĩ.

Mục lục

Biểu

tư ợng / (dấu gạch chéo), 11

MỘT

ABI (giao diện nhị phân ứng dụng), 4, 5 hàm
abort(), 282 đư ờng dẫn
tuyệt đối, 11, 212 thời gian
tuyệt đối, 309 danh
sách kiểm soát truy cập (ACL), 18
ACL (danh sách kiểm soát truy
cập), 18 hàm adjtime(),
322 hàm adjtimex(), 322–324 giao
diện aio, 112 hàm
alarm(), 282, 330 hàm
alarm_handler(), 331 căn chỉnh dữ
liệu, 252 từ khóa _
alignof _, 346 hàm alloca(),
264 sao chép chuỗi trên
ngăn xếp, 266 phân bổ, 243 ánh xạ bộ nhớ
ẩn danh, 256–260
tạo, 257 ánh xạ /dev/zero, 259

Tiêu chuẩn ANSI C, 7

Bộ lập lịch I/O dự đoán, 117 hệ
thống chống bó cứng phanh (ABS) như hệ thống
thời gian thực, 176
API (giao diện lập trình ứng dụng), 4 chế độ thêm
vào, write(), 34 ứng dụng phụ
thuộc thời
gian và đồng hồ hệ thống, 321 mảng,
phân bổ, 247
hàm asctime() và asctime_r(), 320 sự kiện
không đồng bộ, tín hiệu cho (xem tín hiệu) I/O
không đồng bộ, 112 hoạt
động ghi không đồng bộ, 111 hàm
atexit(), 137 cú pháp _
thuộc tính _, làm đẹp bằng macro tiền xử
lý, 349
biến tự động, 264

B

nhóm quy trình nền, 154 chính sách
lập lịch hàng loạt, 180 hàm
bcmp(), 270 luồng
bdflush, 61 ký hiệu
big-oh, 163 khả năng
tư ơ ng thích nhị phân, 5
tệp nhị phân và tệp văn bản,
66 thiết bị khối,
13 sector,
14 đọc chặn, 30
khối, 15, 62
kích thư ớc
khối, 63 ảnh hư ởng đến hiệu
suất, 63 điểm
dừng, 255 hàm brk(),
256 liên kết bị
hỏng, 12 phân đoạn
bss, 245 lư ợc đồ phân bổ bộ nhớ buddy, 256
kích thư ớc bộ
đệm, 64 I/O đệm (đầu vào/đầu ra),
62 mô tả tệp liên quan, thu thập, 77 kích
thư ớc khối, 63
ảnh hư ởng đến hiệu suất, 63

I/O đệm (tiếp theo) kiểm
soát bộ đệm, 77 lỗi và EOF, 76
mở tệp, 65 chế độ,
65 chư ơ ng trình
mẫu, 72–74
I/O chuẩn, 64 con trỏ tệp,
65 hạn chế, 81

luồng
đóng, 67
đóng tất cả, 67
xả, 75 mở
thông qua mô tả tệp, 66 tìm kiếm
luồng, 74 luồng, đọc
từ, 67–70
đặt lại ký tự, 68 đọc toàn bộ
một dòng, 68 đọc các chuỗi
tùy ý, 69 đọc dữ liệu nhị
phân, 70 đọc một ký tự
tại một thời điểm, 67 luồng, ghi vào,
70–72 căn chỉnh dữ liệu, 71
ghi một ký tự đơ n,
71 ghi một chuỗi, 72 ghi dữ
liệu nhị phân, 72 an
toàn luồng, 79–81 khóa
tệp thủ công, 80 hoạt
động luồng không khóa,
81 I/O đư ợc đệm bởi ngư ời dùng,
62–64 cấu trúc dữ liệu
buffer_head, 61 bộ đệm, 37, 61
bộ đệm bẩn, 60
từ khóa _
_builtin_return_address, 348

C

Trình biên dịch C (xem gcc)
Ngôn ngữ C, 7, 64
GNUC, 339
Thư viện C (libc),
4 hiệu ứng bộ nhớ đệm của quá trình di
chuyển, 173 hàm
calloc( ), 247 khả năng hệ thống, 18
KHÓA CAP_IPC, 276
Khả năng CAP_KILL, 292
CAP_SYS_TIME, 318 câu
lệnh case, 348 lệnh
cd, 216 thiết bị ký
tự và tệp thiết bị ký tự, 13
hàm chdir(), 215 tiến trình
con, 17, 127 kế thừa bộ nhớ và
fork(), 274 (xem thêm tiến trình) hàm chmod(),
200 hàm chown(), 201 hàm
Clearlyr(), 76 hàm
Clock_getres(), 314 hàm
Clock_gettime(), 316 kiểu
Clockid_t(), 313 hàm Clock_nanosleep(),
327 hàm Clock_settime(), 319 hàm
Close(), 41 ví dụ mã,
quyền, mô hình tệp chung xv, 58 Bộ lập lịch
I/O hàng đợi công bằng hoàn chỉnh (CFQ),
118 tránh tắc nghẽn, 61
từ khóa const, 341 kiểm soát thiết bị đầu
cuối, 154 đa nhiệm hợp tác,
163 như ờng nhịn, 163, 166–169 sử dụng hợp
pháp, 167 Linux
phiên bản 2.6, những thay đổi
trong, 168 copy-on-write
(COW), 134, 244 CPU_SETSIZE, 174
hàm creat(), 28 vùng quan trọng, 79,
297 hàm ctime() và hàm
ctime_r(), 320 thời gian
hiện tại, lấy, 315–318 độ phân giải microgiây,
316 độ phân giải nanogiây, 316 thời gian
hiện tại, thiết lập, 318,
319 thư mục làm việc hiện
tại (cwd), 11, 213–217 thay
đổi, 215–217 thu đư ợc, 213–215

D

daemon( ) hàm, 160 daemon,
159–161 liên kết
tư ợng trư ng treo, 225
căn chỉnh dữ liệu,
71 phân đoạn dữ
liệu, 245 quản lý phân đoạn dữ liệu, 255
Bộ lập lịch I/O hạn chót, 116 khóa
đư ợc xác định, 204
phân trang theo yêu cầu,
273 tính quyết định, 187–189

/dev/zero, 259
nút thiết bị, 231–232 số
chính và số phụ, 231 trình tạo số
ngẫu nhiên, 232 nút đặc biệt, 231
thiết bị, 13 /dev/
zero, 259
giao tiếp ngoài
băng tần, 233 hàm difftime( ), 321 I/
O trực tiếp, 40 thư mục,
11, 212–223
tạo, 218 thư mục làm việc
hiện tại (cwd),
213–217 thay đổi, 215–217
lấy, 213–215 mục
nhập thư mục, 212
mục nhập thư mục
(dentry), 11 luồng thư
mục, 220 đóng, 221 đọc từ,
221 liên kết, 212, 223–
228 tên, ký
tự hợp lệ cho, 212
tên, độ dài của, 212
đọc nội dung của, 220–223 lệnh gọi
hệ thống, 222 xóa, 219
thư mục con, 212 hàm dirfd( ),
220 bộ đệm bẩn, 60

dirty_expire_centiseconds,
37 địa chỉ đĩa,
114 hàm dnotify, 235 phân bổ bộ
nhớ động, 245–255 căn
chỉnh, 252–255 căn
chỉnh các kiểu không chuẩn và phức tạp,
254 phân bổ bộ nhớ
căn chỉnh, 252 con trỏ, 255 phân
bổ mảng, 247 giải
phóng bộ nhớ động, 250–252 phân bổ
lại kích
thư ớc, 249

E

sự kiện kích hoạt cạnh, 94
gid hiệu quả, 17
ID ngư ời dùng hiệu quả (uid),
17, 150 chư ơ ng trình lệnh đẩy
ra, 233 thuật toán thang
máy, 116 nhóm entropy
cuối tệp (xem EOF), 232
EOF (kết thúc tệp), 30
lỗi và, 76 kỷ
nguyên, 309
tiện ích epoll,
89 giao diện epoll,
57 hàm epoll_create(), 89 hàm
epoll_ctl(), 90–92 hàm
epoll_wait(), 93 errno, 19–
22 mô tả lỗi,
20 xử lý lỗi, 19–22
lỗi và EOF, 76 giao diện
thăm dò sự kiện, 89–
94 kiểm soát epoll, 90–92 tạo
một thể hiện epoll, 89 sự
kiện kích hoạt cạnh so với kích
hoạt cấp độ, 94 chờ sự kiện, 93 họ hàm
exec, 129–
132 giá trị lỗi, 131
hàm execl(), 129–130 quyền thực thi,
18 hàm exit() và
_exit(), 136 thuộc tính mở
rộng, 203 khóa và giá trị,
204 danh sách, 209 không gian tên,
205 hoạt động, 206 xóa,
210 truy xuất, 206
không gian
tên bảo mật, 206
cài đặt, 208
không gian tên
hệ thống, 205
không gian tên đáng tin
cậy, 206
không gian tên ngư ời
dùng, 206 phân mảnh bên
ngoài, 256

F

Hàm fchdir( ), 215 Hàm
fchmod( ), 200 Hàm
fchown( ), 202 Hàm
fclose( ), 67 Hàm
fcloseall( ), 67 Hàm fd
(xem mô tả tệp) Hàm
fdatasync( ), 37 Giá trị
trả về và mã lỗi, 38 Hàm feof( ),
76 Hàm ferror( ), 76
Hàm fflush( ), 75 Hàm
fgetc( ), 67, 69 Hàm
fgetpos( ), 75

hàm fgets(), 68 hàm
fgetxattr(), 207
Lớp FIFO (vào trước ra trước), 179
FIFO, 13
mô tả tệp, 9, 23 luồng,
mở qua, 66 tệp I/O (đầu
vào/đầu ra), 23–61 lời khuyên,
108–111 lợi thế,
110 hàm
posix_fadvise(), 108 tệp đóng,
41 I/O trực tiếp,
40 giao diện
thăm dò sự kiện (xem giao diện thăm
dò sự kiện)
Bộ lập lịch I/O (xem bộ lập lịch I/O)
bên trong hạt nhân, 57–
61 bộ đệm trang,
59 ghi lại trang, 60
hệ thống tệp ảo, 58
đầu ra tuyến tính,
84 lseek( ), 42–
44 giá trị lỗi,
44 giới hạn, 44
tìm kiếm sau phần cuối của tệp, 43
I/O được ánh xạ bộ nhớ (xem I/O được
ánh xạ bộ
nhớ) I/O được ghép kênh, 47–57
hàm poll(), 53–56 poll()
so với select(), 57 hàm
ppoll(), 56 hàm
pselect(), 52 hàm
select(), 48–53 mở tệp, 24–
29
hàm creat(), 28 hàm
open(), 24–26 chủ sở hữu
các tệp mới, 26 quyền
của các tệp mới, 27 giá trị
trả về và mã lỗi, 29
đọc và ghi theo vị trí, 44 giá
trị lỗi, 45 hàm
readahead(), 110 đọc tệp, 29–
33 giá trị lỗi bổ
sung, 32 đọc không chặn, 32
đọc tất cả các byte, 31
giá trị trả về, 30 giới
hạn kích thước
trên read(), 33 I/O phân
tán/thu thập (xem I/O phân tán/thu thập) I/O
đồng bộ, 37–40 fsync() và
fdatasync(), 37
Cờ O_DSYNC và O_RSYNC, 40
Cờ O_SYNC, 39 hàm
sync(), 39
cắt bớt các tệp, 45
write( ), 33–37
mã lỗi bổ sung, 35 chế độ
thêm vào, 34 hành
vi của write( ), 36 ghi
không chặn, 35 ghi một
phần, 34 giới hạn
kích thước trên,
36 con trỏ tệp,
65 bảng tệp, 23
FILE typedef, 65
hàm fileno( ), 77 tệp,
9–15 thời
gian truy cập, sửa đổi và thay đổi,
198 đóng
tệp, 41 sao chép,
228 xóa, 12
nút thiết
bị, 231–232 số chính và
số phụ, 231 trình tạo số ngẫu
nhiên, 232 nút đặc biệt, 231 thư
mục (xem thư mục)
thuộc tính mở rộng (xem
thuộc tính mở rộng) sự kiện tệp,
giám sát,
234–242 giao diện inotify, 234 theo
dõi, 236–242 tệp I/O
(xem tệp I/O) quyền
sở hữu tệp, 26 quyền
tệp, 27 vị trí tệp
hoặc độ lệch tệp, 9
cắt bớt tệp, 45 tên tệp và
inode, 10 inode,
196 độ dài, 10 liên kết,
11, 223–228
siêu dữ
liệu, 196 hàm để
lấy, 196

Các loại MIME, lưu trữ, chế
độ 205, giá trị pháp lý cho,
200 di chuyển,
229 tác động của việc di chuyển đến và đi từ các
loại tệp, 230 tên,
ký tự hợp lệ cho, 212 tên, độ dài
của, 212 quyền sở hữu,
201 quyền, 199
đối số chế độ, 27
tệp thông thường, 9
tệp đặc biệt,
13 số lần sử
dụng, 223

filesystem gid, 17
filesystem uid, 17
filesystems, 14
    blocks, 15
    filesystem-agnosticism, 204
    links and, 223
    mount and unmounting, 14 supported
    in Linux, 14 hàm
flistxattr(), 210 hàm
flockfile(), 80 hàm
fopen(), 65 nhóm tiến
trình nền trư ớc, 154 hàm fork(),
17, 132–136, 290 dấu gạch chéo xuôi (/),
11 hàm fputc(), 71 hàm
fputs(), 72 hàm
fremovexattr(), 211 hàm
fseek(), 74 hàm fsetpos(), 74 hàm
fsetxattr(), 208 hàm
fstat(), 197 hàm fsync(),
37, 76 giá trị trả về và lỗi
mã, 38 hàm ftell(), 75
hàm ftruncate(), 46 hàm
    ftrylockfile(), 80 thiết bị đầy đủ,
232 tên đư ờng dẫn đủ
điều kiện, 11 hàm hàm hằng
số, 341 hàm nội tuyến, 340
ngăn chặn nội
tuyến, 340 đánh dấu là lỗi thời,
343 đánh
    dấu là không sử dụng, 343
    đánh dấu là đã sử dụng,
        343 hàm cấp phát bộ nhớ, 342
    hàm không trả về, 342 hàm thuần
    túy, 341 hàm funlockfile(),
    80 hàm fwrite(), 72

buộc các hàm kiểm tra giá trị trả về,
    342 hàm hoặc
tham số, đánh dấu là không sử dụng, 343
    địa chỉ trả về
của hàm, thu thập, 348 biến
    toàn cục, đặt
vào sổ đăng ký, 345

GNU C, 339
hàm nội tuyến, 340 hàm
nội tuyến, ngăn chặn, 340 bù trừ
thành viên trong một cấu trúc, 347 hàm
cấp phát bộ nhớ, 342 hàm không trả
về, 342 cấu trúc đóng gói, 343
khả năng di động, cải
thiện, 349 hàm thuần túy, 341
căn chỉnh kiểu, nhận,
346 hàm đã sử dụng, đánh dấu,
343 căn chỉnh biến, tăng, 344
số học void và con trỏ, 349 hàm
get_current_dir_name( ), 214 hàm
getcwd( ), 213, 216 hàm getdents( ), 222
hàm getitimer( ), 332 hàm
getpagesize( ), 98 hàm
getpgid( ), 158 hàm
getpgrp( ), 158 hàm
getpid( ), 128 hàm
getpriority( ), 171 hàm
getrlimit( ), 190 hàm
gets(), 81 hàm getsid(), 156
hàm gettimeofday(), 316 hàm
getwd(), 214 hàm
getxattr(), 207 ghosts,
149 gid (ID nhóm), 17 glibc (GNU
libc), 4 cấp phát bộ
nhớ, 256 biến thanh ghi
toàn cục,
345 hàm gmtime() và
gmtime_r(), 320

G

gcc (nhị phân),
    4 tiêu chuẩn đư ợc hỗ trợ, 8
GCC (Bộ sư u tập trình biên dịch GNU), 4
    Mở rộng ngôn ngữ C, 339–350 chú thích
        nhánh, 345 phạm vi trư ờng
        hợp, 348 hàm hằng
        số, 341 hàm không dùng nữa,
        đánh dấu, 343 kiểu biểu thức, nhận, 346
GNUC, 8, 339
Bộ sư u tập trình biên dịch GNU (xem gcc)
GNU libc (glibc), 4
nhóm ID (gid), 17
nhóm, 17
    quyền sở hữu các quy trình, 127
    nhóm chính hoặc nhóm đăng nhập, 17
Trình quản lý tệp GUI, hành vi đánh hơ i
        loại MIME, 205

H

mối quan hệ cứng,
173 liên kết cứng, 12, 223,
224 hệ thống thời gian thực
cứng, 176 giới hạn tài
nguyên cứng, 190 đồng
hồ phần cứng,
310 tiêu
đề, 19 đống, 245 lệnh hwclock, 310

-

quy trình nhàn rỗi, 126
chính sách lập lịch nhàn rỗi, 180
IEEE (Viện Điện và Điện tử
Kỹ sư), 6 quy
trình init, 17, 126
hàm nội tuyến, 340 ngăn
chặn nội tuyến, 340 từ khóa
nội tuyến, 340 số
inode, thu thập, 196 inode, 10,
196 số lư ợng
liên kết, 12 sự
kiện inotify, 238–240 sự
kiện nâng cao, 239 liên
kết các sự kiện di chuyển, 240 đọc,
238 giao diện
inotify, 234 khởi tạo,
235 hàm
inotify_add_watch(), 236, 238 cấu trúc
inotify_event, 238 hàm
inotify_init(), 235
Viện Điện và Điện tử
Kỹ sư (IEEE), 6 phân
mảnh nội bộ, 256
Tổ chức quốc tế cho
Chuẩn hóa (ISO), 7 giao
tiếp giữa các tiến trình (IPC), 13, 19 bộ đếm
thời gian, 331 trang
không hợp lệ, 244
I/O (đầu vào/đầu ra)
I/O không đồng bộ, 112 I/O
đệm (xem I/O đệm) tệp I/O (xem tệp
I/O)
Ư u tiên I/O, 172
Bộ lập lịch I/O (xem bộ lập lịch I/O)
Thời gian chờ I/O, 40
Các tiến trình liên kết I/O,
164 lệnh gọi hệ thống và 77
Bộ lập lịch I/O, 114–125 địa
chỉ đĩa, 114 vòng đời,
115
hợp nhất và sắp xếp, 115 tối
ư u hóa hiệu suất, 119–125 đọc, 116–119
Bộ lập lịch I/O dự đoán, 117
Hàng đợi công bằng hoàn chỉnh (CFQ) I/O
Lập lịch, 118
Bộ lập lịch I/O hạn chót, 116
Trình lập lịch I/O Noop, 119
lập lịch trong không gian ngư ời
dùng, 120 sắp xếp theo
inode, 121 sắp xếp theo
đư ờng dẫn, 120 sắp xếp theo khối
vật lý, 122 lựa chọn và cấu hình,
119 hàm ioctl( ), 233
IOV_TỐI ĐA, 85
IPC (giao tiếp giữa các tiến trình), 13
ISO (Tổ chức quốc tế về
Chuẩn hóa), 7 cấu trúc
itimerval, 332

J

jiffies counter, 309
jitter, 177
kiểm soát công việc, 154

K

giao
diện ánh xạ tệp hạt nhân (xem I/O đư ợc
ánh xạ bộ nhớ)
I/O (đầu vào/đầu ra), triển khai, 57–61
bộ nhớ đệm
trang, 59 ghi
lại trang, 60 hệ
thống tệp ảo, 58
Bộ lập lịch I/O (xem bộ lập lịch I/O), 114
đệm hạt nhân trái ngư ợc với I/O đệm
ngư ời dùng, 62 lời
khuyên ánh xạ và, 106 đọc
trư ớc, 107 bộ
đếm thời gian hệ
thống, 309 phép đo thời
gian, 308 sử dụng các mô tả
tệp, 23 ứng dụng không gian ngư ời dùng, giao
tiếp
với, 3 thời gian
hạt nhân,
40 phím, 204 hàm kill(), 284,
291, 307 tín
hiệu cho, 303 lệnh kill, 281

L

độ trễ, 177
hàm lchown(), 201 sự kiện
kích hoạt theo cấp độ, 94 hàm
lgetxattr(), 207 libc (thư
viện C), 4 trình bao
bọc likelihood(), 345 I/
O tuyến tính,
84 hàm link(), 224 liên
kết, 11, 212, 223–228 liên
    kết bị hỏng, 12
    liên kết cứng, 12, 224
    số lượng liên
    kết, 12 liên kết tượng
    trưng, 12, 225 hủy liên kết, 227
Thang máy Linus, 116
Linux, 1
    Tiêu chuẩn C và, 7 khả
    năng tương thích về phía trước, 8
    Cơ sở chuẩn Linux (LSB), 8
    Unix so với, 1
Quỹ Linux, 8
Giao diện hệ thống Linux, hàm
xiii listxattr(), 209 hàm
llistxattr(), 210 cân bằng
tải, 173 địa phương
tham chiếu, 59 hàm
localtime() và localtime_r(), 321 đăng nhập, 17
shell đăng
nhập, 17, 154 nhóm
đăng nhập, 17 hàm
lremovexattr(), 211 lệnh ls, 196

LSB (Linux Standard Base), 8
hàm lseek(), 42–44 giá
    trị lỗi, 44 hạn
    chế, 44 tìm kiếm
    sau phần cuối của tệp, 43 hàm
lsetxattr(), 208 hàm
lstat(), 197

Tôi

máy đăng ký, 3 hàm
madvise(), 106–108 giá trị trả
    về và mã lỗi, 108
thực hiện, phụ thuộc thời gian
của, 321 hàm mallinfo(),
263 hàm malloc(), 246
    trình bao bọc xmalloc() cho,
247 hàm malloc0(), 248
Biến môi trường MALLOC_CHECK_, 263 hàm

malloc_usable_size( ) và malloc_trim( ), 262
            hàm mallopt( ),
260 tham số, 261 tệp được
    ánh xạ, 245 ánh
xạ, 245 tuổi bộ đệm
tối đa, 37 hàm
memchr( ), 272 hàm memcmp( ),
270 hàm memfrob( ), 272 hàm
memmem( ), 272 hàm memmove( ),
271 định địa chỉ bộ nhớ và căn
chỉnh dữ liệu, 71 cấp phát bộ
nhớ, 243 cấp phát bộ nhớ nâng
cao, 260–263 hàm malloc_usable_size( ) và malloc_
trim( ), 262 hàm mallopt( ),
    260 chọn cơ chế, 268 gỡ lỗi, 263
        MALLOC_CHECK_, 263 lấy số
        liệu thống kê, 263 bộ nhớ
    động, phân bổ, 245–255 căn chỉnh, 252–255
        phân bổ mảng, 247
        giải phóng bộ nhớ động,
        250–252 thay đổi kích thước phân bổ,
        249 phân bổ cơ hội, 277
    cam kết quá mức và OOM, 277

    phân bổ dựa trên ngăn xếp, 264–268
        sao chép chuỗi trên ngăn xếp, 266 mảng
        có độ dài thay đổi, 267 mở
    khóa bộ nhớ, 275 quản lý
bộ nhớ, 243 ánh xạ bộ nhớ ẩn
    danh, 256–260 tạo, 257 ánh xạ /dev/zero, 259
        phân đoạn dữ
        liệu, quản lý, 255 khóa
    bộ nhớ, 273–277 phân trang theo
    yêu cầu, 273 khóa toàn bộ
        không gian địa chỉ,
        275 khóa giới hạn, 276 khóa một phần
        không gian địa chỉ,
        274 thao tác bộ nhớ, 269–273 so sánh
    byte, 270 xáo trộn byte, 272 di
        chuyển byte, 271 tìm
        kiếm byte, 272 thiết lập
        byte, 269

quản lý bộ nhớ (tiếp theo) đơn vị
quản lý bộ nhớ, 15 không gian
địa chỉ quy trình, 243–245 vùng
bộ nhớ, 245 trang và
phân trang, 243 chia sẻ
và sao chép khi ghi, 244 I/O được
ánh xạ bộ nhớ, 95–108 thay đổi
chế độ bảo vệ của ánh xạ, 104
đưa ra lời
khuyên, 106–108 hàm
mmap( ), 95–99, 100 ưu điểm,
101 nhược điểm,
102 kích thước
trang, 97–98 hàm
munmap( ), 99 thay đổi
kích thước ánh xạ, 102
Tín hiệu SIGBUS và SIGSEGV, 99 đồng
bộ hóa tệp với ánh xạ, 104 hàm memrchr(),
272 hàm memset(), 269 hợp
nhất (trình lập lịch I/
O), 115 siêu dữ liệu, 196 di
chuyển quy
trình, chi phí, 173
Kiểu MIME, lưu trữ, 205 hàm
mincore(), 276 hàm mkdir(),
218, 229 hàm mktime(), 320
hàm mlock(), 274 hàm
mlockall(), 275 hàm
mmap(), 95–99, 258 ưu điểm,
101 nhược điểm, 102 ví dụ, 100
kích thước trang,
97–98 giá trị trả về
và mã lỗi, 99
đối số chế độ, 27,
65 thời gian đơn điệu, 308 điểm
gắn kết, 14 gắn kết, 14
hàm mprotect(), 104 hàm
mremap(), 102 giá
trị trả về và
mã lỗi, 103 hàm msync(),
105 giá trị trả về và mã
lỗi, 105 I/O đa kênh, 47–57 đa nhiệm,
163 lập trình đa luồng,
166 hàm munmap( ), 99, 258

N

đường ống được đặt
tên, 13 không gian
tên, 14 không gian tên cho mỗi quy trình, 15
hàm nanosleep(), 326 căn
chỉnh tự nhiên, 71, 252 hệ
thống tập tin mạng, 14
hàm nice(), 170 giá
trị nice, 169 từ
khóa noinline, 340 I/O
không chặn, 32 ghi
không chặn, 35
Trình lập lịch I/O Noop,
119 từ khóa noreturn,
342 thiết bị null, 231

Ô

Trình lập lịch trình O(1), 163
Cờ O_DSYNC, 40
offset,
74 offsetof() macro,
347 off_t
type, 44 on_exit function, 138
Điều kiện OOM (hết bộ nhớ), 278 hàm
open(), 24–26
Cờ O_DSYNC và O_RSYNC, 40
Cờ O_SYNC, 39
Open Software Foundation (OSF), 7 hàm
opendir(), 220 thời hạn
hoạt động, 176 độ trễ và độ
trễ, 177 phân bổ cơ
hội, 277 đối số gốc, lseek(),
42
Cờ O_RSYNC, 40
OSF (Tổ chức phần mềm mở), 7
Cờ O_SYNC, 39 điều
kiện hết bộ nhớ (OOM), 278 giao tiếp ngoài
băng tần, 233 cam kết quá mức, 277

P

thuộc tính đóng gói, 343
trang, 97–98, bộ nhớ
đệm trang 243, bộ
nhớ đệm trang 59 đọc trước,
kích thước
trang 60, ghi lại trang 15, 60
Macro PAGE_SIZE, 98 tham
số truyền, 3

tham số, đánh dấu là không sử dụng, 343
thư mục cha, 212 quy trình
cha, 17 quy trình cha,
127 (xem thêm quy trình)
ghi một phần, 34 đư ờng
dẫn, 11 tên đư ờng
dẫn, 11,
212 hàm pause(), 287
luồng pdflush, 61 tín hiệu
đang chờ, 297 bit
quyền, 18 không gian tên
cho mỗi quy trình, 15
hàm perror(), 21 pgid (ID nhóm
quy trình), 154 pid (ID
quy trình), 16, 126 phân bổ, 127
kiểu pid_t, 128 con trỏ, 255
hàm poll(), 53–56
như ợc điểm, 89 ví
dụ, 55 giá trị
trả về và mã lỗi, 55
select(), so với, 57

Giao diện hệ điều hành di động (xem
(POSIX)
POSIX, 6
lịch sử, 6
bit bảo vệ và kiến trúc, 96
Đồng hồ POSIX, 313–315 loại
clockid_t, 313 độ phân
giải nguồn thời gian, 314
Bộ đếm thời gian dựa trên đồng hồ POSIX, 333–338
kích hoạt bộ đếm thời
gian, 335 tạo bộ đếm thời gian,
333–335 xóa bộ đếm thời
gian, 338 lấy ngày hết hạn bộ đếm thời
gian, 336 lấy ngày hết hạn bộ đếm thời gian, 337
Macro _POSIX_SAVED_IDS, 153 hàm
posix_fadvise(), 108 giá trị trả về
và mã lỗi, 110 hàm ppoll(), 56 hàm
pread(), 44 đa nhiệm ư u
tiên, 163 lập lịch ư u
tiên, 165 nhóm chính, 17 không gian
địa chỉ quy trình, 243–245 tệp
đư ợc ánh xạ, 245 vùng
bộ nhớ, 245 trang và phân trang, 243
chia sẻ và sao chép
khi ghi, 244 ID quy trình
(pid), 16
thời gian xử lý, 308
cây xử lý, 17 xử
lý, 15–17, 126 truy cập,
18 nhóm xử lý
nền, 154 xử lý con và xử lý cha, 127
xử lý con, đang chờ, 139–149
Các hàm wait3( ) và wait4( )
của BSD, 145
macro con trỏ trạng thái, 140
hàm wait( ), 139 hàm
waitid( ), 143 hàm
waitpid( ), 142 sao chép khi
ghi (COW), 134 daemon, 159–161
học thuyết về quyền ít
đặc quyền nhất, 150 họ hàm exec, 129–132 mô
tả tệp và, 23 nhóm quy trình nền trư ớc,
154 hàm fork( ), 132–136
phân cấp, 127 quy trình khởi tạo, 126

Tiến trình bị ràng buộc I/O,
164 khởi chạy và chờ tiến trình mới,
147 chi phí di
chuyển, 173 đa nhiệm,
163 như ờng chỗ, 166–
169 tiến trình mới,
đang chạy, 129 chức năng nhóm tiến
trình lỗi thời, 158 quyền sở hữu, 127 ư u
tiên (xem trình
lập lịch, ư u tiên) nhóm tiến
trình, 128 lệnh
gọi hệ thống nhóm tiến
trình, 157 nhóm tiến trình, 154–155
ID nhóm tiến trình (pgid),
154 phân cấp tiến trình, 16 ID
tiến trình (pid), 126
phân bổ, 127 ID tiến
trình và ID tiến
trình cha, thu thập, 128 tiến trình bị
ràng buộc bởi bộ
xử lý, 164 đổi cha, 149 giới hạn tài
nguyên, 190–195 giới
hạn cứng và mềm mặc định, 193
Linux, giới hạn tài nguyên đư ợc cung
cấp bởi, 191–
193 thiết lập và truy xuất giới hạn,
194 giới hạn mềm và cứng, 190
danh sách
chạy, 162 quy trình có thể chạy, 162

tiến trình (tiếp theo)
trình lập lịch (xem trình lập
lịch) phiên, 154–157
kết thúc, 136–139
hàm atexit( ), 137 theo
tín hiệu, 137
phư ơ ng pháp cổ điển,
137 hàm exit( ) và _exit( ), 136 hàm
kill theo kernel, 137
hàm on_exit( ), 138
SIGCHILD, 139 luồng,
166 timeslices,
162, 164 ngư ời dùng và
nhóm, 149–154 thay đổi ID,
phư ơ ng pháp BSD, 152 thay đổi ID,
phư ơ ng pháp HP-UX, 152 lấy ID ngư ời dùng
và nhóm, 154 thao tác ID ngư ời dùng/nhóm
đư ợc ư u tiên, 153 ID ngư ời
dùng và nhóm thực, có
hiệu lực và đã lư u, 150 ID ngư ời dùng
hoặc nhóm thực,
có hiệu lực, đang thay đổi, 151 ID
ngư ời dùng và
nhóm thực hoặc đã lư u, đang thay đổi,
151 hỗ trợ cho
ID ngư ời dùng đã lư u, 153 hàm
vfork( ), 135 zombie, 17,
149 đang chờ tiến
trình zombie, 139
(xem thêm quy trình con; quy trình
cha) (xem
thêm hệ thống thời gian thực)
mối quan hệ bộ xử lý, 172–176
hàm sched_getaffinity( ) và sched_setaffinity,
173–176 lập trình lập
trình đa luồng,
166 khái niệm lập trình, 9–22 xử lý
lỗi, 19–22 tệp, 9–15 hệ thống tệp
và không gian tên, 14 tiêu
đề, 19 giao
tiếp giữa các tiến trình, 19 quyền,
18 tiến trình,
15–17 tín hiệu, 19 ngư ời dùng và nhóm,
17

chư ơ ng
trình các vùng quan
trọng, 297 cờ bảo vệ và kiến trúc, 96
Cờ PROT_READ và PROT_EXEC, 96 hàm pselect(), 52
hàm psignal(), 290 API
pthreads, 166 hàm
thuần túy, 341 từ khóa
thuần túy, 341 hàm
pwrite(), 45

R

hàm raise(), 292 tín
hiệu cho, 303
trình tạo số ngẫu nhiên, 232 hàm
read(), 29–33 giá trị lỗi
bổ sung, 32 lần đọc không chặn,
32 lần đọc theo vị trí, 44
giá trị lỗi, 45 đọc tất
cả các byte, 31
giá trị trả về, 30 giới hạn
kích thư ớc, 33 đọc
hàng đợi FIFO,
116 độ trễ đọc, 116 quyền
đọc, 18 readahead,
60, 107 hàm readahead(),
110 giá trị trả về và
mã lỗi, 110 hàm readdir(), 221,
222 hàm readv(), 84

thực hiện, 88 giá trị
trả về, 85 gid thực,
17 thời gian
thực, 308 uid
thực, 17 ID
ngư ời dùng thực,
150 hàm realloc( ), 249 hệ
thống thời gian thực, 176–189
tính quyết định, 187–189
Mối quan hệ CPU và các quy
trình thời gian
thực, 188 dữ liệu lỗi trư ớc và
bộ nhớ khóa, 187
độ trễ, độ trễ và thời hạn, 177 quy
trình thời gian thực, các biện pháp phòng ngừa
với, 186 sched_rr_get_interval,
185 tham số lập lịch, cài đặt, 182–185 phạm vi
ư u tiên hợp lệ,
xác định, 184–185 chính
sách lập lịch và ư u tiên, 178–182 chính sách lập
lịch hàng loạt, 180
Lớp FIFO, 179 chính
sách bình thư ờng, 180
Lớp RR (vòng tròn), 179 thiết
lập, 180–182

hệ thống thời gian thực mềm so với cứng, 176 hỗ trợ trong Linux, 178 (xem thêm quy trình; trình lập lịch)
bản ghi, 9 khả năng nhập lại, 293 hàm đảm bảo khả năng nhập lại, 294
tệp thông thư ờng, 9 tên đư ờng dẫn tư ơ ng đối, 11, 212
định dạng thời gian tư ơ ng đối, 309 hàm remove(), 228 hàm removexattr(), 211 hàm rename(), 229
thay đổi cha mẹ, 17 thay đổi cha mẹ của quy trình, 149 giới hạn tài nguyên của quy trình, 190–195 giới hạn cứng và mềm mặc định, 193 Linux, giới hạn tài nguyên đư ợc cung cấp bởi, 191–193 thiết lập và truy xuất giới hạn, 194 giới hạn mềm và cứng, 190 hàm rewind(), 74 cấu trúc rlimit, 190 RLIMIT_CPU, 190 hàm rmdir( ), 219 thư mục gốc, 11, 212 hệ thống tập tin gốc, 14 root (ngư ời dùng gốc), 17
lớp vòng tròn (RR), 179 danh sách chạy, 162 quy trình có thể chạy, 162

S

ID nhóm đã lư u (gid), 17
ID ngư ời dùng đã lư u (uid), 17, 151 hàm sbrk( ), 256 I/O phân tán/thu thập, 84–89 lợi thế, 84 hàm readv( ) và writev( ), 84 triển khai, 88 giá trị trả về, 85
SCHED_BATCH, 180 hàm sched_getaffinity() và sched_setaffinity(), 173–176 hàm sched_getparam() và sched_setparam(), 182–185 mã lỗi, 183 hàm sched_getscheduler() và sched_ setscheduler(), 180–182

LỊCH TRÌNH KHÁC, 180
LỊCH_ĐỊNH_RR, 179
sched_rr_get_interval, 185 mã lỗi, 186 trình lập lịch, 162–166 cân bằng tải, 173 đa nhiệm, 163 O(1) trình lập lịch trình, 163 lập lịch trình ư u tiên, 165 ư u tiên quy trình, 169–172 hàm getpriority( ) và setpriority( ), 171 Ư u tiên I/O, 172 hàm nice(), 170 mối quan hệ bộ xử lý, 172–176 sched_getaffinity() và sched_ hàm setaffinity, 173–176
sched_rr_get_interval, 185 tham số lập lịch, thiết lập, 182–185 phạm vi ư u tiên hợp lệ, xác định, 184–185 chính sách lập lịch, 178–182 chính sách lập lịch hàng loạt, 180 lớp FIFO, 179 chính sách bình thư ờng, 180 lớp RR (vòng tròn), 179 thiết lập, 180–182 (xem thêm quy trình; hệ thống thời gian thực) hàm sched_yield( ), 166 cách sử dụng hợp pháp, 167 Linux phiên bản 2.6, thay đổi trong, 168 sector, 14 không gian tên bảo mật, 206 vi phạm phân đoạn, tín hiệu cho, 284 phân đoạn, 245 hàm select( ), 48–53 như ợc điểm, 89 poll( ), so với, 57 sử dụng để ngủ, 329
vị trí tuần tự, 60 phiên, 154–157 lệnh gọi hệ thống phiên, 156 hàm setegid(), 152 hàm seteuid(), 152, 153 hàm setitimer(), 282, 284, 332 hàm setpgid(), 157 hàm setpgrp(), 158 hàm setresuid(), 153 hàm setreuid(), 152 hàm setrlimit(), 190 hàm setsid(), 156 tham số setsize, 174 hàm settimeofday(), 318

hàm setuid(), 151, 153 hàm
setxattr(), 208 trư ờng
si_code(), 302
giá trị hợp lệ cho SIGBUS,
303 hàm sigaction(), 298–300
hàm sigaddset(), 296 hàm
sigandset(), 296
Tín hiệu SIGBUS, 99
SIGCHILD, 139
Tín hiệu SIGCONT, hàm
292 sigdelset(), hàm 296
sigemptyset(), hàm 296
sigfillset(), hàm 296
SIGHUP, 154
cấu trúc siginfo_t, 300–302
SIGINT, 154
hàm sigisemptyset(), 296 hàm
sigismember(), 296 hàm signal()
và sigaction(), 139 hàm signal(), 286, 307
tín hiệu, 19, 279–286 tín
hiệu chặn, 296–298
truy xuất tín hiệu đang chờ,
297 chờ một tập hợp tín hiệu, 298
khái niệm, 280 định danh, 280
vùng quan trọng
và, 297 giá trị
có thể đọc đư ợc bằng con
ngư ời so với giá trị số nguyên,
281
Linux, đư ợc hỗ trợ bởi, 281–
286 liệt kê với lệnh kill -l, 281
tải trọng, gửi tín hiệu với, 305 ví
dụ, 306 khả
năng nhập lại,
293 hàm có khả năng nhập lại đư ợc
đảm bảo, 294 gửi,
291–293 ví dụ,
292 quyền, 292
cho một nhóm quy trình,
293 cho chính bạn, 292
SIGINT và SIGTERM, 280
SIGKILL và SIGSTOP, 280 quản
lý tín hiệu, 286–291, 298–305 ví dụ, 287
thực thi và kế
thừa, 289 ánh xạ số tín hiệu
thành chuỗi, 290 trư ờng
si_code, 302–
305 hàm sigaction(),
298–300 cấu trúc siginfo_t, 300–
302 chờ tín hiệu, 287
mặt nạ tín hiệu,
297 bộ tín
hiệu, 295 an toàn
tín hiệu, 294 hàm
sigorset(), 296 hàm
sigpending(), 298 hàm
sigprocmask(), 297 hàm sigqueue(), 305
Tín hiệu SIGSEGV,
hàm sigsuspend() 99, 298
Đặc tả Unix đơ n (xem SUS) hàm
sleep(), 324 ngủ, 324–
330 ổ cắm, 13 mối
quan hệ mềm,
173 liên kết mềm,
225 hệ thống
thời gian thực mềm, 176 giới
hạn tài nguyên mềm, 190
đồng hồ phần mềm, 309
sắp xếp (bộ lập lịch I/O), 115
khả năng tư ơ ng thích
nguồn, 5 tệp đặc
biệt, 13
ngăn xếp, 245 chuỗi trùng lặp
trên, 266 phân bổ bộ nhớ dựa trên ngăn xếp,
264–268 chuỗi, trùng lặp,
266 mảng có độ dài thay đổi,
267 lỗi chuẩn (stderr), 21 I/
O chuẩn, 64 con
trỏ tệp, 65 hạn
chế, 81 thư
viện I/O chuẩn, 64 tiêu
chuẩn, 6 như
đã đề cập trong cuốn sách này,
8 hàm stat(), 197 họ
stat, 196–199 cấu trúc
stat, 197 trư ờng,
197–199 tĩnh ư u
tiên, 178 con trỏ
trạng thái, 140
stderr (lỗi chuẩn), 21
stdin, stdout và các mô tả tệp stderr, 23
stdio, 64
hàm stime(), 318 hàm
strdup(), strdupa() và strndupa(), 266
luồng, 65 các
mô tả tệp
liên quan, lấy for, 77 đóng, 67 đóng
tất cả,
67 các mô
tả tệp, mở via,
66

xả, 75 đọc từ,
67–70 đặt lại ký tự, 68
đọc toàn bộ một dòng, 68 đọc các
chuỗi tùy ý, 69 đọc dữ liệu
nhị phân, 70 đọc từng ký tự một
lần, 67 tìm kiếm một luồng,
74 lấy vị trí luồng hiện tại, 75 ghi vào,
70–72 căn chỉnh dữ liệu,
71 ghi một ký tự đơ n, 71 ghi một chuỗi, 72
ghi dữ liệu nhị phân,
72 (xem thêm I/O đệm)

hàm strerror(), 21 hàm
strerror_r(), 21 hàm
strsignal(), 290 thư mục
con, 212 nhóm bổ sung,
17
SUS (Đặc tả Unix đơ n), 6 lịch sử, 6
tiêu chuẩn
UNIX 95, UNIX 98 và
UNIX 03, 7
liên kết tượng trưng (symlink), 12, 223, 225
hàm symlink(), 226 xử lý đa
đối xứng, 172 hàm sync(), 39 đồng bộ
các hoạt động ghi, 112
đồng bộ hóa, 39 hoạt động đồng bộ, 111 hoạt
động ghi đồng bộ, 111
hàm sysconf(), 98 sys_siglist, 290
lệnh gọi hệ thống (syscall), 3

Gọi I/O và, 77 đồng
hồ hệ thống, điều chỉnh, 321–324
không gian tên hệ thống,
205 lập trình hệ thống, xi, 1–4
Trình biên dịch C, 4
Thư viện C (libc),
4 hàm
truyền tham số, 3
khái niệm lập trình (xem khái niệm lập trình)
tiêu chuẩn,
6 lệnh gọi hệ
thống, 3 lệnh
gọi, 3 phần
mềm hệ thống, xi, 1 bộ
đếm thời gian hệ
thống, 309 tần số bộ đếm thời gian hệ thống, 309

T

vị trí thời gian, 59 tệp
văn bản và tệp nhị phân, 66 phân
đoạn văn bản, 245
Ngôn ngữ lập trình C, xi
Nhóm mở, 6 I/O không
đồng bộ dựa trên luồng, 113 luồng, 79, 166
API pthreads, 166
an toàn luồng, 79–81
khóa tệp thủ công, 80
hoạt động luồng không khóa,
81 an toàn luồng, 79 tích tắc hoặc
jiffy, 309 thời
gian, 308–310

Các hàm chuyển đổi ngôn
ngữ C, 320–321 thời
gian hiện tại, lấy, 315–318 độ phân
giải micro giây, 316 độ phân giải
nano giây, 316 thời gian hiện
tại, thiết lập, 318 hàm
clock_settime(), 319 cấu trúc dữ
liệu, 310–313 kiểu clock_t,
313 timespec (độ
chính xác nano giây), 311 time_t, 310 time_t
và năm nhuận,
315 timeval (độ chính xác micro
giây), 311 cấu trúc tm cho thời gian bị chia
nhỏ
biểu diễn, 312 delta,
322 phép đo
thời đại, 309 hạt nhân, phép
đo bằng, 308 định dạng phép đo, 309

Đồng hồ POSIX, 313–315 loại
clockid_t, 313 độ phân
giải nguồn thời gian, 314
thời gian xử lý, nhận được,
317 ngủ, 324–330 các
giải pháp thay thế cho,
330 độ chính xác micro giây, 325
độ chính xác nano giây, 326–329 hàm
select( ) để di chuyển, 329 tràn bộ đếm
thời gian, 329 đồng hồ
hệ thống, điều chỉnh, 321–324 bộ đếm
thời gian, 330–338
báo động, 330
bộ đếm thời gian khoảng cách, 331–333
Bộ đếm thời gian dựa trên đồng hồ POSIX,
333–338 hàm time(), 315
hàm timer_create(), 334 hàm
timer_delete(), 338

hàm timer_getoverrun(), 337 hàm
timer_gettime(), 336 hàm
timer_settime(), 335 hàm times(),
317 timeslices(), 162,
164 kiểu time_t(),
321 toolchain(),
6 hàm
truncate(), 46 cắt bớt,
10 không gian
tên đáng tin cậy, 206 từ
khóa typeof(), 346

Bạn

uid (ID người dùng),
17 umask, 218
khóa không xác định, 204
Giờ quốc tế, Phối hợp (UTC), 309
Unix, 1
Trình soạn thảo văn bản
Unix, trình bao xi Unlike( ),
345 hàm unlink( ), 227 hủy
liên kết, 12 hủy
gắn kết, 14 thuộc
tính không sử dụng, 343 ID
người dùng (uid),
17 không gian tên người
dùng, 206 thời gian
người dùng, 40, 317 I/O được
đệm bởi người dùng, 62–64 mô tả tệp I/
O (đầu vào/đầu ra) được đệm bởi
người dùng, cách
sử dụng, 23
tên người dùng, 17 người dùng, 17
quyền sở hữu các quy trình, 127 ứng dụng không gian
người
dùng, giao tiếp với hạt nhân, 3 hàm usleep( ), 325
UTC (Giờ quốc tế, Phối hợp), 309

V

trang hợp lệ,
244 mảng có độ dài thay đổi (VLA),
267 hàm variadic, 129 I/
O vectơ , 84, 86
vectơ , 85
hàm vfork(), 135
VFS (xem hệ thống tệp ảo)
không gian địa chỉ ảo, 243
chuyển đổi tệp ảo, 58
hệ thống tệp ảo (VFS), 14, 58

T

hàm wait(), 139 hàm
waitid(), 143 chờ các
tiến trình zombie, 139 hàm waitpid(),
142 thời gian chờ, 308
thuộc tính
warn_unused_result, 342 theo dõi, 236–
242 thêm theo dõi,
236 tùy chọn nâng cao,
240 sự kiện inotify, 238–
240 sự kiện nâng cao, 239
liên kết các sự kiện di
chuyển, 240 đọc, 238 theo dõi mặt
nạ, 236
whence, 74 kích
thư ớc từ, 44

-Tùy chọn Wpointer-arith, hàm
write() 349, 33–37
mã lỗi bổ sung, 35 chế độ
thêm, 34 hành vi
của write(), 36 ghi không
chặn, 35 ghi một phần,
34 ghi theo vị
trí, 44 giá trị lỗi,
45 giới hạn kích
thư ớc trên, 36
hàng đợi FIFO ghi, 116
thứ tự ghi, 36 quyền
ghi, 18 ghi lại, 36 vấn
đề ghi-đói-đọc,
116 hàm writev(), 85 ví dụ, 86 triển
khai, 88 giá trị trả
về, 85

X

XOpen, 7
xattrs (xem các thuộc tính mở
rộng) xmalloc( ) wrapper, 247

Có

tạo ra, 163, 166–169 sử
dụng hợp pháp, 167
Linux phiên bản 2.6, thay đổi trong, 168

Z

thiết bị zero,
232 zombie, 17,
149 đang chờ tiến trình zombie, 139

Về tác giả

Robert Love là ngư ời dùng và hacker Linux từ những ngày đầu. Anh ấy tích cực tham gia—và đam mê—cộng đồng Linux kernel và máy tính để bàn GNOME. Những đóng góp gần đây của anh ấy cho Linux kernel bao gồm công việc về lớp sự kiện kernel và inotify. Những đóng góp liên quan đến GNOME bao gồm Beagle, GNOME Volume Manager, NetworkManager và Project Utopia. Hiện tại, Robert làm việc tại Văn phòng Chư ơ ng trình Nguồn mở tại Google.

Với tư cách là tác giả, Robert chịu trách nhiệm về Phát triển hạt nhân Linux (Novell Press), hiện đang ở phiên bản thứ hai. Ông cũng là đồng tác giả của phiên bản thứ năm của Linux in a Nutshell của O'Reilly. Là biên tập viên cộng tác cho Linux Journal, Robert đã viết nhiều bài báo và đư ợc mời đến nói chuyện trên khắp thế giới về Linux.

Robert tốt nghiệp Đại học Florida với bằng Cử nhân Toán học và bằng Cử nhân Khoa học Máy tính. Đến từ Nam Florida, anh hiện coi Boston là quê hư ơ ng.

Bản quyền

Máy bay cũng được gọi là chim cơ khí và tàu bay, và đôi khi được gọi bằng những cái tên hoa mỹ hơn như Artificial Albatross. Sự nhiệt tình đối với máy bay vẫn còn cao, vì những người đam mê hàng không vẫn chế tạo những máy bay đầu tiên ngày nay.

Hình ảnh bìa và hình ảnh mở đầu chương được lấy từ Kho lưu trữ hình ảnh Dover.

Phông chữ bìa là Adobe ITC Garamond. Phông chữ văn bản là Linotype Birka; phông chữ tiêu đề là Adobe Myriad Condensed; và phông chữ mã là TheSans Mono Condensed của LucasFont.