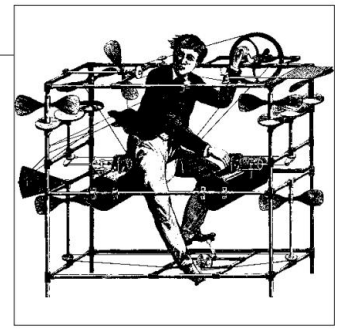
CHƯƠNG 3

Bộ đệm I/O



Nhớ lại từ Chương 1 rằng khối, một trừu tượng hệ thống tập tin, là ngôn ngữ chung của I/O—tất cả các hoạt động đĩa đều diễn ra theo khối. Do đó, hiệu suất I/O là tối ưu khi các yêu cầu được đưa ra trên các ranh giới được căn chỉnh theo khối theo bội số nguyên của kích thước khối.

Sự suy giảm hiệu suất trở nên trầm trọng hơn do số lượng lệnh gọi hệ thống tăng lên để đọc một byte đơn lẻ 1.024 lần thay vì 1.024 byte cùng một lúc.
Ngay cả một loạt các hoạt động được thực hiện trong một kích thước lớn hơn một khối cũng có thể không tối ưu nếu kích thước không phải là bội số nguyên của kích thước khối. Ví dụ, nếu kích thước khối là một kilobyte, các hoạt động trong các khối 1.130 byte vẫn có thể chậm hơn các hoạt động 1.024 byte.

I/O đệm của người dùng

Các chương trình phải đưa ra nhiều yêu cầu I/O nhỏ tới các tệp thông thường thường thực hiện I/O đệm người dùng. Điều này đề cập đến việc đệm được thực hiện trong không gian người dùng, hoặc thủ công bởi ứng dụng, hoặc trong suốt trong thư viện, không phải đệm do nhân thực hiện. Như đã thảo luận trong Chương 2, vì lý do hiệu suất, nhân đệm dữ liệu nội bộ bằng cách trì hoãn ghi, hợp nhất các yêu cầu I/O liền kề và đọc trước. Thông qua các phương tiện khác nhau, việc đệm người dùng cũng nhằm mục đích cải thiện hiệu suất.

Hãy xem xét một ví dụ sử dụng chương trình không gian người dùng dd:

```
dd bs=1 count=2097152 nếu=/dev/zero của=pirate
```

Do tham số bs=1 , lệnh này sẽ sao chép hai megabyte từ thiết bị /dev/zero (một thiết bị ảo cung cấp luồng số không vô tận) vào file pirate theo 2.097.152 khối một byte. Nghĩa là, nó sẽ sao chép dữ liệu thông qua khoảng hai triệu thao tác đọc và ghi—mỗi lần một byte.

Bây giờ hãy xem xét bản sao hai megabyte tương tự, nhưng sử dụng các khối 1.024 byte:

```
dd bs=1024 count=2048 if=/dev/zero of=pirate
```

Hoạt động này sao chép cùng hai megabyte vào cùng một tệp, nhưng lại phát sinh 1.024 lần
ít thao tác đọc và ghi hơn. Cải thiện hiệu suất là rất lớn, vì bạn có thể
xem trong Bảng 3-1. Ở đây, tôi đã ghi lại thời gian thực hiện (sử dụng ba biện pháp khác nhau)
bằng bốn lệnh dd chỉ khác nhau về kích thước khối. Thời gian thực là tổng thời gian đã trôi qua
thời gian đồng hồ treo tường, thời gian người dùng là thời gian dành cho việc thực thi mã chương trình trong người dùng
không gian và thời gian hệ thống là thời gian dành cho việc thực hiện các cuộc gọi hệ thống trong không gian hạt nhân trên
thay mặt cho quá trình.

Bảng 3-1. Ảnh hưởng của kích thước khối đến hiệu suất

Kích thước khối	Thời gian thực	Thời gian sử dụng	Thời gian hệ thống
1	18.707 giây	1,118 giây	17.549 giây
byte 1.024	0,025 giây	0,002 giây	0,023 giây
byte 1.130 byte	0,035 giây	0,002 giây	0,027 giây

Sử dụng các khối 1.024 byte dẫn đến cải thiện hiệu suất rất lớn so với khối byte đơn. Tuy nhiên, bảng cũng
chứng minh rằng sử dụng
kích thước khối lớn hơn—có nghĩa là thậm chí ít lệnh gọi hệ thống hơn—có thể dẫn đến hiệu suất
sự suy thoái nếu các hoạt động không được thực hiện theo bội số của kích thước khối đĩa.
Mặc dù yêu cầu ít cuộc gọi hơn, các yêu cầu 1.130 byte cuối cùng vẫn tạo ra các yêu cầu không liên kết
yêu cầu và do đó kém hiệu quả hơn so với yêu cầu 1.024 byte.

Để tận dụng lợi thế về hiệu suất này đòi hỏi phải có kiến thức trước về khả năng
kích thước khối vật lý. Kết quả trong bảng cho thấy kích thước khối có khả năng cao nhất là 1.024,
một bội số nguyên của 1.024 hoặc một ước số của 1.024. Trong trường hợp của /dev/zero, khối
kích thước thực tế là 4.096 byte.

Kích thước khối

Trên thực tế, các khối thường có kích thước là 512, 1.024, 2.048 hoặc 4.096 byte.

Như Bảng 3-1 chứng minh, hiệu suất tăng đáng kể chỉ đạt được bằng cách thực hiện các hoạt động trong các khối
là bội số nguyên hoặc ước số của kích thước khối. Điều này
là vì hạt nhân và phần cứng nói theo thuật ngữ khối. Vì vậy, sử dụng kích thước khối
hoặc một giá trị phù hợp hoàn toàn bên trong một khối đảm bảo các yêu cầu I/O được căn chỉnh theo khối và
ngăn chặn công việc không cần thiết bên trong hạt nhân.

Việc xác định kích thước khối cho một thiết bị nhất định rất dễ dàng bằng cách sử dụng lệnh gọi hệ thống stat()
(được đề cập trong Chương 7) hoặc lệnh stat(1) . Tuy nhiên, hóa ra là bạn không
thường cần biết kích thước khối thực tế.

Mục tiêu chính khi chọn kích thước cho các hoạt động I/O của bạn là không chọn một kích thước kỳ quặc
kích thước chẳng hạn như 1.130. Không có khối nào trong lịch sử Unix có kích thước là 1.130 byte và việc chọn
kích thước như vậy cho các hoạt động của bạn sẽ dẫn đến I/O không được căn chỉnh sau yêu cầu đầu tiên.
Tuy nhiên, việc sử dụng bất kỳ số nguyên bội hoặc số chia nào của kích thước khối sẽ ngăn chặn việc không căn chỉnh
yêu cầu. Miễn là kích thước bạn chọn giữ mọi thứ được căn chỉnh theo khối, hiệu suất
sẽ tốt. Nhiều hơn sẽ chỉ dẫn đến ít cuộc gọi hệ thống hơn.

Do đó, lựa chọn dễ nhất là thực hiện I/O bằng cách sử dụng kích thước bộ đệm lớn là bội số của kích thước khối thông thường. Cả 4.096 và 8.192 byte đều hoạt động tốt.

Vấn đề, tất nhiên, là các chương trình hiếm khi xử lý theo khối. Các chương trình làm việc với các trường, dòng và ký tự đơn, không phải các khái niệm trừu tượng như khối. Như đã mô tả trước đó, để khắc phục tình trạng này, các chương trình sử dụng I/O đệm người dùng. Khi dữ liệu được ghi, dữ liệu được lưu trữ trong bộ đệm bên trong không gian địa chỉ của chương trình. Khi bộ đệm đạt đến một kích thước cụ thể—kích thước bộ đệm —toàn bộ bộ đệm được ghi ra trong một thao tác ghi duy nhất. Tương tự như vậy, dữ liệu được đọc vào bằng các khối có kích thước bộ đệm, được căn chỉnh theo khối. Khi ứng dụng đưa ra các yêu cầu đọc có kích thước lẻ, các khối của bộ đệm được phân phối từng phần. Cuối cùng, khi bộ đệm trống, một khối lớn khác được căn chỉnh theo khối sẽ được đọc vào. Nếu bộ đệm này có kích thước phù hợp, thì sẽ nhận ra được lợi ích hiệu suất to lớn.

Có thể triển khai bộ đệm người dùng bằng tay trong các chương trình của riêng bạn. Thật vậy, nhiều ứng dụng quan trọng thực hiện chính xác điều này. Tuy nhiên, phần lớn các chương trình đều sử dụng thư viện I/O chuẩn phổ biến (một phần của thư viện C chuẩn), cung cấp giải pháp bộ đệm người dùng mạnh mẽ và có khả năng.

I/O chuẩn

Thư viện C chuẩn cung cấp thư viện I/O chuẩn (thường được gọi đơn giản là stdio), cung cấp giải pháp đệm người dùng độc lập với nền tảng. Thư viện I/O chuẩn dễ sử dụng nhưng mạnh mẽ.

Không giống như các ngôn ngữ lập trình như FORTRAN, ngôn ngữ C không bao gồm bất kỳ hỗ trợ tích hợp hoặc từ khóa nào cung cấp bất kỳ chức năng nào tiên tiến hơn kiểm soát luồng, số học, v.v. - chắc chắn không có hỗ trợ cố hữu nào cho I/O. Khi ngôn ngữ lập trình C phát triển, người dùng đã phát triển các bộ quy trình chuẩn để cung cấp chức năng cốt lõi, chẳng hạn như thao tác chuỗi, quy trình toán học, chức năng thời gian và ngày tháng và I/O. Theo thời gian, các quy trình này đã trưởng thành và với việc phê chuẩn tiêu chuẩn ANSI C vào năm 1989 (C89), cuối cùng chúng đã được chính thức hóa thành thư viện C chuẩn. Mặc dù cả C95 và C99 đều bổ sung một số giao diện mới, nhưng thư viện I/O chuẩn vẫn tương đối không thay đổi kể từ khi được tạo ra vào năm 1989.

Phần còn lại của chương này thảo luận về I/O đệm người dùng liên quan đến I/O tệp và được triển khai trong thư viện C chuẩn—tức là mở, đóng, đọc và ghi tệp thông qua thư viện C chuẩn. Việc ứng dụng sẽ sử dụng I/O chuẩn, giải pháp đệm người dùng tự triển khai hay lệnh gọi hệ thống trực tiếp là quyết định mà các nhà phát triển phải đưa ra cẩn thận sau khi cân nhắc nhu cầu và hành vi của ứng dụng.

Các tiêu chuẩn C luôn để lại một số chi tiết cho mỗi lần triển khai và các lần triển khai thường thêm các tính năng bổ sung. Chương này, giống như phần còn lại của cuốn sách, ghi lại các giao diện và hành vi khi chúng được triển khai trong glibc trên hệ thống Linux hiện đại. Điều này được ghi chú khi Linux đi chệch khỏi tiêu chuẩn cơ bản.

Con trỏ tập tin

Các thói quen I/O chuẩn không hoạt động trực tiếp trên các mô tả tệp. Thay vào đó, chúng sử dụng mã định danh duy nhất của riêng chúng, được gọi là con trỏ tệp. Bên trong thư viện C, con trỏ tệp ánh xạ tới một mô tả tệp. Con trỏ tệp được biểu diễn bằng một con trỏ tới FILE typedef, được định nghĩa trong <stdio.h>.

Theo cách nói I/O chuẩn, một tệp mở được gọi là luồng. Luồng có thể được mở để đọc (luồng đầu vào), ghi (luồng đầu ra) hoặc cả hai (luồng đầu vào/đầu ra).

Mở tập tin

Các tập tin được mở để đọc hoặc ghi thông qua fopen( ):

```
#include <stdio.h>

TỆP * fopen (const char *đường dẫn, const char *chế độ);
```

Hàm này mở đường dẫn tệp theo các chế độ đã cho và liên kết một luồng mới với nó.

Chế độ

Đối số mode mô tả cách mở tệp đã cho. Đây là một trong các chuỗi sau:

r

Mở tệp để đọc. Luồng được định vị ở đầu tệp.

r+

Mở tệp để đọc và ghi. Luồng được định vị ở đầu tệp.

~~

Mở tệp để ghi. Nếu tệp tồn tại, tệp sẽ bị cắt bớt thành độ dài bằng không. Nếu tệp không tồn tại, tệp sẽ được tạo. Luồng được định vị ở đầu tệp.

v+

Mở tệp để đọc và ghi. Nếu tệp tồn tại, tệp sẽ bị cắt bớt thành độ dài bằng không. Nếu tệp không tồn tại, tệp sẽ được tạo. Luồng được định vị ở đầu tệp.

Một

Mở tệp để ghi ở chế độ thêm vào. Tệp được tạo nếu nó không tồn tại.
Luồng được định vị ở cuối tệp. Tất cả các lệnh ghi sẽ được thêm vào tệp.

một+

Mở tệp để đọc và ghi ở chế độ thêm. Tệp được tạo nếu tệp không tồn tại. Luồng được định vị ở cuối tệp. Tất cả các lệnh ghi sẽ thêm vào tệp.

Chế độ được cung cấp cũng có thể chứa ký tự b, mặc dù giá trị này luôn bị bỏ qua trên Linux. Một số hệ điều hành xử lý tệp văn bản và tệp nhị phân khác nhau và chế độ b hướng dẫn tệp được mở ở chế độ nhị phân. Linux, giống như tất cả các hệ thống tuân thủ POSIX, xử lý tệp văn bản và tệp nhị phân giống hệt nhau.

Khi thành công, fopen() trả về một con trỏ FILE hợp lệ . Khi thất bại, nó trả về NULL và đặt errno một cách thích hợp.

Ví dụ, đoạn mã sau mở /etc/manifest để đọc và liên kết nó với luồng:

```
TẬP TIN *stream;

stream = fopen ("/etc/manifest", "r"); nếu (!stream) /
* lỗi */
```

Mở một Luồng thông qua File Descriptor

Hàm fdopen() chuyển đổi một mô tả tệp (fd) đã mở thành một luồng:

```
#include <stdio.h>

TỆP * fdopen (int fd, const char *mode);
```

Các chế độ có thể giống như đối với fopen( ) và phải tương thích với các chế độ ban đầu được sử dụng để mở mô tả tệp. Các chế độ w và w+ có thể được chỉ định, nhưng chúng sẽ không gây ra sự cắt xén. Luồng được định vị tại vị trí tệp được liên kết với mô tả tệp.

Sau khi một mô tả tệp được chuyển đổi thành luồng, I/O không còn được thực hiện trực tiếp trên mô tả tệp nữa. Tuy nhiên, việc thực hiện như vậy là hợp pháp. Lưu ý rằng mô tả tệp không bị trùng lặp mà chỉ được liên kết với luồng mới. Đóng luồng cũng sẽ đóng mô tả tệp.

Nếu thành công, fdopen() trả về một con trỏ tệp hợp lệ; nếu thất bại, nó trả về NULL.

Ví dụ, đoạn mã sau mở /home/kidd/map.txt thông qua lệnh gọi hệ thống open() , sau đó sử dụng trình mô tả tệp sao lưu để tạo luồng liên kết:

```
TỆP *stream; int
fd;

fd = mở ("/home/kidd/map.txt", O_RDONLY); nếu (fd ==
&#8722;1) /* lỗi */


luồng = fdopen (fd, "r"); nếu (!
stream) /* lỗi
           */
```

Đóng luồng

Hàm fclose() đóng một luồng nhất định:

```
#include <stdio.h>

int fclose (TỆP *stream);
```

Bất kỳ dữ liệu nào được đệm và chưa được ghi trước tiên sẽ được xóa. Khi thành công, fclose() trả về 0. Khi thất bại, nó trả về EOF và đặt errno một cách thích hợp.

Đóng tất cả các luồng

Hàm fcloseall() đóng tất cả các luồng liên quan đến quy trình hiện tại, bao gồm luồng chuẩn vào, luồng chuẩn ra và luồng chuẩn lỗi:

```
#xác định _GNU_SOURCE

#include <stdio.h>

int fcloseall (không có giá trị);
```

Trước khi đóng, tất cả các luồng đều được xả. Hàm này luôn trả về 0; nó dành riêng cho Linux.

Đọc từ một luồng

Thư viện C chuẩn triển khai nhiều hàm để đọc từ một luồng mở, từ phổ biến đến bí truyền. Phần này sẽ xem xét ba cách tiếp cận phổ biến nhất để đọc: đọc từng ký tự một, đọc toàn bộ một dòng một và đọc dữ liệu nhị phân. Để đọc từ một luồng, luồng đó phải được mở dưới dạng luồng đầu vào với chế độ thích hợp; nghĩa là bất kỳ chế độ hợp lệ nào ngoại trừ w hoặc a.

Đọc một ký tự tại một thời điểm Thường thì

mô hình I/O lý tưởng chỉ đơn giản là đọc một ký tự tại một thời điểm. Hàm fgetc() được sử dụng để đọc một ký tự duy nhất từ một luồng:

```
#include <stdio.h>

int fgetc (FILE *stream);
```

Hàm này đọc ký tự tiếp theo từ luồng và trả về dưới dạng unsigned char ép kiểu thành int. Việc ép kiểu được thực hiện để có phạm vi đủ để thông báo kết thúc tệp hoặc lỗi: EOF được trả về trong các điều kiện như vậy. Giá trị trả về của fgetc() phải được lưu trữ trong int. Lưu trữ trong char là một lỗi thường gặp nhưng nguy hiểm.

Ví dụ sau đây đọc một ký tự duy nhất từ luồng, kiểm tra lỗi và sau đó in kết quả dưới dạng char:

```
số nguyên c;

c = fgetc (stream); nếu
(c == EOF) /* lỗi
           */
khác
           printf ("c=%c\n", (ký tự) c);
```

Luồng được luồng trỏ tới phải mở để đọc.

Đưa nhân vật trở lại

I/O chuẩn cung cấp một hàm để đẩy một ký tự trở lại luồng, cho phép bạn "xem trộm" luồng và trả về ký tự đó nếu bạn không muốn:

```
#include <stdio.h>

int ungetc (int c, FILE *stream);
```

Mỗi lệnh gọi đẩy ngược c, ép kiểu thành một unsigned char, vào luồng. Khi thành công, c được trả về; khi thất bại, EOF được trả về. Một lần đọc tiếp theo trên luồng sẽ trả về c. Nếu nhiều ký tự được đẩy ngược, chúng được trả về theo thứ tự ngược lại—tức là ký tự được đẩy cuối cùng sẽ được trả về trước. POSIX quy định rằng chỉ một lần đẩy ngược được đảm bảo thành công mà không có các yêu cầu đọc xen kẽ. Đến lượt mình, một số triển khai chỉ cho phép một lần đẩy ngược duy nhất; Linux cho phép vô số lần đẩy ngược, miễn là bộ nhớ khả dụng. Tất nhiên, một lần đẩy ngược luôn thành công.

Nếu bạn thực hiện lệnh gọi xen kẽ đến một hàm tìm kiếm (xem “Tìm kiếm luồng” sau trong chương này) sau khi gọi ungetc( ) nhưng trước khi đưa ra yêu cầu đọc, lệnh này sẽ khiến tất cả các ký tự bị đẩy lùi bị loại bỏ. Điều này đúng giữa các luồng trong một quy trình duy nhất, vì các luồng chia sẻ bộ đệm.

Đọc toàn bộ một dòng

Hàm fgets() đọc một chuỗi từ một luồng nhất định:

```
#include <stdio.h>

char * fgets (char *str, int kích thước, FILE *stream);
```

Hàm này đọc tối đa một byte nhỏ hơn size từ luồng và lưu trữ kết quả trong str. Một ký tự null (\0) được lưu trữ trong bộ đệm sau khi đọc byte. Việc đọc dừng lại sau khi đạt đến EOF hoặc ký tự xuống dòng. Nếu đọc xuống dòng, \n được lưu trữ trong str.

Nếu thành công, str sẽ được trả về; nếu thất bại, NULL sẽ được trả về.

Ví dụ: char

```
buf[LINE_MAX];

nếu (!fgets (buf, LINE_MAX, stream)) /* lỗi */
```

POSIX định nghĩa LINE_MAX trong <limits.h>: đó là kích thước tối đa của dòng đầu vào mà các giao diện thao tác dòng POSIX có thể xử lý. Thư viện C của Linux không có giới hạn như vậy—các dòng có thể có bất kỳ kích thước nào—nhưng không có cách nào để truyền đạt điều đó bằng định nghĩa LINE_MAX . Các chương trình di động có thể sử dụng LINE_MAX để duy trì sự an toàn; nó được đặt ở mức tương đối cao trên Linux. Các chương trình dành riêng cho Linux không cần phải lo lắng về giới hạn về kích thước của các dòng.

Đọc chuỗi tùy ý Thường thì

việc đọc fgets( ) theo dòng rất hữu ích. Nhưng cũng thường xuyên như vậy, nó gây khó chịu. Đôi khi, các nhà phát triển muốn sử dụng một dấu phân cách khác ngoài ký tự xuống dòng. Những lần khác, các nhà phát triển không muốn sử dụng dấu phân cách nào cả—và hiếm khi các nhà phát triển muốn lưu trữ dấu phân cách trong bộ đệm! Nhìn lại, quyết định lưu trữ ký tự xuống dòng trong bộ đệm trả về hiếm khi có vẻ đúng.

Không khó để viết một thay thế fgets() sử dụng fgetc( ). Ví dụ, đoạn mã này đọc n – 1 byte từ luồng vào str, sau đó thêm một ký tự \0 :

```
char *s;
int c;

s =
chuỗi; trong khi (--n > 0 && (c = fgetc (luồng)) !=
         EOF) *s++
= c; *s = '\0';
```

Đoạn mã này có thể được mở rộng để dừng đọc tại dấu phân cách, được chỉ định bởi d (không thể là ký tự null trong ví dụ này):

```
ký tự
*s; int c = 0;

s =
chuỗi; trong khi (--n > 0 && (c = fgetc (luồng)) != EOF && (*s++ = c) != d)
          ;

nếu (c == d)
          *--s = '\0';
khác
          *s = '\0';
```

Đặt d thành \n sẽ cung cấp hành vi tương tự như fgets( ) , trừ việc lưu trữ ký tự xuống dòng trong bộ đệm.

Tùy thuộc vào việc triển khai fgets( ), biến thể này có thể chậm hơn vì nó phát ra các lệnh gọi hàm lặp lại tới fgetc( ). Tuy nhiên, đây không phải là vấn đề giống như ví dụ dd gốc của chúng tôi ! Mặc dù đoạn mã này phát sinh thêm chi phí gọi hàm, nhưng nó không phát sinh chi phí gọi hệ thống và hình phạt I/O không liên kết đè nặng lên dd với bs=1. Những vấn đề sau lớn hơn nhiều.

Đọc dữ liệu nhị phân Đối với

một số ứng dụng, việc đọc từng ký tự hoặc dòng là không đủ. Đôi khi, các nhà phát triển muốn đọc và ghi dữ liệu nhị phân phức tạp, chẳng hạn như cấu trúc C.
Đối với điều này, thư viện I/O chuẩn cung cấp fread( ):

```
#include <stdio.h>

size_t fread (void *buf, size_t kích thước, size_t số, FILE *stream);
```

Một lệnh gọi đến fread() sẽ đọc tối đa nr phần tử dữ liệu, mỗi phần tử có kích thước byte, từ luồng vào bộ đệm được trỏ tới bởi buf. Con trỏ tệp được tiến lên theo số byte đã đọc.

Số lượng phần tử đã đọc (không phải số byte đã đọc!) được trả về. Hàm này chỉ ra lỗi hoặc EOF thông qua giá trị trả về nhỏ hơn nr. Thật không may, không thể biết được điều kiện nào trong hai điều kiện đã xảy ra nếu không sử dụng ferror() và feof() (xem phần sau “Lỗi và Kết thúc tệp”).

Do sự khác biệt về kích thước biến, căn chỉnh, đệm và thứ tự byte, dữ liệu nhị phân được ghi bằng một ứng dụng có thể không được đọc bởi một ứng dụng khác hoặc thậm chí bởi cùng một ứng dụng trên một máy khác.

Ví dụ đơn giản nhất của fread() là đọc một phần tử byte tuyến tính từ một luồng nhất định:

```
char buf[64];
size_t nr;

nr = fread (buf, sizeof(buf), 1, stream); nếu (nr
== 0) /* lỗi */
```

Chúng ta sẽ xem xét các ví dụ phức tạp hơn khi nghiên cứu phần tương ứng của lệnh ghi fread( ), fwrite( ).

Viết vào một Luồng

Giống như đọc, thư viện C chuẩn định nghĩa nhiều hàm để ghi vào luồng mở. Phần này sẽ xem xét ba cách tiếp cận phổ biến nhất để ghi: ghi một ký tự đơn, ghi một chuỗi ký tự và ghi dữ liệu nhị phân.

Các cách tiếp cận viết đa dạng như vậy lý tưởng cho I/O đệm. Để ghi vào một luồng, luồng đó phải được mở như một luồng đầu ra với chế độ thích hợp; tức là bất kỳ chế độ hợp lệ nào ngoại trừ r.

Các vấn đề về căn chỉnh Tất

cả các kiến trúc máy đều có yêu cầu căn chỉnh dữ liệu . Các lập trình viên có xu hướng nghĩ về bộ nhớ chỉ đơn giản là một mảng byte. Tuy nhiên, bộ xử lý của chúng tôi không đọc và ghi từ bộ nhớ theo từng khối có kích thước byte. Thay vào đó, bộ xử lý truy cập bộ nhớ với một mức độ chi tiết cụ thể, chẳng hạn như 2, 4, 8 hoặc 16 byte. Vì không gian địa chỉ của mỗi quy trình bắt đầu từ địa chỉ 0, nên các quy trình phải khởi tạo truy cập từ một địa chỉ là bội số nguyên của mức độ chi tiết.

Do đó, các biến C phải được lưu trữ tại và truy cập từ các địa chỉ được căn chỉnh. Nhìn chung, các biến được căn chỉnh tự nhiên, tức là căn chỉnh tương ứng với kích thước của kiểu dữ liệu C. Ví dụ, một số nguyên 32 bit được căn chỉnh trên một ranh giới 4 byte. Nói cách khác, một int sẽ được lưu trữ tại một địa chỉ bộ nhớ chia hết cho bốn.

Truy cập dữ liệu không thẳng hàng có nhiều hình phạt khác nhau, tùy thuộc vào kiến trúc máy. Một số bộ xử lý có thể truy cập dữ liệu không thẳng hàng, nhưng với hình phạt hiệu suất lớn. Các bộ xử lý khác không thể truy cập dữ liệu không thẳng hàng chút nào và việc cố gắng làm như vậy sẽ gây ra ngoại lệ phần cứng. Tệ hơn nữa, một số bộ xử lý âm thầm loại bỏ các bit bậc thấp để buộc địa chỉ phải thẳng hàng, gần như chắc chắn dẫn đến hành vi không mong muốn.

Thông thường, trình biên dịch tự động căn chỉnh tất cả dữ liệu và căn chỉnh không phải là vấn đề dễ thấy đối với lập trình viên. Việc xử lý các cấu trúc, thực hiện quản lý bộ nhớ bằng tay, lưu dữ liệu nhị phân vào đĩa và giao tiếp qua mạng có thể đưa các vấn đề căn chỉnh lên hàng đầu. Do đó, các lập trình viên hệ thống cần phải thành thạo các vấn đề này!

Chương 8 đề cập sâu hơn đến sự liên kết.

Viết một ký tự đơn

Phần tương ứng của fgetc( ) là fputc( ):

```
#include <stdio.h>

int fputc (int c, TỆP *luồng);
```

Hàm fputc() ghi byte được chỉ định bởi c (ép thành unsigned char) vào luồng được trỏ tới bởi luồng. Sau khi hoàn thành thành công, hàm trả về c.
Nếu không, nó sẽ trả về EOF và errno được thiết lập phù hợp.

Cách sử dụng rất đơn giản:

```
nếu (fputc('p', stream) == EOF) /* lỗi */
```

Ví dụ này ghi ký tự p vào luồng, luồng này phải được mở để ghi.

Viết một chuỗi ký tự

Hàm fputs() được sử dụng để ghi toàn bộ chuỗi vào một luồng nhất định:

```
#include <stdio.h>

int fputs (const char *str, FILE *stream);
```

Một lệnh gọi đến fputs() ghi tất cả chuỗi được phân cách bằng null được trỏ tới bởi str vào luồng được trỏ tới bởi luồng. Khi thành công, fputs() trả về một số không âm. Khi thất bại, nó trả về EOF.

Ví dụ sau đây mở tệp để ghi ở chế độ thêm, ghi chuỗi đã cho vào luồng liên kết, rồi đóng luồng:

```
TẬP TIN *stream;

stream = fopen ("journal.txt", "a"); nếu (!
stream) /* lỗi
          */

if (fputs ("Con tàu được làm bằng gỗ.\n", stream) == EOF) /* lỗi */

nếu (fclose (stream) == EOF) /*
          lỗi */
```

Viết dữ liệu nhị phân Các ký

tự và dòng riêng lẻ sẽ không hiệu quả khi các chương trình cần ghi dữ liệu phức tạp. Để lưu trữ trực tiếp dữ liệu nhị phân như biến C, I/O chuẩn cung cấp fwrite( ):

```
#include <stdio.h>

size_t fwrite (void *buf,
                  size_t kích
                  thước, size_t số,
                  TỆP *stream);
```

Một lệnh gọi đến fwrite( ) sẽ ghi vào luồng lên đến nr phần tử, mỗi phần tử có kích thước byte theo chiều dài, từ dữ liệu được trỏ đến bởi buf. Con trỏ tệp sẽ được tiến lên theo tổng số byte đã ghi.

Số lượng phần tử (không phải số byte!) được ghi thành công sẽ được trả về. Giá trị trả về nhỏ hơn nr biểu thị lỗi.

Chương trình mẫu sử dụng I/O đệm

Bây giờ chúng ta hãy xem một ví dụ—thực tế là một chương trình hoàn chỉnh—tích hợp nhiều giao diện mà chúng ta đã đề cập đến trong chương này. Chương trình này đầu tiên định nghĩa struct pirate, sau đó khai báo hai biến có kiểu đó. Chương trình khởi tạo một trong các biến và sau đó ghi nó ra đĩa thông qua một luồng đầu ra tới

file data. Thông qua một luồng khác, chương trình đọc dữ liệu trở lại từ data trực tiếp vào phiên bản khác của struct pirate. Cuối cùng, chương trình ghi nội dung của cấu trúc để chuẩn hóa ra:

```
#include <stdio.h>

int main (void) {

          TỆP *in, *out;
          struct pirate
                    { name[100]; /* tên thật */ char unsigned long booty;
                    unsigned int beard_len; /*           /* tính theo bảng Anh */
                    tính bằng inch */ } p, blackbeard = { "Edward Teach", 950,
          48 };

          out = fopen ("dữ liệu", "w");
          nếu (!out)
                    { perror ("fopen");
                    trả về 1;
          }

          nếu (!fwrite (&blackbeard, sizeof (struct pirate), 1, out)) {
                    perror ("fwrite");
                    trả về 1;
          }

          nếu (fclose (ra))
                    { perror ("fclose");
                    trả về 1;
          }

          trong = fopen ("dữ liệu",
          "r"); nếu (!
                    in) { perror
                    ("fopen"); trả về 1;
          }

          nếu (!fread (&p, sizeof (struct pirate), 1, in)) { perror
                    ("fread"); trả về 1;

          }

          nếu (fclose (in))
                    { perror ("fclose");
                    trả về 1;
          }

          printf ("tên=\"%s\" chiến lợi phẩm=%lu râu_len=%u\n",
                    p.tên, p.chiến lợi phẩm, p.râu_len);

          trả về 0;
}
```

Tất nhiên, đầu ra là các giá trị ban đầu:

```
tên="Edward Teach" booty=950 beard_len=48
```

Một lần nữa, điều quan trọng cần lưu ý là do sự khác biệt về kích thước biến, căn chỉnh, v.v., dữ liệu nhị phân được ghi bằng một ứng dụng có thể không thể đọc được bởi các ứng dụng khác. Nghĩa là, một ứng dụng khác—hoặc thậm chí cùng một ứng dụng trên một máy khác—có thể không thể đọc lại chính xác dữ liệu được ghi bằng fwrite( ). Trong ví dụ của chúng tôi, hãy xem xét các hậu quả nếu kích thước của unsigned long thay đổi hoặc nếu lượng đệm thay đổi. Những điều này được đảm bảo là chỉ không đổi trên một loại máy cụ thể với ABI cụ thể.

Tìm kiếm một dòng suối

Thông thường, việc thao tác vị trí luồng hiện tại là hữu ích. Có lẽ ứng dụng đang đọc một tệp phức tạp dựa trên bản ghi và cần phải nhảy xung quanh. Hoặc, có lẽ luồng cần được đặt lại về vị trí tệp bằng không. Dù trường hợp nào đi nữa, I/O chuẩn cung cấp một họ giao diện có chức năng tương đương với lệnh gọi hệ thống lseek() (được thảo luận trong Chương 2). Hàm fseek() , là giao diện tìm kiếm I/O chuẩn phổ biến nhất, thao tác vị trí tệp của luồng theo offset và wherece:

```
#include <stdio.h>

int fseek (FILE *stream, độ lệch dài, int wherece);
```

Nếu whence được đặt thành SEEK_SET, vị trí tệp được đặt thành offset. Nếu whence được đặt thành SEEK_CUR, vị trí tệp được đặt thành vị trí hiện tại cộng với offset. Nếu whence được đặt thành SEEK_END, vị trí tệp được đặt thành cuối tệp cộng với offset.

Sau khi hoàn thành thành công, fseek() trả về 0, xóa chỉ báo EOF và hoàn tác các hiệu ứng (nếu có) của ungetc( ). Khi có lỗi, nó trả về -1 và errno được đặt phù hợp. Các lỗi phổ biến nhất là luồng không hợp lệ (EBADF) và đối số wherece không hợp lệ (EINVAL).

Ngoài ra, I/O chuẩn cung cấp fsetpos( ):

```
#include <stdio.h>

int fsetpos (FILE *stream, fpos_t *pos);
```

Hàm này đặt vị trí luồng của luồng thành pos. Nó hoạt động giống như fseek() với đối số whence là SEEK_SET. Khi thành công, nó trả về 0. Nếu không, nó trả về -1 và errno được đặt thành phù hợp. Hàm này (cùng với hàm tương ứng fgetpos( ) mà chúng ta sẽ đề cập sau) chỉ được cung cấp cho các nền tảng khác (không phải Unix) có các kiểu phức tạp biểu diễn vị trí luồng. Trên các nền tảng đó, hàm này là cách duy nhất để đặt vị trí luồng thành một giá trị tùy ý, vì kiểu C long có lẽ là không đủ. Các ứng dụng dành riêng cho Linux không cần sử dụng giao diện này, mặc dù chúng có thể sử dụng, nếu chúng muốn có thể di chuyển sang tất cả các nền tảng có thể.

I/O chuẩn cũng cung cấp rewind( ) như một phím tắt:

```
#include <stdio.h>

void tua lại (FILE *stream);
```

Lời kêu gọi này:

```
tua lại (phát trực tiếp);
```

đặt lại vị trí trở lại điểm bắt đầu của luồng. Nó tương đương với:

```
fseek(luồng, 0, SEEK_SET);
```

ngoại trừ việc nó cũng xóa chỉ báo lỗi.

Lưu ý rằng rewind() không có giá trị trả về và do đó không thể truyền đạt trực tiếp các điều kiện lỗi. Người gọi muốn xác định sự tồn tại của lỗi nên xóa errno trước khi gọi và kiểm tra xem biến có khác không sau đó không. Ví dụ:

```
errno = 0;
tua lại (phát trực
tuyến); nếu (errno)
          /* lỗi */
```

Lấy vị trí luồng hiện tại Không giống như lseek( ),

fseek() không trả về vị trí đã cập nhật. Một giao diện riêng được cung cấp cho mục đích này. Hàm ftell() trả về vị trí luồng hiện tại của luồng:

```
#include <stdio.h>

dài ftell (FILE *stream);
```

Khi có lỗi, nó trả về -1 và errno được thiết lập phù hợp.

Ngoài ra, I/O chuẩn cung cấp fgetpos( ):

```
#include <stdioh.h>

int fgetpos (TỆP *stream, fpos_t *pos);
```

Khi thành công, fgetpos() trả về 0 và đặt vị trí luồng hiện tại của luồng vào pos. Khi thất bại, nó trả về -1 và đặt errno một cách thích hợp. Giống như fsetpos( ), fgetpos() chỉ được cung cấp cho các nền tảng không phải Linux có các loại vị trí tệp phức tạp.

Xả một luồng

Thư viện I/O chuẩn cung cấp giao diện để ghi bộ đệm người dùng vào nhân, đảm bảo rằng tất cả dữ liệu được ghi vào luồng được xóa qua write( ). Hàm fflush() cung cấp chức năng này:

```
#include <stdio.h>

int fflush (TỆP *stream);
```

Khi gọi, bất kỳ dữ liệu chưa ghi nào trong luồng được luồng trỏ tới sẽ được xả vào kernel. Nếu luồng là NULL, tất cả các luồng đầu vào mở trong quy trình sẽ được xả. Khi thành công, fflush() trả về 0. Khi thất bại, nó trả về EOF và errno được đặt phù hợp.

Để hiểu tác dụng của fflush( ), bạn phải hiểu sự khác biệt giữa bộ đệm được duy trì bởi thư viện C và bộ đệm riêng của nhân. Tất cả các lệnh gọi được mô tả trong chương này đều hoạt động với bộ đệm được duy trì bởi thư viện C, nằm trong không gian người dùng, không phải không gian nhân. Đó là nơi cải thiện hiệu suất phát huy tác dụng—bạn đang ở trong không gian người dùng và do đó chạy mã người dùng, không phải phát lệnh gọi hệ thống. Lệnh gọi hệ thống chỉ được phát khi đĩa hoặc một phương tiện nào đó khác phải được truy cập.

fflush() chỉ ghi dữ liệu được đệm bởi người dùng ra bộ đệm hạt nhân. Hiệu ứng này giống như khi không sử dụng bộ đệm người dùng và sử dụng write() trực tiếp. Nó không đảm bảo rằng dữ liệu được cam kết vật lý với bất kỳ phương tiện nào—đối với nhu cầu đó, hãy sử dụng một cái gì đó như fsync() (xem “I/O đồng bộ” trong Chương 2). Rất có thể, bạn sẽ muốn gọi fflush( ) , theo sau ngay lập tức là fsync()): nghĩa là, trước tiên hãy đảm bảo rằng bộ đệm người dùng được ghi ra hạt nhân, sau đó đảm bảo rằng bộ đệm của hạt nhân được ghi ra đĩa.

Lỗi và Kết thúc Tệp

Một số giao diện I/O chuẩn, chẳng hạn như fread( ), truyền thông tin lỗi trở lại cho người gọi kém, vì chúng không cung cấp cơ chế để phân biệt giữa lỗi và EOF. Với những lệnh gọi này và trong những trường hợp khác, có thể hữu ích khi kiểm tra trạng thái của một luồng nhất định để xác định xem luồng đó có gặp lỗi hay đã đến cuối tệp hay không. I/O chuẩn cung cấp hai giao diện cho mục đích này. Hàm ferror() kiểm tra xem chỉ báo lỗi có được đặt trên luồng hay không:

```
bao gồm <stdio.h>

int ferror (TỆP *stream);
```

Chỉ báo lỗi được thiết lập bởi các giao diện I/O chuẩn khác để phản hồi lại điều kiện lỗi. Hàm trả về giá trị khác không nếu chỉ báo được thiết lập và trả về 0 nếu không.

Hàm feof() kiểm tra xem chỉ báo EOF có được đặt trên luồng hay không :

```
bao gồm <stdio.h>

int feof (TỆP *stream);
```

Chỉ báo EOF được thiết lập bởi các giao diện I/O chuẩn khác khi đến cuối tệp. Hàm này trả về giá trị khác không nếu chỉ báo được thiết lập và trả về 0 nếu không.

Hàm clearr() xóa lỗi và các chỉ báo EOF cho luồng:

```
#include <stdio.h>

void clearr (TỆP *stream);
```

Nó không có giá trị trả về và không thể lỗi (không có cách nào để biết liệu luồng không hợp lệ có được cung cấp hay không). Bạn chỉ nên gọi đến clearr() sau khi kiểm tra các chỉ báo lỗi và EOF , vì chúng sẽ bị loại bỏ không thể phục hồi sau đó. Ví dụ:

```
/* 'f' là một luồng hợp lệ */

if (ferror (f))
         printf ("Lỗi ở f!\n");

nếu (feof (f))
         printf ("EOF trên f!\n");

rõ ràng hơn (f);
```

Nhận được mô tả tập tin liên quan

Đôi khi, việc lấy mô tả tệp sao lưu một luồng nhất định sẽ có lợi.
Ví dụ, có thể hữu ích khi thực hiện lệnh gọi hệ thống trên luồng, thông qua mô tả tệp của nó, khi không có hàm I/O chuẩn liên quan. Để lấy mô tả tệp hỗ trợ luồng, hãy sử dụng fileno( ):

```
#include <stdio.h>

int fileno (TỆP *stream);
```

Khi thành công, fileno( ) trả về mô tả tệp được liên kết với luồng. Khi thất bại, nó trả về -1. Điều này chỉ có thể xảy ra khi luồng đã cho không hợp lệ, trong trường hợp đó, hàm đặt errno thành EBADF.

Việc trộn lẫn các lệnh gọi I/O chuẩn với các lệnh gọi hệ thống thường không được khuyến khích. Các lập trình viên phải thận trọng khi sử dụng fileno() để đảm bảo hành vi phù hợp. Cụ thể, có lẽ nên xả luồng trước khi thao tác với mô tả tệp sao lưu. Bạn hầu như không bao giờ nên trộn lẫn các hoạt động I/O thực tế.

Kiểm soát bộ đệm

I/O chuẩn triển khai ba loại bộ đệm người dùng và cung cấp cho các nhà phát triển một giao diện để kiểm soát loại và kích thước của bộ đệm. Các loại bộ đệm người dùng khác nhau phục vụ các mục đích khác nhau và lý tưởng cho các tình huống khác nhau. Sau đây là các tùy chọn:

Không đệm
: Không có đệm người dùng nào được thực hiện. Dữ liệu được gửi trực tiếp đến hạt nhân. Vì đây là sự đối lập của đệm người dùng, tùy chọn này không được sử dụng phổ biến. Lỗi chuẩn, theo mặc định, là không đệm.

Bộ đệm dòng

được thực hiện trên cơ sở từng dòng. Với mỗi ký tự xuống dòng, bộ đệm được gửi đến kernel. Bộ đệm dòng có ý nghĩa đối với các luồng được xuất ra màn hình. Do đó, đây là bộ đệm mặc định được sử dụng cho các thiết bị đầu cuối (out chuẩn được đệm dòng theo mặc định).

Buffered khối

đệm được thực hiện trên cơ sở từng khối. Đây là loại đệm được thảo luận ở phần đầu của chương này và lý tưởng cho các tệp. Theo mặc định, tất cả các luồng liên kết với tệp đều được đệm khối. I/O chuẩn sử dụng thuật ngữ đệm đầy đủ cho đệm khối.

Hầu hết thời gian, loại đệm mặc định là đúng và tối ưu. Tuy nhiên, I/O chuẩn cung cấp giao diện để kiểm soát loại đệm được sử dụng:

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t kích thước);
```

Hàm setvbuf() thiết lập kiểu đệm của luồng thành chế độ, phải là một trong những chế độ sau:

_IONBF

Không đệm

_IOLBF

đệm dòng

_IOFBF

Khối đệm

Ngoại trừ _IONBF, trong trường hợp đó buf và size bị bỏ qua, buf có thể trỏ đến một bộ đệm có kích thước byte mà I/O chuẩn sẽ sử dụng làm bộ đệm cho luồng đã cho. Nếu buf là NULL, một bộ đệm được glibc tự động phân bổ.

Hàm setvbuf() phải được gọi sau khi mở luồng, nhưng trước khi bất kỳ hoạt động nào khác được thực hiện trên đó. Nó trả về 0 nếu thành công và giá trị khác không nếu không.

Bộ đệm được cung cấp, nếu có, phải tồn tại khi luồng được đóng. Một lỗi thường gặp là khai báo bộ đệm là một biến tự động trong phạm vi kết thúc trước khi luồng được đóng. Đặc biệt, hãy cẩn thận không cung cấp bộ đệm cục bộ cho main( ), rồi không đóng luồng một cách rõ ràng. Ví dụ, sau đây là lỗi:

```
#include <stdio.h>

int main (void)
{
        char buf[BUFSIZ];

        /* đặt stdin thành bộ đệm khối với bộ đệm BUFSIZ */ setvbuf (stdout,
        buf, _IOFBF, BUFSIZ);
```

```
            printf ("Arrr!\n");

            trả về 0;
    }
```

Lỗi này có thể được khắc phục bằng cách đóng luồng một cách rõ ràng trước khi thoát khỏi phạm vi hoặc bằng cách biến buf thành một biến toàn cục.

Nhìn chung, các nhà phát triển không cần phải bận tâm đến việc đệm trên luồng. Ngoại trừ lỗi chuẩn, các thiết bị đầu cuối được đệm theo dòng và điều đó có lý. Các tệp được đệm theo khối và điều đó cũng có lý. Kích thước bộ đệm mặc định cho việc đệm khối là BUFSIZ, được định nghĩa trong <stdio.h> và thường là lựa chọn tối ưu (gấp nhiều lần kích thước khối thông thường).

An toàn luồng

Luồng là nhiều loại thực thi trong một tiến trình duy nhất. Một cách để khái niệm hóa chúng là nhiều tiến trình chia sẻ một không gian địa chỉ. Luồng có thể chạy bất kỳ lúc nào và có thể ghi đè lên dữ liệu được chia sẻ trừ khi cẩn thận đồng bộ hóa quyền truy cập vào dữ liệu hoặc biến nó thành cục bộ luồng. Các hệ điều hành hỗ trợ luồng cung cấp cơ chế khóa (cấu trúc lập trình đảm bảo loại trừ lẫn nhau) để đảm bảo rằng các luồng không giẫm đạp lên nhau. I/O chuẩn sử dụng các cơ chế này. Tuy nhiên, chúng không phải lúc nào cũng đầy đủ. Ví dụ, đôi khi bạn muốn khóa một nhóm lệnh gọi, mở rộng vùng quan trọng (khối mã chạy mà không bị luồng khác can thiệp) từ một hoạt động I/O thành nhiều hoạt động. Trong các tình huống khác, bạn có thể muốn loại bỏ hoàn toàn khóa để cải thiện hiệu quả.* Trong phần này, chúng ta sẽ thảo luận về cách thực hiện cả hai.

Các hàm I/O chuẩn vốn an toàn cho luồng. Về mặt nội bộ, chúng liên kết một khóa, một số lượng khóa và một luồng sở hữu với mỗi luồng mở. Bất kỳ luồng nào được đưa ra cũng phải có được khóa và trở thành luồng sở hữu trước khi đưa ra bất kỳ yêu cầu I/O nào. Hai hoặc nhiều luồng hoạt động trên cùng một luồng không thể xen kẽ các hoạt động I/O chuẩn và do đó, trong bối cảnh của các lệnh gọi hàm đơn, các hoạt động I/O chuẩn là nguyên tử.

Tất nhiên, trong thực tế, nhiều ứng dụng yêu cầu tính nguyên tử cao hơn ở cấp độ các lệnh gọi hàm riêng lẻ. Ví dụ, nếu nhiều luồng phát hành các yêu cầu ghi, mặc dù các lệnh ghi riêng lẻ sẽ không xen kẽ và dẫn đến đầu ra bị nhiễu, ứng dụng có thể muốn hoàn thành tất cả các yêu cầu ghi mà không bị gián đoạn. Để cho phép điều này, I/O chuẩn cung cấp một họ các hàm để thao tác riêng lẻ khóa liên quan đến một luồng.

* Thông thường, việc loại bỏ khóa sẽ dẫn đến một loạt các vấn đề. Nhưng một số chương trình có thể triển khai rõ ràng việc sử dụng luồng của chúng để phân bổ tất cả I/O cho một luồng duy nhất. Trong trường hợp đó, không cần phải có quá nhiều chi phí cho khóa.

Khóa tập tin thủ công

Hàm flockfile() đợi cho đến khi luồng không còn bị khóa nữa, sau đó lấy khóa, tăng số lượng khóa, trở thành luồng sở hữu luồng và
trả về:

```
#include <stdio.h>

void flockfile (FILE *stream);
```

Hàm funlockfile() giảm số lượng khóa liên quan đến luồng:

```
#include <stdio.h>

void funlockfile (FILE *stream);
```

Nếu số lượng khóa đạt đến 0, luồng hiện tại sẽ từ bỏ quyền sở hữu luồng. Một luồng khác hiện có thể lấy được khóa.

Các lệnh gọi này có thể lồng nhau. Nghĩa là, một luồng đơn có thể phát ra nhiều lệnh gọi flockfile() và luồng sẽ không mở khóa cho đến khi tiến trình phát ra số lượng lệnh gọi funlockfile() tương ứng .

Hàm ftrylockfile() là phiên bản không chặn của flockfile( ):

```
#include <stdio.h>

int ftrylockfile (TỆP *stream);
```

Nếu luồng hiện đang bị khóa, ftrylockfile() không làm gì cả và ngay lập tức trả về giá trị khác không. Nếu luồng hiện không bị khóa, nó sẽ lấy khóa, tăng số lượng khóa, trở thành luồng sở hữu luồng và trả về 0.

Hãy cùng xem xét một ví dụ:

```
flockfile (luồng);

fputs ("Danh sách kho báu:\n", stream);
fputs (" (1) 500 đồng tiền vàng\n", stream);
fputs ("      (2) Bộ đồ ăn được trang trí tuyệt đẹp\n", stream);

funlockfile (luồng);
```

Mặc dù các hoạt động fputs() riêng lẻ không bao giờ có thể chạy đua—ví dụ, chúng ta sẽ không bao giờ kết thúc với bất kỳ thứ gì xen kẽ với “Danh sách kho báu”—một hoạt động I/O chuẩn khác từ luồng khác đến cùng luồng này có thể xen kẽ giữa hai lệnh gọi fputs() . Lý tưởng nhất là một ứng dụng được thiết kế sao cho nhiều luồng không gửi I/O đến cùng một luồng. Tuy nhiên, nếu ứng dụng của bạn cần làm như vậy và bạn cần một vùng nguyên tử lớn hơn một hàm duy nhất, flockfile() và các hàm tương tự có thể cứu vãn tình hình.

Hoạt động luồng không khóa Có một

lý do thứ hai để thực hiện khóa thủ công trên luồng. Với khả năng kiểm soát khóa chi tiết hơn và chính xác hơn mà chỉ có lập trình viên ứng dụng mới có thể cung cấp, có thể giảm thiểu chi phí khóa và cải thiện hiệu suất. Để đạt được mục đích này, Linux cung cấp một họ các hàm, họ hàng với các giao diện I/O chuẩn thông thường, không thực hiện bất kỳ khóa nào. Trên thực tế, chúng là các đối tác không khóa của I/O chuẩn:

```
#xác định _GNU_SOURCE

#include <stdio.h>

int fgetc_unlocked (FILE *stream);
char *fgets_unlocked (char *str, int kích thước, FILE *stream);
size_t fread_unlocked (void *buf, size_t kích thước, size_t số,
                       TỆP *luồng);
int fputc_unlocked (int c, TỆP *luồng); int
fputs_unlocked (const char *str, TỆP *luồng); kích thước_t
fwrite_unlocked (void *buf, kích thước_t kích thước, kích thước_t nr,
                         TỆP *stream);
int fflush_unlocked (TỆP *stream); int
feof_unlocked (TỆP *stream); int
ferror_unlocked (TỆP *stream); int
fileno_unlocked (TỆP *stream); void
clearr_unlocked (TỆP *stream);
```

Tất cả các hàm này đều hoạt động giống hệt như các hàm bị khóa của chúng, ngoại trừ việc chúng không kiểm tra hoặc lấy khóa liên quan đến luồng đã cho . Nếu cần khóa, trách nhiệm của lập trình viên là đảm bảo khóa được lấy và giải phóng thủ công.

Mặc dù POSIX định nghĩa một số biến thể mở khóa của các hàm I/O chuẩn, nhưng không có hàm nào ở trên được POSIX định nghĩa. Tất cả đều dành riêng cho Linux, mặc dù nhiều hệ thống Unix khác hỗ trợ một tập hợp con.

Phê bình về I/O chuẩn

Mặc dù I/O chuẩn được sử dụng rộng rãi, một số chuyên gia chỉ ra những sai sót trong đó. Một số hàm, chẳng hạn như fgets( ), đôi khi không đủ. Các hàm khác, chẳng hạn như gets( ), tệ đến mức chúng gần như bị loại khỏi các tiêu chuẩn.

Khiếu nại lớn nhất với I/O chuẩn là tác động về hiệu suất từ bản sao kép. Khi đọc dữ liệu, I/O chuẩn sẽ phát lệnh gọi hệ thống read() đến kernel, sao chép dữ liệu từ kernel vào bộ đệm I/O chuẩn. Khi một ứng dụng sau đó phát lệnh yêu cầu đọc qua I/O chuẩn—ví dụ, sử dụng fgetc( )—dữ liệu được sao chép lại, lần này từ bộ đệm I/O chuẩn đến bộ đệm được cung cấp. Yêu cầu ghi hoạt động theo cách ngược lại: dữ liệu được sao chép một lần từ bộ đệm được cung cấp đến bộ đệm I/O chuẩn, và sau đó từ bộ đệm I/O chuẩn đến kernel qua write( ).

Một triển khai thay thế có thể tránh được việc sao chép kép bằng cách yêu cầu mỗi yêu cầu đọc trả về một con trỏ vào bộ đệm I/O chuẩn. Sau đó, dữ liệu có thể được đọc trực tiếp, bên trong bộ đệm I/O chuẩn, mà không cần phải sao chép thêm.
Trong trường hợp ứng dụng muốn dữ liệu trong bộ đệm cục bộ của riêng nó—có thể là để ghi vào đó—nó luôn có thể thực hiện sao chép thủ công. Việc triển khai này sẽ cung cấp một giao diện "miễn phí", cho phép các ứng dụng báo hiệu khi chúng hoàn tất với một phần nhất định của bộ đệm đọc.

Việc ghi sẽ phức tạp hơn một chút, nhưng vẫn có thể tránh được việc sao chép hai lần.
Khi đưa ra yêu cầu ghi, quá trình triển khai sẽ ghi lại con trỏ. Cuối cùng, khi sẵn sàng xả dữ liệu vào nhân, quá trình triển khai có thể duyệt danh sách các con trỏ đã lưu trữ, ghi dữ liệu ra. Điều này có thể được thực hiện bằng cách sử dụng scatter-gather I/O, thông qua writev( ) và do đó chỉ cần một lệnh gọi hệ thống duy nhất. (Chúng ta sẽ thảo luận về scatter-gather I/O trong chương tiếp theo.)

Có những thư viện đệm người dùng cực kỳ tối ưu, giải quyết vấn đề sao chép kép với các triển khai tương tự như những gì chúng ta vừa thảo luận. Ngoài ra, một số nhà phát triển chọn triển khai các giải pháp đệm người dùng của riêng họ. Nhưng bất chấp những giải pháp thay thế này, I/O chuẩn vẫn phổ biến.

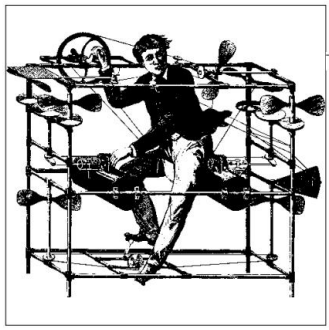
Phần kết luận

I/O chuẩn là thư viện đệm người dùng được cung cấp như một phần của thư viện C chuẩn.
Modulo một vài lỗi, đây là một giải pháp mạnh mẽ và rất phổ biến. Trên thực tế, nhiều lập trình viên C không biết gì ngoài I/O chuẩn. Chắc chắn, đối với I/O đầu cuối, nơi bộ đệm dựa trên dòng là lý tưởng, I/O chuẩn là trò chơi duy nhất trong thị trấn. Ai đã từng trực tiếp sử dụng write() để in ra chuẩn?

I/O chuẩn—và bộ đệm người dùng nói chung—có ý nghĩa khi bất kỳ điều nào sau đây là đúng:

- Bạn có thể hình dung ra nhiều lệnh gọi hệ thống và bạn muốn giảm thiểu chi phí chung bằng cách kết hợp nhiều cuộc gọi thành ít cuộc gọi.
- Hiệu suất là yếu tố quan trọng và bạn muốn đảm bảo rằng tất cả I/O đều diễn ra trong các khối có kích thước khối trên các ranh giới được căn chỉnh theo khối. • Các mẫu truy cập của bạn dựa trên ký tự hoặc dòng và bạn muốn các giao diện làm cho việc truy cập đó trở nên dễ dàng mà không cần phải thực hiện các lệnh gọi hệ thống không cần thiết. • Bạn thích giao diện cấp cao hơn so với các lệnh gọi hệ thống Linux cấp thấp.

Tuy nhiên, tính linh hoạt nhất tồn tại khi bạn làm việc trực tiếp với các lệnh gọi hệ thống Linux. Trong chương tiếp theo, chúng ta sẽ xem xét các dạng I/O nâng cao và các lệnh gọi hệ thống liên quan.



CHƯƠNG 4
Tệp I/O nâng cao

Trong Chương 2, chúng ta đã xem xét các lệnh gọi hệ thống I/O cơ bản trong Linux. Các lệnh gọi này không chỉ tạo thành cơ sở của I/O tệp mà còn là nền tảng của hầu như mọi giao tiếp trên Linux. Trong Chương 3, chúng ta đã xem xét cách đệm không gian người dùng thường cần thiết trên các lệnh gọi hệ thống I/O cơ bản và chúng ta đã nghiên cứu một giải pháp đệm không gian người dùng cụ thể, thư viện I/O chuẩn của C. Trong chương này, chúng ta sẽ xem xét các lệnh gọi hệ thống I/O nâng cao hơn mà Linux cung cấp: I/O phân tán/tập hợp Cho phép một lệnh gọi duy nhất để đọc hoặc ghi dữ liệu vào và

ra khỏi nhiều bộ đệm

cùng một lúc; hữu ích để nhóm các trường của các cấu trúc dữ liệu khác nhau lại với nhau để tạo thành một giao dịch I/O.

Epoll

Cải thiện các lệnh gọi hệ thống poll() và select() được mô tả trong Chương 2; hữu ích khi phải thăm dò hàng trăm mô tả tệp trong một chương trình duy nhất.

I/O ánh xạ bộ nhớ Ánh

xạ một tệp vào bộ nhớ, cho phép I/O tệp xảy ra thông qua thao tác bộ nhớ đơn giản; hữu ích cho một số kiểu I/O nhất định.

Tư vấn tệp

Cho phép một quy trình cung cấp gợi ý cho hạt nhân về các tình huống sử dụng của nó; có thể dẫn đến cải thiện hiệu suất I/O.

I/O không đồng bộ

Cho phép một tiến trình đưa ra các yêu cầu I/O mà không cần chờ chúng hoàn tất; hữu ích để xử lý khối lượng công việc I/O lớn mà không cần sử dụng luồng.

Chương này sẽ kết thúc bằng một cuộc thảo luận về các cân nhắc về hiệu suất và các hệ thống I/O của hạt nhân.

Phân tán/Tập hợp I/O

I/O phân tán/tập hợp là một phương pháp nhập và xuất trong đó một lệnh gọi hệ thống duy nhất ghi vào một vectơ bộ đệm từ một luồng dữ liệu duy nhất hoặc, ngược lại, đọc vào một vectơ bộ đệm từ một luồng dữ liệu duy nhất. Kiểu I/O này được đặt tên như vậy vì dữ liệu được phân tán vào hoặc tập hợp từ vectơ bộ đệm đã cho. Một tên gọi khác cho phương pháp nhập và xuất này là I/O vectơ. Để so sánh, các lệnh gọi hệ thống đọc và ghi chuẩn mà chúng ta đã đề cập trong Chương 2 cung cấp I/O tuyến tính.

I/O phân tán/tập hợp có một số ưu điểm so với phương pháp I/O tuyến tính:

Xử lý tự nhiên hơn Nếu
: dữ liệu của bạn được phân đoạn tự nhiên—ví dụ, các trường của tệp tiêu đề được xác định trước—thì I/O vectơ cho phép thao tác trực quan.

Hiệu quả
: Một hoạt động I/O vectơ đơn có thể thay thế nhiều hoạt động I/O tuyến tính.

Hiệu suất
: Ngoài việc giảm số lượng lệnh gọi hệ thống được phát hành, việc triển khai I/O theo vectơ có thể cung cấp hiệu suất được cải thiện so với việc triển khai I/O tuyến tính thông qua các tối ưu hóa nội bộ.

Tính nguyên
: từ Không giống như nhiều hoạt động I/O tuyến tính, một tiến trình có thể thực hiện một hoạt động I/O vectơ duy nhất mà không có nguy cơ xen kẽ hoạt động từ một tiến trình khác.

Cả phương pháp I/O tự nhiên hơn và tính nguyên tử đều có thể đạt được mà không cần cơ chế I/O phân tán/thu thập. Một quy trình có thể nối các vectơ rời rạc thành một bộ đệm duy nhất trước khi ghi và phân tách bộ đệm được trả về thành nhiều vectơ sau khi đọc—tức là, một ứng dụng không gian người dùng có thể thực hiện phân tán và thu thập theo cách thủ công. Tuy nhiên, giải pháp như vậy không hiệu quả và cũng không thú vị để triển khai.

readv() và writev()

POSIX 1003.1-2001 định nghĩa và Linux triển khai một cặp lệnh gọi hệ thống thực hiện phân tán/thu thập I/O. Việc triển khai Linux đáp ứng tất cả các mục tiêu được liệt kê trong phần trước.

Hàm readv() đọc số phân đoạn từ mô tả tệp fd vào bộ đệm được mô tả bởi iov:

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

ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

Hàm writev() ghi tối đa count phân đoạn từ bộ đệm được mô tả bởi iov vào bộ mô tả tệp fd:

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

ssize_t writev (int fd,
                const struct iovec *iov, int
                count);
```

Các hàm readv() và writev() có chức năng tương tự như hàm read() và write( ), ngoại trừ việc có nhiều bộ đệm được đọc từ hoặc ghi vào.

Mỗi cấu trúc iovec mô tả một bộ đệm rời rạc độc lập, được gọi là một phân đoạn:

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

struct iovec
        { void *iov_base;      /* con trỏ đến đầu bộ đệm */ /* kích
        kích thước_t iov_len;  thước của bộ đệm tính bằng byte */
};
```

Một tập hợp các phân đoạn được gọi là một vectơ. Mỗi phân đoạn trong vectơ mô tả địa chỉ và độ dài của bộ đệm trong bộ nhớ mà dữ liệu sẽ được ghi hoặc đọc vào.
Hàm readv() điền đầy đủ từng bộ đệm iov_len byte trước khi tiến hành đến bộ đệm tiếp theo. Hàm writev() luôn ghi ra tất cả các byte iov_len đầy đủ trước khi tiến hành đến bộ đệm tiếp theo. Cả hai hàm luôn hoạt động trên các phân đoạn theo thứ tự, bắt đầu bằng iov[0], sau đó là iov[1], v.v., cho đến iov[count-1].

Giá trị trả về

Khi thành công, readv() và writev() trả về số byte đã đọc hoặc đã ghi tương ứng. Số này phải là tổng của tất cả các giá trị count iov_len . Khi có lỗi, các lệnh gọi hệ thống trả về -1 và đặt errno cho phù hợp. Các lệnh gọi hệ thống này có thể gặp bất kỳ lỗi nào của các lệnh gọi hệ thống read() và write() và khi nhận được các lỗi như vậy, sẽ đặt cùng một mã errno . Ngoài ra, các tiêu chuẩn còn định nghĩa hai tình huống lỗi khác.

Đầu tiên, vì kiểu trả về là ssize_t , nếu tổng của tất cả các giá trị count iov_len lớn hơn SSIZE_MAX, sẽ không có dữ liệu nào được chuyển, -1 sẽ được trả về và errno sẽ được đặt thành EINVAL.

Thứ hai, POSIX quy định rằng count phải lớn hơn 0 và nhỏ hơn hoặc bằng IOV_MAX, được định nghĩa trong <limits.h>. Trong Linux, IOV_MAX hiện tại là 1024. Nếu count bằng 0, hệ thống sẽ trả về 0.* Nếu count lớn hơn IOV_MAX, không có dữ liệu nào được truyền, các lệnh sẽ trả về -1 và errno được đặt thành EINVAL.

* Lưu ý rằng các hệ thống Unix khác có thể đặt errno thành EINVAL nếu count là 0. Điều này được cho phép rõ ràng theo các tiêu chuẩn, trong đó nêu rằng EINVAL có thể được đặt nếu giá trị đó là 0 hoặc hệ thống có thể xử lý trường hợp số không theo một cách khác (không có lỗi).

> Tối ưu hóa Count Trong một hoạt
>
> động I/O theo vectơ, hạt nhân Linux phải phân bổ các cấu trúc dữ liệu nội bộ để biểu diễn từng phân đoạn. Thông thường, việc phân bổ này sẽ diễn ra theo kiểu động, dựa trên kích thước của count. Tuy nhiên, như một sự tối ưu hóa, hạt nhân Linux tạo ra một mảng nhỏ các phân đoạn trên ngăn xếp mà nó sử dụng nếu count đủ nhỏ, phủ nhận nhu cầu phân bổ động các phân đoạn và do đó cung cấp một sự gia tăng nhỏ về hiệu suất. Ngưỡng này hiện là tám, vì vậy nếu count nhỏ hơn hoặc bằng 8, hoạt động I/O theo vectơ diễn ra theo cách rất hiệu quả về bộ nhớ ngoài ngăn xếp hạt nhân của quy trình.
>
> Nhiều khả năng là bạn sẽ không có lựa chọn về số lượng phân đoạn cần chuyển cùng lúc trong một hoạt động I/O vectơ nhất định. Tuy nhiên, nếu bạn linh hoạt và đang cân nhắc về một giá trị nhỏ, thì việc chọn giá trị bằng tám hoặc ít hơn chắc chắn sẽ cải thiện hiệu quả.

Ví dụ writev()

Hãy xem xét một ví dụ đơn giản viết ra một vectơ gồm ba đoạn, mỗi đoạn chứa một chuỗi có kích thước khác nhau. Chương trình độc lập này đủ hoàn chỉnh để chứng minh writev( ), nhưng cũng đủ đơn giản để dùng làm đoạn mã hữu ích:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/uio.h>

int chính ( )
{
          cấu trúc iovec iov[3];
          ssize_t nr;
          int fd, i;

          char *buf[] =
                    { "Thuật ngữ cướp biển bắt nguồn từ từ boucan.\n", "Boucan là một
                    khung gỗ dùng để nấu thịt.\n", "Buccaneer là tên gọi ở Tây Ấn dành
                    cho cướp biển.\n" };

          fd = mở ("buccaneer.txt", O_WRONLY | O_CREAT | O_TRUNC); nếu (fd ==
          -1) { perror
                    ("mở"); trả về 1;

          }

          /* điền vào ba cấu trúc iovec */ cho (i = 0; i
          < 3; i++) { iov[i].iov_base =
                    buf[i]; iov[i].iov_len = strlen
                    (buf[i]);
          }
```

```
            /* chỉ với một lần gọi, ghi tất cả ra */ nr = writev
            (fd, iov, 3); if (nr == -1)
            { perror
                    ("writev"); return 1;


            } printf ("đã ghi %d byte\n", nr);

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

            trả về 0;
    }
```

Chạy chương trình sẽ cho kết quả mong muốn:

```
$ ./writev
đã viết 148 byte
```

Tương tự như việc đọc tệp:

```
$ cat buccaneer.txt Thuật
ngữ buccaneer xuất phát từ từ boucan.
Boucan là một khung gỗ dùng để nấu thịt.
Buccaneer là tên gọi ở Tây Ấn dành cho cướp biển.
```

Ví dụ readv( )

Bây giờ, chúng ta hãy xem xét một chương trình ví dụ sử dụng lệnh gọi hệ thống readv( ) để đọc từ tệp văn bản được tạo trước đó bằng cách sử dụng I/O vectơ. Ví dụ độc lập này cũng đơn giản nhưng đầy đủ:

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

int chính ( )
{
          char foo[48], bar[51], baz[49];
          struct iovec iov[3];
          ssize_t nr;
          int fd, i;

          fd = mở ("buccaneer.txt", O_RDONLY); nếu (fd
          == -1) { perror
                    ("mở"); trả về 1;

          }

          /* thiết lập cấu trúc iovec của chúng ta */
```

```
        iov[0].iov_base = foo;
        iov[0].iov_len = sizeof(foo);
        iov[1].iov_base = bar;
        iov[1].iov_len = sizeof(bar);
        iov[2].iov_base = baz;
        iov[2].iov_len = sizeof(baz);

        /* đọc vào các cấu trúc bằng một lệnh gọi duy nhất */ nr =
        readv (fd, iov, 3); if (nr
        == -1) { perror
                 ("readv"); return 1;

        }

        đối với (i = 0; i < 3; i++)
                 printf ("%d: %s", i, (ký tự *) iov[i].iov_base);

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

        trả về 0;
}
```

Chạy chương trình này sau khi chạy chương trình trước đó sẽ cho kết quả sau:

```
$ ./readv 0:
Thuật ngữ buccaneer xuất phát từ từ boucan.
1: Boucan là một khung gỗ dùng để nấu thịt.
2: Buccaneer là tên gọi ở Tây Ấn dành cho cướp biển.
```

Thực hiện

Một triển khai ngây thơ của readv() và writev() có thể được thực hiện trong không gian người dùng như một vòng lặp đơn giản, tương tự như sau:

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

ssize_t ngây thơ_writev (int fd, const struct iovec *iov, int đếm) {

        kích thước_t ret =
        0; int i;

        đối với (i = 0; i < đếm; i++)
                 { ssize_t nr;

                 nr = ghi (fd, iov[i].iov_base, iov[i].iov_len); nếu (nr
                 == -1) { ret =
                          -1; ngắt;


                 } ret += số;
```

```
            }

            trả về ret;
    }
```

May mắn thay, đây không phải là triển khai Linux: Linux triển khai readv() và writev() dưới dạng các lệnh gọi hệ thống và thực hiện I/O phân tán/thu thập nội bộ. Trên thực tế, tất cả I/O bên trong hạt nhân Linux đều được vectơ hóa; read() và write() được triển khai dưới dạng I/O vectơ với vectơ chỉ có một phân đoạn.

Giao diện thăm dò sự kiện

Nhận ra những hạn chế của cả poll() và select( ), hạt nhân Linux 2.6* đã giới thiệu tiện ích event poll (epoll). Mặc dù phức tạp hơn hai giao diện trước đó, epoll giải quyết được vấn đề hiệu suất cơ bản mà cả hai đều gặp phải và bổ sung thêm một số tính năng mới.

Cả poll() và select() (được thảo luận trong Chương 2) đều yêu cầu danh sách đầy đủ các mô tả tệp để theo dõi trong mỗi lần gọi. Sau đó, hạt nhân phải duyệt danh sách của từng mô tả tệp cần theo dõi. Khi danh sách này lớn lên—nó có thể chứa hàng trăm hoặc thậm chí hàng nghìn mô tả tệp—việc duyệt danh sách trong mỗi lần gọi sẽ trở thành nút thắt về khả năng mở rộng.

Epoll tránh được vấn đề này bằng cách tách đăng ký giám sát khỏi giám sát thực tế. Một lệnh gọi hệ thống khởi tạo ngữ cảnh epoll, một lệnh khác thêm các mô tả tệp được giám sát vào hoặc xóa chúng khỏi ngữ cảnh và lệnh thứ ba thực hiện chờ sự kiện thực tế.

Tạo một phiên bản Epoll mới

Một bối cảnh epoll được tạo thông qua epoll_create( ):

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

int epoll_create (kích thước int)
```

Một lệnh gọi thành công đến epoll_create( ) sẽ khởi tạo một thể hiện epoll mới và trả về một mô tả tệp được liên kết với thể hiện đó. Mô tả tệp này không liên quan đến tệp thực; nó chỉ là một trình xử lý được sử dụng với các lệnh gọi tiếp theo sử dụng tiện ích epoll.
Tham số size là một gợi ý cho kernel về số lượng file descriptor sẽ được giám sát; nó không phải là số lượng tối đa. Truyền vào một xấp xỉ tốt sẽ mang lại hiệu suất tốt hơn, nhưng không yêu cầu số lượng chính xác. Khi có lỗi, lệnh gọi trả về -1 và đặt errno thành một trong các giá trị sau:

EINVAL

Tham số kích thước không phải là số dương.

* Epoll được giới thiệu trong phiên bản phát triển 2.5.44 và giao diện được hoàn thiện vào phiên bản 2.5.66.

ENFILE

Hệ thống đã đạt đến giới hạn về tổng số tệp đang mở.

ENOMEM

Không đủ bộ nhớ để hoàn tất thao tác.

Một cuộc gọi điển hình là:

```
int epfd;

epfd = epoll_create (100); /* dự định theo dõi ~100 fds */ nếu
(epfd < 0)
         perror ("epoll_create");
```

Bộ mô tả tệp được trả về từ epoll_create() phải bị hủy thông qua lệnh gọi close() sau khi quá trình thăm dò kết thúc.

Kiểm soát Epoll

Lệnh gọi hệ thống epoll_ctl( ) có thể được sử dụng để thêm mô tả tệp vào và xóa mô tả tệp khỏi ngữ cảnh epoll nhất định:

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

int epoll_ctl (int epfd,
                 int op,
                 int fd,
                 struct epoll_event *event);
```

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

```
struct epoll_event { _
         _u32 sự kiện; /* sự kiện */
         union
               { void *ptr;
                  int fd;
                  _ _u32 u32;
                  _ _u64 u64;
         } dữ liệu;
};
```

Một lệnh gọi thành công đến epoll_ctl() sẽ kiểm soát phiên bản epoll được liên kết với mô tả tệp epfd. Tham số op chỉ định thao tác sẽ được thực hiện đối với tệp được liên kết với fd. Tham số event mô tả thêm hành vi của thao tác.

Sau đây là các giá trị hợp lệ cho tham số op :

EPOLL_CTL_ADD

Thêm một màn hình trên tệp được liên kết với mô tả tệp fd vào phiên bản epoll được liên kết với epfd, theo các sự kiện được xác định trong event.

EPOLL_CTL_DEL

Xóa màn hình trên tệp được liên kết với mô tả tệp fd khỏi phiên bản epoll được liên kết với epfd.

EPOLL_CTL_MOD

Sửa đổi màn hình hiện có của fd với các sự kiện được cập nhật do sự kiện chỉ định.

Trường sự kiện trong cấu trúc epoll_event liệt kê các sự kiện cần theo dõi trên mô tả tệp đã cho. Nhiều sự kiện có thể được bitwise-ORed cùng nhau. Sau đây là các giá trị hợp lệ:

EPOLLERR

Đã xảy ra lỗi trên tệp. Sự kiện này luôn được theo dõi, ngay cả khi không được chỉ định.

EPOLLET

Cho phép hành vi kích hoạt cạnh cho màn hình của tệp (xem phần sắp tới "Sự kiện kích hoạt cạnh so với kích hoạt cấp độ"). Hành vi mặc định là kích hoạt cấp độ.

EPOLLHUP Đã

xảy ra sự cố treo máy trên tệp. Sự kiện này luôn được theo dõi, ngay cả khi không được chỉ định.

EPOLLIN

Có thể đọc tập tin mà không bị chặn.

EPOLLONESHOT

Sau khi sự kiện được tạo và đọc, tệp sẽ tự động không còn được giám sát nữa.
Mặt nạ sự kiện mới phải được chỉ định thông qua EPOLL_CTL_MOD để bật lại chế độ theo dõi.

EPOLLOUT Có

thể ghi vào tệp mà không bị chặn.

EPOLLPRI Có

dữ liệu ngoài băng tần khẩn cấp có thể đọc được.

Trường dữ liệu bên trong cấu trúc event_poll dành cho mục đích sử dụng riêng của người dùng. Nội dung được trả về cho người dùng khi nhận được sự kiện được yêu cầu. Thực hành phổ biến là đặt event.data.fd thành fd, giúp dễ dàng tra cứu mô tả tệp nào gây ra sự kiện.

Khi thành công, epoll_ctl() trả về 0. Khi thất bại, lệnh gọi trả về -1 và đặt errno thành một trong các giá trị sau:

EBADF

epfd không phải là một thể hiện epoll hợp lệ hoặc fd không phải là một mô tả tệp hợp lệ.

Op EEXIST

là EPOLL_CTL_ADD, nhưng fd đã được liên kết với epfd.

EINVAL

epfd không phải là một thể hiện epoll, epfd giống như fd hoặc op không hợp lệ.

Lệnh

ENOENT là EPOLL_CTL_MOD hoặc EPOLL_CTL_DEL, nhưng fd không liên kết với epfd.

ENOMEM

Không đủ bộ nhớ để xử lý yêu cầu.

EPERM

fd không hỗ trợ epoll.

Ví dụ, để thêm một watch mới vào tệp được liên kết với fd vào thể hiện epoll epfd, bạn sẽ viết:

```
struct epoll_event sự kiện;
int ret;

event.data.fd = fd; /* trả lại fd cho chúng ta sau */
event.events = EPOLLIN | EPOLLOUT;

ret = epoll_ctl (epfd, EPOLL_CTL_ADD, fd, &event); nếu
(ret)
      lỗi ("epoll_ctl");
```

Để sửa đổi một sự kiện hiện có trên tệp được liên kết với fd trên phiên bản epoll epfd, bạn sẽ viết:

```
struct epoll_event sự kiện;
int ret;

event.data.fd = fd; /* trả lại fd cho chúng ta sau */
event.events = EPOLLIN;

ret = epoll_ctl (epfd, EPOLL_CTL_MOD, fd, &event); nếu
(ret)
      lỗi ("epoll_ctl");
```

Ngược lại, để xóa một sự kiện hiện có trên tệp được liên kết với fd khỏi thể hiện epoll epfd, bạn sẽ viết:

```
struct epoll_event sự kiện;
int ret;

ret = epoll_ctl (epfd, EPOLL_CTL_DEL, fd, &event); nếu
(ret)
      lỗi ("epoll_ctl");
```

Lưu ý rằng tham số sự kiện có thể là NULL khi op là EPOLL_CTL_DEL, vì không có mặt nạ sự kiện nào để cung cấp. Tuy nhiên, các phiên bản Kernel trước 2.6.9 đã kiểm tra sai tham số này để biết tham số này không phải là NULL. Để có thể chuyển sang các kernel cũ hơn này, bạn nên truyền vào một con trỏ không phải NULL hợp lệ ; nó sẽ không bị động đến. Kernel 2.6.9 đã sửa lỗi này.

Chờ đợi sự kiện với Epoll

Cuộc gọi hệ thống epoll_wait() chờ các sự kiện trên các mô tả tệp được liên kết với phiên bản epoll đã cho:

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

int epoll_wait (int epfd,
                struct epoll_event *events, int
                maxevents, int
                timeout);
```

Một lệnh gọi đến epoll_wait() sẽ chờ đến thời gian chờ tính bằng mili giây cho các sự kiện trên các tệp được liên kết với phiên bản epoll epfd. Khi thành công, các sự kiện sẽ trỏ đến bộ nhớ chứa các cấu trúc epoll_event mô tả từng sự kiện, tối đa là maxevents sự kiện.
Giá trị trả về là số sự kiện hoặc -1 nếu xảy ra lỗi, trong trường hợp này errno được đặt thành một trong những giá trị sau:

EBADF

epfd không phải là mô tả tập tin hợp lệ.

MẶC ĐỊNH

Quá trình này không có quyền ghi vào bộ nhớ được các sự kiện trỏ tới.

TRỰC TIẾP

Cuộc gọi hệ thống bị ngắt bởi một tín hiệu trước khi nó có thể hoàn tất.

EINVAL

epfd không phải là một thể hiện epoll hợp lệ hoặc maxevents bằng hoặc nhỏ hơn 0.

Nếu thời gian chờ là 0, cuộc gọi sẽ trả về ngay lập tức, ngay cả khi không có sự kiện nào khả dụng, trong trường hợp đó, cuộc gọi sẽ trả về 0. Nếu thời gian chờ là -1, cuộc gọi sẽ không trả về cho đến khi có sự kiện khả dụng.

Khi cuộc gọi trả về, trường sự kiện của cấu trúc epoll_event mô tả các sự kiện đã xảy ra. Trường dữ liệu chứa bất kỳ thứ gì người dùng đặt trước khi gọi epoll_ctl( ).

Ví dụ đầy đủ về epoll_wait() trông như thế này:

```
#define SỰ KIỆN TỐI ĐA     64

cấu trúc epoll_event *sự kiện;
int nr_events, i, epfd;

sự kiện = malloc (sizeof (struct epoll_event) * MAX_EVENTS); nếu (!
events) { perror
          ("malloc"); trả về 1;

}

nr_events = epoll_wait (epfd, sự kiện, MAX_EVENTS, -1); nếu
(nr_events < 0) { lỗi
          ("epoll_wait");
```

```
            miễn phí (sự
            kiện); trả về 1;
    }

    đối với (i = 0; i < nr_events; i++)
            { printf ("sự kiện=%ld trên
                    fd=%d\n",
                    events[i].events, events[i].data.fd);

            /*
             * Bây giờ, theo events[i].events, chúng ta có thể vận hành trên *
             events[i].data.fd mà không cần chặn. */

    }

    miễn phí (sự kiện);
```

Chúng ta sẽ tìm hiểu về các hàm malloc() và free() trong Chương 8.

Sự kiện kích hoạt cạnh so với kích hoạt cấp độ Nếu

giá trị EPOLLET được đặt trong trường sự kiện của tham số sự kiện được truyền cho epoll_ctl( ) , thì việc theo dõi trên fd sẽ được kích hoạt cạnh, trái ngược với kích hoạt cấp độ.

Hãy xem xét các sự kiện sau đây giữa nhà sản xuất và người tiêu dùng giao tiếp qua đường ống Unix:

1. Nhà sản xuất ghi 1 KB dữ liệu vào đường ống.
2. Người tiêu dùng thực hiện lệnh epoll_wait() trên đường ống, chờ đường ống chứa dữ liệu và có thể đọc được.

Với một đồng hồ kích hoạt theo mức, lệnh gọi đến epoll_wait( ) ở bước 2 sẽ trả về ngay lập tức, cho thấy đường ống đã sẵn sàng để đọc. Với một đồng hồ kích hoạt theo cạnh, lệnh gọi này sẽ không trả về cho đến khi bước 1 xảy ra. Nghĩa là, ngay cả khi đường ống có thể đọc được khi gọi epoll_wait( ), lệnh gọi sẽ không trả về cho đến khi dữ liệu được ghi vào đường ống.

Level-triggered là hành vi mặc định. Đây là cách poll() và select() hoạt động và là điều mà hầu hết các nhà phát triển mong đợi. Hành vi Edge-triggered yêu cầu một cách tiếp cận khác đối với lập trình, thường sử dụng non-blocking I/O và kiểm tra cẩn thận EAGAIN.

Thuật ngữ này xuất phát từ kỹ thuật điện. Một ngắt kích hoạt mức được đưa ra bất cứ khi nào một đường được khẳng định. Một ngắt kích hoạt cạnh chỉ được gây ra trong quá trình tăng hoặc giảm cạnh của sự thay đổi trong khẳng định. Ngắt kích hoạt mức hữu ích khi trạng thái của sự kiện (đường được khẳng định) được quan tâm. Ngắt kích hoạt cạnh hữu ích khi bản thân sự kiện (đường được khẳng định) được quan tâm.

Ánh xạ các tập tin vào bộ nhớ

Là một giải pháp thay thế cho I/O tệp chuẩn, hạt nhân cung cấp một giao diện cho phép ứng dụng ánh xạ tệp vào bộ nhớ, nghĩa là có sự tương ứng một-một giữa địa chỉ bộ nhớ và một từ trong tệp. Sau đó, lập trình viên có thể truy cập tệp trực tiếp thông qua bộ nhớ, giống hệt với bất kỳ khối dữ liệu lưu trú trong bộ nhớ nào khác—thậm chí có thể cho phép ghi vào vùng bộ nhớ để ánh xạ ngược lại tệp trên đĩa một cách minh bạch.

POSIX.1 chuẩn hóa—và Linux triển khai— lệnh gọi hệ thống mmap( ) để ánh xạ các đối tượng vào bộ nhớ. Phần này sẽ thảo luận về mmap( ) liên quan đến việc ánh xạ các tệp vào bộ nhớ để thực hiện I/O; trong Chương 8, chúng ta sẽ xem xét các ứng dụng khác của mmap( ).

mmap( )

Một lệnh gọi đến mmap( ) yêu cầu kernel ánh xạ len byte của đối tượng được biểu diễn bởi mô tả tệp fd, bắt đầu từ byte offset vào tệp, vào bộ nhớ. Nếu addr được bao gồm, nó chỉ ra sở thích sử dụng địa chỉ bắt đầu đó trong bộ nhớ. Quyền truy cập được quyết định bởi prot và hành vi bổ sung có thể được cung cấp bởi cờ:

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

trống rỗng * mmap (void *addr,
                   size_t len,
                   int prot,
                   int flags,
                   int fd,
                   off_t offset);
```

Tham số addr đưa ra gợi ý cho kernel về nơi tốt nhất để ánh xạ tệp. Đây chỉ là gợi ý; hầu hết người dùng đều truyền 0. Lệnh gọi trả về địa chỉ thực trong bộ nhớ nơi bắt đầu ánh xạ.

Tham số prot mô tả mức độ bảo vệ bộ nhớ mong muốn của ánh xạ. Nó có thể là PROT_NONE, trong trường hợp đó các trang trong ánh xạ này có thể không được truy cập (không có ý nghĩa gì!), hoặc là OR bitwise của một hoặc nhiều cờ sau:

PROT_READ
: Các trang có thể được đọc.

PROT_WRITE
: Có thể ghi các trang.

PROT_EXEC
: Các trang có thể được thực thi.

Bảo vệ bộ nhớ mong muốn không được xung đột với chế độ mở của tệp. Ví dụ, nếu chương trình mở tệp chỉ đọc, prot không được chỉ định PROT_WRITE.

> Cờ bảo vệ, Kiến trúc và Bảo mật Trong khi POSIX định nghĩa
>
> bốn bit bảo vệ (đọc, ghi, thực thi và tránh xa), một số kiến trúc chỉ hỗ trợ một tập hợp con của các bit này. Ví dụ, bộ xử lý thường không phân biệt giữa các hành động đọc và thực thi. Trong trường hợp đó, bộ xử lý có thể chỉ có một cờ "đọc". Trên các hệ thống đó, PROT_READ ngụ ý PROT_EXEC. Cho đến gần đây, kiến trúc x86 là một hệ thống như vậy.
>
> Tất nhiên, việc dựa vào hành vi như vậy là không khả thi. Các chương trình khả thi phải luôn đặt PROT_EXEC nếu chúng có ý định thực thi mã trong ánh xạ.
>
> Tình huống ngược lại là một lý do khiến các cuộc tấn công tràn bộ đệm trở nên phổ biến: ngay cả khi ánh xạ nhất định không chỉ định quyền thực thi, bộ xử lý vẫn có thể cho phép thực thi.
>
> Bộ xử lý x86 gần đây đã giới thiệu bit NX (không thực thi), cho phép ánh xạ có thể đọc được nhưng không thể thực thi. Trên các hệ thống mới hơn này, PROT_READ không còn ngụ ý PROT_EXEC nữa.

Đối số flags mô tả loại ánh xạ và một số thành phần của hành vi của nó. Nó là phép OR bitwise của các giá trị sau:

MAP_FIXED
: Hướng dẫn mmap() xử lý addr như một yêu cầu, không phải là một gợi ý. Nếu kernel không thể đặt ánh xạ tại địa chỉ đã cho, lệnh gọi sẽ thất bại. Nếu tham số địa chỉ và độ dài chồng lên một ánh xạ hiện có, các trang chồng lên nhau sẽ bị loại bỏ và thay thế bằng ánh xạ mới. Vì tùy chọn này yêu cầu hiểu biết sâu sắc về không gian địa chỉ của quy trình, nên tùy chọn này không thể chuyển đổi và không được khuyến khích sử dụng.

MAP_PRIVATE
: Chỉ ra rằng ánh xạ không được chia sẻ. Tệp được ánh xạ sao chép khi ghi và mọi thay đổi được thực hiện trong bộ nhớ bởi quy trình này không được phản ánh trong tệp thực tế hoặc trong ánh xạ của các quy trình khác.

MAP_SHARED
: Chia sẻ bản đồ với tất cả các quy trình khác ánh xạ cùng tệp này. Ghi vào bản đồ tương đương với ghi vào tệp. Đọc từ bản đồ sẽ phản ánh các lần ghi của các quy trình khác.

Phải chỉ định MAP_SHARED hoặc MAP_PRIVATE , nhưng không phải cả hai. Các cờ khác, nâng cao hơn được thảo luận trong Chương 8.

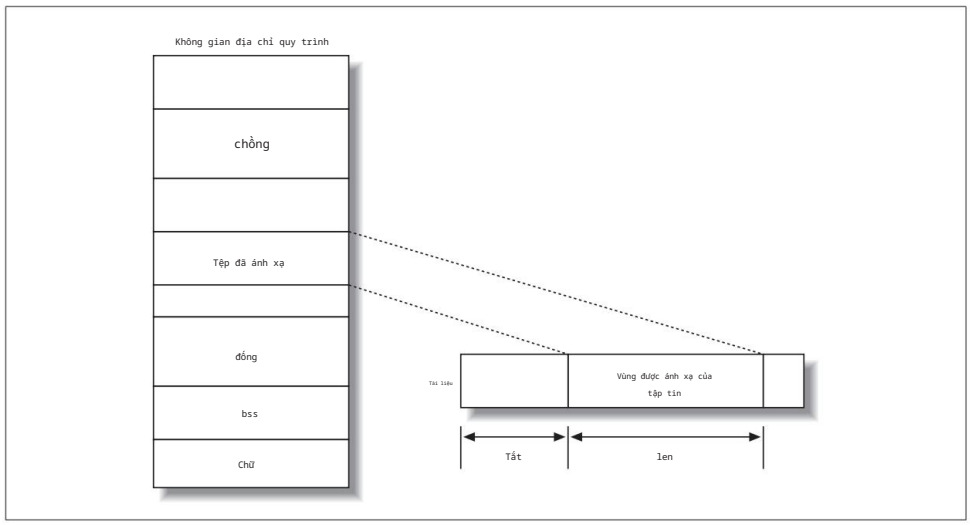
Khi bạn ánh xạ một mô tả tệp, số tham chiếu của tệp sẽ tăng lên. Do đó, bạn có thể đóng mô tả tệp sau khi ánh xạ tệp và quy trình của bạn vẫn có thể truy cập vào tệp đó. Việc giảm tương ứng số tham chiếu của tệp sẽ xảy ra khi bạn hủy ánh xạ tệp hoặc khi quy trình kết thúc.

Ví dụ, đoạn mã sau đây ánh xạ tệp được fd hỗ trợ, bắt đầu bằng byte đầu tiên và mở rộng thành len byte, thành ánh xạ chỉ đọc:

```
không có *p;

p = mmap (0, len, PROT_READ, MAP_SHARED, fd, 0); nếu (p == MAP_FAILED) perror ("mmap");
```

Hình 4-1 cho thấy tác động của các tham số được cung cấp với mmap() trên việc ánh xạ giữa tệp và không gian địa chỉ của một tiến trình.



Hình 4-1. Ánh xạ một tệp vào không gian địa chỉ của một quy trình

Kích thước

trang Trang là đơn vị bộ nhớ nhỏ nhất có thể có các quyền và hành vi riêng biệt. Do đó, trang là khối xây dựng của ánh xạ bộ nhớ, đến lượt nó là khối xây dựng của không gian địa chỉ quy trình.

Lệnh gọi hệ thống mmap() hoạt động trên các trang. Cả tham số addr và offset phải được căn chỉnh trên ranh giới có kích thước trang. Nghĩa là chúng phải là bội số nguyên của kích thước trang.

Do đó, các ánh xạ là bội số nguyên của các trang. Nếu tham số len do người gọi cung cấp không được căn chỉnh trên ranh giới trang—có lẽ vì kích thước của tệp cơ sở không phải là bội số của kích thước trang—ánh xạ được làm tròn lên trang đầy đủ tiếp theo. Các byte bên trong bộ nhớ được thêm vào này, giữa byte hợp lệ cuối cùng và phần cuối của ánh xạ, được điền bằng không. Bất kỳ lần đọc nào từ vùng đó sẽ trả về số không. Bất kỳ lần ghi nào vào bộ nhớ đó sẽ không ảnh hưởng đến tệp sao lưu, ngay cả khi nó được ánh xạ là MAP_SHARED. Chỉ có len byte ban đầu mới được ghi lại vào tệp.

sysconf( ). Phương pháp POSIX chuẩn để lấy kích thước trang là sử dụng sysconf( ), có thể truy xuất nhiều thông tin cụ thể của hệ thống:

```
#include <unistd.h>

sysconf dài (tên int);
```

Một lệnh gọi đến sysconf( ) trả về giá trị của mục cấu hình name, hoặc -1 nếu name không hợp lệ. Khi có lỗi, lệnh gọi sẽ đặt errno thành EINVAL. Vì -1 có thể là giá trị hợp lệ cho một số mục (ví dụ: limits, trong đó -1 nghĩa là không giới hạn), nên có thể xóa errno trước khi gọi và kiểm tra giá trị của nó sau đó.

POSIX định nghĩa _SC_PAGESIZE (và một từ đồng nghĩa, _SC_PAGE_SIZE) là kích thước của một trang, tính bằng byte. Do đó, việc lấy kích thước trang rất đơn giản:

```
kích thước trang dài = sysconf (_SC_PAGESIZE);
```

getpagesize(). Linux cũng cung cấp hàm getpagesize() :

```
#include <unistd.h>

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

Một lệnh gọi đến getpagesize() cũng sẽ trả về kích thước của một trang, tính bằng byte. Cách sử dụng thậm chí còn đơn giản hơn sysconf( ):

```
int kích thước trang = lấy kích thước trang ();
```

Không phải tất cả các hệ thống Unix đều hỗ trợ chức năng này; chức năng này đã bị loại bỏ khỏi bản sửa đổi 1003.1-2001 của tiêu chuẩn POSIX. Chức năng này được đưa vào đây để hoàn thiện.

PAGE_SIZE. Kích thước trang cũng được lưu trữ tĩnh trong macro PAGE_SIZE, được định nghĩa trong <asm/page.h>. Do đó, cách thứ ba có thể để lấy kích thước trang là:

```
int kích thước trang = KÍCH THƯỚC TRANG;
```

Tuy nhiên, không giống như hai tùy chọn đầu tiên, cách tiếp cận này lấy kích thước trang hệ thống tại thời điểm biên dịch chứ không phải thời gian chạy. Một số kiến trúc hỗ trợ nhiều loại máy với các kích thước trang khác nhau và một số loại máy thậm chí còn hỗ trợ nhiều kích thước trang! Một tệp nhị phân duy nhất phải có thể chạy trên tất cả các loại máy trong một kiến trúc nhất định—tức là bạn phải có thể xây dựng nó một lần và chạy nó ở mọi nơi.
Mã hóa cứng kích thước trang sẽ vô hiệu hóa khả năng đó. Do đó, bạn nên xác định kích thước trang khi chạy.
Vì addr và offset thường là 0, nên yêu cầu này không quá khó để đáp ứng.

Hơn nữa, các phiên bản kernel trong tương lai có thể sẽ không xuất macro này vào không gian người dùng. Chúng tôi sẽ đề cập đến nó trong chương này do nó thường xuyên xuất hiện trong mã Unix, nhưng bạn không nên sử dụng nó trong các chương trình của riêng bạn. Phương pháp sysconf() là lựa chọn tốt nhất của bạn.

Giá trị trả về và mã lỗi

Khi thành công, lệnh gọi đến mmap() trả về vị trí của bản đồ. Khi thất bại, lệnh gọi trả về MAP_FAILED và đặt errno một cách thích hợp. Lệnh gọi đến mmap() không bao giờ trả về 0.

Các giá trị errno có thể bao gồm:

TRUY CẬP

Mô tả tệp được cung cấp không phải là tệp thông thường hoặc chế độ mở tệp này xung đột với prot hoặc cờ.

EAGAIN

Tệp đã bị khóa thông qua khóa tệp.

EBADF

Mô tả tệp đã cho không hợp lệ.

EINVAL

Một hoặc nhiều tham số addr, len hoặc off không hợp lệ.

ENFILE

Đã đạt đến giới hạn toàn hệ thống về số lượng tệp đang mở.

ENODEV

Hệ thống tập tin mà tập tin cần ánh xạ nằm trên đó không hỗ trợ ánh xạ bộ nhớ.

ENOMEM

Quá trình này không có đủ bộ nhớ.

EOVERFLOW Kết

quả của addr+len vượt quá kích thước của không gian địa chỉ.

Đã đưa

ra EPERM PROT_EXEC , nhưng hệ thống tập tin được gắn kết noexec.

Tín hiệu liên quan

Có hai tín hiệu liên quan đến các vùng được ánh xạ:

SIGBUS

Tín hiệu này được tạo ra khi một tiến trình cố gắng truy cập vào một vùng ánh xạ không còn hợp lệ nữa, ví dụ như khi tệp bị cắt bớt sau khi được ánh xạ.

SIGSEGV

Tín hiệu này được tạo ra khi một tiến trình cố gắng ghi vào vùng được ánh xạ chỉ đọc.

Bản đồ mun( )

Linux cung cấp lệnh gọi hệ thống munmap() để xóa ánh xạ được tạo bằng mmap( ):

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

int munmap (void *addr, size_t len);
```

Một lệnh gọi đến munmap( ) sẽ xóa bất kỳ ánh xạ nào chứa các trang nằm ở bất kỳ đâu trong không gian địa chỉ quy trình bắt đầu từ addr, phải được căn chỉnh theo trang và tiếp tục cho len byte. Sau khi ánh xạ đã bị xóa, vùng bộ nhớ được liên kết trước đó không còn hợp lệ nữa và các lần truy cập tiếp theo sẽ dẫn đến tín hiệu SIGSEGV .

Thông thường, munmap() được truyền giá trị trả về và tham số len từ lần gọi mmap( ) trước đó .

Khi thành công, munmap() trả về 0; khi thất bại, nó trả về -1 và errno được thiết lập phù hợp. Giá trị errno chuẩn duy nhất là EINVAL, chỉ định rằng một hoặc nhiều tham số không hợp lệ.

Ví dụ, đoạn mã sau đây sẽ hủy ánh xạ bất kỳ vùng bộ nhớ nào có các trang nằm trong khoảng [addr,addr+len]:

```
nếu (munmap (địa chỉ, len) == -1)
        lỗi ("munmap");
```

Ví dụ về bản đồ

Hãy xem xét một chương trình ví dụ đơn giản sử dụng mmap() để in một tệp do người dùng chọn để chuẩn hóa:

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

int chính (int argc, char *argv[]) {

        cấu trúc stat sb;
        off_t len;
        char *p;
        int fd;

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

        fd = mở (argv[1], O_RDONLY); nếu
        (fd == -1)
                { perror ("mở");
                trả về 1;
        }

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

        }
```

```
        nếu (!S_ISREG (sb.st_mode))
                { fprintf (stderr, "%s không phải là tệp\n", argv[1]);
                trả về 1;
        }

        p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0); nếu (p
        == MAP_FAILED) { perror
                ("mmap"); trả về 1;

        }

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

        for (len = 0; len < sb.st_size; len++)
                putchar (p[len]);

        nếu (munmap (p, sb.st_size) == -1)
                { perror ("munmap");
                trả về 1;
        }

        trả về 0;
}
```

Lệnh gọi hệ thống không quen thuộc duy nhất trong ví dụ này là fstat( ), chúng ta sẽ đề cập đến nó trong Chương 7. Tất cả những gì bạn cần biết tại thời điểm này là fstat() trả về thông tin về một tệp nhất định. Macro S_ISREG() có thể kiểm tra một số thông tin này, để chúng ta có thể đảm bảo rằng tệp nhất định là tệp thông thường (khác với tệp thiết bị hoặc thư mục) trước khi chúng ta ánh xạ tệp đó. Hành vi của các tệp không thông thường khi được ánh xạ phụ thuộc vào thiết bị sao lưu. Một số tệp thiết bị có thể mmap; các tệp không thông thường khác không thể mmap và sẽ đặt errno thành EACCESS.

Phần còn lại của ví dụ sẽ rất đơn giản. Chương trình được truyền một tên tệp làm đối số. Nó mở tệp, đảm bảo đó là tệp thông thường, ánh xạ tệp, đóng tệp, in tệp theo từng byte ra chuẩn đầu ra, sau đó hủy ánh xạ tệp khỏi bộ nhớ.

Ưu điểm của mmap()

Việc thao tác các tệp thông qua mmap( ) có một số lợi thế so với các lệnh gọi hệ thống read( ) và write( ) chuẩn . Trong số đó có:

- Đọc từ và ghi vào tệp được ánh xạ bộ nhớ giúp tránh việc sao chép không cần thiết xảy ra khi sử dụng lệnh gọi hệ thống read() hoặc write() , trong đó dữ liệu phải được sao chép vào và ra khỏi bộ đệm không gian người dùng.
- Ngoài bất kỳ lỗi trang tiềm ẩn nào, việc đọc và ghi vào tệp được ánh xạ bộ nhớ không gây ra bất kỳ cuộc gọi hệ thống hoặc chi phí chuyển đổi ngữ cảnh nào. Nó đơn giản như việc truy cập bộ nhớ.

- Khi nhiều tiến trình ánh xạ cùng một đối tượng vào bộ nhớ, dữ liệu được chia sẻ giữa tất cả các tiến trình. Ánh xạ chỉ đọc và ánh xạ có thể ghi được chia sẻ được chia sẻ toàn bộ; ánh xạ có thể ghi riêng tư có các trang chưa phải là COW (sao chép khi ghi) được chia sẻ.

- Tìm kiếm xung quanh bản đồ liên quan đến các thao tác con trỏ tầm thường. Không có cần có lệnh gọi hệ thống lseek() .

Vì những lý do này, mmap() là lựa chọn thông minh cho nhiều ứng dụng.

Nhược điểm của mmap( )

Có một số điểm cần lưu ý khi sử dụng mmap( ):

- Ánh xạ bộ nhớ luôn là một số nguyên trang có kích thước. Do đó, sự khác biệt giữa kích thước của tệp sao lưu và một số nguyên trang bị "lãng phí" như không gian trống. Đối với các tệp nhỏ, một tỷ lệ phần trăm đáng kể của ánh xạ có thể bị lãng phí. Ví dụ, với các trang 4 KB, ánh xạ 7 byte lãng phí 4.089 byte.

- Các ánh xạ bộ nhớ phải phù hợp với không gian địa chỉ của quy trình. Với không gian địa chỉ 32 bit, một số lượng lớn các ánh xạ có kích thước khác nhau có thể dẫn đến phân mảnh không gian địa chỉ, khiến việc tìm các vùng liền kề trống lớn trở nên khó khăn. Tất nhiên, vấn đề này ít rõ ràng hơn nhiều với không gian địa chỉ 64 bit.

- Có chi phí phát sinh trong việc tạo và duy trì ánh xạ bộ nhớ và các cấu trúc dữ liệu liên quan bên trong hạt nhân. Chi phí phát sinh này thường được khắc phục bằng cách loại bỏ bản sao kép được đề cập trong phần trước, đặc biệt là đối với các tệp lớn hơn và được truy cập thường xuyên.

Vì những lý do này, lợi ích của mmap( ) được nhận thấy rõ nhất khi tệp được ánh xạ có kích thước lớn (và do đó bất kỳ không gian lãng phí nào cũng chỉ là một phần trăm nhỏ trong tổng số ánh xạ) hoặc khi tổng kích thước của tệp được ánh xạ chia đều cho kích thước trang (và do đó không có không gian lãng phí).

Thay đổi kích thước của

một bản đồ Linux cung cấp lệnh gọi hệ thống mremap() để mở rộng hoặc thu hẹp kích thước của một bản đồ nhất định. Chức năng này dành riêng cho Linux:

```
#xác định _GNU_SOURCE

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

trống rỗng * mremap (void *addr, size_t old_size,
                     size_t new_size, cờ dài không dấu);
```

Một lệnh gọi đến mremap( ) mở rộng hoặc thu hẹp ánh xạ trong vùng [addr,addr+old_size) thành kích thước mới new_size. Kernel có khả năng di chuyển ánh xạ cùng lúc, tùy thuộc vào không gian khả dụng trong không gian địa chỉ của tiến trình và giá trị của cờ.

Dấu mở [trong [addr,addr+old_size) chỉ ra rằng vùng bắt đầu bằng (và bao gồm) địa chỉ thấp, trong khi dấu đóng) chỉ ra rằng vùng dừng ngay trước (không bao gồm) địa chỉ cao.
Quy ước này được gọi là ký hiệu khoảng.

Tham số cờ có thể là 0 hoặc MREMAP_MAYMOVE, chỉ định rằng hạt nhân có thể tự do di chuyển ánh xạ, nếu cần, để thực hiện thay đổi kích thước theo yêu cầu. Việc thay đổi kích thước lớn có nhiều khả năng thành công hơn nếu hạt nhân có thể di chuyển ánh xạ.

Giá trị trả về và mã lỗi

Khi thành công, mremap() trả về một con trỏ đến bản đồ bộ nhớ mới được thay đổi kích thước. Khi thất bại, nó trả về MAP_FAILED và đặt errno thành một trong những giá trị sau:

LẠI LẦN NỮA
: Vùng bộ nhớ bị khóa và không thể thay đổi kích thước.

MẶC ĐỊNH
: Một số trang trong phạm vi đã cho không phải là trang hợp lệ trong không gian địa chỉ của quy trình hoặc đã xảy ra sự cố khi ánh xạ lại các trang đã cho.

EINVAL
: Một lập luận là không hợp lệ.

ENOMEM
: Không thể mở rộng phạm vi đã cho nếu không di chuyển (và MREMAP_MAYMOVE không được cung cấp) hoặc không có đủ không gian trống trong không gian địa chỉ của quy trình.

Các thư viện như glibc thường sử dụng mremap() để triển khai realloc( ) hiệu quả, đây là giao diện để thay đổi kích thước khối bộ nhớ ban đầu lấy được thông qua malloc( ). Ví dụ:

```
void * realloc (void *addr, size_t len) {

        size_t old_size = look_up_mapping_size (địa chỉ);
        void *p;

        p = mremap (addr, old_size, len, MREMAP_MAYMOVE); nếu (p
        == MAP_FAILED) trả về
                NULL; trả về
        p;
}
```

Điều này chỉ có hiệu quả nếu tất cả các phân bổ malloc() đều là các ánh xạ ẩn danh duy nhất; tuy nhiên, nó vẫn là một ví dụ hữu ích về mức tăng hiệu suất có thể đạt được. Ví dụ này giả định rằng lập trình viên đã viết một hàm look_up_mapping_size() .

Thư viện GNUC sử dụng mmap() và họ để thực hiện một số phân bổ bộ nhớ. Chúng ta sẽ xem xét chủ đề đó sâu hơn trong Chương 8.

Thay đổi bảo vệ của bản đồ

POSIX định nghĩa giao diện mprotect() để cho phép các chương trình thay đổi quyền của các vùng bộ nhớ hiện có:

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

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

Một lệnh gọi đến mprotect() sẽ thay đổi chế độ bảo vệ cho các trang bộ nhớ chứa trong [addr,addr+len), trong đó addr được căn chỉnh theo trang. Tham số prot chấp nhận các giá trị giống như prot được cung cấp cho mmap( ): PROT_NONE, PROT_READ, PROT_WRITE và PROT_EXEC. Các giá trị này không phải là phép cộng; nếu một vùng bộ nhớ có thể đọc được và prot được đặt thành chỉ PROT_WRITE, lệnh gọi sẽ làm cho vùng đó chỉ có thể ghi được.

Trên một số hệ thống, mprotect() chỉ có thể hoạt động trên các ánh xạ bộ nhớ được tạo trước đó thông qua mmap( ). Trên Linux, mprotect() có thể hoạt động trên bất kỳ vùng bộ nhớ nào.

Giá trị trả về và mã lỗi

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

TRUY CẬP

Bộ nhớ không thể được cấp các quyền mà prot yêu cầu. Điều này có thể xảy ra, ví dụ, nếu bạn cố gắng thiết lập ánh xạ của tệp đã mở chỉ đọc thành có thể ghi.

EINVAL

Tham số addr không hợp lệ hoặc không được căn chỉnh theo trang.

ENOMEM

Bộ nhớ hạt nhân không đủ để đáp ứng yêu cầu hoặc một hoặc nhiều trang trong vùng bộ nhớ đã cho không phải là một phần hợp lệ của địa chỉ quy trình
không gian.

Đồng bộ hóa một tập tin với một bản đồ

POSIX cung cấp một lệnh gọi hệ thống fsync( ) được ánh xạ vào bộ nhớ tương đương mà chúng ta đã thảo luận trong Chương 2:

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

int msync (void *addr, size_t len, int cờ);
```

Một lệnh gọi đến msync() sẽ đẩy ngược trở lại đĩa bất kỳ thay đổi nào được thực hiện đối với tệp được ánh xạ qua mmap( ), đồng bộ hóa tệp được ánh xạ với ánh xạ. Cụ thể, tệp hoặc tập hợp con của tệp được liên kết với ánh xạ bắt đầu tại địa chỉ bộ nhớ addr và tiếp tục cho len byte được đồng bộ hóa với đĩa. Đối số addr phải được căn chỉnh theo trang; nó thường là giá trị trả về từ một lệnh gọi mmap() trước đó .

Nếu không gọi msync( ), sẽ không có đảm bảo rằng một ánh xạ bẩn sẽ được ghi lại vào đĩa cho đến khi tệp được bỏ ánh xạ. Điều này khác với hành vi của write( ), trong đó bộ đệm bị bẩn như một phần của quá trình ghi và được xếp hàng để ghi lại vào đĩa. Khi ghi vào ánh xạ bộ nhớ, quá trình này sẽ trực tiếp sửa đổi các trang của tệp trong bộ đệm trang của hạt nhân, mà không cần sự tham gia của hạt nhân. Hạt nhân có thể không đồng bộ hóa bộ đệm trang và đĩa bất cứ lúc nào sớm.

Tham số cờ kiểm soát hành vi của hoạt động đồng bộ hóa. Đây là phép OR từng bit của các giá trị sau:

MS_ASYNC
: Chỉ định rằng đồng bộ hóa phải diễn ra không đồng bộ. Bản cập nhật được lên lịch, nhưng lệnh gọi msync() trả về ngay lập tức mà không cần chờ quá trình ghi diễn ra.

MS_INVALIDATE
: Chỉ định rằng tất cả các bản sao lưu đệm khác của ánh xạ sẽ bị vô hiệu hóa. Bất kỳ quyền truy cập nào trong tương lai vào bất kỳ ánh xạ nào của tệp này sẽ phản ánh trên đĩa mới được đồng bộ hóa nội dung.

MS_SYNC
: Chỉ định rằng đồng bộ hóa phải diễn ra đồng bộ. Lệnh gọi msync() sẽ không trả về cho đến khi tất cả các trang được ghi lại vào đĩa.

Phải chỉ định MS_ASYNC hoặc MS_SYNC , nhưng không được chỉ định cả hai.

Cách sử dụng rất đơn giản:

```
nếu (msync (địa chỉ, len, MS_ASYNC) ==
        -1) lỗi ("msync");
```

Ví dụ này đồng bộ hóa không đồng bộ (có thể nói là nhanh gấp 10 lần) với đĩa tệp được ánh xạ trong vùng [addr,addr+len).

Giá trị trả về và mã lỗi

Khi thành công, msync() trả về 0. Khi thất bại, lệnh gọi trả về -1 và đặt errno một cách thích hợp. Sau đây là các giá trị errno hợp lệ :

EINVAL
: Tham số cờ có cả MS_SYNC và MS_ASYNC được đặt, trừ khi một trong ba cờ hợp lệ được đặt hoặc địa chỉ không được căn chỉnh theo trang.

ENOMEM

Vùng bộ nhớ đã cho (hoặc một phần của nó) không được ánh xạ. Lưu ý rằng Linux sẽ trả về ENOMEM, theo quy định của POSIX, khi được yêu cầu đồng bộ hóa một vùng chỉ chưa được ánh xạ một phần, nhưng nó vẫn sẽ đồng bộ hóa bất kỳ ánh xạ hợp lệ nào trong vùng đó.

Trước phiên bản 2.4.19 của hạt nhân Linux, msync() trả về EFAULT thay cho ENOMEM.

Đưa ra lời khuyên về một bản đồ Linux cung

cấp một lệnh gọi hệ thống có tên là madvise( ) để cho phép các tiến trình đưa ra lời khuyên và gợi ý cho hạt nhân về cách chúng định sử dụng một bản đồ. Sau đó, hạt nhân có thể tối ưu hóa hành vi của mình để tận dụng mục đích sử dụng dự định của bản đồ. Trong khi hạt nhân Linux điều chỉnh hành vi của mình một cách động và thường cung cấp hiệu suất tối ưu mà không cần lời khuyên rõ ràng, việc cung cấp lời khuyên như vậy có thể đảm bảo hành vi lưu trữ đệm và đọc trước mong muốn cho một số khối lượng công việc.

Một lệnh gọi đến madvise( ) sẽ tư vấn cho kernel về cách ứng xử liên quan đến các trang trong bản đồ bộ nhớ bắt đầu từ addr và mở rộng cho len byte:

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

int madvise (void *addr,
             size_t len,
             int lời khuyên);
```

Nếu len bằng 0, kernel sẽ áp dụng lời khuyên cho toàn bộ ánh xạ bắt đầu từ addr.
Tham số advice mô tả lời khuyên, có thể là một trong những lời khuyên sau:

MADV_BÌNH_NHẬT

Ứng dụng không có lời khuyên cụ thể nào về phạm vi bộ nhớ này. Nó nên được coi là bình thường.

MADV_RANDOM

Ứng dụng có ý định truy cập các trang trong phạm vi được chỉ định theo thứ tự ngẫu nhiên (không theo trình tự).

MADV_SEQUENTIAL

Ứng dụng có ý định truy cập các trang trong phạm vi được chỉ định theo trình tự, từ địa chỉ thấp đến địa chỉ cao.

MADV_WILLNEED

Ứng dụng có ý định truy cập các trang trong phạm vi được chỉ định trong tương lai gần.

MADV_DONTNEED

Ứng dụng không có ý định truy cập vào các trang trong phạm vi đã chỉ định trong tương lai gần.

Các sửa đổi hành vi thực tế mà hạt nhân thực hiện để phản hồi lời khuyên này là cụ thể cho từng triển khai: POSIX chỉ quyết định ý nghĩa của lời khuyên, không phải bất kỳ hậu quả tiềm ẩn nào. Hạt nhân 2.6 hiện tại hoạt động như sau để phản hồi các giá trị lời khuyên :

MADV_NORMAL

Hạt nhân hoạt động như bình thường, thực hiện một lượng đọc trước vừa phải.

MADV_RANDOM

Hạt nhân vô hiệu hóa chức năng đọc trước, chỉ đọc lượng dữ liệu tối thiểu trên mỗi thao tác đọc vật lý.

MADV_SEQUENTIAL

Hạt nhân thực hiện đọc trước một cách chủ động.

MADV_WILLNEED

Hạt nhân khởi tạo lệnh đọc trước, đọc các trang đã cho vào bộ nhớ.

MADV_DONTNEED

Kernel giải phóng mọi tài nguyên liên quan đến các trang đã cho và loại bỏ mọi trang bẩn và chưa được đồng bộ hóa. Các lần truy cập tiếp theo vào dữ liệu được ánh xạ sẽ khiến dữ liệu được phân trang từ tệp sao lưu.

Cách sử dụng thông thường là:

```
int ret;

ret = madvise (addr, len, MADV_SEQUENTIAL);
nếu (ret <
        0) perror ("madvise");
```

Cuộc gọi này hướng dẫn cho hạt nhân rằng tiến trình này có ý định truy cập vào vùng bộ nhớ [addr,addr+len) theo trình tự.

Đọc trước

Khi nhân Linux đọc các tệp từ đĩa, nó thực hiện một quá trình tối ưu hóa được gọi là đọc trước. Nghĩa là, khi một yêu cầu được thực hiện cho một đoạn nhất định của một tệp, nhân cũng đọc đoạn tiếp theo của tệp. Nếu một yêu cầu được thực hiện sau đó cho đoạn đó—như trường hợp khi đọc tệp theo trình tự—nhân có thể trả về dữ liệu được yêu cầu ngay lập tức. Vì đĩa có bộ đệm theo dõi (về cơ bản, đĩa cứng thực hiện đọc trước nội bộ của riêng chúng) và vì các tệp thường được sắp xếp theo trình tự trên đĩa, nên quá trình tối ưu hóa này có chi phí thấp.

Một số readahead thường có lợi, nhưng kết quả tối ưu phụ thuộc vào câu hỏi về việc thực hiện readahead bao nhiêu. Một tệp được truy cập tuần tự có thể được hưởng lợi từ cửa sổ readahead lớn hơn, trong khi một tệp được truy cập ngẫu nhiên có thể thấy readahead là chi phí không đáng kể.

Như đã thảo luận trong “Kernel Internals” ở Chương 2, kernel điều chỉnh động kích thước của cửa sổ đọc trước để phản hồi tỷ lệ trúng đích bên trong cửa sổ đó. Nhiều lần trúng đích ngụ ý rằng một cửa sổ lớn hơn sẽ có lợi thế; ít lần trúng đích hơn gợi ý một cửa sổ nhỏ hơn. Lệnh gọi hệ thống madvise() cho phép các ứng dụng tác động đến kích thước cửa sổ ngay từ đầu.

Giá trị trả về và mã lỗi

Khi thành công, madvise() trả về 0. Khi thất bại, nó trả về -1 và errno được thiết lập phù hợp. Sau đây là các lỗi hợp lệ:

EAGAIN

Tài nguyên hạt nhân bên trong (có thể là bộ nhớ) không khả dụng. Quá trình có thể thử lại.

EBADF

Vùng này tồn tại nhưng không ánh xạ tệp.

EINVAL

Tham số len là số âm, addr không được căn chỉnh theo trang, tham số advice không hợp lệ hoặc các trang đã bị khóa hoặc chia sẻ với MADV_DONTNEED.

EIO

Đã xảy ra lỗi I/O nội bộ với MADV_WILLNEED.

ENOMEM

Vùng đã cho không phải là ánh xạ hợp lệ trong không gian địa chỉ của quy trình này hoặc MADV_WILLNEED đã được cung cấp nhưng không có đủ bộ nhớ để phân trang trong các vùng đã cho.

Lời khuyên cho I/O tệp thông thường

Trong tiểu mục trước, chúng ta đã xem xét việc cung cấp lời khuyên về ánh xạ bộ nhớ. Trong phần này, chúng ta sẽ xem xét việc cung cấp lời khuyên cho hạt nhân về I/O tệp thông thường. Linux cung cấp hai giao diện để cung cấp lời khuyên như vậy: posix_fadvise() và readahead( ).

Cuộc gọi hệ thống posix_fadvise( ) Giao diện

tư vấn đầu tiên, như tên gọi của nó ám chỉ, được chuẩn hóa bởi POSIX 1003.1-2003:

```
#include <fcntl.h>

int posix_fadvise (int fd,
                   off_t offset,
                   off_t len,
                   int advice);
```

Một lệnh gọi đến posix_fadvise( ) cung cấp cho kernel lời khuyên gợi ý về tệp descrip-tor fd trong khoảng [offset,offset+len). Nếu len là 0, lời khuyên sẽ áp dụng cho phạm vi [offset,length of file]. Cách sử dụng phổ biến là chỉ định 0 cho len và offset, áp dụng lời khuyên cho toàn bộ tệp.

Các tùy chọn tư vấn có sẵn tương tự như các tùy chọn dành cho madvise( ). Chính xác một trong những tùy chọn sau đây phải được cung cấp để tư vấn:

POSIX_FADV_NORMAL Ứng
dụng không có lời khuyên cụ thể nào để đưa ra về phạm vi tệp này. Nó phải được coi là bình thường.

POSIX_FADV_RANDOM Ứng
dụng có ý định truy cập dữ liệu trong phạm vi được chỉ định theo thứ tự ngẫu nhiên (không tuần tự).

POSIX_FADV_SEQUENTIAL Ứng
dụng có ý định truy cập dữ liệu trong phạm vi được chỉ định theo trình tự, từ địa chỉ thấp đến địa chỉ cao.

POSIX_FADV_WILLNEED Ứng
dụng có ý định truy cập dữ liệu trong phạm vi đã chỉ định trong tương lai gần.

POSIX_FADV_NOREUSE Ứng
dụng có ý định truy cập dữ liệu trong phạm vi được chỉ định trong tương lai gần, nhưng chỉ một lần.

POSIX_FADV_DONTNEED Ứng
dụng không có ý định truy cập các trang trong phạm vi đã chỉ định trong tương lai gần.

Giống như madvise( ), phản hồi thực tế cho lời khuyên đưa ra là cụ thể cho từng triển khai—thậm chí các phiên bản khác nhau của hạt nhân Linux cũng có thể phản ứng khác nhau. Sau đây là các phản hồi hiện tại:

POSIX_FADV_NORMAL Hạt
nhân hoạt động như bình thường, thực hiện một lượng đọc trước vừa phải.

POSIX_FADV_RANDOM Hạt
nhân vô hiệu hóa chức năng đọc trước, chỉ đọc lượng dữ liệu tối thiểu trên mỗi thao tác đọc vật lý.

POSIX_FADV_SEQUENTIAL Hạt
nhân thực hiện đọc trước một cách mạnh mẽ, tăng gấp đôi kích thước của cửa sổ đọc trước.

POSIX_FADV_WILLNEED Hạt
nhân khởi tạo lệnh readahead để bắt đầu đọc vào bộ nhớ các trang đã cho.

POSIX_FADV_NOREUSE Hiện
tại, hành vi giống như đối với POSIX_FADV_WILLNEED; các hạt nhân trong tương lai có thể thực hiện tối ưu hóa bổ sung để khai thác hành vi “sử dụng một lần”. Gợi ý này không có phần bổ sung madvise() .

POSIX_FADV_DONTNEED
Kernel xóa bất kỳ dữ liệu được lưu trong bộ nhớ đệm nào trong phạm vi đã cho khỏi bộ nhớ đệm trang. Lưu ý rằng gợi ý này, không giống như những gợi ý khác, có hành vi khác với madvise() tương ứng.

Ví dụ, đoạn mã sau đây hướng dẫn hạt nhân rằng toàn bộ tệp được biểu diễn bằng mô tả tệp fd sẽ được truy cập theo cách ngẫu nhiên, không tuần tự:

```
int ret;

ret = posix_fadvise (fd, 0, 0, POSIX_FADV_RANDOM);
nếu (ret ==
        -1) lỗi ("posix_fadvise");
```

Giá trị trả về và mã lỗi

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

EBADF

Mô tả tập tin đã cho không hợp lệ.

EINVAL

Lời khuyên đưa ra không hợp lệ, mô tả tệp đưa ra tham chiếu đến một đường ống hoặc lời khuyên được chỉ định không thể áp dụng cho tệp đưa ra.

Lệnh gọi hệ thống readahead() Lệnh gọi hệ

thống posix_fadvise () là lệnh gọi mới trong nhân Linux 2.6. Trước đó, lệnh gọi hệ thống readahead() có sẵn để cung cấp hành vi giống hệt với gợi ý POSIX_FADV_WILLNEED .

Không giống như posix_fadvise( ), readahead() là một giao diện dành riêng cho Linux:

```
#include <fcntl.h>

ssize_t đọc trước (int fd,
                   off64_t bù trừ,
                   size_t đếm);
```

Một lệnh gọi tới readahead() sẽ điền vào bộ nhớ đệm trang vùng [offset,offset+count) từ mô tả tệp fd.

Giá trị trả về và mã lỗi

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

EBADF

Mô tả tập tin đã cho không hợp lệ.

EINVAL

Mô tả tệp đã cho không ánh xạ tới tệp hỗ trợ đọc trước.

Lời khuyên là rẻ Một số

lượng nhỏ khối lượng công việc ứng dụng phổ biến có thể dễ dàng được hưởng lợi từ một lời khuyên có thiện chí nhỏ cho hạt nhân. Lời khuyên như vậy có thể đi một chặng đường dài hướng tới việc giảm thiểu

gánh nặng của I/O. Với ổ cứng chậm như vậy, và bộ xử lý hiện đại thì nhanh như vậy, mọi sự giúp đỡ nhỏ đều có ích, và lời khuyên tốt có thể giúp ích rất nhiều.

Trước khi đọc một đoạn của tệp, một tiến trình có thể cung cấp gợi ý POSIX_FADV_WILLNEED để hướng dẫn hạt nhân đọc tệp vào bộ đệm trang. I/O sẽ diễn ra không đồng bộ, ở chế độ nền. Khi ứng dụng cuối cùng truy cập tệp, thao tác có thể hoàn tất mà không tạo ra I/O chặn.

Ngược lại, sau khi đọc hoặc ghi nhiều dữ liệu—ví dụ, trong khi liên tục truyền video vào đĩa—một tiến trình có thể cung cấp gợi ý POSIX_FADV_DONTNEED để hướng dẫn nhân xóa phần tệp đã cho khỏi bộ đệm trang. Một hoạt động truyền phát lớn có thể liên tục lấp đầy bộ đệm trang. Nếu ứng dụng không bao giờ có ý định truy cập dữ liệu nữa, điều này có nghĩa là bộ đệm trang sẽ được lấp đầy bằng dữ liệu thừa, với cái giá phải trả là dữ liệu hữu ích hơn. Do đó, việc ứng dụng truyền phát video yêu cầu định kỳ xóa dữ liệu đã truyền phát khỏi bộ đệm là hợp lý.

Một tiến trình có ý định đọc toàn bộ một tệp có thể cung cấp gợi ý POSIX_FADV_SEQUENTIAL , hướng dẫn kernel thực hiện readahead tích cực. Ngược lại, một tiến trình biết rằng nó sẽ truy cập tệp một cách ngẫu nhiên, tìm kiếm qua lại, có thể cung cấp gợi ý POSIX_FADV_RANDOM , hướng dẫn kernel rằng readahead sẽ chẳng có giá trị gì ngoài chi phí vô ích.

Đồng bộ, Đồng bộ và Không đồng bộ Hoạt động

Các hệ thống Unix sử dụng các thuật ngữ synchronized, nonsynchronized, synchronize và asynchronous một cách thoải mái, mà không quan tâm nhiều đến thực tế là chúng dễ gây nhầm lẫn—trong tiếng Anh, sự khác biệt giữa “synchronous” và “synchronized” không đáng kể!

Một hoạt động ghi đồng bộ không trả về cho đến khi dữ liệu đã ghi được— ít nhất là —lưu trữ trong bộ đệm đệm của hạt nhân. Một hoạt động đọc đồng bộ không trả về cho đến khi dữ liệu đã đọc được lưu trữ trong bộ đệm không gian người dùng do ứng dụng cung cấp. Mặt khác, một hoạt động ghi không đồng bộ có thể trả về trước khi dữ liệu thậm chí rời khỏi không gian người dùng; một hoạt động đọc không đồng bộ có thể trả về trước khi dữ liệu đã đọc khả dụng. Nghĩa là, các hoạt động chỉ có thể được xếp hàng để thực hiện sau. Tất nhiên, trong trường hợp này, phải có một số cơ chế để xác định thời điểm hoạt động thực sự hoàn tất và mức độ thành công như thế nào.

Hoạt động đồng bộ hóa hạn chế hơn và an toàn hơn hoạt động đồng bộ đơn thuần. Hoạt động ghi đồng bộ hóa sẽ xả dữ liệu vào đĩa, đảm bảo dữ liệu trên đĩa luôn được đồng bộ hóa so với bộ đệm hạt nhân tương ứng. Hoạt động đọc đồng bộ hóa luôn trả về bản sao dữ liệu mới nhất, có thể là từ đĩa.

Tóm lại, các thuật ngữ đồng bộ và không đồng bộ đề cập đến việc liệu các hoạt động I/O có chờ một số sự kiện (ví dụ: lưu trữ dữ liệu) trước khi trả về hay không. Trong khi đó, các thuật ngữ đồng bộ hóa và không đồng bộ hóa chỉ rõ chính xác sự kiện nào phải xảy ra (ví dụ: ghi dữ liệu vào đĩa).

Thông thường, các hoạt động ghi của Unix là đồng bộ và không đồng bộ; các hoạt động đọc là đồng bộ và đồng bộ.* Đối với các hoạt động ghi, mọi sự kết hợp của các đặc điểm này đều có thể thực hiện được, như Bảng 4-1 minh họa.

Bảng 4-1. Đồng bộ các hoạt động ghi

	Các hoạt động	Các hoạt động ghi
Đồng bộ	ghi đồng bộ sẽ không trả về cho đến khi dữ liệu được lưu vào đĩa. Đây là hành vi nếu O_SYNC được chỉ định trong khi mở tệp.	không đồng bộ sẽ không trả về cho đến khi dữ liệu được lưu trữ trong bộ đệm hạt nhân. Đây là hành vi thông thường.
Không đồng bộ	Các hoạt động ghi sẽ trả về ngay khi yêu cầu được xếp hàng. Khi hoạt động ghi cuối cùng được thực hiện, dữ liệu được đảm bảo sẽ nằm trên đĩa.	Các hoạt động ghi sẽ trả về ngay khi yêu cầu được xếp hàng. Khi hoạt động ghi cuối cùng được thực thi, dữ liệu được đảm bảo ít nhất được lưu trữ trong bộ đệm hạt nhân.

Các hoạt động đọc luôn được đồng bộ hóa vì việc đọc dữ liệu cũ không có nhiều ý nghĩa. Tuy nhiên, các hoạt động như vậy có thể đồng bộ hoặc không đồng bộ, như minh họa trong Bảng 4-2.

Bảng 4-2. Đồng bộ các hoạt động đọc

	Các hoạt động
Đồng bộ	Đọc đồng bộ sẽ không trả về cho đến khi dữ liệu được cập nhật và lưu trữ trong bộ đệm được cung cấp (đây là hành vi thông thường).
Không đồng bộ	Hoạt động đọc sẽ trả về ngay khi yêu cầu được xếp hàng, nhưng khi hoạt động đọc cuối cùng được thực thi, dữ liệu trả về sẽ là dữ liệu mới nhất.

Trong Chương 2, chúng ta đã thảo luận về cách thực hiện đồng bộ hóa các lệnh ghi (thông qua cờ O_SYNC) và cách đảm bảo rằng tất cả các I/O được đồng bộ hóa tại một thời điểm nhất định (thông qua fsync() và các lệnh tương tự). Bây giờ, hãy xem xét những gì cần thiết để thực hiện các lệnh đọc và ghi không đồng bộ.

I/O không đồng bộ Thực

hiện I/O không đồng bộ yêu cầu hỗ trợ hạt nhân ở các lớp thấp nhất.
POSIX 1003.1-2003 định nghĩa các giao diện aio mà may mắn là Linux đã triển khai.
Thư viện aio cung cấp một nhóm các hàm để gửi I/O không đồng bộ và nhận thông báo khi hoàn tất:

* Các hoạt động đọc về mặt kỹ thuật cũng không được đồng bộ hóa, giống như các hoạt động ghi, nhưng hạt nhân đảm bảo rằng bộ đệm trang chứa dữ liệu cập nhật. Nghĩa là, dữ liệu của bộ đệm trang luôn giống hệt hoặc mới hơn dữ liệu trên đĩa. Theo cách này, hành vi trong thực tế luôn được đồng bộ hóa. Có rất ít lý lẽ để hành xử theo bất kỳ cách nào khác.

```
#include <aio.h>

/* khối điều khiển I/O không đồng bộ */
struct aiocb
        { int aio_filedes; /* mô tả tệp */ int aio_lio_opcode; /*
        thao tác thực hiện */ int aio_reqprio; /* yêu cầu độ lệch ưu
        tiên */ volatile void *aio_buf; /* con trỏ đến bộ đệm */ size_t
        aio_nbytes; /* độ dài của thao tác */ struct sigevent
        aio_sigevent; /* số tín hiệu và giá trị */


        /* các thành viên nội bộ, riêng tư theo sau... */
};

int aio_read (struct aiocb *aiocbp); int
aio_write (struct aiocb *aiocbp); int
aio_error (const struct aiocb *aiocbp); int
aio_return (struct aiocb *aiocbp); int
aio_cancel (int fd, struct aiocb *aiocbp); int
aio_fsync (int op, struct aiocb *aiocbp); int
aio_suspend (const struct aiocb * const cblist[],
                  int n,
                  const struct timespec *timeout);
```

Linux chỉ hỗ trợ aio trên các

tệp được mở bằng cờ O_DIRECT . Để thực hiện I/ O không đồng bộ trên các tệp thông thường được mở mà không có O_DIRECT, chúng ta phải nhìn vào bên trong, hướng tới một giải pháp của riêng mình. Nếu không có hỗ trợ hạt nhân, chúng ta chỉ có thể hy vọng xấp xỉ I/O không đồng bộ, đưa ra kết quả tương tự như thực tế.

Trước tiên, hãy xem lý do tại sao một nhà phát triển ứng dụng lại muốn có I/O không

đồng bộ: • Thực hiện I/O mà không bị

chặn • Tách biệt các hành động xếp hàng I/O, gửi I/O tới kernel và nhận thông báo hoàn tất thao tác

Điểm đầu tiên là vấn đề về hiệu suất. Nếu các hoạt động I/O không bao giờ bị chặn, chi phí quá mức của I/O sẽ bằng không và một quy trình không cần phải bị ràng buộc bởi I/O. Điểm thứ hai là vấn đề về thủ tục, chỉ đơn giản là một phương pháp xử lý I/O khác.

Cách phổ biến nhất để đạt được các mục tiêu này là sử dụng luồng (các vấn đề lập lịch được thảo luận kỹ lưỡng trong Chương 5 và 6). Cách tiếp cận này bao gồm các tác vụ lập trình sau:

1. Tạo một nhóm “luồng công việc” để xử lý tất cả I/O.
2. Triển khai một bộ giao diện để đưa các hoạt động I/O vào hàng đợi công việc.
3. Yêu cầu mỗi giao diện này trả về một mô tả I/O xác định duy nhất hoạt động I/O liên quan. Trong mỗi luồng công nhân, lấy các yêu cầu I/O từ đầu hàng đợi và gửi chúng, chờ hoàn tất.

4. Sau khi hoàn tất, đưa kết quả của hoạt động (giá trị trả về, mã lỗi, bất kỳ dữ liệu đã đọc nào) vào hàng đợi kết quả.
5. Triển khai một bộ giao diện để truy xuất thông tin trạng thái từ hàng đợi kết quả, sử dụng các mô tả I/O trả về ban đầu để xác định từng thao tác.

Điều này cung cấp hành vi tương tự như giao diện aio của POSIX , mặc dù có chi phí quản lý luồng lớn hơn.

Bộ lập lịch I/O và hiệu suất I/O

Trong một hệ thống hiện đại, khoảng cách hiệu suất tương đối giữa các đĩa và phần còn lại của hệ thống khá lớn—và ngày càng mở rộng. Thành phần tệ nhất của hiệu suất đĩa là quá trình di chuyển đầu đọc/ghi từ một phần của đĩa sang phần khác, một hoạt động được gọi là tìm kiếm. Trong một thế giới mà nhiều hoạt động được đo bằng một số chu kỳ bộ xử lý (có thể mất tất cả một phần ba nano giây mỗi lần), một lần tìm kiếm đĩa đơn có thể trung bình hơn tám mili giây—vẫn là một con số nhỏ, chắc chắn rồi, nhưng dài hơn 25 triệu lần so với một chu kỳ bộ xử lý đơn!

Với sự chênh lệch về hiệu suất giữa các ổ đĩa và phần còn lại của hệ thống, sẽ vô cùng thô thiển và kém hiệu quả khi gửi các yêu cầu I/O đến đĩa theo thứ tự chúng được phát hành. Do đó, các hạt nhân hệ điều hành hiện đại triển khai các trình lập lịch I/O, hoạt động để giảm thiểu số lượng và kích thước của các tìm kiếm đĩa bằng cách thao tác thứ tự các yêu cầu I/O được phục vụ và thời điểm chúng được phục vụ. Các trình lập lịch I/O hoạt động rất hiệu quả để giảm thiểu các hình phạt về hiệu suất liên quan đến truy cập đĩa.

Địa chỉ đĩa Để hiểu

vai trò của bộ lập lịch I/O, một số thông tin cơ bản là cần thiết. Đĩa cứng định địa chỉ dữ liệu của chúng bằng cách sử dụng địa chỉ dựa trên hình học quen thuộc của các trụ, đầu và sector, hoặc địa chỉ CHS. Một ổ cứng bao gồm nhiều đĩa, mỗi đĩa bao gồm một đĩa đơn, trục chính và đầu đọc/ghi. Bạn có thể nghĩ về mỗi đĩa như một đĩa CD (hoặc bản ghi) và tập hợp các đĩa trong một đĩa như một chồng đĩa CD. Mỗi đĩa được chia thành các rãnh giống như vòng tròn, giống như trên một đĩa CD. Sau đó, mỗi rãnh được chia thành một số nguyên sector.

Để định vị một đơn vị dữ liệu cụ thể trên đĩa, logic của ổ đĩa yêu cầu ba thông tin: giá trị xi lanh, đầu đọc và sector. Giá trị xi lanh chỉ định track mà dữ liệu nằm trên đó. Nếu bạn đặt các đĩa lên trên nhau, một track nhất định sẽ tạo thành một xi lanh qua mỗi đĩa. Nói cách khác, một xi lanh được biểu diễn bằng một track ở cùng khoảng cách từ tâm trên mỗi đĩa. Giá trị đầu đọc xác định chính xác đầu đọc/ghi (và do đó là chính xác đĩa) đang được đề cập. Bây giờ, tìm kiếm được thu hẹp lại thành một track duy nhất trên một đĩa duy nhất. Sau đó, đĩa sử dụng giá trị sector-tor để xác định một sector chính xác trên track. Bây giờ, tìm kiếm đã hoàn tất:

ổ cứng biết đĩa nào, track nào và sector nào để tìm dữ liệu. Nó có thể định vị đầu đọc/ghi của đĩa đúng trên track đúng và đọc từ hoặc ghi vào sector cần thiết.

May mắn thay, các ổ cứng hiện đại không buộc máy tính phải giao tiếp với đĩa của chúng theo các xi lanh, đầu đọc và sector. Thay vào đó, các ổ cứng hiện đại ánh xạ một số khối duy nhất (còn gọi là khối vật lý hoặc khối thiết bị) trên mỗi bộ ba xi lanh/đầu đọc/sector—trên thực tế, một khối ánh xạ tới một sector cụ thể. Sau đó, các hệ điều hành hiện đại có thể định địa chỉ ổ cứng bằng các số khối này—một quy trình được gọi là định địa chỉ khối logic (LBA)—và ổ cứng sẽ dịch số khối đó thành địa chỉ CHS chính xác.* Mặc dù không có gì đảm bảo, nhưng ánh xạ khối sang CHS có xu hướng tuần tự: khối vật lý n có xu hướng nằm cạnh vật lý trên đĩa với khối logic n + 1. Ánh xạ tuần tự này rất quan trọng, như chúng ta sẽ sớm thấy.

Trong khi đó, hệ thống tập tin chỉ tồn tại trong phần mềm. Chúng hoạt động trên các đơn vị riêng của chúng, được gọi là khối logic (đôi khi được gọi là khối hệ thống tập tin hoặc, gây nhầm lẫn, chỉ là khối). Kích thước khối logic phải là bội số nguyên của kích thước khối vật lý. Nói cách khác, các khối logic của hệ thống tập tin sẽ ánh xạ tới một hoặc nhiều khối vật lý của đĩa.

Cuộc sống của một trình lập lịch I/O

Bộ lập lịch I/O thực hiện hai hoạt động cơ bản: hợp nhất và sắp xếp. Hợp nhất là quá trình lấy hai hoặc nhiều yêu cầu I/O liền kề và kết hợp chúng thành một yêu cầu duy nhất. Hãy xem xét hai yêu cầu, một yêu cầu để đọc từ khối đĩa 5 và một yêu cầu để đọc từ khối đĩa 6 đến 7. Các yêu cầu này có thể được hợp nhất thành một yêu cầu duy nhất để đọc từ khối đĩa 5 đến 7. Tổng lượng I/O có thể giống nhau, nhưng số lượng hoạt động I/O giảm đi một nửa.

Sắp xếp, thao tác quan trọng hơn trong hai thao tác, là quá trình sắp xếp các yêu cầu I/O đang chờ xử lý theo thứ tự khối tăng dần. Ví dụ, với các thao tác I/O cho các khối 52, 109 và 7, trình lập lịch I/O sẽ sắp xếp các yêu cầu này theo thứ tự 7, 52 và 109. Nếu sau đó một yêu cầu được đưa ra cho khối 81, nó sẽ được chèn vào giữa các yêu cầu cho khối 52 và 109. Sau đó, trình lập lịch I/O sẽ phân phối các yêu cầu đến đĩa theo thứ tự chúng tồn tại trong hàng đợi: 7, sau đó là 52, sau đó là 81 và cuối cùng là 109.

Theo cách này, chuyển động của đầu đĩa được giảm thiểu. Thay vì các chuyển động có khả năng ngẫu nhiên—từ đây đến đó và ngược lại, tìm kiếm khắp đĩa—đầu đĩa di chuyển theo cách trơn tru, tuyến tính. Vì tìm kiếm là phần tốn kém nhất của I/O đĩa, hiệu suất được cải thiện.

* Giới hạn về kích thước tuyệt đối của số khối này phần lớn chịu trách nhiệm cho các giới hạn khác nhau về tổng kích thước ổ đĩa trong nhiều năm.

Trợ giúp đọc Mỗi yêu cầu

đọc phải trả về dữ liệu cập nhật. Do đó, nếu dữ liệu được yêu cầu không có trong bộ đệm trang, quá trình đọc phải chặn cho đến khi dữ liệu có thể được đọc từ đĩa—một hoạt động có khả năng kéo dài. Chúng tôi gọi đây là độ trễ đọc tác động đến hiệu suất.

Một ứng dụng thông thường có thể khởi tạo nhiều yêu cầu I/O đọc trong một thời gian ngắn. Vì mỗi yêu cầu được đồng bộ hóa riêng lẻ, nên các yêu cầu sau phụ thuộc vào việc hoàn thành các yêu cầu trước đó. Hãy xem xét việc đọc mọi tệp trong một thư mục. Ứng dụng mở tệp đầu tiên, đọc một phần của tệp đó, đợi dữ liệu, đọc một phần khác, v.v., cho đến khi toàn bộ tệp được đọc. Sau đó, ứng dụng bắt đầu lại, trên tệp tiếp theo.
Các yêu cầu được tuần tự hóa: không thể đưa ra yêu cầu tiếp theo cho đến khi yêu cầu hiện tại hoàn tất.

Điều này hoàn toàn trái ngược với các yêu cầu ghi, (ở trạng thái mặc định, không đồng bộ) không cần khởi tạo bất kỳ I/O đĩa nào cho đến một thời điểm nào đó trong tương lai. Do đó, theo quan điểm của ứng dụng không gian người dùng, các yêu cầu ghi sẽ truyền phát, không bị cản trở bởi hiệu suất của đĩa. Hành vi truyền phát này chỉ làm trầm trọng thêm vấn đề đối với các lần đọc: khi ghi truyền phát, chúng có thể chiếm hết sự chú ý của nhân và đĩa. Hiện tượng này được gọi là vấn đề ghi-đói-đọc .

Nếu một trình lập lịch I/O luôn sắp xếp các yêu cầu mới theo thứ tự chèn, thì có thể làm chết đói các yêu cầu đến các khối xa vô thời hạn. Hãy xem xét ví dụ trước của chúng tôi. Nếu các yêu cầu mới liên tục được gửi đến các khối trong, chẳng hạn, 50, thì yêu cầu đến khối 109 sẽ không bao giờ được phục vụ. Vì độ trễ đọc là rất quan trọng, nên hành vi này sẽ làm giảm đáng kể hiệu suất hệ thống. Do đó, các trình lập lịch I/O sử dụng một cơ chế để ngăn chặn tình trạng chết đói.

Một cách tiếp cận đơn giản—chẳng hạn như cách tiếp cận được thực hiện bởi trình lập lịch I/O của nhân Linux 2.4, Linus Elevator*—là chỉ cần dừng sắp xếp chèn nếu có một yêu cầu đủ cũ trong hàng đợi. Điều này đánh đổi hiệu suất tổng thể để có được sự công bằng cho mỗi yêu cầu và, trong trường hợp đọc, cải thiện độ trễ. Vấn đề là phương pháp này hơi quá đơn giản. Nhận ra điều này, nhân Linux 2.6 đã chứng kiến sự sụp đổ của Linus Elevator và công bố một số trình lập lịch I/O mới thay thế.

Bộ lập lịch I/O thời hạn

Deadline I/O Scheduler được giới thiệu để giải quyết các vấn đề với trình lập lịch I/O 2.4 và các thuật toán thang máy truyền thống nói chung. Linus Elevator duy trì một danh sách được sắp xếp các yêu cầu I/O đang chờ xử lý. Yêu cầu I/O ở đầu hàng đợi là yêu cầu tiếp theo được phục vụ. Deadline I/O Scheduler giữ nguyên hàng đợi này, nhưng nâng cấp mọi thứ lên một bậc bằng cách giới thiệu hai hàng đợi bổ sung: hàng đợi FIFO đọc và hàng đợi FIFO ghi. Các mục trong mỗi hàng đợi này được sắp xếp theo thời gian gửi

* Đúng vậy, người đàn ông này có một trình lập lịch I/O được đặt theo tên ông. Trình lập lịch I/O đôi khi được gọi là thuật toán thang máy, vì chúng giải quyết một vấn đề tương tự như việc giữ cho thang máy chạy trơn tru.

(thực tế, lệnh vào trước thì lệnh ra trước). Hàng đợi FIFO đọc, như tên gọi của nó, chỉ chứa các yêu cầu đọc. Tương tự như vậy, hàng đợi FIFO ghi chỉ chứa các yêu cầu ghi. Mỗi yêu cầu trong hàng đợi FIFO được gán một giá trị hết hạn. Hàng đợi FIFO đọc có thời gian hết hạn là 500 mili giây. Hàng đợi FIFO ghi có thời gian hết hạn là năm giây.

Khi một yêu cầu I/O mới được gửi, nó được sắp xếp chèn vào hàng đợi chuẩn và được đặt ở cuối hàng đợi FIFO tương ứng (đọc hoặc ghi). Thông thường, ổ cứng được gửi các yêu cầu I/O từ đầu hàng đợi được sắp xếp chuẩn. Điều này tối đa hóa thông lượng toàn cầu bằng cách giảm thiểu tìm kiếm, vì hàng đợi bình thường được sắp xếp theo số khối (như với Linus Elevator).

Tuy nhiên, khi mục ở đầu một trong các hàng đợi FIFO trở nên cũ hơn giá trị hết hạn liên quan đến hàng đợi của nó, trình lập lịch I/O sẽ ngừng phân phối các yêu cầu I/O từ hàng đợi chuẩn và bắt đầu phục vụ các yêu cầu từ hàng đợi đó—yêu cầu ở đầu hàng đợi FIFO được phục vụ, cộng thêm một vài yêu cầu bổ sung để đảm bảo. Trình lập lịch I/O cần kiểm tra và chỉ xử lý các yêu cầu ở đầu hàng đợi, vì đó là những yêu cầu cũ nhất.

Theo cách này, Deadline I/O Scheduler có thể áp dụng một thời hạn mềm cho các yêu cầu I/O. Mặc dù không hứa rằng một yêu cầu I/O sẽ được phục vụ trước thời hạn hết hạn của nó, nhưng nói chung, I/O Scheduler sẽ phục vụ các yêu cầu gần thời hạn hết hạn của chúng. Do đó, Deadline I/O Scheduler tiếp tục cung cấp thông lượng toàn cầu tốt mà không làm chết đói bất kỳ yêu cầu nào trong một thời gian dài không thể chấp nhận được. Vì các yêu cầu đọc được cấp thời hạn hết hạn ngắn hơn, nên vấn đề ghi-sống-đọc được giảm thiểu.

Bộ lập lịch I/O dự đoán Hành

vi của Bộ lập lịch I/O Deadline là tốt, nhưng không hoàn hảo. Hãy nhớ lại cuộc thảo luận của chúng ta về sự phụ thuộc vào đọc. Với Bộ lập lịch I/O Deadline, yêu cầu đọc đầu tiên trong một loạt các lần đọc được phục vụ trong thời gian ngắn, tại hoặc trước thời điểm hết hạn của nó, và sau đó bộ lập lịch I/O sẽ quay lại phục vụ các yêu cầu I/O từ hàng đợi đã sắp xếp—cho đến nay, mọi thứ vẫn ổn. Nhưng giả sử ứng dụng sau đó lao vào và gửi cho chúng ta một yêu cầu đọc khác? Cuối cùng, thời điểm hết hạn của nó cũng sẽ đến gần và bộ lập lịch I/O sẽ gửi nó đến đĩa, đĩa sẽ tìm kiếm để xử lý yêu cầu ngay lập tức, sau đó tìm kiếm lại để tiếp tục xử lý các yêu cầu từ hàng đợi đã sắp xếp. Việc tìm kiếm qua lại này có thể tiếp tục trong một thời gian vì nhiều ứng dụng thể hiện hành vi này. Trong khi độ trễ được giữ ở mức tối thiểu, thông lượng toàn cục không tốt lắm vì các yêu cầu đọc liên tục đến và đĩa phải liên tục tìm kiếm qua lại để xử lý chúng. Hiệu suất sẽ được cải thiện nếu đĩa chỉ tạm dừng để chờ một lần đọc khác và không di chuyển đi nơi khác để phục vụ hàng đợi đã sắp xếp một lần nữa. Nhưng thật không may, khi ứng dụng được lên lịch và gửi yêu cầu đọc phụ thuộc tiếp theo, trình lập lịch I/O đã chuyển số.

Vấn đề lại bắt nguồn từ những lần đọc phụ thuộc chết tiệt đó—mỗi yêu cầu đọc mới chỉ được đưa ra khi yêu cầu trước đó được trả về, nhưng khi ứng dụng nhận được dữ liệu đọc, được lên lịch chạy và gửi yêu cầu đọc tiếp theo, trình lập lịch I/O đã di chuyển và bắt đầu phục vụ các yêu cầu khác. Điều này dẫn đến một cặp tìm kiếm bị lãng phí cho mỗi lần đọc: đĩa tìm kiếm đến lần đọc, phục vụ nó, rồi tìm kiếm lại. Giá như có một cách nào đó để trình lập lịch I/O biết—để dự đoán—rằng một lần đọc khác sẽ sớm được gửi đến cùng một phần của đĩa, thay vì tìm kiếm qua lại, nó có thể đợi để dự đoán lần đọc tiếp theo. Việc lưu những lần tìm kiếm khủng khiếp đó chắc chắn sẽ đáng giá vài mili giây chờ đợi.

Đây chính xác là cách Anticipatory I/O Scheduler hoạt động. Nó bắt đầu tồn tại như Deadline I/O Scheduler, nhưng được tặng kèm thêm một cơ chế dự đoán. Khi một yêu cầu đọc được gửi đi, Anticipatory I/O Scheduler sẽ phục vụ yêu cầu đó trong thời hạn của nó, như thường lệ. Tuy nhiên, không giống như Deadline I/O Scheduler, Anticipatory I/O Scheduler sau đó sẽ ngồi và chờ, không làm gì cả, trong tối đa sáu mili giây.
Rất có khả năng ứng dụng sẽ đưa ra một lệnh đọc khác đến cùng một phần của hệ thống tệp trong sáu mili giây đó. Nếu vậy, yêu cầu đó sẽ được phục vụ ngay lập tức và Anticipatory I/O Scheduler sẽ chờ thêm một chút. Nếu sáu mili giây trôi qua mà không có yêu cầu đọc nào, Anticipatory I/O Scheduler sẽ quyết định rằng nó đã đoán sai và quay lại bất cứ việc gì nó đã làm trước đó (tức là phục vụ hàng đợi được sắp xếp theo tiêu chuẩn). Ngay cả khi một số lượng yêu cầu vừa phải được dự đoán chính xác, thì vẫn tiết kiệm được rất nhiều thời gian—hai lần tìm kiếm tốn kém ở mỗi lần thực hiện. Vì hầu hết các lần đọc đều phụ thuộc, nên việc dự đoán sẽ mang lại lợi ích trong phần lớn thời gian.

Bộ lập lịch I/O CFQ

Bộ lập lịch I/O Complete Fair Queuing (CFQ) hoạt động để đạt được các mục tiêu tương tự, mặc dù thông qua một cách tiếp cận khác.* Với CFQ, mỗi quy trình được chỉ định hàng đợi riêng và mỗi hàng đợi được chỉ định một khoảng thời gian. Bộ lập lịch I/O truy cập từng hàng đợi theo kiểu vòng tròn, phục vụ các yêu cầu từ hàng đợi cho đến khi khoảng thời gian của hàng đợi hết hoặc cho đến khi không còn yêu cầu nào nữa. Trong trường hợp sau, Bộ lập lịch I/O CFQ sau đó sẽ ở trạng thái nhàn rỗi trong một khoảng thời gian ngắn—theo mặc định là 10 ms—chờ một yêu cầu mới trên hàng đợi. Nếu dự đoán có hiệu quả, bộ lập lịch I/O sẽ tránh tìm kiếm. Nếu không, việc chờ đợi là vô ích và bộ lập lịch sẽ chuyển sang hàng đợi của quy trình tiếp theo.

Trong hàng đợi của mỗi tiến trình, các yêu cầu được đồng bộ hóa (như đọc) được ưu tiên hơn các yêu cầu không được đồng bộ hóa. Theo cách này, CFQ ưu tiên đọc và ngăn ngừa vấn đề ghi-đói-đọc. Do thiết lập hàng đợi theo từng tiến trình, CFQ I/O Scheduler công bằng với tất cả các tiến trình, đồng thời vẫn cung cấp hiệu suất toàn cục tốt.

CFQ I/O Scheduler phù hợp với hầu hết các khối lượng công việc và là lựa chọn đầu tiên tuyệt vời.

* Văn bản sau đây thảo luận về CFQ I/O Scheduler khi nó được triển khai hiện tại. Các phiên bản trước không sử dụng timeslices hoặc thuật toán dự đoán, nhưng hoạt động theo cách tương tự.

Noop I/O Scheduler Noop

I/O Scheduler là trình lập lịch cơ bản nhất hiện có. Nó không thực hiện bất kỳ sắp xếp nào, chỉ thực hiện hợp nhất cơ bản. Nó được sử dụng cho các thiết bị chuyên dụng không yêu cầu (hoặc thực hiện) sắp xếp yêu cầu riêng của chúng.

Chọn và Cấu hình Bộ lập lịch I/O của Bạn Bộ lập lịch I/O mặc định có thể được chọn tại thời điểm khởi động thông qua tham số dòng lệnh của hạt nhân iosched . Các tùy chọn hợp lệ là , cfq, deadline và noop. Bộ lập lịch I/O cũng có thể được chọn trong thời gian chạy trên cơ sở từng thiết bị thông qua /sys/block/device/queue/scheduler, trong đó device là thiết bị khối đang được đề cập. Đọc tệp này trả về bộ lập lịch I/O hiện tại; việc ghi một trong các tùy chọn hợp lệ vào tệp này sẽ đặt bộ lập lịch I/O. Ví dụ: để đặt hda của thiết bị thành Bộ lập lịch I/O CFQ, bạn sẽ thực hiện như sau:

```
# echo cfq > /sys/block/hda/queue/scheduler
```

Thư mục /sys/block/device/queue/iosched chứa các tệp cho phép người quản trị truy xuất và đặt các giá trị có thể điều chỉnh liên quan đến trình lập lịch I/O. Các tùy chọn chính xác phụ thuộc vào trình lập lịch I/O hiện tại. Việc thay đổi bất kỳ cài đặt nào trong số này đều yêu cầu quyền root.

Một lập trình viên giỏi viết các chương trình không phụ thuộc vào hệ thống con I/O cơ bản. Tuy nhiên, kiến thức về hệ thống con này chắc chắn có thể giúp người ta viết mã tối ưu.

Tối ưu hóa hiệu suất I/O Vì I/O đĩa chậm hơn nhiều so với hiệu suất của các thành phần khác trong hệ thống, nhưng I/O lại là một khía cạnh quan trọng của điện toán hiện đại, nên việc tối đa hóa hiệu suất I/O là rất quan trọng.

Giảm thiểu các hoạt động I/O (bằng cách hợp nhất nhiều hoạt động nhỏ hơn thành ít hoạt động lớn hơn), thực hiện I/O theo kích thước khối hoặc sử dụng bộ đệm người dùng (xem Chương 3) và tận dụng các kỹ thuật I/O tiên tiến, chẳng hạn như I/O theo vectơ, I/O theo vị trí (xem Chương 2) và I/O không đồng bộ là những bước quan trọng cần luôn cân nhắc khi lập trình hệ thống.

Tuy nhiên, các ứng dụng quan trọng và I/O đòi hỏi khắt khe nhất có thể sử dụng các thủ thuật bổ sung để tối đa hóa hiệu suất. Mặc dù hạt nhân Linux, như đã thảo luận trước đó, sử dụng các trình lập lịch I/O nâng cao để giảm thiểu các tìm kiếm đĩa đáng sợ, các ứng dụng không gian người dùng có thể hoạt động theo cùng một cách tương tự để cải thiện hiệu suất hơn nữa.

Lên lịch I/O trong không gian

người dùng Các ứng dụng sử dụng nhiều I/O phát ra một số lượng lớn yêu cầu I/O và cần trích xuất từng ounce hiệu suất có thể sắp xếp và hợp nhất các yêu cầu I/O đang chờ xử lý của chúng, thực hiện các nhiệm vụ giống như trình lập lịch I/O của Linux. * Tại sao phải

thực hiện cùng một công việc hai lần, nếu bạn biết trình lập lịch I/O sẽ sắp xếp các yêu cầu theo khối, giảm thiểu tìm kiếm và cho phép đầu đĩa di chuyển theo cách tuyến tính, mượt mà? Hãy xem xét một ứng dụng gửi một số lượng lớn các yêu cầu I/O chưa được sắp xếp. Các yêu cầu này đến hàng đợi của trình lập lịch I/O theo thứ tự ngẫu nhiên. Trình lập lịch I/O thực hiện công việc của mình, sắp xếp và hợp nhất các yêu cầu trước khi gửi chúng ra đĩa—nhưng các yêu cầu bắt đầu truy cập vào đĩa trong khi ứng dụng vẫn đang tạo I/O và gửi yêu cầu. Trình lập lịch I/O chỉ có thể sắp xếp một tập hợp nhỏ các yêu cầu—ví dụ, một số ít từ ứng dụng này và bất kỳ yêu cầu nào khác đang chờ xử lý—tại một thời điểm. Mỗi đợt yêu cầu của ứng dụng đều được sắp xếp gọn gàng, nhưng toàn bộ hàng đợi và bất kỳ yêu cầu nào trong tương lai đều không được tính vào.

Do đó, nếu một ứng dụng tạo ra nhiều yêu cầu, đặc biệt là nếu chúng liên quan đến dữ liệu trên toàn bộ đĩa, thì ứng dụng có thể được hưởng lợi từ việc sắp xếp các yêu cầu trước khi gửi chúng, đảm bảo chúng đến được bộ lập lịch I/O theo thứ tự mong muốn.

Tuy nhiên, một ứng dụng không gian người dùng không được trao quyền truy cập vào cùng thông tin như hạt nhân. Ở các cấp độ thấp nhất bên trong trình lập lịch I/O, các yêu cầu đã được chỉ định theo khối đĩa vật lý. Việc sắp xếp chúng là điều dễ dàng. Nhưng trong không gian người dùng, các yêu cầu được chỉ định theo tệp và độ lệch. Các ứng dụng không gian người dùng phải thăm dò thông tin và đưa ra những phỏng đoán có căn cứ về bố cục của hệ thống tệp.

Với mục tiêu xác định thứ tự tìm kiếm thân thiện nhất dựa trên danh sách các yêu cầu I/O đối với các tệp cụ thể, các ứng dụng không gian người dùng có một số tùy chọn. Chúng có thể sắp xếp dựa trên:

- Đường dẫn đầy đủ
- Số inode
- Khối đĩa vật lý của tập tin

Mỗi lựa chọn này đều có sự đánh đổi. Chúng ta hãy cùng xem xét từng lựa chọn một cách ngắn gọn.

Sắp xếp theo đường dẫn. Sắp xếp theo tên đường dẫn là cách dễ nhất, nhưng kém hiệu quả nhất, để xấp xỉ sắp xếp theo khối. Do các thuật toán bố trí được hầu hết các hệ thống tệp sử dụng, các tệp trong mỗi thư mục—và do đó các thư mục chia sẻ thư mục cha—có xu hướng nằm cạnh nhau trên đĩa. Xác suất các tệp trong cùng một thư mục được tạo ra cùng thời điểm chỉ khuếch đại đặc điểm này.

* Người ta chỉ nên áp dụng các kỹ thuật được thảo luận ở đây cho các ứng dụng quan trọng, đòi hỏi nhiều I/O. Việc sắp xếp các yêu cầu I/O—giả sử thậm chí có bất kỳ thứ gì để sắp xếp—của các ứng dụng không đưa ra nhiều yêu cầu như vậy là ngớ ngẩn và không cần thiết.

Do đó, sắp xếp theo đường dẫn gần đúng với vị trí vật lý của các tệp trên đĩa. Chắc chắn là hai tệp trong cùng một thư mục có nhiều khả năng nằm gần nhau hơn hai tệp ở các phần hoàn toàn khác nhau của hệ thống tệp. Nhược điểm của cách tiếp cận này là nó không tính đến sự phân mảnh: hệ thống tệp càng phân mảnh thì việc sắp xếp theo đường dẫn càng ít hữu ích. Ngay cả khi bỏ qua sự phân mảnh, sắp xếp theo đường dẫn chỉ gần đúng với thứ tự theo khối thực tế. Về mặt tích cực, sắp xếp theo đường dẫn ít nhất cũng có thể áp dụng cho tất cả các hệ thống tệp. Bất kể cách tiếp cận bố cục tệp nào, vị trí thời gian cho thấy sắp xếp theo đường dẫn sẽ ít nhất là chính xác một cách nhẹ nhàng. Đây cũng là một cách sắp xếp dễ thực hiện.

Sắp xếp theo inode. Inode là các cấu trúc Unix chứa siêu dữ liệu liên quan đến từng tệp. Trong khi dữ liệu của tệp có thể sử dụng nhiều khối đĩa vật lý, thì mỗi tệp chỉ có một inode, chứa thông tin như kích thước tệp, quyền, chủ sở hữu, v.v. Chúng ta sẽ thảo luận sâu hơn về inode trong Chương 7. Hiện tại, bạn cần biết hai sự thật: mọi tệp đều có một inode liên quan đến nó và các inode được gán các số duy nhất.

Sắp xếp theo inode tốt hơn sắp xếp theo đường dẫn, giả sử rằng mối quan hệ này:

```
số inode của file i < số inode của file j
```

ngụ ý, nói chung, rằng:

```
khối vật lý của tệp i < khối vật lý của tệp j
```

Điều này chắc chắn đúng đối với các hệ thống tập tin theo phong cách Unix như ext2 và ext3. Mọi thứ đều có thể đối với các hệ thống tập tin không sử dụng inode thực tế, nhưng số inode (bất kể nó có thể ánh xạ tới cái gì) vẫn là một phép tính gần đúng bậc nhất tốt.

Việc lấy số inode được thực hiện thông qua lệnh gọi hệ thống stat( ) , cũng được thảo luận trong Chương 7. Với inode được liên kết với tệp liên quan đến mỗi yêu cầu I/O, các yêu cầu có thể được sắp xếp theo thứ tự tăng dần theo số inode.

Sau đây là một chương trình đơn giản in ra số inode của một tệp nhất định:

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

/*
 * get_inode - trả về inode của tệp được liên kết * với mô tả tệp đã cho
 hoặc -1 nếu lỗi */

int lấy_inode (int fd) {

        cấu trúc stat
        buf; int ret;

        ret = fstat(fd, &buf);
```

```
        nếu (ret < 0)
                { perror ("fstat");
                trả về -1;
        }

        trả về buf.st_ino;
}

int chính (int argc, char *argv[]) {

        int fd, inode;

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

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

        inode = get_inode(fd);
        printf ("%d\n", inode);

        trả về 0;
}
```

Hàm get_inode() có thể dễ dàng thích ứng để sử dụng trong chương trình của bạn.

Sắp xếp theo số inode có một số ưu điểm: số inode dễ lấy, dễ sắp xếp và là phép xấp xỉ tốt cho bố cục tệp vật lý. Nhược điểm chính là phân mảnh làm giảm phép xấp xỉ, phép xấp xỉ chỉ là phỏng đoán và phép xấp xỉ kém chính xác hơn đối với các hệ thống tệp không phải Unix.

Tuy nhiên, đây là phương pháp được sử dụng phổ biến nhất để lập lịch các yêu cầu I/O trong không gian người dùng.

Sắp xếp theo khối vật lý. Tất nhiên, cách tiếp cận tốt nhất để thiết kế thuật toán thang máy của riêng bạn là sắp xếp theo khối đĩa vật lý. Như đã thảo luận trước đó, mỗi tệp được chia thành các khối logic, là các đơn vị phân bổ nhỏ nhất của một hệ thống tệp. Kích thước của một khối logic phụ thuộc vào hệ thống tệp; mỗi khối logic ánh xạ tới một khối vật lý duy nhất. Do đó, chúng ta có thể tìm số khối logic trong một tệp, xác định chúng ánh xạ tới khối vật lý nào và sắp xếp dựa trên điều đó.

Nhân cung cấp phương pháp để lấy khối đĩa vật lý từ số khối logic của tệp. Điều này được thực hiện thông qua lệnh gọi hệ thống ioctl() , được thảo luận trong Chương 7, với lệnh FIBMAP :

```
ret = ioctl (fd, FIBMAP, &block);
nếu (ret <
        0) lỗi ("ioctl");
```

Ở đây, fd là mô tả tệp của tệp đang xét, và block là khối logic mà chúng ta muốn xác định khối vật lý của nó. Khi trả về thành công, block được thay thế bằng số khối vật lý. Các khối logic được truyền vào có chỉ mục bằng không và tương đối với tệp. Nghĩa là, nếu một tệp được tạo thành từ tám khối logic, các giá trị hợp lệ là từ 0 đến 7.

Do đó, việc tìm ánh xạ logic-to-physical-block là một quá trình gồm hai bước. Đầu tiên, chúng ta phải xác định số khối trong một tệp nhất định. Điều này được thực hiện thông qua lệnh gọi hệ thống stat() . Thứ hai, đối với mỗi khối logic, chúng ta phải đưa ra yêu cầu ioctl() để tìm khối vật lý tương ứng.

Sau đây là một chương trình mẫu để thực hiện điều đó cho một tệp được truyền vào trên dòng lệnh:

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

/*
 * get_block - đối với tệp được liên kết với fd đã cho, trả về * khối vật lý ánh xạ tới
 logical_block */

int lấy_khối (int fd, int khối_logic) {

          int ret;

          ret = ioctl (fd, FIBMAP, &logical_block); nếu
          (ret < 0) { lỗi
                    ("ioctl"); trả về
                    -1;
          }

          trả về logical_block;
}

/*
 * get_nr_blocks - trả về số khối logic * được sử dụng bởi tệp được liên
 kết với fd */

int lấy_nr_blocks (int fd) {

          cấu trúc stat buf;
          int ret;

          ret = fstat (fd, &buf);
          nếu (ret < 0)
                    { perror ("fstat");
                    trả về -1;
          }
```

```
        trả về buf.st_blocks;
}

/*
 * print_blocks - đối với mỗi khối logic được sử dụng bởi tệp * liên kết với fd, in ra
 để chuẩn hóa bộ * "(khối logic, khối vật lý)" */


void print_blocks (int fd) {

        int nr_blocks, i;

        nr_blocks = get_nr_blocks (fd); nếu
        (nr_blocks < 0)
                { fprintf (stderr, "get_nr_blocks không thành công!
                \n"); trả về;
        }

        nếu (nr_blocks == 0)
                { printf ("không có khối nào được phân
                bổ\n");
        trả về; } nếu không thì nếu
                (nr_blocks == 1) printf ("1 khối\n\n");
        khác
                printf ("%d khối\n\n", nr_blocks);

        đối với (i = 0; i < nr_blocks; i++)
                { int phys_block;

                phys_block = get_block (fd, i); nếu
                (phys_block < 0) {
                        fprintf (stderr, "get_block không thành công!\n");
                        trả về;

                } nếu (!phys_block)
                        tiếp tục;

                printf ("(%u, %u) ", i, khối vật lý);
        }

        putchar('\n');
}

int chính (int argc, char *argv[]) {

        số nguyên fd;

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

        fd = mở (argv[1], O_RDONLY); nếu
        (fd < 0) {
```

```
                perror ("mở"); trả
                về 1;
        }

        khối in (fd);

        trả về 0;
}
```

Vì các tệp có xu hướng liền kề và sẽ rất khó (tốt nhất là) sắp xếp các yêu cầu I/O của chúng ta theo từng khối logic, nên việc sắp xếp dựa trên vị trí của khối logic đầu tiên của một tệp nhất định là hợp lý. Do đó, get_nr_blocks() không cần thiết và các ứng dụng của chúng ta có thể sắp xếp dựa trên giá trị trả về từ:

```
lấy_khối(fd, 0);
```

Nhược điểm của FIBMAP là nó yêu cầu khả năng CAP_SYS_RAWIO —thực chất là quyền root. Do đó, các ứng dụng không phải root không thể sử dụng cách tiếp cận này. Hơn nữa, trong khi lệnh FIBMAP được chuẩn hóa, việc triển khai thực tế của nó được giao cho các hệ thống tệp. Trong khi các hệ thống phổ biến như ext2 và ext3 hỗ trợ nó, thì một con quái vật bí truyền hơn có thể không. Lệnh gọi ioctl() sẽ trả về EINVAL nếu FIBMAP không được hỗ trợ.

Tuy nhiên, một trong những ưu điểm của cách tiếp cận này là nó trả về khối đĩa vật lý thực tế mà tệp nằm ở đó, đó chính xác là thứ bạn muốn sắp xếp. Ngay cả khi bạn sắp xếp tất cả I/O thành một tệp duy nhất dựa trên vị trí của chỉ một khối (trình lập lịch I/O của hạt nhân sắp xếp từng yêu cầu riêng lẻ theo từng khối), cách tiếp cận này gần như đạt đến thứ tự tối ưu. Tuy nhiên, yêu cầu gốc là một chút không bắt đầu đối với nhiều người.

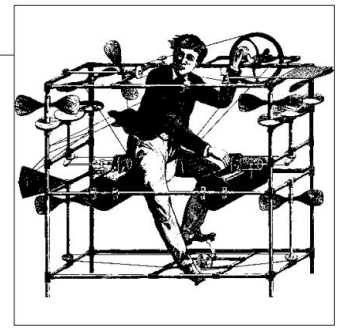
Phần kết luận

Trong ba chương trước, chúng ta đã đề cập đến mọi khía cạnh của I/O tệp trong Linux. Trong Chương 2, chúng ta đã xem xét những điều cơ bản về I/O tệp Linux—thực sự là cơ sở của lập trình Unix—với các lệnh gọi hệ thống như read( ), write( ), open( ) và close( ). Trong Chương 3, chúng ta đã thảo luận về bộ đệm không gian người dùng và việc triển khai thư viện C chuẩn. Trong chương này, chúng ta đã thảo luận về nhiều khía cạnh khác nhau của I/O nâng cao, từ các lệnh gọi hệ thống I/O mạnh hơn nhưng phức tạp hơn đến các kỹ thuật tối ưu hóa và tìm kiếm đĩa tốn kém hiệu suất.

Trong hai chương tiếp theo, chúng ta sẽ xem xét quản lý quy trình: tạo, hủy và quản lý quy trình. Tiến lên!

CHƯƠNG 5

Quản lý quy trình



Như đã đề cập trong Chương 1, các tiến trình là sự trừu tượng cơ bản nhất trong hệ thống Unix, sau các tệp. Là mã đối tượng đang thực thi—các chương trình đang hoạt động, còn sống, đang chạy—các tiến trình không chỉ là ngôn ngữ lắp ráp; chúng bao gồm dữ liệu, tài nguyên, trạng thái và máy tính ảo.

Trong chương này, chúng ta sẽ xem xét các nguyên tắc cơ bản của quy trình, từ khi tạo đến khi kết thúc. Các nguyên tắc cơ bản vẫn tương đối không thay đổi kể từ những ngày đầu của Unix. Ở đây, trong chủ đề quản lý quy trình, sự trường tồn và tư duy hướng tới tương lai của thiết kế ban đầu của Unix tỏa sáng rực rỡ nhất. Unix đã đi theo một con đường thú vị, một con đường hiếm khi được đi qua, và tách biệt hành động tạo quy trình mới khỏi hành động tải ảnh nhị phân mới. Mặc dù hai nhiệm vụ được thực hiện song song hầu hết thời gian, nhưng sự phân chia này đã cho phép rất nhiều sự tự do để thử nghiệm và phát triển cho từng nhiệm vụ. Con đường ít người đi này vẫn tồn tại cho đến ngày nay và trong khi hầu hết các hệ điều hành cung cấp một lệnh gọi hệ thống duy nhất để khởi động một chương trình mới, Unix yêu cầu hai lệnh: một fork và một exec. Nhưng trước khi chúng ta tìm hiểu về các lệnh gọi hệ thống đó, chúng ta hãy xem xét kỹ hơn về bản thân quy trình.

ID quy trình

Mỗi tiến trình được biểu diễn bằng một mã định danh duy nhất, ID tiến trình (thường được viết tắt là pid). Mã định danh pid được đảm bảo là duy nhất tại bất kỳ thời điểm nào. Nghĩa là, trong khi tại thời điểm t0 chỉ có thể có một tiến trình có pid 770 (nếu bất kỳ tiến trình nào tồn tại với giá trị như vậy), thì không có gì đảm bảo rằng tại thời điểm t1 sẽ không có một tiến trình khác có pid 770. Tuy nhiên, về cơ bản, hầu hết mã đều cho rằng hạt nhân không dễ dàng cấp lại mã định danh tiến trình—một giả định mà, như bạn sẽ thấy ngay sau đây, là khá an toàn.

Tiến trình nhàn rỗi— tiến trình mà hạt nhân “chạy” khi không có tiến trình nào khác có thể chạy—có pid 0. Tiến trình đầu tiên mà hạt nhân thực thi sau khi khởi động hệ thống, được gọi là tiến trình init, có pid 1. Thông thường, tiến trình init trên Linux là chương trình init . Chúng tôi sử dụng thuật ngữ “init” để chỉ cả tiến trình ban đầu mà hạt nhân chạy và chương trình cụ thể được sử dụng cho mục đích đó.

Trừ khi người dùng chỉ rõ cho hạt nhân biết tiến trình nào cần chạy (thông qua tham số dòng lệnh init kernel), hạt nhân phải tự xác định một tiến trình init phù hợp—một ví dụ hiếm hoi khi hạt nhân quyết định chính sách. Hạt nhân Linux thử bốn tệp thực thi, theo thứ tự sau:

1. /sbin/init: Vị trí ưu tiên và có khả năng xảy ra cao nhất cho quy trình init. 2. /etc/
init: Một vị trí có khả năng xảy ra khác cho quy trình init. 3. /bin/
init: Một vị trí có thể xảy ra cho quy trình init. 4. /bin/sh: Vị
trí của Bourne shell, mà hạt nhân sẽ cố gắng chạy nếu không tìm thấy quy trình init.

Tiến trình đầu tiên trong số các tiến trình này tồn tại được thực hiện như tiến trình init. Nếu cả bốn tiến trình đều không thực hiện được, hạt nhân Linux sẽ dừng hệ thống với trạng thái hoảng loạn.

Sau khi chuyển giao từ kernel, quy trình init xử lý phần còn lại của quy trình khởi động. Thông thường, điều này bao gồm khởi tạo hệ thống, khởi động nhiều dịch vụ khác nhau và khởi chạy chương trình đăng nhập.

Phân bổ ID quy trình

Theo mặc định, kernel áp đặt giá trị ID tiến trình tối đa là 32768. Điều này nhằm mục đích tương thích với các hệ thống Unix cũ hơn, sử dụng các loại 16 bit nhỏ hơn cho ID tiến trình. Quản trị viên hệ thống có thể đặt giá trị cao hơn thông qua /proc/sys/kernel/pid_max, đổi không gian pid lớn hơn để giảm khả năng tương thích.

Nhân phân bổ ID tiến trình cho các tiến trình theo cách tuyến tính nghiêm ngặt. Nếu pid 17 là số cao nhất hiện được phân bổ, pid 18 sẽ được phân bổ tiếp theo, ngay cả khi tiến trình được gán pid 17 lần cuối không còn chạy khi tiến trình mới bắt đầu. Nhân không tái sử dụng các giá trị ID tiến trình cho đến khi nó bao quanh từ trên cùng—tức là, các giá trị trước đó sẽ không được tái sử dụng cho đến khi giá trị trong /proc/sys/kernel/pid_max được phân bổ. Do đó, mặc dù Linux không đảm bảo tính duy nhất của ID tiến trình trong thời gian dài, nhưng hành vi phân bổ của nó cung cấp ít nhất sự thoải mái trong ngắn hạn về tính ổn định và tính duy nhất của các giá trị pid.

Phân cấp quy trình Quy trình sinh

ra một quy trình mới được gọi là parent; quy trình mới được gọi là child. Mọi quy trình đều được sinh ra từ một quy trình khác (trừ quy trình init). Do đó, mọi child đều có parent. Mối quan hệ này được ghi lại trong ID quy trình parent của mỗi quy trình (ppid), là pid của parent của child.

Mỗi tiến trình được sở hữu bởi một người dùng và một nhóm. Quyền sở hữu này được sử dụng để kiểm soát quyền truy cập vào tài nguyên. Đối với hạt nhân, người dùng và nhóm chỉ là các giá trị số nguyên.
Thông qua các tệp /etc/passwd và /etc/group, các số nguyên này được ánh xạ tới các tên có thể đọc được bằng con người mà người dùng Unix quen thuộc, chẳng hạn như người dùng root hoặc

group wheel (nói chung, hạt nhân Linux không quan tâm đến các chuỗi mà con người có thể đọc được và thích xác định các đối tượng bằng số nguyên). Mỗi tiến trình con kế thừa quyền sở hữu người dùng và nhóm của tiến trình cha.

Mỗi tiến trình cũng là một phần của một nhóm tiến trình, nhóm này chỉ đơn giản thể hiện mối quan hệ của nó với các tiến trình khác và không được nhầm lẫn với khái niệm người dùng/nhóm đã đề cập ở trên. Các tiến trình con thường thuộc về cùng một nhóm tiến trình như các tiến trình cha của chúng. Ngoài ra, khi một shell khởi động một đường ống (ví dụ: khi người dùng nhập ls | less), tất cả các lệnh trong đường ống đều đi vào cùng một nhóm tiến trình. Khái niệm về một nhóm tiến trình giúp dễ dàng gửi tín hiệu đến hoặc lấy thông tin về toàn bộ đường ống, cũng như tất cả các tiến trình con của các tiến trình trong đường ống. Theo quan điểm của người dùng, một nhóm tiến trình có liên quan chặt chẽ đến một công việc.

pid_t

Về mặt lập trình, ID tiến trình được biểu diễn bằng kiểu pid_t , được định nghĩa trong tệp tiêu đề <sys/types.h>. Kiểu C hỗ trợ chính xác là dành riêng cho kiến trúc và không được định nghĩa bởi bất kỳ tiêu chuẩn C nào. Tuy nhiên, trên Linux, pid_t thường là một typedef cho kiểu int C.

Lấy ID quy trình và ID quy trình cha

Lệnh gọi hệ thống getpid() trả về ID tiến trình của tiến trình đang gọi:

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

pid_t getpid (không có giá trị);
```

Lệnh gọi hệ thống getppid() trả về ID tiến trình của tiến trình cha đang gọi:

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

pid_t getppid (không có giá trị);
```

Không lệnh gọi nào trả về lỗi. Do đó, cách sử dụng rất đơn giản:

```
printf ("Pid của tôi=%d\n", getpid ( ));
printf ("Pid của cha mẹ=%d\n", getppid ( ));
```

Làm sao chúng ta biết được pid_t là một số nguyên có dấu? Câu hỏi hay! Câu trả lời, đơn giản là chúng ta không biết. Mặc dù chúng ta có thể yên tâm cho rằng pid_t là một int trên Linux, nhưng việc đoán như vậy vẫn làm mất đi ý định của kiểu trừu tượng và làm tổn hại đến khả năng chuyển đổi. Thật không may, giống như tất cả các typedef trong C, không có cách dễ dàng nào để in các giá trị pid_t —đây là một phần của sự trừu tượng và về mặt kỹ thuật, chúng ta cần một hàm pid_to_int() , mà chúng ta không có. Tuy nhiên, việc xử lý các giá trị này như các số nguyên, ít nhất là đối với mục đích của printf( ), là điều phổ biến.

Chạy một quy trình mới

Trong Unix, hành động tải vào bộ nhớ và thực thi một hình ảnh chương trình tách biệt với hành động tạo một tiến trình mới. Một lệnh gọi hệ thống (thực tế là một lệnh gọi từ một họ lệnh gọi) tải một chương trình nhị phân vào bộ nhớ, thay thế nội dung trước đó của không gian địa chỉ và bắt đầu thực thi chương trình mới. Điều này được gọi là thực thi một chương trình mới và chức năng được cung cấp bởi họ lệnh gọi exec .

Một lệnh gọi hệ thống khác được sử dụng để tạo một tiến trình mới, ban đầu là bản sao gần giống của tiến trình cha. Thông thường, tiến trình mới sẽ thực thi ngay một chương trình mới. Hành động tạo một tiến trình mới được gọi là forking, và chức năng này được cung cấp bởi lệnh gọi hệ thống fork() . Hai hành động—đầu tiên là fork, để tạo một tiến trình mới, và sau đó là exec, để tải một hình ảnh mới vào tiến trình đó—do đó cần có để thực thi một hình ảnh chương trình mới trong một tiến trình mới. Chúng ta sẽ đề cập đến các lệnh gọi exec trước, sau đó là fork( ).

Họ lệnh gọi Exec Không có hàm

exec đơn lẻ; thay vào đó, có một họ các hàm exec được xây dựng trên một lệnh gọi hệ thống đơn lẻ. Trước tiên, hãy xem lệnh gọi đơn giản nhất trong số các lệnh gọi này, execl( ):

```
#include <unistd.h>

int execl (const char *path,
           const char
           *arg, ...);
```

Một lệnh gọi đến execl( ) thay thế hình ảnh quy trình hiện tại bằng một hình ảnh mới bằng cách tải vào bộ nhớ chương trình được trỏ đến bởi path. Tham số arg là đối số đầu tiên của chương trình này. Dấu ba chấm biểu thị số lượng đối số thay đổi— hàm execl( ) là variadic, nghĩa là các đối số bổ sung có thể tùy ý theo sau, từng cái một. Danh sách các đối số phải được kết thúc bằng NULL.

Ví dụ, đoạn mã sau đây thay thế chương trình đang thực thi bằng /bin/vi:

```
int ret;

ret = execl ("/bin/vi", "vi", NULL);
nếu (ret ==
        -1) lỗi ("execl");
```

Lưu ý rằng chúng ta tuân theo quy ước Unix và truyền “vi” làm đối số đầu tiên của chương trình. Shell đặt thành phần cuối cùng của đường dẫn, “vi,” vào đối số đầu tiên khi nó phân nhánh/thực thi các tiến trình, do đó, một chương trình có thể kiểm tra đối số đầu tiên của nó, argv[0], để khám phá tên của ảnh nhị phân của nó. Trong nhiều trường hợp, một số tiện ích hệ thống xuất hiện dưới dạng các tên khác nhau đối với người dùng thực chất là một chương trình duy nhất có liên kết cứng cho nhiều tên của chúng. Chương trình sử dụng đối số đầu tiên để xác định hành vi của nó.

Một ví dụ khác, nếu bạn muốn chỉnh sửa tệp /home/kidd/hooks.txt, bạn có thể thực thi đoạn mã sau:

```
int ret;

ret = execl ("/bin/vi", "vi", "/home/kidd/hooks.txt", NULL); nếu
(ret == -1)
        lỗi ("execl");
```

Thông thường, execl( ) không trả về. Một lệnh gọi thành công kết thúc bằng cách nhảy đến điểm vào của chương trình mới và mã vừa được thực thi không còn tồn tại trong không gian địa chỉ của tiến trình. Tuy nhiên, khi xảy ra lỗi, execl( ) trả về -1 và đặt errno để chỉ ra vấn đề. Chúng ta sẽ xem xét các giá trị errno có thể có sau trong phần này.

Một lệnh gọi exec1( ) thành công không chỉ thay đổi không gian địa chỉ và hình ảnh tiến trình mà còn thay đổi một số thuộc tính khác của tiến trình:

- Bất kỳ tín hiệu nào đang chờ xử lý đều bị mất.
- Bất kỳ tín hiệu nào mà quy trình đang bắt được (xem Chương 9) đều được trả về hành vi mặc định của chúng vì trình xử lý tín hiệu không còn tồn tại trong không gian địa chỉ của quy trình.
- Bất kỳ khóa bộ nhớ nào (xem Chương 8) đều bị loại bỏ.
- Hầu hết các thuộc tính luồng đều được trả về giá trị mặc định.
- Hầu hết các số liệu thống kê quy trình được đặt lại.
- Mọi thứ liên quan đến bộ nhớ của quy trình, bao gồm bất kỳ tệp nào được ánh xạ, đều được bị rơi.
- Bất kỳ thứ gì chỉ tồn tại trong không gian người dùng, bao gồm các tính năng của thư viện C, chẳng hạn như hành vi atexit() , đều bị loại bỏ.

Tuy nhiên, nhiều thuộc tính của quy trình không thay đổi. Ví dụ, pid, pid cha, mức độ ưu tiên và người dùng và nhóm sở hữu đều giữ nguyên.

Thông thường, các tệp mở được kế thừa qua một exec. Điều này có nghĩa là chương trình mới được thực thi có toàn quyền truy cập vào tất cả các tệp mở trong quy trình gốc, giả sử nó biết các giá trị mô tả tệp. Tuy nhiên, đây thường không phải là hành vi mong muốn. Thực hành thông thường là đóng tệp trước exec, mặc dù cũng có thể hướng dẫn kernel thực hiện việc này tự động thông qua fcntl( ).

Phần còn lại của gia đình

Ngoài execl( ), còn có năm thành viên khác của họ exec:

```
#include <unistd.h>

int execlp (const char *file,
            const char
            *arg, ...);
```

```
int execle (const char *path,
            const char *arg,
            ...,
            char * const envp[]);

int execv (const char *path, char *const argv[]);

int execvp (const char *file, char *const argv[]);

int execve (const char *tên tệp,
            char *const argv[],
            char *const envp[]);
```

Các ký hiệu ghi nhớ rất đơn giản. l và v phân định xem các đối số được cung cấp thông qua danh sách hay mảng (vectơ). p biểu thị rằng đường dẫn đầy đủ của người dùng được tìm kiếm cho tệp đã cho. Các lệnh sử dụng các biến thể p có thể chỉ định tên tệp, miễn là nó nằm trong đường dẫn của người dùng. Cuối cùng, e lưu ý rằng một môi trường mới cũng được cung cấp cho quy trình mới. Thật kỳ lạ, mặc dù không có lý do kỹ thuật nào cho việc bỏ sót, nhưng họ exec không chứa thành viên nào vừa tìm kiếm đường dẫn vừa lấy một môi trường mới. Điều này có thể là do các biến thể p được triển khai để sử dụng bởi các shell và các quy trình được thực thi bằng shell thường kế thừa môi trường của chúng từ shell.

Các thành viên của họ exec chấp nhận một mảng hoạt động tương tự, ngoại trừ việc một mảng được xây dựng và truyền vào thay vì một danh sách. Việc sử dụng một mảng cho phép các đối số được xác định tại thời điểm chạy. Giống như danh sách đối số variadic, mảng phải được kết thúc bằng NULL.

Đoạn mã sau sử dụng execvp() để thực thi vi, như chúng ta đã làm trước đó:

```
const char *args[] = { "vi", "/home/kidd/hooks.txt", NULL };
int ret;

ret = execvp ("vi", args);
if (ret == -1)
        perror ("execvp");
```

Giả sử /bin nằm trong đường dẫn của người dùng, thì hoạt động này tương tự như ví dụ cuối cùng.

Trong Linux, chỉ có một thành viên của họ exec là lệnh gọi hệ thống. Phần còn lại là các wrapper trong thư viện C xung quanh lệnh gọi hệ thống. Vì các lệnh gọi hệ thống variadic sẽ khó triển khai, tốt nhất là như vậy, và vì khái niệm về đường dẫn của người dùng chỉ tồn tại trong không gian người dùng, nên tùy chọn duy nhất cho lệnh gọi hệ thống đơn lẻ là execve( ). Nguyên mẫu lệnh gọi hệ thống giống hệt với lệnh gọi người dùng.

Giá trị lỗi

Khi thành công, các lệnh gọi hệ thống exec không trả về. Khi thất bại, các lệnh gọi trả về -1 và đặt errno thành một trong các giá trị sau:

E2BIG

Tổng số byte trong danh sách đối số được cung cấp (arg) hoặc môi trường (envp) quá lớn.

EACCESS

Tiến trình thiếu quyền tìm kiếm cho một thành phần trong đường dẫn; đường dẫn không phải là tệp thông thường; tệp đích không được đánh dấu là có thể thực thi; hoặc hệ thống tệp mà đường dẫn hoặc tệp lưu trú được gắn kết noexec.

EFAULT

Một con trỏ đã cho là không hợp lệ.

EIO

Đã xảy ra lỗi I/O cấp thấp (lỗi này không tốt).

EISDIR

Thành phần cuối cùng trong đường dẫn, hay trình thông dịch, là một thư mục.

ELOOP

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

EMFILE

Quá trình gọi đã đạt đến giới hạn đối với các tệp đang mở.

ENFILE

Đã đạt đến giới hạn toàn hệ thống về số lượng tệp đang mở.

ENOENT

Mục tiêu của đường dẫn hoặc tệp không tồn tại hoặc thư viện chia sẻ cần thiết không tồn tại.

ENOEXEC

Mục tiêu của đường dẫn hoặc tệp là tệp nhị phân không hợp lệ hoặc dành cho kiến trúc máy khác.

ENOMEM

Không có đủ bộ nhớ hạt nhân để thực thi chương trình mới.

ENOTDIR Một

thành phần không phải cuối cùng trong đường dẫn không phải là một thư mục.

EPERM

Hệ thống tập tin mà đường dẫn hoặc tệp lưu trú được gắn kết nosuid, người dùng không phải là root và đường dẫn hoặc tệp có bit suid hoặc sgid được thiết lập.

ETXTBSY

Mục tiêu của đường dẫn hoặc tệp được mở để ghi bởi một tiến trình khác.

Hệ thống gọi fork( )

Một tiến trình mới chạy cùng một hình ảnh như tiến trình hiện tại có thể được tạo thông qua lệnh gọi hệ thống fork() :

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

pid_t fork (không có giá trị);
```

Một lệnh gọi thành công đến fork() sẽ tạo ra một tiến trình mới, giống hệt về hầu hết mọi mặt với tiến trình gọi. Cả hai tiến trình đều tiếp tục chạy, trả về từ fork() như thể không có gì đặc biệt xảy ra.

Tiến trình mới được gọi là "con" của tiến trình gốc, tiến trình gốc này được gọi là "cha". Trong tiến trình con, một lệnh gọi fork() thành công trả về 0. Trong tiến trình cha, fork() trả về pid của tiến trình con. Tiến trình con và tiến trình cha giống hệt nhau ở hầu hết mọi khía cạnh, ngoại trừ một vài điểm khác biệt cần thiết:

- PID của đứa trẻ, tất nhiên, được phân bổ mới và khác với pid của cha mẹ.
- PID cha của tiến trình con được đặt thành pid của tiến trình cha. •

Thống kê tài nguyên được đặt lại về 0 trong tiến trình con.

- Bất kỳ tín hiệu đang chờ xử lý nào cũng sẽ bị xóa và không được thừa hưởng bởi tệp con (xem Chương 9).
- Bất kỳ khóa tệp nào đã có được đều không được thừa hưởng bởi tệp con.

Khi xảy ra lỗi, tiến trình con không được tạo, fork() trả về -1 và errno được thiết lập phù hợp. Có hai giá trị errno có thể có , với ba ý nghĩa có thể có:

LẠI LẦN NỮA

Hạt nhân không phân bổ được một số tài nguyên nhất định, chẳng hạn như pid mới hoặc đã đạt đến giới hạn tài nguyên RLIMIT_NPROC (rlimit) (xem Chương 6).

ENOMEM

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

Cách sử dụng rất đơn giản:

```
pid_t pid;

pid = fork ( );
nếu (pid > 0)
         printf ("Tôi là cha của pid=%d!\n", pid); nếu không
thì nếu (!pid)
         printf ("Tôi là em bé!\n"); nếu
không thì nếu (pid ==
         -1) perror ("fork");
```

Cách sử dụng phổ biến nhất của fork( ) là tạo một tiến trình mới trong đó một hình ảnh nhị phân mới sau đó được tải—hãy nghĩ đến một shell chạy một chương trình mới cho người dùng hoặc một tiến trình tạo ra một chương trình trợ giúp. Đầu tiên, tiến trình tạo ra một tiến trình mới, sau đó tiến trình con thực thi một hình ảnh nhị phân mới. Sự kết hợp "fork cộng với exec" này thường xuyên và đơn giản. Ví dụ sau tạo ra một tiến trình mới chạy nhị phân /bin/windlass:

```
pid_t pid;

pid = fork ( );
nếu (pid == -1)
         perror ("fork");

/* đứa trẻ      ... */
```

```
    nếu (!pid)
            { const char *args[] = { "tời", NULL }; int ret;


            ret = execv ("/bin/windlass", args);
            nếu (ret == -1)
                    { lỗi ("execv");
                    thoát (EXIT_FAILURE);
            }
    }
```

Tiến trình cha tiếp tục chạy mà không có thay đổi nào, ngoại trừ việc nó hiện có một tiến trình con mới. Lệnh gọi đến execv( ) thay đổi tiến trình con để chạy chương trình /bin/windlass .

Copy-on-write

Trong các hệ thống Unix ban đầu, forking rất đơn giản, nếu không muốn nói là ngây thơ. Khi được gọi, kernel tạo các bản sao của tất cả các cấu trúc dữ liệu bên trong, sao chép các mục bảng trang của tiến trình, sau đó thực hiện sao chép từng trang một của không gian địa chỉ của tiến trình cha vào không gian địa chỉ mới của tiến trình con. Nhưng bản sao từng trang này, ít nhất là theo quan điểm của kernel, tốn thời gian.

Các hệ thống Unix hiện đại hoạt động tối ưu hơn. Thay vì sao chép toàn bộ không gian địa chỉ của máy cha, các hệ thống Unix hiện đại như Linux sử dụng các trang sao chép khi ghi (COW).

Copy-on-write là một chiến lược tối ưu hóa lười biếng được thiết kế để giảm thiểu chi phí chung của việc sao chép tài nguyên. Tiền đề rất đơn giản: nếu nhiều người dùng yêu cầu quyền truy cập đọc vào bản sao tài nguyên của riêng họ, thì không cần phải tạo các bản sao trùng lặp của tài nguyên. Thay vào đó, mỗi người dùng có thể được trao một con trỏ đến cùng một tài nguyên. Miễn là không có người dùng nào cố gắng sửa đổi "bản sao" tài nguyên của mình, thì ảo tưởng về quyền truy cập độc quyền vào tài nguyên vẫn còn và chi phí chung của một bản sao được tránh. Nếu một người dùng cố gắng sửa đổi bản sao tài nguyên của mình, tại thời điểm đó, tài nguyên được sao chép một cách minh bạch và bản sao được cung cấp cho người dùng đang sửa đổi. Người dùng, không bao giờ khôn ngoan hơn, sau đó có thể sửa đổi bản sao tài nguyên của mình trong khi những người dùng khác tiếp tục chia sẻ phiên bản gốc, không thay đổi. Do đó có tên: bản sao chỉ xảy ra khi ghi.

Lợi ích chính là nếu người dùng không bao giờ sửa đổi bản sao tài nguyên của mình, thì không bao giờ cần bản sao. Ưu điểm chung của thuật toán lười biếng—là chúng trì hoãn các hành động tốn kém cho đến thời điểm cuối cùng có thể—cũng được áp dụng.

Trong ví dụ cụ thể về bộ nhớ ảo, copy-on-write được triển khai trên cơ sở từng trang. Do đó, miễn là một tiến trình không sửa đổi toàn bộ không gian địa chỉ của nó, thì không cần phải sao chép toàn bộ không gian địa chỉ. Khi hoàn tất một nhánh, tiến trình cha và tiến trình con tin rằng mỗi tiến trình có một không gian địa chỉ duy nhất, trong khi thực tế chúng đang chia sẻ các trang gốc của tiến trình cha—lần lượt có thể được chia sẻ với các tiến trình cha hoặc con khác, v.v.!

Việc triển khai hạt nhân rất đơn giản. Các trang được đánh dấu là chỉ đọc và là sao chép khi ghi trong các cấu trúc dữ liệu liên quan đến trang của hạt nhân. Nếu một trong hai quy trình cố gắng sửa đổi một trang, lỗi trang sẽ xảy ra. Sau đó, hạt nhân xử lý lỗi trang bằng cách tạo một bản sao của trang một cách minh bạch; tại thời điểm này, thuộc tính sao chép khi ghi của trang sẽ bị xóa và không còn được chia sẻ nữa.

Vì kiến trúc máy hiện đại cung cấp hỗ trợ cấp phần cứng cho tính năng sao chép khi ghi trong các đơn vị quản lý bộ nhớ (MMU) nên trò chơi này rất đơn giản và dễ thực hiện.

Copy-on-write còn có lợi ích lớn hơn trong trường hợp forking. Bởi vì một tỷ lệ phần trăm lớn các fork được theo sau bởi một exec, việc sao chép không gian địa chỉ của parent vào không gian địa chỉ của child thường là một sự lãng phí thời gian hoàn toàn: nếu child thực hiện một hình ảnh nhị phân mới, không gian địa chỉ trước đó của nó sẽ bị xóa. Copy-on-write tối ưu hóa cho trường hợp này.

vfork( )

Trước khi xuất hiện các trang sao chép khi ghi, các nhà thiết kế Unix quan tâm đến việc sao chép không gian địa chỉ lãng phí trong quá trình phân nhánh được theo sau ngay bởi một lệnh exec.
Do đó, các nhà phát triển BSD đã công bố lệnh gọi hệ thống vfork() trong 3.0BSD:

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


pid_t vfork (không có giá trị);
```

Một lệnh gọi thành công vfork() có cùng hành vi như fork( ), ngoại trừ việc tiến trình con phải ngay lập tức thực hiện một lệnh gọi thành công đến một trong các hàm exec hoặc thoát bằng cách gọi _exit() (được thảo luận trong phần tiếp theo). Lệnh gọi hệ thống vfork() tránh sao chép không gian địa chỉ và bảng trang bằng cách tạm dừng tiến trình cha cho đến khi tiến trình con kết thúc hoặc thực thi một ảnh nhị phân mới. Trong thời gian tạm thời, tiến trình cha và tiến trình con chia sẻ—mà không có ngữ nghĩa sao chép khi ghi—không gian địa chỉ và các mục nhập bảng trang của chúng. Trên thực tế, công việc duy nhất được thực hiện trong vfork() là sao chép các cấu trúc dữ liệu hạt nhân bên trong. Do đó, tiến trình con không được sửa đổi bất kỳ bộ nhớ nào trong không gian địa chỉ.

Lệnh gọi hệ thống vfork() là một di tích và không bao giờ nên được triển khai trên Linux, mặc dù cần lưu ý rằng ngay cả với lệnh copy-on-write, vfork() vẫn nhanh hơn fork() vì các mục trong bảng trang không cần phải được sao chép.* Tuy nhiên, sự ra đời của lệnh copy-on-write làm suy yếu mọi lập luận cho một giải pháp thay thế cho fork( ).
Trên thực tế, cho đến phiên bản Linux 2.2.0, vfork() chỉ đơn giản là một lớp bao bọc xung quanh fork( ). Vì các yêu cầu đối với vfork() yếu hơn các yêu cầu đối với fork( ) , nên việc triển khai vfork() như vậy là khả thi.

* Mặc dù hiện không phải là một phần của nhân Linux 2.6, một bản vá triển khai các mục bảng trang chia sẻ sao chép khi ghi đã được đưa lên Danh sách gửi thư nhân Linux (lkml). Nếu được hợp nhất, sẽ hoàn toàn không có lợi ích gì khi sử dụng vfork( ).

Nói một cách chính xác, không có triển khai vfork() nào là không có lỗi: hãy xem xét tình huống nếu lệnh gọi exec bị lỗi! Cha mẹ sẽ bị tạm dừng vô thời hạn trong khi con tìm ra cách thực hiện hoặc cho đến khi thoát.

Kết thúc một quá trình

POSIX và C89 đều định nghĩa một hàm chuẩn để kết thúc quy trình hiện tại:

```
#include <stdlib.h>

void thoát (trạng thái int);
```

Một lệnh gọi exit() thực hiện một số bước tắt máy cơ bản, sau đó hướng dẫn kernel kết thúc tiến trình. Hàm này không có cách nào trả về lỗi—thực tế là nó không bao giờ trả về lỗi. Do đó, không có bất kỳ lệnh nào theo sau lệnh gọi exit() .

Tham số trạng thái được sử dụng để biểu thị trạng thái thoát của quy trình. Các chương trình khác—cũng như người dùng tại shell—có thể kiểm tra giá trị này. Cụ thể, trạng thái & 0377 được trả về cho chương trình cha. Chúng ta sẽ xem xét cách lấy giá trị trả về sau trong chương này.

EXIT_SUCCESS và EXIT_FAILURE được định nghĩa là các cách di động để biểu diễn thành công và thất bại. Trên Linux, 0 thường biểu diễn thành công; giá trị khác không, chẳng hạn như 1 hoặc -1, tương ứng với thất bại.

Do đó, một lối thoát thành công chỉ đơn giản như thế này:

```
thoát (EXIT_SUCCESS);
```

Trước khi kết thúc tiến trình, thư viện C thực hiện các bước tắt máy sau theo thứ tự:

1. Gọi bất kỳ hàm nào được đăng ký với atexit() hoặc on_exit( ) theo thứ tự ngược lại với thứ tự đăng ký của chúng. (Chúng ta sẽ thảo luận về các hàm này sau trong chương này.)
2. Xóa sạch tất cả các luồng I/O chuẩn đang mở (xem Chương 3).
3. Xóa mọi tệp tạm thời được tạo bằng hàm tmpfile() .

Các bước này hoàn tất mọi công việc mà quy trình cần thực hiện trong không gian người dùng, do đó exit() gọi lệnh gọi hệ thống _exit() để cho phép nhân xử lý phần còn lại của quy trình kết thúc:

```
#include <unistd.h>

void _exit(trạng thái int);
```

Khi một tiến trình thoát, hạt nhân sẽ dọn sạch tất cả các tài nguyên mà nó tạo ra thay mặt cho tiến trình đó mà không còn được sử dụng nữa. Điều này bao gồm, nhưng không giới hạn ở, bộ nhớ được phân bổ, các tệp đang mở và các semaphore của Hệ thống V. Sau khi dọn dẹp, hạt nhân sẽ hủy tiến trình và thông báo cho tiến trình cha về sự kết thúc của tiến trình con.

Các ứng dụng có thể gọi _exit() trực tiếp, nhưng động thái như vậy hiếm khi có ý nghĩa: hầu hết các ứng dụng cần thực hiện một số công việc dọn dẹp do lệnh thoát hoàn toàn cung cấp, chẳng hạn như xóa luồng stdout . Tuy nhiên, lưu ý rằng người dùng vfork() nên gọi _exit( ) chứ không phải exit( ) sau một fork.

Trong một bước đột phá tuyệt vời của sự dư thừa, tiêu chuẩn ISO C99 đã bổ sung Hàm _Exit() có hành vi giống hệt như hàm _exit():

```
#include <stdlib.h>

void _Exit(trạng thái int);
```

Các cách khác để kết thúc Cách cổ

điển để kết thúc một chương trình không phải là thông qua một lệnh gọi hệ thống rõ ràng, mà chỉ đơn giản là "rơi khỏi phần cuối" của chương trình. Trong trường hợp của C, điều này xảy ra khi hàm main() trả về. Tuy nhiên, cách tiếp cận "rơi khỏi phần cuối" vẫn gọi một lệnh gọi hệ thống: trình biên dịch chỉ cần chèn một _exit() ngầm định sau mã tắt máy của chính nó. Thực hành mã hóa tốt là trả về trạng thái thoát một cách rõ ràng, thông qua exit( ) hoặc bằng cách trả về giá trị từ main( ). Shell sử dụng giá trị thoát để đánh giá sự thành công hay thất bại của các lệnh. Lưu ý rằng một lệnh trả về thành công là exit(0) hoặc lệnh trả về từ main() là 0.

Một tiến trình cũng có thể kết thúc nếu nó được gửi một tín hiệu có hành động mặc định là kết thúc tiến trình. Các tín hiệu như vậy bao gồm SIGTERM và SIGKILL (xem Chương 9).

Một cách cuối cùng để kết thúc quá trình thực thi của một chương trình là bằng cách gây ra cơn thịnh nộ của hạt nhân. Hạt nhân có thể giết một quy trình vì thực thi một lệnh bất hợp pháp, gây ra vi phạm phân đoạn, hết bộ nhớ, v.v.

đã thoát( )

POSIX 1003.1-2001 định nghĩa và Linux triển khai lệnh gọi thư viện atexit() , được sử dụng để đăng ký các hàm sẽ được gọi khi kết thúc quy trình:

```
#include <stdlib.h>

int atexit(void (*hàm)(void));
```

Việc gọi thành công atexit() sẽ đăng ký hàm đã cho để chạy trong quá trình kết thúc quy trình bình thường; tức là khi một quy trình bị kết thúc thông qua exit() hoặc return from main( ). Nếu một quy trình gọi hàm exec, danh sách các hàm đã đăng ký sẽ bị xóa (vì các hàm không còn tồn tại trong không gian địa chỉ của quy trình mới). Nếu một quy trình kết thúc thông qua tín hiệu, các hàm đã đăng ký sẽ không được gọi.

Hàm được đưa ra không có tham số và không trả về giá trị. Nguyên mẫu có dạng:

```
void hàm_của_tôi (void);
```

Các hàm được gọi theo thứ tự ngược lại với thứ tự chúng được đăng ký. Nghĩa là, các hàm được lưu trữ trong một ngăn xếp và hàm vào cuối cùng sẽ là hàm ra đầu tiên (LIFO). Các hàm đã đăng ký không được gọi exit( ), nếu không chúng sẽ bắt đầu một quá trình đệ quy vô tận. Nếu một hàm cần kết thúc quá trình chấm dứt sớm, thì nó phải gọi _exit( ). Tuy nhiên, hành vi như vậy không được khuyến khích vì một hàm có thể quan trọng có thể không chạy.

Tiêu chuẩn POSIX yêu cầu atexit() hỗ trợ ít nhất ATEXIT_MAX các hàm đã đăng ký và giá trị này phải ít nhất là 32. Giá trị tối đa chính xác có thể được lấy thông qua sysconf() và giá trị _SC_ATEXIT_MAX:

```
dài atexit_max;

atexit_max = sysconf(_SC_ATEXIT_MAX); printf
("atexit_max=%ld\n", atexit_max);
```

Nếu thành công, atexit() trả về 0. Nếu có lỗi, nó trả về -1.

Đây là một ví dụ đơn giản:

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

vô hiệu hóa (void)
{
          printf ("atexit( ) đã thành công!\n");
}

int main (void) {

          nếu (atexit (ra))
                    fprintf(stderr, "atexit( ) không thành công!\n");

          trả về 0;
}
```

khi thoát ( )

SunOS 4 đã định nghĩa hàm tương đương của riêng nó với atexit( ) và glibc của Linux hỗ trợ hàm này:

```
#include <stdlib.h>

int khi thoát (void (*hàm)(int, void *), void *arg);
```

Hàm này hoạt động giống như atexit( ), nhưng nguyên mẫu của hàm đã đăng ký thì khác:

```
void my_function (int status, void *arg);
```

Đối số status là giá trị được truyền cho exit() hoặc được trả về từ main( ). Đối số arg là tham số thứ hai được truyền cho on_exit( ). Cần phải cẩn thận để đảm bảo rằng bộ nhớ được trỏ tới bởi arg là hợp lệ khi hàm cuối cùng được gọi.

Phiên bản mới nhất của Solaris không còn hỗ trợ chức năng này nữa. Bạn nên sử dụng atexit() tuân thủ tiêu chuẩn thay thế.

SIGCHLD

Khi một tiến trình kết thúc, hạt nhân gửi tín hiệu SIGCHLD đến tiến trình cha. Theo mặc định, tín hiệu này bị bỏ qua và tiến trình cha không thực hiện hành động nào. Tuy nhiên, các tiến trình có thể chọn xử lý tín hiệu này thông qua các lệnh gọi hệ thống signal() hoặc sigaction() .
Những cuộc gọi này và phần còn lại của thế giới tín hiệu tuyệt vời này sẽ được đề cập trong Chương 9.

Tín hiệu SIGCHLD có thể được tạo và gửi bất kỳ lúc nào, vì việc kết thúc của child là không đồng bộ so với parent của nó. Nhưng thường thì parent muốn tìm hiểu thêm về việc kết thúc của child, hoặc thậm chí chờ sự kiện xảy ra một cách rõ ràng. Điều này có thể thực hiện được với các lệnh gọi hệ thống được thảo luận tiếp theo.

Đang chờ các tiến trình con đã kết thúc

Việc nhận thông báo qua tín hiệu rất hữu ích, nhưng nhiều bậc phụ huynh muốn có thêm thông tin khi một trong các tiến trình con của họ kết thúc—ví dụ: giá trị trả về của tiến trình con.

Nếu một tiến trình con hoàn toàn biến mất khi bị chấm dứt, như người ta có thể mong đợi, sẽ không còn dấu vết nào để tiến trình cha điều tra. Do đó, những nhà thiết kế ban đầu của Unix đã quyết định rằng khi một tiến trình con chết trước tiến trình cha của nó, thì hạt nhân sẽ đưa tiến trình con vào trạng thái tiến trình đặc biệt. Một tiến trình ở trạng thái này được gọi là zom-bie. Chỉ một bộ khung tối thiểu của những gì từng là tiến trình—một số cấu trúc dữ liệu hạt nhân cơ bản chứa dữ liệu có khả năng hữu ích—được giữ lại. Một tiến trình ở trạng thái này chờ tiến trình cha của nó hỏi về trạng thái của nó (một quy trình được gọi là chờ tiến trình zombie). Chỉ sau khi tiến trình cha có được thông tin được lưu giữ về tiến trình con bị chấm dứt thì tiến trình đó mới chính thức thoát và không còn tồn tại nữa ngay cả khi là một zombie.

Nhân Linux cung cấp một số giao diện để lấy thông tin về các con đã kết thúc. Giao diện đơn giản nhất như vậy, được định nghĩa bởi POSIX, là wait( ):

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

pid_t chờ (int *status);
```

Một lệnh gọi wait() trả về pid của một child đã kết thúc, hoặc -1 nếu có lỗi. Nếu không có child nào kết thúc, lệnh gọi sẽ chặn cho đến khi một child kết thúc. Nếu một child đã kết thúc, lệnh gọi sẽ trả về ngay lập tức. Do đó, lệnh gọi wait() để phản hồi tin tức về sự kết thúc của child—ví dụ, khi nhận được SIGCHLD —sẽ luôn trả về mà không chặn.

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

TRẺ EM
: Quá trình gọi không có tiến trình con nào.

TRỰC TIẾP
: Trong lúc chờ đợi, tôi nhận được tín hiệu và cuộc gọi đã được trả lời sớm.

Nếu không phải NULL, con trỏ trạng thái sẽ chứa thông tin bổ sung về phần tử con. Vì POSIX cho phép các triển khai xác định các bit trong trạng thái theo cách chúng thấy phù hợp, nên tiêu chuẩn cung cấp một họ các macro để diễn giải tham số:

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

int WIFEXITED (trạng thái);
int WIFSIGNALED (trạng thái);
int WIFSTOPPED (trạng thái);
int WIFCONTINUED (trạng thái);

int WEXITSTATUS (trạng thái);
int WTERMSIG (trạng thái);
int WSTOPSIG (trạng thái);
int WCOREDUMP (trạng thái);
```

Bất kỳ macro nào trong hai macro đầu tiên đều có thể trả về true (giá trị khác không), tùy thuộc vào cách quy trình kết thúc. Macro đầu tiên, WIFEXITED, trả về true nếu quy trình kết thúc bình thường—tức là nếu quy trình gọi _exit( ). Trong trường hợp này, macro WEXITSTATUS cung cấp tám bit bậc thấp đã được truyền cho _exit( ).

WIFSIGNALED trả về true nếu một tín hiệu gây ra sự kết thúc của tiến trình (xem Chương 9 để biết thêm thảo luận về các tín hiệu). Trong trường hợp này, WTERMSIG trả về số tín hiệu gây ra sự kết thúc và WCOREDUMP trả về true nếu tiến trình dump lõi để phản hồi lại việc nhận được tín hiệu. WCOREDUMP không được POSIX định nghĩa, mặc dù nhiều hệ thống Unix, bao gồm cả Linux, hỗ trợ WCOREDUMP.

WIFSTOPPED và WIFCONTINUED trả về true nếu quy trình đã dừng hoặc tiếp tục, tương ứng, và hiện đang được theo dõi thông qua lệnh gọi hệ thống ptrace() . Các điều kiện này thường chỉ áp dụng khi triển khai trình gỡ lỗi, mặc dù khi được sử dụng với waitpid() (xem tiểu mục sau), chúng cũng được sử dụng để triển khai kiểm soát công việc. Thông thường, wait() chỉ được sử dụng để truyền đạt thông tin về việc chấm dứt quy trình. Nếu WIFSTOPPED là true, WSTOPSIG cung cấp số tín hiệu đã dừng quy trình. WIFCONTINUED không được POSIX định nghĩa, mặc dù các tiêu chuẩn trong tương lai định nghĩa nó cho waitpid( ). Kể từ hạt nhân Linux 2.6.10, Linux cũng cung cấp macro này cho wait( ).

Hãy cùng xem một chương trình ví dụ sử dụng wait() để tìm hiểu điều gì đã xảy ra với chương trình con của nó:

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

int main (void) {

        int trạng
        thái; pid_t pid;

        nếu (!fork ())
                trả về 1;
```

```
        pid = chờ (&trạng thái);
        nếu (pid == -1)
                lỗi ("chờ");

        printf ("pid=%d\n", pid);

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

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

        nếu (WIFSTOPPED (trạng thái))
                printf ("Đã dừng bởi tín hiệu=%d\n",
                        WSTOPSIG (trạng thái));

        nếu (WIFCONTINUED (trạng thái))
                printf ("Tiếp tục\n");

       trả về 0;
}
```

Chương trình này phân nhánh một child, sau đó thoát ngay lập tức. Tiến trình cha sau đó thực hiện lệnh gọi hệ thống wait() để xác định trạng thái của child. Tiến trình in ra pid của child và cách nó chết. Bởi vì trong trường hợp này child bị chấm dứt bằng cách trả về từ main( ), chúng ta biết rằng chúng ta sẽ thấy đầu ra tương tự như sau:

```
$ ./wait
pid=8529
Kết thúc bình thường với trạng thái thoát=1
```

Nếu thay vì để đứa trẻ trả về, chúng ta yêu cầu nó gọi abort( ),* tự gửi cho nó Tín hiệu SIGABRT , thay vào đó chúng ta sẽ thấy nội dung tương tự như sau:

```
$ ./wait
pid=8678
Bị giết bởi tín hiệu=6
```

Chờ một quy trình cụ thể Việc quan sát

hành vi của các quy trình con là rất quan trọng. Tuy nhiên, thông thường, một quy trình có nhiều con và không muốn chờ tất cả chúng mà chỉ muốn chờ một quy trình con cụ thể. Một giải pháp là thực hiện nhiều lần gọi wait( ), mỗi lần ghi chú giá trị trả về. Tuy nhiên, điều này khá phức tạp—nếu sau này bạn muốn kiểm tra trạng thái của một quy trình đã kết thúc khác thì sao? Quy trình cha sẽ phải lưu tất cả đầu ra của wait() , trong trường hợp cần sau này.

* Được định nghĩa trong tiêu đề <stdlib.h>.

Nếu bạn biết pid của tiến trình mà bạn muốn chờ, bạn có thể sử dụng lệnh gọi hệ thống waitpid() :

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

pid_t waitpid (pid_t pid, int *status, int tùy chọn);
```

Lệnh gọi waitpid() là phiên bản mạnh mẽ hơn của wait( ). Các tham số bổ sung của nó cho phép tinh chỉnh.

Tham số pid chỉ định chính xác tiến trình hoặc các tiến trình nào cần chờ. Giá trị của nó chia thành bốn nhóm:

< -1

Đợi bất kỳ tiến trình con nào có ID nhóm tiến trình bằng với giá trị tuyệt đối của giá trị này. Ví dụ, truyền -500 sẽ đợi bất kỳ tiến trình nào trong nhóm tiến trình 500.

-1

Chờ bất kỳ tiến trình con nào. Đây là hành vi tương tự như wait( ).

0

Chờ bất kỳ tiến trình con nào thuộc cùng nhóm tiến trình với tiến trình gọi.

> 0

Chờ bất kỳ tiến trình con nào có pid chính xác là giá trị được cung cấp. Ví dụ, truyền 500 sẽ chờ tiến trình con có pid là 500.

Tham số trạng thái hoạt động giống hệt như tham số duy nhất của wait( ) và có thể được vận hành bằng các macro đã thảo luận trước đó.

Tham số options là một OR nhị phân của không hoặc nhiều tùy chọn sau:

WNOHANG

Không chặn, nhưng phải trả về ngay lập tức nếu chưa có tiến trình con phù hợp nào kết thúc (hoặc dừng hoặc tiếp tục).

Đã theo dõi

Nếu được đặt, WIFSTOPPED được đặt, ngay cả khi quy trình gọi không theo dõi quy trình con. Cờ này cho phép triển khai kiểm soát công việc chung hơn, như trong shell.

WTIẾP TỤC

Nếu được đặt, WIFCONTINUED được đặt ngay cả khi quy trình gọi không theo dõi quy trình con. Cũng như WUNTRACED, cờ này hữu ích để triển khai shell.

Khi thành công, waitpid() trả về pid của tiến trình có trạng thái đã thay đổi. Nếu WNOHANG được chỉ định và tiến trình con hoặc các tiến trình con được chỉ định vẫn chưa thay đổi trạng thái, waitpid() trả về 0. Khi có lỗi, lệnh gọi trả về -1 và errno được đặt thành một trong ba giá trị sau: