Hướng dẫn Giao tiếp Liên tiến trình của Beej

Brian “Beej Jorgensen” Hall

v1.5.5, Copyright © April 18, 2026

Contents

Chapter 1

Giới thiệu

Bạn biết cái gì dễ không? `fork()` dễ lắm. Bạn có thể fork ra hàng loạt tiến trình mới cả ngày và để chúng xử lý từng phần của bài toán một cách song song. Tất nhiên, mọi chuyện sẽ đơn giản nhất khi các tiến trình không cần giao tiếp với nhau trong lúc chạy mà cứ ngồi đó làm việc của mình.

Tuy nhiên, khi bạn bắt đầu `fork()` các tiến trình, bạn sẽ ngay lập tức nghĩ đến những ứng dụng đa người dùng thú vị nếu các tiến trình có thể nói chuyện với nhau dễ dàng. Vì vậy bạn thử tạo một mảng toàn cục rồi `fork()` để xem nó có được chia sẻ không. (Tức là, xem liệu cả tiến trình con lẫn tiến trình cha có dùng chung mảng đó không.) Và rồi dĩ nhiên bạn phát hiện ra rằng tiến trình con có bản sao riêng của mảng, còn tiến trình cha hoàn toàn không hay biết về bất kỳ thay đổi nào mà tiến trình con thực hiện.

Làm thế nào để những "ông" này nói chuyện với nhau, chia sẻ cấu trúc dữ liệu, và nhìn chung là hòa thuận với nhau? Tài liệu này thảo luận về một số phương pháp *Giao tiếp Liên tiến trình* (IPC) có thể thực hiện được điều đó, trong đó một số phương pháp phù hợp hơn với những tác vụ nhất định so với các phương pháp khác.

1.1 Đối tượng độc giả

Nếu bạn biết C hoặc C++ và khá quen với môi trường Unix (hoặc môi trường POSIX nào đó hỗ trợ các system call này) thì tài liệu này dành cho bạn. Nếu bạn chưa quen lắm, thì cũng đừng lo—bạn vẫn có thể hiểu được. Tuy nhiên tôi giả định rằng bạn có một lượng kinh nghiệm lập trình C nhất định.

Giống như Hướng dẫn Lập trình Mạng của Beej sử dụng Internet Sockets[1], những tài liệu này được thiết kế để làm bàn đạp đưa người đọc nói trên vào lĩnh vực IPC bằng cách cung cấp một cái nhìn tổng quan súc tích về các kỹ thuật IPC khác nhau. Đây không phải là bộ tài liệu đầy đủ toàn diện về chủ đề này, theo bất kỳ nghĩa nào. Như tôi đã nói, mục đích của nó chỉ đơn giản là giúp bạn có được chỗ đứng trong thế giới thú vị của IPC.

1.2 Nền tảng và Trình biên dịch

Các ví dụ trong tài liệu này được biên dịch trên Linux bằng `gcc`. Chúng cũng có thể biên dịch được ở bất kỳ nơi nào có trình biên dịch Unix tốt.

1.3 Trang chủ chính thức

Địa chỉ chính thức của tài liệu này là `https://beej.us/guide/bgipc/`[2].

[1] https://beej.us/guide/bgnet
[2] https://beej.us/guide/bgipc

1.4 Chính sách Email

Nhìn chung tôi sẵn lòng trả lời các câu hỏi qua email, vì vậy bạn cứ thoải mái gửi thư, nhưng tôi không thể đảm bảo sẽ trả lời. Cuộc sống của tôi khá bận rộn và đôi khi tôi thực sự không có thời gian để trả lời câu hỏi của bạn. Trong những trường hợp đó, tôi thường chỉ xóa email đi thôi. Không phải vì cá nhân; chỉ là tôi không bao giờ có đủ thời gian để đưa ra câu trả lời chi tiết mà bạn cần.

Về nguyên tắc, câu hỏi càng phức tạp thì tôi càng ít có khả năng trả lời. Nếu bạn thu hẹp câu hỏi của mình trước khi gửi và đảm bảo bao gồm mọi thông tin liên quan (như nền tảng, trình biên dịch, thông báo lỗi bạn gặp, và bất cứ thứ gì bạn nghĩ có thể giúp tôi xử lý sự cố), bạn sẽ có nhiều khả năng nhận được phản hồi hơn.

Nếu bạn không nhận được phản hồi, hãy tiếp tục mày mò, cố tìm câu trả lời, và nếu vẫn chưa ra thì viết lại cho tôi với những thông tin bạn đã tìm được, biết đâu sẽ đủ để tôi giúp được.

Sau khi đã "dạy bảo" bạn về cách viết và không viết cho tôi, tôi chỉ muốn nói rằng tôi *thực sự* trân trọng tất cả những lời khen mà hướng dẫn này đã nhận được trong nhiều năm qua. Đó là một nguồn động lực thực sự, và tôi rất vui khi biết nó đang được dùng vào mục đích tốt đẹp! `:-)` Cảm ơn bạn!

1.5 Sao chép trang web

Bạn hoàn toàn được chào đón để sao chép trang web này, dù công khai hay riêng tư. Nếu bạn sao chép công khai và muốn tôi đặt link đến trang của bạn từ trang chính, hãy nhắn tôi tại `beej@beej.us`.

1.6 Ghi chú cho Người dịch

Nếu bạn muốn dịch hướng dẫn này sang ngôn ngữ khác, hãy viết cho tôi tại [`beej@beej.us`] và tôi sẽ đặt link đến bản dịch của bạn từ trang chính. Bạn có thể tự do thêm tên và thông tin liên hệ của mình vào bản dịch.

Vui lòng lưu ý các hạn chế về giấy phép trong phần Bản quyền và Phân phối bên dưới.

1.7 Bản quyền và Phân phối

Chapter 2

Nhập môn `fork()`

"Fork" (dĩa), ngoài việc là một trong những từ trông ngày càng kỳ lạ hơn sau khi bạn gõ đi gõ lại nhiều lần, còn đề cập đến cách Unix tạo ra các tiến trình mới. Tài liệu này cung cấp một bài nhập môn nhanh và thực tế về `fork()`, vì system call này sẽ xuất hiện trong các tài liệu IPC khác. Nếu bạn đã biết hết về `fork()` rồi thì có thể bỏ qua tài liệu này.

2.1 "Hãy tìm đến Hẻm Núi Nguy Hiểm Muôn Đời"

`fork()` có thể được coi như là tờ vé đến với quyền năng. Quyền năng đôi khi lại là tờ vé dẫn đến hủy diệt. Do đó, bạn phải cẩn thận khi mày mò với `fork()` trên hệ thống của mình, đặc biệt khi mọi người đang gấp rút làm đồ án cuối kỳ gần đến hạn và sẵn sàng "xử lý" bất kỳ thứ gì làm hệ thống chết đứng. Không phải là bạn không bao giờ được chơi với `fork()`, chỉ là bạn cần thận trọng. Nó giống như nuốt gươm vậy—nếu cẩn thận, bạn sẽ không tự mổ bụng mình.

Vì bạn vẫn còn đây, tôi nghĩ tốt hơn là tôi nên nói thẳng vào vấn đề. Như tôi đã nói, `fork()` là cách Unix khởi động các tiến trình mới. Về cơ bản, cách hoạt động là thế này: tiến trình cha (tiến trình đã tồn tại) `fork()` ra một tiến trình con (tiến trình mới). Tiến trình con nhận được một *bản sao* dữ liệu của cha. *Voila!* Bạn có hai tiến trình từ chỗ chỉ có một!

Tất nhiên, có đủ loại bẫy mà bạn phải đối phó khi `fork()` các tiến trình, nếu không sysadmin của bạn sẽ nổi giận với bạn khi bạn làm đầy bảng tiến trình của hệ thống và họ phải ấn nút reset máy.

Trước tiên, bạn cần biết điều gì đó về hành vi của tiến trình trong Unix. Khi một tiến trình chết, nó không thực sự biến mất hoàn toàn. Nó đã chết nên không còn chạy nữa, nhưng một mảnh nhỏ còn chờ đợi để tiến trình cha thu dọn. Mảnh nhỏ này chứa giá trị trả về từ tiến trình con và một số thứ linh tinh khác. Vì vậy sau khi tiến trình cha `fork()` ra một tiến trình con, nó phải `wait()` (hoặc `waitpid()`) để chờ tiến trình con đó thoát. Chính hành động `wait()` này mới cho phép tất cả những gì còn sót lại của tiến trình con biến mất.

Tất nhiên, có một ngoại lệ cho quy tắc trên: tiến trình cha có thể bỏ qua tín hiệu `SIGCHLD` (là `SIGCLD` trên một số hệ thống cũ hơn) và khi đó nó sẽ không cần phải `wait()`. Điều này có thể được thực hiện (trên các hệ thống hỗ trợ nó) như sau:

```
main()
{
    signal(SIGCHLD, SIG_IGN);  /* now I don't have to wait()! */
    .
    .
    fork();fork();fork();  /* Rabbits, rabbits, rabbits! */
```

Bây giờ, khi một tiến trình con chết mà không được `wait()`, nó thường sẽ hiển thị trong danh sách `ps` dưới dạng " `<defunct>` ". Nó sẽ ở trạng thái này cho đến khi tiến trình cha `wait()` nó, hoặc được xử lý như đã đề cập bên dưới.

Bây giờ có một quy tắc khác bạn phải học: khi tiến trình cha chết trước khi nó `wait()` tiến trình con (giả sử nó không bỏ qua `SIGCHLD`), tiến trình con sẽ được nhận làm con của tiến trình `init` (PID 1). Đây không phải là vấn đề nếu tiến trình con vẫn đang sống tốt và trong tầm kiểm soát. Tuy nhiên, nếu tiến trình con đã ở trạng thái defunct rồi, chúng ta sẽ gặp rắc rối. Vì tiến trình cha ban đầu không thể `wait()` nữa vì nó đã chết. Vậy làm sao `init` biết để `wait()` các *tiến trình zombie* này?

Câu trả lời: đó là phép thuật! Thực ra trên một số hệ thống, `init` định kỳ hủy tất cả các tiến trình defunct mà nó sở hữu. Trên các hệ thống khác, nó thẳng thừng từ chối trở thành cha của bất kỳ tiến trình defunct nào, thay vào đó hủy chúng ngay lập tức. Nếu bạn đang dùng một trong các hệ thống kiểu trước, bạn có thể dễ dàng viết một vòng lặp làm đầy bảng tiến trình bằng các tiến trình defunct thuộc sở hữu của `init`. Sysadmin của bạn sẽ vui lòng lắm đấy?

Nhiệm vụ của bạn: đảm bảo tiến trình cha của bạn hoặc bỏ qua `SIGCHLD`, hoặc `wait()` tất cả các con mà nó đã `fork()`. Thực ra bạn không *luôn luôn* phải làm vậy (ví dụ nếu bạn đang khởi động một daemon hay gì đó), nhưng hãy lập trình cẩn thận nếu bạn là người mới với `fork()`. Nếu không, cứ thoải mái phóng thẳng lên tầng bình lưu.

Tóm lại: các con trở thành defunct cho đến khi cha `wait()`, trừ khi cha đang bỏ qua `SIGCHLD`. Hơn nữa, các con (còn sống hoặc defunct) mà cha chết mà không `wait()` chúng (một lần nữa giả sử cha không bỏ qua `SIGCHLD`) sẽ trở thành con của tiến trình `init`, nơi xử lý chúng khá thẳng tay.

2.2 "Tôi đã sẵn sàng tinh thần! Cho tôi *Cái Nút* đó!"

Được thôi! Đây là một ví dụ[1] về cách sử dụng `fork()`:

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

int main(void)
{
    pid_t pid;
    int rv;

    switch(pid = fork()) {
    case -1:
        perror("fork");  /* something went wrong */
        exit(1);         /* parent exits */

    case 0:
        printf(" CHILD: This is the child process!\n");
        printf(" CHILD: My PID is %d\n", getpid());
        printf(" CHILD: My parent's PID is %d\n", getppid());
        printf(" CHILD: Enter my exit status (make it small): ");
        scanf(" %d", &rv);
        printf(" CHILD: I'm outta here!\n");
        exit(rv);

    default:
        printf("PARENT: This is the parent process!\n");
        printf("PARENT: My PID is %d\n", getpid());
        printf("PARENT: My child's PID is %d\n", pid);
        printf("PARENT: I'm now waiting for my child to exit()...\n");
```

[1] https://beej.us/guide/bgipc/source/examples/fork1.c

```
        wait(&rv);
        printf("PARENT: My child's exit status is: %d\n", WEXITSTATUS(rv));
        printf("PARENT: I'm outta here!\n");
    }

    return 0;
}
```

Có rất nhiều điều cần lưu ý từ ví dụ này, vậy ta cứ bắt đầu từ đầu nhé.

`pid_t` là kiểu tiến trình tổng quát. Trong Unix, đây là một `short`. Vì vậy tôi gọi `fork()` và lưu giá trị trả về vào biến `pid`. `fork()` rất dễ, vì nó chỉ có thể trả về ba giá trị:

Giá trị trả về	Mô tả
`0`	Nếu nó trả về `0`, bạn là tiến trình con. Bạn có thể lấy PID của cha bằng cách gọi `getppid()`. Tất nhiên, bạn có thể lấy PID của chính mình bằng cách gọi `getpid()`.
`-1`	Nếu nó trả về `-1`, có điều gì đó đã xảy ra sai, và không có tiến trình con nào được tạo. Dùng `perror()` để xem điều gì đã xảy ra. Có lẽ bạn đã làm đầy bảng tiến trình—nếu bạn quay lại bạn sẽ thấy sysadmin đang đến với chiếc rìu cứu hỏa.
Bất kỳ giá trị nào khác	Bất kỳ giá trị nào khác được trả về bởi `fork()` có nghĩa là bạn là tiến trình cha và giá trị trả về là PID của con bạn. Đây là cách duy nhất để lấy PID của con bạn, vì không có lệnh `getcpid()` (hiển nhiên do mối quan hệ một-nhiều giữa cha và con.)

Khi tiến trình con cuối cùng gọi `exit()`, giá trị trả về được truyền sẽ đến tiến trình cha khi nó `wait()`. Như bạn có thể thấy từ lệnh `wait()`, có sự kỳ lạ khi chúng ta in giá trị trả về. Cái `WEXITSTATUS()` này là gì vậy? Đó là một macro trích xuất giá trị trả về thực sự của tiến trình con từ giá trị mà `wait()` trả về. Đúng, còn nhiều thông tin ẩn trong `int` đó. Tôi để bạn tự tra cứu.

"Làm thế nào," bạn hỏi, "`wait()` biết phải đợi tiến trình nào? Ý tôi là, vì tiến trình cha có thể có nhiều con, `wait()` thực sự đợi cái nào?" Câu trả lời đơn giản, bạn ơi: nó đợi cái nào thoát ra đầu tiên. Nếu cần, bạn có thể chỉ định chính xác con nào cần đợi bằng cách gọi `waitpid()` với PID của con bạn làm đối số.

Một điều thú vị khác cần lưu ý từ ví dụ trên là cả cha và con đều dùng biến `rv`. Điều này có nghĩa là nó được chia sẻ giữa các tiến trình không? *KHÔNG!* Nếu vậy thì tôi đã không viết hết mọi thứ về IPC này. *Mỗi tiến trình có bản sao riêng của tất cả các biến.* Còn nhiều thứ khác cũng được sao chép, nhưng bạn sẽ phải đọc trang `man` để biết thêm.

Một lưu ý cuối về chương trình trên: tôi đã dùng câu lệnh switch để xử lý `fork()`, và điều đó không phải là điển hình. Thông thường bạn sẽ thấy câu lệnh if ở đó; đôi khi ngắn như:

```
if (!fork()) {
        printf("I'm the child!\n");
        exit(0);
    } else {
        printf("I'm the parent!\n");
        wait(NULL);
    }
```

À phải—ví dụ trên cũng minh họa cách `wait()` nếu bạn không quan tâm đến giá trị trả về của tiến trình con: chỉ cần gọi nó với `NULL` làm đối số.

2.3 Tóm tắt

Bây giờ bạn đã biết tất cả về hàm `fork()` oai phong! Nó hữu ích hơn một túi giun ướt trong hầu hết các tình huống tính toán cường độ cao, và bạn có thể gây ấn tượng với bạn bè ở các buổi tiệc. Tôi thề đấy. Thử đi.

Chapter 3

Signals (Tín hiệu)

Có một phương pháp đôi khi rất hữu ích để một tiến trình "quấy rầy" tiến trình khác: signal. Về cơ bản, một tiến trình có thể "raise" (phát) một signal và gửi nó đến một tiến trình khác. Signal handler (chỉ là một hàm) của tiến trình đích sẽ được gọi và tiến trình có thể xử lý nó.

Đây là một cơ chế thú vị khác với những gì bạn có thể quen: chương trình của bạn đang chạy vui vẻ làm công việc của nó, và rồi một signal được phát và chương trình của bạn bị ngắt. Code của bạn có thể đang ở giữa một hàm tính π đến 1,21 tỷ chữ số thập phân, và đột nhiên nó dừng lại và quyền điều khiển chuyển sang một hàm khác bạn đã viết (là *signal handler*) để xử lý signal.

Và khi signal handler trả về, quyền điều khiển nhảy lại vào phép tính π của bạn và tiếp tục từ chỗ đã dừng. Hoặc có thể chương trình chỉ đơn giản là kết thúc! Tất cả phụ thuộc vào signal và việc bạn có quyết định xử lý nó hay không và xử lý như thế nào.

Tất nhiên, ma quỷ ở trong các chi tiết, và trên thực tế những gì bạn được phép thực hiện an toàn bên trong signal handler của mình khá hạn chế. Dẫu vậy, signal vẫn cung cấp một dịch vụ hữu ích.

Ví dụ, một tiến trình có thể muốn tạm thời dừng một tiến trình khác, và điều này có thể được thực hiện bằng cách gửi signal `SIGSTOP` đến tiến trình đó. Để tiếp tục, tiến trình phải nhận signal `SIGCONT` [1]. Làm sao tiến trình biết phải làm điều này khi nhận được một signal nhất định? Thực ra nhiều signal được định nghĩa sẵn và tiến trình có một default signal handler để xử lý chúng.

Default handler? Đúng vậy. Lấy `SIGINT` làm ví dụ. Đây là signal ngắt mà một tiến trình nhận được khi người dùng nhấn `CTRL-C`. Default signal handler cho `SIGINT` khiến tiến trình thoát! Nghe quen không? Thực ra, như bạn có thể tưởng tượng, bạn có thể ghi đè signal `SIGINT` để làm bất cứ điều gì bạn muốn (hoặc không làm gì cả!) Bạn có thể khiến tiến trình in ra "Interrupt?! No way, Jose!" và tiếp tục công việc vui vẻ của nó.

Vậy giờ bạn biết rằng bạn có thể khiến tiến trình của mình phản ứng với hầu hết mọi signal theo bất kỳ cách nào bạn muốn. Tất nhiên, có những ngoại lệ vì nếu không thì sẽ quá dễ để hiểu. Hãy xem `SIGKILL` nổi tiếng, signal số 9. Bạn đã từng gõ "`kill -9 nnnn`" để tắt một tiến trình số `nnnn` đang chạy loạn không? Bạn đã gửi cho nó `SIGKILL`. Bạn cũng có thể nhớ rằng không tiến trình nào có thể thoát khỏi "`kill -9`", và điều đó hoàn toàn đúng. `SIGKILL` là một trong những signal bạn **không thể** thêm signal handler riêng. `SIGSTOP` đã đề cập ở trên cũng thuộc danh mục này.

(Ghi chú thêm: bạn thường dùng lệnh Unix `kill` mà không chỉ định signal cần gửi...vậy signal đó là gì? Câu trả lời: `SIGTERM`. Bạn có thể viết handler riêng cho `SIGTERM` để tiến trình của bạn không phản ứng với lệnh "`kill`" thông thường, và người dùng phải dùng "`kill -9`" để kết thúc tiến trình.)

Tất cả signal đều được định nghĩa sẵn không? Sẽ ra sao nếu bạn muốn gửi một signal có ý nghĩa chỉ bạn hiểu đến một tiến trình? Có hai signal không được đặt trước: `SIGUSR1` và `SIGUSR2`. Bạn hoàn toàn tự do dùng chúng cho bất cứ điều gì và xử lý chúng theo bất kỳ cách nào bạn chọn. (Ví dụ, chương trình

[1] Mẹo vui: khi bạn nhấn `CTRL-Z` trong terminal trong khi đang chạy một chương trình ở foreground, nó sẽ gửi `SIGSTOP` đến tiến trình đó và shell báo cáo rằng nó đã bị dừng hoặc tạm dừng. Nếu bạn gõ `fg`, nó sẽ đưa tiến trình đó trở lại foreground và gửi `SIGCONT` để tiếp tục chạy từ chỗ đã dừng.

CD player của tôi có thể phản ứng với `SIGUSR1` bằng cách chuyển sang bài hát tiếp theo. Bằng cách đó, tôi có thể điều khiển nó từ dòng lệnh bằng cách gõ "`kill -SIGUSR1 nnnn`".)

3.1 Bắt Signal để Vui và Kiếm Lợi!

Như bạn có thể đoán, lệnh Unix "kill" là một cách để gửi signal đến một tiến trình. Thật trùng hợp không thể tin được, có một system call gọi là `kill()` làm điều tương tự. Nó nhận một số signal (như được định nghĩa trong `signal.h`) và một process ID làm đối số. Ngoài ra, còn có một thư viện routine gọi là `raise()` có thể dùng để phát một signal trong chính tiến trình đó.

Câu hỏi nóng bỏng vẫn còn đó: làm thế nào để bạn bắt một `SIGTERM` đang bay? Bạn cần gọi `sigaction()` và cho nó biết tất cả các chi tiết về signal bạn muốn bắt và hàm bạn muốn gọi để xử lý nó.

Đây là phân tích `sigaction()`:

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

Tham số đầu tiên, `sig` là signal cần bắt. Đây có thể là (có lẽ "nên" là) một tên ký hiệu từ `signal.h` kiểu như `SIGINT`. Đó là phần dễ.

Trường tiếp theo, `act` là con trỏ đến một `struct sigaction` có nhiều trường bạn có thể điền vào để kiểm soát hành vi của signal handler. (Con trỏ đến chính hàm signal handler được bao gồm trong `struct`.)

Cuối cùng `oact` có thể là `NULL`, nhưng nếu không, nó trả về thông tin signal handler *cũ* đã có trước đó. Điều này hữu ích nếu bạn muốn khôi phục signal handler trước đó vào một lúc nào đó sau.

Chúng ta sẽ tập trung vào ba trường này trong `struct sigaction`:

Signal	Mô tả
`sa_handler`	Hàm signal handler
`sa_mask`	Tập hợp các signal cần chặn trong khi signal này đang được xử lý
`sa_flags`	Các cờ để thay đổi hành vi của handler, hoặc `0`

`sa_handler` là con trỏ đến một hàm trả về `void` và nhận một tham số `int` duy nhất (sẽ giữ số signal mà nó đang xử lý). Bạn cũng có thể chỉ định `SIG_IGN` để bỏ qua signal, hoặc `SIG_DEF` để đặt nó về hành động mặc định.

Còn trường `sa_mask`? Khi bạn đang xử lý một signal, bạn có thể muốn chặn các signal khác không được gửi đến, và bạn có thể làm điều này bằng cách thêm chúng vào `sa_mask`. Đây là một "tập hợp", nghĩa là bạn có thể thực hiện các phép toán tập hợp bình thường để thao tác chúng: `sigemptyset()`, `sigfillset()`, `sigaddset()`, `sigdelset()`, và `sigismember()`. Trong ví dụ này, chúng ta sẽ chỉ xóa tập hợp và không chặn bất kỳ signal nào khác.

Ví dụ luôn hữu ích! Đây là một ví dụ xử lý `SIGINT`, có thể được gửi bằng cách nhấn `^C`, có tên là `sigint.c`[2]:

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

[2] https://beej.us/guide/bgipc/source/examples/sigint.c

```
void sigint_handler(int sig)
{
    (void)sig;
    const char msg[] = "Ahhh! SIGINT!\n";
    write(1, msg, sizeof msg  - 1);
}

int main(void)
{
    char s[200];
    struct sigaction sa = {
        .sa_handler = sigint_handler,
        .sa_flags = 0, // or SA_RESTART
    };
    sigemptyset(&sa.sa_mask);

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    printf("Enter a string:\n");

    if (fgets(s, sizeof s, stdin) == NULL)
        perror("fgets");
    else
        printf("You entered: %s\n", s);

    return 0;
}
```

Chương trình này có hai hàm: `main()` thiết lập signal handler (sử dụng lệnh gọi `sigaction()`), và `sigint_handler()` là bản thân signal handler.

Điều gì xảy ra khi bạn chạy nó? Nếu bạn đang nhập một chuỗi và nhấn `^C`, lệnh gọi `gets()` thất bại và đặt biến toàn cục `errno` thành `EINTR`. Ngoài ra, `sigint_handler()` được gọi và thực hiện công việc của nó, vì vậy bạn thực sự thấy:

```
Enter a string:
the quick brown fox jum^CAhhh! SIGINT!
fgets: Interrupted system call
```

Và rồi nó thoát. Này—đây là kiểu handler gì vậy, nếu nó chỉ thoát ra bất kể thế nào?

Thực ra, có một vài điều đang diễn ra ở đây. Đầu tiên, bạn sẽ nhận thấy rằng signal handler đã được gọi, vì nó in ra "Ahhh! SIGINT!" Nhưng sau đó `fgets()` trả về lỗi, cụ thể là `EINTR`, hay "Interrupted system call". Thấy đó, một số system call có thể bị ngắt bởi signal, và khi điều này xảy ra, chúng trả về lỗi. Bạn có thể thấy code như thế này (đôi khi được trích dẫn như là một cách dùng `goto` có thể bỏ qua):

```
restart:
    if (some_system_call() == -1) {
        if (errno == EINTR) goto restart;
        perror("some_system_call");
        exit(1);
    }
```

Thay vì dùng `goto` như vậy, bạn có thể đặt `sa_flags` để bao gồm `SA_RESTART`. Ví dụ, nếu chúng ta thay đổi code handler `SIGINT` thành:

```
    sa.sa_flags = SA_RESTART;
```

Thì kết quả chạy của chúng ta trông như thế này hơn:

```
Enter a string:
Hello^CAhhh! SIGINT!
Er, hello!^CAhhh! SIGINT!
This time fer sure!
You entered: This time fer sure!
```

Một số system call có thể bị ngắt, và một số có thể được khởi động lại. Điều này phụ thuộc vào hệ thống.

3.2 Còn `signal()` thì sao

ANSI C định nghĩa một hàm gọi là `signal()` có thể được dùng để bắt signal. Nó không đáng tin cậy hoặc đầy đủ tính năng như `sigaction()`, vì vậy việc sử dụng `signal()` thường không được khuyến khích.

3.3 Một số signal để bạn trở nên nổi tiếng

Đây là danh sách các signal bạn (rất có thể) có sẵn:

Signal	Mô tả
`SIGABRT`	Signal hủy tiến trình.
`SIGALRM`	Đồng hồ báo thức.
`SIGFPE`	Phép toán số học sai.
`SIGHUP`	Ngắt kết nối (Hangup).
`SIGILL`	Lệnh không hợp lệ.
`SIGINT`	Signal ngắt terminal.
`SIGKILL`	Kill (không thể bắt hoặc bỏ qua).
`SIGPIPE`	Ghi vào pipe mà không có ai đọc.
`SIGQUIT`	Signal thoát terminal.
`SIGSEGV`	Tham chiếu bộ nhớ không hợp lệ.
`SIGTERM`	Signal kết thúc tiến trình.
`SIGUSR1`	Signal do người dùng định nghĩa 1.
`SIGUSR2`	Signal do người dùng định nghĩa 2.
`SIGCHLD`	Tiến trình con kết thúc hoặc bị dừng.
`SIGCONT`	Tiếp tục thực thi, nếu đã bị dừng.
`SIGSTOP`	Dừng thực thi (không thể bắt hoặc bỏ qua).
`SIGTSTP`	Signal dừng terminal.
`SIGTTIN`	Tiến trình nền đang cố đọc.
`SIGTTOU`	Tiến trình nền đang cố ghi.
`SIGBUS`	Lỗi bus.
`SIGPOLL`	Sự kiện có thể polling.
`SIGPROF`	Hết thời gian đếm profiling.
`SIGSYS`	System call không hợp lệ.
`SIGTRAP`	Bẫy trace/breakpoint.
`SIGURG`	Dữ liệu băng thông cao có sẵn tại socket.
`SIGVTALRM`	Hết thời gian đồng hồ ảo.
`SIGXCPU`	Vượt quá giới hạn thời gian CPU.
`SIGXFSZ`	Vượt quá giới hạn kích thước file.

Mỗi signal có default signal handler riêng của nó, hành vi của chúng được định nghĩa trong các trang man trên hệ thống của bạn.

3.4 Những Con Rồng của Reentrancy

Nếu bạn đang bận làm gì đó với dữ liệu global hoặc static (gọi biến đó là `alvin`) và rồi bạn bị ngắt, điều gì xảy ra nếu handler *cũng* sửa đổi `alvin`? Và rồi handler trả về và `alvin` đã bị sửa đổi sau lưng bạn! Và hàm của bạn không có cách nào biết điều đó! Tệ hơn, các cấu trúc dữ liệu lớn có thể chỉ được ghi *một phần* khi handler được gọi, dẫn đến rách dữ liệu và trạng thái hỗn loạn khủng khiếp.

Chúng ta gọi đây là *vấn đề reentrancy*.

Điều đó có nghĩa là gì? Nếu tôi được phép, tôi sẽ lười biếng trích dẫn bài viết Wikipedia về reentrancy[3]:

> Reentrant code được thiết kế để an toàn và có thể dự đoán khi nhiều instance của cùng một hàm được gọi đồng thời hoặc liên tiếp nhanh chóng. Một chương trình máy tính hoặc chương trình con được gọi là reentrant nếu nhiều lần gọi có thể chạy đồng thời an toàn trên nhiều processor, hoặc nếu trên hệ thống đơn processor, việc thực thi của nó có thể bị ngắt và một lần thực thi mới có thể được khởi động an toàn (nó có thể được "re-entered"). Sự ngắt có thể được gây ra bởi hành động nội bộ như nhảy hoặc gọi [...], hoặc bởi hành động bên ngoài như một ngắt hoặc signal.

Đây là một ví dụ minh họa[4], liệt kê một phần bên dưới. Hàm `increment()` không phải reentrant đối với các signal bất đồng bộ.

Hãy tưởng tượng hàm `increment()` từ từ tăng `count` toàn cục. Nhưng chờ đã! Nếu signal handler kích hoạt lúc này, nó sẽ đặt `count` thành một giá trị mà `increment()` không mong đợi! Và rồi mọi thứ sẽ nổ tung.

(Chúng ta sẽ đến `volatile sig_atomic_t` sau; bây giờ chỉ cần giả sử đó là `int`.)

```
volatile sig_atomic_t count;

void handler(int sig)
{
    (void)sig;

    count = 123;
}

void increment(void)
{
    int next_count = count + 1;

    printf("Count is %d, next should be %d\n", count, next_count);

    // Sleep to slow down time to demo the problem
    sleep(2);
    count++;

    if (count == next_count)
        puts("Everything is swell!");
    else
        printf("%d != %d! Aaa! ERROR DOES NOT COMPUTE!\n", count,
            next_count);
}
```

[3] https://en.wikipedia.org/wiki/Reentrancy_(computing)
[4] https://beej.us/guide/bgipc/source/examples/sigcount.c

Bài học của bạn: bất cứ khi nào bạn dựa vào một loại trạng thái chia sẻ nào đó, bạn có thể gặp rắc rối với signal nếu signal handler cũng sửa đổi trạng thái chia sẻ đó.

Đây là một ví dụ khác sử dụng `strtok()` [5], là một hàm nổi tiếng về tính không-reentrant.

```
void handler(int sig)
{
    (void)sig;

    char x[] = "Hello, world!";
    char *token;

    if ((token = strtok(x, " ")) != NULL) do {
        write(1, "In handler: ", 12);
        write(1, token, strlen(token));
        write(1, "\n", 1);
    } while ((token = strtok(NULL, " ")) != NULL);
}

void tokenizer(void)
{
    char s[] = "The quick brown fox jumped over the lazy dogs";
    char *token;

    if ((token = strtok(s, " ")) != NULL) do {
        printf("In main: %s\n", token);
        // Sleep to slow down time to demo the problem
        sleep(1);
    } while ((token = strtok(NULL, " ")) != NULL);

    puts("Done tokenizing");
}
```

Giả sử rằng hai giây sau khi vào hàm `tokenizer()`, signal handler được gọi. `handler()` thực hiện tokenize riêng trên chuỗi của nó và in các token[6].

Nếu mọi thứ diễn ra tốt và hợp lý, chúng ta sẽ thấy đầu ra này (nhưng thực tế không phải vậy):

```
In main: The
In main: quick
In handler: Hello,
In handler: world!
In main: brown
In main: fox
In main: jumped
In main: over
In main: the
In main: lazy
In main: dogs
Done tokenizing
```

Thấy signal xảy ra, được xử lý, và `tokenizer()` tiếp tục từ chỗ đã dừng không? Tuyệt vời đúng không?

Thay vào đó chúng ta thấy điều này (có lẽ):

[5] https://beej.us/guide/bgipc/source/examples/sigstrtok.c

[6] Và nó dùng `write()` vì `printf()` không phải reentrant!

```
In main: The
In main: quick
In handler: Hello,
In handler: world!
Done tokenizing
```

Phần còn lại đâu rồi?

`strtok()` duy trì một số trạng thái nội bộ trong một biến `static`. Hàm `tokenize()` của chúng ta đang mong đợi trạng thái ở một trạng thái nhất định, và signal handler đã ghi đè lên nó, khiến `tokenize()` hoạt động sai.

Và điều này làm cho `strtok()` không-reentrant (và do liên kết, `tokenize()` cũng không-reentrant).

Nhưng cách sửa thì dễ. Chúng ta chỉ cần một phiên bản reentrant của `strtok()` không có trạng thái chia sẻ nội bộ. Và chúng ta có một cái trong `strtok_r()`. Với hàm đó, *chúng ta* sở hữu trạng thái và chúng ta truyền nó vào cho `strtok_r()` sử dụng. Mọi phần của code muốn có vòng lặp `strtok_r()` sẽ có trạng thái riêng và không ai giẫm lên chân của người khác.

Đây là code cho `strtok_r()` trong hàm `tokenizer()` (tương tự cho hàm `handler()`):

```
    char *lasts;

    if ((token = strtok_r(s, " ", &lasts)) != NULL) do {
        printf("In main: %s\n", token);
        // Sleep to slow down time to demo the problem
        sleep(1);
    } while ((token = strtok_r(NULL, " ", &lasts)) != NULL);
```

Thấy cách chúng ta theo dõi trạng thái của mình trong `lasts` không? Nếu bạn thay thế tất cả `strtok()` bằng `strtok_r()` trong chương trình demo, nó sẽ hoạt động đúng vì tất cả chức năng được sử dụng bởi cả `handler()` và `tokenizer()` đều là reentrant.

3.5 Dữ liệu Global Chia sẻ

Bạn không thể an toàn thay đổi bất kỳ dữ liệu chia sẻ nào (ví dụ global), với một ngoại lệ đáng chú ý: các biến được khai báo là storage class và kiểu `volatile sig_atomic_t`. Đây là một kiểu integer giữ một số phạm vi giá trị; spec C đảm bảo rằng bạn sẽ ít nhất có thể giữ 0 đến 127, bao gồm cả hai đầu. Nhưng phạm vi thực tế phụ thuộc vào hệ thống và liệu kiểu có dấu hay không. (Bạn có thể xem `SIG_ATOMIC_MIN` và `SIG_ATOMIC_MAX` để biết giới hạn của mình.)

Spec rất thận trọng. Về cơ bản nó nói bạn đang hành động rất tệ nếu bạn làm bất cứ điều gì với dữ liệu global ngoài việc gán vào một biến kiểu `volatile sig_atomic_t`. Nhưng điều đó không *hoàn toàn* đúng. *Có lẽ* an toàn khi đọc từ biến cũng vậy, nhưng hãy lưu ý rằng ngay khi bạn đọc và ghi vào cùng một biến, bạn chắc chắn đang mở bản thân cho một số điều kiện race tùy thuộc vào ai khác đọc và sửa đổi các giá trị đó.

Một ngoại lệ khác là dữ liệu global chia sẻ không bao giờ thay đổi. Nếu bạn thiết lập một số biến global trước khi signal handler được cài đặt, và bạn không bao giờ thay đổi các giá trị đó, thì signal handler có thể đọc chúng một cách tự do. Chúng có thể thuộc bất kỳ kiểu nào.

Đây là một ví dụ xử lý `SIGUSR1` bằng cách đặt một cờ global, sau đó được kiểm tra trong vòng lặp chính để xem liệu handler đã được gọi chưa. Đây là `sigusr.c`[7]:

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

[7] https://beej.us/guide/bgipc/source/examples/sigusr.c

```
#include <unistd.h>
#include <errno.h>
#include <signal.h>

volatile sig_atomic_t got_usr1;

void sigusr1_handler(int sig)
{
    got_usr1 = 1;
}

int main(void)
{
    struct sigaction sa = {
        .sa_handler = sigusr1_handler,
        .sa_flags = 0, // or SA_RESTART
    };
    sigemptyset(&sa.sa_mask);

    got_usr1 = 0;

    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while (!got_usr1) {
        printf("PID %d: working hard...\n", getpid());
        sleep(1);
    }

    printf("Done in by SIGUSR1!\n");

    return 0;
}
```

Khởi động nó trong một cửa sổ, rồi dùng `kill -USR1` trong cửa sổ khác để kết thúc nó. Chương trình `sigusr` tiện lợi in ra process ID của nó để bạn có thể truyền nó vào `kill`:

```
$ sigusr
PID 5023: working hard...
PID 5023: working hard...
PID 5023: working hard...
```

Sau đó trong cửa sổ kia, gửi cho nó signal `SIGUSR1`:

```
$ kill -USR1 5023
```

Và chương trình sẽ phản hồi:

```
PID 5023: working hard...
PID 5023: working hard...
Done in by SIGUSR1!
```

(Và phản hồi sẽ ngay lập tức ngay cả khi `sleep()` vừa được gọi— `sleep()` bị ngắt bởi signal.)

Việc cấu trúc code theo cách này hơi phản trực giác. Chẳng phải handler nên có tất cả logic xử lý và

một đoạn code khác hay sao? Điều gì xảy ra nếu signal được phát khi code khác đang chạy mà không thể xử lý nó?

Đó là một nhược điểm nhỏ, nhưng cấu trúc code theo cách này có một lợi ích lớn: *tạm biệt, vấn đề reentrancy!* Và điều đó không hề tệ.

3.5.1 Độ an toàn signal bất đồng bộ

Trước những cạm bẫy reentrancy bạn có thể gặp, bạn phải cẩn thận khi thực hiện các lệnh gọi hàm trong signal handler của mình, và thực sự, khi handler của bạn sửa đổi bất kỳ trạng thái global nào mà các hàm khác có thể đang sử dụng.

Các hàm đó phải *an toàn với signal bất đồng bộ (async-signal-safe)*, điều này thường có nghĩa là hàm không làm bất cứ điều gì có thể gây ra vấn đề reentrancy.

Nói chung, bạn có thể không an toàn với signal bất đồng bộ nếu bạn làm bất kỳ điều nào trong số này:

- Sửa đổi một biến global không-atomic trong hàm của bạn.
- Đọc một biến global không-constant mà không phải atomic.
- Dùng dữ liệu `static` trong hàm hoặc trong handler của bạn.
- Gọi bất kỳ hàm nào khác không phải async-signal-safe.

Khá hạn chế.

Bạn có thể tò mò, ví dụ, tại sao signal handler trong ví dụ trước của tôi gọi `write()` để xuất thông báo thay vì `printf()`. Câu trả lời là POSIX nói rằng `write()` là async-signal-safe (vì vậy an toàn khi gọi từ bên trong handler), trong khi `printf()` thì không.

Các hàm thư viện và system call là async-signal-safe và có thể được gọi từ bên trong signal handler của bạn là (hít thở):

`_Exit()`, `_exit()`, `abort()`, `accept()`, `access()`, `aio_error()`, `aio_return()`, `aio_suspend()`, `alarm()`, `bind()`, `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()`, `chdir()`, `chmod()`, `chown()`, `clock_gettime()`, `close()`, `connect()`, `creat()`, `dup()`, `dup2()`, `execle()`, `execve()`, `fchmod()`, `fchown()`, `fcntl()`, `fdatasync()`, `fork()`, `fpathconf()`, `fstat()`, `fsync()`, `ftruncate()`, `getegid()`, `geteuid()`, `getgid()`, `getgroups()`, `getpeername()`, `getpgrp()`, `getpid()`, `getppid()`, `getsockname()`, `getsockopt()`, `getuid()`, `kill()`, `link()`, `listen()`, `lseek()`, `lstat()`, `mkdir()`, `mkfifo()`, `open()`, `pathconf()`, `pause()`, `pipe()`, `poll()`, `posix_trace_event()`, `pselect()`, `raise()`, `read()`, `readlink()`, `recv()`, `recvfrom()`, `recvmsg()`, `rename()`, `rmdir()`, `select()`, `sem_post()`, `send()`, `sendmsg()`, `sendto()`, `setgid()`, `setpgid()`, `setsid()`, `setsockopt()`, `setuid()`, `shutdown()`, `sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `sleep()`, `signal()`, `sigpause()`, `sigpending()`, `sigprocmask()`, `sigqueue()`, `sigset()`, `sigsuspend()`, `sockatmark()`, `socket()`, `socketpair()`, `stat()`, `symlink()`, `sysconf()`, `tcdrain()`, `tcflow()`, `tcflush()`, `tcgetattr()`, `tcgetpgrp()`, `tcsendbreak()`, `tcsetattr()`, `tcsetpgrp()`, `time()`, `timer_getoverrun()`, `timer_gettime()`, `timer_settime()`, `times()`, `umask()`, `uname()`, `unlink()`, `utime()`, `wait()`, `waitpid()`, and `write()`.

Tất nhiên, bạn có thể gọi các hàm của riêng mình từ bên trong signal handler (miễn là chúng là async-signal-safe và không gọi bất kỳ hàm không-async-signal-safe nào).

Trong chương tiếp theo, chúng ta sẽ xem xét một số pattern để phản ứng an toàn khi một signal được phát.

3.6 Những Gì Tôi Đã Lướt Qua

Hầu như tất cả mọi thứ. Có hàng tấn cờ, realtime signal, kết hợp signal với thread, che giấu signal, `longjmp()` và signal, và nhiều hơn nữa. Tôi có một chương tiếp theo với tài liệu chuyên sâu hơn, nhưng tôi có thể tạo cả một hướng dẫn riêng chỉ từ chủ đề này!

Tất nhiên, đây chỉ là hướng dẫn "bắt đầu", nhưng trong nỗ lực cuối cùng để cung cấp cho bạn thêm thông tin, đây là danh sách các trang man với nhiều thông tin hơn:

Xử lý signal:

- `sigaction()` [8]
- `sigwait()` [9]
- `sigwaitinfo()` [10]
- `sigtimedwait()` [11]
- `sigsuspend()` [12]
- `sigpending()` [13]

Gửi signal:

- `kill()` [14]
- `raise()` [15]
- `sigqueue()` [16]

Các phép toán tập hợp:

- `sigemptyset()` [17]
- `sigfillset()` [18]
- `sigaddset()` [19]
- `sigdelset()` [20]
- `sigismember()` [21]

Khác:

- `sigprocmask()` [22]
- `sigaltstack()` [23]
- `siginterrupt()` [24]
- `sigsetjmp()` [25]
- `siglongjmp()` [26]
- `signal()` [27]

[8] https://man.archlinux.org/man/sigaction.2
[9] https://man.archlinux.org/man/sigwait.3
[10] https://man.archlinux.org/man/sigwaitinfo.2
[11] https://man.archlinux.org/man/sigtimedwait.2
[12] https://man.archlinux.org/man/sigsuspend.2
[13] https://man.archlinux.org/man/sigpending.2
[14] https://man.archlinux.org/man/kill.2
[15] https://man.archlinux.org/man/raise.3
[16] https://man.archlinux.org/man/sigqueue.3
[17] https://man.archlinux.org/man/sigemptyset.3
[18] https://man.archlinux.org/man/sigfillset.3
[19] https://man.archlinux.org/man/sigaddset.3
[20] https://man.archlinux.org/man/sigdelset.3
[21] https://man.archlinux.org/man/sigismember.3
[22] https://man.archlinux.org/man/sigprocmask.2
[23] https://man.archlinux.org/man/sigaltstack.2
[24] https://man.archlinux.org/man/siginterrupt.3
[25] https://man.archlinux.org/man/sigsetjmp.3
[26] https://man.archlinux.org/man/siglongjmp.3
[27] https://man.archlinux.org/man/signal.2

Chapter 4

Signals Phần II

Trong phần này của hướng dẫn, chúng ta sẽ xem xét cách chặn signal, và một số thực hành tốt nhất để viết các hàm signal handler mà không gặp rắc rối nghiêm trọng. Nhưng trước tiên, hãy đi vào một số chi tiết tinh tế.

4.1 Các Trường Hợp Biên

Hãy nói về những thứ kỳ lạ.

Điều gì xảy ra nếu signal handler của bạn đang chạy và một signal khác đến? Một cách hợp lý, theo mặc định, signal thứ hai sẽ bị hoãn lại cho đến khi signal handler kết thúc[1].

OK vậy thì... Điều gì xảy ra nếu đã có một signal bị hoãn và một signal khác đến? Trong trường hợp đó, hai signal bị gộp thành một và chỉ có một cái đến! Nếu bạn nhận được một signal, bạn có thể chắc chắn rằng nó đã được phát một hoặc nhiều lần trước khi handler của bạn thấy nó.

Vì vậy đừng mong đợi một *số lần đếm.* Khi handler của bạn được gọi, tất cả những gì bạn có thể chắc chắn là signal đã được phát ít nhất một lần.

Bây giờ trở lại những thứ thú vị.

4.2 Chặn Signal

Bạn có thể chặn signal không đến. Điều này không loại bỏ signal; nó chỉ giữ chúng lại một lúc. Nếu bạn đang chặn một signal và nó đến, sẽ không có gì xảy ra... cho đến khi bạn bỏ chặn và nó sẽ đến ngay lập tức.

Bạn làm điều này với lệnh gọi `sigprocmask()` [2]. Hàm này thao tác bảng signal bị chặn của từng tiến trình.

Đây là nguyên mẫu:

```
#include <signal.h>

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

Hơi lộn xộn, nhưng `how` đang nói "chặn hay bỏ chặn". Và `set` là tập hợp các signal cần chặn. Cuối cùng, `oset` là tập hợp *trước đó* của các signal bị chặn để bạn có thể chuyển lại sau. Bạn có thể đặt `oset` thành `NULL` nếu bạn không quan tâm đến tập hợp trước đó.

[1] Bạn có thể ghi đè điều này với `SA_NODEFER` trong `sa_flags`, nhưng đó chắc chắn là con đường dẫn đến điên loạn.

[2] Nếu bạn đang dùng POSIX thread, hãy dùng tương đương `pthread_sigmask()` thay thế, để thực hiện điều này trên cơ sở từng thread.

Trường `how` có thể được đặt thành ba thứ tuyệt vời:

`how`	Mô tả
`SIG_BLOCK`	Thêm signal vào danh sách signal đang bị chặn hiện tại.
`SIG_UNBLOCK`	Loại bỏ signal khỏi danh sách signal đang bị chặn hiện tại.
`SIG_SETMASK`	Đặt danh sách signal đang bị chặn hiện tại chính xác thành danh sách này.

Vậy hãy thử trong demo này, `sigblock.c` [3]:

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

int main(void)
{
    sigset_t mask, oldmask;

    // Make a set with SIGINT in it
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);

    // Block everything in that set
    sigprocmask(SIG_BLOCK, &mask, &oldmask);

    // SIGINT is blocked for now!
    puts("Try to ^C out of here! You can't for 5 seconds!");
    sleep(5);

    // Back to how it was before, presumably without blocking SIGINT
    puts("Ok, now you can.");
    sigprocmask(SIG_SETMASK, &oldmask, NULL);

    puts("If you hit ^C, this won't print.");
}
```

Nếu bạn nhấn `CTRL-C` trong khi `sleep()`, bạn sẽ thấy chương trình không bị ngắt. Bạn đang tạo ra `SIGINT`, nhưng chúng bị chặn. Và chúng sẽ được gửi ngay khi bị bỏ chặn, điều xảy ra với `sigprocmask(SIG_SETMASK...` ở dòng 22.

Và vì chúng bị bỏ chặn và chúng ta đang dùng default handler (thoát), tiến trình sẽ thoát ngay sau khi chúng ta bỏ chặn chúng, **trước** khi `puts()` cuối cùng.

4.3 Thực hành Hàm Signal Handler

Vì các hàm signal handler bị hạn chế như vậy, pattern chung mà các lập trình viên thích là để signal handler thực sự không làm gì ngoài việc thông báo cho code chính rằng điều gì đó đã xảy ra, và chỉ vậy thôi.

Hãy xem một biến thể của ví dụ trước. Lưu ý đây **không** phải là cách bạn nên code điều này.

```
volatile sig_atomic_t signal_happened;

void handler(int sig)
{
```

[3] https://beej.us/guide/bgipc/source/examples/sigblock.c

```
    signal_happened = 1;
}

int main(void)
{
    // ... signal setup ...

    while (!signal_happened) { /* spin */ }

    puts("Signal happened!");
    signal_happened = 0;

    // ... etc. ...
}
```

Bạn không muốn làm điều này vì nó chỉ quay vòng ngốn CPU như thể không có ngày mai trong khi chờ signal. Nhưng đó là một ví dụ cơ bản về pattern chung. Chúng ta chỉ cần loại bỏ spin-wait.

Điều này có nghĩa là chúng ta sẽ cho tiến trình chính ngủ theo cách nào đó, ví dụ:

```
while (!signal_happened) { sleep(1000000); }
```

Tốt hơn rồi! Giả sử bạn không chỉ định `SA_RESTART`, `sleep()` sẽ thất bại với `EINTR` ngay khi signal được phát và bạn sẽ thoát khỏi vòng lặp. Đúng là nó thức dậy để kiểm tra mỗi mười một ngày rưỡi, và điều đó dùng một chút CPU, nhưng đó là điều tôi có thể chấp nhận.

Và, như chúng ta đã thấy trước đó, điều tuyệt vời ở đây là signal handler đã không làm gì ngoài việc thực hiện một ghi atomic vào một biến global. Mọi thứ khác được xử lý gọn gàng bởi chương trình, vì vậy chúng ta không phải lo lắng về các ghi không-atomic hoặc race condition.

Nhưng chương trình đó thật *nhàm chán*! Nó không làm gì cả!

Nếu chúng ta muốn ứng dụng làm việc **và** xử lý signal thì sao? Ôi thôi, đừng phát điên.

Đoán xem! Chúng ta có các lựa chọn. Tôi sẽ đưa ra hai ở đây, và bạn thực sự có thể dùng bất kỳ cái nào phù hợp. Cả hai đều giả định bạn đang sử dụng thứ gì đó như `select()` hoặc `poll()` để xử lý các sự kiện I/O bất đồng bộ và đó là thứ điều khiển chương trình của bạn. Hoặc ít nhất, chúng giả định rằng bạn có thể điều chỉnh code để làm điều đó.

Và nếu bạn cần ôn lại, hãy xem *Hướng dẫn Lập trình Mạng của Beej*[4], đặc biệt là các phần về `poll()`[5] và `select()`[6].

4.3.1 Sử dụng Pipe

Nếu bạn đã sử dụng `select()` hoặc `poll()` để chờ đợi các sự kiện, cách tiếp cận này có thể hoạt động cho bạn khá tiện.

Ý tưởng là bạn sẽ tạo một pipe. Tiến trình chính thêm đầu đọc của pipe vào tập hợp file descriptor mà `select()` hoặc `poll()` của nó đang chờ.

Sau đó, khi một signal đến, signal handler ghi một định danh đơn giản vào pipe. Sau đó tiến trình chính sẽ trở về từ `select()` hoặc `poll()` và bạn có thể xem những gì trong pipe. Định danh cho bạn biết signal nào đã được xử lý.

Đây là một đoạn từ chương trình demo `pipesig.c`[7]. Nó chờ nhập văn bản từ `stdin` cũng như chờ thông tin đến trên pipe. (Trong trường hợp này, chúng ta sẽ dùng `poll()`, nhưng `select()` cũng hoạt động tốt như nhau.) Nó khởi động một tiến trình nền phát `SIGUSR1` trên tiến trình cha mỗi vài giây.

[4] https://beej.us/guide/bgnet/
[5] https://beej.us/guide/bgnet/html/split/slightly-advanced-techniques.html#poll
[6] https://beej.us/guide/bgnet/html/split/slightly-advanced-techniques.html#select
[7] https://beej.us/guide/bgipc/source/examples/pipesig.c

```
int pipefd[2];

void handler(int sig)
{
    (void)sig;
    write(pipefd[1], "1", 1);
}
```

Đó là tất cả cho signal handler! Nó chỉ đưa một ký tự ASCII `1` vào pipe. Hết.

Hãy xem cách nó được xử lý (code đã được đơn giản hóa ở đây trong văn bản—xem nguồn đầy đủ để thấy cách nó hoạt động):

```
struct pollfd pollfds[2] = {
    { .fd=0, .events=POLLIN },
    { .fd=pipefd[0], .events=POLLIN },
};

st = poll(pollfds, 2, 0);

// ...

if ((pollfds[0].revents & POLLIN)) {
    if (fgets(line, sizeof line, stdin) == NULL) return;

    int len = strlen(line);
    if (line[len-1] == '\n') line[len-1] = '\0';

    if (strcmp(line, "quit") == 0) return;

    printf("You entered: \"%s\"\n", line);
}

else if ((pollfds[1].revents & POLLIN)) {
    char sigdata[1024];

    int count = read(pipefd[0], sigdata, sizeof sigdata);

    for (int i = 0; i < count; i++)
        if (sigdata[i] == '1')
            printf("SIGUSR1 occurred\n");
}
```

Ở đó chúng ta thiết lập mảng `pollfds` để theo dõi file descriptor `0` (standard input có thể từ bàn phím) và file descriptor `pipefd[0]`, đầu đọc của pipe.

Nếu chúng ta nhận được gì đó từ `stdin`, chúng ta xử lý nó bằng cách in ra. Nếu chúng ta nhận được gì đó trên pipe, chúng ta kiểm tra định danh và in ra điều gì đã xảy ra. (Rõ ràng tôi có một số vấn đề ở đây nếu có hơn 1024 signal xảy ra trước khi tôi thức dậy để xử lý `poll()`, nhưng việc sửa điều đó được để lại như một bài tập cho bạn và môi trường tính toán hiệu năng cao của bạn.)

Đây là một số đầu ra từ một lần chạy mẫu:

```
Enter lines of text, or "quit" to quit.
hi
You entered: "hi"
SIGUSR1 occurred
```

```
This is a long lSIGUSR1 occurred
ine of text
You entered: "This is a long line of text"
SIGUSR1 occurred
quit
Quitting, sending SIGTERM to child
```

Khá đơn giản. Đúng, signal handler đang gọi `write()` và dùng một pipe descriptor global không phải atomic, nhưng chúng ta chỉ đặt pipe descriptor ở đầu lần chạy trước khi signal handler được cài đặt. Và chúng ta không sửa đổi nó sau đó. Vì vậy sự an toàn tương đối được đảm bảo.

4.3.2 Sử dụng `pselect()`

Nếu bạn đã dùng `select()`, đây có thể là cách sạch hơn so với pipe để thông báo cho tiến trình rằng một signal đã xảy ra.

Một hacker Unix thông minh đã nghĩ theo cách này: nếu có một phiên bản của `select()` có thể thức dậy khi một trong số các signal cụ thể được phát? Và nó có thể làm điều này ngoài tất cả việc giám sát file descriptor mà nó thường làm?

Và vì vậy họ đã tạo ra điều đó.

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

int pselect(int nfds,
            fd_set *restrict readfds,
            fd_set *restrict writefds,
            fd_set *restrict errorfds,
            const struct timespec *restrict timeout,
            const sigset_t *restrict sigmask);
```

Trông giống `select()` phải không? Sự khác biệt duy nhất là:

- Timeout là `struct timespec` thay vì `struct timeval`.
- Chúng ta có `sigmask` như tham số cuối.

Trong demo, chúng ta sẽ để `timeout` là `NULL` nên nó không bao giờ hết thời gian, nhưng bạn hoàn toàn có thể thêm vào nếu muốn.

Và `sigmask` nên giữ một tập hợp các signal cần bị chặn trong khi gọi `pselect()` ... cái mà **không** nên bao gồm signal bạn đang xử lý!

Nghe có vẻ vô nghĩa. Hãy xem cách tiếp cận tổng thể mà tiến trình chính sẽ thực hiện:

1. Thiết lập signal handler, giả sử cho `SIGUSR1`.
2. Chặn `SIGUSR1` với `sigprocmask()`.
3. Gọi `pselect()` với `sigmask` **không** bao gồm `SIGUSR1`.
4. Khi `pselect()` trả về, kiểm tra xem có phải do signal không.

Vậy nếu `SIGUSR1` bị chặn, làm sao nó lọt qua được? Đây là phần ma thuật.

`pselect()` lấy `sigmask` bạn truyền vào và đặt signal mask của tiến trình thành nó. Giả sử bạn truyền một tập hợp rỗng. Trong trường hợp đó, không có signal nào bị chặn, và tất cả chúng sẽ lọt qua. Vì vậy trong khi bạn đang gọi `pselect()`, `SIGUSR1` không bị chặn và nó có thể hoạt động.

Và rồi (cho phần ma thuật kia), sau khi signal đến, `pselect()` *khôi phục* signal mask của tiến trình về trạng thái trước đó.

Kết quả thực tế của tất cả điều này là tiến trình của bạn đã chặn `SIGUSR1` ở mọi nơi *ngoại trừ* trong khi `pselect()` đang được gọi! Điều này cho bạn quyền kiểm soát trung tâm về thời điểm xử lý signal và phải làm gì.

Pseudocode cho `pselect()` trông gần như thế này:

```
pselect(readset, timeout, sigmask):
    sigprocmask(SIG_SETMASK, sigmask, oldmask);
    select(readset, timeout)
    sigprocmask(SIG_SETMASK, oldmask, NULL);
```

Tính năng chính là, vì đây là một syscall, tất cả điều này xảy ra nguyên tử từ góc nhìn của chúng ta. Chúng ta không thể viết điều này trong user space mà không bị racy.

Hãy xem ví dụ `pselect.c` [8], giống như ví dụ `poll()` ở trên, ngoại trừ nó dùng `pselect()`. Đây là handler.

```
volatile sig_atomic_t sigusr1_happened;

void handler(int sig)
{
    sigusr1_happened = 1;
}
```

Một lần nữa, ngắn gọn. Chúng ta chỉ đặt một cờ atomic global cho biết signal đã xảy ra. Hãy xem phần code chính (một lần nữa, đã chỉnh sửa cho ngắn gọn):

```
sigset_t mask, oldmask;
sigemptyset(&mask);
sigaddset(&mask, SIGUSR1);
sigprocmask(SIG_BLOCK, &mask, &oldmask);

// ...

FD_ZERO(&readfds);
FD_SET(0, &readfds);
st = pselect(1, &readfds, NULL, NULL, NULL, &oldmask);

if (st == -1 && errno == EINTR) {
    if (sigusr1_happened) {
        sigusr1_happened = 0;
        printf("SIGUSR1 occurred\n");
    }

} else if (st > 0 && FD_ISSET(0, &readfds)) {
    if (fgets(line, sizeof line, stdin) == NULL)
        return;

    int len = strlen(line);
    if (line[len-1] == '\n') line[len-1] = '\0';

    if (strcmp(line, "quit") == 0)
        return;

    printf("You entered: \"%s\"\n", line);
}
```

Một vài điều cần giải thích ở đây.

- Chúng ta tạo một `mask` mới với `SIGUSR1` trong đó và chặn signal đó.

[8] https://beej.us/guide/bgipc/source/examples/pselect.c

- Chúng ta giữ mask cũ (trong trường hợp này là tập hợp rỗng), và chúng ta sẽ dùng nó làm tập hợp để chặn với `pselect()`.
- Chúng ta thêm file descriptor `0` (`stdin`) vào `readfds` để `pselect()` sẽ trả về nếu chúng ta gõ gì đó.
- Chúng ta gọi `pselect()`.
- Nếu nó trả về `-1` và `errno` là `EINTR`, nghĩa là `pselect()` bị ngắt bởi một signal! Chúng ta sau đó kiểm tra global của mình để xem có phải là signal của chúng ta không.
- Nếu nó trả về dương (`0` nghĩa là hết thời gian), phải là một trong các file descriptor của chúng ta. Chúng ta kiểm tra xem có phải file descriptor `0` (`stdin`) không, và nếu vậy, chúng ta đọc dữ liệu với `fgets()`.

Vì vậy, một lần nữa, chúng ta có phần xử lý signal ở vòng lặp chính nơi nó nằm trong tầm kiểm soát của chúng ta và chúng ta không phải đối phó với các vấn đề đồng thời khó chịu.

Phần `oldmask` đó khá kỳ lạ. Bằng cách làm như vậy, về cơ bản chúng ta đang nói với `pselect()` rằng chúng ta chỉ muốn được thông báo khi `SIGUSR1` đến và không phải signal nào khác. Chúng ta chặn tất cả những gì đã bị chặn trước khi chúng ta thêm `SIGUSR1` vào tập hợp. (Trong trường hợp này, không có signal nào khác, vì vậy `oldmask` là tập hợp rỗng.)

4.4 Kết luận

Vậy là đó. Một số kỹ thuật lạ lùng mà chúng ta có để thực sự xử lý đúng các POSIX signal. Những điểm chính là bạn có thể xử lý tất cả các loại signal và bạn có thể chặn việc gửi chúng. Ngoài ra bạn chỉ nên sửa đổi các biến global trong signal handler nếu chúng là atomic. Và nếu các biến global được ghi vào ở bất kỳ đâu trong khi handler đang hoạt động, chúng cũng nên là atomic.

Và nếu bạn muốn xử lý signal đúng cách, thực sự nên xử lý ở vòng lặp chính trừ khi bạn chỉ bỏ qua chúng. Và bạn có thể dùng pipe hoặc `pselect()` để hỗ trợ điều này.

Lập trình an toàn, và chú ý những con rồng!

Chapter 5

Pipes (Đường ống)

Không có hình thức IPC nào đơn giản hơn pipe. Được triển khai trên mọi hương vị Unix, `pipe()` và `fork()` tạo nên chức năng đằng sau " `|` " trong " `ls | more` ". Chúng hữu ích một cách vừa phải cho những thứ hay ho, nhưng là cách học tốt về các phương pháp IPC cơ bản.

Vì chúng quá quá dễ, tôi sẽ không dành nhiều thời gian cho chúng. Chúng ta sẽ chỉ xem qua vài ví dụ.

5.1 "Những cái pipe này sạch đấy!"

Đợi đã! Không nhanh vậy. Tôi có thể cần định nghĩa "file descriptor" ở điểm này. Để tôi nói thế này: bạn biết về " `FILE*` " từ `stdio.h` chứ? Bạn biết cách bạn có tất cả những hàm hay ho như `fopen()`, `fclose()`, `fwrite()`, v.v.? Thực ra, những hàm đó là các hàm cấp cao được triển khai bằng *file descriptor*, sử dụng các system call như `open()`, `creat()`, `close()`, và `write()`. File descriptor đơn giản là các `int` tương tự với `FILE*` trong `stdio.h`.

Ví dụ, `stdin` là file descriptor "0", `stdout` là "1", và `stderr` là "2". Tương tự, bất kỳ file nào bạn mở bằng `fopen()` đều có file descriptor riêng, mặc dù chi tiết này bị ẩn khỏi bạn. (File descriptor này có thể được lấy từ `FILE*` bằng cách dùng macro `fileno()` từ `stdio.h`.)

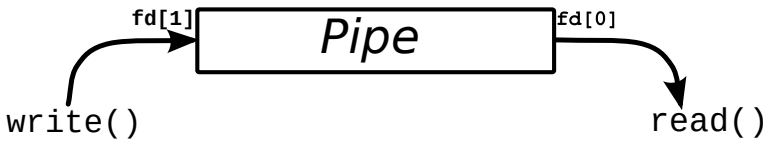

Figure 5.1: Cách một pipe được tổ chức.

Về cơ bản, một lần gọi hàm `pipe()` trả về một cặp file descriptor. Một trong số các descriptor này được kết nối với đầu ghi của pipe, và cái kia được kết nối với đầu đọc. Bất cứ thứ gì có thể được ghi vào pipe, và đọc từ đầu kia theo thứ tự nó đến. Trên nhiều hệ thống, pipe sẽ đầy sau khi bạn ghi khoảng 10K vào chúng mà không đọc gì ra.

Như một ví dụ vô dụng[1], chương trình sau tạo, ghi vào, và đọc từ một pipe.

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

int main(void)
{
```

[1] https://beej.us/guide/bgipc/source/examples/pipe1.c

```
    int pfds[2];
    char buf[30];

    if (pipe(pfds) == -1) {
        perror("pipe");
        exit(1);
    }

    printf("writing to file descriptor #%d\n", pfds[1]);
    write(pfds[1], "test", 5);
    printf("reading from file descriptor #%d\n", pfds[0]);
    read(pfds[0], buf, 5);
    printf("read \"%s\"\n", buf);

    return 0;
}
```

Như bạn có thể thấy, `pipe()` nhận một mảng hai `int` làm đối số. Giả sử không có lỗi, nó kết nối hai file descriptor và trả về chúng trong mảng. Phần tử đầu tiên của mảng là đầu đọc của pipe, phần tử thứ hai là đầu ghi.

5.2 `fork()` và `pipe()`—bạn có quyền lực!

Từ ví dụ trên, khá khó thấy những thứ này có thể hữu ích như thế nào. Thực ra, vì đây là tài liệu IPC, hãy đưa `fork()` vào và xem điều gì xảy ra. Giả sử bạn là một đặc vụ liên bang hàng đầu được giao nhiệm vụ làm cho một tiến trình con gửi từ "test" đến tiến trình cha. Không hào hứng lắm, nhưng không ai bảo rằng khoa học máy tính sẽ là X-Files, Mulder.

Đầu tiên, chúng ta sẽ để tiến trình cha tạo một pipe. Thứ hai, chúng ta sẽ `fork()`. Giờ, trang man `fork()` cho biết rằng tiến trình con sẽ nhận được bản sao của tất cả file descriptor của cha, và điều này bao gồm bản sao của các file descriptor của pipe. *Alors*, tiến trình con sẽ có thể gửi thứ gì đó đến đầu ghi của pipe, và tiến trình cha sẽ nhận nó từ đầu đọc. Như thế này[2]:

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

int main(void)
{
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
```

[2] https://beej.us/guide/bgipc/source/examples/pipe2.c

```
        wait(NULL);
    }

    return 0;
}
```

Xin lưu ý, chương trình của bạn nên có nhiều kiểm tra lỗi hơn của tôi. Tôi đôi khi bỏ qua nó để giúp mọi thứ rõ ràng hơn.

Dù sao, ví dụ này giống như ví dụ trước, ngoại trừ bây giờ chúng ta `fork()` ra một tiến trình mới và để nó ghi vào pipe, trong khi tiến trình cha đọc từ nó. Kết quả đầu ra sẽ tương tự như sau:

```
PARENT: reading from pipe
 CHILD: writing to the pipe
 CHILD: exiting
PARENT: read "test"
```

Trong trường hợp này, tiến trình cha cố đọc từ pipe trước khi tiến trình con ghi vào đó. Khi điều này xảy ra, tiến trình cha được gọi là *block*, hay ngủ, cho đến khi dữ liệu đến để đọc. Có vẻ tiến trình cha đã cố đọc, đi ngủ, tiến trình con ghi và thoát, và tiến trình cha thức dậy và đọc dữ liệu.

Hoan hô!! Bạn vừa thực hiện một số giao tiếp liên tiến trình! Đơn giản đến mức kinh hoàng phải không? Tôi cá là bạn vẫn đang nghĩ rằng không có nhiều ứng dụng cho `pipe()` và, thực ra, bạn có thể đúng. Các hình thức IPC khác thường hữu ích hơn và thường thú vị hơn.

5.3 Tìm kiếm Pipe như chúng ta biết

Trong nỗ lực khiến bạn nghĩ rằng pipe thực sự là những thú đáng tin cậy, tôi sẽ cho bạn một ví dụ về việc sử dụng `pipe()` trong một tình huống quen thuộc hơn. Thử thách: triển khai "`ls | wc -l`" trong C.

Điều này yêu cầu sử dụng thêm một vài hàm mà bạn có thể chưa từng nghe đến: `exec()` và `dup()`. Họ hàm `exec()` thay thế tiến trình đang chạy hiện tại bằng tiến trình nào đó được truyền vào `exec()`. Đây là hàm chúng ta sẽ dùng để chạy `ls` và `wc -l`. `dup()` nhận một file descriptor đang mở và tạo một bản sao (bản nhân đôi) của nó. Đây là cách chúng ta sẽ kết nối standard output của `ls` với standard input của `wc`. Thấy đó, stdout của `ls` chảy vào pipe, và stdin của `wc` chảy vào từ pipe. Pipe nằm ngay ở giữa!

Dù sao, đây là code[3]:

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

int main(void)
{
    int pfds[2];

    pipe(pfds);

    if (!fork()) {
        close(1);       /* close normal stdout */
        dup(pfds[1]);   /* make stdout same as pfds[1] */
        close(pfds[0]); /* we don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);       /* close normal stdin */
```

[3] https://beej.us/guide/bgipc/source/examples/pipe3.c

```
        dup(pfds[0]);   /* make stdin same as pfds[0] */
        close(pfds[1]); /* we don't need this */
        execlp("wc", "wc", "-l", NULL);
    }

    return 0;
}
```

Tôi sẽ ghi chú thêm về tổ hợp `close()` / `dup()` vì nó khá kỳ lạ. `close(1)` giải phóng file descriptor 1 (standard output). `dup(pfds[1])` tạo bản sao của đầu ghi của pipe trong file descriptor đầu tiên có sẵn, là "1", vì chúng ta vừa đóng cái đó. Theo cách này, bất cứ thứ gì `ls` ghi vào standard output (file descriptor 1) sẽ thay vào đó đi vào `pfds[1]` (đầu ghi của pipe). Phần code `wc` hoạt động theo cách tương tự, ngoại trừ ngược lại.

5.4 Tóm tắt

Không có nhiều điều để nói về một chủ đề đơn giản như vậy. Thực ra, hầu như chẳng có gì. Có lẽ cách dùng tốt nhất của pipe là cách bạn quen thuộc nhất: gửi standard output của một lệnh đến standard input của lệnh khác. Đối với các mục đích sử dụng khác, nó khá hạn chế và thường có các kỹ thuật IPC khác hoạt động tốt hơn.

Chapter 6

FIFOs

Một FIFO ("First In, First Out", đọc là "Fy-Foh") đôi khi còn được biết đến là *named pipe* (pipe có tên). Tức là, nó giống như một pipe, ngoại trừ nó có tên! Trong trường hợp này, tên đó là tên của một file mà nhiều tiến trình có thể `open()` và đọc ghi vào.

Khía cạnh sau này của FIFO được thiết kế để khắc phục một trong những nhược điểm của pipe thông thường: bạn không thể nắm lấy một đầu của pipe thông thường được tạo bởi một tiến trình không liên quan. Thấy đó, nếu tôi chạy hai bản sao riêng lẻ của một chương trình, chúng đều có thể gọi `pipe()` bao nhiêu tùy thích mà vẫn không thể nói chuyện với nhau. (Đây là vì bạn phải `pipe()`, rồi `fork()` để có một tiến trình con có thể giao tiếp với cha thông qua pipe.) Tuy nhiên, với FIFO, mỗi tiến trình không liên quan chỉ cần `open()` pipe và truyền dữ liệu qua đó.

6.1 Một FIFO Mới Ra Đời

Vì FIFO thực sự là một file trên đĩa, bạn phải làm một số thứ cầu kỳ để tạo nó. Không khó lắm. Bạn chỉ cần gọi `mkfifo()` với các đối số thích hợp. Đây là một lệnh gọi `mkfifo()` tạo ra một FIFO:

```
mkfifo("myfifo", 0644);
```

Trong ví dụ trên, file FIFO sẽ được gọi là "`myfifo`". Đối số thứ hai đặt quyền truy cập cho file đó (octal 644, hay `rw-r--r--`) cũng có thể được đặt bằng cách OR các macro từ `sys/stat.h`. Quyền này giống như quyền bạn sẽ đặt bằng lệnh `chmod`.

(Ghi chú thêm: một FIFO cũng có thể được tạo từ dòng lệnh bằng lệnh Unix `mkfifo`.)

6.1.1 Ghi chú Lịch sử: `mknod`

Cách gốc để tạo một FIFO là với `mknod()`, nhưng cách này đã bị loại bỏ. Hiện tại, hai lệnh gọi này là tương đương:

```
mknod("myfifo", S_IFIFO | 0644, 0);   // old way
mkfifo("myfifo", 0644);               // new way
```

Trong trường hợp lệnh gọi `mknod()`, trước đây bạn phải làm thêm một chút công việc bằng cách chỉ định chế độ tạo trong đối số thứ hai (OR thêm S_IFIFO) và số thiết bị như là đối số cuối. Đối số cuối này bị bỏ qua khi tạo FIFO, vì vậy bạn có thể đặt bất cứ thứ gì vào đó.

Nhưng bạn nên dùng `mkfifo()` để tạo FIFO nếu hệ thống của bạn hỗ trợ.

6.2 Người sản xuất và Người tiêu thụ

Sau khi FIFO được tạo, một tiến trình có thể khởi động và mở nó để đọc hoặc ghi bằng cách dùng system call `open()` tiêu chuẩn.

Vì tiến trình dễ hiểu hơn khi bạn có một ít code trong bụng, tôi sẽ trình bày ở đây hai chương trình sẽ gửi dữ liệu qua FIFO. Một là `speak.c` gửi dữ liệu qua FIFO, và cái kia được gọi là `tick.c`, vì nó hút dữ liệu ra khỏi FIFO.

Đây là `speak.c`[1]:

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

#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mkfifo(FIFO_NAME, 0644);

    printf("waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY);
    printf("got a reader--type some stuff\n");

    while (gets(s), !feof(stdin)) {
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("speak: wrote %d bytes\n", num);
    }

    return 0;
}
```

`speak` làm là tạo FIFO, sau đó cố `open()` nó. Bây giờ, điều sẽ xảy ra là lệnh gọi `open()` sẽ *block* cho đến khi một tiến trình khác mở đầu kia của pipe để đọc. (Có cách khắc phục điều này—xem `O_NDELAY`, bên dưới.) Tiến trình đó là `tick.c`[2], hiển thị ở đây:

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

[1] https://beej.us/guide/bgipc/source/examples/speak.c
[2] https://beej.us/guide/bgipc/source/examples/tick.c

```
#define FIFO_NAME "american_maid"

int main(void)
{
    char s[300];
    int num, fd;

    mkfifo(FIFO_NAME, 0644);

    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");

    do {
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("tick: read %d bytes: \"%s\"\n", num, s);
        }
    } while (num > 0);

    return 0;
}
```

Giống như `speak.c`, `tick` sẽ block trên `open()` nếu không có ai ghi vào FIFO. Ngay khi ai đó mở FIFO để ghi, `tick` sẽ bừng tỉnh.

Thử đi! Khởi động `speak` và nó sẽ block cho đến khi bạn khởi động `tick` trong một cửa sổ khác. (Ngược lại, nếu bạn khởi động `tick`, nó sẽ block cho đến khi bạn khởi động `speak` trong cửa sổ khác.) Gõ thoải mái trong cửa sổ `speak` và `tick` sẽ hút hết tất cả.

Bây giờ, thoát ra khỏi `speak`. Chú ý điều gì xảy ra: `read()` trong `tick` trả về 0, báo hiệu EOF. Theo cách này, đầu đọc có thể biết khi nào tất cả người ghi đã đóng kết nối của họ đến FIFO. “Cái gì?” bạn hỏi “Có thể có nhiều người ghi vào cùng một pipe không?” Tất nhiên! Điều đó có thể rất hữu ích, bạn biết đó. Có lẽ tôi sẽ chỉ cho bạn sau trong tài liệu này cách điều này có thể được khai thác.

Nhưng bây giờ, hãy kết thúc chủ đề này bằng cách xem điều gì xảy ra khi bạn thoát ra khỏi `tick` trong khi `speak` đang chạy. “Broken Pipe”! Điều đó nghĩa là gì? Thực ra, điều đã xảy ra là khi tất cả người đọc của một FIFO đóng và người ghi vẫn còn mở, người ghi sẽ nhận signal SIGPIPE vào lần tiếp theo nó cố `write()`. Default signal handler cho signal này in ra “Broken Pipe” và thoát. Tất nhiên, bạn có thể xử lý điều này lịch sự hơn bằng cách bắt SIGPIPE thông qua lệnh gọi `signal()`.

Cuối cùng, điều gì xảy ra nếu bạn có nhiều người đọc? Thực ra, những điều kỳ lạ xảy ra. Đôi khi một trong các người đọc nhận được tất cả mọi thứ. Đôi khi nó xen kẽ giữa các người đọc. Tại sao bạn muốn có nhiều người đọc vậy?

6.3 `O_NDELAY`! Tôi KHÔNG THỂ BỊ DỪNG!

Trước đó, tôi đã đề cập rằng bạn có thể khắc phục lệnh gọi `open()` đang block nếu không có người đọc hoặc người ghi tương ứng. Cách để làm điều này là gọi `open()` với cờ `O_NDELAY` được đặt trong đối số chế độ:

```
fd = open(FIFO_NAME, O_WRONLY | O_NDELAY);
```

Điều này sẽ khiến `open()` trả về `-1` nếu không có tiến trình nào đang mở file để đọc.

Tương tự, bạn có thể mở tiến trình đọc bằng cờ `O_NDELAY`, nhưng điều này có hiệu ứng khác: tất cả các lần cố `read()` từ pipe sẽ đơn giản trả về `0` byte đọc nếu không có dữ liệu trong pipe. (Tức là, `read()` sẽ không còn block cho đến khi có một số dữ liệu trong pipe.) Lưu ý rằng bạn không còn có thể biết liệu `read()` có trả về `0` vì không có dữ liệu trong pipe, hay vì người ghi đã thoát. Đây là cái giá của quyền lực, nhưng lời khuyên của tôi là hãy cố gắng gắn bó với blocking bất cứ khi nào có thể.

6.4 Xen kẽ Dữ liệu

Điều gì xảy ra nếu bạn có nhiều người ghi đang đổ dữ liệu vào pipe cùng một lúc? Nó có thể bị xen kẽ không?

Có thể! Tùy thuộc vào lượng dữ liệu bạn đổ vào trong một lần gọi `write()`. Miễn là bạn không vượt quá `PIPE_BUF` byte trong `write()`, nó sẽ là atomic[3]. Và điều đó tốt!

Điều đó nói rằng, không có gì bắt buộc rằng các lần gọi `read()` tương ứng lấy ra từng phần dữ liệu riêng lẻ. Chúng ta có thể có điều này xảy ra:

```
write "Foo"
write "bar"
```

Và rồi một lần đọc cho chúng ta:

```
read "Foobar"
```

Hoặc có thể lần đọc bị ngắt!

```
read "Foob"
read "ar"
```

Vì vậy ngay cả khi bạn có các lần ghi atomic, bạn sẽ cần một số cấu trúc bổ sung ở đầu đọc để đảm bảo bạn đang lấy đúng dữ liệu ra phía kia. Đôi khi điều này được thực hiện bằng cách thêm tiền tố dữ liệu bằng độ dài hoặc có các tin nhắn có độ dài cố định.

Nhưng trong mọi trường hợp, bạn sẽ phải đảm bảo rằng bạn có một tin nhắn hoàn chỉnh, hoặc bạn sẽ phải gọi `read()` lại cho đến khi có.

6.5 Ghi chú Kết thúc

Có tên của pipe ngay trên đĩa chắc chắn làm cho mọi thứ dễ dàng hơn phải không? Các tiến trình không liên quan có thể giao tiếp qua pipe! (Đây là khả năng mà bạn sẽ thấy mình ước gì nếu bạn dùng pipe thông thường quá lâu.) Dẫu vậy, chức năng của pipe có thể không hoàn toàn là những gì bạn cần cho các ứng dụng của mình. Hàng đợi tin nhắn có thể phù hợp hơn với bạn, nếu hệ thống của bạn hỗ trợ chúng.

[3]POSIX nói `PIPE_BUF` sẽ ít nhất 512 byte. Vì vậy đó là vùng an toàn di động của bạn.

Chapter 7

Khóa File

Khóa file cung cấp một cơ chế rất đơn giản nhưng cực kỳ hữu ích để phối hợp các truy cập file. Trước khi tôi bắt đầu trình bày chi tiết, hãy để tôi tiết lộ cho bạn một số bí mật về khóa file:

Có hai loại cơ chế khóa: bắt buộc (mandatory) và tư vấn (advisory). Các hệ thống bắt buộc sẽ thực sự ngăn các lệnh `read()` và `write()` vào file. Một số hệ thống Unix hỗ trợ chúng. Tuy nhiên, tôi sẽ bỏ qua chúng trong toàn bộ tài liệu này, thay vào đó chỉ nói về advisory lock. Với hệ thống advisory lock, các tiến trình vẫn có thể đọc và ghi từ một file trong khi nó bị khóa. Vô dụng không? Không hẳn, vì có cách để một tiến trình kiểm tra sự tồn tại của một khóa trước khi đọc hoặc ghi. Thấy đó, đây là một loại hệ thống khóa *hợp tác*. Điều này đủ dễ dàng cho hầu hết tất cả các trường hợp cần khóa file.

Vì điều đó đã được giải thích xong, bất cứ khi nào tôi đề cập đến khóa từ đây trở đi trong tài liệu này, tôi đề cập đến advisory lock. Vậy thôi.

Bây giờ, hãy để tôi phân tích khái niệm khóa thêm một chút. Có hai loại khóa (advisory!): read lock (khóa đọc) và write lock (khóa ghi) (còn được gọi là shared lock và exclusive lock tương ứng.) Cách read lock hoạt động là chúng không can thiệp vào các read lock khác. Ví dụ, nhiều tiến trình có thể khóa một file để đọc cùng một lúc. Tuy nhiên, khi một tiến trình có write lock trên một file, không có tiến trình nào khác có thể kích hoạt read lock hoặc write lock cho đến khi nó được giải phóng. Một cách dễ hiểu là có thể có nhiều người đọc đồng thời, nhưng chỉ có thể có một người ghi tại một thời điểm.

Một điều cuối cùng trước khi bắt đầu: có nhiều cách để khóa file trong các hệ thống Unix. System V thích `lockf()`, mà cá nhân tôi nghĩ là tệ. Các hệ thống tốt hơn hỗ trợ `flock()` cung cấp kiểm soát tốt hơn đối với khóa, nhưng vẫn còn thiếu một số cách. Để tính di động và đầy đủ, tôi sẽ nói về cách khóa file bằng `fcntl()`. Tuy nhiên tôi khuyến khích bạn sử dụng một trong các hàm kiểu `flock()` cấp cao hơn nếu phù hợp với nhu cầu của bạn, nhưng tôi muốn trình bày một cách di động về toàn bộ phạm vi quyền lực mà bạn có trong tầm tay. (Nếu hệ thống System V Unix của bạn không hỗ trợ `fcntl()` kiểu POSIX, bạn sẽ phải đối chiếu thông tin sau đây với trang man `lockf()` của mình.)

7.1 Đặt khóa

Hàm `fcntl()` làm hầu như mọi thứ trên hành tinh, nhưng chúng ta sẽ chỉ dùng nó để khóa file. Đặt khóa bao gồm điền vào một `struct flock` (khai báo trong `fcntl.h`) mô tả loại khóa cần thiết, `open()` file với chế độ phù hợp, và gọi `fcntl()` với các đối số thích hợp, *comme ça*:

```
struct flock fl = {
    .l_type   = F_WRLCK,  /* F_RDLCK, F_WRLCK, F_UNLCK      */
    .l_whence = SEEK_SET, /* SEEK_SET, SEEK_CUR, SEEK_END   */
    .l_start  = 0,        /* Offset from l_whence           */
    .l_len    = 0,        /* length, 0 = to EOF             */
    // .l_pid             /* PID holding lock; F_RDLCK only */
};
int fd;
```

```
fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl);  /* F_GETLK, F_SETLK, F_SETLKW */
```

Điều gì vừa xảy ra? Hãy bắt đầu với `struct flock` vì các trường trong đó được dùng để *mô tả* hành động khóa đang diễn ra. Đây là một số định nghĩa trường:

Trường	Mô tả
`l_type`	Đây là nơi bạn chỉ định loại khóa bạn muốn đặt. Nó là `F_RDLCK`, `F_WRLCK`, hoặc `F_UNLCK` nếu bạn muốn đặt read lock, write lock, hoặc xóa khóa, tương ứng.
`l_whence`	Trường này xác định điểm bắt đầu của trường `l_start` (giống như offset cho offset). Nó có thể là `SEEK_SET`, `SEEK_CUR`, hoặc `SEEK_END`, cho đầu file, vị trí file hiện tại, hoặc cuối file.
`l_start`	Đây là offset bắt đầu tính theo byte của khóa, tương đối với `l_whence`.
`l_len`	Đây là độ dài của vùng khóa tính theo byte (bắt đầu từ `l_start` tương đối với `l_whence`).
`l_pid`	Process ID của tiến trình đang giữ khóa. Được kernel đặt khi dùng lệnh F_RDLCK.

Trong ví dụ của chúng ta, chúng ta nói với nó để tạo khóa loại `F_WRLCK` (write lock), bắt đầu tương đối với `SEEK_SET` (đầu file), offset `0`, độ dài `0` (giá trị zero có nghĩa là "khóa đến cuối file"), với PID được đặt thành `getpid()`.

Bước tiếp theo là `open()` file, vì `flock()` cần một file descriptor của file đang bị khóa. Lưu ý rằng khi bạn mở file, bạn cần mở nó trong cùng *chế độ* như bạn đã chỉ định trong khóa, như được hiển thị trong bảng bên dưới. Nếu bạn mở file trong chế độ sai cho một loại khóa nhất định, `fcntl()` sẽ trả về `-1` và `errno` sẽ được đặt thành `EBADF`.

`.l_type`	Chế độ
`F_RDLCK`	`O_RDONLY` hoặc `O_RDWR`
`F_WRLCK`	`O_WRONLY` hoặc `O_RDWR`

Cuối cùng, lệnh gọi `fcntl()` thực sự đặt, xóa, hoặc lấy khóa. Đối số thứ hai (`cmd`) của `fcntl()` cho biết phải làm gì với dữ liệu được truyền vào trong `struct flock`. Danh sách sau tóm tắt những gì mỗi `cmd` của `fcntl()` thực hiện:

`cmd`	Mô tả
`F_SETLKW`	Đối số này yêu cầu `fcntl()` cố lấy khóa được yêu cầu trong cấu trúc `struct flock`. Nếu không thể lấy khóa (vì ai đó khác đã khóa rồi), `fcntl()` sẽ đợi (block) cho đến khi khóa được giải phóng, sau đó sẽ tự đặt khóa. Đây là lệnh rất hữu ích. Tôi dùng nó mọi lúc.
`F_SETLK`	Hàm này gần giống với `F_SETLKW`. Sự khác biệt duy nhất là hàm này sẽ không đợi nếu không thể lấy khóa. Nó sẽ trả về ngay với `-1`. Hàm này có thể được dùng để xóa khóa bằng cách đặt trường `l_type` trong `struct flock` thành `F_UNLCK`.
`F_GETLK`	Nếu bạn chỉ muốn kiểm tra xem có khóa không, nhưng không muốn đặt khóa, bạn có thể dùng lệnh này. Nó tìm qua tất cả các khóa file cho đến khi tìm thấy một cái xung đột với khóa bạn chỉ định trong `struct flock`. Sau đó nó sao chép thông tin khóa xung đột vào `struct` và trả về cho bạn. Nếu không tìm thấy khóa xung đột, `fcntl()` trả về `struct` như bạn đã truyền vào, ngoại trừ đặt trường `l_type` thành `F_UNLCK`.

Trong ví dụ trên của chúng ta, chúng ta gọi `fcntl()` với `F_SETLKW` như đối số, vì vậy nó block cho đến khi có thể đặt khóa, rồi đặt nó và tiếp tục.

7.2 Xóa khóa

Ôi! Sau tất cả những thứ khóa ở trên, đã đến lúc để làm điều gì đó dễ: mở khóa! Thực ra, điều này đơn giản hơn khi so sánh. Tôi sẽ chỉ tái sử dụng ví dụ đầu tiên đó và thêm code để mở khóa nó ở cuối:

```
struct flock fl = {
    .l_type   = F_WRLCK,  /* F_RDLCK, F_WRLCK, F_UNLCK      */
    .l_whence = SEEK_SET, /* SEEK_SET, SEEK_CUR, SEEK_END   */
    .l_start  = 0,        /* Offset from l_whence           */
    .l_len    = 0,        /* length, 0 = to EOF             */
    // .l_pid             /* PID holding lock; F_RDLCK only */
};
int fd;

fd = open("filename", O_WRONLY);  /* get the file descriptor */
fcntl(fd, F_SETLKW, &fl);  /* set the lock, waiting if necessary */
.
.
.
fl.l_type = F_UNLCK;     /* tell it to unlock the region */
fcntl(fd, F_SETLK, &fl); /* set the region to unlocked   */
```

Bây giờ, tôi đã để code khóa cũ trong đó để tương phản cao, nhưng bạn có thể thấy rằng tôi chỉ thay đổi trường `l_type` thành `F_UNLCK` (để các trường khác hoàn toàn không thay đổi!) và gọi `fcntl()` với `F_SETLK` như lệnh. Dễ thôi!

7.3 Một chương trình demo

Ở đây, tôi sẽ bao gồm một chương trình demo, `lockdemo.c`, đợi người dùng nhấn return, sau đó khóa nguồn của nó, đợi một lần return khác, rồi mở khóa. Bằng cách chạy chương trình này trong hai (hoặc nhiều hơn) cửa sổ, bạn có thể thấy cách các chương trình tương tác trong khi đợi khóa.

Về cơ bản, cách dùng là: nếu bạn chạy `lockdemo` mà không có đối số dòng lệnh, nó sẽ cố lấy write lock (`F_WRLCK`) trên nguồn của nó (`lockdemo.c`). Nếu bạn khởi động nó với bất kỳ đối số dòng lệnh nào, nó sẽ cố lấy read lock (`F_RDLCK`) trên nó.

Đây là mã nguồn[1]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
        struct flock fl = {
            .l_type = F_WRLCK,
            .l_whence = SEEK_SET,
            .l_start = 0,
            .l_len = 0,
        };
```

[1] https://beej.us/guide/bgipc/source/examples/lockdemo.c

```
    int fd;

    if (argc > 1)
        fl.l_type = F_RDLCK;

    if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
        perror("open");
        exit(1);
    }

    printf("Press <RETURN> to try to get lock: ");
    getchar();
    printf("Trying to get lock...");

    if (fcntl(fd, F_SETLKW, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("got lock\n");
    printf("Press <RETURN> to release lock: ");
    getchar();

    fl.l_type = F_UNLCK;  /* set to unlock same region */

    if (fcntl(fd, F_SETLK, &fl) == -1) {
        perror("fcntl");
        exit(1);
    }

    printf("Unlocked.\n");

    close(fd);

    return 0;
}
```

Biên dịch thằng đó lên và bắt đầu mày mò với nó trong vài cửa sổ. Lưu ý rằng khi một `lockdemo` có read lock, các instance khác của chương trình có thể lấy read lock của riêng chúng mà không có vấn đề gì. Chỉ khi write lock được lấy thì các tiến trình khác mới không thể lấy khóa bất kỳ loại nào.

Một điều nữa cần lưu ý là bạn không thể lấy write lock nếu có bất kỳ read lock nào trên cùng vùng của file. Tiến trình đang đợi lấy write lock sẽ đợi cho đến khi tất cả các read lock được giải phóng. Một hệ quả của điều này là bạn có thể tiếp tục thêm read lock (vì read lock không ngăn các tiến trình khác lấy read lock) và bất kỳ tiến trình nào đang đợi write lock sẽ ngồi đó và chết đói. Không có quy tắc nào ở bất kỳ đâu ngăn bạn thêm nhiều read lock hơn nếu có một tiến trình đang đợi write lock. Bạn phải cẩn thận.

Trong thực tế, bạn có thể sẽ chủ yếu dùng write lock để đảm bảo truy cập độc quyền vào file trong một thời gian ngắn trong khi nó đang được cập nhật; đó là cách dùng phổ biến nhất của khóa theo những gì tôi đã thấy. Và tôi đã thấy tất cả…thực ra tôi đã thấy một cái…một cái nhỏ …một hình ảnh—thực ra tôi đã nghe về chúng.

7.4 Tóm tắt

Khóa thật tuyệt. Đôi khi, tuy nhiên, bạn có thể cần kiểm soát nhiều hơn đối với các tiến trình trong một tình huống nhà sản xuất-người tiêu thụ. Vì lý do này, nếu không có lý do nào khác, bạn nên xem tài liệu về semaphore System V (hoặc POSIX, thực ra; chúng không giống nhau) nếu hệ thống của bạn hỗ

trợ loài thú đó. Chúng cung cấp một tương đương mở rộng hơn và ít nhất là ngang bằng về chức năng với file lock.

Chapter 8

Hàng Đợi Tin Nhắn POSIX

Ngày xưa, chúng ta chỉ có hàng đợi tin nhắn System V, nhưng những người bạn tốt bụng tại POSIX[1] đã chuẩn hóa những thứ này một phần để ta có thể sử dụng chúng một cách khả chuyển hơn.

Và vậy là hôm nay chúng ta đang ở đây trong năm huy hoàng %YEAR% và sắp chứng kiến sức mạnh hủy diệt của trạm chiến đấu đã hoàn toàn đi vào hoạt động. [*Tiếng thở của Darth Vader*].

Xin lỗi. Tôi bị cuốn vào rồi. Chúng ta đang làm gì thế nhỉ?

8.1 Hàng Đợi Tin Nhắn Là Gì?

Nói chung, ta muốn có thể gửi các *tin nhắn* (những khối byte tùy ý) vào *một cái gì đó*, và sau đó để các tiến trình khác nhận những tin nhắn đó.

Và có lẽ ta muốn chúng được sắp xếp theo một thứ tự nào đó, chẳng hạn *vào trước ra trước* như những thứ FIFO mà ta đã bàn đến.

May mắn thay, một queue (hàng đợi) là cấu trúc dữ liệu FIFO, và cũng may mắn là ta có một tin nhắn muốn gửi. Tin nhắn. Hàng đợi. Hàng đợi tin nhắn!

Vậy là ta sẽ có một (hoặc nhiều) *sender* (người gửi) đổ tin nhắn vào hàng đợi ở một đầu, và ta sẽ có một (hoặc nhiều) *receiver* (người nhận) đọc tin nhắn ra từ hàng đợi ở đầu kia.

8.2 Tại Sao Dùng Cái Này?

Ta có một số ưu điểm so với FIFO thông thường. Một là các tin nhắn sẽ không bị chia tách (xen kẽ) nếu có nhiều sender cùng gửi một lúc. (Điều này có thể xảy ra trong FIFO với các tin nhắn lớn hơn.)

Ưu điểm khác là ta có thể gán cho các tin nhắn một *mức ưu tiên* để kiểm soát thứ tự chúng được giao.

Cuối cùng, giống như FIFO, các hàng đợi này có thể được tham gia hoặc rời đi bất cứ lúc nào. Các tiến trình mới chỉ cần mở hàng đợi bằng một tên đã được thỏa thuận trước và đều biết. Sẽ nói thêm về điều đó sau.

8.3 Ưu Tiên

Mỗi lần bạn gửi một tin nhắn vào hàng đợi, bạn gắn kèm một *priority* (ưu tiên) cho biết (một cách mơ hồ) tin nhắn đó nên được giao nhanh như thế nào. Ưu tiên chỉ là một số nguyên không dấu, trong đó `0` là ưu tiên thấp nhất, và một số nguyên lớn hơn nào đó, được chỉ định bởi `MQ_PRIO_MAX`, là cao nhất.

Đặc tả không nêu rõ mức cao nhất ngoài giá trị macro phụ thuộc hệ thống đó. Trang man của Linux đề xuất giữ mức ưu tiên trong khoảng 0 đến 31, bao gồm cả hai đầu, để đảm bảo khả năng khả chuyển tối đa.

[1] https://en.wikipedia.org/wiki/POSIX

Và nếu bạn thực sự cần hơn 32 mức ưu tiên... thật ra, bạn đang xây dựng cái gì vậy?

Dù sao, khi bạn nhận một tin nhắn, bạn sẽ nhận được cái được gửi với ưu tiên cao nhất (dù nó được gửi muộn hơn các tin nhắn có ưu tiên thấp hơn). Nếu có hai tin nhắn cùng mức ưu tiên cao nhất, chúng đấu tay đôi với nhau đến cùng.

Không, đó không đúng. Nếu có sự ngang bằng về ưu tiên, các tin nhắn bị buộc được lấy ra theo thứ tự chúng được gửi (FIFO).

8.4 Xác Định Một Hàng Đợi

Hàng đợi tin nhắn được xác định bởi một *name* (tên), là một chuỗi nên bắt đầu bằng dấu gạch chéo (`/`) và không có dấu gạch chéo nào khác trong đó. (Mọi thứ trở nên "phụ thuộc cài đặt" nếu bạn vi phạm những quy tắc đó.)

Ví dụ, đây là một tên hàng đợi: `/waco_kid`. Rất thú vị. Tất cả các chương trình muốn sử dụng hàng đợi đó phải biết tên đó từ trước để có thể mở nó.

8.5 Cách Tiếp Cận Tổng Quát

8.5.1 Mở Hàng Đợi

Cả sender và receiver đều phải làm điều tương tự ở bước đầu: mở (kết nối tới) hàng đợi tin nhắn. Điều này được thực hiện bằng system call `mq_open()`. (Và ở đây bạn sẽ thấy file header `mqueue.h` mà bạn cần cho tất cả những thứ này.)

Hàng đợi cũng được tạo ra bằng lệnh gọi này. Nếu nó chưa tồn tại và các flag cùng đối số phù hợp được truyền vào `mq_open()`, hàng đợi sẽ được tạo ra lúc đó.

Điều đáng chú ý ở đây là hàng đợi bạn tạo không bao giờ biến mất cho đến khi bạn vứt máy tính đi hoặc bạn *unlink* hàng đợi đó, tùy cái nào đến trước. Sẽ nói thêm về unlink sau.

Hãy xem syscall này!

```
#include <mqueue.h>
#include <fcntl.h>  // For the O_ flags

mqd_t mq_open(const char *name, int oflag, ...);
```

Bạn truyền tên vào đối số đầu tiên, ví dụ `/waco_kid`, rồi truyền một số flag vào `oflag`. Và tùy theo flag, bạn có thể truyền thêm một số thứ với dấu chấm lửng đáng sợ kia ở cuối.

Các flag được kết hợp bằng OR theo bit. Đầu tiên, bạn phải nói rõ muốn mở để đọc (nhận), ghi (gửi), hay cả hai. Ngoài ra, bạn có thể yêu cầu tạo hàng đợi nếu nó chưa tồn tại. Và bạn có thể chỉ định hàng đợi có nên ở chế độ *blocking* hay không. Sẽ nói thêm về điều đó sau.

Flag	Mô tả
`O_RDONLY`	Mở chỉ để nhận
`O_WRONLY`	Mở chỉ để gửi
`O_RDWR`	Mở cho cả nhận và gửi
`O_CREAT`	Tạo hàng đợi nếu chưa tồn tại
`O_NONBLOCK`	Tạo hàng đợi không blocking

Ví dụ:

```
mqd_t mq = mq_open("/waco_kid", O_RDONLY);
```

Nhưng ở đây ta đến phần dấu chấm lửng! Nếu ta chỉ định `O_CREAT`, ta có thể làm thêm nhiều thứ!

Cụ thể, ta có thể đặt quyền (ai được phép kết nối vào), điều ta làm giống như bất kỳ quyền file Unix tiêu chuẩn nào. Trong ví dụ dưới đây, ta dùng quyền `0644`, tức là `rw-r--r--`. Rồi tôi đặt `NULL` cho đối số thứ tư — ta sẽ sớm thấy ý nghĩa của nó.

```
mqd_t mqdes = mq_open("/waco_kid", O_RDONLY | O_CREAT, 0644, NULL);
```

Như vậy, lệnh đó sẽ tạo một hàng đợi tin nhắn! Và nó làm vậy với số lượng tin nhắn tối đa và kích thước tin nhắn tối đa mặc định.

Nếu bạn muốn thứ gì đó khác so với mặc định thì sao? Bạn có thể dùng đối số thứ tư để chỉ định bằng `struct mq_attr`.

Đây là các trường liên quan cho việc tạo hàng đợi:

```
struct mq_attr {
    long mq_maxmsg;    // Max message count
    long mq_msgsize;   // Max message size
}
```

Bạn có thể kiểm soát số lượng tin nhắn tối đa có thể có trong hàng đợi cùng một lúc với `mq_maxmsg`. Không có giá trị tối đa cố định trong đặc tả, nhưng số tối đa bạn có thể chỉ định trên máy Linux của tôi là 10. Có vẻ khá thấp, nhưng bạn phải tưởng tượng rằng kernel đang giữ tất cả những tin nhắn này cho đến khi ai đó nhận chúng, và nó không muốn dùng hết bộ nhớ của bạn để làm điều đó. Nếu mọi thứ hoạt động trơn tru, các tiến trình khác sẽ tiêu thụ tin nhắn nhanh như khi bạn tạo ra chúng.

> Và, như chúng ta đều biết, *mọi thứ **luôn luôn** chạy trơn tru!*

Ngoài ra, mỗi tin nhắn không thể lớn hơn `mq_msgsize`. Lại không có giá trị tối đa được định nghĩa, nhưng trên máy Linux của tôi là 8 KB.

> **Bạn có thể tự tìm hiểu điều này** trên Linux bằng cách xem một số file trong `/proc`
>
> ```
> cat /proc/sys/fs/mqueue/msgsize_max   # max mq_msgsize
> cat /proc/sys/fs/mqueue/msg_max       # max mq_maxmsg
> cat /proc/sys/fs/mqueue/queues_max    # max queues
> ```

Ta sẽ thấy những gì ta có thể làm thêm với `struct mq_attr` sau, bao gồm kiểm tra xem có bao nhiêu tin nhắn trong hàng đợi.

8.5.2 Gửi Thứ Gì Đó Vào Hàng Đợi

OK! Bây giờ ta đã mở và tạo hàng đợi, ta có thể gửi đồ vào nó!

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
            size_t msg_len, unsigned int msg_prio);
```

Lệnh đó gửi tin nhắn trỏ bởi `msg_ptr` (có độ dài `msg_len` byte) vào định danh hàng đợi ta lấy từ `mq_open()`. Và nó gửi với ưu tiên `msg_prio`.

Vậy thôi. Đây là một ví dụ không có kiểm tra lỗi:

```
mqd_t mqdes = mq_open("/waco_kid", O_RDONLY | O_CREAT, 0644, NULL);

mq_send(mqdes, "Play chess", 10, 0);
```

Nếu bạn gửi khi hàng đợi đầy, lệnh gọi sẽ block cho đến khi có gì đó xóa một tin nhắn khỏi hàng đợi để nhường chỗ.

8.5.3 Nhận Thứ Gì Đó Từ Hàng Đợi

Chiều ngược lại là nhận. Về cơ bản giống nhau.

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char msg_ptr[msg_len],
                   size_t msg_len, unsigned int *msg_prio);
```

Lệnh đó sẽ nhận một tin nhắn từ hàng đợi được xác định bởi `mqdes`. Nó lưu tin nhắn vào `msg_ptr`, cái này phải là một buffer ít nhất `msg_len` byte, nếu không thì có chuyện đấy. Ồ, và `msg_len` cũng phải ít nhất bằng kích thước tin nhắn tối đa (mà bạn tùy chọn đặt với `mq_open()`), không thì cũng có chuyện.

Cuối cùng, nếu bạn quan tâm đến ưu tiên của tin nhắn này, bạn có thể truyền con trỏ tới một `unsigned int` trong `msg_prio` để giữ nó. Hoặc bạn có thể truyền `NULL` cho đối số đó nếu không quan tâm.

```
mqd_t mqdes = mq_open("/waco_kid", O_RDONLY);

char msg[8192];
ssize_t recv_len;
unsigned int msg_prio;

recv_len = mq_receive(mqdes, msg, sizeof msg, &msg_prio);

// Print it to stdout
write(1, msg, recv_len);
```

Nhắc lại, bạn nên kiểm tra lỗi cho những lệnh đó.

8.5.4 Đóng Hàng Đợi

Khi bạn dùng xong hàng đợi *trong một tiến trình cụ thể*, bạn có thể đóng nó lại. (Hàng đợi sẽ tiếp tục tồn tại cho đến khi bị unlink.)

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Khá đơn giản. Đây là một ví dụ cho đầy đủ:

```
mqd_t mqdes = mq_open("/waco_kid", O_RDONLY);

// ...
// Làm việc với hàng đợi một lúc rồi xong.
// ...

mq_close(mqdes);
```

8.6 Ví Dụ Sender

Hãy làm một ví dụ hoàn chỉnh. Đoạn code này sẽ nhắc bạn nhập ưu tiên và tin nhắn cách nhau bằng dấu cách, kiểu như `5 Hello`. Nhập dòng trắng để thoát.

Và nó sẽ gửi chuỗi kết thúc null ra hàng đợi.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <mqueue.h>

/**
 * Input a priority and message from the keyboard.
 *
 * Really fragile--for demo purposes only.
 */
int input(char *buf, size_t bufsize, unsigned int *msg_prio)
{
    printf("Priority and message (e.g. 2 hi): ");
    fflush(stdout);
    fgets(buf, bufsize - 1, stdin);
    buf[bufsize - 1] = '\0';

    char *token = strtok(buf, " \n");

    if (token == NULL)
        return 0;

    *msg_prio = atoi(token);  // Get priority

    token = strtok(NULL, "\n");
    int msg_len = strlen(token) + 1;

    memmove(buf, token, msg_len);

    return msg_len;
}

int main(void)
{
    char msg[128];

    struct mq_attr attr = {
        .mq_maxmsg = 3,
        .mq_msgsize = 256
    };

    mqd_t mqdes = mq_open("/mq_test", O_WRONLY | O_CREAT, 0644,
                          &attr);

    for (;;) {
        unsigned int msg_prio;

        int msg_len = input(msg, sizeof msg, &msg_prio);

        if (msg_len == 0)
            break;

        printf("sending \"%s\" (%d bytes) at priority %u\n", msg,
               msg_len, msg_prio);

        if (mq_send(mqdes, msg, msg_len, msg_prio) == -1) {
```

```
            perror("mq_send");
        }
    }

    mq_close(mqdes);
}
```

Chạy chương trình đó và gửi một vài thứ. Lưu ý rằng số lượng tin nhắn tối đa trong hàng đợi tại một thời điểm được đặt là `3`, vì vậy khi bạn cố gửi thứ thứ tư, nó sẽ block cho đến khi bạn khởi động một receiver.

```
$ ./mq_sender
  Priority and message (e.g. 2 hi): 1 hello
  sending "hello" (6 bytes) at priority 1
  Priority and message (e.g. 2 hi): 2 and this
  sending "and this" (9 bytes) at priority 2
  Priority and message (e.g. 2 hi): 0 low priority
  sending "low priority" (13 bytes) at priority 0
  Priority and message (e.g. 2 hi): 2 a fourth message
  sending "a fourth message" (17 bytes) at priority 2
```

Và ở đó tôi đã bị block. Cứ để nó ngồi đó, và hãy khởi động một receiver trong terminal khác.

8.7 Ví Dụ Receiver

Đây là một receiver nhận tin nhắn.

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    mqd_t mqdes = mq_open("/mq_test", O_RDONLY);

    char msg[128];
    char msg_len = sizeof msg;
    unsigned int msg_prio;
    ssize_t bytes_received;

    for (;;) {
        bytes_received = mq_receive(mqdes, msg, msg_len,
                                    &msg_prio);

        if (bytes_received == -1) {
            perror("mq_receive");
            return 1;
        }

        printf("received \"%s\" (%zd bytes) at priority %u\n", msg,
               bytes_received, msg_prio);
    }
```

```
    mq_close(mqdes);
}
```

Giả sử bạn vẫn đang chạy sender ở trên trong một cửa sổ, và bạn vừa khởi động cái này trong cửa sổ khác, bạn sẽ thấy ngay hai điều.

Một là receiver sẽ ngốm hết và in ra tất cả các tin nhắn trong hàng đợi. Điều kia là sender sẽ ngay lập tức được unblock và cho bạn cơ hội gõ thêm tin nhắn khác.

Receiver sẽ in ra:

```
$ ./mq_receiver
  received "and this" (9 bytes) at priority 2
  received "a fourth message" (17 bytes) at priority 2
  received "hello" (6 bytes) at priority 1
  received "low priority" (13 bytes) at priority 0
```

Chú ý thứ tự! Tin nhắn thứ tư ta gửi thực ra đến thứ hai. Tại sao? Vì nó có ưu tiên 2, nên nó được nhận trước bất cứ thứ gì có ưu tiên thấp hơn, dù những tin nhắn ưu tiên thấp hơn đó được gửi trước. Kẻ chen hàng!

Và lúc này, nếu bạn gõ gì đó ở sender, chúng sẽ ngay lập tức xuất hiện trên receiver.

8.8 Nhiều Tiến Trình

Nếu bạn có nhiều sender, chúng sẽ đổ tin nhắn vào cùng một hàng đợi theo cách trực quan. Không vấn đề gì.

Nếu bạn có nhiều receiver, tôi thực ra không chắc đặc tả nói gì về điều đó. Nhưng khi tôi chạy trên Linux, có vẻ như các receiver thay nhau nhận tin nhắn.

Và đây là hành vi hợp lý. Có lẽ bạn có một tiến trình tạo ra các công việc và đưa chúng vào hàng đợi, và bạn có nhiều tiến trình chạy các công việc đó, tất cả đều với tay vào hàng đợi để lấy việc tiếp theo cần làm.

Hãy thử! Mở thêm một cửa sổ nữa, khởi động receiver thứ hai, và xem các tin nhắn từ sender đi đâu.

8.9 Unlink (Xóa) Hàng Đợi

Nếu tất cả các chương trình `mq_close()` hàng đợi, nó có biến mất không? ***Không***, nó không biến mất. Nó vẫn còn đó. Và nếu có tin nhắn trong đó, chúng cũng còn đó.

Bạn phải *unlink* hàng đợi, điều này hơi tương tự như xóa một file.

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

Bạn chỉ cần truyền tên mà bạn đã tạo hàng đợi:

```
mq_unlink("/waco_kid");
```

Và vậy là xong.

Cũng gần như vậy. Có một vài chi tiết phức tạp. Nếu bạn unlink hàng đợi, nó thực sự tiếp tục tồn tại cho đến khi tất cả người dùng của hàng đợi đã thoát hoặc `mq_close()` nó.

Vậy nếu *mọi người* đã đóng nó **và** bạn unlink nó, thì nó biến mất.

Ngoài ra, nếu bạn unlink nó nhưng vẫn giữ nó mở, một tiến trình khác có thể tạo một hàng đợi khác cùng tên mà bạn đã dùng.

> Đây thực ra chính xác là cách xóa file (sử dụng syscall `unlink()`) hoạt động. Bạn có thể unlink một file và giữ nó mở; file thực sự không bị xóa khỏi đĩa cho đến khi nó được unlink **và** tất cả các tiến trình đã đóng nó.

Đây là một ví dụ unlink hàng đợi tin nhắn từ các ví dụ trước. Nếu bạn không chạy cái này, hàng đợi sẽ tồn tại cho đến khi bạn khởi động lại máy.

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

int main(void)
{
    if (mq_unlink("/mq_test") == -1) {
        perror("/mq_test");
        return 1;
    }
}
```

8.10 Siêu Dữ Liệu Hàng Đợi

Bạn có thể xem các thuộc tính của hàng đợi, một vài trong số đó bạn có thể đã đặt trong lệnh `mq_open()`.

Các lệnh gọi này dùng người bạn cũ `struct mq_attr` để giữ thông tin.

```
struct mq_attr {
    long mq_flags;     // O_NONBLOCK?
    long mq_maxmsg;    // Max message count
    long mq_msgsize;   // Max message size
    long mq_curmsgs;   // How many messages in queue
};
```

Bạn có thể biết hàng đợi được tạo là non-blocking hay không bằng cách nhìn vào `mq_flags`. Nếu bạn AND theo bit với `O_NONBLOCK` và kết quả khác không, thì nó là non-blocking. Không, không, không.

Và, rõ ràng, bạn có thể biết hàng đợi đầy đến mức nào bằng cách nhìn vào `mq_curmsgs`.

Bạn cũng có thể đặt các thuộc tính, nhưng thứ duy nhất bạn được phép đặt là `mq_flags`. Vì vậy đây là cách bạn có thể chuyển một hàng đợi từ blocking sang non-blocking hoặc ngược lại. Tất cả các trường khác bị bỏ qua khi đặt thuộc tính.

Đây là getter và setter:

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

Setter cũng trả về cho bạn các thuộc tính trước đó trong `oldattr`, nếu nó không phải `NULL`.

Đây là một ví dụ xem có bao nhiêu tin nhắn trong hàng đợi rồi chuyển nó sang non-blocking.

```
struct mq_attr attr;

mq_getattr(mqdes, &attr);
```

```
printf("Currently in queue: %ld\n", attr.mq_curmsgs);

attr.mq_flags |= O_NONBLOCK;
mq_setattr(mqdes, &attr, NULL);
```

8.11 Hết Giờ!

Tôi sẽ không đi vào quá nhiều chi tiết ở đây, nhưng với những lệnh gọi block này, đôi khi ta muốn chỉ chờ một khoảng thời gian nhất định trước khi bỏ cuộc.

Bạn có thể dùng các lệnh gọi `mq_timedsend()` và `my_timedreceive()` hoạt động giống như `mq_send()` và `mq_receive()` ngoại trừ chúng có thêm `struct timespec` ở cuối cho phép bạn chỉ định thời gian chờ tối đa.

Nếu hết thời gian, lệnh gọi trả về `-1` và `errno` được đặt thành `ETIMEDOUT`.

Để nhắc nhanh, `struct timespec` có hai trường:

- `tv_sec` số giây, cộng thêm...
- `tv_nsec` số nanosecond

Có 1.000.000.000 (một tỷ) nanosecond trong một giây, vì vậy trường `tv_nsec` nằm trong khoảng từ `0` đến `999999999`.

Đây là một `struct timespec` cho bạn thời gian chờ 3,75 giây:

```
struct timespec timelimit = {
    .tv_sec = 3,
    .tv_nsec = 750000000
}
```

8.12 Blocking và Non-Blocking

Nói chung, nếu bạn cố gửi một tin nhắn vào hàng đợi đầy, lệnh gọi `mq_send()` sẽ block.

Và nếu bạn cố nhận từ một hàng đợi trống bằng `mq_receive()`, lệnh gọi sẽ block.

Nếu điều đó không mong muốn, bạn có thể đặt hàng đợi sang non-blocking. Bạn làm điều này bằng cách truyền flag `O_NONBLOCK` vào `mq_open()`, hoặc bằng cách đặt nó sau đó bằng `mq_setattr()`.

Nếu bạn đặt hàng đợi là non-blocking, tất cả những lệnh gọi *sẽ* block trong trường hợp bình thường sẽ trả về `-1` và `errno` sẽ được đặt thành `EWOULDBLOCK`.

Chapter 9

Hàng Đợi Tin Nhắn System V

Những người mang lại cho chúng ta System V đã bao gồm một số tính năng IPC tuyệt vời đã được triển khai trên nhiều nền tảng (bao gồm Linux, tất nhiên.) Tài liệu này mô tả cách sử dụng và chức năng của Hàng Đợi Tin Nhắn System V cực kỳ ấn tượng!

Bây giờ, trước khi bắt đầu, thông tin này đã khá *cũ*. Ừ thì, thực ra thông tin vẫn còn tốt, nhưng có một API hàng đợi tin nhắn POSIX mới hơn, như mô tả trong chương trước của hướng dẫn này, phù hợp hơn cho cuộc sống hiện đại. Nhưng có thể bạn đang dùng máy cũ hơn hoặc chỉ muốn trải nghiệm hoài cổ. Nếu vậy, hãy đọc tiếp!

Như thường lệ, tôi muốn trình bày tổng quan trước khi đi vào chi tiết. Một hàng đợi tin nhắn hoạt động giống như FIFO, nhưng hỗ trợ thêm một số tính năng. Nhìn chung, các tin nhắn được lấy ra khỏi hàng đợi theo thứ tự chúng được đưa vào. Tuy nhiên, cụ thể hơn, có những cách để lấy một số tin nhắn nhất định ra khỏi hàng đợi trước khi chúng đến lượt. Giống như chen hàng vậy. (Nhân tiện, đừng cố chen hàng khi ghé thăm công viên giải trí Great America ở Silicon Valley, vì bạn có thể bị bắt vì điều đó. Họ coi việc chen hàng rất *nghiêm túc* ở đó.)

Về mặt sử dụng, một tiến trình có thể tạo một hàng đợi tin nhắn mới, hoặc kết nối vào một hàng đợi đã có. Theo cách thứ hai này, hai tiến trình có thể trao đổi thông tin qua cùng một hàng đợi tin nhắn. Tuyệt.

Thêm một điều về System V IPC: khi bạn tạo một hàng đợi tin nhắn, nó không biến mất cho đến khi bạn xóa nó, giống như cách các file không biến mất cho đến khi bạn xóa chúng một cách rõ ràng. Tất cả các tiến trình từng sử dụng nó có thể thoát, nhưng hàng đợi vẫn tồn tại. Một thói quen tốt là dùng lệnh `ipcs` để kiểm tra xem có hàng đợi tin nhắn nào của bạn không dùng đang lơ lửng ở đó không. Bạn có thể xóa chúng bằng lệnh `ipcrm`, điều này tốt hơn là để sysadmin ghé thăm và nói rằng bạn đã chiếm hết mọi hàng đợi tin nhắn khả dụng trên hệ thống.

9.1 Hàng Đợi Của Tôi Ở Đâu?

Hãy bắt đầu thôi! Trước tiên, bạn muốn kết nối vào một hàng đợi, hoặc tạo nó nếu chưa tồn tại. Lệnh gọi để thực hiện điều này là syscall `msgget()`:

```
int msgget(key_t key, int msgflg);
```

`msgget()` trả về ID hàng đợi tin nhắn khi thành công, hoặc `-1` khi thất bại (và nó đặt `errno`, tất nhiên.)

Các đối số có vẻ hơi kỳ lạ, nhưng có thể hiểu được sau một chút vật lộn. Đầu tiên, `key` là một định danh duy nhất trên toàn hệ thống mô tả hàng đợi bạn muốn kết nối vào (hoặc tạo). Mọi tiến trình khác muốn kết nối vào hàng đợi này sẽ phải dùng cùng `key`.

Đối số kia, `msgflg` cho `msgget()` biết phải làm gì với hàng đợi đó. Để tạo một hàng đợi, trường này phải được đặt bằng `IPC_CREAT` OR theo bit với các quyền cho hàng đợi này. (Quyền hàng đợi giống

như quyền file Unix tiêu chuẩn—hàng đợi nhận user-id và group-id của chương trình tạo ra chúng.)

Một lệnh gọi mẫu được đưa ra trong phần sau.

9.2 "Anh có phải người giữ Key không?"

Chuyện về `key` này là gì vậy? Làm thế nào để tạo một cái? Vâng, vì kiểu `key_t` thực ra chỉ là một `long`, bạn có thể dùng bất kỳ số nào bạn muốn. Nhưng nếu bạn hardcode số đó và một chương trình khác không liên quan cũng hardcode cùng số đó nhưng muốn một hàng đợi khác thì sao? Giải pháp là dùng hàm `ftok()` tạo ra một key từ hai đối số:

```
key_t ftok(const char *`path`, int `id`);
```

OK, điều này đang trở nên kỳ lạ. Về cơ bản, `path` chỉ cần là đường dẫn đến một file xác định duy nhất ứng dụng này; đường dẫn đến file cấu hình của ứng dụng là một chuỗi thường được dùng (khả năng nào hai ứng dụng sẽ dùng cùng file cấu hình?). Đối số kia, `id` thường chỉ được đặt thành một ký tự tùy ý, như 'A'. Hàm `ftok()` dùng thông tin về file đã đặt tên (như số inode, v.v.) và `id` để tạo ra một `key` có thể là duy nhất cho `msgget()`. Các chương trình muốn dùng cùng hàng đợi phải tạo ra cùng `key`, vì vậy chúng phải truyền cùng tham số vào `ftok()`.

Cuối cùng, đến lúc thực hiện lệnh gọi:

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

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);
```

Trong ví dụ trên, tôi đặt quyền trên hàng đợi thành `666` (hoặc `rw-rw-rw-`, nếu điều đó dễ hiểu hơn với bạn). Và bây giờ ta có `msqid` sẽ được dùng để gửi và nhận tin nhắn từ hàng đợi.

9.3 Gửi Vào Hàng Đợi

Sau khi bạn đã kết nối vào hàng đợi tin nhắn bằng `msgget()`, bạn đã sẵn sàng gửi và nhận tin nhắn. Đầu tiên, việc gửi:

Mỗi tin nhắn gồm hai phần, được định nghĩa trong cấu trúc mẫu `struct msgbuf`, như định nghĩa trong `sys/msg.h`:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Trường `mtype` được dùng sau này khi lấy tin nhắn từ hàng đợi, và có thể được đặt thành bất kỳ số dương nào. `mtext` là dữ liệu sẽ được thêm vào hàng đợi.

"Cái gì?! Bạn chỉ có thể đưa mảng một byte lên hàng đợi tin nhắn thôi sao?! Vô dụng!!" Vâng, không hẳn vậy. Bạn có thể dùng bất kỳ cấu trúc nào bạn muốn để đưa tin nhắn vào hàng đợi, miễn là phần tử đầu tiên là một long. Ví dụ, ta có thể dùng cấu trúc này để lưu trữ đủ loại thứ:

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
```

```
        int cruelty;
        int booty_value;
    } info;
};
```

OK, vậy làm thế nào ta truyền thông tin này vào một hàng đợi tin nhắn? Câu trả lời rất đơn giản, các bạn ơi: chỉ cần dùng `msgsnd()`:

```
int msgsnd(int msqid, const void *msgp,
           size_t msgsz, int msgflg);
```

`msqid` là định danh hàng đợi tin nhắn được trả về bởi `msgget()`. Con trỏ `msgp` là con trỏ đến dữ liệu bạn muốn đưa vào hàng đợi. `msgsz` là kích thước tính bằng byte của dữ liệu cần thêm vào hàng đợi (không tính kích thước của phần tử `mtype`). Cuối cùng, `msgflg` cho phép bạn đặt một số tham số flag tùy chọn, mà ta sẽ bỏ qua bây giờ bằng cách đặt nó thành `0`.

Cách tốt nhất để lấy kích thước dữ liệu cần gửi là thiết lập đúng từ đầu. Trường đầu tiên của `struct` phải là một `long`, như ta đã thấy. Để an toàn và khả chuyển, chỉ nên có một trường bổ sung. Nếu bạn cần nhiều hơn một, hãy bọc chúng vào một `struct` giống như `struct pirate_msgbuf` ở trên.

Khi cần lấy kích thước dữ liệu cần gửi, chỉ cần lấy kích thước của trường thứ hai:

```
struct cheese_msgbuf {
    long mtype;
    char name[20];
};

/* calculate the size of the data to send: */

struct cheese_msgbuf mbuf;
int size;

size = sizeof mbuf.name;

/* Or, without a declared variable: */

size = sizeof ((struct cheese_msgbuf*)0)->name;
```

Hoặc, nếu bạn có nhiều trường khác nhau, hãy đưa chúng vào một `struct` và dùng toán tử sizeof trên đó. Điều này có thể rất tiện lợi, vì bây giờ cấu trúc con có thể có tên để tham chiếu. Đây là đoạn code thêm một trong các cấu trúc cướp biển của ta vào hàng đợi tin nhắn:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb = {2, { "L'Olonais", 'S', 80, 10, 12035 } };

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* stick him on the queue */
/* struct pirate_info is the sub-structure */
msgsnd(msqid, &pmb, sizeof(struct pirate_info), 0);
```

Ngoài việc nhớ kiểm tra lỗi từ các giá trị trả về của tất cả các hàm này, đó là tất cả những gì cần làm. Ồ, vâng: lưu ý rằng tôi tùy ý đặt trường `mtype` thành `2` ở đó. Điều đó sẽ quan trọng trong phần tiếp theo.

9.4 Nhận Từ Hàng Đợi

Bây giờ ta đã có tên cướp biển đáng sợ Francis L'Olonais kẹt trong hàng đợi tin nhắn của ta, làm thế nào để lấy anh ta ra? Như bạn có thể tưởng tượng, có một hàm đối xứng với `msgsnd()`: đó là `msgrcv()`. Thật sáng tạo.

Một lệnh gọi `msgrcv()` để làm điều đó trông như thế này:

```
#include <sys/msg.h>
#include <stddef.h>

key_t key;
int msqid;
struct pirate_msgbuf pmb; /* where L'Olonais is to be kept */

key = ftok("/home/beej/somefile", 'b');
msqid = msgget(key, 0666 | IPC_CREAT);

/* get him off the queue! */
msgrcv(msqid, &pmb, sizeof(struct pirate_info), 2, 0);
```

Có một điều mới cần lưu ý trong lệnh gọi `msgrcv()`: số `2`! Nó có nghĩa là gì? Đây là tóm tắt của lệnh gọi:

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
```

Số `2` ta chỉ định trong lệnh gọi là `msgtyp` được yêu cầu. Nhớ lại rằng ta đã đặt `mtype` tùy ý thành `2` trong phần `msgsnd()` của tài liệu này, vì vậy đó sẽ là cái được lấy ra từ hàng đợi.

Thực ra, hành vi của `msgrcv()` có thể thay đổi đáng kể bằng cách chọn `msgtyp` là dương, âm, hoặc bằng không:

`msgtyp`	Hiệu ứng trên `msgrcv()`
Không	Lấy tin nhắn tiếp theo trong hàng đợi, bất kể `mtype` của nó.
Dương	Lấy tin nhắn tiếp theo có `mtype` *bằng* `msgtyp` đã chỉ định.
Âm	Lấy tin nhắn đầu tiên trong hàng đợi có trường `mtype` nhỏ hơn hoặc bằng giá trị tuyệt đối của đối số `msgtyp`.

Vì vậy, điều thường xảy ra là bạn chỉ muốn tin nhắn tiếp theo trong hàng đợi, bất kể `mtype` là gì. Như vậy, bạn sẽ đặt tham số `msgtyp` thành `0`.

9.5 Xóa Một Hàng Đợi Tin Nhắn

Đến lúc nào đó bạn sẽ phải xóa một hàng đợi tin nhắn. Như tôi đã nói trước đó, chúng sẽ tồn tại cho đến khi bạn xóa chúng một cách rõ ràng; điều quan trọng là bạn làm điều này để không lãng phí tài nguyên hệ thống. OK, vậy bạn đã dùng hàng đợi tin nhắn này cả ngày, và nó đã cũ rồi. Bạn muốn tiêu diệt nó. Có hai cách:

1. Dùng lệnh Unix `ipcs` để lấy danh sách các hàng đợi tin nhắn đã định nghĩa, rồi dùng lệnh `ipcrm` để xóa hàng đợi.

2. Viết một chương trình để làm điều đó cho bạn.

Thường thì lựa chọn thứ hai là phù hợp nhất, vì bạn có thể muốn chương trình của mình dọn dẹp hàng đợi vào một lúc nào đó. Để làm điều này cần giới thiệu thêm một hàm: `msgctl()`.

Tóm tắt của `msgctl()` là:

```
int msgctl(int msqid, int cmd,
           struct msqid_ds *buf);
```

Tất nhiên, `msqid` là định danh hàng đợi lấy từ `msgget()`. Đối số quan trọng là `cmd` cho `msgctl()` biết cách hành xử. Nó có thể là nhiều thứ, nhưng ta chỉ nói về `IPC_RMID`, dùng để xóa hàng đợi tin nhắn. Đối số `buf` có thể được đặt thành `NULL` cho mục đích của `IPC_RMID`.

Giả sử ta có hàng đợi ta đã tạo ở trên để chứa các cướp biển. Bạn có thể xóa hàng đợi đó bằng cách gọi lệnh sau:

```
#include <sys/msg.h>
.
.
msgctl(msqid, IPC_RMID, NULL);
```

Và hàng đợi tin nhắn không còn nữa. (Tất nhiên, kiểm tra lỗi trên các giá trị trả về này luôn luôn phù hợp!)

9.6 Chương Trình Mẫu, Ai Muốn Xem Không?

Để cho đầy đủ, tôi sẽ bao gồm một cặp chương trình sẽ giao tiếp bằng hàng đợi tin nhắn. Chương trình đầu tiên, `kirk.c` thêm tin nhắn vào hàng đợi tin nhắn, và `spock.c` lấy chúng ra.

Đây là mã nguồn cho `kirk.c`[1]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }
```

[1] https://beej.us/guide/bgipc/source/examples/kirk.c

```
    if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }

    printf("Enter lines of text, ^D to quit:\n");

    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
        int len = strlen(buf.mtext);

        /* ditch newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';

        if (msgsnd(msqid, &buf, len, 0) == -1)
            perror("msgsnd");
    }

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }

    return 0;
}
```

Cách `kirk` hoạt động là nó cho phép bạn nhập các dòng văn bản. Mỗi dòng được gói vào một tin nhắn và thêm vào hàng đợi tin nhắn. Hàng đợi tin nhắn sau đó được đọc bởi `spock`.

Đây là mã nguồn cho `spock.c` [2]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void)
{
    struct my_msgbuf buf;
    int msqid;
    key_t key;

    if ((key = ftok("kirk.c", 'B')) == -1) {  /* same key as kirk.c */
        perror("ftok");
        exit(1);
    }
```

[2] https://beej.us/guide/bgipc/source/examples/spock.c

```
    if ((msqid = msgget(key, 0644)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }

    printf("spock: ready to receive messages, captain.\n");

    for(;;) { /* Spock never quits! */
        if (msgrcv(msqid, &buf, sizeof buf.mtext, 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("spock: \"%s\"\n", buf.mtext);
    }

    return 0;
}
```

Lưu ý rằng `spock`, trong lệnh gọi `msgget()`, không bao gồm tùy chọn `IPC_CREAT`. Ta để `kirk` tạo hàng đợi tin nhắn, và `spock` sẽ trả về lỗi nếu anh ta chưa làm vậy.

Hãy chú ý điều gì xảy ra khi bạn đang chạy cả hai trong các cửa sổ riêng biệt và bạn kill một trong hai. Cũng thử chạy hai bản sao của `kirk` hoặc hai bản sao của `spock` để có ý tưởng về điều gì xảy ra khi bạn có hai reader hoặc hai writer. Một bài trình diễn thú vị khác là chạy `kirk`, nhập một loạt tin nhắn, rồi chạy `spock` và xem nó lấy tất cả tin nhắn trong một lần. Chỉ cần nghịch ngợm với những chương trình đồ chơi này sẽ giúp bạn hiểu những gì thực sự đang xảy ra.

9.7 Tóm Tắt

Hàng đợi tin nhắn còn nhiều điều hơn những gì bài hướng dẫn ngắn này có thể trình bày. Hãy chắc chắn xem trang man để biết bạn có thể làm gì thêm, đặc biệt trong lĩnh vực `msgctl()`. Ngoài ra, còn có các tùy chọn khác bạn có thể truyền vào các hàm khác để kiểm soát cách `msgsnd()` và `msgrcv()` xử lý khi hàng đợi đầy hoặc trống tương ứng.

Chapter 10

Semaphore System V

Bạn còn nhớ khóa file không? Vâng, semaphore có thể được coi như một cơ chế khóa advisory rất tổng quát. Bạn có thể dùng chúng để kiểm soát truy cập vào file, bộ nhớ dùng chung, và thực ra bất cứ thứ gì bạn muốn. Chức năng cơ bản của semaphore là bạn có thể đặt nó, kiểm tra nó, hoặc chờ cho đến khi nó được xóa rồi đặt nó ("test-n-set"). Dù những thứ tiếp theo có phức tạp đến đâu, hãy nhớ ba thao tác đó.

Tài liệu này sẽ cung cấp tổng quan về chức năng semaphore, và kết thúc bằng một chương trình sử dụng semaphore để kiểm soát truy cập vào một file. (Nhiệm vụ này, thực ra, có thể dễ dàng được xử lý bằng khóa file, nhưng nó là một ví dụ tốt vì dễ hiểu hơn, chẳng hạn như bộ nhớ dùng chung.)

10.1 Lấy Một Số Semaphore

Với System V IPC, bạn không lấy các semaphore đơn lẻ; bạn lấy *tập hợp* semaphore. Bạn có thể, tất nhiên, lấy một tập hợp semaphore chỉ có một semaphore, nhưng điểm quan trọng là bạn có thể có cả một loạt semaphore chỉ bằng cách tạo một tập hợp semaphore duy nhất.

Làm thế nào bạn tạo tập hợp semaphore? Nó được thực hiện bằng một lệnh gọi tới `semget()`, trả về ID semaphore (từ đây gọi là `semid`):

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

int semget(key_t key, int nsems, int semflg);
```

`key` là gì? Đó là một định danh duy nhất được các tiến trình khác nhau sử dụng để xác định tập hợp semaphore này. (Cái `key` này sẽ được tạo ra bằng `ftok()`, mô tả trong phần Hàng Đợi Tin Nhắn.)

Đối số tiếp theo, `nsems`, là (bạn đoán đúng rồi!) số semaphore trong tập hợp semaphore này. Số tối đa phụ thuộc hệ thống, nhưng có lẽ khoảng 32000. Nếu bạn cần nhiều hơn (đồ tham lam!), chỉ cần lấy thêm một tập hợp semaphore khác. Bạn có thể truyền `0` nếu đang kết nối vào một tập hợp semaphore đã tồn tại, nhưng phải chỉ định số dương nếu bạn đang tạo một tập hợp semaphore mới.

Cuối cùng, có đối số `semflg`. Nó cho `semget()` biết quyền trên tập hợp semaphore mới là gì, bạn đang tạo tập mới hay chỉ muốn kết nối vào tập đã có, và các thứ khác mà bạn có thể tìm hiểu. Để tạo một tập mới, quyền có thể được OR theo bit với `IPC_CREAT`.

Đây là một lệnh gọi ví dụ tạo `key` bằng `ftok()` và tạo một tập hợp 10 semaphore, với quyền 666 (`rw-rw-rw-`):

```
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
key_t key;
int semid;

key = ftok("/home/beej/somefile", 'E');
semid = semget(key, 10, 0666 | IPC_CREAT);
```

Chúc mừng! Bạn vừa tạo một tập hợp semaphore mới! Sau khi chạy chương trình, bạn có thể kiểm tra bằng lệnh `ipcs`. (Đừng quên xóa nó khi dùng xong bằng `ipcrm`!)

Chờ đã! Cảnh báo! *¡Advertencia! ¡No pongas las manos en la tolva!* (Đó là câu tiếng Tây Ban Nha duy nhất tôi học được khi làm việc ở Pizza Hut năm 1990. Nó được in trên máy cán bột.) Hãy chú ý điều này:

Khi bạn lần đầu tạo một số semaphore, chúng đều chưa được khởi tạo; cần một lệnh gọi khác để đánh dấu chúng là trống (cụ thể là `semop()` hoặc `semctl()`—xem các phần tiếp theo.) Điều này có nghĩa là gì? Ý nghĩa là việc tạo semaphore không phải *atomic* (nói cách khác, nó không phải là một quá trình một bước). Nếu hai tiến trình đang cố tạo, khởi tạo và sử dụng semaphore cùng một lúc, một điều kiện race có thể xảy ra.

Một cách để vượt qua khó khăn này là có một tiến trình khởi tạo duy nhất tạo và khởi tạo semaphore từ lâu trước khi các tiến trình chính bắt đầu chạy. Tiến trình chính chỉ truy cập nó, không bao giờ tạo hay hủy nó.

Stevens đề cập đến vấn đề này là "lỗi chết người" của semaphore. Ông giải quyết nó bằng cách tạo tập hợp semaphore với cờ `IPC_EXCL`. Nếu tiến trình 1 tạo nó trước, tiến trình 2 sẽ trả về lỗi trong lệnh gọi (với `errno` được đặt thành `EEXIST`.) Lúc đó, tiến trình 2 sẽ phải chờ cho đến khi semaphore được khởi tạo bởi tiến trình 1. Làm sao biết được? Hóa ra, nó có thể gọi `semctl()` lặp đi lặp lại với cờ `IPC_STAT`, và xem thành viên `sem_otime` của cấu trúc `struct semid_ds` được trả về. Nếu giá trị đó khác không, có nghĩa là tiến trình 1 đã thực hiện một thao tác trên semaphore với `semop()`, có lẽ để khởi tạo nó.

Để xem ví dụ về điều này, hãy xem chương trình trình diễn `semdemo.c`[1], bên dưới, trong đó tôi tái triển khai một cách tổng quát code của Stevens.

Trong thời gian đó, hãy chuyển sang phần tiếp theo và xem cách khởi tạo các semaphore vừa tạo.

10.2 Kiểm Soát Semaphore Của Bạn Với `semctl()`

Sau khi bạn tạo các tập hợp semaphore, bạn phải khởi tạo chúng về một giá trị dương để cho thấy tài nguyên đang sẵn sàng sử dụng. Hàm `semctl()` cho phép bạn thực hiện thay đổi giá trị atomic cho các semaphore riêng lẻ hoặc toàn bộ tập hợp semaphore.

```
int semctl(int semid, int semnum, int cmd, ... /*arg*/);
```

`semid` là ID tập hợp semaphore bạn lấy từ lệnh gọi `semget()` trước đó. `semnum` là ID của semaphore mà bạn muốn thao tác giá trị. `cmd` là những gì bạn muốn làm với semaphore đó. Đối số cuối cùng, "`arg`", nếu cần, phải là một `union semun`, sẽ được bạn định nghĩa trong code của bạn là một trong những thứ sau:

```
union semun {
    int val;               /* used for SETVAL only */
    struct semid_ds *buf;  /* used for IPC_STAT and IPC_SET */
    ushort *array;         /* used for GETALL and SETALL */
};
```

(Lưu ý rằng `union semun` bây giờ được định nghĩa trong các file header của các hệ thống Linux hiện đại. Tuy nhiên, tôi không biết feature test macro nào để xác định điều này, vì vậy chỉ định nghĩa union này nếu hệ thống của bạn chưa có. Đọc tài liệu `semctl()` để biết thêm thông tin.)

[1] https://beej.us/guide/bgipc/source/examples/semdemo.c

Các trường khác nhau trong `union semun` được sử dụng tùy thuộc vào giá trị của tham số `cmd` cho `semctl()` (danh sách một phần như sau—xem trang man cục bộ của bạn để biết thêm):

`cmd`	Hiệu ứng
`SETVAL`	Đặt giá trị của semaphore đã chỉ định thành giá trị trong thành viên `val` của `union semun` được truyền vào.
`GETVAL`	Trả về giá trị của semaphore đã cho.
`SETALL`	Đặt giá trị của tất cả semaphore trong tập hợp thành các giá trị trong mảng trỏ bởi thành viên `array` của `union semun` được truyền vào. Tham số `semnum` cho `semctl()` không được dùng.<
`GETALL`	Lấy giá trị của tất cả semaphore trong tập hợp và lưu chúng vào mảng trỏ bởi thành viên `array` của `union semun` được truyền vào. Tham số `semnum` cho `semctl()` không được dùng.
`IPC_RMID`	Xóa tập hợp semaphore đã chỉ định khỏi hệ thống. Tham số `semnum` bị bỏ qua.
`IPC_STAT`	Tải thông tin trạng thái về tập hợp semaphore vào cấu trúc `struct semid_ds` trỏ bởi thành viên `buf` của `union semun`.

Để tham khảo, đây là nội dung (rút gọn) của `struct semid_ds` được dùng trong `union semun`:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Last change time */
    unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

Ta sẽ dùng thành viên `sem_otime` đó sau khi ta viết `initsem()` trong code mẫu bên dưới.

10.3 `semop()`: Sức Mạnh Atomic!

Tất cả các thao tác đặt, lấy, hoặc test-n-set một semaphore đều sử dụng syscall `semop()`. Syscall này là đa năng, và chức năng của nó được điều khiển bởi một cấu trúc được truyền vào nó, `struct sembuf`:

```
/* Warning! Members might not be in this order! */

struct sembuf {
    ushort sem_num;
    short sem_op;
    short sem_flg;
};
```

Tất nhiên, `sem_num` là số của semaphore trong tập hợp mà bạn muốn thao tác. Rồi, `sem_op` là những gì bạn muốn làm với semaphore đó. Nó mang các ý nghĩa khác nhau, tùy thuộc vào `sem_op` là dương, âm, hay bằng không, như được trình bày trong bảng sau:

`sem_op`	Điều xảy ra
Âm	Phân bổ tài nguyên. Block tiến trình gọi cho đến khi giá trị của semaphore lớn hơn hoặc bằng giá trị tuyệt đối của `sem_op`. (Tức là chờ cho đến khi đủ tài nguyên được giải phóng bởi các tiến trình khác để tiến trình này có thể phân bổ.) Sau đó cộng (thực chất là trừ, vì nó âm) giá trị `sem_op` vào giá trị semaphore.
Dương	Giải phóng tài nguyên. Giá trị `sem_op` được cộng vào giá trị semaphore.
Không	Tiến trình này sẽ chờ cho đến khi semaphore đạt giá trị 0.

Vậy, về cơ bản, những gì bạn làm là điền vào một `struct sembuf` với bất kỳ giá trị nào bạn muốn, rồi gọi `semop()`, như thế này:

```
int semop(int semid, struct sembuf *sops,
          unsigned int nsops);
```

Đối số `semid` là số lấy từ lệnh gọi `semget()`. Tiếp theo là `sops`, là con trỏ tới `struct sembuf` bạn đã điền với các lệnh semaphore. Tuy nhiên, nếu muốn, bạn có thể tạo một mảng các `struct sembuf` để thực hiện một loạt các thao tác semaphore cùng một lúc. Cách `semop()` biết bạn đang làm điều này là đối số `nsop`, cho biết có bao nhiêu `struct sembuf` bạn đang gửi. Nếu bạn chỉ có một, hãy đặt `1` cho đối số này.

Một trường trong `struct sembuf` mà tôi chưa đề cập là trường `sem_flg` cho phép chương trình chỉ định các cờ để sửa đổi thêm hiệu ứng của lệnh gọi `semop()`.

Một trong số các cờ này là `IPC_NOWAIT`, như tên gợi ý, làm cho lệnh gọi `semop()` trả về với lỗi `EAGAIN` nếu nó gặp tình huống thông thường sẽ block. Điều này tốt cho các tình huống bạn muốn "thăm dò" xem bạn có thể phân bổ tài nguyên hay không.

Một cờ rất hữu ích khác là cờ `SEM_UNDO`. Nó làm cho `semop()` ghi lại, theo một cách nào đó, sự thay đổi được thực hiện đối với semaphore. Khi chương trình thoát, kernel sẽ tự động hoàn tác tất cả các thay đổi được đánh dấu bằng cờ `SEM_UNDO`. Tất nhiên, chương trình của bạn nên cố gắng hết mức để giải phóng bất kỳ tài nguyên nào nó đánh dấu bằng semaphore, nhưng đôi khi điều này không thể thực hiện được khi chương trình của bạn nhận được `SIGKILL` hoặc một số sự cố khủng khiếp khác xảy ra.

10.4 Xóa Một Semaphore

Có hai cách để loại bỏ semaphore: một là dùng lệnh Unix `ipcrm`. Cách kia là thông qua lệnh gọi `semctl()` với `cmd` được đặt thành `IPC_RMID`.

Về cơ bản, bạn muốn gọi `semctl()` và đặt `semid` thành ID semaphore mà bạn muốn xóa. `cmd` nên được đặt thành `IPC_RMID`, cho `semctl()` biết xóa tập hợp semaphore này. Tham số `semnum` không có nghĩa gì trong bối cảnh `IPC_RMID` và có thể chỉ cần đặt thành không.

Đây là một lệnh gọi ví dụ để xóa một tập hợp semaphore:

```
int semid;
.
.
semid = semget(...);
.
.
semctl(semid, 0, IPC_RMID);
```

Dễ như ăn kẹo.

10.5 Chương Trình Mẫu

Có hai chương trình. Chương trình đầu tiên, `semdemo.c`, tạo semaphore nếu cần thiết, và thực hiện một số thao tác khóa file giả tạo trên nó trong một bài trình diễn rất giống bài trong tài liệu Khóa File. Chương trình thứ hai, `semrm.c` dùng để xóa semaphore (một lần nữa, `ipcrm` có thể được dùng để thực hiện điều này.)

Ý tưởng là chạy `semdemo.c` trong một vài cửa sổ và xem tất cả các tiến trình tương tác như thế nào. Khi xong, dùng `semrm.c` để xóa semaphore. Bạn cũng có thể thử xóa semaphore trong khi đang chạy `semdemo.c` chỉ để xem các loại lỗi nào được tạo ra.

Đây là `semdemo.c`[2], bao gồm một hàm có tên `initsem()` vượt qua các điều kiện race của semaphore theo phong cách Stevens:

```
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define MAX_RETRIES 10

#ifdef NEED_SEMUN
/* Defined in sys/sem.h as required by POSIX now */
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
};
#endif

/*
** initsem() -- more-than-inspired by W. Richard Stevens' UNIX Network
** Programming 2nd edition, volume 2, lockvsem.c, page 295.
*/
int initsem(key_t key, int nsems)  /* key from ftok() */
{
    int i;
    union semun arg;
    struct semid_ds buf;
    struct sembuf sb;
    int semid;

    semid = semget(key, nsems, IPC_CREAT | IPC_EXCL | 0666);

    if (semid >= 0) { /* we got it first */
            sb.sem_op = 1; sb.sem_flg = 0;
            arg.val = 1;

            printf("press return\n"); getchar();

            for(sb.sem_num = 0; sb.sem_num < nsems; sb.sem_num++) {
                    /* do a semop() to "free" the semaphores. */
                    /* this sets the sem_otime field, as needed below. */
                    if (semop(semid, &sb, 1) == -1) {
                            int e = errno;
                            semctl(semid, 0, IPC_RMID); /* clean up */
                            errno = e;
                            return -1; /* error, check errno */
                    }
            }

    } else if (errno == EEXIST) { /* someone else got it first */
            int ready = 0;

            semid = semget(key, nsems, 0); /* get the id */
```

[2]https://beej.us/guide/bgipc/source/examples/semdemo.c

```
            if (semid < 0) return semid; /* error, check errno */

            /* wait for other process to initialize the semaphore: */
            arg.buf = &buf;
            for(i = 0; i < MAX_RETRIES && !ready; i++) {
                    semctl(semid, nsems-1, IPC_STAT, arg);
                    if (arg.buf->sem_otime != 0) {
                            ready = 1;
                    } else {
                            sleep(1);
                    }
            }
            if (!ready) {
                    errno = ETIME;
                    return -1;
            }
    } else {
            return semid; /* error, check errno */
    }

    return semid;
}

int main(void)
{
    key_t key;
    int semid;
    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = -1;  /* set to allocate resource */
    sb.sem_flg = SEM_UNDO;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = initsem(key, 1)) == -1) {
            perror("initsem");
            exit(1);
    }

    printf("Press return to lock: ");
    getchar();
    printf("Trying to lock...\n");

    if (semop(semid, &sb, 1) == -1) {
            perror("semop");
            exit(1);
    }

    printf("Locked.\n");
    printf("Press return to unlock: ");
    getchar();
```

```
    sb.sem_op = 1; /* free resource */
    if (semop(semid, &sb, 1) == -1) {
            perror("semop");
            exit(1);
    }

    printf("Unlocked\n");

    return 0;
}
```

Đây là `semrm.c`[3] để xóa semaphore khi bạn xong:

```
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int main(void)
{
    key_t key;
    int semid;
    union semun arg;

    if ((key = ftok("semdemo.c", 'J')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* grab the semaphore set created by seminit.c: */
    if ((semid = semget(key, 1, 0)) == -1) {
            perror("semget");
            exit(1);
    }

    /* remove it: */
    if (semctl(semid, 0, IPC_RMID, 0) == -1) {
            perror("semctl");
            exit(1);
    }

    return 0;
}
```

Thật thú vị không! Tôi chắc chắn bạn sẽ từ bỏ Quake[4] chỉ để chơi với những thứ semaphore này cả ngày!

10.6 Tóm Tắt

Có lẽ tôi đã nói nhẹ về tính hữu dụng của semaphore. Tôi đảm bảo với bạn, chúng rất rất rất hữu ích trong tình huống đồng thời. Chúng thường còn nhanh hơn cả khóa file thông thường. Ngoài ra, bạn có thể dùng chúng trên những thứ khác không phải file, chẳng hạn như Vùng Nhớ Dùng Chung! Thực ra, đôi khi khó mà sống thiếu chúng, thẳng thắn mà nói.

[3] https://beej.us/guide/bgipc/source/examples/semrm.c
[4] Hoặc bất kỳ trò chơi FPS gây nghiện nào hiện tại.

Bất cứ khi nào bạn có nhiều tiến trình chạy qua một đoạn code quan trọng, bạn cần semaphore. Bạn có hàng tỷ cái—cứ dùng chúng đi.

Chapter 11

Vùng Nhớ Dùng Chung System V

Điều thú vị về các vùng nhớ dùng chung là chúng đúng như tên gọi: một vùng nhớ được chia sẻ giữa các tiến trình. Ý tôi là, hãy nghĩ về tiềm năng của điều này! Bạn có thể cấp phát một khối thông tin người chơi cho một trò chơi nhiều người chơi và để mỗi tiến trình truy cập vào đó tùy ý! Vui, vui, vui. (Tất nhiên, các file được ánh xạ bộ nhớ cũng làm được điều tương tự và có thêm ưu điểm là tính bền vững, dù có những điểm lưu ý tương tự áp dụng cho bộ nhớ dùng chung.)

Như thường lệ, có nhiều điều cần chú ý hơn, nhưng tất cả khá dễ về lâu dài. Bạn chỉ cần kết nối vào vùng nhớ dùng chung, và lấy một con trỏ đến vùng nhớ. Bạn có thể đọc và ghi vào con trỏ này và mọi thay đổi bạn thực hiện sẽ hiển thị cho tất cả những người khác kết nối vào vùng. Không có gì đơn giản hơn. Ừ thì, thực ra có, nhưng tôi chỉ đang cố làm bạn thoải mái hơn thôi.

11.1 Tạo Vùng và Kết Nối

Tương tự như các hình thức System V IPC khác, một vùng nhớ dùng chung được tạo và kết nối thông qua lệnh gọi `shmget()`:

```
int shmget(key_t key, size_t size, int shmflg);
```

Khi hoàn thành thành công, `shmget()` trả về một định danh cho vùng nhớ dùng chung. Đối số `key` nên được tạo theo cách tương tự như được trình bày trong tài liệu Hàng Đợi Tin Nhắn, sử dụng `ftok()`. Đối số tiếp theo, `size`, là kích thước tính bằng byte của vùng nhớ dùng chung. Cuối cùng, `shmflg` nên được đặt thành quyền của vùng OR theo bit với `IPC_CREAT` nếu bạn muốn tạo vùng, nhưng có thể là `0` trong trường hợp khác. (Không sao khi chỉ định `IPC_CREAT` mọi lúc—nó chỉ kết nối bạn nếu vùng đã tồn tại.)

Đây là một lệnh gọi ví dụ tạo một vùng 1K với quyền `644` (`rw-r--r--`):

```
key_t key;
int shmid;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

(Thực tế có thể không tạo được vùng 1K, vì hệ điều hành được phép tăng kích thước để phù hợp với bất kỳ ràng buộc nội bộ nào nó có. Ví dụ, trên hệ thống với trang bộ nhớ ảo 4K, kích thước có khả năng sẽ được tăng lên 4K. Tất nhiên, chương trình của bạn sẽ không biết hay quan tâm; đây chỉ là chi tiết triển khai.)

Nhưng làm thế nào bạn lấy con trỏ tới dữ liệu đó từ handle `shmid`? Câu trả lời nằm ở lệnh gọi `shmat()`, trong phần tiếp theo.

11.2 Gắn Vào—Lấy Con Trỏ Đến Vùng

Trước khi bạn có thể sử dụng một vùng nhớ dùng chung, bạn phải gắn bản thân vào nó bằng lệnh gọi `shmat()`:

```
void *shmat(int `shmid`, void *`shmaddr`, int `shmflg`);
```

Tất cả nghĩa là gì? Vâng, `shmid` là ID bộ nhớ dùng chung bạn lấy từ lệnh gọi `shmget()`. Tiếp theo là `shmaddr`, mà bạn có thể dùng để nói với `shmat()` địa chỉ cụ thể nào cần dùng, nhưng bạn chỉ cần đặt nó thành `0` và để hệ điều hành chọn địa chỉ cho bạn. Cuối cùng, `shmflg` có thể được đặt thành `SHM_RDONLY` nếu bạn chỉ muốn đọc từ nó, `0` trong trường hợp khác. (Xem trang man để biết các flag hữu ích khác có thể được bao gồm.)

Đây là một ví dụ đầy đủ hơn về cách lấy con trỏ đến một vùng nhớ dùng chung:

```
key_t key;
int shmid;
char *data;

key = ftok("/home/beej/somefile3", 'R');
shmid = shmget(key, 1024, 0644 | IPC_CREAT);
data = shmat(shmid, (void *)0, 0);
```

Và *boom*! Bạn đã có con trỏ đến vùng nhớ dùng chung! Lưu ý rằng `shmat()` trả về một con trỏ `void`, và ta đang xử lý nó, trong trường hợp này, như một con trỏ `char`. Bạn có thể xử lý nó như bất cứ thứ gì bạn muốn, tùy thuộc vào loại dữ liệu bạn có trong đó. Con trỏ đến mảng của các cấu trúc đều được chấp nhận như bất cứ thứ gì khác.

Ngoài ra, điều thú vị cần lưu ý là `shmat()` trả về `-1` khi thất bại (như `mmap()`). Nhưng làm thế nào bạn lấy `-1` trong một con trỏ `void`? Chỉ cần cast trong quá trình so sánh để kiểm tra lỗi:

```
data = shmat(shmid, (void *)0, 0);
if (data == MAP_FAILED)
    perror("shmat");
```

(Điều quan trọng cần lưu ý là số nguyên đang được cast thành con trỏ, không phải giá trị trả về con trỏ đang được cast thành số nguyên. Đó là sự khác biệt tinh tế, nhưng cái sau không phải lúc nào cũng khả chuyển giữa các kiến trúc. Cũng lưu ý rằng việc cast là sang `void*` chứ không phải `char*`, như bạn có thể mong đợi. Vì ngôn ngữ đảm bảo rằng các cast ẩn từ `void*` sang bất kỳ loại con trỏ nào khác luôn an toàn và đáng tin cậy, tốt hơn là dùng `void*` và để compiler làm việc.)

Tất cả những gì bạn phải làm bây giờ là thay đổi dữ liệu nó trỏ đến theo kiểu con trỏ thông thường. Có một số mẫu trong phần tiếp theo.

11.3 Đọc và Ghi

Giả sử bạn có con trỏ `data` từ ví dụ trên. Đó là con trỏ `char`, vì vậy ta sẽ đọc và ghi char từ nó. Hơn nữa, để đơn giản, giả sử vùng nhớ dùng chung 1K chứa một chuỗi kết thúc null.

Không thể đơn giản hơn. Vì đó chỉ là một chuỗi trong đó, ta có thể in nó như thế này:

```
printf("shared contents: %s\n", data);
```

Và ta có thể lưu thứ gì đó vào đó dễ dàng như thế này:

```
printf("Enter a string: ");
fgets(data, 1024, stdin);
```

Tất nhiên, như tôi đã nói trước đó, bạn có thể có dữ liệu khác trong đó chứ không chỉ `char`. Tôi chỉ dùng chúng làm ví dụ. Tôi chỉ giả định rằng bạn đã đủ quen với con trỏ trong C để có thể xử lý bất kỳ loại dữ liệu nào bạn nhét vào đó.

11.4 Tách Ra Và Xóa Vùng

Khi bạn dùng xong vùng nhớ dùng chung, chương trình của bạn nên tách bản thân ra khỏi nó bằng lệnh gọi `shmdt()` (nếu bạn không làm, điều này sẽ tự động xảy ra khi tiến trình kết thúc):

```
int shmdt(void *`shmaddr`);
```

Đối số duy nhất, `shmaddr`, là địa chỉ bạn lấy từ `shmat()`. Hàm trả về `-1` khi lỗi, `0` khi thành công.

Khi bạn tách ra khỏi vùng, nó không bị hủy. Cũng không bị xóa khi *mọi người* tách ra khỏi nó. Bạn phải xóa nó một cách cụ thể bằng lệnh gọi `shmctl()`, tương tự như các lệnh gọi kiểm soát cho các hàm System V IPC khác:

```
shmctl(shmid, IPC_RMID, NULL);
```

Lệnh gọi trên xóa vùng nhớ dùng chung, giả sử không có ai khác gắn vào nó. Hàm `shmctl()` làm được nhiều hơn thế, và đáng để tìm hiểu. (Theo cách riêng của bạn, tất nhiên, vì đây chỉ là tổng quan!)

Như thường lệ, bạn có thể xóa vùng nhớ dùng chung từ dòng lệnh bằng lệnh Unix `ipcrm`. Ngoài ra, hãy đảm bảo rằng bạn không để lại bất kỳ vùng nhớ dùng chung nào không dùng đến đang lãng phí tài nguyên hệ thống. Tất cả các đối tượng System V IPC bạn sở hữu có thể được xem bằng lệnh `ipcs`.

11.5 Đồng Thời

Các vấn đề đồng thời là gì? Vâng, vì bạn có nhiều tiến trình sửa đổi vùng nhớ dùng chung, một số lỗi có thể xảy ra khi các cập nhật vào vùng xảy ra đồng thời. Truy cập *đồng thời* này hầu như luôn là vấn đề khi bạn có nhiều writer vào một đối tượng dùng chung.

Cách giải quyết là dùng Semaphore để khóa vùng nhớ dùng chung trong khi một tiến trình đang ghi vào nó. (Đôi khi khóa sẽ bao gồm cả việc đọc và ghi vào bộ nhớ dùng chung, tùy thuộc vào những gì bạn đang làm.)

Một cuộc thảo luận thực sự về tính đồng thời nằm ngoài phạm vi của tài liệu này, và bạn có thể muốn xem bài viết Wikipedia về vấn đề này[1]. Tôi chỉ để lại điều này: nếu bạn bắt đầu thấy sự không nhất quán kỳ lạ trong dữ liệu dùng chung của mình khi bạn kết nối hai tiến trình trở lên vào nó, rất có thể bạn đang có vấn đề đồng thời.

11.6 Code Mẫu

Bây giờ tôi đã chuẩn bị cho bạn về tất cả các mối nguy hiểm của truy cập đồng thời vào vùng nhớ dùng chung mà không dùng semaphore, tôi sẽ cho bạn xem một bản demo làm chính xác điều đó. Vì đây không phải ứng dụng quan trọng, và ít có khả năng bạn đang truy cập dữ liệu dùng chung cùng lúc với tiến trình khác, tôi sẽ bỏ semaphore ra để đơn giản.

Chương trình này làm một trong hai điều: nếu bạn chạy nó mà không có tham số dòng lệnh, nó in nội dung của vùng nhớ dùng chung. Nếu bạn đưa cho nó một tham số dòng lệnh, nó lưu tham số đó vào vùng nhớ dùng chung.

Đây là code cho `shmdemo.c`[2]:

[1] https://en.wikipedia.org/wiki/Concurrency
[2] https://beej.us/guide/bgipc/source/examples/shmdemo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
            fprintf(stderr, "usage: shmdemo [data_to_write]\n");
            exit(1);
    }

    /* make the key: */
    if ((key = ftok("shmdemo.c", 'R')) == -1) {
            perror("ftok");
            exit(1);
    }

    /* connect to (and possibly create) the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
            perror("shmget");
            exit(1);
    }

    /* attach to the segment to get a pointer to it: */
    data = shmat(shmid, (void *)0, 0);

    /* we _could_ use MAP_FAILED, but technically that's not */
    /* the defined return value. System V failed on this one! */
    if (data == (void *)(-1)) {
            perror("shmat");
            exit(1);
    }

    /* read or modify the segment, based on the command line: */
    if (argc == 2) {
            printf("writing to segment: \"%s\"\n", argv[1]);
            strncpy(data, argv[1], SHM_SIZE);
            data[SHM_SIZE-1] = '\0';
    } else
            printf("segment contains: \"%s\"\n", data);

    /* detach from the segment: */
    if (shmdt(data) == -1) {
            perror("shmdt");
            exit(1);
    }
```

```
    return 0;
}
```

Thông thường hơn, một tiến trình sẽ gắn vào vùng và chạy một lúc trong khi các chương trình khác đang thay đổi và đọc vùng dùng chung. Thú vị khi xem một tiến trình cập nhật vùng và thấy các thay đổi xuất hiện với các tiến trình khác. Một lần nữa, để đơn giản, code mẫu không làm điều đó, nhưng bạn có thể thấy dữ liệu được chia sẻ giữa các tiến trình độc lập như thế nào.

Ngoài ra, không có code nào ở đây để xóa vùng—hãy đảm bảo làm điều đó khi bạn dùng xong.

Chapter 12

File Được Ánh Xạ Bộ Nhớ

Có một lúc nào đó bạn muốn đọc và ghi từ và vào các file để thông tin được chia sẻ giữa các tiến trình. Hãy nghĩ theo cách này: hai tiến trình cùng mở một file và cùng đọc và ghi từ nó, do đó chia sẻ thông tin. Vấn đề là, đôi khi thật phiền phức khi phải thực hiện tất cả những `fseek()` và những thứ tương tự để di chuyển xung quanh. Sẽ dễ dàng hơn nếu bạn có thể chỉ ánh xạ một phần của file vào bộ nhớ, và lấy một con trỏ đến nó? Rồi bạn có thể đơn giản sử dụng phép tính số học con trỏ để lấy (và đặt) dữ liệu trong file.

Vâng, đây chính xác là một file được ánh xạ bộ nhớ. Phần thú vị là khi bạn thực hiện thay đổi vào bộ nhớ (bằng cách thay đổi những thứ mà con trỏ trỏ đến), *nó thực sự thay đổi chính file đó.* Bộ nhớ đột nhiên trở thành một cửa sổ nhìn vào file và bạn có thể thay đổi nó trực tiếp qua cửa sổ đó.

Và nó thực sự rất dễ sử dụng nữa. Một vài lệnh gọi đơn giản, kết hợp với một vài quy tắc đơn giản, và bạn đang ánh xạ như người điên.

12.1 Bắt Đầu

Trước khi ánh xạ file vào bộ nhớ, bạn cần lấy một file descriptor cho nó bằng cách sử dụng syscall `open()`:

```
int fd;

fd = open("mapdemofile", O_RDWR);
```

Trong ví dụ này, ta đã mở file để truy cập đọc/ghi. Bạn có thể mở nó ở bất kỳ chế độ nào bạn muốn, nhưng nó phải khớp với chế độ được chỉ định trong tham số `prot` của lệnh gọi `mmap()` bên dưới.

Để ánh xạ bộ nhớ cho file, bạn dùng syscall `mmap()`, được định nghĩa như sau:

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int fildes, off_t off);
```

Thật nhiều tham số! Đây là từng cái một:

Tham số	Mô tả
`addr`	Đây là địa chỉ ta muốn file được ánh xạ vào. Cách tốt nhất để dùng cái này là đặt nó thành `NULL` và để hệ điều hành chọn cho bạn. Nếu bạn bảo nó dùng địa chỉ mà hệ điều hành không thích (ví dụ nếu nó không phải bội số của kích thước trang bộ nhớ ảo), nó sẽ báo lỗi.

Tham số	Mô tả
`len`	Tham số này là độ dài dữ liệu ta muốn ánh xạ vào bộ nhớ. Có thể là bất kỳ độ dài nào bạn muốn. (Lưu ý: nếu `len` không phải bội số của kích thước trang bộ nhớ ảo, bạn sẽ nhận được một kích thước khối được làm tròn lên đến kích thước đó. Các byte thêm sẽ là 0, và bất kỳ thay đổi nào bạn thực hiện với chúng sẽ không sửa đổi file.)
`prot`	Đối số "bảo vệ" cho phép bạn chỉ định loại truy cập tiến trình này có đối với vùng được ánh xạ bộ nhớ. Đây có thể là sự kết hợp OR theo bit của các giá trị sau: `PROT_READ`, `PROT_WRITE`, và `PROT_EXEC`, lần lượt cho quyền đọc, ghi, và thực thi. Giá trị được chỉ định ở đây phải tương đương hoặc là tập con của các chế độ được chỉ định trong syscall `open()` được dùng để lấy file descriptor.
`flags`	Đây chỉ là các flag linh tinh có thể được đặt cho syscall. Bạn sẽ muốn đặt nó thành `MAP_SHARED` nếu bạn định chia sẻ các thay đổi của mình vào file với các tiến trình khác, hoặc `MAP_PRIVATE` trong trường hợp khác. Nếu bạn đặt nó thành cái sau, tiến trình của bạn sẽ nhận được một bản sao của vùng được ánh xạ, vì vậy bất kỳ thay đổi nào bạn thực hiện sẽ không được phản ánh trong file gốc—do đó, các tiến trình khác sẽ không thể thấy chúng. Ta sẽ không nói về `MAP_PRIVATE` ở đây, vì nó không liên quan nhiều đến IPC.
`fildes`	Đây là nơi bạn đặt file descriptor bạn đã mở trước đó.
`off`	Đây là offset trong file mà bạn muốn bắt đầu ánh xạ từ đó. Một hạn chế: offset này *phải* là bội số của kích thước trang bộ nhớ ảo. Kích thước trang này có thể lấy bằng lệnh gọi `getpagesize()`. Lưu ý rằng các hệ thống 32-bit có thể hỗ trợ các file có kích thước không thể biểu diễn bằng số nguyên không dấu 32-bit, vì vậy kiểu này thường là kiểu 64-bit trên các hệ thống như vậy.

Về giá trị trả về, như bạn có thể đã đoán, `mmap()` trả về `MAP_FAILED` khi lỗi (giá trị `-1` được cast phù hợp để so sánh), và đặt `errno`. Ngược lại, nó trả về con trỏ đến điểm bắt đầu dữ liệu được ánh xạ.

Dù sao, không dài dòng thêm nữa, ta sẽ làm một bản demo ngắn ánh xạ "trang" thứ hai của file vào bộ nhớ. Đầu tiên ta sẽ `open()` nó để lấy file descriptor, rồi ta sẽ dùng `getpagesize()` để lấy kích thước của một trang bộ nhớ ảo và sử dụng giá trị này cho cả `len` và `off`. Theo cách này, ta sẽ bắt đầu ánh xạ từ trang thứ hai, và ánh xạ trong độ dài một trang. (Trên máy Linux của tôi, kích thước trang là 4K.)

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

int fd, pagesize;
char *data;

fd = open("foo", O_RDONLY);
pagesize = getpagesize();
data = mmap((void*)0, pagesize, PROT_READ, MAP_SHARED, fd, pagesize);
```

Sau khi đoạn code này chạy, bạn có thể truy cập byte đầu tiên của phần được ánh xạ của file bằng `data[0]`. Lưu ý có nhiều phép cast kiểu xảy ra ở đây. Ví dụ, `mmap()` trả về `void*`, nhưng ta xử lý nó như `char*`.

Ngoài ra hãy lưu ý rằng ta đã ánh xạ file `PROT_READ` nên ta có quyền truy cập chỉ đọc. Bất kỳ nỗ lực nào ghi vào dữ liệu (`data[0] = 'B'`, ví dụ) sẽ gây ra vi phạm phân đoạn. Mở file `O_RDWR` với `prot` được đặt thành `PROT_READ|PROT_WRITE` nếu bạn muốn truy cập đọc-ghi vào dữ liệu.

12.2 Hủy Ánh Xạ File

Tất nhiên có một hàm `munmap()` để hủy ánh xạ bộ nhớ cho file:

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

Hàm này đơn giản hủy ánh xạ vùng trỏ bởi `addr` (được trả về từ `mmap()`) với độ dài `len` (giống với `len` được truyền vào `mmap()`). `munmap()` trả về `-1` khi lỗi và đặt biến `errno` .

Sau khi bạn hủy ánh xạ file, bất kỳ nỗ lực nào truy cập dữ liệu qua con trỏ cũ sẽ gây ra lỗi phân đoạn. Bạn đã được cảnh báo!

Một lưu ý cuối: file sẽ tự động hủy ánh xạ khi chương trình của bạn thoát, tất nhiên.

12.3 Đồng Thời, Lại Nữa?!

Nếu bạn có nhiều tiến trình thao tác dữ liệu trong cùng một file đồng thời, bạn có thể gặp rắc rối. Bạn có thể phải khóa file hoặc dùng semaphore để điều phối truy cập vào file trong khi một tiến trình can thiệp vào nó. Hãy xem tài liệu Bộ Nhớ Dùng Chung để có thêm (rất ít) thông tin về tính đồng thời.

12.4 Một Mẫu Đơn Giản

Vâng, lại đến lúc code rồi. Tôi có ở đây một chương trình demo ánh xạ mã nguồn của chính nó vào bộ nhớ và in byte tìm thấy ở bất kỳ offset nào bạn chỉ định trên dòng lệnh.

Chương trình hạn chế các offset bạn có thể chỉ định trong phạm vi 0 đến độ dài file. Độ dài file được lấy qua lệnh gọi `stat()` mà bạn có thể chưa thấy trước đây. Nó trả về một cấu trúc đầy thông tin file, một trường trong đó là kích thước tính bằng byte. Đơn giản thôi.

Đây là mã nguồn cho `mmapdemo.c` [1]:

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

int main(int argc, char *argv[])
{
    int fd;
    off_t offset;
    char *data;
    struct stat sbuf;

    if (argc != 2) {
        fprintf(stderr, "usage: mmapdemo offset\n");
        exit(1);
    }

    if ((fd = open("mmapdemo.c", O_RDONLY)) == -1) {
        perror("open");
        exit(1);
    }

    if (stat("mmapdemo.c", &sbuf) == -1) {
        perror("stat");
        exit(1);
```

[1] https://beej.us/guide/bgipc/source/examples/mmapdemo.c

```
    }


    offset = atoi(argv[1]);
    if (offset < 0 || offset > sbuf.st_size-1) {
        fprintf(stderr, "mmapdemo: offset must be in the range 0-%d\n", \
                                                        sbuf.st_size-1);
        exit(1);
    }

    data = mmap((caddr_t)0, sbuf.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    printf("byte at offset %ld is '%c'\n", offset, data[offset]);

    return 0;
}
```

Đó là tất cả những gì cần làm. Biên dịch cái đó và chạy với một số dòng lệnh như:

```
$ mmapdemo 30
byte at offset 30 is 'e'
```

Tôi để lại cho bạn viết một số chương trình thực sự thú vị bằng syscall này.

12.5 Ánh Xạ Bộ Nhớ Ẩn Danh

Bạn có thể `mmap()` một vùng **không** được hỗ trợ bởi file. Đó chỉ là một vùng nhớ được đặt về không mà bạn đột nhiên có quyền truy cập. Có vẻ giống `malloc()`, nhưng, như ta sẽ thấy, có một điểm khác biệt quan trọng lớn.

Có vẻ kỳ lạ khi muốn làm điều này—các thay đổi bạn thực hiện chỉ tồn tại trong bộ nhớ và không được lưu trên đĩa—nhưng nó thực sự cung cấp một cách hay để thiết lập bộ nhớ dùng chung giữa các tiến trình liên quan.

> **Lưu ý rằng điều này không được POSIX hỗ trợ**. Điều đó nói, nó được định nghĩa trên Linux và BSD (bao gồm MacOS), vì vậy ta có độ phủ khá tốt trên tất cả các nền tảng phổ biến.
>
> Lưu ý #1: Một số nền tảng trước đây định nghĩa `MAP_ANON`, nhưng tôi nghĩ hầu hết đã chuyển sang `MAP_ANONYMOUS`. Vì vậy cái sau là lựa chọn khả chuyển hơn.
>
> Lưu ý #2: Một số nền tảng không quan tâm bạn chỉ định gì là file descriptor với các ánh xạ ẩn danh, nhưng MacOS muốn nó là `-1`, vì vậy hãy dùng giá trị đó.

(Một cách dùng khác cho loại `mmap()` này là nếu bạn đang viết bộ cấp phát bộ nhớ riêng của mình tương tự `malloc()`. Trong trường hợp đó, bạn sẽ cần lấy các khối bộ nhớ trực tiếp từ hệ điều hành, và `mmap()` ẩn danh là một cách tuyệt vời để làm điều đó. Nhưng đó không phải IPC, vì vậy ta sẽ không đi vào đó.)

Hãy làm một bản demo. Chương trình này sẽ:

1. Tạo một khối bộ nhớ dùng chung, ẩn danh (tức là không được hỗ trợ bởi file) bằng `mmap()`.
2. Fork một tiến trình con sẽ:
    - Ngủ một giây
    - In những gì có trong bộ nhớ dùng chung.
3. Tiến trình cha sẽ:

- Lưu một chuỗi vào bộ nhớ dùng chung.
- Đợi tiến trình con hoàn tất.

4. Sau đó cả cha và con sẽ `munmap()` bộ nhớ, giải phóng nó.

Điểm khác biệt con voi trong phòng lớn là ta không gọi `open()` ở bất cứ đâu, và ta không có file descriptor để truyền vào `mmap()`. Ta sẽ chỉ đặt cái đó thành `-1` và offset thành `0`.

Đây là mã nguồn cho `mmap_anon.c` [2]:

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

#define DATA_LEN 128 // bytes

#ifndef MAP_ANONYMOUS
#define MAP_ANONYMOUS MAP_ANON
#endif

int main(void)
{
    char *data = mmap(NULL, DATA_LEN, PROT_READ|PROT_WRITE,
                      MAP_SHARED|MAP_ANONYMOUS, -1, 0);

    if (data == NULL) {
        perror("mmap");
        return 1;
    }

    switch (fork()) {
        case -1:
            perror("fork");
            return 1;

        case 0:
            puts("child: sleeping");
            // Snooze so it's very likely the parent wins the race
            sleep(1);
            puts("child: reading");
            printf("child: %s\n", data);
            break;

        default:
            puts("parent: writing");
            strcpy(data, "Hello from shared memory!");
            puts("parent: waiting");
            wait(NULL);
            break;
    }

    munmap(data, 128);
}
```

"Đồng thời, lại nữa, lại nữa?!" Bạn có thể nhận thấy rằng thực sự không có sự đồng bộ hóa nào giữa cha và con. Tôi chỉ lách qua bằng cách để con ngủ để ta có thể khá chắc chắn rằng cha đã ghi dữ liệu

[2] https://beej.us/guide/bgipc/source/examples/mmap_anon.c

xong. Điều đó thực sự không đủ tốt theo bất kỳ nghĩa thực tế nào, vì vậy bạn có thể phải làm thêm với semaphore hoặc thứ gì đó tương tự để có sự phối hợp bạn cần để không có mọi thứ nổ tung.

Và thực ra, nếu bạn có quyền truy cập `pthreads`, chỉ cần dùng nó trong trường hợp này. Mọi thứ khác chỉ là tái phát minh cái bánh xe đó.

12.6 Nhận Xét Về Ánh Xạ Bộ Nhớ

Tôi sẽ thiếu sót nếu không chỉ ra một vài khía cạnh thú vị của việc sử dụng file được ánh xạ trên Linux. Đầu tiên, bộ nhớ mà hệ điều hành phân bổ để sử dụng làm bộ nhớ đệm cho dữ liệu file được ánh xạ là *cùng bộ nhớ* được sử dụng để thực hiện các thao tác đệm file khi các tiến trình khác thực hiện các thao tác `read()` và `write()`! Trong khi các `read()` và `write()` được đảm bảo là atomic bởi POSIX đến một kích thước nhất định, điều đó sẽ bị phá vỡ khi một số tiến trình bỏ qua hoàn toàn các hàm POSIX!

Thứ hai, vì ta đang bỏ qua các hàm POSIX đó, ta có thể đọc và ghi nội dung buffer mà không cần quan tâm đến việc khóa bản ghi có thể được áp dụng cho file descriptor (như đã thảo luận trong một phần trước). Thông thường, điều này không phải là vấn đề lớn—ai sẽ dùng file được ánh xạ bộ nhớ trong một ứng dụng trong khi dùng khóa bản ghi trong ứng dụng khác, khi cả hai đều truy cập cùng file? Nếu file được ghi lại yêu cầu khóa bản ghi, thì tất cả các ứng dụng nên dùng nó. Điều đó nói, không có gì ngăn một ứng dụng sử dụng khóa đọc và ghi ta đã thảo luận trước đó ngay trước khi cập nhật bộ nhớ thuộc về file được ánh xạ.

Thứ ba, vì ta đang bỏ qua các hàm POSIX đó (tôi nghe có vẻ như đĩa hát bị rách không?), hệ thống không có khả năng cung cấp các chiến lược readahead hoặc writebehind có ý nghĩa. Tính đến thời điểm viết bài này, các phiên bản kernel Linux 4.x trở lên *có* triển khai một thuật toán phát hiện khi hai page fault liền nhau xảy ra trong một file được ánh xạ bộ nhớ, và nó thực hiện một lượng readahead tối thiểu (chỉ hai trang, so với readahead có thể cấu hình ở lớp hệ thống file, có thể lên đến 256KB). Hoàn toàn không có writebehind, vì không có cách thực tế nào để phát hiện khi các trang liền kề được ghi dưới các cấu hình phần cứng hiện tại.

Cuối cùng, với tất cả những điều trên, vẫn có những lý do rất hấp dẫn để sử dụng file được ánh xạ bộ nhớ. Lý do chính là các file như vậy, theo định nghĩa, là "bộ nhớ lưu trữ lâu dài", có nghĩa là các ứng dụng không phải tạo các hàm `load()` / `save()` dài dòng cho dữ liệu của chúng nếu chúng sử dụng file được ánh xạ bộ nhớ. Tuy nhiên, bất kỳ dữ liệu nhị phân nào sẽ được ghi theo cách phụ thuộc nền tảng (như thứ tự byte) vì vậy các file đó có thể không khả chuyển.

12.7 Tóm Tắt

File được ánh xạ bộ nhớ có thể rất hữu ích, đặc biệt trên các hệ thống không hỗ trợ các vùng nhớ dùng chung. Thực ra, cả hai rất giống nhau trong hầu hết các khía cạnh. (File được ánh xạ bộ nhớ cũng được commit vào đĩa, vì vậy đây thậm chí có thể là một lợi thế, phải không?) Với khóa file hoặc semaphore, dữ liệu trong một file được ánh xạ bộ nhớ có thể dễ dàng được chia sẻ giữa nhiều tiến trình.

Chapter 13

Unix Socket

Bạn còn nhớ FIFO không? Bạn còn nhớ cách chúng chỉ có thể gửi dữ liệu theo một chiều, giống như Pipe không? Sẽ tuyệt không nếu bạn có thể gửi dữ liệu theo cả hai chiều như với socket?

Vâng, đừng lo nữa, vì đây là câu trả lời: Unix Domain Socket! Trong trường hợp bạn vẫn đang tự hỏi socket là gì, vâng, đó là một đường ống giao tiếp hai chiều, có thể được dùng để giao tiếp qua nhiều *domain* khác nhau. Một trong những domain phổ biến nhất mà socket giao tiếp là Internet, nhưng ta sẽ không bàn đến điều đó ở đây. Tuy nhiên, ta sẽ nói về socket trong domain Unix; tức là, các socket có thể được dùng giữa các tiến trình trên cùng một hệ thống Unix.

Unix socket dùng nhiều lệnh gọi hàm giống như Internet socket, và tôi sẽ không mô tả chi tiết tất cả các lệnh gọi tôi dùng trong tài liệu này. Nếu mô tả về một lệnh gọi cụ thể quá mơ hồ (hoặc nếu bạn chỉ muốn tìm hiểu thêm về Internet socket dù sao), tôi tùy tiện đề xuất Hướng Dẫn Lập Trình Mạng của Beej sử dụng Internet Socket[1]. Tôi biết tác giả đó rất rõ.

13.1 Tổng Quan

Như tôi đã nói trước đó, Unix socket giống như FIFO hai chiều. Tuy nhiên, tất cả giao tiếp dữ liệu sẽ diễn ra qua giao diện socket, thay vì qua giao diện file. Mặc dù Unix socket là một file đặc biệt trong hệ thống file (giống như FIFO), bạn sẽ không dùng `open()` và `read()` —bạn sẽ dùng `socket()`, `bind()`, `recv()`, v.v.

Khi lập trình với socket, bạn thường tạo các chương trình server và client. Server sẽ ngồi lắng nghe các kết nối đến từ client và xử lý chúng. Điều này rất giống với tình huống tồn tại với Internet socket, nhưng có một số khác biệt tinh tế.

Ví dụ, khi mô tả Unix socket nào bạn muốn dùng (tức là đường dẫn đến file đặc biệt là socket), bạn dùng `struct sockaddr_un`, có các trường sau:

```
struct sockaddr_un {
    unsigned short sun_family;  /* AF_UNIX */
    char sun_path[108];
}
```

Đây là cấu trúc bạn sẽ truyền vào hàm `bind()`, hàm liên kết một socket descriptor (một file descriptor) với một file nhất định (tên của file đó nằm trong trường `sun_path`).

13.2 Các Bước Để Là Server

Không đi vào quá nhiều chi tiết, tôi sẽ phác thảo các bước một chương trình server thường phải thực hiện. Trong khi đó, tôi sẽ cố triển khai một "echo server" chỉ đơn giản là echo lại mọi thứ nó nhận được trên socket.

[1] https://beej.us/guide/bgnet

Đây là các bước của server:

1. **Gọi `socket()`**: Một lệnh gọi `socket()` với các đối số đúng sẽ tạo Unix socket:

   ```
   unsigned int s, s2;

   struct sockaddr_un remote, local = {
       .sun_family = AF_UNIX,
       // .sun_path = SOCK_PATH,   // Can't do assignment to an array
   };

   int len;

   s = socket(_AF_UNIX_, SOCK_STREAM, 0);
   ```

   Đối số thứ hai, `SOCK_STREAM`, cho `socket()` biết tạo một stream socket. Vâng, datagram socket (`SOCK_DGRAM`) được hỗ trợ trong domain Unix, nhưng tôi chỉ đề cập đến stream socket ở đây. Những ai tò mò, hãy xem Hướng Dẫn Lập Trình Mạng của Beej[2] để có mô tả tốt về unconnected datagram socket áp dụng hoàn hảo cho Unix socket. Điều duy nhất thay đổi là bạn đang dùng `struct sockaddr_un` thay vì `struct sockaddr_in`.

   Thêm một lưu ý: tất cả các lệnh gọi này trả về `-1` khi lỗi và đặt biến toàn cục `errno` để phản ánh những gì đã xảy ra. Hãy đảm bảo kiểm tra lỗi.

2. **Gọi `bind()`**: Bạn đã lấy được socket descriptor từ lệnh gọi `socket()`, bây giờ bạn muốn bind nó vào một địa chỉ trong domain Unix. (Địa chỉ đó, như tôi đã nói trước đây, là một file đặc biệt trên đĩa.)

   ```
   strcpy(local.sun_path, "/home/beej/mysocket");
   unlink(local.sun_path);
   len = strlen(local.sun_path) + sizeof(local.sun_family);

   bind(s, (struct sockaddr *)&local, len);
   ```

   Lệnh này liên kết socket descriptor "`s`" với địa chỉ Unix socket "`/home/beej/mysocket`". Lưu ý rằng ta đã gọi `unlink()` trước `bind()` để xóa socket nếu nó đã tồn tại. Bạn sẽ nhận lỗi `EINVAL` nếu file đó đã có ở đó.

3. **Gọi `listen()`**: Lệnh này hướng dẫn socket lắng nghe các kết nối đến từ các chương trình client:

   ```
   listen(s, 5);
   ```

   Đối số thứ hai, `5`, là số kết nối đến có thể được xếp hàng đợi trước khi bạn gọi `accept()` bên dưới. Nếu có nhiều kết nối đang chờ được chấp nhận như vậy, các client thêm sẽ tạo ra lỗi `ECONNREFUSED`.

4. **Gọi `accept()`**: Lệnh này sẽ chấp nhận một kết nối từ client. Hàm này trả về *một socket descriptor khác*! Descriptor cũ vẫn đang lắng nghe các kết nối mới, nhưng cái mới này được kết nối với client:

   ```
   len = sizeof(remote);
   s2 = accept(s, &remote, &len);
   ```

   Khi `accept()` trả về, biến `remote` sẽ được điền với `struct sockaddr_un` phía bên kia, và `len` sẽ được đặt thành độ dài của nó. Descriptor `s2` được kết nối với client, và sẵn sàng cho `send()` và `recv()`, như được mô tả trong Hướng Dẫn Lập Trình Mạng[3].

[2] https://beej.us/guide/bgnet
[3] https://beej.us/guide/bgnet

5. **Xử lý kết nối và quay lại `accept()`**: Thường bạn sẽ muốn giao tiếp với client ở đây (ta chỉ echo lại mọi thứ nó gửi cho ta), đóng kết nối, rồi `accept()` một cái mới.

```
while (len = recv(s2, &buf, 100, 0), len > 0)
    send(s2, &buf, len, 0);

/* loop back to accept() from here */
```

6. **Đóng kết nối:** Bạn có thể đóng kết nối bằng cách gọi `close()`, hoặc bằng cách gọi `shutdown`[4].

Sau tất cả những điều đó, đây là một số code cho echo server, `echos.c`[5]. Tất cả những gì nó làm là chờ kết nối trên một Unix socket (có tên, trong trường hợp này, là "echo_socket").

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
    int s, s2, len;
    struct sockaddr_un remote, local = {
            .sun_family = AF_UNIX,
            // .sun_path = SOCK_PATH,   // Can't do assignment to an array
    };
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
    }

    strcpy(local.sun_path, SOCK_PATH);
    unlink(local.sun_path);
    len = strlen(local.sun_path) + sizeof(local.sun_family);
    if (bind(s, (struct sockaddr *)&local, len) == -1) {
            perror("bind");
            exit(1);
    }

    if (listen(s, 5) == -1) {
            perror("listen");
            exit(1);
    }

    for(;;) {
            int done, n;
            printf("Waiting for a connection...\n");
            socklen_t slen = sizeof(remote);
```

[4] https://man.archlinux.org/man/shutdown.2
[5] https://beej.us/guide/bgipc/source/examples/echos.c

```
            if ((s2 = accept(s, (struct sockaddr *)&remote, &slen)) == -1) {
                    perror("accept");
                    exit(1);
            }

            printf("Connected.\n");

            done = 0;
            do {
                    n = recv(s2, str, sizeof(str), 0);
                    if (n <= 0) {
                            if (n < 0) perror("recv");
                            done = 1;
                    }

                    if (!done)
                            if (send(s2, str, n, 0) < 0) {
                                    perror("send");
                                    done = 1;
                            }
            } while (!done);

            close(s2);
    }

    return 0;
}
```

Như bạn có thể thấy, tất cả các bước đã đề cập ở trên đều có trong chương trình này: gọi `socket()`, gọi `bind()`, gọi `listen()`, gọi `accept()`, và thực hiện một số `send()` và `recv()` trên mạng.

13.3 Các Bước Để Là Client

Cần có một chương trình để nói chuyện với server ở trên, phải không? Ngoại trừ với client, nó dễ hơn nhiều vì bạn không phải làm những thứ `listen()` và `accept()` phiền phức. Đây là các bước:

1. Gọi `socket()` để lấy một Unix domain socket để giao tiếp qua.
2. Thiết lập một `struct sockaddr_un` với địa chỉ từ xa (nơi server đang lắng nghe) và gọi `connect()` với nó như một đối số.
3. Giả sử không có lỗi, bạn đã kết nối với phía bên kia! Dùng `send()` và `recv()` tùy thích!

Thế nào về code để nói chuyện với echo server ở trên? Không vấn đề gì, bạn bè ơi, đây là `echoc.c`[6]:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define SOCK_PATH "echo_socket"
```

[6] https://beej.us/guide/bgipc/source/examples/echoc.c

```
int main(void)
{
    int s, len;
    struct sockaddr_un remote = {
        .sun_family = AF_UNIX,
        // .sun_path = SOCK_PATH,   // Can't do assignment to an array
    };
    char str[100];

    if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    printf("Trying to connect...\n");

    strcpy(remote.sun_path, SOCK_PATH);
    len = strlen(remote.sun_path) + sizeof(remote.sun_family);
    if (connect(s, (struct sockaddr *)&remote, len) == -1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    /* size in fgets() includes the null byte */
    while(printf("> "), fgets(str, sizeof(str), stdin), !feof(stdin)) {
        if (send(s, str, strlen(str)+1, 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((len=recv(s, str, sizeof(str)-1, 0)) > 0) {
            str[len] = '\0';
            printf("echo> %s", str);
        } else {
            if (len < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}
```

Trong code client, tất nhiên bạn sẽ nhận thấy chỉ có một vài syscall được dùng để thiết lập mọi thứ: `socket()` và `connect()`. Vì client sẽ không `accept()` bất kỳ kết nối đến nào, không cần phải `listen()`. Tất nhiên, client vẫn dùng `send()` và `recv()` để truyền dữ liệu. Đó là tóm tắt.

13.4 `socketpair()`—Pipe Full-Duplex Nhanh

Nếu bạn muốn một `pipe()`, nhưng muốn dùng một pipe duy nhất để gửi và nhận dữ liệu từ *cả hai phía*? Vì pipe là một chiều (với ngoại lệ trong SYSV), bạn không thể làm được! Tuy nhiên có một giải pháp: dùng Unix domain socket, vì chúng có thể xử lý dữ liệu hai chiều.

Thật phiền phức! Thiết lập tất cả code đó với `listen()` và `connect()` và những thứ như vậy chỉ để truyền dữ liệu theo cả hai chiều! Nhưng đoán xem không! Bạn không cần phải làm vậy!

Đúng vậy, có một syscall tuyệt vời được gọi là `socketpair()`, đủ tốt bụng để trả về cho bạn một cặp *socket đã được kết nối sẵn*! Không cần thêm công việc nào từ phía bạn; bạn có thể ngay lập tức dùng các socket descriptor này để giao tiếp liên tiến trình.

Ví dụ, hãy thiết lập hai tiến trình. Cái đầu tiên gửi một `char` đến cái thứ hai, và cái thứ hai chuyển ký tự thành chữ hoa và trả về. Đây là một số code đơn giản để làm chính xác điều đó, được gọi là `spair.c`[7] (không có kiểm tra lỗi để rõ ràng hơn):

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(void)
{
    int sv[2]; /* the pair of socket descriptors */
    char buf; /* for data exchange between processes */

    if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) == -1) {
            perror("socketpair");
            exit(1);
    }

    if (!fork()) {  /* child */
            read(sv[1], &buf, 1);
            printf("child: read '%c'\n", buf);
            buf = toupper(buf);  /* make it uppercase */
            write(sv[1], &buf, 1);
            printf("child: sent '%c'\n", buf);

    } else { /* parent */
            write(sv[0], "b", 1);
            printf("parent: sent 'b'\n");
            read(sv[0], &buf, 1);
            printf("parent: read '%c'\n", buf);
            wait(NULL); /* wait for child to die */
    }

    return 0;
}
```

Đúng là đây là một cách tốn kém để chuyển ký tự sang chữ hoa, nhưng điều thực sự quan trọng là bạn có giao tiếp đơn giản đang diễn ra ở đây.

Một điều nữa cần lưu ý là `socketpair()` nhận cả domain (`AF_UNIX`) và kiểu socket (`SOCK_STREAM`). Những thứ này có thể là bất kỳ giá trị hợp lệ nào, tùy thuộc vào các thủ tục trong kernel mà bạn muốn xử lý code của mình, và liệu bạn muốn stream hay datagram socket. Tôi chọn socket `AF_UNIX` vì đây là tài liệu Unix socket và chúng nhanh hơn socket `AF_INET` một chút, theo tôi nghe.

Cuối cùng, bạn có thể tò mò tại sao tôi dùng `write()` và `read()` thay vì `send()` và `recv()`. Vâng, tóm lại, tôi đang lười biếng. Bạn thấy, bằng cách dùng các syscall này, tôi không phải nhập đối số `flags`

[7] https://beej.us/guide/bgipc/source/examples/spair.c

mà `send()` và `recv()` dùng, và tôi luôn đặt nó thành không dù sao. Tất nhiên, socket descriptor chỉ là file descriptor như bất kỳ cái nào khác, vì vậy chúng phản hồi tốt với nhiều syscall thao tác file.

Chapter 14

Tài Nguyên IPC Bổ Sung

14.1 Sách

Dưới đây là một số cuốn sách mô tả một số quy trình tôi đã thảo luận trong hướng dẫn này, cũng như các chi tiết Unix cụ thể:

Bach, Maurice J. *The Design of the UNIX Operating System.* Published by Prentice-Hall, 1986. ISBN 0132017997[1].

W. Richard Stevens. *Unix Network Programming, volumes 1-2.* Published by Prentice Hall. ISBNs for volumes 1-2: 0131411551[2], 0130810819[3].

W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Published by Addison Wesley. ISBN 0201433079[4].

14.2 Tài Liệu Trực Tuyến Khác

Trang chủ UNIX Network Programming Volume 2[5]—bao gồm mã nguồn từ cuốn sách xuất sắc của Stevens.

Hướng Dẫn Lập Trình Viên Linux[6]—phần chuyên sâu về IPC.

UNIX System Calls and Subroutines using C[7]—chứa một số thông tin IPC khiêm tốn.

Nhân Linux[8]—cách nhân Linux triển khai IPC.

14.3 Trang Man Linux

Đây là các trang hướng dẫn Linux. Nếu bạn chạy một phiên bản Unix khác, hãy xem các trang man của riêng bạn, vì những trang này có thể không hoạt động trên hệ thống của bạn.

- `accept()` [9],
- `bind()` [10],
- `connect()` [11],

[1] https://beej.us/guide/url/unixdesign
[2] https://beej.us/guide/url/unixnet1
[3] https://beej.us/guide/url/unixnet2
[4] https://beej.us/guide/url/advunix
[5] http://www.kohala.com/start/unpv22e/unpv22e.html
[6] http://tldp.org/LDP/lpg/node7.html
[7] https://users.cs.cf.ac.uk/Dave.Marshall/C/
[8] https://tldp.org/LDP/tlk/ipc/ipc.html
[9] https://man.archlinux.org/man/accept.2
[10] https://man.archlinux.org/man/bind.2
[11] https://man.archlinux.org/man/connect.2

- `dup()` [12],
- `exec()` [13],
- `exit()` [14],
- `fcntl()` [15],
- `fileno()` [16],
- `fork()` [17],
- `ftok()` [18],
- `getpagesize()` [19],
- `ipcrm` [20],
- `ipcs` [21],
- `kill` [22],
- `kill()` [23],
- `listen()` [24],
- `lockf()` [25],
- `lseek()` [26] (for the `l_whence` field in `struct flock`),
- `mknod` [27],
- `mknod()` [28],
- `mmap()` [29],
- `msgctl()` [30],
- `msgget()` [31],
- `msgsnd()` [32],
- `munmap()` [33],
- `open()` [34],
- `pipe()` [35],
- `ps` [36],
- `raise()` [37],
- `read()` [38],
- `recv()` [39],
- `semctl()` [40],

[12] https://man.archlinux.org/man/dup.2
[13] https://man.archlinux.org/man/exec.2
[14] https://man.archlinux.org/man/exit.2
[15] https://man.archlinux.org/man/fcntl.2
[16] https://man.archlinux.org/man/fileno.3
[17] https://man.archlinux.org/man/fork.2
[18] https://man.archlinux.org/man/ftok.3
[19] https://man.archlinux.org/man/getpagesize.2
[20] https://man.archlinux.org/man/ipcrm.8
[21] https://man.archlinux.org/man/ipcs.8
[22] https://man.archlinux.org/man/kill.1
[23] https://man.archlinux.org/man/kill.2
[24] https://man.archlinux.org/man/listen.2
[25] https://man.archlinux.org/man/lockf.2
[26] https://man.archlinux.org/man/lseek.2
[27] https://man.archlinux.org/man/mknod.1
[28] https://man.archlinux.org/man/mknod.2
[29] https://man.archlinux.org/man/mmap.2
[30] https://man.archlinux.org/man/msgctl.2
[31] https://man.archlinux.org/man/msgget.2
[32] https://man.archlinux.org/man/msgsnd.2
[33] https://man.archlinux.org/man/munmap.2
[34] https://man.archlinux.org/man/open.2
[35] https://man.archlinux.org/man/pipe.2
[36] https://man.archlinux.org/man/ps.1
[37] https://man.archlinux.org/man/raise.3
[38] https://man.archlinux.org/man/read.2
[39] https://man.archlinux.org/man/recv.2
[40] https://man.archlinux.org/man/semctl.2

- `semget()` [41],
- `semop()` [42],
- `send()` [43],
- `shmat()` [44],
- `shmctl()` [45],
- `shmdt()` [46],
- `shmget()` [47],
- `sigaction()` [48],
- `signal()` [49],
- signals[50],
- `sigpending()` [51],
- `sigprocmask()` [52],
- sigsetops[53],
- `sigsuspend()` [54],
- `socket()` [55],
- `socketpair()` [56],
- `stat()` [57],
- `wait()` [58],
- `waitpid()` [59],
- `write()` [60].

[41] https://man.archlinux.org/man/semget.2
[42] https://man.archlinux.org/man/semop.2
[43] https://man.archlinux.org/man/send.2
[44] https://man.archlinux.org/man/shmat.2
[45] https://man.archlinux.org/man/shmctl.2
[46] https://man.archlinux.org/man/shmdt.2
[47] https://man.archlinux.org/man/shmget.2
[48] https://man.archlinux.org/man/sigaction.2
[49] https://man.archlinux.org/man/signal.2
[50] https://man.archlinux.org/man/signal.7
[51] https://man.archlinux.org/man/sigpending.2
[52] https://man.archlinux.org/man/sigprocmask.2
[53] https://man.archlinux.org/man/sigsetopts.2
[54] https://man.archlinux.org/man/sigsuspend.2
[55] https://man.archlinux.org/man/socket.2
[56] https://man.archlinux.org/man/socketpair.2
[57] https://man.archlinux.org/man/stat.2
[58] https://man.archlinux.org/man/wait.2
[59] https://man.archlinux.org/man/waitpid.2
[60] https://man.archlinux.org/man/write.2