

[Mục 3.8. `sock_ntop` và các hàm liên quan](#)[Mục 3.9. Chức năng đọc, viết và đọc dòng](#)[Mục 3.10. Bản tóm tắt](#)[Bài tập](#)

3.1 Giới thiệu

Chuỗi này bắt đầu mô tả về API socket. Chúng ta bắt đầu với cấu trúc địa chỉ socket, cấu trúc này có trong hầu hết các ví dụ trong văn bản. Các cấu trúc này có thể được truyền theo hai hướng: từ tiến trình đến kernel và từ kernel đến tiến trình. Trường hợp sau là một ví dụ về đối số kết quả-giá trị và chúng ta sẽ gặp các ví dụ khác về các đối số này trong suốt văn bản.

Các hàm chuyển đổi địa chỉ chuyển đổi giữa biểu diễn văn bản của một địa chỉ và giá trị nhị phân đi vào cấu trúc địa chỉ ở cắm. Hầu hết mã IPv4 hiện tại đều sử dụng `inet_addr` và `inet_ntoa`, nhưng hai hàm mới, `inet_pton` và `inet_ntop`, xử lý cả IPv4 và IPv6.

Một vấn đề với các chức năng chuyển đổi địa chỉ này là chúng phụ thuộc vào loại địa chỉ được chuyển đổi: IPv4 hoặc IPv6. Chúng tôi sẽ phát triển một tập hợp các hàm có tên bắt đầu bằng `sock_` hoạt động với cấu trúc địa chỉ socket theo kiểu độc lập với giao thức. Chúng tôi sẽ sử dụng những điều này trong suốt văn bản để làm cho mã của chúng tôi trở nên độc lập với giao thức.

3.2 Cấu trúc địa chỉ socket

Hầu hết các hàm socket đều yêu cầu một con trỏ tới cấu trúc địa chỉ socket làm đối số. Mỗi bộ giao thức được hỗ trợ xác định cấu trúc địa chỉ socket riêng của nó. Tên của các cấu trúc này bắt đầu bằng `sockaddr_` và kết thúc bằng một hậu tố duy nhất cho mỗi giao thức

Thư ứng hàng.

Cấu trúc địa chỉ ở cắm IPv4

Cấu trúc địa chỉ ở cắm IPv4, thường được gọi là "cấu trúc địa chỉ ở cắm Internet", được đặt tên là `sockaddr_in` và được xác định bằng cách bao gồm `<netinet/in.h>` tiêu đề. [Hình 3.1](#) hiện định nghĩa POSIX.

Hình 3.1 Cấu trúc địa chỉ socket Internet (IPv4): `sockaddr_in`.

```

cấu trúc in_addr {
    in_addr_t s_addr; /*Địa chỉ IPv4 32-bit */
    /* byte mạng được sắp xếp */
};

cấu trúc sockaddr_in {
    uint8_t sin_len; gia đình của anh /*độ dài của cấu trúc (16) */
    sa_family_t áy; /* AF_INET */
    in_port_t sin_port; /* Số cổng TCP hoặc UDP 16 bit */
    /* byte mạng được sắp xếp */
    cấu trúc in_addr sin_addr; /*Địa chỉ IPv4 32-bit */
    /* byte mạng được sắp xếp */
    ký tự sin_zero[8]; /*không được sử dụng*/
};

```

Có một số điểm chúng ta cần làm rõ về cấu trúc địa chỉ socket nói chung bằng ví dụ này:

- Thành viên chiều dài, `sin_len`, đã được thêm vào với 4.3BSD-Reno, khi hỗ trợ cho các giao thức OSI được thêm vào ([Hình 1.15](#)). Trước bản phát hành này, thành viên đầu tiên là `sin_family`, về mặt lịch sử là một đoạn ngắn không dấu. Không phải tất cả các nhà cung cấp đều hỗ trợ trường độ dài cho cấu trúc địa chỉ ở cắm và đặc tả POSIX không yêu cầu thành viên này. Kiểu dữ liệu mà chúng tôi hiển thị, `uint8_t`, là điển hình và các hệ thống tuân thủ POSIX cung cấp các kiểu dữ liệu có dạng này ([Hình 3.2](#)).

Hình 3.2. Các kiểu dữ liệu được yêu cầu bởi đặc tả POSIX.

Datatype	Description	Header
<code>int8_t</code>	Signed 8-bit integer	<code><sys/types.h></code>
<code>uint8_t</code>	Unsigned 8-bit integer	<code><sys/types.h></code>
<code>int16_t</code>	Signed 16-bit integer	<code><sys/types.h></code>
<code>uint16_t</code>	Unsigned 16-bit integer	<code><sys/types.h></code>
<code>int32_t</code>	Signed 32-bit integer	<code><sys/types.h></code>
<code>uint32_t</code>	Unsigned 32-bit integer	<code><sys/types.h></code>
<code>sa_family_t</code>	Address family of socket address structure	<code><sys/socket.h></code>
<code>socklen_t</code>	Length of socket address structure, normally <code>uint32_t</code>	<code><sys/socket.h></code>
<code>in_addr_t</code>	IPv4 address, normally <code>uint32_t</code>	<code><netinet/in.h></code>
<code>in_port_t</code>	TCP or UDP port, normally <code>uint16_t</code>	<code><netinet/in.h></code>

Có trường độ dài giúp đơn giản hóa việc xử lý địa chỉ ở cắm có độ dài thay đổi cấu trúc.

- Ngay cả khi có trường độ dài, chúng ta không bao giờ cần thiết lập nó và không bao giờ cần kiểm tra nó, trừ khi chúng ta đang xử lý các socket định tuyến ([Chương 18](#)). Nó được sử dụng trong kernel bởi các thủ tục xử lý các cấu trúc địa chỉ socket từ các họ giao thức khác nhau (ví dụ: mã bảng định tuyến).

Bốn hàm socket chuyển cấu trúc địa chỉ socket từ tiến trình đến kernel, [liên kết](#), [kết nối](#), [sendto](#) và [sendmsg](#), tất cả đều đi qua hàm [sockargs](#) trong quá trình triển khai có nguồn gốc từ Berkeley (tr. 452 của TCPv2).

Hàm này sao chép cấu trúc địa chỉ socket từ tiến trình và đặt rõ ràng thành viên [sin_len](#) của nó theo kích thước của cấu trúc được truyền làm đối số cho bốn hàm này. Năm hàm ở cắm chuyển cấu trúc địa chỉ ở cắm từ hạt nhân đến tiến trình, [chấp nhận](#), [recvfrom](#), [recvmsg](#), [getpeername](#) và [getsockname](#), tất cả đều đặt thành viên [sin_len](#) trước khi quay lại tiến trình.

Thật không may, thông thường không có thử nghiệm thời gian biên dịch đơn giản nào để xác định xem việc triển khai có xác định truthor độ dài cho cấu trúc địa chỉ ở cắm của nó hay không. Trong mã của chúng tôi, chúng tôi kiểm tra hằng số [HAVE_SOCKADDR_SA_LEN](#) của chính mình ([Hình D.2](#)), nhưng việc xác định hằng số này hay không đòi hỏi phải cố gắng biên dịch một chương trình thử nghiệm đơn giản sử dụng phần cấu trúc tùy chọn này và xem liệu quá trình biên dịch có thành công hay không. Chúng ta sẽ thấy trong [Hình 3.4](#) rằng việc triển khai IPv6 được yêu cầu để xác định [SIN6_LEN](#) nếu cấu trúc địa chỉ ở cắm có truthor độ dài. Một số triển khai IPv4 cung cấp truthor độ dài của cấu trúc địa chỉ ở cắm cho ứng dụng dựa trên tùy chọn thời gian biên dịch (ví dụ: [_SOCKADDR_LEN](#)). Tính năng này cung cấp khả năng tương thích cho phiên bản cũ hơn các chương trình.

- Đặc tả POSIX chỉ yêu cầu ba thành viên trong cấu trúc:

[sin_family](#), [sin_addr](#) và [sin_port](#). Việc triển khai tuân thủ POSIX có thể chấp nhận được việc xác định các thành viên cấu trúc bổ sung và điều này là bình thường đối với cấu trúc địa chỉ ở cắm Internet. Hầu như tất cả các triển khai đều thêm thành viên [sin_zero](#) để tất cả các cấu trúc địa chỉ socket có kích thước tối thiểu là 16 byte.

- Chúng tôi hiển thị các kiểu dữ liệu POSIX cho [s_addr](#), [sin_family](#) và [sin_port](#) các thành viên. Kiểu dữ liệu [in_addr_t](#) phải là loại số nguyên không dấu có ít nhất 32 bit, [in_port_t](#) phải là loại số nguyên không dấu có ít nhất 16 bit và [sa_family_t](#) có thể là bất kỳ loại số nguyên không dấu nào. Số sau thư mục là số nguyên không dấu 8 bit nếu việc triển khai hỗ trợ truthor độ dài hoặc số nguyên 16 bit không dấu nếu truthor độ dài không được hỗ trợ. [Hình 3.2](#) liệt kê ba kiểu dữ liệu do POSIX xác định này, cùng với một số kiểu dữ liệu POSIX khác mà chúng ta sẽ gặp phải.
- Bạn cũng sẽ gặp các kiểu dữ liệu [u_char](#), [u_short](#), [u_int](#) và [u_long](#), tất cả đều không dấu. Đặc tả POSIX xác định những điều này với lưu ý rằng chúng đã lỗi thời. Chúng được cung cấp để tương thích ngược.
- Cả địa chỉ IPv4 và số cổng TCP hoặc UDP luôn được lưu trữ trong cấu trúc theo thứ tự byte mạng. Chúng ta phải nhận thức được điều này khi sử dụng những thành viên này. Chúng ta sẽ nói thêm về sự khác biệt giữa thứ tự byte máy chủ và thứ tự byte mạng trong [Phần 3.4](#).

- Địa chỉ IPv4 32-bit có thể được truy cập theo hai cách khác nhau. Ví dụ, nếu `serv` được định nghĩa là cấu trúc địa chỉ ở cảm Internet thì `serv.sin_addr` sẽ tham chiếu địa chỉ IPv4 32 bit dưới dạng cấu trúc `in_addr`, trong khi `serv.sin_addr.s_addr` tham chiếu cùng địa chỉ IPv4 32 bit dưới dạng `in_addr_t` (thường là địa chỉ 32- không dấu). Chúng ta phải chắc chắn rằng mình đang tham chiếu địa chỉ IPv4 một cách chính xác, đặc biệt khi địa chỉ này được sử dụng làm đối số cho một hàm, vì trình biên dịch thường truyền các cấu trúc khác với số nguyên.

Lý do thành viên `sin_addr` là một cấu trúc chứ không chỉ là `in_addr_t`, là do lịch sử. Các bản phát hành trước đó (4.2BSD) đã xác định cấu trúc `in_addr` là sự kết hợp của nhiều cấu trúc khác nhau, để cho phép truy cập vào từng byte trong số 4 byte và cả hai giá trị 16 bit có trong địa chỉ IPv4 32 bit. Cái này đã được sử dụng

với các địa chỉ lớp A, B và C để lấy các byte thích hợp của địa chỉ.

Nhưng với sự ra đời của mạng con và sau đó là sự biến mất của các lớp địa chỉ khác nhau với địa chỉ không phân lớp ([Phần A.4](#)), nhu cầu về liên minh đã biến mất. Hầu hết các hệ thống ngày nay đã loại bỏ liên kết và chỉ định nghĩa `in_addr` là một cấu trúc có một thành viên `in_addr_t` duy nhất.

- Thành phần `sin_zero` không được sử dụng như ng chúng ta luôn đặt nó về 0 khi điền vào một trong các cấu trúc này. Theo quy ước, chúng tôi luôn đặt toàn bộ cấu trúc về 0 trước khi điền vào, không chỉ thành viên `sin_zero`.

Mặc dù hầu hết các cách sử dụng cấu trúc không yêu cầu thành viên này phải là 0, nhưng khi liên kết một địa chỉ IPv4 không có ký tự đại diện, thành viên này phải là 0 (trang 731-732 của TCPv2).

- Cấu trúc địa chỉ socket chỉ được sử dụng trên một máy chủ nhất định: Bản thân cấu trúc này không được giao tiếp giữa các máy chủ khác nhau, mặc dù một số trường nhất định (ví dụ: địa chỉ IP và cổng) được sử dụng để liên lạc.

Cấu trúc địa chỉ ở cảm chung

Cấu trúc địa chỉ socket luôn được truyền bằng tham chiếu khi được truyền dưới dạng đối số cho bất kỳ hàm socket nào. Nhưng bất kỳ hàm socket nào lấy một trong các con trả này làm đối số đều phải xử lý các cấu trúc địa chỉ socket từ bất kỳ họ giao thức nào được hỗ trợ.

Một vấn đề này sinh trong cách khai báo kiểu con trả được truyền. Với ANSI C, giải pháp rất đơn giản: `void *` là loại con trả chung. Tuy nhiên, các hàm socket có trước ANSI C và giải pháp được chọn vào năm 1982 là xác định cấu trúc địa chỉ socket chung trong tiêu đề `<sys/socket.h>`, mà chúng tôi hiển thị trong [Hình 3.3](#).

[Hình 3.3](#) Cấu trúc địa chỉ socket chung: sockaddr.

```
cấu trúc sockaddr {
    uint8_t sa_len;
    in_family_t in_family; in_data[14]; /* Họ địa chỉ: giá trị AF_XXX */
    ký tự          *địa chỉ dành riêng cho giao thức */
};
```

Sau đó, các hàm socket được định nghĩa là lấy một con trỏ tới cấu trúc địa chỉ socket chung, như được hiển thị ở đây trong nguyên mẫu hàm ANSI C cho liên kết.

chức năng:

```
int bind(int, struct sockaddr *, socklen_t);
```

Điều này yêu cầu bất kỳ lệnh gọi nào đến các hàm này đều phải chuyển con trỏ tới cấu trúc địa chỉ ở cắm dành riêng cho giao thức để trở thành con trỏ tới cấu trúc địa chỉ ở cắm chung. Ví dụ,

```
struct sockaddr_in phục vụ;           /* Cấu trúc địa chỉ ở cắm IPv4 */

/* dién vào dịch vụ{} */

bind(sockfd, (struct sockaddr *) &serv, sizeof(serv));
```

Nếu chúng ta bỏ qua kiểu "`(struct sockaddr *)`", trình biên dịch C sẽ tạo ra cảnh báo có dạng "cảnh báo: chuyển arg 2 của 'liên kết' từ loại con trỏ không tương thích," giả sử các tiêu đề của hệ thống có nguyên mẫu ANSI C cho liên kết chức năng.

Từ quan điểm của một lập trình viên ứng dụng, việc sử dụng duy nhất các cấu trúc địa chỉ ở cắm chung này là truyền con trỏ tới các cấu trúc giao thức cụ thể.

Hãy nhớ lại [trong Phần 1.2](#) rằng trong tiêu đề `unp.h`, chúng ta định nghĩa `SA` là chuỗi "`struct sockaddr`" chỉ để rút ngắn đoạn mã mà chúng ta phải viết để truyền các con trỏ này.

Từ quan điểm của kernel, một lý do khác để sử dụng con trỏ tới cấu trúc địa chỉ ở cắm chung làm đối số là kernel phải lấy con trỏ của lệnh gọi, truyền nó tới `struct sockaddr *`, sau đó xem giá trị của `sa_family` để xác định loại cấu trúc. Như quan điểm của một lập trình viên ứng dụng, sẽ đơn giản hơn nếu kiểu con trỏ là `void *`, bỏ qua nhu cầu về kiểu truyền rõ ràng.

Cấu trúc địa chỉ ở cắm IPv6

Địa chỉ ở cắm IPv6 được xác định bằng cách bao gồm tiêu đề <netinet/in.h> và chúng tôi hiển thị nó trong [Hình 3.4](#).

[Hình 3.4](#) Cấu trúc địa chỉ socket IPv6: sockaddr_in6.

```
cấu trúc in6_addr {
    uint8_t s6_addr[16];           /* Địa chỉ IPv6 128 bit */
    /* byte mạng được sắp xếp */
};

#define SIN6_LEN             /* cần thiết cho các bài kiểm tra thời gian biên dịch */

cấu trúc sockaddr_in6 {
    uint8_t sin6_len;           /* độ dài của cấu trúc này (28) */
    sa_family_t sin6_family;   /* AF_INET6 */
    in_port_t sin6_port;       /* cổng lớp vận chuyển */
    /* byte mạng được sắp xếp */
    uint32_t sin6_flowinfo;   /* thông tin luồng, không xác định */
    cấu trúc in6_addr sin6_addr; /* Địa chỉ IPv6 */
    /* byte mạng được sắp xếp */
    uint32_t sin6_scope_id;   /* tập hợp các giao diện cho một phạm vi */
};
```

Các phần mở rộng của API ở cắm cho IPv6 được xác định trong RFC 3493 [Gilligan et al. 2003].

Lưu ý các điểm sau về [Hình 3.4](#):

- Hằng số **SIN6_LEN** phải được xác định nếu hệ thống hỗ trợ độ dài thành viên cho cấu trúc địa chỉ socket.
- Họ IPv6 là **AF_INET6**, trong khi họ IPv4 là **AF_INET**.
- Các thành viên trong cấu trúc này được sắp xếp sao cho nếu **sockaddr_in6** cấu trúc được căn chỉnh 64 bit, thành viên **sin6_addr** 128 bit cũng vậy. Trên một số bộ xử lý 64 bit, việc truy cập dữ liệu có giá trị 64 bit được tối ưu hóa nếu được lưu trữ trên ranh giới 64 bit.
- Thành viên **sin6_flowinfo** được chia thành hai trường:
 - 20 bit bậc thấp là nhãn luồng
 - 12 bit bậc cao được dự trữ

Trư ờng nhãn luồng được mô tả trong [Hình A.2](#). Việc sử dụng trư ờng nhãn luồng vẫn là một chủ đề nghiên cứu.

- `Sin6_scope_id` xác định vùng phạm vi trong đó địa chỉ trong phạm vi có ý nghĩa, phổ biến nhất là chỉ mục giao diện cho địa chỉ liên kết cục bộ ([Phản A.5](#)).

Cấu trúc địa chỉ ở cảm chung mới

Cấu trúc địa chỉ ở cảm chung mới đã được xác định là một phần của API ở cảm IPv6, để khắc phục một số thiếu sót của `struct sockaddr` hiện có. Không giống như `struct sockaddr`, `struct sockaddr_storage` mới đủ lớn để chứa bất kỳ loại địa chỉ socket nào được hệ thống hỗ trợ. Cấu trúc `sockaddr_storage` được xác định bằng cách bao gồm tiêu đề `<netinet/in.h>` mà chúng tôi hiển thị trong [Hình 3.5](#).

Hình 3.5 Cấu trúc địa chỉ socket lưu trữ: `sockaddr_storage`.

```
cấu trúc sockaddr_storage {
    phụ thuộc ss_len; /* độ dài của cấu trúc này (thực hiện
    uint8_t) */

    sa_family_t ss_family; /* Các phần /* Họ địa chỉ: giá trị AF_XXX */
    /* Tùy thuộc vào việc triển khai để cung cấp:
        * a) cần chỉnh đủ để đáp ứng các yêu cầu cần chỉnh của
        *      tất cả các loại địa chỉ socket mà hệ thống hỗ trợ.
        * b) đủ bộ nhớ để chứa bất kỳ loại địa chỉ socket nào mà
        *      hệ thống hỗ trợ.
    */
};

};


```

Kiểu `sockaddr_storage` cung cấp cấu trúc địa chỉ socket chung khác với `struct sockaddr` theo hai cách:

Một. Nếu bất kỳ cấu trúc địa chỉ socket nào mà hệ thống hỗ trợ có yêu cầu cần chỉnh thì `sockaddr_storage` sẽ cung cấp yêu cầu cần chỉnh nghiêm ngặt nhất.

b. `sockaddr_storage` đủ lớn để chứa bất kỳ địa chỉ socket nào
cấu trúc mà hệ thống hỗ trợ.

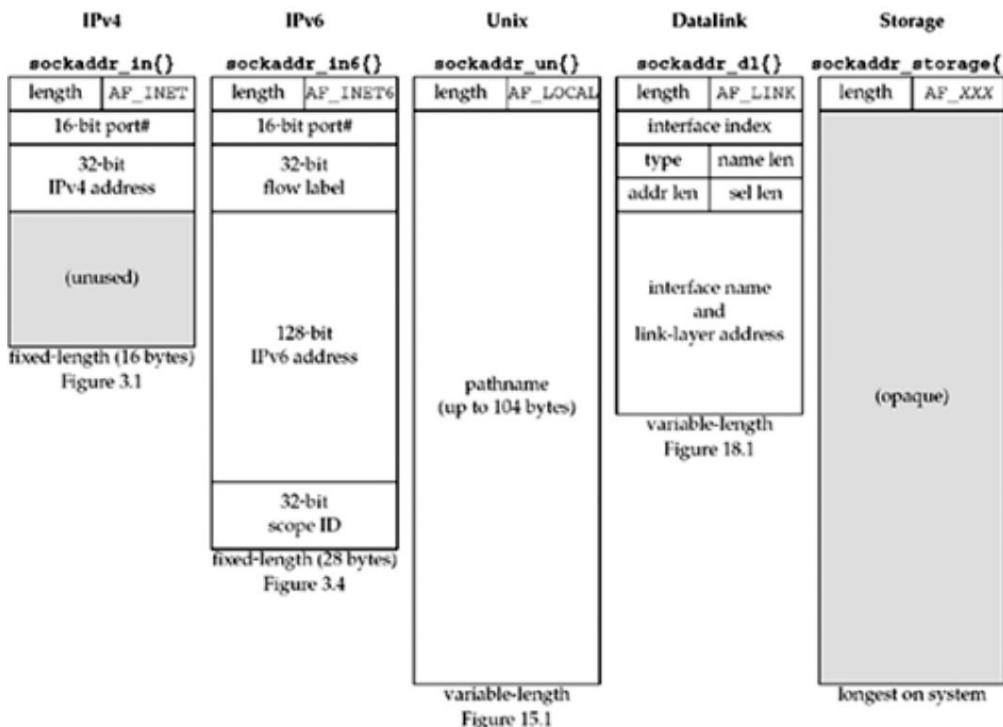
Lưu ý rằng các trường của cấu trúc `sockaddr_storage` không rõ ràng đối với người dùng, ngoại trừ `ss_family` và `ss_len` (nếu có). `sockaddr_storage` phải được truyền hoặc sao chép sang cấu trúc địa chỉ ở cảm thích hợp cho địa chỉ được cung cấp trong `ss_family` để truy cập vào bất kỳ trường nào khác.

So sánh cấu trúc địa chỉ socket

[Hình 3.6](#) cho thấy sự so sánh của năm cấu trúc địa chỉ socket mà chúng ta sẽ gặp trong văn bản này: IPv4, IPv6, miền Unix ([Hình 15.1](#)), datalink ([Hình 18.1](#)) và bộ lưu trữ. Trong [hình này](#), chúng ta giả sử rằng tất cả các cấu trúc địa chỉ socket đều chứa

trự ờng có độ dài một byte, trự ờng họ đó cũng chiếm một byte và bất kỳ trự ờng nào phải có ít nhất một số bit chính xác là số bit đó.

Hình 3.6. So sánh các cấu trúc địa chỉ socket khác nhau.



Hai trong số các cấu trúc địa chỉ socket có độ dài cố định, trong khi cấu trúc miền Unix và cấu trúc liên kết dữ liệu có độ dài thay đổi. Để xử lý các cấu trúc có độ dài thay đổi, bắt cứ khi nào chúng ta chuyển một con trỏ tới cấu trúc địa chỉ ở cắm làm đối số cho một trong các hàm ở cắm, chúng ta sẽ chuyển độ dài của nó làm đối số khác. Chúng tôi hiển thị kích thước tính bằng byte (để triển khai 4.4BSD) của các cấu trúc có độ dài cố định bên dưới mỗi cấu trúc.

Bản thân cấu trúc `sockaddr_un` không có độ dài thay đổi ([Hình 15.1](#)), nhưng lưu ờng thông tin tên đư ờng dẫn bên trong cấu trúc-có độ dài thay đổi. Khi truyền con trỏ tới các cấu trúc này, chúng ta phải cẩn thận với cách xử lý trự ờng độ dài, cả trự ờng độ dài trong chính cấu trúc địa chỉ socket (nếu đư ợc triển khai hỗ trợ) và độ dài đến và đi từ kernel.

Hình này thể hiện phong cách mà chúng ta áp dụng xuyên suốt văn bản: tên cấu trúc luôn đư ợc hiển thị bằng phông chữ đậm hơn, theo sau là dấu ngoặc nhọn, như trong `sockaddr_in{}`.

Chúng tôi đã lưu ý trự ờng đó rằng trự ờng độ dài đã đư ợc thêm vào tất cả các cấu trúc địa chỉ ở cắm với bản phát hành Reno 4.3BSD. Nếu trự ờng độ dài có mặt cùng với bản phát hành ban đầu của ở cắm, thì sẽ không cần đổi số độ dài cho tất cả các hàm của ở cắm: ví dụ: đổi số thứ ba để [liên kết và kết nối](#). Thay vào đó, kích thước của cấu trúc có thể đư ợc chứa trong trự ờng chiều dài của cấu trúc.

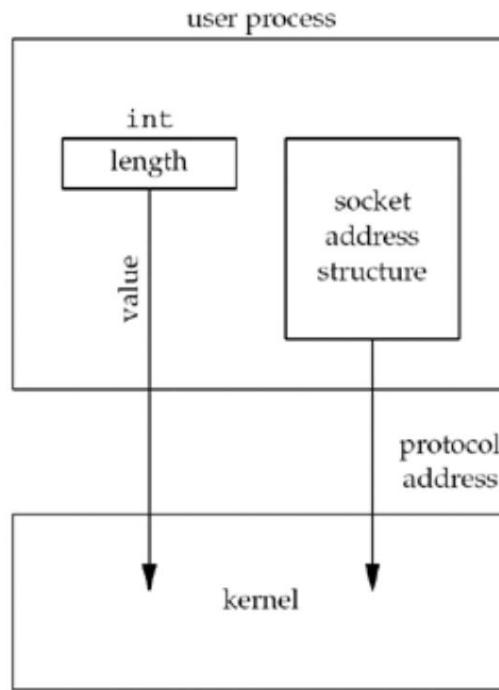
3.3 Đổi số giá trị-kết quả

Chúng tôi đã đề cập rằng khi cấu trúc địa chỉ socket được truyền tới bất kỳ hàm socket nào, nó luôn được truyền bằng tham chiếu. Nghĩa là, một con trỏ tới cấu trúc được truyền. Độ dài của cấu trúc cũng được truyền dưới dạng đổi số. Nhưng cách mà độ dài được truyền phụ thuộc vào hướng mà cấu trúc được truyền: từ tiến trình đến kernel hoặc ngược lại.

1. Ba hàm, **liên kết**, **kết nối** và **gửi tới**, chuyển cấu trúc địa chỉ socket từ tiến trình đến kernel. Một đổi số cho ba hàm này là con trỏ tới cấu trúc địa chỉ ở cắm và đổi số khác là kích thước nguyên của cấu trúc, như trong
- 2.
- 3.
- 4.
5. `struct sockaddr_in serv;`
- 6.
7. `/* điền vào serv{} */`
8. `kết nối (sockfd, (SA *) &serv, sizeof(serv));`
- 9.

Vì hạt nhân được truyền cả con trỏ và kích thước của những gì con trỏ trả tới, nên nó biết chính xác lượng dữ liệu cần sao chép từ tiến trình vào hạt nhân. [Hình 3.7](#) thể hiện kịch bản này.

Hình 3.7. Cấu trúc địa chỉ socket được truyền từ tiến trình tới kernel.



Chúng ta sẽ thấy trong chương tiếp theo rằng kiểu dữ liệu cho kích thước của cấu trúc địa chỉ socket thực sự là `socklen_t` chứ không phải `int`, nhưng đặc tả POSIX khuyến nghị rằng `socklen_t` nên được định nghĩa là `uint32_t`.

10. Bốn chức năng, `chấp nhận`, `recvfrom`, `getsockname` và `getpeername`, vuốt qua cấu trúc địa chỉ socket từ kernel đến tiến trình, hứa ngay lập tức với kích thước đó. Hai trong số các đối số của bốn hàm này là con trỏ tới cấu trúc địa chỉ socket cùng với một con trỏ tới một số nguyên chứa kích thước của cấu trúc, như trong

11.

12.

13.

```

14. struct sockaddr_un cli; /* Tên miền Unix */
15. socklen_t len;
16.
17. len = sizeof(cli);           /* len là một giá trị */
18. getpeername(unixfd, (SA *) &cli, &len);
19. /* len có thể đã thay đổi */
20.

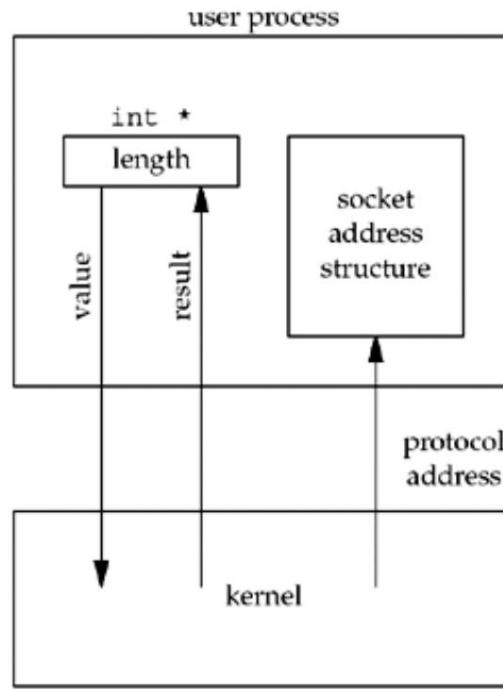
```

Lý do kích thước thay đổi từ số nguyên thành con trỏ thành số nguyên là vì kích thước vừa là giá trị khi hàm được gọi (nó cho kernel biết kích thước của cấu trúc để kernel không ghi quá cuối của cấu trúc khi điền nó vào) và kết quả khi hàm trả về (nó cho quá trình biết lưu lượng thông tin mà kernel thực sự lưu trữ trong cấu trúc).

Loại đối số này được gọi là đối số giá trị-kết quả. [Hình 3.8](#) cho thấy điều này

kịch bản.

Hình 3.8. Cấu trúc địa chỉ socket được truyền từ kernel tới tiến trình.



Chúng ta sẽ thấy một ví dụ về các đối số giá trị-kết quả trong [Hình 4.11.](#)

Chúng ta đã nói về cấu trúc địa chỉ socket được truyền giữa tiến trình và kernel. Đối với một triển khai như 4.4BSD, trong đó tất cả các hàm socket là các lệnh gọi hệ thống bên trong kernel, điều này đúng. Nhưng trong một số triển khai, đặc biệt là System V, các hàm socket chỉ là các hàm thư viện thực thi như một phần của quy trình người dùng thông thường. Cách các hàm này giao tiếp với ngăn xếp giao thức trong kernel là một chi tiết triển khai thư ứng không ảnh hưởng đến chúng tôi. Tuy nhiên, để đơn giản, chúng ta sẽ tiếp tục nói về các cấu trúc này khi được truyền giữa tiến trình và kernel bằng cách chức năng như [liên kết](#) và [kết nối](#). (Chúng ta sẽ thấy trong [Phần C.1](#) rằng việc triển khai Hệ thống V thực sự chuyển các cấu trúc địa chỉ socket giữa các tiến trình và kernel, nhưng là một phần của [STREAMS](#)

tin nhắn.)

Hai hàm khác chuyển đổi cấu trúc địa chỉ socket: [recvmsg](#) và [sendmsg](#) ([Phần 14.5](#)). Tuy nhiên, chúng ta sẽ thấy rằng [trường độ dài](#) không phải là đối số của hàm mà là thành viên cấu trúc.

Khi sử dụng các đối số kết quả giá trị cho độ dài của cấu trúc địa chỉ ở cắm, nếu cấu trúc địa chỉ ở cắm có độ dài cố định ([Hình 3.6](#)), giá trị được hạt nhân trả về sẽ luôn có [kích thước cố định](#) đó: 16 cho một [sockaddr_in](#) IPv4 và 28 cho một Ví dụ: IPv6 [sockaddr_in6](#). Nhưng với cấu trúc địa chỉ ở cắm có độ dài thay đổi (ví dụ: tên miền Unix [sockaddr_un](#)), giá trị được trả về có thể nhỏ hơn kích thước tối đa của cấu trúc (như chúng ta sẽ thấy trong [Hình 15.2](#)).

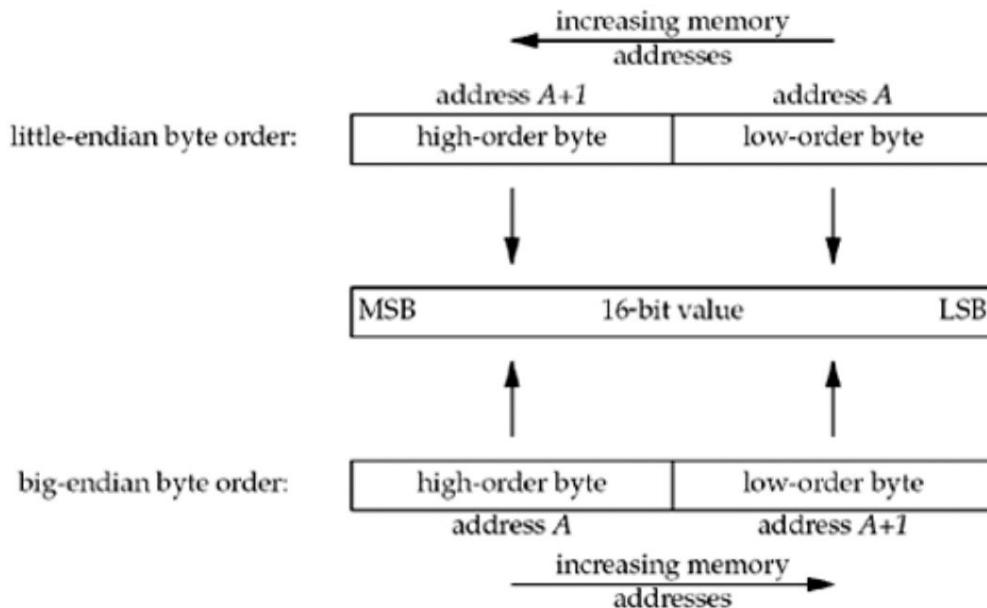
Với lập trình mạng, ví dụ phổ biến nhất của đối số kết quả giá trị là độ dài của cấu trúc địa chỉ ở cắm được trả về. Tuy nhiên, chúng ta sẽ gặp các đối số kết quả giá trị khác trong văn bản này:

- Ba đối số ở giữa của hàm `select` (Phần 6.3) _____
- Đối số độ dài cho hàm `getsockopt` (Phần 7.2) _____
- Các thành viên `msg_namelen` và `msg_controllen` của cấu trúc `msghdr`,
khi được sử dụng với `recvmsg` (Phần 14.5) _____
- Thành viên `ifc_len` của cấu trúc `ifconf` (Hình 17.2) _____
- Đối số đầu tiên trong hai đối số độ dài của hàm `sysctl` (Phần 18.4) _____

3.4 Chức năng sắp xếp byte

Hãy xem xét một số nguyên 16 bit được tạo thành từ 2 byte. Có hai cách để lưu trữ hai byte trong bộ nhớ: với byte thứ tự thấp ở địa chỉ bắt đầu, được gọi là thứ tự byte cuối nhỏ hoặc với byte thứ tự cao ở địa chỉ bắt đầu, được gọi là thứ tự byte cuối lớn. . Chúng tôi hiển thị hai định dạng này trong Hình 3.9.

Hình 3.9. Thứ tự byte cuối nhỏ và thứ tự byte cuối lớn cho số nguyên 16 bit.



Trong hình này, chúng tôi hiển thị các địa chỉ bộ nhớ tăng dần từ phải sang trái ở trên cùng và từ trái sang phải ở dưới cùng. Chúng tôi cũng hiển thị bit có trọng số cao nhất (MSB) là bit ngoài cùng bên trái của giá trị 16 bit và bit có trọng số thấp nhất (LSB) là bit ngoài cùng bên phải.

Các thuật ngữ "little-endian" và "big-endian" cho biết phần cuối của giá trị đa byte, đầu nhỏ hay đầu lớn, được lưu trữ tại địa chỉ bắt đầu của giá trị.

Thật không may, không có tiêu chuẩn nào giữa hai thứ tự byte này và chúng tôi gặp phải các hệ thống sử dụng cả hai định dạng. Chúng tôi gọi thứ tự byte được sử dụng bởi một hệ thống nhất định là thứ tự byte máy chủ. ChưƠng trình trong [Hình 3.10](#) in thứ tự byte máy chủ.

Hình 3.10 ChưƠng trình xác định thứ tự byte máy chủ.

giới thiệu/byteorder.c

```

1 #bao gồm          "unp.h"

2 số nguyên

3 chính(int argc, char **argv)
4 {
5     liên hiệp {
6         ngắn short;
7         char c[sizeof(ngắn)];
8     } Và;

9     un.s = 0x0102;
10    printf("%s: ", CPU_VENDOR_OS);
11    if (sizeof(ngắn) == 2) {
12        if (un.c[0] == 1 && un.c[1] == 2)
13            printf("big-endian\n");
14        khác nếu (un.c[0] == 2 && un.c[1] == 1)
15            printf("little-endian\n");
16        khác
17        printf("không rõ\n");
18    } khác
19    printf("sizeof(ngắn) = %d\n", sizeof(ngắn));
20
21 }

```

Chúng tôi lưu trữ giá trị hai byte `0x0102` trong số nguyên ngắn và sau đó xem xét hai byte liên tiếp, `c[0]` (`địa chỉ A` trong [Hình 3.9](#)) và `c[1]` (`địa chỉ A+1` trong [Hình 3.9](#)), để xác định thứ tự byte.

Chuỗi `CPU_VENDOR_OS` được xác định bởi chưƠng trình `autoconf` GNU khi phần mềm trong sách này được định cấu hình và nó xác định loại CPU, nhà cung cấp và bản phát hành hệ điều hành. Chúng tôi hiển thị một số ví dụ ở đây trong đầu ra của chưƠng trình này khi chạy trên các hệ thống khác nhau trong [Hình 1.16](#).

```
thứ tự byte freebsd4 %
i386-unknown-freebsd4.8: endian nhỏ
```

```
thứ tự macosx% byte
powerpc-apple-darwin6.6: endian lớn
```

```
thứ tự byte freebsd5 %
sparc64-unknown-freebsd5.1: big-endian
```

```
aix % thứ tự byte
powerpc-ibm-aix5.1.0.0: endian lớn
```

```
thứ tự hpx % byte
hppa1.1-hp-hpx11.11: endian lớn
```

```
thứ tự % byte của linux
i586-pc-linux-gnu: endian nhỏ
```

```
Solaris % thứ tự byte
sparc-sun-solaris2.9: endian lớn
```

Chúng ta đã nói về thứ tự byte của số nguyên 16 bit; rõ ràng, cuộc thảo luận tương tự cũng áp dụng cho số nguyên 32 bit.

Hiện tại có nhiều hệ thống có thể thay đổi giữa thứ tự byte nhỏ và lớn, đôi khi khi thiết lập lại hệ thống, đôi khi ở thời gian chạy.

Với tư cách là người lập trình mạng, chúng ta phải giải quyết những khác biệt về thứ tự byte này vì các giao thức mạng phải chỉ định thứ tự byte mạng. Ví dụ: trong phân đoạn TCP, có số cổng 16 bit và địa chỉ IPv4 32 bit. Ngăn xếp giao thức gửi và ngăn xếp giao nhận phải thống nhất về thứ tự byte của các trường nhiều byte này sẽ được truyền đi. Các giao thức Internet sử dụng thứ tự byte lớn cho các số nguyên nhiều byte này.

Về lý thuyết, việc triển khai có thể lưu trữ các trường trong cấu trúc địa chỉ ở cắm theo thứ tự byte máy chủ, sau đó chuyển đổi sang và từ thứ tự byte mạng khi di chuyển các trường đến và đi từ các tiêu đề giao thức, giúp chúng ta không phải lo lắng về chi tiết này. Tuy nhiên, cả lịch sử và đặc tả POSIX đều nói rằng một số trường nhất định trong cấu trúc địa chỉ ở cắm phải được duy trì theo thứ tự byte mạng. Do đó, mối quan tâm của chúng tôi là chuyển đổi giữa thứ tự byte máy chủ và thứ tự byte mạng. Chúng tôi sử dụng bốn hàm sau để chuyển đổi giữa hai thứ tự byte này.

```
#include <netinet/in.h>

uint16_t htons(uint16_t Host16bitvalue) ;

uint32_t htonl(uint32_t Host32bitvalue) ;

Cả hai đều trả về: giá trị theo thứ tự byte mạng

uint16_t ntohs(uint16_t net16bitvalue) ;

uint32_t ntohl(uint32_t net32bitvalue) ;

Cả hai đều trả về: giá trị theo thứ tự byte máy chủ
```

Trong tên của các chức năng này, **h** là viết tắt của máy chủ, **n** là viết tắt của mạng, **s** là viết tắt và **l** là viết tắt của dài. Các thuật ngữ "ngắn" và "dài" là các tạo tác lịch sử từ quá trình triển khai VAX kỹ thuật số của 4.2BSD. Thay vào đó, chúng ta nên coi **s** là giá trị 16 bit (chẳng hạn như số cổng TCP hoặc UDP) và **l** là giá trị 32 bit (chẳng hạn như địa chỉ IPv4). Thật vậy, trên Digital Alpha 64 bit, một số nguyên dài chiếm 64 bit, tuy nhiên các hàm **htonl** và **ntohl** hoạt động trên các giá trị 32 bit.

Khi sử dụng các hàm này, chúng ta không quan tâm đến các giá trị thực tế (big-endian hoặc little-endian) cho thứ tự byte máy chủ và thứ tự byte mạng. Nhưng gì chúng ta phải làm là gọi hàm thích hợp để chuyển đổi một giá trị nhất định giữa thứ tự byte máy chủ và mạng. Trên những hệ thống có cùng thứ tự byte với các giao thức Internet (big-endian), bốn hàm này thường được định nghĩa là macro rõ ràng.

Chúng ta sẽ nói nhiều hơn về vấn đề sắp xếp byte, liên quan đến dữ liệu chứa trong gói mạng, trái ngược với các trường trong tiêu đề giao thức, trong [Phần 5.18](#) và [Bài tập 5.8](#).

Chúng tôi vẫn chưa định nghĩa thuật ngữ "byte". Chúng tôi sử dụng thuật ngữ này để chỉ số lượng 8 bit vì hầu hết tất cả các hệ thống máy tính hiện tại đều sử dụng byte 8 bit. Hầu hết các tiêu chuẩn Internet sử dụng thuật ngữ octet thay vì byte để chỉ số lượng 8 bit. Điều này bắt đầu từ những ngày đầu của TCP/IP vì phần lớn công việc ban đầu được thực hiện trên các hệ thống như DEC-10, không sử dụng byte 8 bit.

Một quy ước quan trọng khác trong các tiêu chuẩn Internet là thứ tự bit. Trong nhiều tiêu chuẩn Internet, bạn sẽ thấy "hình ảnh" của các gói trông giống nhau sau (đây là 32 bit đầu tiên của tiêu đề IPv4 từ RFC 791):

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1
Version IHL Type of Service		Total Length	
-----+-----+-----+	-----+-----+-----+	-----+-----+-----+	-----+-----+-----+

Điều này thể hiện bốn byte theo thứ tự chúng xuất hiện trên dây; bit ngoài cùng bên trái là quan trọng nhất. Tuy nhiên, việc đánh số bắt đầu bằng số 0 được gán cho bit quan trọng nhất. Đây là ký hiệu mà bạn nên làm quen để dễ đọc các định nghĩa giao thức trong RFC hơn.

Một lỗi lập trình phổ biến trong những năm 1980 là phát triển mã trên các máy trạm của Sun (Motorola 68000 lớn) và quên gọi bất kỳ chức năng nào trong bốn chức năng này. Mã hoạt động tốt trên các máy trạm này nhưng sẽ không hoạt động khi được chuyển sang các máy endian nhỏ (chẳng hạn như VAX).

3.5 Hàm thao tác byte

Có hai nhóm hàm hoạt động trên các truờng nhiều byte mà không diễn giải dữ liệu và không giả định rằng dữ liệu là chuỗi C kết thúc bằng null. Chúng ta cần những loại hàm này khi xử lý cấu trúc địa chỉ socket vì chúng ta cần thao tác với các truờng như địa chỉ IP, có thể chứa byte bằng 0 nhưng không phải là chuỗi ký tự C. Các hàm bắt đầu bằng `str` (cho chuỗi), được xác định bằng cách bao gồm tiêu đề `<string.h>`, xử lý các chuỗi ký tự C kết thúc bằng null.

Nhóm hàm đầu tiên, có tên bắt đầu bằng `b` (đối với byte), là từ 4.2BSD và vẫn được cung cấp bởi hầu hết mọi hệ thống hỗ trợ các hàm socket. Nhóm hàm thứ hai, có tên bắt đầu bằng `mem` (dành cho bộ nhớ), lấy từ tiêu chuẩn ANSI C và được cung cấp cùng với bất kỳ hệ thống nào hỗ trợ thư viện ANSI C.

Đầu tiên chúng tôi trình bày các hàm dẫn xuất Berkeley, mặc dù hàm duy nhất chúng tôi sử dụng trong văn bản này là `bzero`. (Chúng tôi sử dụng nó vì nó chỉ có hai đối số và dễ nhớ hơn `memset` ba đối số, như được giải thích ở trang 8.) Bạn có thể gấp hai hàm khác, `bcopy` và `bcmpl`, trong các ứng dụng hiện có.

```
#include <strings.h>

void bzero(void *dest, size_t nbytes);

void bcopy(const void *src, void *dest, size_t nbytes);

int bcmpl(const void *ptr1, const void *ptr2, size_t nbytes);
```

Trả về: 0 nếu bằng, khác 0 nếu không bằng

Đây là lần đầu tiên chúng tôi gấp phải vòng loại `const` ANSI C. Trong ba cách sử dụng ở đây, nó chỉ ra rằng những gì được trả tới bởi con trả có đặc tính này, `src`, `ptr1`, và

`ptr2`, không bị hàm này sửa đổi. Nói cách khác, bộ nhớ đư ợc trả tới bởi con trỏ `const` đư ợc đọc như ng không đư ợc hàm sửa đổi.

`bzero` đặt số byte đư ợc chỉ định thành 0 ở đích. Chúng ta thư ờng sử dụng hàm này để khởi tạo cấu trúc địa chỉ socket về 0. `bcopy` di chuyển số byte đư ợc chỉ định từ nguồn đến đích. `bcmpl` so sánh hai chuỗi byte tùy ý. Giá trị trả về bằng 0 nếu hai chuỗi byte giống hệt nhau; nếu không thì đó là khác không.

Các chức năng sau đây là các chức năng ANSI C:

```
#include <string.h>

void *memset(void *dest, int c, size_t len);

void *memcpy(void *dest, const void *src, size_t nbytes);

int memcmp(const void *ptr1, const void *ptr2, size_t nbytes);

Trả về: 0 nếu bằng, <0 hoặc >0 nếu không bằng nhau (xem văn bản)
```

`memset` đặt số byte đư ợc chỉ định thành giá trị `c` ở đích. `memcpy` tương tự như `bcopy`, như ng thứ tự của hai đối số con trỏ bị hoán đổi. `bản sao` xử lý chính xác các trường chòng chéo, trong khi hành vi của `memcpy` không đư ợc xác định nếu nguồn và đích trùng nhau. Hàm `memmove` ANSI C phải đư ợc sử dụng khi các trường trùng nhau.

Một cách để nhớ thứ tự của hai con trỏ cho `memcpy` là hãy nhớ rằng chúng đư ợc viết theo thứ tự từ trái sang phải giống như câu lệnh gán trong C:

```
đích = src;
```

Một cách để nhớ thứ tự của hai đối số cuối cùng trong `memset` là nhận ra rằng tất cả các hàm `memXXX` của ANSI C đều yêu cầu một đối số có độ dài và nó luôn là đối số cuối cùng.

`memcmp` so sánh hai chuỗi byte tùy ý và trả về 0 nếu chúng giống nhau. Nếu không giống nhau, giá trị trả về sẽ lớn hơn 0 hoặc nhỏ hơn 0, tùy thuộc vào việc byte không bằng nhau đầu tiên đư ợc trả tới bởi `ptr1` lớn hơn hay nhỏ hơn byte tương ứng đư ợc chỉ ra bởi `ptr2`. Việc so sánh đư ợc thực hiện với giả định hai byte không bằng nhau là **ký tự không dấu**.

3.6 Hàm 'inet_aton', 'inet_addr' và 'inet_ntoa'

Chúng ta sẽ mô tả hai nhóm chức năng chuyển đổi địa chỉ trong phần này và phần tiếp theo. Họ chuyển đổi địa chỉ Internet giữa các chuỗi ASCII (những gì con người thích sử dụng) và các giá trị nhị phân theo thứ tự byte mạng (các giá trị được lưu trữ trong cấu trúc địa chỉ ở cắm).

1. `inet_aton`, `inet_ntoa` và `inet_addr` chuyển đổi địa chỉ IPv4 từ một địa chỉ chuỗi thập phân có dấu chấm (ví dụ: "206.168.112.96") thành giá trị nhị phân theo thứ tự byte mạng 32 bit của nó. Bạn có thể sẽ gặp những hàm này trong nhiều mã hiện có.
2. Các hàm mới hơn, `inet_pton` và `inet_ntop`, xử lý cả địa chỉ IPv4 và IPv6. Chúng tôi mô tả hai chức năng này trong phần tiếp theo và sử dụng chúng xuyên suốt văn bản.

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);

Trả về: 1 nếu chuỗi hợp lệ, 0 nếu có lỗi

in_addr_t inet_addr(const char *strptr);

Trả về: địa chỉ IPv4 theo thứ tự byte mạng nhị phân 32 bit; INADDR_NONE nếu có lỗi

char *inet_ntoa(struct in_addr inaddr);

Trả về: con trả tới chuỗi thập phân có dấu chấm
```

Cái đầu tiên trong số này, `inet_aton`, chuyển đổi chuỗi ký tự C được trả tới bởi strptr thành giá trị thứ tự byte mạng nhị phân 32 bit của nó, được lưu trữ thông qua con trả addrptr. Nếu thành công, trả về 1; nếu không thì trả về 0.

Một tính năng không có giấy tờ của `inet_aton` là nếu addrptr là con trả null thì hàm vẫn thực hiện xác thực chuỗi đầu vào như ng không lưu trữ bất kỳ kết quả nào.

`inet_addr` thực hiện chuyển đổi tương tự, trả về giá trị thứ tự byte mạng nhị phân 32-bit làm giá trị trả về. Vẫn dè với chức năng này là tất cả 232 các giá trị nhị phân có thể là địa chỉ IP hợp lệ (0.0.0.0 đến 255.255.255.255), như ng hàm trả về hằng số `INADDR_NONE` (thường là 32 bit một) khi có lỗi. Điều này có nghĩa là hàm này không thể xử lý chuỗi thập phân có dấu chấm 255.255.255.255 (địa chỉ quang bá giới hạn IPv4, Mục 20.2) vì hàm này không thể xử lý được vì giá trị nhị phân của nó thường như biểu thị lỗi của hàm.

Một vấn đề tiềm ẩn với `inet_addr` là một số trang man cho biết nó trả về -1 khi có lỗi, thay vì `INADDR_NONE`. Điều này có thể dẫn đến các vấn đề, tùy thuộc vào C

trình biên dịch, khi so sánh giá trị trả về của hàm (giá trị không dấu) với hằng số âm.

Ngày nay, `inet_addr` không còn được dùng nữa và thay vào đó, bất kỳ mã mới nào cũng nên sử dụng `inet_aton`.

Tốt hơn hết là sử dụng các chức năng mới hơn được mô tả trong phần tiếp theo để xử lý cả IPv4 và IPv6.

Hàm `inet_ntoa` chuyển đổi địa chỉ IPv4 theo thứ tự byte mạng nhị phân 32 bit thành chuỗi thập phân chấm tự ứng. Chuỗi được trả đến bởi giá trị trả về của hàm nằm trong bộ nhớ tĩnh. Điều này có nghĩa là hàm này không được vào lại, điều này chúng ta sẽ thảo luận trong [Phần 11.18](#). Cuối cùng, hãy lưu ý rằng hàm này lấy một cấu trúc làm đối số của nó chứ không phải một con trỏ tới một cấu trúc.

Các hàm lấy cấu trúc thực tế làm đối số rất hiếm. Việc truyền một con trỏ tới cấu trúc là phổ biến hơn.

3.7 Hàm 'inet_pton' và 'inet_ntop'

Hai chức năng này mới với IPv6 và hoạt động với cả địa chỉ IPv4 và IPv6.

Chúng tôi sử dụng hai chức năng này trong suốt văn bản. Các chữ cái "p" và "n" tương ứng cho cách trình bày và số. Định dạng trình bày cho một địa chỉ thường là chuỗi ASCII và định dạng số là giá trị nhị phân đi vào địa chỉ ở cắm kết cấu.

```
#include <arpa/inet.h>

int inet_pton(int family, const char *strptr, void *addrptr);

Trả về: 1 nếu OK, 0 nếu đầu vào không phải là định dạng trình bày hợp lệ, -1 nếu có lỗi

const char *inet_ntop(int family, const void *addrptr, char *strptr, size_t len);

Trả về: con trỏ tới kết quả nếu OK, NULL bị lỗi
```

Đối số họ cho cả hai hàm là `AF_INET` hoặc `AF_INET6`. Nếu họ không được hỗ trợ, cả hai hàm đều trả về lỗi với `errno` được đặt thành `EAFNOSUPPORT`.

Hàm đầu tiên có gắng chuyển đổi chuỗi được trả tới bởi strptr, lưu trữ kết quả nhị phân thông qua con trỏ addrptr. Nếu thành công, giá trị trả về là 1. Nếu chuỗi đầu vào không phải là định dạng trình bày hợp lệ cho họ đã chỉ định, thì trả về 0.

`inet_ntop` thực hiện chuyển đổi ngược lại, từ số (addrptr) sang trình bày (strptr). Đối số len là kích thước của đích, để ngăn hàm tràn bộ đệm của người gọi. Để giúp chỉ định kích thước này, hai định nghĩa sau được xác định bằng cách bao gồm tiêu đề `<netinet/in.h>`:

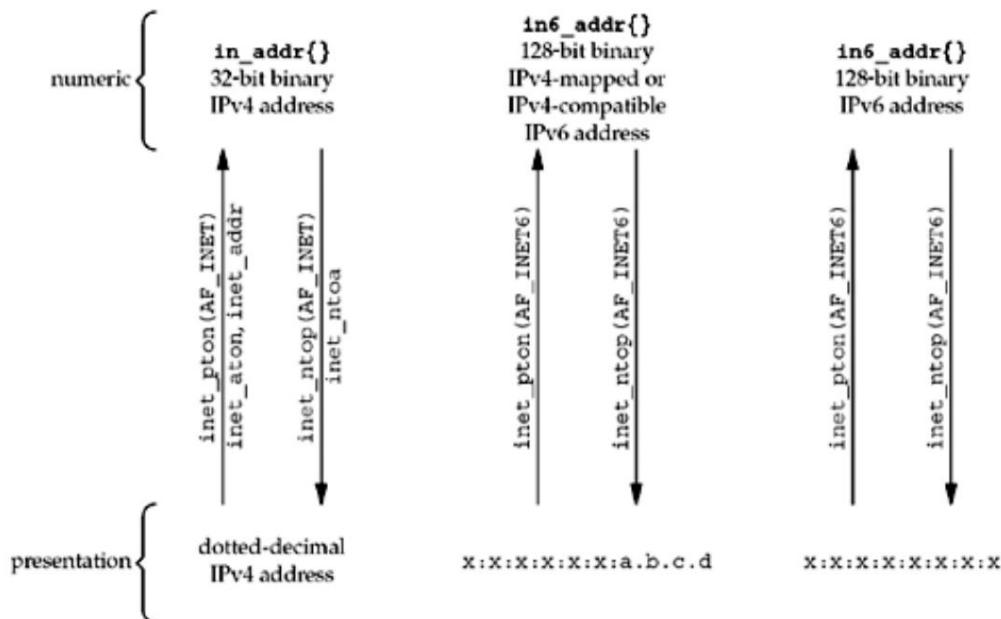
```
#xác định INET_ADDRSTRLEN #xác định INET6_ADDRSTRLEN 16          /* cho IPv4 thập phân châm */ 46          /* cho chuỗi hex IPv6 */
```

Nếu len quá nhỏ để giữ định dạng trình bày kết quả, bao gồm cả null kết thúc, một con trỏ null sẽ được trả về và `errno` được đặt thành `ENOSPC`.

Đối số strptr của `inet_ntop` không thể là con trỏ rỗng. Người gọi phải phân bổ bộ nhớ cho đích và chỉ định kích thước của nó. Nếu thành công, con trỏ này là giá trị trả về của hàm.

Hình 3.11 tóm tắt năm chức năng mà chúng tôi đã mô tả trong phần này và phần trước.

Hình 3.11. Tổng hợp các hàm chuyển đổi địa chỉ.



Ví dụ

Ngay cả khi hệ thống của bạn chưa hỗ trợ IPv6, bạn có thể bắt đầu sử dụng các chức năng mới hơn này bằng cách thay thế các lệnh gọi có dạng

```
foo.sin_addr.s_addr = inet_addr(cp);
```

với

```
inet_pton(AF_INET, cp, &foo.sin_addr);
```

và thay thế các cuộc gọi có dạng

```
ptr = inet_ntoa(foo.sin_addr);
```

với

```
char str[INET_ADDRSTRLEN];
ptr = inet_ntop(AF_INET, &foo.sin_addr, str, sizeof(str));
```

[Hình 3.12](#) cho thấy một định nghĩa đơn giản về `inet_pton` chỉ hỗ trợ IPv4. Tương tự, [Hình 3.13](#) cho thấy một phiên bản đơn giản của `inet_ntop` chỉ hỗ trợ IPv4.

Hình 3.12 Phiên bản đơn giản của `inet_pton` chỉ hỗ trợ IPv4.

libfree/inet_pton_ipv4.c

```
10 int
11 inet_pton(int họ, const char *strptr, void *addrptr)
12 {
13     if (gia đình == AF_INET) {
14         cấu trúc in_addr in_val;
15
16         if (inet_aton(strptr, &in_val)) {
17             memcpy(addrptr, &in_val, sizeof(struct in_addr));
18             trả lại (1);
19         }
20         trả về (0);
21     }
22 }
```

```

20      }
21      errno = EAFNOSUPPORT;
22      trả lại (-1);
23 }

```

Hình 3.13 Phiên bản đơn giản của `inet_ntop` chỉ hỗ trợ IPv4.

`libfree/inet_ntop_ipv4.c`

```

8 ký tự hằng *

9 inet_ntop(int họ, const void *addrptr, char *strptr, size_t len)
10 {
11     const u_char *p = (const u_char *) addrptr;

12     if (gia đình == AF_INET) {
13         nhiệt độ ký tự [INET_ADDRSTRLEN];

14         sprintf(temp, sizeof(temp), "%d.%d.%d.%d", p[0], p[1], p[2],
p[3]);
15         if (strlen(temp) >= len) {
16             errno = ENOSPC;
17             trả lại (NULL);
18         }
19         strcpy(strptr, temp);
20         trả lại (strptr);
21     }
22     errno = EAFNOSUPPORT;
23     trả lại (NULL);
24 }

```

3.8 'sock_ntop' và các hàm liên quan

Một vấn đề cơ bản với `inet_ntop` là nó yêu cầu người gọi chuyển con trỏ tới địa chỉ phân. Địa chỉ này thường được chứa trong cấu trúc địa chỉ socket, yêu cầu người gọi phải biết định dạng của cấu trúc và họ địa chỉ. Tức là để sử dụng nó chúng ta phải viết code dạng

```

struct sockaddr_in addr;

inet_ntop(AF_INET, &addr.sin_addr, str, sizeof(str));

```

cho IPv4, hoặc

```
cấu trúc sockaddr_in6 addr6;
inet_ntop(AF_INET6, &addr6.sin6_addr, str, sizeof(str));
```

cho IPv6. Điều này làm cho mã của chúng tôi phụ thuộc vào giao thức.

Để giải quyết vấn đề này, chúng ta sẽ viết hàm riêng của mình có tên `sock_ntop` để đưa con trả tới cấu trúc địa chỉ socket, xem bên trong cấu trúc và gọi hàm thích hợp để trả về định dạng trình bày của địa chỉ.

```
#include "unp.h"

char *sock_ntop(const struct sockaddr *sockaddr, socklen_t addrlen);
Trả về: con trả không null nếu OK, NULL bị lỗi
```

Đây là ký hiệu chúng tôi sử dụng cho các hàm của chính chúng tôi (các hàm hệ thống không chuẩn) mà chúng tôi sử dụng trong suốt cuốn sách: hộp xung quanh nguyên mẫu hàm và giá trị trả về được gạch ngang. Tiêu đề được đưa vào đầu thư ờng là tiêu đề `unp.h` của chúng tôi.

`sockaddr` trả tới cấu trúc địa chỉ socket có độ dài là `addrlen`. Hàm sử dụng bộ đệm tĩnh của chính nó để giữ kết quả và một con trả tới bộ đệm này là giá trị trả về.

Lưu ý rằng việc sử dụng bộ nhớ tĩnh cho kết quả sẽ ngăn không cho hàm được nhập lại hoặc an toàn theo luồng. Chúng ta sẽ nói nhiều hơn về điều này trong [Phần 11.18](#). Chúng tôi đã làm điều này quyết định thiết kế chức năng này để cho phép chúng ta dễ dàng gọi nó từ những ví dụ đơn giản trong cuốn sách.

Định dạng trình bày là dạng thập phân chấm của địa chỉ IPv4 hoặc dạng chuỗi hex của địa chỉ IPv6 được bao quanh bởi dấu ngoặc, theo sau là dấu kết thúc (chúng tôi sử dụng dấu hai chấm, tương tự như cú pháp URL), theo sau là số công thập phân, theo sau là bởi một ký tự null. Do đó, kích thước bộ đệm ít nhất phải là `INET_ADDRSTRLEN` cộng với 6 byte cho IPv4 ($16 + 6 = 22$) hoặc `INET6_ADDRSTRLEN` cộng với 8 byte cho IPv6 ($46 + 8 = 54$).

Chúng tôi chỉ hiển thị mã nguồn cho `tructure AF_INET` trong [Hình 3.14](#).

Hình 3.14 Hàm `sock_ntop` của chúng ta.

`lib/sock_ntop.c`

```

5 ký tự *
6 sock_ntop(const struct sockaddr *sa, socklen_t salen)
7 {
8     char portstr[8];
9     char tĩnh str[128];           /* Tên miền Unix lớn nhất */

10    chuyển đổi (sang->to_family) {
11        trường hợp AF_INET:{
12            struct sockaddr_in *sin = (struct sockaddr_in *) sa;
13
14            if (inet_ntop(AF_INET, &sin->sin_addr, str, sizeof(str)) ==
VÔ GIÁ TRỊ
15                trả lại (NULL);
16            if (ntohs(sin->sin_port) != 0) {
17                snprintf(portstr, sizeof(portstr), "%d",
18                        ntohs(sin->sin_port));
19            }
20            strcat(str, portstr);
21        }
}

```

Có một số chức năng khác mà chúng tôi xác định để hoạt động trên cấu trúc địa chỉ ở cắm và những chức năng này sẽ đơn giản hóa khả năng di chuyển mã của chúng tôi giữa IPv4 và IPv6.

#include "unp.h"	
int sock_bind_wild(int sockfd, int family);	Trả về: 0 nếu OK, -1 nếu có lỗi
int sock_cmp_addr(const struct sockaddr *sockaddr1,	
const struct sockaddr *sockaddr2, socklen_t addrlen);	
Trả về: 0 nếu địa chỉ cùng họ và các cổng bằng nhau, nếu không thì khác 0	
int sock_cmp_port(const struct sockaddr *sockaddr1,	
const struct sockaddr *sockaddr2, socklen_t addrlen);	
Trả về: 0 nếu địa chỉ cùng họ và các cổng bằng nhau, nếu không thì khác 0	
int sock_get_port(const struct sockaddr *sockaddr, socklen_t addrlen);	

```
#include "unp.h"

Trả về: số cổng không âm cho địa chỉ IPv4 hoặc IPv6, nếu không -1

char *sock_ntop_host(const struct sockaddr *sockaddr, socklen_t addrlen);

Trả về: con trỏ không null nếu OK, NULL bị lỗi

void sock_set_addr(const struct sockaddr *sockaddr, socklen_t addrlen, void *ptr);

void sock_set_port(const struct sockaddr *sockaddr, socklen_t addrlen, int
Hải cảng);

void sock_set_wild(struct sockaddr *sockaddr, socklen_t addrlen);
```

sock_bind_wild liên kết địa chỉ tự đại diện và cổng tạm thời với ổ cắm. **sock_cmp_addr** so sánh phần địa chỉ của hai cấu trúc địa chỉ socket và **sock_cmp_port** so sánh số cổng của hai cấu trúc địa chỉ socket. **sock_get_port** chỉ trả về số cổng và **sock_ntop_host** chỉ chuyển đổi phần máy chủ của cấu trúc địa chỉ socket thành định dạng bản trình bày (không phải số cổng). **sock_set_addr** chỉ đặt phần địa chỉ của cấu trúc địa chỉ ổ cắm thành giá trị được trả bởi ptr và **sock_set_port** chỉ đặt số cổng của cấu trúc địa chỉ ổ cắm. **sock_set_wild** đặt phần địa chỉ của cấu trúc địa chỉ ổ cắm thành ký tự đại diện. Giống như tất cả các hàm trong văn bản, chúng tôi cung cấp hàm bao bọc có tên bắt đầu bằng "**S**" cho tất cả các hàm này trả về các giá trị khác với **void** và thường gọi hàm bao bọc từ các chương trình của chúng tôi. Chúng tôi không hiển thị mã nguồn của tất cả các chức năng này nhưng nó có sẵn miễn phí (xem Lời nói đầu).

3.9 Chức năng 'đọc', 'viết' và 'đọc dòng'

Ổ cắm luồng (ví dụ: ổ cắm TCP) thể hiện hành vi **đọc** và **ghi** các chức năng khác với I/O tệp thông thường. Việc **đọc** hoặc **ghi** trên ổ cắm luồng có thể nhập hoặc xuất ít byte hơn yêu cầu nhưng đây không phải là tình trạng lỗi. Lý do là có thể đạt tới giới hạn bộ đệm cho ổ cắm trong kernel. Tất cả những gì cần thiết để nhập hoặc xuất các byte còn lại là để người gọi lại chức năng **đọc** hoặc **ghi**. Một số phiên bản Unix cũng thể hiện hành vi này khi ghi hơn 4.096 byte vào một đường ống. Tình huống này luôn có thể xảy ra trên ổ cắm luồng có chức năng **đọc** như ng thư ờng chỉ thấy với **chức năng ghi** nếu ổ cắm không bị chặn.

Tuy nhiên, chúng tôi luôn gọi hàm **ghi** thay vì **ghi**, trong trường hợp việc triển khai trả về số lượng ngắn.

Chúng tôi cung cấp ba chức năng sau đây mà chúng tôi sử dụng bất cứ khi nào chúng tôi đọc hoặc ghi vào ô cảm luồng:

```
#include "unp.h"

ssize_t readn(int filedes, void *buff, size_t nbytes);

ssize_t write(int filedes, const void *buff, size_t nbytes);

ssize_t readline(int filedes, void *buff, size_t maxlen);

Trả về tất cả: số byte được đọc hoặc ghi, lỗi -1
```

[Hình 3.15](#) thể hiện chức năng đọc , [Hình 3.16](#) thể hiện chức năng ghi và [Hình 3.17](#) thể hiện chức năng đọc dòng .

Hình 3.15 Hàm đọc: Đọc n byte từ bộ mô tả.

lib/readn.c

```
1 #bao gồm          "unp.h"

2 ssize_t 3           /* Đọc "n" byte từ bộ mô tả. */
readn(int fd, void *vptr, size_t n)
4 {
5     size_t nleft;
6     ssize_t nread;
7     char *ptr;

8     ptr = vptr;
9     nleft = n;
10    trong khi (nleft > 0) {
11        if ( (nread = read(fd, ptr, nleft)) < 0) {
12            nếu (errno == EINTR)
13                nđọc = 0;           /* và gọi lại read() */
14            khác
15                trả lại (-1);
16            } khác nếu (nread == 0)
17            phá vỡ;           /*EOF */

18        nleft -= nread;
19        ptr += nread;
20    }
21    trả lại (n - nleft);      /* trả về >= 0 */
22 }
```

Hình 3.16 Hàm ghi: Ghi n byte vào bộ mô tả.

lib/writen.c

```

1 #include "unp.h"

2 ssize_t 3                                /* Ghi "n" byte vào bộ mô tả. */
3   đã viết(int fd, const void *vptr, size_t n)
4 {
5   size_t nleft;
6   ssize_t đư ợc viết;
7   const char *ptr;

8   ptr = vptr;
9   nleft = n;
10  trong khi (nleft > 0) {
11    if ( (nwriting = write(fd, ptr, nleft)) <= 0) {
12      if (đư ợc viết < 0 && errno == EINTR)
13        nviết = 0;           /* và gọi lại write() */
14      khác
15      trả lại (-1);       /* lỗi */
16    }

17    nleft -= đư ợc viết;
18    ptr += đư ợc viết;
19  }
20  trả lại (n);
21 }
```

Hình 3.17 Hàm readline: Đọc một dòng văn bản từ bộ mô tả, mỗi lần một byte.

kiểm tra/readline1.c

```

1 #bao gồm          "unp.h"

2 /* PHIÊN BẢN CHẬM ĐẦU - chỉ là ví dụ */
3 cõ_t
4 dòng đọc(int fd, void *vptr, size_t maxlen)
5 {
6   ssize_t n, rc;
7   ký tự      c, *ptr;

8   ptr = vptr;
9   cho (n = 1; n < maxlen; n++) {
```

```

10      lại:
11      if ( (rc = read(fd, &c, 1)) == 1) {
12          *ptr++ = c;
13          nếu (c == '\n')
14              phá vỡ;           /* dòng mới được lưu trữ, giống như fgets() */
15          } khác nếu (rc == 0) {
16              *ptr = 0;
17              trả lại (n - 1);    /* EOF, n - 1 byte đã được đọc */
18          } khác {
19              nếu (errno == EINTR)
20                  đi lại lần nữa;
21              trả lại (-1);     /* lỗi, lỗi không được thiết lập bởi read() */
22          }
23      }

24      *ptr = 0;           /* null kết thúc như fgets() */
25      trả lại (n);
26 }

```

Ba chức năng của chúng tôi tìm kiếm lỗi **EINTR** (cuộc gọi hệ thống bị gián đoạn do tín hiệu bắt được, chúng tôi sẽ thảo luận chi tiết hơn trong [Phần 5.9](#)) và tiếp tục đọc hoặc ghi nếu xảy ra lỗi. Chúng tôi xử lý lỗi ở đây, thay vì buộc người gọi gọi lại **readn** hoặc **write**, vì mục đích của ba hàm này là ngăn người gọi phải xử lý số lượng ngắn.

Trong [Phần 14.3](#), chúng tôi sẽ đề cập rằng cờ **MSG_WAITALL** có thể được sử dụng với **recv** để thay thế nhu cầu về một chức năng **đọc** riêng biệt.

Lưu ý rằng hàm **readline** của chúng tôi gọi hàm **đọc** của hệ thống một lần cho mỗi byte dữ liệu. Điều này rất kém hiệu quả và tại sao chúng tôi lại nhận xét mã này là "CHẬM ĐAU ĐAU". Khi gặp phải mong muốn đọc các dòng từ ổ cứng, việc chuyển sang thư viện I/O tiêu chuẩn (được gọi là "stdio") là khá hấp dẫn. Chúng ta sẽ thảo luận chi tiết về cách tiếp cận này trong [Phần 14.8](#), nhưng nó có thể là một con đường nguy hiểm. Bộ đệm stdio tương tự giải quyết vấn đề hiệu suất này tạo ra nhiều vấn đề hậu cần có thể dẫn đến các lỗi ẩn sâu trong ứng dụng của bạn. Lý do là trạng thái của bộ đệm stdio không bị lộ. Để giải thích thêm điều này, hãy xem xét giao thức dựa trên dòng giữa máy khách và máy chủ, trong đó một số máy khách và máy chủ sử dụng giao thức đó có thể được triển khai theo thời gian (thực sự khá phổ biến; ví dụ: có nhiều trình duyệt Web và máy chủ Web được viết đặc lập cho đặc tả HTTP). Kỹ thuật "lập trình phòng thủ" tốt yêu cầu các chương trình này không chỉ mong đợi các chương trình tương tự của chúng tuân theo giao thức mạng mà còn kiểm tra cả lưu lượng truy cập mạng không mong muốn. Những vi phạm giao thức như vậy phải được báo cáo dưới dạng lỗi để các lỗi được phát hiện và sửa chữa (đồng thời phát hiện các nỗ lực độc hại), đồng thời để các ứng dụng mạng có thể khôi phục sau sự cố lưu lượng truy cập và tiếp tục hoạt động nếu có thể. Sử dụng stdio để đệm dữ liệu nhằm nâng cao hiệu suất trong

trừ ớc những mục tiêu này vì ứng dụng không có cách nào để biết liệu dữ liệu không mong muốn có được lưu giữ trong bộ đệm stdio tại bất kỳ thời điểm nào hay không.

Có nhiều giao thức mạng dựa trên đường truyền như SMTP, HTTP, giao thức kết nối điều khiển FTP và ngón tay. Vì vậy, mong muốn được làm việc trên dây chuyền cứ liên tục xuất hiện. Nhưng lời khuyên của chúng tôi là hãy suy nghĩ về vùng đệm chữ không phải về dòng. Viết mã của bạn để đọc vùng đệm dữ liệu và nếu dự kiến có một dòng, hãy kiểm tra bộ đệm để xem liệu nó có chứa dòng đó hay không.

Hình 3.18 cho thấy một phiên bản nhanh hơn của hàm `readline`, sử dụng bộ đệm riêng của nó thay vì bộ đệm stdio. Quan trọng nhất, trạng thái bộ đệm bên trong `của đường đọc` được hiển thị, do đó người gọi có thể biết chính xác những gì đã nhận được. Ngay cả với tính năng này, `dòng đọc` vẫn có thể có vấn đề, như chúng ta sẽ thấy trong Phần 6.3. Các chức năng hệ thống như `select` vẫn không biết về bộ đệm bên trong `của readline`, vì vậy một chương trình được viết bắt cần có thẻ dễ dàng thấy chính nó đang chờ trong `select` đối với dữ liệu đã nhận được và lưu trữ trong bộ đệm `của readline`. Đối với vấn đề đó, việc kết hợp các lệnh gọi `readn` và `readline` sẽ không hoạt động như mong đợi trừ khi `readn` được sửa đổi để kiểm tra bộ đệm nội bộ là tốt.

Hình 3.18 Phiên bản tốt hơn của chức năng đọc dòng.

lib/readline.c

```

1 #include "unp.h"

2 int tĩnh read_cnt;
3 char tĩnh *read_ptr;
4 char tĩnh read_buf[MAXLINE];

5 kích thư ớc tĩnh_t
6 my_read(int fd, char *ptr)
7 {

    số 8     nếu (read_cnt <= 0) {
8         lại:
9             if ( (read_cnt = read(fd, read_buf, sizeof(read_buf))) < 0) {
10                 nếu (errno == EINTR)
11                     đi lại lần nữa;
12                 trả lại (-1);
13             }
14             } khác nếu (read_cnt == 0)
15             trả về (0);
16             read_ptr = read_buf;
17     }

```

```

18     read_cnt--;
19     *ptr = *read_ptr++;
20     trả lại (1);
21 }
22 cỡ_t
23 dòng đọc(int fd, void *vptr, size_t maxlen)
24 {
25     ssize_t n, rc;
26     ký tự c, *ptr;

27     ptr = vptr;
28     cho (n = 1; n < maxlen; n++) {
29         if ((rc = my_read(fd, &c)) == 1) {
30             *ptr++ = c;
31             nếu (c == '\n')
32                 phá vỡ; /* dòng mới được lưu trữ, giống như fgets() */
33             } khác nếu (rc == 0) {
34                 *ptr = 0;
35                 trả lại (n - 1); /* EOF, n - 1 byte đã được đọc */
36             } khác
37             trả lại (-1); /* lỗi, lỗi không được thiết lập bởi read() */
38         }

39         *ptr = 0; /* null kết thúc như fgets() */
40         trả lại (n);
41 }

42 cỡ_t
43 readlinebuf(void **vptrptr)
44 {
45     nếu (read_cnt)
46         *vptrptr = read_ptr;
47     trả lại (read_cnt);
48 }

```

2-21 Hàm bên trong `my_read` đọc tối đa `MAXLINE` ký tự cùng một lúc và sau đó trả về chúng, từng ký tự một.

29 Thay đổi duy nhất đối với chính hàm `readline` là gọi `my_read` thay vì `read`.

42-48 Một chức năng mới, `readlinebuf`, hiển thị trạng thái bộ đệm bên trong để người gọi có thể kiểm tra xem liệu có nhận được nhiều dữ liệu hơn ngoài một dòng hay không.

Thật không may, bằng cách sử dụng các biến `tĩnh` trong `readline.c` để duy trì thông tin trạng thái qua các lệnh gọi liên tiếp, các hàm này không được nhập lại hoặc an toàn theo luồng.

Chúng ta sẽ thảo luận vấn đề này trong [Phần 11.18 và 26.5](#). Chúng tôi sẽ phát triển một phiên bản an toàn cho luồng bằng cách sử dụng dữ liệu dành riêng cho luồng trong [Hình 26.11](#).

3.10 Tóm tắt

Cấu trúc địa chỉ socket là một phần không thể thiếu của mọi chương trình mạng. Chúng tôi phân bổ chúng, di chuyển chúng và chuyển con trỏ tới chúng tới các hàm socket khác nhau. Đôi khi chúng ta chuyển con trỏ tới một trong các cấu trúc này tới hàm socket và nó sẽ di chuyển vào nội dung. Chúng tôi luôn chuyển các cấu trúc này bằng tham chiếu (nghĩa là chúng tôi chuyển một con trỏ tới cấu trúc chứ không phải chính cấu trúc đó) và chúng tôi luôn chuyển kích thước của cấu trúc dưới dạng một đối số khác. Khi một hàm socket di chuyển vào một cấu trúc, độ dài cũng được truyền bằng tham chiếu để hàm có thể cập nhật giá trị của nó. Chúng tôi gọi đây là những đối số kết quả giá trị.

Cấu trúc địa chỉ ở cắm có khả năng tự xác định vì chúng luôn bắt đầu bằng một truờng ("nhóm") xác định họ địa chỉ có trong cấu trúc. Các triển khai mới hơn hỗ trợ cấu trúc địa chỉ ở cắm có độ dài thay đổi cũng chứa truờng độ dài ở đầu, truờng này chứa độ dài của toàn bộ cấu trúc.

Hai hàm chuyển đổi địa chỉ IP giữa định dạng trình bày (những gì chúng ta viết, chẳng hạn như ký tự ASCII) và định dạng số (những gì đi vào cấu trúc địa chỉ ở cắm) là `inet_ntop` và `inet_pton`. Mặc dù chúng ta sẽ sử dụng hai chức năng này trong các chương tiếp theo nhưng chúng phụ thuộc vào giao thức. Một kỹ thuật tốt hơn là thao tác các cấu trúc địa chỉ socket như các đối tượng mờ đục, chỉ biết con trỏ tới cấu trúc và kích thước của nó. Chúng tôi đã sử dụng phương pháp này để phát triển một tập hợp các hàm `sock_` giúp làm cho chương trình của chúng tôi không phụ thuộc vào giao thức. Chúng tôi sẽ hoàn thành việc phát triển các công cụ độc lập với giao thức của mình trong [Chương 11](#) với `getaddrinfo`

và các hàm `getnameinfo`.

Ở cắm TCP cung cấp luồng byte cho ứng dụng: Không có điểm đánh dấu bắn ghi.

Giá trị trả về từ một lần `đọc` có thể nhỏ hơn giá trị chúng tôi yêu cầu, nhưng điều này không cho thấy có lỗi. Để giúp đọc và ghi một luồng byte, chúng tôi đã phát triển ba hàm, `readn`, `write` và `readline` mà chúng tôi sẽ sử dụng trong toàn bộ văn bản. Tuy nhiên, các chương trình mạng nên được viết để hoạt động trên bộ đệm hơn là trên dòng.

Bài tập

[3.1](#) Tại sao các đối số kết quả giá trị như độ dài của cấu trúc địa chỉ socket phải được truyền bằng tham chiếu?

[3.2](#) Tại sao cả hàm **đọc** và hàm **ghi** đều sao chép con trỏ **void*** vào con trỏ **char*** ?

3.3 Các hàm **inet_aton** và **inet_addr** theo truyền thống được tự do chấp nhận chuỗi địa chỉ IPv4 thập phân có dấu chấm: cho phép từ một đến bốn số được phân tách bằng dấu thập phân và cho phép số **0x** đứng đầu để chỉ định số thập lục phân hoặc số **0** đứng đầu để chỉ định một số bát phân. (Hãy thử **telnet 0xe** để xem hành vi này.) **inet_pton** nghiêm ngặt hơn nhiều với địa chỉ IPv4 và yêu cầu chính xác bốn số cách nhau ba dấu thập phân, với mỗi số là số thập phân từ **0** đến **255**. **inet_pton** không cho phép số thập phân có dấu chấm được chỉ định khi họ địa chỉ là **AF_INET6**, mặc dù người ta có thể lập luận rằng những điều này nên được cho phép và giá trị trả về phải là địa chỉ IPv6 được ánh xạ IPv4 cho chuỗi thập phân có dấu chấm ([Hình A.10](#)).

Viết hàm mới có tên **inet_pton_loose** để xử lý các tình huống sau: Nếu họ địa chỉ là **AF_INET** và **inet_pton** trả về **0**, hãy gọi **inet_aton** và xem liệu nó có thành công hay không. Tương tự, nếu họ địa chỉ là **AF_INET6** và **inet_pton**

trả về **0**, gọi **inet_aton** và nếu thành công, hãy trả về địa chỉ IPv6 được ánh xạ IPv4.

Chương 4. Ở cắm TCP cơ bản

[Mục 4.1. Giới thiệu](#)

[Mục 4.2. chức năng ở cắm](#)

[Mục 4.3. chức năng kết nối](#)

[Mục 4.4. chức năng liên kết](#)

[Mục 4.5. chức năng lắng nghe](#)

[Mục 4.6. chấp nhận chức năng](#)

[Mục 4.7. Chức năng fork và exec](#)

[Mục 4.8. Máy chủ đồng thời](#)

[Mục 4.9. đóng chức năng](#)

[Mục 4.10. Hàm gethostname và getpeername](#)

Mục 4.11. Bản tóm tắtBài tập

4.1 Giới thiệu

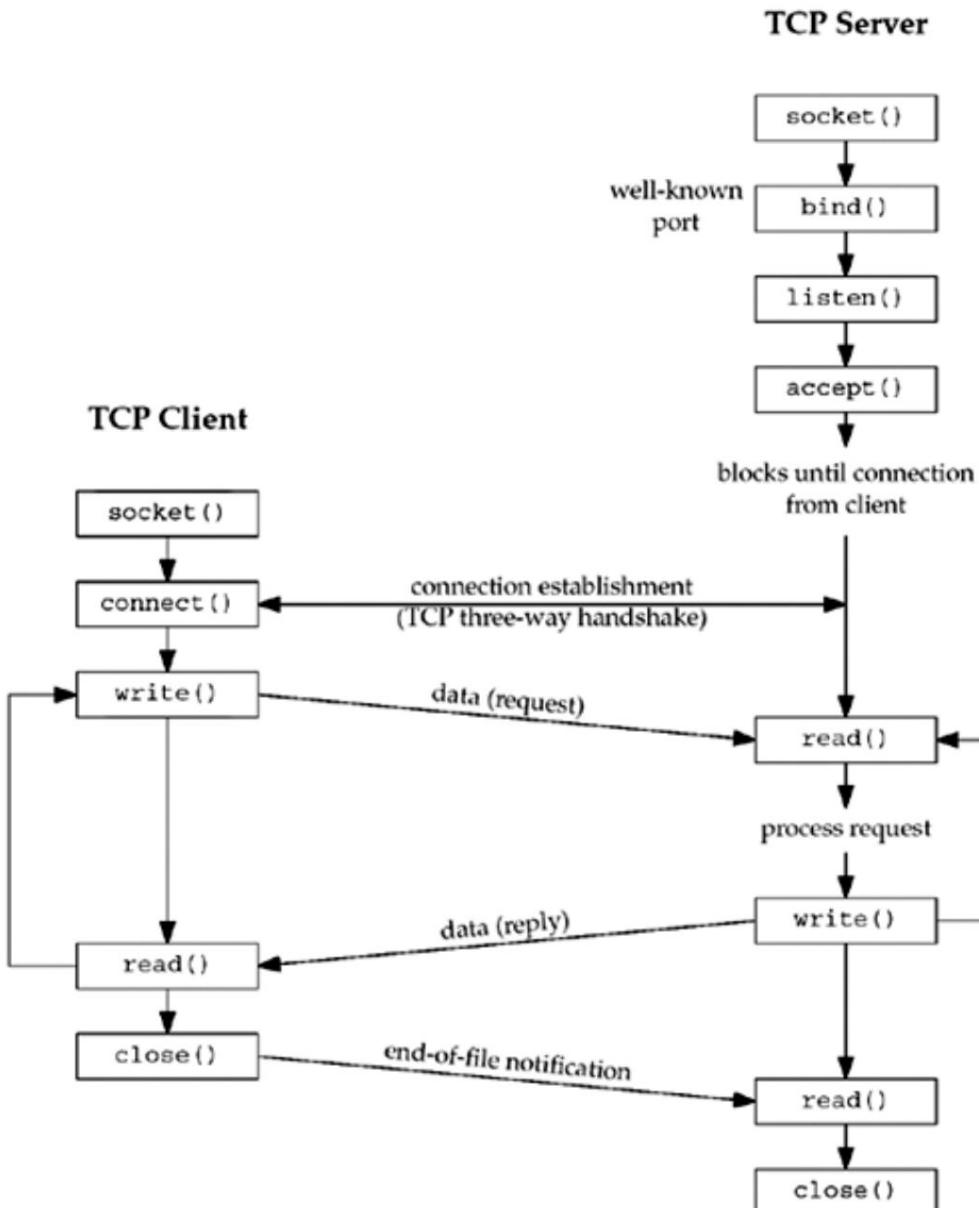
Chương này mô tả các hàm socket cơ bản cần thiết để viết một máy khách và máy chủ TCP hoàn chỉnh. Đầu tiên chúng ta sẽ mô tả tất cả các hàm socket cơ bản mà chúng ta sẽ sử dụng, sau đó phát triển máy khách và máy chủ trong chương tiếp theo. Chúng tôi sẽ làm việc với máy khách và máy chủ này trong suốt văn bản, cài tiến nó nhiều lần ([Hình 1.12](#) và [1.13](#)).

Chúng tôi cũng sẽ mô tả các máy chủ đồng thời, một kỹ thuật Unix phổ biến để cung cấp khả năng xử lý đồng thời khi nhiều máy khách được kết nối với cùng một máy chủ cùng một lúc. Mỗi kết nối máy khách khiến máy chủ phân nhánh một quy trình mới chỉ dành cho máy khách đó. Trong chương này, chúng ta chỉ xem xét mô hình một tiến trình trên mỗi máy khách bằng cách sử dụng `fork`, như ng chúng ta sẽ xem xét mô hình một luồng trên mỗi máy khách khác khi chúng ta mô tả các luồng trong [Chương 26](#).

[Hình 4.1](#) cho thấy dòng thời gian của kịch bản diễn hình diễn ra giữa máy khách và máy chủ TCP. Đầu tiên, máy chủ được khởi động, sau đó, một máy khách được khởi động để kết nối với máy chủ. Chúng tôi giả định rằng máy khách gửi yêu cầu đến máy chủ, máy chủ xử lý yêu cầu và máy chủ gửi phản hồi lại cho máy khách. Điều này tiếp tục cho đến khi máy khách đóng kết nối, gửi thông báo cuối tệp đến máy chủ. Sau đó, máy chủ sẽ đóng kết nối và

chấm dứt hoặc chờ kết nối máy khách mới.

Hình 4.1. Các hàm socket cho máy khách/máy chủ TCP cơ bản.



4.2 Chức năng 'ở cắm'

Để thực hiện I/O mạng, điều đầu tiên một tiến trình phải làm là gọi hàm `socket`, chỉ định loại giao thức truyền thông mong muốn (TCP sử dụng IPv4, UDP sử dụng IPv6, giao thức luồng miền Unix, v.v.).

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

```
int socket ( họ int, kiểu int , giao thức int );
```

Trả về: bộ mô tả không âm nếu OK, -1 nếu có lỗi

họ chỉ định họ giao thức và là một trong các hằng số được hiển thị trong [Hình 4.2](#).

Đối số này thường được gọi là mặc định thay vì gia đình. Loại ô cắm là một trong các hằng số được hiển thị trong [Hình 4.3](#). Đối số giao thức cho chức năng ô cắm phải được đặt thành loại giao thức cụ thể được tìm thấy trong [Hình 4.4](#) hoặc 0 để chọn mặc định của hệ thống cho sự kết hợp giữa họ và loại đã cho.

Hình 4.2. Các hằng số họ giao thức cho hàm socket.

<i>family</i>	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 15)
AF_ROUTE	Routing sockets (Chapter 18)
AF_KEY	Key socket (Chapter 19)

Hình 4.3. loại ô cắm cho chức năng ô cắm.

<i>type</i>	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket
SOCK_RAW	raw socket

Hình 4.4. giao thức ô cắm cho AF_INET hoặc AF_INET6.

<i>Protocol</i>	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Không phải tất cả sự kết hợp giữa họ và loại socket đều hợp lệ. [Hình 4.5](#) cho thấy các kết hợp hợp lệ, cùng với các giao thức thực tế hợp lệ cho mỗi cặp. Các ô được đánh dấu "Có" là hợp lệ như ng không có từ viết tắt hữu ích. Các ô trống không được hỗ trợ.

Hình 4.5. Sự kết hợp của họ và loại cho chức năng ô cắm.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP SCTP	TCP SCTP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_SEQPACKET	SCTP	SCTP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Bạn cũng có thể gặp hằng số PF_XXX tương ứng làm đối số đầu tiên cho socket. Chúng tôi sẽ nói thêm về điều này ở cuối phần này.

Chúng tôi lưu ý rằng bạn có thể gặp **AF_UNIX** (tên Unix lịch sử) thay vì **AF_LOCAL** (tên POSIX) và chúng tôi sẽ nói thêm về điều này trong [Chương 15](#).

Có các giá trị khác cho các đối số họ và kiểu . Ví dụ: 4.4BSD hỗ trợ cả **AF_NS** (giao thức Xerox NS, thường được gọi là XNS) và **AF_ISO** (giao thức OSI). Tương tự, loại **SOCK_SEQPACKET**, ở cắm gói tuần tự, được triển khai bởi cả giao thức Xerox NS và giao thức OSI và chúng tôi sẽ mô tả cách sử dụng nó với SCTP trong [Phần 9.2](#). Tuy nhiên, TCP là giao thức luồng byte và chỉ hỗ trợ các ở cắm **SOCK_STREAM** .

Linux hỗ trợ một loại socket mới, **SOCK_PACKET**, cung cấp quyền truy cập vào liên kết dữ liệu, tương tự như BPF và DLPI trong [Hình 2.1](#). Chúng ta sẽ nói nhiều hơn về điều này ở [Chương 29](#).

Ở cắm khóa, **AF_KEY**, mới hơn các ở cắm khác. Nó cung cấp hỗ trợ cho bảo mật mật mã. Tương tự như cách ở cắm định tuyến (**AF_ROUTE**) là một giao diện với bảng định tuyến của hạt nhân, ở cắm khóa là một giao diện vào bảng khóa của hạt nhân. Xem [Chương 19](#) để biết chi tiết.

Khi thành công, hàm **socket** trả về một giá trị nguyên nhỏ không âm, tương tự như bộ mô tả tệp. Chúng tôi gọi đây là bộ mô tả ở cắm hoặc sockfd. Để có được bộ mô tả ở cắm này, tất cả những gì chúng tôi đã chỉ định là họ giao thức (IPv4, IPv6 hoặc Unix) và loại ở cắm (luồng, datagram hoặc raw). Chúng tôi chưa chỉ định địa chỉ giao thức cụ bộ hoặc địa chỉ giao thức nư ớc ngoài.

AF_xxx so với PF_xxx

Tiền tố "**AF_**" là viết tắt của "họ địa chỉ" và tiền tố "**PF_**" là viết tắt của "họ giao thức". Về mặt lịch sử, mục đích là một họ giao thức duy nhất có thể hỗ trợ nhiều họ địa chỉ và giá trị **PF_** được sử dụng để tạo ở cắm và giá trị **AF_** được sử dụng trong cấu trúc địa chỉ ở cắm. Như ng trên thực tế, một họ giao thức hỗ trợ nhiều họ địa chỉ chia bao giờ được hỗ trợ và tiêu đề `<sys/socket.h>` xác định giá trị **PF_** cho một giao thức nhất định bằng với giá trị **AF_** cho giao thức đó. Mặc dù không có gì đảm bảo rằng sự bình đẳng giữa hai giao thức này sẽ luôn đúng, như ng nếu bất kỳ ai thay đổi điều này đối với các giao thức hiện có, rất nhiều mã hiện có sẽ bị hỏng. Để phù hợp với thực tiễn mã hóa hiện có, chúng tôi chỉ sử dụng hằng số **AF_** trong văn bản này, mặc dù bạn có thể gặp giá trị **PF_** , chủ yếu trong các cuộc gọi

đến ở cắm.

Nhìn vào 137 chương trình gọi **socket** trong bản phát hành BSD/OS 2.1 cho thấy 143 lệnh gọi chỉ định giá trị **AF_** và chỉ 8 lệnh gọi chỉ định giá trị **PF_** .

Về mặt lịch sử, lý do cho các tập hợp hằng số tương tự có tiền tố **AF_** và **PF_** bắt nguồn từ 4.1cBSD [Lanciani 1996] và một phiên bản của hàm **socket** có trục phiền bản mà chúng tôi đang mô tả (xuất hiện cùng với 4.2BSD). Phiên bản ở cắm 4.1cBSD có bốn đối số, một trong số đó là con trỏ tới **sockproto**

kết cấu. Thành viên đầu tiên của cấu trúc này được đặt tên là `sp_family` và giá trị của nó là một trong các giá trị `PF_`. Thành viên thứ hai, `sp_protocol`, là số giao thức, tương tự như đối số thứ ba của `socket` ngày nay. Việc chỉ định cấu trúc này là cách duy nhất để chỉ định họ giao thức. Do đó, trong hệ thống ban đầu này, các giá trị `PF_` được sử dụng làm thẻ cấu trúc để chỉ định họ giao thức trong cấu trúc `sockproto` và các giá trị `AF_` được sử dụng làm thẻ cấu trúc để chỉ định họ địa chỉ trong cấu trúc địa chỉ socket. Cấu trúc `sockproto` vẫn ở dạng 4.4BSD (trang 626-627 của TCPv2), nhưng chỉ được hạt nhân sử dụng nội bộ. Định nghĩa ban đầu có nhận xét "họ giao thức" cho thành viên `sp_family`, nhưng điều này đã được thay đổi thành "họ địa chỉ" trong mã nguồn 4.4BSD.

Để gây nhầm lẫn hơn nữa sự khác biệt này giữa các hằng số `AF_` và `PF_`, cấu trúc dữ liệu hạt nhân Berkeley chứa giá trị được so sánh với đối số đầu tiên của `socket` (thành viên `dom_family` của cấu trúc `miền`, trang 187 của TCPv2) có nhận xét rằng nó chứa giá trị `AF_`. Tuy nhiên, một số tên `miền`

các cấu trúc bên trong kernel được khởi tạo thành giá trị `AF_` tương ứng (tr. 192 của TCPv2) trong khi các cấu trúc khác được khởi tạo thành giá trị `PF_` (tr. 646 của TCPv2 và tr. 229 của TCPv3).

Như một ghi chú lịch sử khác, trang man 4.2BSD dành cho `socket`, ra mắt vào tháng 7 năm 1983, gọi đối số đầu tiên của nó là af và liệt kê các giá trị có thể có dưới dạng hằng số `AF_`.

Cuối cùng, chúng tôi lưu ý rằng tiêu chuẩn POSIX chỉ định rằng đối số đầu tiên cho `socket` là giá trị `PF_` và giá trị `AF_` được sử dụng cho cấu trúc địa chỉ ở cắm. Nhưng, sau đó chỉ xác định một giá trị họ trong cấu trúc `addrinfo` (Phần 11.6), dành cho sử dụng trong cuộc gọi đến ở cắm hoặc trong cấu trúc địa chỉ ở cắm!

4.3 Chức năng 'kết nối'

Chức năng `kết nối` được máy khách TCP sử dụng để thiết lập kết nối với TCP máy chủ.

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

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

Trả về: 0 nếu OK, -1 nếu có lỗi

`sockfd` là một bộ mô tả socket được hàm `socket` trả về. Đối số thứ hai và thứ ba là một con trỏ tới cấu trúc địa chỉ socket và kích thước của nó, như được mô tả trong Phần 3.3. Cấu trúc địa chỉ socket phải chứa địa chỉ IP và số cổng của máy chủ. Chúng ta đã thấy một ví dụ về hàm này trong Hình 1.5.

Máy khách không phải gọi **liên kết** (mà chúng tôi sẽ mô tả trong phần tiếp theo) trước khi gọi **kết nối**: kernel sẽ chọn cả cổng tạm thời và địa chỉ IP nguồn nếu cần.

Trong trường hợp ở cắm TCP, chức năng **kết nối** sẽ khởi tạo quá trình bắt tay ba bước của TCP ([Phần 2.6](#)). Hàm chỉ trả về **khi kết nối được thiết lập** hoặc xảy ra lỗi. Có thể có một số lỗi trả về khác nhau.

1. Nếu máy khách TCP không nhận được phản hồi nào đối với phân đoạn SYN của nó, **ETIMEDOUT** sẽ được trả về.

Ví dụ: 4.4BSD sẽ gửi một SYN khi **kết nối** được gọi, một SYN khác sau 6 giây và một SYN khác sau 24 giây (tr. 828 của TCPv2). Nếu không nhận được phản hồi sau tổng cộng 75 giây, lỗi sẽ được trả về.

Một số hệ thống cung cấp quyền kiểm soát quản trị đối với thời gian chờ này; xem Phụ lục E của TCPv1.

2. Nếu phản hồi của máy chủ đối với SYN của máy khách là đặt lại (RST), điều này cho biết rằng không có quá trình nào đang chờ kết nối trên máy chủ lưu trữ tại cổng được chỉ định (tức là quá trình máy chủ có thể không chạy). Đây là một lỗi cứng và lỗi **ECONNREFUSED** được trả về máy khách ngay khi nhận được RST.

RST là một loại phân đoạn TCP được TCP gửi khi có sự cố.

Ba điều kiện tạo ra RST là: khi SYN đến một cổng không có máy chủ lắng nghe (điều chúng tôi vừa mô tả), khi TCP muốn hủy kết nối hiện có và khi TCP nhận được một phân đoạn cho kết nối không tồn tại. (TCPv1 [trang 246-250] chứa thông tin bổ sung.)

3. Nếu SYN của khách hàng gửi ra một ICMP "dịch không thể truy cập" từ một số

bộ định tuyến trung gian, đây được coi là một lỗi phần mềm. Nhân máy khách lưu tin nhắn nhưng vẫn tiếp tục gửi các SYN vào cùng thời điểm giữa mỗi SYN như trong kịch bản đầu tiên. Nếu không nhận được phản hồi sau một khoảng thời gian cố định (75 giây cho 4.4BSD), lỗi ICMP đã lưu sẽ được trả về quy trình dưới dạng **EHOSTUNREACH** hoặc **ENETUNREACH**. Cũng có thể hệ thống từ xa không thể truy cập được bằng bất kỳ tuyến nào trong bảng chuyển tiếp của hệ thống cục bộ hoặc cuộc gọi **kết nối** sẽ quay lại mà không phải chờ đợi gì cả.

Nhiều hệ thống trước đó, chẳng hạn như 4.2BSD, đã hủy bỏ nỗ lực thiết lập kết nối một cách không chính xác khi nhận được "destination unreachable" ICMP.

Điều này sai vì lỗi ICMP này có thể chỉ ra tình trạng nhất thời. Ví dụ: có thể tình trạng này là do sự cố định tuyến gây ra.
trở nên chuẩn xác.

Lưu ý rằng **ENETUNREACH** không được liệt kê trong [Hình A.15](#), ngay cả khi lỗi cho biết mạng đích không thể truy cập được. Mạng không thể truy cập được

đư ợc coi là lỗi thời và các ứng dụng chỉ nên coi ENETUNREACH và EHOSTUNREACH là cùng một lỗi.

Chúng ta có thể thấy các tình trạng lỗi khác nhau này với ứng dụng khách đơn giản của mình từ [Hình 1.5, Tru ớc tiên](#), chúng tôi chỉ định máy chủ cục bộ (127.0.0.1), đang chạy máy chủ ban ngày và xem kết quả đầu ra.

```
Solaris % ban ngàytcpcli 127.0.0.1
Chủ nhật ngày 27 tháng 7 22:01:51 2003
```

Để xem định dạng khác cho thư trả lời đư ợc trả về, chúng tôi chỉ định IP của máy khác địa chỉ (trong ví dụ này là địa chỉ IP của máy HP-UX).

```
Solaris % ban ngàytcpcli 192.6.38.100
Chủ nhật ngày 27 tháng 7 22:04:59 PDT 2003
```

Tiếp theo, chúng tôi chỉ định một địa chỉ IP trên mạng con cục bộ (192.168.1/24) như ng ID máy chủ (100) không tồn tại. Nghĩa là, không có máy chủ nào trên mạng con có ID máy chủ là 100, vì vậy khi máy chủ khách gửi yêu cầu ARP (yêu cầu máy chủ đó phản hồi bằng địa chỉ phần cứng của nó), nó sẽ không bao giờ nhận đư ợc phản hồi ARP.

```
Solaris % ban ngàytcpcli 192.168.1.100
lỗi kết nối: Đã hết thời gian kết nối
```

Chúng tôi chỉ gặp lỗi sau khi hết thời gian **kết nối** (khoảng bốn phút với Solaris 9). Lưu ý rằng hàm **err_sys** của chúng tôi in chuỗi mà con người có thể đọc đư ợc liên quan đến lỗi **ETIMEDOUT**.

Ví dụ tiếp theo của chúng tôi là chỉ định một máy chủ (bộ định tuyến cục bộ) không chạy vào ban ngày máy chủ.

```
Solaris % ban ngàytcpcli 192.168.1.5
```

lỗi kết nối: Kết nối bị từ chối

Máy chủ phản hồi ngay lập tức bằng RST.

Ví dụ cuối cùng của chúng tôi chỉ định một địa chỉ IP không thể truy cập được trên Internet. Nếu chúng tôi xem các gói bằng **tcpdump**, chúng tôi sẽ thấy rằng bộ định tuyến cách đó sáu bước nhảy sẽ trả về lỗi không thể truy cập máy chủ ICMP.

```
Solaris % ban ngàytcpcli 192.3.4.5
```

lỗi kết nối: Không có đường đến máy chủ

Giống như lỗi **ETIMEDOUT**, trong ví dụ này, **kết nối** trả về **EHOSTUNREACH** chỉ xảy ra lỗi sau khi chờ khoảng thời gian được chỉ định.

Theo sơ đồ chuyển đổi trạng thái TCP ([Hình 2.4](#)), **kết nối** chuyển từ trạng thái ĐÓNG (trạng thái mà ở cắm bắt đầu khi nó được tạo bởi ở cắm).

function) sang trạng thái SYN_SENT, sau đó chuyển sang trạng thái THÀNH LẬP thành công. Nếu **kết nối** không thành công, ở cắm không còn sử dụng được nữa và phải đóng lại. Chúng tôi không thể gọi lại **kết nối** trên ở cắm. Trong [Hình 11.10](#), chúng ta sẽ thấy điều đó khi chúng ta gọi **connect** trong một vòng lặp, thử từng địa chỉ IP cho một máy chủ nhất định cho đến khi một địa chỉ hoạt động, mỗi lần **kết nối** không thành công, chúng ta phải đóng bộ mô tả socket và gọi lại **socket**.

4.4 Chức năng 'liên kết'

Hàm **liên kết** gán địa chỉ giao thức cụ bộ cho ở cắm. Với các giao thức Internet, địa chỉ giao thức là sự kết hợp của địa chỉ IPv4 32 bit hoặc địa chỉ IPv6 128 bit, cùng với số cổng TCP hoặc UDP 16 bit.

#include <sys/socket.h>	
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);	
	Trả về: 0 nếu OK, -1 nếu có lỗi

Trong lịch sử, mô tả trang man về **liên kết** đã nói "**liên kết** gán tên cho một ở cắm không tên." Việc sử dụng thuật ngữ "tên" gây nhầm lẫn và mang ý nghĩa của tên miền ([Chương 11](#)) chẳng hạn như **foo.bar.com**. **Chức năng liên kết** không có gì

để làm với tên. `bind` gán một địa chỉ giao thức cho một ô cắm và ý nghĩa của địa chỉ giao thức đó tùy thuộc vào giao thức.

Đối số thứ hai là một con trỏ tới một địa chỉ dành riêng cho giao thức và đối số thứ ba là kích thước của cấu trúc địa chỉ này. Với TCP, việc gọi `liên kết` cho phép chúng ta chỉ định số cổng, địa chỉ IP, cả hai hoặc không.

- Máy chủ liên kết cổng nổi tiếng của chúng khi chúng khởi động. Chúng ta thấy điều này trong [Hình 1.9](#).

Nếu máy khách hoặc máy chủ TCP không thực hiện việc này, kernel sẽ chọn một cổng tạm thời cho ô cắm khi `kết nối` hoặc `nghe` được gọi. Đó là điều bình thường đối với TCP máy khách cho phép kernel chọn một cổng tạm thời, trừ khi ứng dụng yêu cầu một cổng dành riêng ([Hình 2.10](#)), như ng hiêm khi máy chủ TCP cho phép kernel chọn một cổng tạm thời, vì các máy chủ được biết đến bởi cổng nổi tiếng của chúng.

Các ngoại lệ đối với quy tắc này là các máy chủ Gọi thủ tục từ xa (RPC). Họ thường để kernel chọn một cổng tạm thời cho socket nghe của mình vì cổng này sau đó được đăng ký với trình ánh xạ cổng RPC. Khách hàng phải liên hệ với người lập bản đồ cổng để lấy cổng tạm thời trước khi họ có thể `kết nối`

đến máy chủ. Điều này cũng áp dụng cho các máy chủ RPC sử dụng UDP.

- Một tiến trình có thể `liên kết` một địa chỉ IP cụ thể với socket của nó. Địa chỉ IP phải thuộc về một giao diện trên máy chủ. Đối với máy khách TCP, điều này chỉ định địa chỉ IP nguồn sẽ được sử dụng cho các gói dữ liệu IP được gửi trên ô cắm. Đối với máy chủ TCP, điều này hạn chế ô cắm nhận các kết nối máy khách đến chỉ dành cho địa chỉ IP đó.

Thông thường, máy khách TCP không `liên kết` địa chỉ IP với ô cắm của nó. Hạt nhân chọn địa chỉ IP nguồn khi ô cắm được kết nối, dựa trên giao diện gửi đi được sử dụng, giao diện này dựa trên tuyến đường cần thiết để đến máy chủ (tr. 737 của TCPv2).

Nếu máy chủ TCP không liên kết địa chỉ IP với ô cắm của nó, hạt nhân sẽ sử dụng địa chỉ IP đích của SYN của máy khách làm địa chỉ IP nguồn của máy chủ (tr. 943 của TCPv2).

Như chúng tôi đã nói, việc gọi `liên kết` cho phép chúng tôi chỉ định địa chỉ IP, cổng, cả hai hoặc không.

[Hình 4.6](#) tóm tắt các giá trị mà chúng ta đặt `sin_addr` và `sin_port` hoặc `sin6_addr` và `sin6_port`, tùy thuộc vào kết quả mong muốn.

[Hình 4.6](#). Kết quả khi chỉ định địa chỉ IP và/hoặc số cổng để liên kết.

Process specifies		Result
IP address	port	
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

Nếu chúng ta chỉ định số cổng là 0, kernel sẽ chọn một cổng tạm thời khi `liên kết` được gọi. Nhưng nếu chúng ta chỉ định một địa chỉ IP ký tự đại diện, kernel sẽ không chọn địa chỉ IP cụ bộ cho đến khi socket được kết nối (TCP) hoặc một datagram được gửi trên socket (UDP).

Với IPv4, địa chỉ ký tự đại diện được chỉ định bởi hằng số `INADDR_ANY`, có giá trị thường là 0. Điều này báo cho kernel chọn địa chỉ IP. Chúng ta đã thấy việc sử dụng điều này trong [Hình 1.9](#) với bài tập

```
struct sockaddr_in servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);                                /*ký tự đại diện*/
```

Mặc dù cách này hoạt động với IPv4, trong đó địa chỉ IP là giá trị 32 bit có thể được biểu diễn dưới dạng hằng số đơn giản (0 trong trường hợp này), chúng tôi không thể sử dụng kỹ thuật này với IPv6 vì địa chỉ IPv6 128 bit được lưu trữ trong một cấu trúc. (Trong C, chúng ta không thể biểu diễn một cấu trúc hằng số ở vé phải của phép gán.) Để giải quyết vấn đề này, chúng ta viết

```
struct sockaddr_in6 phục vụ;
serv.sin6_addr = in6addr_any;                                                 /*ký tự đại diện*/
```

Hệ thống phân bổ và khởi tạo biến `in6addr_any` thành hằng số `IN6ADDR_ANY_INIT`. Tiêu đề `<netinet/in.h>` chứa phần khai báo `bên ngoài` cho `in6addr_any`.

Giá trị của `INADDR_ANY` (0) giống nhau theo thứ tự byte mạng hoặc máy chủ, do đó việc sử dụng `htonl` là không thực sự cần thiết. Tuy nhiên, vì tất cả các hằng số `INADDR_` được xác định bởi tiêu đề `<netinet/in.h>` đều được xác định theo thứ tự byte máy chủ, nên chúng ta nên sử dụng `htonl` với bất kỳ hằng số nào trong số này.

Nếu chúng ta yêu cầu kernel chọn số cổng tạm thời cho socket của mình, hãy lưu ý rằng **liên kết** không trả về giá trị đã chọn. Thật vậy, nó không thể trả về giá trị này vì đối số thứ hai để **liên kết** có vòng loại **const**. Để có được giá trị của cổng tạm thời do kernel gán, chúng ta phải gọi **getsockname** để trả về địa chỉ giao thức.

Một ví dụ phổ biến về quy trình liên kết địa chỉ IP không phải ký tự đại diện với ô cắm là máy chủ cung cấp máy chủ Web cho nhiều tổ chức (Phần 14.2 của TCPv3).

Đầu tiên, mỗi tổ chức có tên miền riêng, chẳng hạn như www.Organisation.com.

Tiếp theo, tên miền của mỗi tổ chức ánh xạ tới một địa chỉ IP khác nhau, như ng thư ờng nằm trên cùng một mạng con. Ví dụ: nếu mạng con là 198.69.10 thì địa chỉ IP của tổ chức đầu tiên có thể là 198.69.10.128, 198.69.10.129 tiếp theo, v.v. Sau đó, tất cả các địa chỉ IP này được đặt bí danh trên một giao diện mạng duy nhất (ví dụ: sử dụng tùy chọn **bí danh** của lệnh **ifconfig** trên 4.4BSD) để lớp IP sẽ chấp nhận các gói dữ liệu đến dành cho bất kỳ địa chỉ bí danh nào. Cuối cùng, một bản sao của máy chủ HTTP được khởi động cho mỗi tổ chức và mỗi bản sao chỉ **liên kết** địa chỉ IP của tổ chức đó.

Một kỹ thuật thay thế là chạy một máy chủ duy nhất liên kết địa chỉ ký tự đại diện.

Khi có kết nối đến, máy chủ gọi **getsockname** để lấy địa chỉ IP đích từ máy khách, địa chỉ này trong cuộc thảo luận của chúng ta ở trên có thể là 198.69.10.128, 198.69.10.129, v.v. Sau đó, máy chủ xử lý yêu cầu của khách hàng dựa trên địa chỉ IP mà kết nối được cấp.

Một lợi thế trong việc liên kết một địa chỉ IP không phải ký tự đại diện là việc phân kênh của một địa chỉ IP đích nhất định tới một quy trình máy chủ nhất định sau đó sẽ được hạt nhân thực hiện.

Chúng ta phải cẩn thận để phân biệt giữa giao diện nơi gói đến và địa chỉ IP đích của gói đó.

Trong [Phần 8.8](#), chúng ta sẽ nói về mô hình hệ thống đầu yếu và mô hình hệ thống đầu cuối mạnh. Hầu hết các triển khai đều sử dụng cái trước, nghĩa là gói đến có địa chỉ IP đích xác định giao diện khác với giao diện mà gói đến. (Điều này giả sử một máy chủ nhiều nơi.) Việc liên kết một địa chỉ IP không phải ký tự đại diện sẽ hạn chế các gói dữ liệu sẽ được gửi đến ô cắm chỉ dựa trên địa chỉ IP đích. Nó không nói gì về giao diện đến, trừ khi máy chủ sử dụng mô hình hệ thống đầu cuối mạnh.

Một lỗi phổ biến khi **liên kết** là **EADDRINUSE** ("Địa chỉ đã được sử dụng"). Chúng ta sẽ nói nhiều hơn về điều này trong [Phần 7.5](#) khi nói về các tùy chọn socket **SO_REUSEADDR** và **SO_REUSEPORT**.

4.5 Chức năng 'nghe'

Chức năng **nghe** chỉ được gọi bởi máy chủ TCP và nó thực hiện hai hành động:

1. Khi một ô cắm được tạo bởi chức năng `socket`, nó được coi là một ô cắm đang hoạt động, tức là một ô cắm máy khách sẽ phát hành kết nối. Chức năng `listen` chuyển đổi một ô cắm không được kết nối thành một ô cắm thụ động, cho biết rằng hạt nhân phải chấp nhận các yêu cầu kết nối gửi đến trực tiếp đến ô cắm này. Xét về sơ đồ chuyển trạng thái TCP ([Hình 2.4](#)), lệnh gọi `listen` chuyển socket từ trạng thái ĐÓNG sang trạng thái LISTEN.
2. Đối số thứ hai của hàm này chỉ định số lượng tối đa các kết nối mà kernel sẽ xếp hàng cho socket này.

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

```
#int listen (int sockfd, int backlog);
```

Trả về: 0 nếu OK, -1 nếu có lỗi

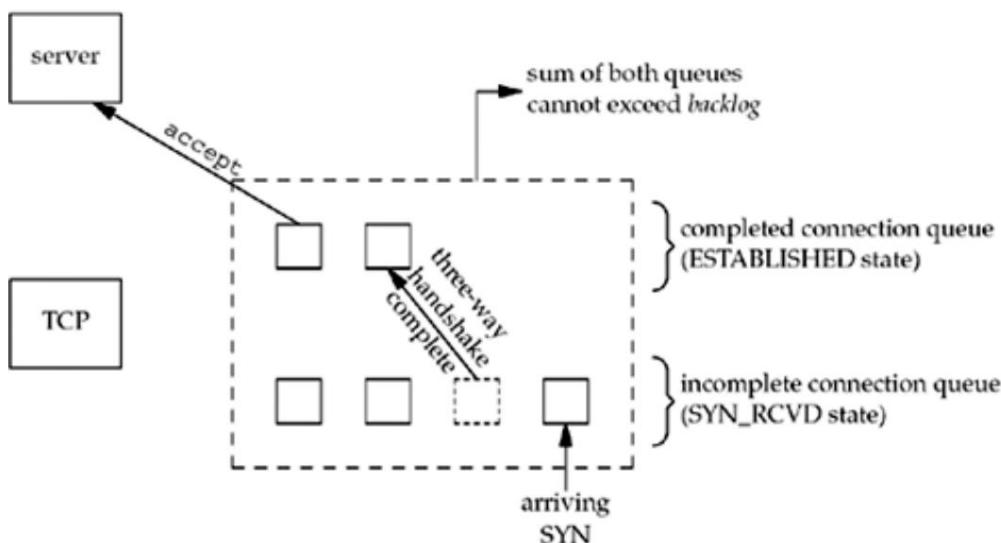
Hàm này thường được gọi sau cả hàm `socket` và hàm `bind` và phải được gọi trước khi gọi hàm `connect`.

Để hiểu rõ hơn, chúng ta phải nhận ra rằng đối với một socket nghe nhất định, kernel duy trì hai hàng đợi:

1. Hàng đợi kết nối chưa hoàn chỉnh, chưa mục nhập cho mỗi SYN đến từ máy khách mà máy chủ đang chờ hoàn thành bắt tay ba chiều TCP. Các socket này ở trạng thái SYN_RCVD ([Hình 2.4](#)).
2. Hàng đợi kết nối đã hoàn thành, trong đó có mục nhập cho mỗi máy khách có người mà quá trình bắt tay ba bước TCP đã hoàn thành. Các ô cắm này ở trạng thái THÀNH LẬP ([Hình 2.4](#)).

[Hình 4.7](#) mô tả hai hàng đợi này cho một socket nghe nhất định.

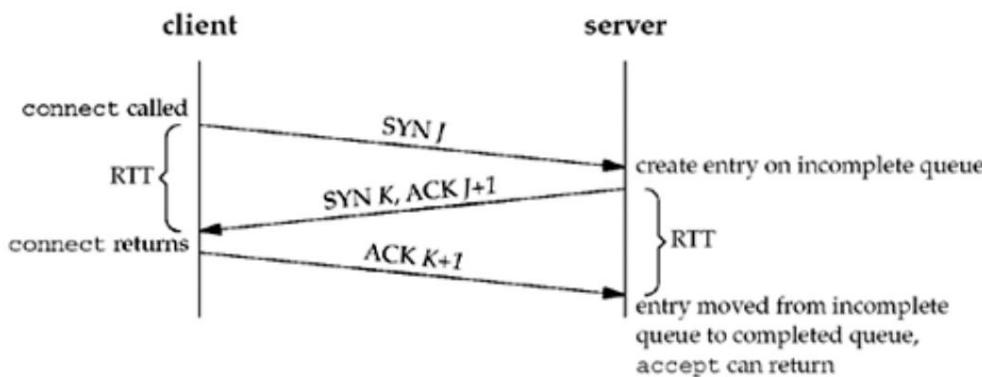
Hình 4.7. Hai hàng đợi được duy trì bởi TCP cho một ô cắm nghe.



Khi một mục nhập được tạo trên hàng đợi chưa hoàn chỉnh, các tham số từ ổ cắm nghe sẽ được sao chép sang kết nối mới được tạo. Cơ chế tạo kết nối hoàn toàn tự động; quá trình máy chủ không được tham gia. [Hình 4.8](#)

mô tả các gói được trao đổi trong quá trình thiết lập kết nối với hai gói này dưới.

Hình 4.8. Bắt tay ba chiều TCP và hai hàng đợi cho ổ cắm nghe.



Khi SYN đến từ máy khách, TCP tạo một mục mới trên hàng đợi chưa hoàn chỉnh và sau đó phản hồi bằng phân đoạn thứ hai của quá trình bắt tay ba chiều: SYN của máy chủ với ACK của SYN của máy khách ([Phần 2.6](#)). Mục nhập này vẫn nằm trong hàng đợi chưa hoàn chỉnh cho đến khi phân đoạn thứ ba của quá trình bắt tay ba chiều đến (ACK của máy khách trong SYN của máy chủ) hoặc cho đến khi mục nhập hết thời gian chờ. (Việc triển khai bắt nguồn từ Berkeley có thời gian chờ là 75 giây đối với những mục nhập chưa hoàn chỉnh này.) Nếu quá trình bắt tay ba chiều hoàn tất bình thường, thì mục nhập sẽ chuyển từ hàng đợi chưa hoàn thành đến cuối hàng đợi đã hoàn thành. Khi các cuộc gọi quy trình **đư ợc chấp nhận**, điều mà chúng tôi sẽ mô tả trong phần tiếp theo, mục nhập đầu tiên trên hàng đợi đã hoàn thành sẽ được trả về quy trình hoặc nếu hàng đợi trống, quy trình sẽ được chuyển sang chế độ ngủ cho đến khi một mục nhập được đặt vào hàng đợi đã hoàn thành.

Có một số điểm cần xem xét liên quan đến việc xử lý hai hàng đợi này.

- Đối số tồn đọng của hàm **listen** tru ớc đây đã chỉ định giá trị tối đa cho tổng của cả hai hàng đợi.

Chưa bao giờ có một định nghĩa chính thức về ý nghĩa của tồn đọng . Trang man 4.2BSD nói rằng nó "xác định độ dài tối đa mà hàng đợi kết nối đang chờ xử lý có thể tăng lên." Nhiều trang hướng dẫn và thậm chí cả đặc tả POSIX sao chép nguyên văn định nghĩa này, như định nghĩa này không cho biết liệu một kết nối đang chờ xử lý có phải là một kết nối ở trạng thái SYN_RCVD, một ở trạng thái THÀNH LẬP chưa được chấp nhận hay không. Định nghĩa lịch sử trong phần này là việc triển khai Berkeley, có niên đại từ 4.2BSD và được nhiều người khác sao chép.

- Việc triển khai bắt nguồn từ Berkeley thêm yếu tố giả mạo vào hồ sơ tồn đọng: Đó là nhau với 1,5 (trang 257 của TCPv1 và trang 462 của TCPv2). Ví dụ: tồn đọng thư ờng được chỉ định là 5 thực sự cho phép tối đa 8 mục được xếp hàng đợi trên các hệ thống này, như chúng tôi hiển thị [trong Hình 4.10](#).

Lý do thêm yếu tố giả tạo này được như đã bị quên lãng trong lịch sử [Joy 1994]. Như nếu chúng ta coi việc tồn đọng là chỉ định số lượng tối đa các kết nối đã hoàn thành mà hạt nhân sẽ xếp hàng cho một socket ([Borman 1997b], như đã thảo luận ngay sau đây), thì lý do cho hệ số giả mạo là để tính đến các kết nối không hoàn chỉnh trên hàng đợi. .

- Không chỉ định tồn đọng bằng 0, vì các cách triển khai khác nhau sẽ giải thích điều này khác nhau ([Hình 4.10](#)). Nếu bạn không muốn bắt kỳ ứng dụng khách nào kết nối với ồ cǎm nghe của mình, hãy đóng ồ cǎm nghe.
- Giả sử quá trình bắt tay ba bước diễn ra bình thường (nghĩa là không bị mất phân đoạn và không truyền lại), một mục vẫn nằm trong hàng đợi kết nối chưa hoàn chỉnh cho một RTT, bắt kể giá trị đó xảy ra giữa một máy khách và máy chủ cụ thể. Phần 14.4 của TCPv3 cho thấy rằng đối với một máy chủ Web, RTT trung bình giữa nhiều máy khách và máy chủ là 187 mili giây.
(Trung vị thư ờng được sử dụng cho thống kê này vì một vài giá trị lớn có thể làm lệch đáng kể giá trị trung bình.)
- Về mặt lịch sử, mã mẫu luôn hiển thị tồn đọng là 5, vì đó là giá trị tối đa được hỗ trợ bởi 4.2BSD. Điều này là đủ vào những năm 1980 khi các máy chủ bận rộn chỉ xử lý vài trăm kết nối mỗi ngày. Nhưng với sự phát triển của World Wide Web (WWW), nơi các máy chủ bận rộn xử lý hàng triệu kết nối mỗi ngày, con số nhỏ này hoàn toàn không đủ (trang 187-192 của TCPv3). Máy chủ HTTP bận phải chỉ định kích thước lớn hơn nhiều tồn đọng và các hạt nhân mới hơn phải hỗ trợ các giá trị lớn hơn.

Nhiều hệ thống hiện tại cho phép quản trị viên sửa đổi giá trị tối đa cho hồ sơ tồn đọng.

- Một vấn đề là: Ứng dụng nên chỉ định giá trị nào cho tồn đọng, vì 5 thư ờng không đủ? Không có câu trả lời dễ dàng cho điều này. Máy chủ HTTP hiện chỉ định một giá trị lớn hơn, nhưng nếu giá trị được chỉ định là một hằng số trong mã nguồn thì để tăng hằng số đó cần phải biên dịch lại máy chủ. Một phương pháp khác là giả sử một số mặc định như ng cho phép tùy chọn dòng lệnh hoặc biến môi trường ghi đè mặc định. Luôn có thể chấp nhận chỉ định một giá trị lớn hơn giá trị được hạt nhân hỗ trợ, vì hạt nhân sẽ âm thầm cắt bớt giá trị về giá trị tối đa mà nó hỗ trợ mà không trả về lỗi (tr. 456 của TCPv2).

Chúng tôi có thể cung cấp một giải pháp đơn giản cho vấn đề này bằng cách sửa đổi hàm bao bọc cho hàm [nghe](#). [Hình 4.9](#) cho thấy mã thực tế. Chúng tôi cho phép biến môi trường [LISTENQ](#) ghi đè giá trị do người gọi chỉ định.

Hình 4.9 Hàm bao bọc để nghe cho phép biến môi trường chỉ định tồn đọng.

lib/wrapsock.c

```

137 khoảng trống
138 Nghe (int fd, int backlog)
139 {
140     char *ptr;

141     /* có thể ghi đè đối số thứ 2 bằng biến môi trường */
142     if ((ptr = getenv("LISTENQ")) != NULL)
143         tồn đọng = atoi (ptr);

144     if (nghe (fd, tồn đọng) < 0)
145         err_sys ("lỗi nghe");
146 }

```

- Các sách hướng dẫn và sách xưa nay đã nói rằng lý do xếp hàng một số lượng kết nối cố định là để xử lý trường hợp tiến trình máy chủ đang bận giữa các cuộc gọi tiếp theo để **chấp nhận**. Điều này ngụ ý rằng trong hai hàng đợi, hàng đợi đã hoàn thành thưòng có nhiều mục hơn hàng đợi chưa hoàn chỉnh. Một lần nữa, các máy chủ Web bận rộn đã cho thấy điều này là sai. Lý do chỉ định lượng tồn đọng lớn là do hàng đợi kết nối chưa hoàn chỉnh có thể tăng lên khi SYN máy khách đến, chờ hoàn thành quá trình bắt tay ba chiều.
- Nếu hàng đợi đầy khi SYN máy khách đến, TCP sẽ bỏ qua SYN đến (trang 930-931 của TCPv2); nó không gửi RST. Điều này là do tình trạng này được coi là tạm thời và máy khách TCP sẽ truyền lại SYN của nó, hy vọng sẽ tìm được chỗ trống trên hàng đợi trong tương lai gần. Nếu máy chủ TCP phản hồi ngay lập tức bằng RST, **kết nối** của máy khách sẽ trả về lỗi, buộc ứng dụng phải xử lý tình trạng này thay vì để quá trình truyền lại thông thưòng của TCP tiếp quản. Ngoài ra, máy khách không thể phân biệt giữa RST phản hồi với SYN, nghĩa là "không có máy chủ ở cổng này" và "có máy chủ ở cổng này nhưng hàng đợi của nó đã đầy."

Một số triển khai sẽ gửi RST khi hàng đợi đầy. Hành vi này không chính xác vì những lý do đã nêu ở trên và trừ khi máy khách của bạn đặc biệt cần tương tác với máy chủ như vậy, tốt nhất bạn nên bỏ qua khả năng này. Việc mã hóa để xử lý trường hợp này làm giảm độ bền của máy khách và gây thêm tải cho mạng trong trường hợp RST thông thường, trong đó cổng thực sự không có máy chủ lắng nghe trên đó.

- Dữ liệu đến sau khi quá trình bắt tay ba bên hoàn tất như ng tru ớc khi cuộc gọi máy chủ **chấp nhận**, phải được xếp hàng bởi máy chủ TCP, có kích thước tối đa bằng bộ đệm nhận của ô cắm được kết nối.

Hình 4.10 cho thấy số lư ợng kết nối được xếp hàng thực tế được cung cấp cho các giá trị khác nhau của đối số tồn đọng cho các hệ điều hành khác nhau trong Hình 1.16. Đối với bảy hệ điều hành khác nhau, có năm cột riêng biệt, thể hiện nhiều cách hiểu khác nhau về ý nghĩa của backlog !

Hình 4.10. Số lư ợng kết nối được xếp hàng thực tế cho các giá trị tồn đọng.

backlog	Maximum actual number of queued connections				
	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

AIX và MacOS có thuật toán Berkeley truyền thống và Solaris thường như cũng rất gần với thuật toán đó. FreeBSD chỉ thêm một vào danh sách tồn đọng.

Chu ơng trình đo các giá trị này được trình bày trong lời giải của [Bài tập 15.4](#).

Như chúng tôi đã nói, về mặt lịch sử, tồn đọng đã chỉ định giá trị tối đa cho tổng của cả hai hàng đợi. Trong năm 1996, một kiểu tấn công mới đã được tung ra trên Internet có tên là SYN Flood [CERT 1996b]. Tin tức viết chu ơng trình gửi SYN với tốc độ cao đến nạn nhân, lấp đầy hàng đợi kết nối chưa hoàn chỉnh cho một hoặc nhiều cổng TCP. (Chúng tôi sử dụng thuật ngữ hacker để chỉ kẻ tấn công, như được mô tả trong [Cheswick, Bellovin và Rubin 2003].) Ngoài ra, địa chỉ IP nguồn của mỗi SYN được đặt thành một số ngẫu nhiên (điều này được gọi là giả mạo IP) để SYN/ACK của máy chủ không đi đến đâu. Điều này cũng ngăn máy chủ biết địa chỉ IP thực của hacker. Bằng cách lấp đầy hàng đợi chưa hoàn chỉnh bằng các SYN giả, các SYN hợp pháp sẽ không được xếp hàng đợi, gây ra tình trạng từ chối dịch vụ cho các khách hàng hợp pháp. Có hai phương pháp thường được sử dụng để xử lý các cuộc tấn công này, được tóm tắt trong [Borman 1997b]. Như ng điều thú vị nhất trong ghi chú này là xem lại ý nghĩa thực sự của việc **lắng nghe** tồn đọng. Nó phải chỉ định số lư ợng kết nối hoàn thành tối đa cho một ô cắm nhất định mà hạt nhân sẽ xếp hàng. Mục đích của việc có giới hạn đối với các kết nối đã hoàn thành này là để ngăn chặn

kernel chấp nhận các yêu cầu kết nối mới cho một socket nhất định khi ứng dụng không chấp nhận chúng (vì bất kỳ lý do gì). Nếu một hệ thống triển khai cách diễn giải này, cũng như BSD/OS 3.0, thì ứng dụng không cần chỉ định các giá trị tồn đọng lớn chỉ vì máy chủ xử lý nhiều yêu cầu của khách hàng (ví dụ: máy chủ Web bộn) hoặc để cung cấp khả năng bảo vệ chống tràn SYN. Hạt nhân xử lý rất nhiều kết nối không đầy đủ, bất kể chúng là hợp pháp hay từ tin tặc.

Như ngay cả với cách giải thích này, các tình huống vẫn xảy ra khi giá trị truyền thống là 5 không đủ.

4.6 Chức năng 'chấp nhận'

Chấp nhận được gọi bởi máy chủ TCP để trả về kết nối đã hoàn thành tiếp theo từ phía trước hàng đợi kết nối đã hoàn thành ([Hình 4.7](#)). Nếu hàng đợi kết nối đã hoàn thành trống, quá trình sẽ được chuyển sang chế độ ngủ (giả sử mặc định là ở cắm chặn).

```
#include <sys/socket.h>
int chấp nhận (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Trả về: bộ mô tả không âm nếu OK, -1 nếu có lỗi

Các đối số cliaddr và addrlen được sử dụng để trả về địa chỉ giao thức của tiến trình ngang hàng được kết nối (máy khách). addrlen là đối số giá trị-kết quả ([Phần 3.3](#)): Truớc cuộc gọi, chúng ta đặt giá trị số nguyên được tham chiếu bởi *addrlen theo kích thước của cấu trúc địa chỉ ở cắm được chỉ ra bởi cliaddr; khi trả về, giá trị số nguyên này chứa số byte thực tế được lưu trữ bởi kernel trong cấu trúc địa chỉ socket.

Nếu **chấp nhận** thành công, giá trị trả về của nó là một bộ mô tả hoàn toàn mới được kernel tự động tạo. Bộ mô tả mới này để cập đến kết nối TCP với máy khách.
Khi thảo luận về **việc chấp nhận**, chúng tôi gọi đối số đầu tiên để **chấp nhận** ở cắm nghe (bộ mô tả được tạo bởi **socket** và sau đó được sử dụng làm đối số đầu tiên cho cả **liên kết** và **lắng nghe**) và chúng tôi gọi giá trị trả về từ **chấp nhận** ở cắm được kết nối. Điều quan trọng là phải phân biệt giữa hai ở cắm này. Một máy chủ nhất định thường chỉ tạo một ở cắm nghe, ở cắm này tồn tại trong suốt thời gian tồn tại của máy chủ. Hạt nhân tạo một ở cắm được kết nối cho mỗi kết nối máy khách được **chấp nhận** (nghĩa là quá trình bắt tay ba chiều TCP hoàn tất). Khi máy chủ phục vụ xong một máy khách nhất định, ở cắm được kết nối sẽ đóng lại.

Hàm này trả về tối đa ba giá trị: mã trả về số nguyên là bộ mô tả ở cắm mới hoặc chỉ báo lỗi, địa chỉ giao thức của quy trình máy khách (thông qua con trỏ cliaddr) và kích thước của địa chỉ này (thông qua addrlen).

con trả). Nếu chúng tôi không quan tâm đến việc trả về địa chỉ giao thức của máy khách, chúng tôi đặt cả cliaddr và addrlen thành con trả null.

[Hình 1.9](#) thể hiện những điểm này. Ở cẩm được kết nối sẽ đóng mỗi lần qua vòng lặp, như ng ở cẩm nghe vẫn mở trong suốt thời gian hoạt động của máy chủ. Chúng ta cũng thấy rằng đối số thứ hai và thứ ba cần chấp nhận là các con trả rỗng, vì chúng ta không quan tâm đến danh tính của khách hàng.

Ví dụ: Đổi số giá trị-kết quả

Bây giờ chúng tôi sẽ trình bày cách xử lý đổi số giá trị-kết quả để chấp nhận bằng cách sửa đổi mã từ [Hình 1.9](#) để in địa chỉ IP và cổng của máy khách. Chúng tôi thể hiện điều này trong [Hình 4.11](#).

Hình 4.11 Máy chủ ban ngày in địa chỉ IP và cổng của máy khách

giới thiệu/daytimetcpsrv1.c

```

1 #include "unp.h" 2
2 #include <time.h>

3 int
4 chính(int argc, char **argv)
5 {
6     int      listenfd, connfd;
7     socklen_t len;
8     struct sockaddr_in servaddr, cliaddr;
9     buff char[MAXLINE];
10    time_t tích tắc;

11    listenfd = Ở cẩm(AF_INET, SOCK_STREAM, 0);
12    bzero(&servaddr, sizeof(servaddr));
13    servaddr.sin_family = AF_INET;
14    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15    servaddr.sin_port = htons(13); /*máy chủ ban ngày */

16    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

17    Nghe(listenfd, LISTENQ);

18    vì (    ;    ) {
19        len = sizeof(cliaddr);

```

```

20         connfd = Chấp nhận(listenfd, (SA *) &cliaddr, &len);
21         printf("kết nối từ %s, port %d\n",
22                 Inet_ntop(AF_INET, &cliaddr.sin_addr, buff,
23                           sizeof(buff)),
24                 ntohs(cliaddr.sin_port));
25
26         tích tắc = thời gian(NULL);
27         sprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
28         Write(connfd, buff, strlen(buff));
29     }

```

Khai báo mới

7-8 Chúng tôi xác định hai biến mới: `len`, sẽ là biến kết quả-giá trị và `cliaddr`, sẽ chứa địa chỉ giao thức của máy khách.

Chấp nhận kết nối và in địa chỉ của khách hàng

19-23 Chúng ta khởi tạo `len` theo kích thước của cấu trúc địa chỉ socket và chuyển một con trỏ tới cấu trúc `cliaddr` và một con trỏ tới `len` làm đối số thứ hai và thứ ba để chấp nhận. Chúng ta gọi `inet_ntop` ([Phần 3.7](#)) để chuyển đổi địa chỉ IP 32 bit trong cấu trúc địa chỉ socket thành chuỗi ASCII thập phân có dấu chấm và gọi `ntohs` ([Phần 3.4](#)) để chuyển đổi số cổng 16 bit từ thứ tự byte mạng sang thứ tự byte máy chủ .

Gọi `sock_ntop` thay vì `inet_ntop` sẽ làm cho máy chủ của chúng tôi độc lập hơn với giao thức, như ng máy chủ này đã phụ thuộc vào IPv4. Chúng tôi sẽ hiển thị phiên bản độc lập với giao thức của máy chủ này trong [Hình 11.13](#).

Nếu chúng tôi chạy máy chủ mới và sau đó chạy ứng dụng khách trên cùng một máy chủ, kết nối với máy chủ của chúng tôi hai lần liên tiếp, chúng tôi có kết quả đầu ra sau từ máy khách:

```

Solaris % ban ngàytcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
Solaris % ban ngàytcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003

```

Trước tiên, chúng tôi chỉ định địa chỉ IP của máy chủ làm địa chỉ loopback (127.0.0.1) và sau đó là địa chỉ IP của chính nó (192.168.1.20). Đây là đầu ra máy chủ tư ứng:

```
Solaris # ban ngàytcpserver1
kết nối từ 127.0.0.1, cổng 43388
kết nối từ 192.168.1.20, cổng 43389
```

Lưu ý điều gì xảy ra với địa chỉ IP của khách hàng. Vì máy khách ban ngày của chúng ta ([Hình 1.5](#)) không gọi liên kết nên chúng ta đã nói trong [Phần 4.4 rằng kernel chọn địa chỉ IP nguồn dựa trên giao diện gửi đi đư ợc sử dụng](#). Trong trường hợp đầu tiên, kernel đặt địa chỉ IP nguồn thành địa chỉ loopback; trong trường hợp thứ hai, nó đặt địa chỉ thành địa chỉ IP của giao diện Ethernet. Chúng ta cũng có thể thấy trong ví dụ này rằng cổng tạm thời đư ợc hạt nhân Solaris chọn là 43388 và sau đó là 43389 (xem lại [Hình 2.10](#)).

Điểm cuối cùng, dấu nhắc shell của chúng tôi dành cho tập lệnh máy chủ thay đổi thành dấu thăng (#), dấu nhắc thư ờng đư ợc sử dụng cho siêu ngư ời dùng. Máy chủ của chúng tôi phải chạy với đặc quyền siêu ngư ời dùng để **liên kết** cổng dành riêng là 13. Nếu chúng tôi không có đặc quyền siêu ngư ời dùng, cuộc gọi để **liên kết** sẽ thất bại:

```
Solaris % ban ngàytcpserver1
lỗi liên kết: Quyền bị từ chối
```

4.7 Chức năng 'fork' và 'exec'

Trước khi mô tả cách viết một máy chủ đồng thời trong phần tiếp theo, chúng ta phải mô tả chức năng Unix **fork**. Chức năng này (bao gồm các biến thể của nó đư ợc cung cấp bởi một số hệ thống) là cách duy nhất trong Unix để tạo một quy trình mới.

#include <unistd.h>
ngã ba pid_t(void);
Trả về: 0 ở con, ID tiến trình của con ở cha, -1 nếu có lỗi

Nếu bạn chưa từng thấy hàm này trước đây thì phần khó hiểu **nhất** là nó đư ợc gọi một lần như ng lại trả về hai lần. Nó trả về một lần trong quá trình gọi (đư ợc gọi là tiến trình cha) với giá trị trả về là ID tiến trình của tiến trình mới đư ợc tạo

(đứa trẻ). Nó cũng trả về một lần ở tiến trình con, với giá trị trả về là 0. Do đó, giá trị trả về cho biết tiến trình đó là tiến trình cha hay tiến trình con.

Lý do `fork` trả về 0 ở phần tử con, thay vì ID tiến trình của phần tử cha, là vì phần tử con chỉ có một phần tử cha và nó luôn có thể lấy ID tiến trình của phần tử cha bằng cách gọi `getppid`. Mặt khác, cha mẹ có thể có bất kỳ số lượng con nào và không có cách nào để lấy được ID tiến trình của con cái đó. Nếu cha mẹ muốn theo dõi ID tiến trình của tất cả các con của nó, thì nó phải ghi lại các giá trị trả về từ `ngã ba`.

Tất cả các bộ mô tả mở trong phần tử cha truớc khi lệnh gọi `fork` được chia sẻ với phần tử con sau khi `fork` quay trở lại. Chúng ta sẽ thấy tính năng này được các máy chủ mạng sử dụng: Cuộc gọi cha `chấp nhận` và sau đó gọi `fork`. Ở cẩm được kết nối sau đó sẽ được chia sẻ giữa cha mẹ và con. Thông thường, trẻ sẽ đọc và ghi ở cẩm được kết nối và cha mẹ sẽ đóng ở cẩm được kết nối.

Có hai cách sử dụng `nǐa diễn hình`:

1. Một tiến trình tạo một bản sao của chính nó để một bản sao có thể xử lý một thao tác trong khi bản sao kia thực hiện một nhiệm vụ khác. Đây là *diễn hình* cho các máy chủ mạng. Chúng ta sẽ thấy nhiều ví dụ về điều này sau trong văn bản.
2. Một tiến trình muốn thực thi một chương trình khác. Vì cách duy nhất để tạo một tiến trình mới là gọi `fork`, nên truớc tiên tiến trình này gọi `fork` để tạo một bản sao của chính nó, sau đó một trong các bản sao (thường là tiến trình con) gọi `exec` (mô tả tiếp theo) để thay thế chính nó bằng chương trình mới. Đây là *diễn hình* cho các chương trình như `shell`.

Cách duy nhất mà Unix có thể thực thi một tệp chương trình thực thi trên đĩa là một quy trình hiện có gọi một trong sáu hàm `exec`. (Chúng ta thường gọi chung là "hàm `exec`" khi việc gọi cái nào trong số sáu cái đó không quan trọng.) `exec` thay thế hình ảnh tiến trình hiện tại bằng tệp chương trình mới và chương trình mới này thường bắt đầu ở hàm `chính`. ID tiến trình không thay đổi. Chúng ta gọi tiến trình gọi `exec` là tiến trình gọi và chương trình mới được thực thi là chương trình mới.

Các sách hướng dẫn và sách cũ hơn gọi chương trình mới là quy trình mới một cách không chính xác, điều này sai vì quy trình mới không được tạo.

Sự khác biệt trong sáu hàm `exec` là: (a) liệu tệp chương trình cần thực thi được chỉ định bằng tên tệp hay tên đường dẫn; (b) liệu các đối số của chương trình mới được liệt kê từng cái một hay được tham chiếu thông qua một mảng con trỏ; và (c) liệu môi trường của quá trình gọi có được chuyển sang chương trình mới hay không hoặc liệu môi trường mới có được chỉ định hay không.

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

```
#include <unistd.h>

int execl (const char *tên chương trình, const char *arg0, ... /* (char *) 0 */ );

int execv (const char *tên chương trình, char *const argv[]);

int execle (const char *tên chương trình, const char *arg0, ...
             /* (char *) 0, char *const envp[] */ );

int execve (const char *pathname, char *const argv[], char *const envp[]);

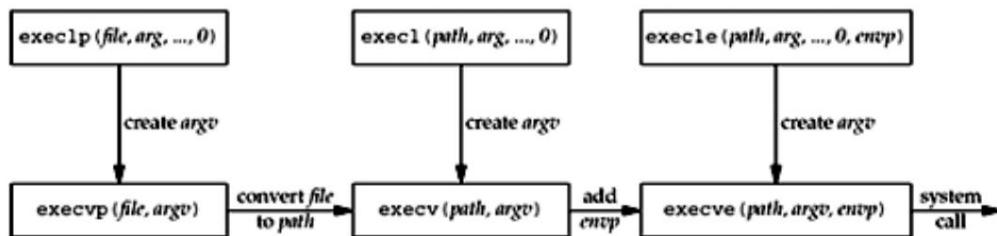
int execlp (const char *tên tập, const char *arg0, ... /* (char *) 0 */ );

int execvp (const char *tên tập, char *const argv[]);
```

Các hàm này chỉ trả về cho người gọi nếu xảy ra lỗi. Nếu không, quyền điều khiển sẽ chuyển sang phần bắt đầu của chương trình mới, thông thường là chức năng **chính**.

Mỗi quan hệ giữa sáu chức năng này được thể hiện trong [Hình 4.12.](#) Thông thư ống, chỉ `execve` là lệnh gọi hệ thống trong kernel và năm cái còn lại là các hàm thư viện thực hiện cuộc gọi .

Hình 4.12. Mối quan hệ giữa sáu chức năng điều hành.



Lưu ý những khác biệt sau đây giữa sáu chức năng này:

1. Ba hàm ở hàng trên cùng chỉ định mỗi chuỗi đối số là một đối số riêng biệt cho hàm `exec`, với một con trỏ null kết thúc số lượng đối số có thể thay đổi. Ba hàm ở hàng thứ hai có một `argv`mảng, chứa các con trỏ tới chuỗi đối số. Mảng `argv` này phải chứa một con trỏ null để chỉ định phần cuối của nó, vì số lượng không được chỉ định.
 2. Hai hàm ở cột bên trái chỉ định đối số tên tệp . Đây làđược chuyển đổi thành tên đường dẫn bằng biến môi trường `PATH` hiện tại . Nếu đối số tên tệp của `execvp` hoặc `execv` chứa dấu gạch chéo (/) ở bất kỳ đâu trong chuỗi thì biến `PATH` không được sử dụng. Bốn hàm ở hai cột bên phải chỉ định một đối số tên đường dẫnđủ điều kiện .

3. Bốn hàm ở hai cột bên trái không chỉ định rõ ràng
 con trỏ môi trường Thay vào đó, giá trị hiện tại của biến môi trường bên ngoài được sử dụng để
 xây dựng danh sách môi trường được chuyển cho chương trình mới. Hai hàm ở cột bên phải chỉ định danh
 sách môi trường rõ ràng. Mảng con trỏ envp phải được kết thúc bằng con trỏ null.

Các bộ mô tả mở trong quá trình truy cập khi gọi `exec` thư ứng vẫn mở trên toàn bộ `exec`. Chúng tôi sử dụng vòng loại "bình
 thư ứng" vì điều này có thể bị vô hiệu hóa bằng cách sử dụng `fcntl` để đặt cờ mô tả `FD_CLOEXEC`. Máy chủ `inetd` sử dụng
 tính năng này, như chúng tôi sẽ mô tả trong [Phần 13.5](#).

4.8 Máy chủ đồng thời

Máy chủ trong [Hình 4.11](#) là một máy chủ lặp. Đối với một cái gì đó đơn giản như một máy chủ ban
 ngày, điều này là ổn. Như ng khi một yêu cầu của khách hàng có thể mất nhiều thời gian hơn để phục
 vụ, chúng tôi không muốn ràng buộc một máy chủ với một khách hàng; chúng tôi muốn xử lý nhiều khách
 hàng cùng một lúc. Cách đơn giản nhất để viết một máy chủ đồng thời trong Unix là [phân nhánh](#) một
 tiến trình con để xử lý từng máy khách. [Hình 4.13](#) thể hiện phác thảo của một hệ thống đồng thời điển hình
 máy chủ.

Hình 4.13 Sơ lược về máy chủ đồng thời điển hình.

```
pid_t pid;
int listenfd, connfd;

listenfd = Ở cắm( ... );

/* diễn socksaddr_in{} bằng công thông dụng của máy chủ */
Bind(listenfd, ... );
Nghe(listenfd, LISTENQ);

vì (      ;      ) {
    connfd = Chấp nhận (listenfd, ... );           /* có lẽ sẽ chặn */

    if( (pid = Fork()) == 0) {
        Đóng(nghefd);          /* trẻ em đóng ở cắm nghe */
        phái(connfd);          /* xử lý yêu cầu */
        Đóng(connfd);          /* hoàn tất với client này */
        thoát (0);             /* con kết thúc */
    }
}
```

```

Đóng(connfd);           /* cha mẹ đóng socket đã kết nối */
}

```

Khi một kết nối được thiết lập, **chấp nhận** trả về, máy chủ gọi **fork** và tiến trình con phục vụ máy khách (trên **connfd**, ô cắm được kết nối) và tiến trình mẹ chờ kết nối khác (trên **listenfd**, ô cắm nghe). Phụ huynh đóng cửa

ô cắm được kết nối kể từ khi đưa trẻ xử lý máy khách mới.

Trong [Hình 4.13](#), chúng ta giả định rằng hàm **doit** thực hiện bất cứ điều gì được yêu cầu để phục vụ máy khách. Khi hàm này trả về, chúng ta **đóng** ô cắm được kết nối ở phần tử con một cách rõ ràng. Điều này là không bắt buộc vì câu lệnh tiếp theo gọi **exit** và một phần của việc kết thúc quá trình là đóng tất cả các bộ mô tả đang mở của kernel. Việc có bao gồm lệnh gọi rõ ràng này để **đóng** hay không là vấn đề sở thích lập trình cá nhân.

Chúng ta đã nói [trong Phần 2.6](#) rằng việc gọi **đóng** trên ô cắm TCP sẽ khiến FIN được gửi đi, theo sau là trình tự chấm dứt kết nối TCP thông thường. Tại sao không **đóng**

của **connfd** trong [Hình 4.13](#) bởi cha mẹ chấm dứt kết nối của nó với máy khách? Để hiểu điều gì đang xảy ra, chúng ta phải hiểu rằng mọi tệp hoặc ô cắm đều có số lượng tham chiếu. Số lượng tham chiếu được duy trì trong mục nhập bảng tệp (trang 57-60 của APUE). Đây là số lượng bộ mô tả hiện đang mở tham chiếu đến tệp hoặc ô cắm này. Trong [Hình 4.13](#), sau khi **socket** trả về, mục nhập bảng tệp được liên kết với **listenfd** có số lượng tham chiếu là 1. Sau khi trả về **chấp nhận**, mục nhập bảng tệp được liên kết với **connfd** có số lượng tham chiếu là 1. Tuy nhiên, sau khi **fork** trả về, cả hai bộ mô tả đều được chia sẻ (tức là trùng lặp) giữa cha và con, vì vậy các mục trong bảng tệp được liên kết với cả hai ô cắm hiện có số tham chiếu là 2. Do đó, khi cha đóng **connfd**, nó chỉ giảm số tham chiếu từ 2 xuống 1 và chỉ thế thôi. Việc đơn dẹp và phân bổ lại ô cắm thực tế không xảy ra cho đến khi số tham chiếu đạt đến 0. Điều này sẽ xảy ra sau đó khi trẻ

đóng **connfd**.

Chúng ta cũng có thể hình dung các socket và kết nối xuất hiện trong [Hình 4.13](#) như sau.

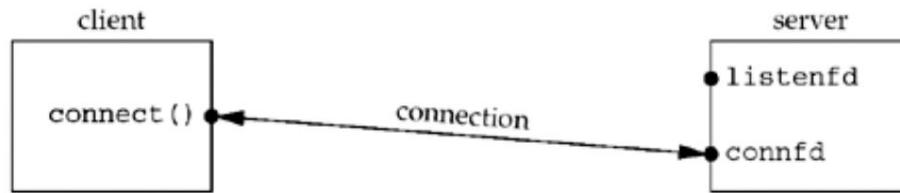
Đầu tiên, [Hình 4.14](#) hiển thị trạng thái của máy khách và máy chủ trong khi máy chủ bị chặn trong cuộc gọi **chấp nhận** và yêu cầu kết nối đến từ máy khách.

Hình 4.14. Trạng thái của client/server trước khi gọi để chấp nhận trả lại.



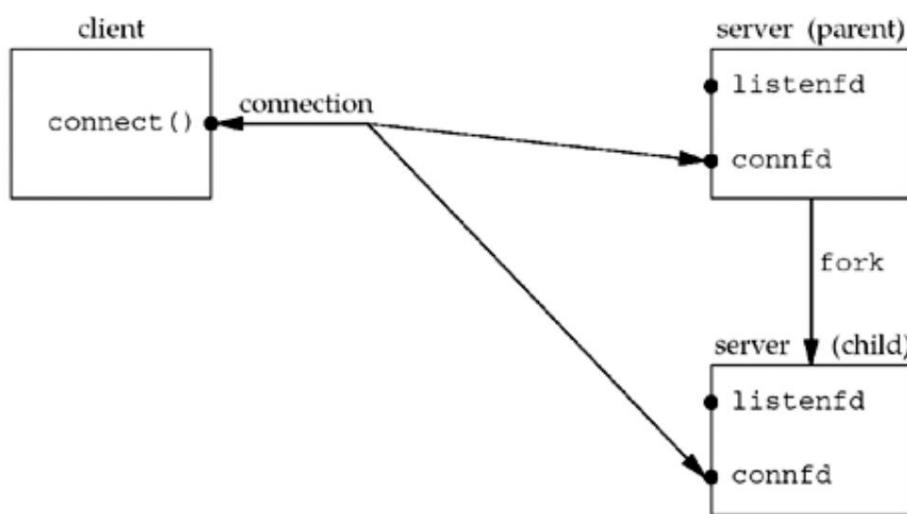
Ngay sau khi **chấp nhận** trả lại, chúng ta có kịch bản như trong [Hình 4.15](#). Kết nối được kernel **chấp nhận** và một socket mới, **connfd**, được tạo. Đây là một ô cắm được kết nối và giờ đây dữ liệu có thể được đọc và ghi qua kết nối.

Hình 4.15. Trạng thái của máy khách/máy chủ sau khi trả về từ chấp nhận.



Bú ớc tiếp theo trong máy chủ đồng thời là gọi `fork`. [Hình 4.16](#) thể hiện trạng thái sau khi `ngā ba` trở lại.

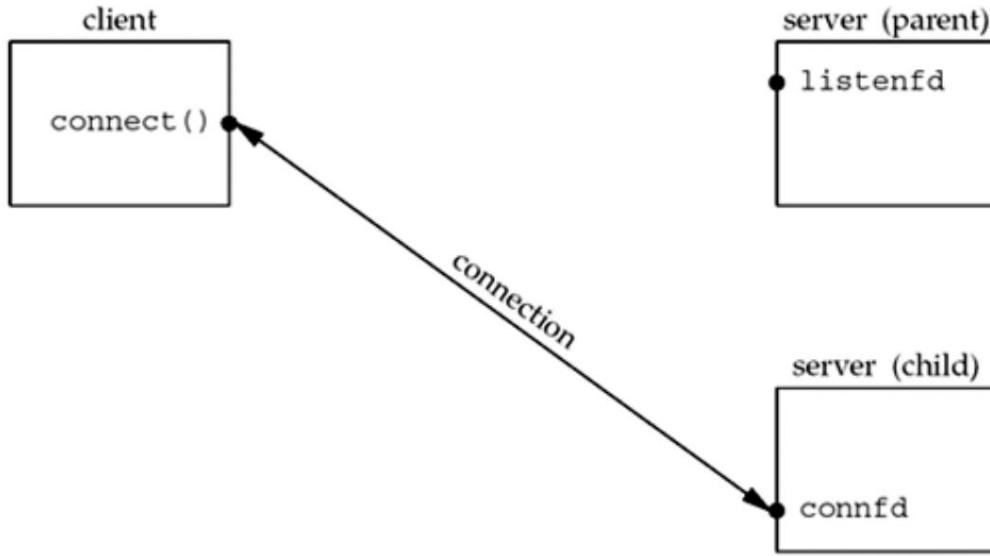
Hình 4.16. Trạng thái của máy khách/máy chủ sau khi fork quay trở lại.



Lưu ý rằng cả hai bộ mô tả, `listenfd` và `connfd`, đều được chia sẻ (trùng lặp) giữa cha và con.

Bú ớc tiếp theo là cha mẹ đóng ở cắm được kết nối và trẻ đóng ở cắm nghe. Điều này được thể hiện trong [Hình 4.17](#).

Hình 4.17. Trạng thái của máy khách/máy chủ sau khi cha và con đóng các ồ cắm thích hợp.



Đây là trạng thái cuối cùng mong muốn của ô cảm. Trẻ đang xử lý kết nối với máy khách và cấp độ gốc có thể gọi lại [chấp nhận](#) trên ô cảm nghe để xử lý kết nối máy khách tiếp theo.

4.9 Chức năng 'đóng'

Chức năng [đóng](#) Unix bình thường cũng được sử dụng để đóng socket và chấm dứt TCP sự liên quan.

```

#include <unistd.h>

int đóng (int sockfd);

Trả về: 0 nếu OK, -1 nếu có lỗi
  
```

Hành động mặc định của [việc đóng](#) bằng ô cảm TCP là đánh dấu ô cảm là đã đóng và quay trở lại quá trình ngay lập tức. Quá trình này không thể sử dụng bộ mô tả socket nữa: Nó không thể được sử dụng làm đối số để [đọc](#) hoặc [ghi](#). Tuy nhiên, TCP sẽ cố gắng gửi bất kỳ dữ liệu nào đã được xếp hàng đợi để gửi đến đầu bên kia và sau khi điều này xảy ra, trình tự chấm dứt kết nối TCP bình thường sẽ diễn ra ([Phần 2.6](#)).

Trong [Phần 7.5](#), chúng tôi sẽ mô tả tùy chọn ô cảm [SO_LINGER](#), tùy chọn này cho phép chúng tôi thay đổi hành động mặc định này bằng ô cảm TCP. Trong phần đó, chúng tôi cũng sẽ mô tả những gì ứng dụng TCP phải làm để đảm bảo rằng ứng dụng ngang hàng đã nhận được bất kỳ dữ liệu chưa thanh toán nào.

Số lư ợng tham chiếu mô tả

Ở cuối [Phần 4.8](#), chúng tôi đã đề cập rằng khi quy trình mẹ trong máy chủ đồng thời của chúng tôi đóng ở cắm được kết nối, điều này chỉ làm giảm số lượng tham chiếu cho bộ mô tả. Vì số lượng tham chiếu vẫn lớn hơn 0 nên lệnh gọi đóng này không bắt đầu trình tự chấm dứt kết nối bốn gói của TCP. Đây là hành vi mà chúng tôi muốn với máy chủ đồng thời của mình có ở cắm được kết nối được chia sẻ

giữa cha mẹ và con cái.

Nếu chúng ta thực sự muốn gửi FIN trên kết nối TCP, có thể sử dụng chức năng tắt máy ([Phần 6.6](#)) thay vì đóng. Chúng tôi sẽ mô tả động lực cho việc này trong [Phần 6.5](#).

Chúng ta cũng phải biết điều gì sẽ xảy ra trong máy chủ đồng thời của mình nếu máy chủ gốc không gọi close đối với mỗi socket được kết nối được trả về bởi chấp nhận. Đầu tiên, tiến trình gốc cuối cùng sẽ hết bộ mô tả, vì thường có giới hạn về số lượng bộ mô tả mà bất kỳ tiến trình nào cũng có thể mở bất kỳ lúc nào. Nhưng quan trọng hơn, không ai của các kết nối máy khách sẽ bị chấm dứt. Khi trẻ đóng kết nối socket, số tham chiếu của nó sẽ tăng từ 2 lên 1 và nó sẽ duy trì ở mức 1 vì cha mẹ không bao giờ đóng ở cắm được kết nối. Điều này sẽ ngăn trình tự chấm dứt kết nối của TCP xảy ra và kết nối sẽ vẫn mở.

4.10 Hàm 'getsockname' và 'getpeername'

Hai hàm này trả về địa chỉ giao thức cục bộ được liên kết với ô cắm ([getsockname](#)) hoặc địa chỉ giao thức ngoài ngoài được liên kết với ô cắm ([getpeername](#)).

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

int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);
```

Cả hai đều trả về: 0 nếu OK, -1 nếu có lỗi

Lưu ý rằng đối số cuối cùng cho cả hai hàm là đối số kết quả giá trị. Nghĩa là, cả hai hàm đều điền vào cấu trúc địa chỉ ô cắm được trả bởi localaddr hoặc ngang hàng.

Chúng tôi đã đề cập trong cuộc thảo luận về ràng buộc rằng thuật ngữ "tên" gây hiểu nhầm. Hai hàm này trả về địa chỉ giao thức được liên kết với một trong hai đầu của kết nối mạng, đối với IPV4 và IPV6 là sự kết hợp giữa địa chỉ IP và số cổng. Những chức năng này không quan gì đến tên miền ([Chương 11](#)).

Hai chức năng này là cần thiết vì những lý do sau:

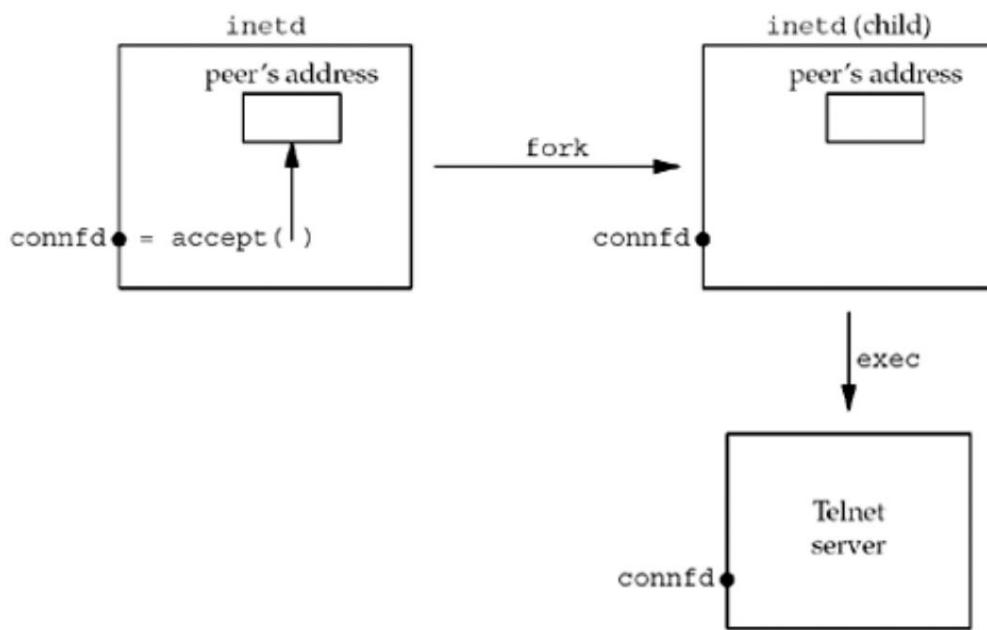
- Sau khi **kết nối** trở lại thành công trong máy khách TCP không gọi **liên kết**, **getsockname** trả về địa chỉ IP cục bộ và số cổng cục bộ được hạt nhân gán cho kết nối.
- Sau khi gọi **liên kết** với số cổng là 0 (yêu cầu kernel chọn số cổng cục bộ), **getsockname** trả về số cổng cục bộ đã được gán.
- getsockname** có thể được gọi để lấy họ địa chỉ của socket, như chúng tôi trình bày trong [Hình 4.19](#).

- Trong máy chủ TCP **liên kết** địa chỉ IP ký tự đại diện ([Hình 1.9](#)), một khi kết nối được thiết lập với máy khách (**chấp nhận** trả về thành công), máy chủ có thể gọi **getsockname** để lấy địa chỉ IP cục bộ được gán cho kết nối. Đôi số mô tả ô cắm trong lệnh gọi này phải là đôi số của ô cắm được kết nối chứ không phải ô cắm nghe.

- Khi một máy chủ được **thực thi** bởi quá trình gọi **chấp nhận**, cách duy nhất máy chủ có thể lấy được danh tính của máy khách bằng cách gọi **getpeername**. Đây là điều xảy ra bất cứ khi nào **inetd** ([Phần 13.5](#)) phân nhánh và **thực thi** một máy chủ TCP.
- [Hình 4.18](#) cho thấy kịch bản này. Cuộc gọi **inetd chấp nhận** (hộp trên cùng bên trái) và hai giá trị được trả về: bộ mô tả ô cắm được kết nối, **connfd**, là giá trị trả về của hàm và hộp nhỏ mà chúng tôi gắn nhãn "địa chỉ ngang hàng" (cấu trúc địa chỉ ô cắm Internet) chứa địa chỉ IP và số cổng của máy khách. **fork** được gọi và một phần tử con của **inetd** được tạo. Vì trê bắt đầu bằng

một bản sao hình ảnh bộ nhớ của cha mẹ, cấu trúc địa chỉ ô cắm có sẵn cho con, cũng như bộ mô tả ô cắm được kết nối (vì các bộ mô tả được chia sẻ giữa cha mẹ và con). Nhưng khi đứa trẻ **thực thi** máy chủ thực (giả sử máy chủ Telnet mà chúng tôi hiển thị), hình ảnh bộ nhớ của đứa trẻ được thay thế bằng tệp chươn trình mới cho máy chủ Telnet (tức là cấu trúc địa chỉ socket chứa địa chỉ của máy ngang hàng bị mất), và bộ mô tả ô cắm được kết nối vẫn mở trên toàn bộ **tệp thực thi**. Một trong những lệnh gọi hàm đầu tiên được máy chủ Telnet thực hiện là **getpeername** để lấy địa chỉ IP và số cổng của máy khách.

Hình 4.18. Ví dụ về inetd sinh ra một máy chủ.



Rõ ràng máy chủ Telnet trong ví dụ cuối cùng này phải biết giá trị của `connfd` khi nó khởi động. Có hai cách phổ biến để làm điều này. Đầu tiên, quá trình gọi `exec` có thể định dạng số mô tả dữ ờ i dạng chuỗi ký tự và chuyển nó dữ ờ i dạng đối số dòng lệnh cho chương trình mới **đư ợc thực thi**. Ngoài ra, có thể thiết lập quy ước rằng một bộ mô tả nhất định luôn được đặt cho ô cảm được kết nối trước khi gọi `exec`. Cái sau là những gì `inetd` thực hiện, luôn đặt các bộ mô tả 0, 1 và 2 làm ô cảm được kết nối.

Ví dụ: Lấy họ địa chỉ của một socket

Hàm `sockfd_to_family` được hiển thị trong [Hình 4.19](#) trả về họ địa chỉ của ô cảm.

[Hình 4.19](#) Trả về họ địa chỉ của socket.

`lib/sockfd_to_family.c`

```

1 #include "unp.h"
2 số nguyên
3 sockfd_to_family(int sockfd)
4 {
5     struct sockaddr_storage ss;
6     socklen_t len;
7     len = sizeof(ss);
8     if (getsockname(sockfd, (SA *) &ss, &len) < 0)
9         trả lại (-1);

```

```
10      trả về (ss.ss_family);
11 }
```

Phân bổ phòng cho cấu trúc địa chỉ ở cắm lớn nhất

5 Vì chúng tôi không biết nên phân bổ loại cấu trúc địa chỉ ở cắm nào nên chúng tôi sử dụng giá trị `sockaddr_storage` vì nó có thể chứa bất kỳ cấu trúc địa chỉ ở cắm nào được hệ thống hỗ trợ.

Gọi `getockname`

7-10 Chúng tôi gọi `getockname` và trả về họ địa chỉ.

Vì đặc tả POSIX cho phép lệnh gọi tới `getockname` trên một ô cắm không liên kết nên hàm này sẽ hoạt động với mọi bộ mô tả ô cắm mở.

4.11 Tóm tắt

Tất cả máy khách và máy chủ đều bắt đầu bằng lệnh gọi tới ô cám, trả về bộ mô tả ô cám.

Sau đó, khách hàng gọi `kết nối`, trong khi máy chủ gọi `liên kết, nghe` và `chấp nhận`. Các ô cám thường được đóng bằng chức năng `đóng` tiêu chuẩn, mặc dù chúng ta sẽ thấy một cách khác để thực hiện điều này với chức năng `tắt máy` (Phần 6.6) và chúng ta cũng sẽ kiểm tra tác động của `tùy chọn ô cám SOLINGER` (Phần 7.5).

Hầu hết các máy chủ TCP đều hoạt động đồng thời, với máy chủ gọi `nhánh` cho mọi kết nối máy khách mà nó xử lý. Chúng ta sẽ thấy rằng hầu hết các máy chủ UDP đều có tính lặp lại. Trong khi hai mô hình này đã được sử dụng thành công trong nhiều năm, trong Chương 30 chúng ta sẽ xem xét các tùy chọn thiết kế máy chủ khác sử dụng các luồng và quy trình.

Bài tập

4.1 Trong Phần 4.4, chúng tôi đã nêu rằng hằng số `INADDR` được xác định bởi

Tiêu đề `<netinet/in.h>` theo thứ tự byte máy chủ. Làm thế nào chúng ta có thể nói điều này?

4.2 Sửa đổi Hình 1.5 để gọi `getockname` sau khi `kết nối` trả về thành công. In địa chỉ IP cục bộ và cổng cục bộ được gán cho ô cám TCP bằng `sock_ntop`.

Các cổng tạm thời trong hệ thống của bạn nằm trong phạm vi nào (Hình 2.10) ?

4.3 Trong máy chủ đồng thời, giả sử máy chủ con chạy trู ớc sau lệnh gọi `fork`. Các sau đó con sẽ hoàn thành dịch vụ của máy khách trู ớc khi lệnh gọi `fork` quay trở lại cha mẹ. Điều gì xảy ra trong hai lệnh gọi `đóng` trong Hình 4.13?

4.4 Trong [Hình 4.11](#), trư ớc tiên hãy thay đổi cổng của máy chủ từ 13 thành 9999 (để chúng ta không cần đặc quyền siêu người dùng để khởi động chương trình). Hủy bỏ cuộc gọi để lắng nghe. Điều gì xảy ra?

4.5 Tiếp tục bài tập trư ớc. Xóa cuộc gọi để [liên kết](#) như ng cho phép cuộc gọi đến [Nghe](#). Điều gì xảy ra?

Chương 5. Ví dụ về máy khách/máy chủ TCP

[Mục 5.1. Giới thiệu](#)

[Mục 5.2. Máy chủ TCP Echo: Chức năng chính](#)

[Mục 5.3. Máy chủ TCP Echo: Chức năng str_echo](#)

[Mục 5.4. TCP Echo Client: Chức năng chính](#)

[Mục 5.5. Máy khách TCP Echo: Hàm str_cli](#)

[Mục 5.6. Khởi động bình thường](#)

[Mục 5.7. Chấm dứt bình thường](#)

[Mục 5.8. Xử lý tín hiệu POSIX](#)

[Mục 5.9. Xử lý tín hiệu SIGCHLD](#)

[Mục 5.10. chờ đợi và chức năng waitpid](#)

[Mục 5.11. Hủy kết nối trư ớc khi chấp nhận Trả về](#)

[Mục 5.12. Chấm dứt quá trình máy chủ](#)

[Mục 5.13. Tín hiệu SIGPIPE](#)

[Mục 5.14. Sư cố máy chủ/máy chủ](#)

[Mục 5.15. Sư cố và khởi động lại máy chủ/máy chủ](#)

[Mục 5.16. Tắt máy chủ/máy chủ](#)

[Mục 5.17. Tóm tắt ví dụ về TCP](#)

[Mục 5.18. Định dạng dữ liệu](#)

Mục 5.19. Bản tóm tắtBài tập

5.1 Giới thiệu

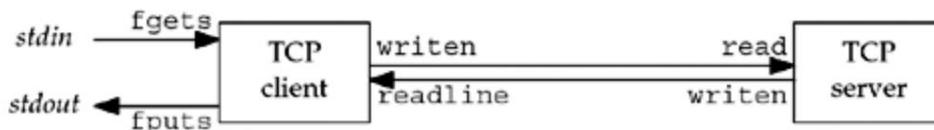
Bây giờ chúng ta sẽ sử dụng các hàm cơ bản từ chương truy cập để viết một ví dụ máy khách/máy chủ TCP hoàn chỉnh.

Ví dụ đơn giản của chúng tôi là một máy chủ echo thực hiện các bước sau:

1. Máy khách đọc một dòng văn bản từ đầu vào tiêu chuẩn của nó và ghi dòng đó vào máy chủ.
2. Máy chủ đọc đường truyền từ đầu vào mạng của nó và phản hồi đường truyền đó trở lại khách hàng.
3. Máy khách đọc dòng phản hồi và in nó ra đầu ra tiêu chuẩn.

Hình 5.1 mô tả máy khách/máy chủ đơn giản này cùng với các chức năng được sử dụng cho đầu vào và đầu ra.

Hình 5.1. Máy khách và máy chủ echo đơn giản.



Chúng tôi hiển thị hai mũi tên giữa máy khách và máy chủ, nhưng đây thực sự là một kết nối TCP song công hoàn toàn. Các hàm `fget` và `fputs` lấy từ thư viện I/O tiêu chuẩn và các hàm `ghi` và `đọc dòng` được trình bày trong [Phần 3.9](#).

Mặc dù chúng tôi sẽ phát triển cách triển khai máy chủ echo của riêng mình, nhưng hầu hết việc triển khai TCP/IP đều cung cấp một máy chủ như vậy, sử dụng cả TCP và UDP ([Phần 2.12](#)). Chúng tôi cũng sẽ sử dụng [máy chủ này](#) với máy khách của riêng mình.

Một máy khách/máy chủ lặp lại các dòng đầu vào là một ví dụ hợp lệ nhưng đơn giản về ứng dụng mạng. Tất cả các bước cơ bản cần thiết để triển khai bất kỳ máy khách/máy chủ nào đều được minh họa bằng ví dụ này. Để mở rộng ví dụ này vào ứng dụng của riêng bạn, tất cả những gì bạn cần làm là thay đổi những gì máy chủ thực hiện với thông tin đầu vào mà nó nhận được từ các máy khách.

Bên cạnh việc chạy máy khách và máy chủ của chúng tôi ở chế độ bình thường (nhập một dòng và xem tiếng vang), chúng tôi kiểm tra rất nhiều điều kiện biên cho ví dụ này: điều gì xảy ra khi máy khách và máy chủ được khởi động; điều gì xảy ra khi khách hàng chấm dứt bình thường; điều gì sẽ xảy ra với máy khách nếu quá trình máy chủ kết thúc trước khi máy khách hoàn tất; điều gì sẽ xảy ra với máy khách nếu máy chủ lưu trữ gấp sáu lần; và như thế. Bằng cách tìm kiếm ở tất cả các tình huống này và hiểu được điều gì xảy ra ở cấp độ mạng cũng như cách thức

Điều này xuất hiện trong API socket, chúng ta sẽ hiểu thêm về những gì diễn ra ở các cấp độ này và cách viết mã ứng dụng của chúng ta để xử lý các tình huống này.

Trong tất cả các ví dụ này, chúng tôi có các hằng số dành riêng cho giao thức "được mã hóa cứng" như địa chỉ và cổng. Có hai lý do cho việc này. Đầu tiên, chúng ta phải hiểu chính xác những gì cần được lưu trữ trong cấu trúc địa chỉ dành riêng cho giao thức. Thứ hai, chúng tôi chưa đề cập đến các chức năng thư viện có thể làm cho tính năng này dễ mang theo hơn.

Các chức năng này sẽ được đề cập trong [Chương 11](#).

Bây giờ chúng ta lưu ý rằng chúng ta sẽ thực hiện nhiều thay đổi đối với cả máy khách và máy chủ trong các chương tiếp theo khi chúng ta tìm hiểu thêm về lập trình mạng ([Hình 1.12](#) và [1.13](#)).

5.2 TCP Echo Server: Chức năng 'chính'

Máy khách và máy chủ TCP của chúng tôi tuân theo dòng chức năng mà chúng tôi đã sơ đồ hóa trong [Hình 4.1](#).

Chúng tôi hiển thị chương trình máy chủ đồng thời trong [Hình 5.2](#).

Tạo socket, liên kết cổng nối tiếng của máy chủ

[9-15](#) Ở cắm TCP được tạo. Cấu trúc địa chỉ ở cắm Internet được diễn bằng

địa chỉ ký tự đại diện (`INADDR_ANY`) và cổng phổ biến của máy chủ (`SERV_PORT`, được xác định là 9877 trong tiêu đề `unp.h` của chúng tôi). Việc liên kết địa chỉ ký tự đại diện cho hệ thống biết rằng chúng tôi sẽ chấp nhận kết nối dành cho bất kỳ giao diện cục bộ nào, trong trường hợp hệ thống là nhiều nhà. Sự lựa chọn số cổng TCP của chúng tôi dựa trên [Hình 2.10](#). Nó phải lớn hơn 1023 (chúng tôi không cần cổng dành riêng), lớn hơn 5000 (để tránh xung đột với các cổng tạm thời được phân bổ bởi nhiều triển khai có nguồn gốc từ Berkeley), nhỏ hơn 49152 (để tránh xung đột với phạm vi "chính xác" của cổng tạm thời) và nó không được xung đột với bất kỳ cổng đã đăng ký nào. Ở cắm được chuyển đổi thành ở cắm nghe bằng cách [lắng nghe](#).

Đợi kết nối máy khách hoàn tất

[17-18](#) Máy chủ chặn cuộc gọi để chấp nhận, chờ kết nối máy khách hoàn tất.

Máy chủ đồng thời

[19-24](#) Đối với mỗi khách hàng, `fork` sinh ra một con và con đó xử lý khách hàng mới. Như chúng ta đã thảo luận ở [Phần 4.8](#), trẻ đóng ở cắm nghe và phụ huynh đóng ở cắm được kết nối. Sau đó đưa trẻ gọi `str_echo` ([Hình 5.3](#)) để xử lý máy khách.

[Hình 5.2](#) Máy chủ tiếng vang TCP (được cải thiện trong [Hình 5.12](#)).

`tcpdisserv/tcpser01.c`

```

1 #bao gồm          "unp.h"

2 số nguyên

3 chính(int argc, char **argv)
4 {
5     int      listenfd, connfd;
6     pid_t   trẻ con;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;

9     listenfd = Ô cắm (AF_INET, SOCK_STREAM, 0);

10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
13    servaddr.sin_port = htons (SERV_PORT);

14    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

15    Nghe(listenfd, LISTENQ);

16    vì (    ;    )  {
17        clilen = sizeof(cliaddr);
18        connfd = Chấp nhận(listenfd, (SA *) &cliaddr, &clilen);

19        if ( (childpid = Fork()) == 0) { /* tiến trình con */
20            Đóng(nghefd);           /* đóng socket nghe */
21            str_echo(connfd); /* xử lý yêu cầu */
22            thoát (0);
23        }
24        Đóng(connfd);           /* cha mẹ đóng socket đã kết nối */
25    }
26 }
```

5.3 Máy chủ TCP Echo: Chức năng 'str_echo'

Hàm `str_echo`, được hiển thị trong [Hình 5.3](#), thực hiện quá trình xử lý của máy chủ cho mỗi client: Nó đọc dữ liệu từ client và gửi lại cho client.

Đọc bộ đệm và lặp lại bộ đệm

8-9 **đọc** đọc dữ liệu từ ô cắm và dòng đưa ra phản hồi trả lại máy khách bằng cách **ghi**. Nếu máy khách đóng kết nối (trường hợp bình thường), việc nhận FIN của máy khách sẽ khiến số **lần đọc** của máy khách trả về 0. Điều này khiến hàm **str_echo** trả về, kết thúc máy khách trong [Hình 5.2](#)

5.4 TCP Echo Client: Chức năng 'chính'

[Hình 5.4](#) thể hiện chức năng **chính** của máy khách TCP.

Hình 5.3 Hàm **str_echo**: phản hồi dữ liệu trên socket.

lib/str_echo.c

```

1 #include "unp.h"

2 khoảng trắng

3 str_echo(int sockfd)
4 {
5     cõ_t n;
6     char buf[MAXLINE];

7 lần nữa:
8     trong khi ( (n = read(sockfd, buf, MAXLINE)) > 0)
9         Đã viết(sockfd, buf, n);

10    if (n < 0 && errno == EINTR)
11        đi lại lần nữa;
12    ngược lại nếu (n < 0)
13        err_sys("str_echo: lỗi đọc");
14 }
```

Hình 5.4 Máy khách TCP echo.

tcpcliserv/tcpli01.c

```

1 #include "unp.h"

2 số nguyên

3 chính(int argc, char **argv)
4 {
5     int      sockfd;
6     struct sockaddr_in servaddr;
```

```

7     nếu (argc != 2)
        err_quit("cách sử dụng: tcpcli <IPaddress>");

9     sockfd = Ở cắm(AF_INET, SOCK_STREAM, 0);

10    bzero(&servaddr, sizeof(servaddr));
11    servaddr.sin_family = AF_INET;
12    servaddr.sin_port = htons(SERV_PORT);
13    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

14    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

15    str_cli(stdin, sockfd);           /* làm tắt cả */

16    thoát (0);
17 }

```

Tạo socket, điền cấu trúc địa chỉ socket Internet

9-13 Ở cắm TCP được tạo và cấu trúc địa chỉ ở cắm Internet được điền vào địa chỉ IP và số cổng của máy chủ. Chúng tôi lấy địa chỉ IP của máy chủ từ đối số dòng lệnh và cổng nối tiếng của máy chủ (SERV_PORT) là từ tiêu đề `unp.h` của chúng tôi.

Kết nối với máy chủ

Kết nối 14-15 thiết lập kết nối với máy chủ. Hàm `str_cli` ([Hình 5.5](#)) xử lý phần còn lại của quá trình xử lý máy khách.

5.5 Máy khách TCP Echo: Chức năng 'str_cli'

Hàm này, như trong [Hình 5.5](#), xử lý vòng lặp xử lý máy khách: Nó đọc một dòng văn bản từ đầu vào tiêu chuẩn, ghi nó vào máy chủ, đọc lại dòng phản hồi của máy chủ và xuất dòng văn bản phản hồi sang đầu ra tiêu chuẩn.

[Hình 5.5](#) Hàm `str_cli`: vòng lặp xử lý máy khách.

`lib/str_cli.c`

```
1 #include "unp.h"
```

2 khoảng trắng

```

3 str_cli(TÂP_TIN *fp, int sockfd)
4 {
5     char sendline[MAXLINE], recvline[MAXLINE];
6
7     while (Fgets(sendline, MAXLINE, fp) != NULL) {
8
9         if (Readline(sockfd, recvline, MAXLINE) == 0)
10            err_quit("str_cli: máy chủ đã kết thúc sớm");
11
12 }

```

Đọc một dòng, ghi vào máy chủ

6-7 fgets đọc một dòng văn bản và viết gửi dòng đó đến máy chủ.

Đọc dòng phản hồi từ máy chủ, ghi vào đầu ra tiêu chuẩn

Để ờng đọc 8-10 đọc dòng đư ợc phản hồi từ máy chủ và fputs ghi nó vào đầu ra tiêu chuẩn.

Quay trở lại vấn đề chính

11-12 Vòng lặp kết thúc khi fgets trả về một con trỏ null, xảy ra khi nó gặp lỗi cuối tệp (EOF) hoặc lỗi. Hàm bao bọc Fgets của chúng tôi kiểm tra lỗi và hủy bỏ nếu xảy ra, do đó Fgets chỉ trả về một con trỏ null khi một gặp phải phần cuối của tập tin.

5.6 Khởi động bình thư ờng

Mặc dù ví dụ về TCP của chúng tôi rất nhỏ (khoảng 150 dòng mã cho hai hàm chính str_echo, str_cli, readline và write), nhưng điều cần thiết là chúng tôi phải hiểu cách máy khách và máy chủ bắt đầu, cách chúng kết thúc và quan trọng nhất là điều gì sẽ xảy ra. xảy ra khi có sự cố xảy ra: máy chủ máy khách gặp sự cố, quy trình máy khách gặp sự cố, kết nối mạng bị mất, v.v. Chỉ bằng cách hiểu rõ các điều kiện biên này và sự tương tác của chúng với các giao thức TCP/IP, chúng ta mới có thể viết đư ợc các máy khách và máy chủ mạnh mẽ có thể xử lý các điều kiện này.

Trước tiên, chúng tôi khởi động máy chủ ở chế độ nền trên máy chủ linux.

```
linux% tcpserv01 &
[1] 17870
```

Khi máy chủ khởi động, nó gọi **socket**, **liên kết**, **nghe** và **chấp nhận**, chặn cuộc gọi **chấp nhận**. (Chúng tôi chưa khởi động máy khách.) Trước khi khởi động máy khách, chúng tôi chạy chương trình **netstat** để xác minh trạng thái ở cắm nghe của máy chủ.

```
linux% netstat -a
```

Kết nối Internet đang hoạt động (máy chủ và đã thiết lập)				Tình trạng	
Proto	Recv-Q	Send-Q	Địa chỉ cục bộ		địa chỉ nút ngoài
tcp	0	0	*:9877	*:*	NGHE

Ở đây chúng tôi chỉ hiển thị dòng đầu ra đầu tiên (tiêu đề), cộng với dòng mà chúng tôi quan tâm. Lệnh này hiển thị trạng thái của tất cả các ỗ cắm trên hệ thống, có thể có nhiều đầu ra. Chúng ta phải chỉ định cờ **-a** để xem các ỗ cắm nghe.

Đầu ra là những gì chúng tôi mong đợi. Ở cắm ở trạng thái LISTEN với ký tự đại diện cho địa chỉ IP cục bộ và cổng cục bộ là 9877. **netstat** in dấu hoa thị cho địa chỉ IP 0 (**INADDR_ANY**, ký tự đại diện) hoặc cho cổng 0.

Sau đó, chúng tôi khởi động máy khách trên cùng một máy chủ, chỉ định địa chỉ IP của máy chủ là 127.0.0.1 (địa chỉ vòng lặp ngược). Chúng tôi cũng có thể chỉ định địa chỉ IP bình thường (không lặp lại) của máy chủ.

```
linux% tcpcli01 127.0.0.1
```

Máy khách gọi **socket** và **connect**, kết nối sau sẽ thực hiện quá trình bắt tay ba chiều của TCP. Khi quá trình bắt tay ba chiều hoàn tất, **kết nối** trả về trong máy khách và **chấp nhận** trả lại trong máy chủ. Kết nối được thiết lập. Các bước sau đây sẽ diễn ra:

1. Máy khách gọi **str_cli**, điều này sẽ chặn lệnh gọi tới **fgets**, vì chúng ta có chưa gõ một dòng đầu vào.

2. Khi chấp nhận trả về trong máy chủ, nó gọi `fork` và đưa trè gọi `str_echo`.

Hàm này gọi `readline`, gọi `read`, chặn trong khi chờ một dòng được gửi từ máy khách.

3. Mất khác, máy chủ gốc chấp nhận lại cuộc gọi và chặn trong khi chờ kết nối máy khách tiếp theo.

Chúng tôi có ba quy trình và cả ba quy trình đều ở chế độ ngủ (bị chặn): máy khách, máy chủ gốc và máy chủ con.

Khi quá trình bắt tay ba chiêu hoàn tất, chúng tôi có tình liết kê bù ợc của máy khách trư ợc và sau đó là các bù ợc của máy chủ. Có thể thấy lý do trong [Hình 2.5](#): kết nối trả về khi máy khách nhận được phân đoạn bắt tay thứ hai, nhưng chấp nhận không trả về cho đến khi phân đoạn bắt tay thứ ba được máy chủ nhận được, một nữa RTT sau khi kết nối trở lại.

Chúng tôi có tình chạy máy khách và máy chủ trên cùng một máy chủ vì đây là cách dễ nhất để thử nghiệm các ứng dụng máy khách/máy chủ. Vì chúng ta đang chạy máy khách và máy chủ trên cùng một máy chủ nên `netstat` hiện hiển thị hai dòng đầu ra bổ sung, tương ứng với kết nối TCP:

```
linux% netstat -a
```

Kết nối Internet đang hoạt động (máy chủ và đã thiết lập)

Proto	Recv-Q	Send-Q	Địa chỉ cục bộ	Địa chỉ ngoài	Tình trạng
tcp	0	0	máy chủ cục bộ:9877	máy chủ cục bộ: 42758	
THÀNH LẬP					
tcp	0	0	máy chủ cục bộ:42758	máy chủ cục bộ:9877	
THÀNH LẬP					
tcp	0	0	*:9877	*	NGHE

Dòng đầu tiên trong số ESTABLISHED tương ứng với socket của máy chủ con, vì cổng cục bộ là 9877. Dòng thứ hai trong số ESTABLISHED là ô cảm của máy khách, vì cổng cục bộ là 42758. Nếu chúng ta đang chạy máy khách và máy chủ trên các máy chủ khác nhau, máy chủ máy khách sẽ chỉ hiển thị ô cảm của máy khách và máy chủ máy chủ sẽ chỉ hiển thị hai ô cảm máy chủ.

Chúng ta cũng có thể sử dụng lệnh `ps` để kiểm tra trạng thái và mối quan hệ của những quá trình.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
      PID PPID TT          LỆNH STAT          WCHAN
22038 22036 diễm/6 giây -bash 17870 22038 diễm/
6 giây           ./tcpserv01       chờ_for_connect
19315 17870 diễm/6S       ./tcpserv01       tcp_data_wait
19314 22038 diễm/6 giây       ./tcpcli01 127.0 read_chan
```

(Chúng tôi đã sử dụng các đối số rất cụ thể cho `ps` để chỉ hiển thị cho chúng tôi thông tin liên quan đến cuộc thảo luận này.) Trong kết quả đầu ra này, chúng tôi đã chạy máy khách và máy chủ từ cùng một cửa sổ (`pts/6`, viết tắt của thiết bị đầu cuối giả số 6) . Các cột PID và PPID hiển thị mối quan hệ cha và con. Chúng ta có thể nói rằng `tcpserv01` đầu tiên dòng là cha và dòng `tcpserv01` thứ hai là con vì PPID của con là PID của cha. Ngoài ra, PPID của cha mẹ là shell (`bash`).

Cột STAT cho cả ba quy trình mạng của chúng tôi là "S", nghĩa là quy trình đang ở chế độ ngủ (đang chờ thứ gì đó). Khi một tiến trình đang ngủ, cột WCHAN chỉ định điều kiện. Linux in `wait_for_connect` khi một quy trình bị chặn ở chế độ chấp nhận hoặc kết nối, `tcp_data_wait` khi một quy trình bị chặn trên đầu vào hoặc đầu ra của ống cắm hoặc `read_chan` khi một quy trình bị chặn trên I/O đầu cuối. Do đó, các giá trị WCHAN cho ba quy trình mạng của chúng tôi có ý nghĩa.

5.7 Chấm dứt thông thư ờng

Tại thời điểm này, kết nối được thiết lập và bắt cứ điều gì chúng ta gõ vào máy khách đều được vọng lại.

linux% tcpcli01 127.0.0.1	chúng tôi đã hiển thị dòng này trước đó
Chào thế giới	bây giờ chúng ta gõ cái này
Chào thế giới	và dòng được lặp lại
tạm biệt	
tạm biệt	
^D	Control-D là ký tự EOF cuối cùng của chúng tôi

Chúng tôi nhập hai dòng, mỗi dòng được lặp lại và sau đó chúng tôi nhập ký tự EOF cuối cùng (Control-D), ký tự này sẽ kết thúc ứng dụng khách. Nếu chúng ta thực thi ngay `netstat`, chúng ta có

```
linux% netstat -a | grep 9877
tcp        0      0 *:9877                           *:*                               NGHE
tcp        0      0 máy chủ cục bộ:42758           localhost:9877 TIME_WAIT
```

Phía kết nối của máy khách (vì cổng cục bộ là 42758) chuyển sang trạng thái TIME_WAIT ([Phần 2.7](#)) và máy chủ vẫn đang chờ kết nối máy khách khác. (Lần này chúng tôi chuyển đầu ra của `netstat` thành `grep`, chỉ in các dòng có cổng phổ biến của máy chủ của chúng tôi. Làm như vậy cũng sẽ xóa dòng tiêu đề.)

Chúng tôi có thể làm theo các bước liên quan đến việc chấm dứt hợp đồng thông thư ờng với khách hàng của chúng tôi và máy chủ:

1. Khi chúng ta gõ ký tự EOF, `fgets` trả về một con trỏ null và hàm `str_cli` ([Hình 5.5](#)) trả về.
 2. Khi `str_cli` quay lại chức năng `chính` của máy khách ([Hình 5.4](#)), `chức năng sau` kết thúc bằng cách gọi `exit`.
 3. Một phần của việc chấm dứt quá trình là việc đóng tất cả các bộ mô tả đang mở, do đó ở cẩm máy khách được đóng bởi hạt nhân. Điều này sẽ gửi FIN đến máy chủ và máy chủ TCP sẽ phản hồi bằng ACK. Đây là nửa đầu của chuỗi kết thúc kết nối TCP. Tại thời điểm này, socket máy chủ ở trạng thái CLOSE_WAIT và socket máy khách ở trạng thái FIN_WAIT_2 ([Hình 2.4](#) và [2.5](#)).
 4. Khi máy chủ TCP nhận được FIN, máy chủ con sẽ bị chặn trong cuộc gọi tới `readline` ([Hình 5.3](#)), và `readline` sau đó trả về 0. Điều này khiến hàm `str_echo` quay trở lại main của máy chủ.
 5. Máy chủ con kết thúc bằng cách gọi `exit` ([Hình 5.2](#)).
 6. Tất cả các bộ mô tả mở trong máy chủ con đều bị đóng. Việc trẻ đóng ở cẩm được kết nối sẽ khiến hai phân đoạn cuối cùng của việc chấm dứt kết nối TCP diễn ra: FIN từ máy chủ đến máy khách và ACK từ máy khách ([Hình 2.5](#)). Tại thời điểm này, kết nối bị chấm dứt hoàn toàn.
- Ở cẩm máy khách chuyển sang trạng thái TIME_WAIT.
7. Cuối cùng, tín hiệu `SIGCHLD` được gửi đến máy chủ cha khi máy chủ con chấm dứt. Điều này xảy ra trong ví dụ này, như ng chúng tôi không bắt được tín hiệu trong mã của mình và hành động mặc định của tín hiệu sẽ bị bỏ qua. Vì vậy, đứa trẻ rơi vào trạng thái zombie. Chúng ta có thể xác minh điều này bằng lệnh `ps`.

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
PID PPID TT          LỆNH STAT          WCHAN
22038 22036 điem/6 S -bash                đọc_chan
17870 22038 điem/6 giây       ./tcpser01    chờ_for_connect
19315 17870 điem/6 Z          [tcpser01 <defu do_exit]
```

STAT của đứa trẻ bây giờ là `Z` (đối với zombie).

Chúng tôi cần dọn dẹp các quy trình zombie của mình và thực hiện việc này đòi hỏi phải xử lý các tín hiệu Unix. Trong phần tiếp theo, chúng tôi sẽ trình bày tổng quan về xử lý tín hiệu.

5.8 Xử lý tín hiệu POSIX

Tín hiệu là một thông báo cho một quá trình rằng một sự kiện đã xảy ra. Tín hiệu đến khi được gọi là ngắt phần mềm. Tín hiệu thường xảy ra không đồng bộ. Bằng cách này, chúng tôi muốn nói rằng một quá trình không biết trước xác chính xác khi nào tín hiệu sẽ xuất hiện.

Tín hiệu có thể được gửi

- Từ một tiến trình tới một tiến trình khác (hoặc tới chính nó)
- Bằng kernel tới một tiến trình

Tín hiệu `SIGCHLD` mà chúng tôi đã mô tả ở cuối phần trước là tín hiệu được hạt nhân gửi bất cứ khi nào một quá trình kết thúc, tới tiến trình mẹ của quá trình kết thúc. quá trình.

Mỗi tín hiệu đều có một cách xử lý, còn được gọi là hành động liên quan đến tín hiệu. Chúng tôi thiết lập cách sắp xếp tín hiệu bằng cách gọi hàm `sigaction` (được mô tả ngắn gọn) và chúng tôi có ba lựa chọn cho cách sắp xếp:

1. Chúng tôi có thể cung cấp một hàm được gọi bất cứ khi nào một tín hiệu cụ thể xảy ra. Chức năng này được gọi là bộ xử lý tín hiệu và hành động này được gọi là bắt tín hiệu.
Không thể bắt được hai tín hiệu `SIGKILL` và `SIGSTOP`. Hàm của chúng ta được gọi với một đối số nguyên duy nhất là số tín hiệu và hàm không trả về gì cả. Do đó, nguyên mẫu chức năng của nó là
- 2.
- 3.
- 4.
5. `xử lý void (int signo);`
- 6.

Đối với hầu hết các tín hiệu, việc gọi `sigaction` và chỉ định một hàm sẽ được gọi khi tín hiệu xuất hiện là tất cả những gì cần thiết để bắt được tín hiệu. Nhưng sau này chúng ta sẽ thấy rằng một số tín hiệu, `SIGIO`, `SIGPOLL` và `SIGURG`, đều yêu cầu các hành động bổ sung trong một phần của quy trình để bắt được tín hiệu.

7. Chúng ta có thể bỏ qua tín hiệu bằng cách đặt vị trí của nó thành `SIG_IGN`. Hai tín hiệu không thể bỏ qua `SIGKILL` và `SIGSTOP`.

8. Chúng ta có thể đặt cách sắp xếp mặc định cho tín hiệu bằng cách đặt cách sắp xếp của nó thành **SIG_DFL**. Mặc định thường là chấm dứt một quá trình khi nhận được tín hiệu, với một số tín hiệu nhất định cũng tạo ra hình ảnh cốt lõi của quá trình trong thư mục làm việc hiện tại của nó. Có một số tín hiệu mà cách xử lý mặc định của chúng sẽ bị bỏ qua: **SIGCHLD** và **SIGURG** (được gửi khi có dữ liệu ngoài băng tần, [Chương 24](#)) là hai tín hiệu mà chúng ta sẽ gặp trong văn bản này.

chức năng tín hiệu

Cách POSIX để thiết lập cách xử lý tín hiệu là gọi hàm **sigaction**.

Tuy nhiên, điều này trở nên phức tạp vì một đối số của hàm là một cấu trúc mà chúng ta phải phân bổ và điền vào. Một cách dễ dàng hơn để thiết lập cách sắp xếp tín hiệu là gọi hàm **tín hiệu**. Đối số đầu tiên là tên tín hiệu và đối số thứ hai là con trỏ tới hàm hoặc một trong các hằng số **SIG_IGN** hoặc **SIG_DFL**. Như ng, **tín hiệu**

là một chức năng lịch sử có trước POSIX. Các cách triển khai khác nhau cung cấp ngữ nghĩa tín hiệu khác nhau khi nó được gọi, mang lại khả năng tương thích ngược, trong khi POSIX giải thích rõ ràng ngữ nghĩa khi **sigaction** được gọi. Giải pháp là xác định hàm riêng của chúng ta có tên **tín hiệu** chỉ gọi hàm **sigaction** POSIX. Điều này cung cấp một giao diện đơn giản với ngữ nghĩa POSIX mong muốn. Chúng tôi đưa hàm này vào thư viện của riêng mình cùng với **err_XXX**

các hàm và các hàm bao bọc của chúng tôi, chẳng hạn như các hàm mà chúng tôi chỉ định khi xây dựng bất kỳ chương trình nào trong văn bản này. Hàm này được hiển thị trong [Hình 5.6](#) (hàm bao bọc tương ứng, **Tín hiệu**, không được hiển thị ở đây vì nó sẽ giống nhau cho dù được gọi là hàm của chúng tôi hay hàm **tín hiệu** do nhà cung cấp cung cấp).

Hàm tín hiệu Hình 5.6 gọi hàm **sigaction** POSIX.

lib/tín hiệu.c

```

1 #include "unp.h"

2 dấu hiệu *
Tín hiệu 3 (int signo, Sigfunc *func)
4 {
5     cấu trúc hành động sigaction, oact;

6     hành động.sa_handler = func;
7     sigemptyset (&act.sa_mask);
8     hành động.sa_flags = 0;
9     if (ký == SIGALRM) {

10 #ifdef SA_INTERRUPT
11         Act.sa_flags |= SA_INTERRUPT;                                /* SunOS 4.x */
12 #endif
13     } khác {

```

```

14 #ifdef SA_RESTART
15         Act.sa_flags |= SA_RESTART; /*SVR4, 4.4BSD */
16 #endif
17     }
18     if (sigaction (signo, &act, &oact) < 0)
19         trả lại (SIG_ERR);
20     trả về (oact.sa_handler);
21 }

```

Đơn giản hóa nguyên mẫu hàm bằng typedef

2-3 Nguyên mẫu hàm thông thường cho **tín hiệu** rất phức tạp do mức độ của các dấu ngoặc đơn lồng nhau.

```
void (*signal (int signo, void (*func) (int))) (int);
```

Để đơn giản hóa điều này, chúng tôi xác định loại **Sigfunc** trong tiêu đề **unp.h** của chúng tôi là

```
typedef void Sigfunc(int);
```

nêu rõ rằng trình xử lý tín hiệu là các hàm có đối số nguyên và hàm không trả về gì (**void**).
Nguyên mẫu hàm sau đó trở thành

```
Tín hiệu Sigfunc * (int signo, Sigfunc *func);
```

Con trả tới hàm xử lý tín hiệu là đối số thứ hai của hàm, đồng thời là giá trị trả về của hàm.

Đặt trình xử lý

6 Thành viên **sa_handler** của cấu trúc **sigaction** được đặt thành đối số **func**.

Đặt mặt nạ tín hiệu cho trình xử lý

7 POSIX cho phép chúng ta chỉ định một tập hợp các tín hiệu sẽ bị chặn khi bộ xử lý tín hiệu của chúng ta được gọi. Bất kỳ tín hiệu nào bị chặn đều không thể được chuyển đến một tiến trình. Chúng tôi đặt thành viên `sa_mask` thành tập hợp trống, điều đó có nghĩa là sẽ không có tín hiệu bổ sung nào bị chặn trong khi trình xử lý tín hiệu của chúng tôi đang chạy. POSIX đảm bảo rằng tín hiệu bị bắt luôn bị chặn trong khi trình xử lý của nó đang thực thi.

Đặt cờ SA_RESTART

8-17 `SA_RESTART` là cờ tùy chọn. Khi cờ được đặt, cuộc gọi hệ thống bị gián đoạn bởi tín hiệu này sẽ được kernel tự động khởi động lại. (Chúng ta sẽ nói nhiều hơn về các cuộc gọi hệ thống bị gián đoạn trong phần tiếp theo khi chúng ta tiếp tục ví dụ của mình.) Nếu tín hiệu bị bắt không phải là `SIGALRM`, chúng ta chỉ định cờ `SA_RESTART` nếu được xác định. (Lý do tạo trường hợp đặc biệt cho `SIGALRM` là mục đích tạo ra tín hiệu này thường là để đặt thời gian chờ cho thao tác I/O, như chúng tôi sẽ trình bày trong [Phần 14.2](#), trong trường hợp đó, chúng tôi muốn lệnh gọi hệ thống bị chặn tới [bi_gian_duan_bởi_tín_hiệu](#).)

Một số hệ thống cũ hơn, đặc biệt là SunOS 4.x, tự động khởi động lại cuộc gọi hệ thống bị gián đoạn theo mặc định và sau đó xác định phần bổ sung của cờ này là `SA_INTERRUPT`. Nếu cờ này được xác định, chúng tôi sẽ đặt nó nếu tín hiệu bị bắt là `SIGALRM`.

Gọi tín hiệu

18-20 Chúng ta gọi `sigaction` và sau đó trả về hành động cũ cho tín hiệu làm giá trị trả về của hàm `tín_hiệu`.

Trong suốt phần này, chúng ta sẽ sử dụng hàm `tín_hiệu` từ [Hình 5.6](#).

Ngữ nghĩa tín hiệu POSIX

Chúng tôi tóm tắt các điểm sau về xử lý tín hiệu trên hệ thống tuân thủ POSIX:

- Sau khi cài đặt bộ xử lý tín hiệu, nó vẫn được cài đặt. (Hệ thống cũ hơn đã xóa bộ xử lý tín hiệu mỗi lần nó được thực thi.)
- Trong khi bộ xử lý tín hiệu đang thực thi, tín hiệu được gửi sẽ bị chặn. Hơn nữa, bất kỳ tín hiệu bổ sung nào được chỉ định trong bộ tín hiệu `sa_mask` được chuyển tới `sigaction` khi cài đặt trình xử lý cũng đều bị chặn. Trong [Hình 5.6](#), chúng ta đặt `sa_mask` thành tập trống, nghĩa là không có tín hiệu bổ sung nào bị chặn ngoài tín hiệu bị bắt.
- Nếu tín hiệu được tạo ra một hoặc nhiều lần trong khi nó bị chặn thì thông thư ờng chỉ được gửi một lần sau khi tín hiệu được bỏ chặn. Nghĩa là, theo mặc định, Unix tín hiệu không được xếp hàng đợi. Chúng ta sẽ thấy một ví dụ về điều này trong phần tiếp theo. Tiêu chuẩn thời gian thực POSIX, 1003.1b, xác định một số tín hiệu đáng tin cậy được xếp hàng đợi, nhưng chúng tôi không sử dụng chúng trong văn bản này.
- Có thể chặn và bỏ chặn một cách có chọn lọc một tập hợp tín hiệu bằng chức năng `sigprocmask`. Điều này cho phép chúng tôi bảo vệ vùng mã quan trọng bằng cách

ngăn chặn các tín hiệu nhất định bị bắt trong khi vùng mã đó đang thực thi.

5.9 Xử lý tín hiệu 'SIGCHLD'

Mục đích của trạng thái zombie là duy trì thông tin về đứa trẻ để cha mẹ tìm nạp sau này. Thông tin này bao gồm ID tiến trình của tiến trình con, trạng thái chấm dứt của nó và thông tin về việc sử dụng tài nguyên của tiến trình con (thời gian CPU, bộ nhớ, v.v.). Nếu một tiến trình kết thúc và tiến trình đó có các tiến trình con ở trạng thái zombie, thì ID tiến trình cha của tất cả các tiến trình zombie con được đặt thành 1 (giá trị `init` quá trình), quá trình này sẽ kế thừa các phần tử con và dọn sạch chúng (tức là `init` sẽ đợi chúng để loại bỏ zombie). Một số hệ thống Unix hiển thị cột LỰA CHỌN cho quy trình zombie dưới dạng `<không còn tồn tại>`.

Xử lý Zombie

Rõ ràng là chúng tôi không muốn bỏ lại thây ma xung quanh. Chúng chiếm dung lư ợng trong kernel và cuối cùng chúng ta có thể hết tiến trình. Bất cứ khi nào chúng ta `chia tay` trẻ em, chúng ta phải `chờ đợi` để chúng không trở thành thây ma. Để thực hiện điều này, chúng tôi thiết lập trình xử lý tín hiệu để bắt `SIGCHLD` và trong trình xử lý đó, chúng tôi gọi `chờ`. (Chúng tôi sẽ mô tả các hàm `wait` và `waitpid` trong [Phần 5.10.](#)) Chúng tôi thiết lập trình xử lý tín hiệu bằng cách thêm lệnh gọi hàm

Tín hiệu (`SIGCHLD`, `sig_chld`);

trong [Hình 5.2](#), sau khi gọi tới `listen`. (Việc này phải được thực hiện đôi khi trước khi chúng ta phân tách thành phần con đầu tiên và chỉ cần thực hiện một lần.) Sau đó, chúng ta xác định bộ xử lý tín hiệu, hàm `sig_chld`, được hiển thị trong [Hình 5.7](#).

Hình 5.7 Phiên bản của bộ xử lý tín hiệu `SIGCHLD` gọi chờ (được cải thiện trong [Hình 5.11](#)).

`tcpcliserv/sigchldwait.c`

```
1 #bao gồm "unp.h"
```

```
2 khoảng trắng
```

```
3 sig_chld(int sig)
```

```
4 {
```

```
5     pid_t pid;
```

```

6     int      chỉ số;

7     pid = chờ(&stat);
8
9     printf("con %d đã chấm dứt\", pid);
10    trở lại;
11 }

```

Cảnh báo: Không nên gọi các hàm I/O tiêu chuẩn như `printf` trong bộ xử lý tín hiệu, vì những lý do mà chúng ta sẽ thảo luận trong [Phần 11.18](#). Chúng tôi gọi `printf` ở đây như [một công cụ chẩn đoán](#) để xem khi nào đưa trẻ kết thúc.

Trong System V và Unix 98, tiến trình con của một tiến trình không trở thành zombie nếu tiến trình đó đặt vị trí `SIGCHLD` thành `SIG_IGN`. Thật không may, điều này chỉ hoạt động trong System V và Unix 98. POSIX tuyên bố rõ ràng rằng hành vi này không được chỉ định.

Cách di động để xử lý zombie là bắt `SIGCHLD` và gọi `wait` hoặc `waitpid`.

Nếu chúng ta biên dịch chương trình [này](#)-[Hình 5.2](#), với lệnh gọi tới `Signal`, với `sig_chld` handler-trong Solaris 9 và sử dụng chức năng `tín hiệu` từ thư viện hệ thống (không phải phiên bản của chúng tôi từ [Hình 5.6](#)), chúng tôi có những điều sau:

năng lự ợng mặt trời% tcpserv02 &	khởi động máy chủ ở chế độ nền
[2] 16939	
năng lự ợng mặt trời% tcpcli01 127.0.0.1	sau đó khởi động ứng dụng khách ở nền tru ớc
Chào bạn	chúng tôi gõ cái này
Chào bạn	và điều này đư ợc lặp lại
^D	chúng tôi gõ ký tự EOF của chúng tôi
con 16942 chấm dứt	xuất ra bằng <code>printf</code> trong bộ xử lý tín hiệu
chấp nhận lỗi: Chức năng chính của cuộc gọi hệ thống bị gián đoạn bị hủy bỏ	

Trình tự các bước như sau:

1. Chúng tôi chấm dứt ứng dụng khách bằng cách nhập ký tự EOF của chúng tôi. Máy khách TCP gửi một FIN tới máy chủ và máy chủ phản hồi bằng ACK.
2. Việc nhận FIN sẽ chuyển EOF đến **dòng đọc đang chờ xử lý của trẻ**. Các con chấm dứt.
3. Nút gốc bị chặn trong cuộc gọi **chấp nhận** khi tín hiệu `SIGCHLD` đư ợc gửi. Hàm `sig_chld` thực thi (trình xử lý tín hiệu của chúng tôi), `chờ` tìm nạp PID và trạng thái kết thúc của trẻ, đồng thời `printf` đư ợc gọi từ trình xử lý tín hiệu. Bộ xử lý tín hiệu trả về.
4. Do phụ huynh bắt đư ợc tín hiệu trong khi phụ huynh bị chặn trong cuộc gọi hệ thống chậm (**chấp nhận**), kernel khiến **việc chấp nhận** trả về lỗi

EINTR (cuộc gọi hệ thống bị gián đoạn). Lớp cha không xử lý được lỗi này ([Hình 5.2](#)) nên nó hủy bỏ.

Mục đích của ví dụ này là để chỉ ra rằng khi viết các chương trình mạng bắt tín hiệu, chúng ta phải nhận biết được các cuộc gọi hệ thống bị gián đoạn và phải xử lý chúng. Trong ví dụ cụ thể này, chạy trên Solaris 9, chức năng **tín hiệu** được cung cấp trong thư viện C tiêu chuẩn không khiếu lệnh gọi hệ thống bị gián đoạn được hạt nhân tự động khởi động lại. Nghĩa là cờ **SA_RESTART** mà chúng ta đặt trong [Hình 5.6](#) không được đặt bởi chức năng **tín hiệu** trong thư viện hệ thống. Một số hệ thống khác tự động khởi động lại cuộc gọi **hệ thống bị gián đoạn**. Nếu chúng ta chạy ví dụ tương tự trong 4.4BSD, sử dụng phiên bản thư viện của hàm **tín hiệu**, kernel sẽ khởi động lại lệnh gọi hệ thống bị gián đoạn và **việc chấp nhận** sẽ không trả về lỗi. Để xử lý vấn đề tiềm ẩn này giữa các hệ điều hành khác nhau là một lý do khiếu chúng tôi xác định phiên bản **tín hiệu** của riêng mình

chức năng mà chúng ta sử dụng xuyên suốt văn bản ([Hình 5.6](#)).

Là một phần của quy ước mã hóa được sử dụng trong văn bản này, chúng tôi luôn mã hóa một **kết quả trả về rõ ràng** trong các bộ xử lý tín hiệu của chúng ta ([Hình 5.7](#)), mặc dù việc rơi ra khỏi phần cuối của hàm cũng thực hiện điều tương tự đối với hàm trả về **void**. Khi đọc mã, câu lệnh return không cần thiết đóng vai trò như một lời nhắc nhớ rằng lệnh return có thể làm gián đoạn cuộc gọi hệ thống.

Xử lý các cuộc gọi hệ thống bị gián đoạn

Chúng tôi đã sử dụng thuật ngữ "cuộc gọi hệ thống chậm" để mô tả **việc chấp nhận** và chúng tôi sử dụng thuật ngữ này cho bất kỳ cuộc gọi hệ thống nào có thể chặn vĩnh viễn. Nghĩa là, cuộc gọi hệ thống không bao giờ cần quay lại. Hầu hết các chức năng mạng đều thuộc loại này. Ví dụ: không có gì đảm bảo rằng lệnh gọi **chấp nhận** của máy chủ sẽ quay trở lại nếu không có máy khách nào kết nối với máy chủ. Tương tự, lệnh gọi **đọc** trong [Hình 5.3](#) của máy chủ của chúng tôi sẽ không bao giờ quay trở lại nếu máy **khách không bao giờ gửi** một dòng để máy chủ phản hồi. Các ví dụ khác về lệnh gọi hệ thống chậm là việc đọc và ghi đƣờng ống và thiết bị đầu cuối. Một ngoại lệ đáng chú ý là I/O đĩa, thư ờng trả về cho người gọi (giả sử không có lỗi phần cứng nghiêm trọng).

Quy tắc cơ bản áp dụng ở đây là khi một tiến trình bị chặn trong lệnh gọi hệ thống chậm và tiến trình đó bắt được tín hiệu và trình xử lý tín hiệu quay trở lại, lệnh gọi hệ thống có thể trả về lỗi **EINTR**. Một số hạt nhân tự động khởi động lại một số cuộc gọi hệ thống bị gián đoạn. Về tính di động, khi chúng ta viết một chương trình bắt tín hiệu (hầu hết các máy chủ đồng thời đều bắt được **SIGCHLD**), chúng ta phải chuẩn bị cho các lệnh gọi hệ thống chậm tới trả lại **EINTR**. Các vấn đề về tính di động xảy ra do các từ hạn định "có thể" và "một số" đã được sử dụng trước đó và thực tế là việc hỗ trợ cờ POSIX **SA_RESTART** là tùy chọn. Ngay cả khi việc triển khai hỗ trợ cờ **SA_RESTART** thì không phải tất cả các cuộc gọi hệ thống bị gián đoạn đều có thể tự động được khởi động lại. Ví dụ: hầu hết các triển khai có nguồn gốc từ Berkeley không bao giờ tự động khởi động lại **lựa chọn** và một số triển khai này không bao giờ khởi động lại **chấp nhận** hoặc **recvfrom**.

Để xử lý việc chấp nhận bị gián đoạn, chúng ta thay đổi lời gọi chấp nhận trong [Hình 5.2, phần bắt đầu](#) của vòng lặp `for`, thành như sau:

```

vì (      ;   )  {
    clilen = sizeof(cliaddr);
    if ( (connfd = chấp nhận (listenfd, (SA *) &cliaddr, &clilen)) < 0)
    {
        nếu (errno == EINTR)
            Tiếp tục;           /* quay lại for () */
        khác
            err_sys ("chấp nhận lỗi");
    }
}

```

Lưu ý rằng chúng ta gọi hàm chấp nhận chứ không phải hàm bao bọc `Chấp nhận`, vì chúng ta phải tự mình xử lý sự cố của chức năng.

Những gì chúng tôi đang làm trong đoạn mã này là khởi động lại cuộc gọi hệ thống bị gián đoạn. Điều này tốt cho việc chấp nhận, cùng với các chức năng như `đọc`, `viết`, `chọn` và `mở`. Nhưng có một chức năng mà chúng ta không thể khởi động lại: `kết nối`. Nếu hàm này trả về `EINTR` thì chúng ta không thể gọi lại vì làm như vậy sẽ trả về lỗi ngay lập tức. Khi `kết nối` bị gián đoạn do tín hiệu bắt được và không được tự động khởi động lại, chúng ta phải gọi `select` để chờ kết nối hoàn tất, như chúng tôi sẽ mô tả trong [Phần 16.3](#).

5.10 Chức năng 'chờ' và 'waitpid'

Trong [Hình 5.7](#), chúng ta gọi hàm `chờ` để xử lý thành phần con bị kết thúc.

#include <sys/wait.h>
<code>pid_t chờ (int *statloc);</code>
<code>pid_t waitpid (pid_t pid, int *statloc, int tùy chọn);</code>
Cả hai đều trả về: ID tiến trình nếu OK, 0 hoặc -1 bị lỗi

`wait` và `waitpid` đều trả về hai giá trị: giá trị trả về của hàm là ID tiến trình của tiến trình con bị kết thúc và trạng thái kết thúc của tiến trình con (một số nguyên) được trả về thông qua con trỏ `statloc`. Có ba macro mà chúng ta có thể gọi để kiểm tra trạng thái chấm dứt và cho chúng tôi biết liệu đứa trẻ bị chấm dứt bình thường hay bị giết

bởi một tín hiệu, hoặc bị dừng lại do điều khiển công việc. Sau đó, các macro bổ sung cho phép chúng tôi tìm nạp trạng thái thoát của thành phần con hoặc giá trị của tín hiệu đã giết chết thành phần con hoặc giá trị của tín hiệu điều khiển công việc đã dừng thành phần con. Chúng ta sẽ sử dụng macro `WIFEXITED` và `WEXITSTATUS` trong [Hình 15.10](#) cho mục đích này.

Nếu không có thành phần con nào bị kết thúc cho tiến trình đang gọi `chờ`, nhưng tiến trình đó có một hoặc nhiều thành phần con vẫn đang thực thi, thì `hãy đợi` các khôi con đầu tiên trong số các thành phần con hiện có kết thúc.

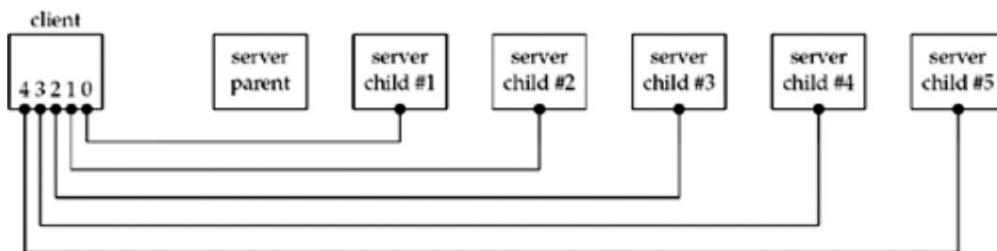
`waitpid` cho phép chúng ta kiểm soát nhiều hơn quá trình nào sẽ chờ và có chặn hay không. Đầu tiên, đổi số pid cho phép chúng ta chỉ định ID tiến trình mà chúng ta muốn chờ.

Giá trị -1 có nghĩa là hãy đợi đứa con đầu tiên của chúng ta kết thúc. (Có các tùy chọn khác, xử lý ID nhóm quy trình, nhưng chúng tôi không cần chúng trong văn bản này.) Đổi số tùy chọn cho phép chúng tôi chỉ định các tùy chọn bổ sung. Lựa chọn phổ biến nhất là `WNOHANG`. Tùy chọn này yêu cầu kernel không chặn nếu không có con nào bị chấm dứt.

Sự khác biệt giữa chờ đợi và chờ đợi

Bây giờ chúng ta minh họa sự khác biệt giữa các hàm `wait` và `waitpid` khi được sử dụng để đợi dẹp các phần tử con bị kết thúc. Để làm điều này, chúng ta sửa đổi máy khách TCP như trong [Hình 5.9](#). Máy khách thiết lập năm kết nối với máy chủ và sau đó chỉ sử dụng kết nối đầu tiên (`sockfd[0]`) trong lệnh gọi tới `str_cli`. Mục đích của việc thiết lập nhiều kết nối là sinh ra nhiều máy chủ con từ máy chủ đồng thời, như trong [Hình 5.8](#).

Hình 5.8. Máy khách có năm kết nối được thiết lập tới cùng một máy chủ đồng thời.



Hình 5.9 Máy khách TCP thiết lập năm kết nối với máy chủ.

`tcpcliserv/tcpcli04.c`

1 #bao gồm "unp.h"

2 số nguyên

3 chính (int argc, char **argv)

4 {

```

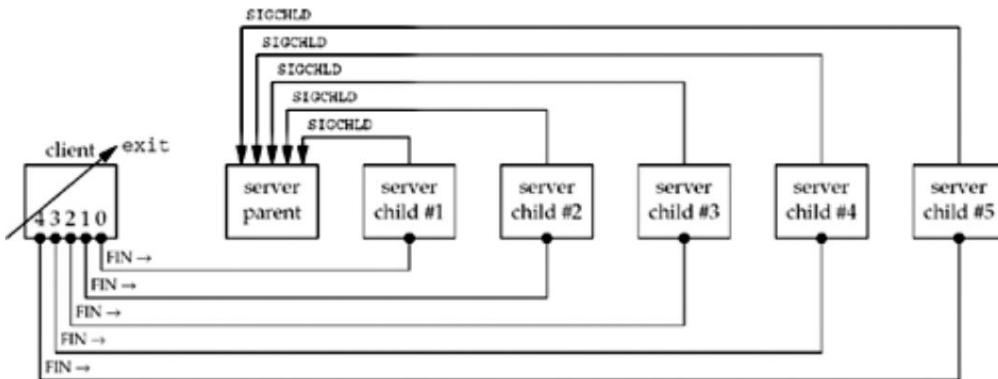
5     int      tôi, sockfd[5];
6     struct sockaddr_in servaddr;
7
8     nếu (argc != 2)
9         err_quit ("cách sử dụng: tcpcli <IPaddress>");
10
11    vì (i = 0; i < 5; i++) {
12        sockfd[i] = Ở cắm (AF_INET, SOCK_STREAM, 0);
13
14        bzero (&servaddr, sizeof (servaddr));
15        servaddr.sin_family = AF_INET;
16        servaddr.sin_port = htons (SERV_PORT);
17        Inet_pton (AF_INET, argv[1], &servaddr.sin_addr);
18
19        Kết nối (sockfd[i], (SA *) &servaddr, sizeof (servaddr));
20    }
21
22    str_cli (stdin, sockfd[0]); /* làm tắt cả */
23
24    thoát (0);
25 }

```

Khi máy khách kết thúc, tất cả các bộ mô tả mở sẽ tự động được đóng bởi kernel (chúng tôi không gọi đóng mà chỉ gọi là **thoát**) và tất cả năm kết nối đều bị chấm dứt cùng một lúc. Điều này làm cho năm FIN được gửi đi, một FIN trên mỗi kết nối, từ đó khiến cả năm máy chủ con chấm dứt cùng một lúc. Điều này gây ra

năm tín hiệu **SIGCHLD** sẽ được gửi đến nút cha cùng lúc, như chúng tôi trình bày trong [Hình 5.10](#).

Hình 5.10. Khách hàng chấm dứt, đóng tất cả năm kết nối, chấm dứt cả năm kết nối con.



Chính việc xuất hiện nhiều lần của cùng một tín hiệu đã gây ra vấn đề mà chúng ta sắp gặp.

Đầu tiên chúng tôi chạy máy chủ ở chế độ nền và sau đó là ứng dụng khách mới của chúng tôi. Máy chủ của chúng tôi là [Hình 5.2](#), được sửa đổi thành [tín hiệu cuộc gọi để thiết lập](#) [Hình 5.7](#) làm bộ xử lý tín hiệu cho [SIGCHLD](#).

```
linux% tcpserv03 &
```

```
[1] 20419
```

```
linux% tcpcli04 127.0.0.1
```

Xin chào	chúng tôi gõ cái này
Xin chào	và nó đư ợc vang vọng
^D	sau đó chúng tôi nhập ký tự EOF của mình
con 20426 chấm dứt	đầu ra của máy chủ

Điều đầu tiên chúng tôi nhận thấy là chỉ có một [printf](#) đư ợc xuất ra, khi chúng tôi cho rằng cả năm phần tử con đều đã kết thúc. Nếu chúng ta thực thi [ps](#), chúng ta thấy rằng bốn phần tử con còn lại vẫn tồn tại dưới dạng zombie.

PID TTY	THỜI GIAN CMD
20419 đi ểm/6	00:00:00 tcpserv03
20421 đi ểm/6	00:00:00 tcpserv03 <không còn tồn tại>
20422 đi ểm/6	00:00:00 tcpserv03 <không còn tồn tại>
20423 đi ểm/6	00:00:00 tcpserv03 <không còn tồn tại>

Việc thiết lập bộ xử lý tín hiệu và gọi [wait](#) từ bộ xử lý đó là không đủ để ngăn chặn zombie. Vẫn đè là tất cả năm tín hiệu đư ợc tạo ra trước khi bộ xử lý tín hiệu đư ợc thực thi và bộ xử lý tín hiệu chỉ đư ợc thực thi một lần vì các tín hiệu Unix thư ờng không đư ợc xếp hàng đợi. Hơn nữa, vấn đề này là không xác định.

Trong ví dụ chúng ta vừa chạy, với máy khách và máy chủ trên cùng một máy chủ, trình xử lý tín hiệu đư ợc thực thi một lần, để lại bốn zombie. Nhưng nếu chúng ta chạy máy khách và máy chủ trên các máy chủ khác nhau, trình xử lý tín hiệu thư ờng đư ợc thực thi hai lần: một lần do tín hiệu đầu tiên đư ợc tạo và do bốn tín hiệu còn lại xảy ra trong khi trình xử lý tín hiệu đang thực thi, nên trình xử lý đư ợc gọi chỉ một lần nữa thôi. Điều này đe 1 lại ba

zombie. Nhưng đôi khi, có thể phụ thuộc vào thời gian FIN đến máy chủ, trình xử lý tín hiệu đư ợc thực thi ba hoặc thậm chí bốn lần.

Giải pháp đúng là gọi [waitpid](#) thay vì [chờ đợi](#). [Hình 5.11](#) hiển thị phiên bản hàm [sig_chld](#) xử lý [SIGCHLD](#) chính xác. Phiên bản này hoạt động vì chúng tôi gọi [waitpid](#) trong một vòng lặp, tìm nạp trạng thái của bất kỳ phần tử con nào của chúng tôi đã kết thúc. Chúng ta phải chỉ định tùy chọn [WNOHANG](#) : Điều này yêu cầu [waitpid](#) không chặn nếu có các phần tử con đang chạy chưa kết thúc. Trong [Hình 5.7](#), chúng ta không thể gọi

chờ trong một vòng lặp, vì không có cách nào ngăn chặn việc chờ đợi bị chặn nếu có các phần tử con đang chạy chư a kết thúc.

[Hình 5.12](#) hiển thị phiên bản cuối cùng của máy chủ của chúng tôi. Nó xử lý chính xác việc trả về EINTR từ việc chấp nhận và nó thiết lập một trình xử lý tín hiệu ([Hình 5.11](#)) gọi ~~waitpid~~ cho tất cả trẻ em bị chấm dứt.

Hình 5.11 Phiên bản cuối cùng (chính xác) của hàm sig_chld gọi waitpid.

tcpcliserv/sigchldwaitpid.c

```

1 #include "unp.h"

2 khoảng trắng

3 sig_chld(int sig)
4 {
5     pid_t pid;
6     int      chỉ số;

7     while ( (pid = waitpid(-1, &stat, WNOHANG)) > 0)
8         printf("con %d đã chấm dứt\n", pid);
9     trở lại;
10 }
```

Hình 5.12 Phiên bản cuối cùng (chính xác) của máy chủ TCP xử lý lỗi EINTR từ khi chấp nhận.

tcpcliserv/tcpserv04.c

```

1 #bao gồm          "unp.h"
2 số nguyên

3 chính(int argc, char **argv)
4 {
5     int      listenfd, connfd;
6     pid_t   trẻ con;
7     socklen_t clilen;
8     struct sockaddr_in cliaddr, servaddr;
9     void sig_chld(int);

10    listenfd = Ở cẩm (AF_INET, SOCK_STREAM, 0);

11    bzero (&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14    servaddr.sin_port = htons(SERV_PORT);
```

```

15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16     Nghe(listenfd, LISTENQ);

17     Tín hiệu (SIGCHLD, sig_chld); /* phải gọi waitpid() */

18     vì (      ;   ) {
19         clilen = sizeof(cliaddr);
20         if ( (connfd = chấp nhận (listenfd, (SA *) &cliaddr, &clilen)) < 0)
21             {
22                 nếu (errno == EINTR)
23                     Tiếp tục;           /* quay lại for() */
24                 khác
25                 err_sys("chấp nhận lỗi");
26             }
27
28         if ( (childpid = Fork()) == 0) { /* tiến trình con */
29             Đóng(nghefd);           /* đóng socket nghe */
30             str_echo(connfd); /* xử lý yêu cầu */
31             thoát (0);
32         }
33     }

```

Mục đích của phần này là trình bày ba tình huống mà chúng ta có thể gặp phải khi lập trình mạng:

1. Chúng ta phải bắt được tín hiệu **SIGCHLD** khi **phân nhánh** các tiến trình con.
2. Chúng ta phải xử lý các cuộc gọi hệ thống bị gián đoạn khi bắt được tín hiệu.
3. Trình xử lý **SIGCHLD** phải được mã hóa chính xác bằng cách sử dụng **waitpid** để ngăn chặn bất kỳ zombie nào bị bỏ lại xung quanh.

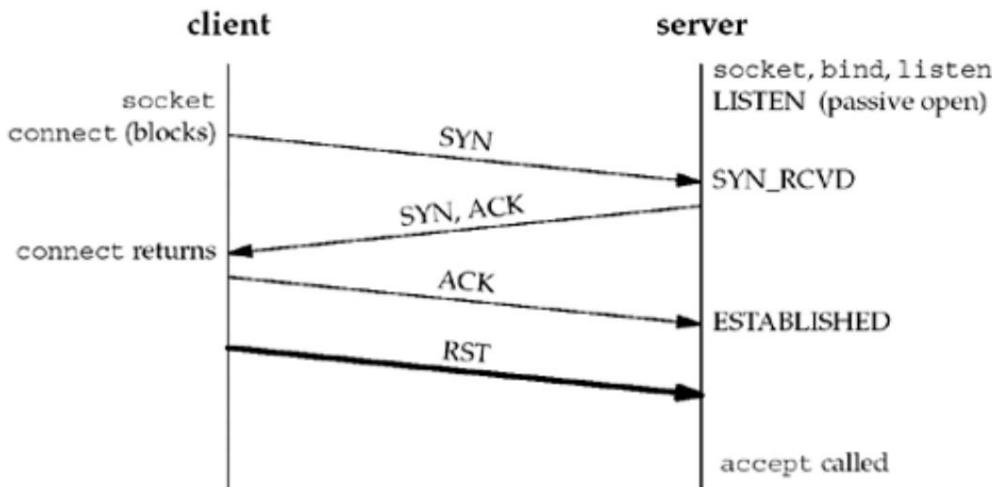
Phiên bản cuối cùng của máy chủ TCP của chúng tôi ([Hình 5.12](#)), cùng với trình xử lý **SIGCHLD** trong [Hình 5.11](#), [xử lý cả ba kịch bản](#).

5.11 Hủy kết nối trư ớc khi 'chấp nhận' Trả về

Có một điều kiện khác tương tự như ví dụ về cuộc gọi hệ thống bị gián đoạn ở phần trư ớc có thể khiến **việc chấp nhận** trả về một lỗi không nghiêm trọng, trong trường hợp đó chúng tôi

chỉ nên gọi lại **chấp nhận**. Chuỗi các gói được hiển thị trong [Hình 5.13](#) đã được nhìn thấy [trên các máy chủ](#) bận (thu ờng là các máy chủ Web bận).

Hình 5.13. Nhận RST cho kết nối THÀNH LẬP tru ờc khi chấp nhận đư ợc gọi.



Tại đây, quá trình bắt tay ba chiều hoàn tất, kết nối đư ợc thiết lập và sau đó máy khách TCP sẽ gửi RST (đặt lại). Về phía máy chủ, kết nối đư ợc xếp hàng bởi TCP của nó, chờ quá trình máy chủ gọi **chấp nhận** khi RST đến. Một thời gian sau, các cuộc gọi xử lý máy chủ **chấp nhận**.

Một cách dễ dàng để phòng tình huống này là khởi động máy chủ, yêu cầu nó gọi **socket**, **liên kết** và **lắng nghe**, sau đó chuyển sang chế độ ngủ trong một khoảng thời gian ngắn trước khi gọi **chấp nhận**.

Trong khi quá trình máy chủ đang ngủ, hãy khởi động máy khách và yêu cầu nó gọi **socket** và **kết nối**. Ngay khi **kết nối** trở lại, hãy đặt tùy chọn ô cảm **SO_Linger** để tạo RST (mà chúng tôi sẽ mô tả trong [Phần 7.5](#) và hiển thị ví dụ trong [Hình 16.21](#)) và kết thúc.

Thật không may, điều xảy ra với kết nối bị hủy lại phụ thuộc vào việc triển khai.

Việc triển khai có nguồn gốc từ Berkeley xử lý hoàn toàn kết nối bị hủy trong kernel và quy trình máy chủ không bao giờ nhìn thấy nó. Tuy nhiên, hầu hết các triển khai SVR4 đều trả về lỗi cho quy trình dưới dạng trả về từ **chấp nhận** và lỗi phụ thuộc vào việc triển khai. Việc triển khai SVR4 này trả về lỗi EPROTO ("lỗi giao thức"), như ng POSIX chỉ định rằng kết quả trả về phải đư ợc **ECONNABORTED**

thay vào đó ("phản mềm khi kết nối bị hủy"). Lý do thay đổi POSIX là EPROTO cũng đư ợc trả về khi một số sự kiện nghiêm trọng liên quan đến giao thức xảy ra trên hệ thống con luồng. Việc trả về cùng một lỗi đối với hành động hủy bỏ không nghiêm trọng của một kết nối đã đư ợc thiết lập bởi máy khách khiến máy chủ không thể biết liệu có nên gọi lại **chấp nhận** hay không. Trong trường hợp xảy ra lỗi ECONNABORTED, máy chủ có thể bỏ qua lỗi và chỉ cần gọi lại **chấp nhận**.

Các bút ờc liên quan đến hạt nhân có nguồn gốc từ Berkeley không bao giờ chuyển lỗi này sang quy trình có thể đư ợc thực hiện trong TCPv2. RST đư ợc xử lý trên p. 964, gây ra **tcp_close**

đư ợc gọi là. Hàm này gọi `in_pcbodetach` trên p. 897, lần lượt gọi `sofree` trên p. 719. `sofree` (tr. 473) phát hiện ra rằng ổ cắm bị hủy vẫn còn trong hàng đợi kết nối đã hoàn thành của ổ cắm nghe và loại bỏ ổ cắm khỏi hàng đợi và giải phóng ổ cắm. Khi máy chủ bắt đầu gọi `chấp nhận`, nó sẽ không bao giờ biết rằng kết nối đã hoàn thành đã bị xóa khỏi hàng đợi.

Chúng ta sẽ quay lại các kết nối bị hủy bỏ này trong [Phần 16.6](#) và xem chúng có thể gây ra sự cố như thế nào khi kết hợp với `select` và socket nghe ở chế độ chặn thông thư ờng.

5.12 Chấm dứt quá trình máy chủ

Bây giờ chúng ta sẽ khởi động máy khách/máy chủ của mình và sau đó hủy tiến trình con của máy chủ. Điều này mô phỏng sự cố của quá trình máy chủ, vì vậy chúng tôi có thể thấy điều gì xảy ra với máy khách. (Chúng ta phải cẩn thận để phân biệt giữa sự cố của tiến trình máy chủ mà chúng tôi sắp mô tả và sự cố của máy chủ máy chủ mà chúng tôi sẽ mô tả trong [Phần 5.14.](#)) Các bước sau đây diễn ra:

1. Chúng tôi khởi động máy chủ và máy khách rồi gõ một dòng cho máy khách để xác minh rằng tất cả đều ổn. Dòng đó đư ợc lặp lại bình thường bởi máy chủ con.

2. Chúng tôi tìm ID tiến trình của máy chủ con và **tiêu diệt** nó. Là một phần của quá trình chấm dứt, tất cả các mô tả mở trong phần tử con đều bị đóng. Điều này khiến FIN đư ợc gửi đến máy khách và máy khách TCP phản hồi bằng ACK. Đây là nửa đầu của quá trình chấm dứt kết nối TCP.

3. Tín hiệu **SIGCHLD** đư ợc gửi đến máy chủ mẹ và đư ợc xử lý chính xác ([Hình 5.12](#)).

4. Không có gì xảy ra ở máy khách. TCP máy khách nhận FIN từ TCP máy chủ và phản hồi bằng ACK, như ng vẫn đề là tiến trình máy khách bị chặn trong cuộc gọi tới `fgets` đang chờ một dòng từ thiết bị đầu cuối.

5. Chạy `netstat` vào thời điểm này sẽ hiển thị trạng thái của các ổ cắm.

6.

7.

số 8.

9. linux% `netstat -a | grep 9877`

10. `tcp 0 0 *:9877 *:*` NGHE

11. `tcp 0 0 máy chủ cục bộ:9877 máy chủ cục bộ:43604`

`FIN_WAIT2`

12. `tcp 1 0 máy chủ cục bộ:43604 máy chủ cục bộ:9877`

`CLOSE_WAIT`

13.

Từ [Hình 2.4](#), chúng ta thấy rằng một nửa chuỗi kết nối TCP đã diễn ra.

14. Chúng ta vẫn có thể gõ một dòng đầu vào cho máy khách. Đây là những gì xảy ra ở máy khách bắt đầu từ Bước 1:

linux % `tcpcli01 127.0.0.1` khởi động máy khách

Xin chào

dòng đầu tiên chúng ta gõ

Xin chào

đư ợc lặp lại chính xác ở đây chúng tôi giết máy chủ con trên máy chủ

dòng khác

sau đó chúng tôi gõ dòng thứ hai cho máy khách

`str_cli : máy chủ bị chấm dứt sớm`

15. Khi chúng ta gõ "một dòng khác", `str_cli` gọi `sẽ đư ợc ghi` và máy khách TCP sẽ gửi dữ liệu đến máy chủ. Điều này đư ợc TCP cho phép vì việc nhận FIN của máy khách TCP chỉ cho biết rằng tiến trình máy chủ đã đóng kết nối và sẽ không gửi thêm bất kỳ dữ liệu nào nữa. Việc nhận FIN không thông báo cho máy khách TCP rằng quá trình máy chủ đã kết thúc (trong trường hợp này là nó đã kết thúc). Chúng ta sẽ đề cập lại vấn đề này trong [Phần 6.6](#) khi nói về cơ chế nửa đóng của TCP.

16. Khi máy chủ TCP nhận đư ợc dữ liệu từ máy khách, nó sẽ phản hồi bằng RST do quá trình mở ở cắm đó đã chấm dứt. Chúng tôi có thể xác minh rằng RST đã đư ợc gửi bằng cách xem các gói bằng `tcpdump`.

17. Quy trình máy khách sẽ không thấy RST vì nó gọi `readline` ngay sau lệnh gọi `write` và `readline` trả về 0 (EOF) ngay lập tức do FIN đã nhận đư ợc ở Bước 2. Khách hàng của chúng tôi không mong đợi nhận đư ợc EOF tại thời điểm này [điểm \(Hình 5.5\)](#) nên nó thoát với thông báo lỗi "máy chủ bị chấm dứt sớm".

18. Khi máy khách kết thúc (bằng cách gọi `err_quit` trong [Hình 5.5](#)), tất cả `các bộ mô-tả mở` của nó sẽ bị đóng.

Những gì chúng tôi đã mô tả cũng phụ thuộc vào thời gian của ví dụ. Cuộc gọi tới `đư ờng dây đọc` của máy khách có thể xảy ra trước khi máy khách nhận đư ợc RST của máy chủ hoặc có thể xảy ra sau đó. Nếu `dòng đọc` xảy ra trước khi nhận đư ợc RST, như chúng tôi đã trình bày trong ví dụ của mình, thì kết quả là một EOF không mong muốn trong máy khách. Nhưng nếu RST đến trước thì kết quả là lỗi ECONNRESET ("Thiết lập lại kết nối ngang hàng") từ `dòng đọc`.

Vấn đề trong ví dụ này là máy khách bị chặn lệnh gọi tới `fgets` khi FIN đến trên ổ cắm. Máy khách thực sự đang làm việc với hai bộ mô-tả-ở cắm và đầu vào của người dùng-và thay vì chỉ chặn đầu vào từ một trong hai nguồn (vì `str_cli` hiện đư ợc mã hóa), nó sẽ chặn đầu vào từ một trong hai nguồn. Thật vậy, đây là mục đích của các hàm `chọn` và `thêm dò` mà chúng tôi sẽ mô tả trong

Chú ý 6. Khi chúng ta code lại hàm `str_cli` ở [Mục 6.4](#), ngay khi chúng ta kill máy chủ con, máy khách sẽ được thông báo về FIN đã nhận.

5.13 Tín hiệu 'SIGPIPE'

Điều gì xảy ra nếu máy khách bỏ qua lỗi trả về từ `dòng đọc` và ghi thêm dữ liệu vào máy chủ? Điều này có thể xảy ra, ví dụ: nếu máy khách cần thực hiện hai lần ghi vào máy chủ trước khi đọc lại bất cứ thứ gì, với lần ghi đầu tiên gởi ra RST.

Quy tắc được áp dụng là: Khi một quy trình ghi vào ổ cắm đã nhận được RST, tín hiệu `SIGPIPE` sẽ được gửi đến quy trình đó. Hành động mặc định của tín hiệu này là chấm dứt quá trình, do đó quá trình phải bắt được tín hiệu để tránh bị chấm dứt ngoài ý muốn.

Nếu quá trình bắt được tín hiệu và trả về từ bộ xử lý tín hiệu hoặc bỏ qua tín hiệu thì thao tác ghi sẽ trả về `EPIPE`.

Một câu hỏi thư ờng gặp (FAQ) trên Usenet là làm thế nào để có được tín hiệu này trong lần ghi đầu tiên chứ không phải lần thứ hai. Điều này là không thể. Theo thảo luận của chúng tôi ở trên, lần ghi đầu tiên sẽ tạo ra RST và lần ghi thứ hai sẽ tạo ra tín hiệu. Bạn có thể ghi vào ổ cắm đã nhận được FIN như ng sẽ có lỗi khi ghi vào ổ cắm đã nhận được RST.

Để xem điều gì xảy ra với `SIGPIPE`, chúng tôi sửa đổi ứng dụng khách của mình như [Hình 5.14](#).

Hình 5.14 `str_cli` gọi hai lần.

`tcpcliserv/str_cli11.c`

```

1 #bao gồm          "unp.h"
2 khoảng trắng
3 str_cli(TÂP_TIN *fp, int sockfd)
4 {
5     dòng gửi char [MAXLINE], recvline [MAXLINE];
6
7     Đã viết(sockfd, sendline, 1);
8     ngủ(1);
9     Đã viết(sockfd, sendline + 1, strlen(sendline) - 1);

```

```

10         if (Readline(sockfd, recvline, MAXLINE) == 0)
11             err_quit("str_cli: máy chủ đã kết thúc sớm");
12
13     Fputs(recvline, stdout);
14 }

```

7-9 Tất cả những gì chúng tôi đã thay đổi là gọi `write` hai lần: lần đầu tiên byte đầu tiên của dữ liệu được ghi vào ổ cắm, sau đó là tạm dừng một giây, tiếp theo là phần còn lại của dòng. Mục đích là để người viết đầu tiên gửi ra RST và sau đó để cái thứ hai được viết để tạo `SIGPIPE`.

Nếu chúng tôi chạy ứng dụng khách trên máy chủ Linux của mình, chúng tôi sẽ nhận được:

linux% `tcpclill 127.0.0.1`

Chào bạn	chúng tôi gõ dòng này
Chào bạn	điều này được lặp lại bởi máy chủ
	ở đây chúng tôi giết máy chủ con
tạm biệt	sau đó chúng ta gõ dòng này
Ông bị hỏng	cái này được in bởi shell

Chúng tôi khởi động máy khách, nhập một dòng, xem dòng đó lặp lại chính xác và sau đó chấm dứt máy chủ con trên máy chủ. Sau đó, chúng tôi nhập một dòng khác ("tạm biệt") và shell cho chúng tôi biết quy trình đã chết với tín hiệu `SIGPIPE` (một số shell không in bất cứ thứ gì khi một tiến trình chết mà không hủy lỗi, nhưng shell chúng tôi đang sử dụng cho ví dụ này, `bash`, cho chúng ta biết điều chúng ta muốn biết).

Cách được đề xuất để xử lý `SIGPIPE` tùy thuộc vào ứng dụng muốn làm gì khi điều này xảy ra. Nếu không có gì đặc biệt để làm thì việc thiết lập cách xử lý tín hiệu thành `SIG_IGN` thật dễ dàng, giả sử rằng các hoạt động đầu ra tiếp theo sẽ gặp lỗi `EPIPE` và chấm dứt. Nếu cần có các hành động đặc biệt khi tín hiệu xảy ra (có thể ghi vào tệp nhật ký), thì tín hiệu sẽ được bắt và mọi hành động mong muốn có thể được thực hiện trong bộ xử lý tín hiệu. Tuy nhiên, hãy lưu ý rằng nếu sử dụng nhiều ổ cắm thì việc truyền tín hiệu sẽ không cho chúng tôi biết ổ cắm nào gặp lỗi. Nếu chúng ta cần biết `thao tác ghi` nào gây ra lỗi thì chúng ta phải bỏ qua tín hiệu hoặc trả về từ bộ xử lý tín hiệu và xử lý `EPIPE` từ `thao tác ghi`.

5.14 Sự cố máy chủ máy chủ

Kịch bản này sẽ kiểm tra xem điều gì xảy ra khi máy chủ gặp sự cố. Để mô phỏng điều này, chúng ta phải chạy máy khách và máy chủ trên các máy chủ khác nhau. Sau đó, chúng tôi khởi động máy chủ, khởi động máy khách, nhập một dòng tới máy khách để xác minh rằng kết nối đã hoạt động, ngắt kết nối máy chủ khỏi mạng và nhập một dòng khác vào máy khách.

Điều này cũng bao gồm trường hợp máy chủ không thể truy cập được khi máy khách gửi dữ liệu (tức là một số bộ định tuyến trung gian ngừng hoạt động sau khi kết nối được thiết lập).

Các bước sau đây diễn ra:

1. Khi máy chủ gặp sự cố, không có gì được gửi đi trên mạng hiện có kết nối. Nghĩa là, chúng tôi giả định máy chủ gặp sự cố và không bị ngắt vận hành tắt (điều này chúng tôi sẽ đề cập trong [Phần 5.16](#)).
2. Chúng tôi nhập một dòng đầu vào cho máy khách, nó được viết bằng cách [ghi \(Hình 5.5\) và được](#) máy khách TCP gửi dưới dạng một phân đoạn dữ liệu. Sau đó, máy khách sẽ chặn cuộc gọi đến [đư ờng dẫn](#), chờ phản hồi lặp lại.
3. Nếu chúng ta xem mạng bằng [tcpdump](#), chúng ta sẽ thấy máy khách TCP liên tục truyền lại phân đoạn dữ liệu, cố gắng nhận ACK từ máy chủ.

[Phần 25.11](#) của TCPv2 hiển thị một mẫu điển hình cho việc truyền lại TCP: Việc triển khai có nguồn gốc từ Berkeley truyền lại phân đoạn dữ liệu 12 lần, đợi khoảng 9 phút trước khi bỏ cuộc. Khi máy khách TCP cuối cùng từ bỏ (giả sử máy chủ của máy chủ chưa được khởi động lại trong thời gian này hoặc nếu máy chủ của máy chủ không gặp sự cố nhưng không thể truy cập được trên mạng, giả sử máy chủ vẫn không thể truy cập được), lỗi sẽ được trả về máy khách quá trình. Vì máy khách bị chặn lệnh gọi tới [readline](#) nên nó sẽ trả về lỗi.

Giả sử máy chủ máy chủ gặp sự cố và không có phản hồi nào đối với các phân đoạn dữ liệu của máy khách thì lỗi là [ETIMEDOUT](#). Nhưng nếu một số bộ định tuyến trung gian xác định rằng máy chủ không thể truy cập được và phản hồi bằng thông báo "không thể truy cập đích" ICMP thì lỗi là [EHOSTUNREACH](#)

hoặc [ENETUNREACH](#).

Mặc dù khách hàng của chúng tôi phát hiện ra (cuối cùng) rằng thiết bị ngang hàng không hoạt động hoặc không thể truy cập được, nhưng có những lúc chúng tôi muốn phát hiện điều này nhanh hơn là đợi chín phút. Giải pháp là đặt thời gian chờ cho lệnh gọi đến [đư ờng dẫn](#) mà chúng ta sẽ thảo luận trong [Phần 14.2.](#)

Kịch bản mà chúng ta vừa thảo luận phát hiện ra rằng máy chủ chỉ gặp sự cố khi chúng ta gửi dữ liệu đến máy chủ đó. Nếu chúng tôi muốn phát hiện sự cố của máy chủ lùu trữ ngay cả khi chúng tôi không chủ động gửi dữ liệu cho nó thì cần phải có một kỹ thuật khác. Chúng ta sẽ thảo luận về tùy chọn ô cảm [SO_KEEPALIVE](#) trong [Phần 7.5.](#)

5.15 Sự cố và khởi động lại máy chủ máy chủ

Trong trường hợp này, chúng tôi sẽ thiết lập kết nối giữa máy khách và máy chủ, sau đó giả sử máy chủ lưu trữ gặp sự cố và khởi động lại. Trong phần trước, máy chủ vẫn ngừng hoạt động khi chúng tôi gửi dữ liệu cho nó. Ở đây, chúng tôi sẽ cho phép máy chủ khởi động lại trước khi gửi dữ liệu. Cách dễ nhất để mô phỏng điều này là thiết lập kết nối, ngắt kết nối máy chủ khỏi mạng, tắt máy chủ lưu trữ rồi khởi động lại, sau đó kết nối lại máy chủ lưu trữ với mạng.

Chúng tôi không muốn khách hàng nhìn thấy

máy chủ máy chủ ngừng hoạt động (chúng tôi sẽ đề cập đến vấn đề này trong [Phần 5.16](#)).

Như đã nêu ở phần trước, nếu máy khách không chủ động gửi dữ liệu đến máy chủ khi máy chủ gặp sự cố thì máy khách không biết rằng máy chủ máy chủ đã gặp sự cố.

(Điều này giả định rằng chúng tôi không sử dụng tùy chọn ô cắm `SO_KEEPALIVE`.) Các bước sau đây diễn ra:

1. Chúng tôi khởi động máy chủ và sau đó là máy khách. Chúng tôi gõ một dòng để xác minh rằng kết nối được thiết lập.
2. Máy chủ gặp sự cố và khởi động lại.
3. Chúng tôi nhập một dòng đầu vào tới máy khách, dòng này được gửi dưới dạng phân đoạn dữ liệu TCP tới máy khách. Máy chủ lưu trữ.
4. Khi máy chủ khởi động lại sau khi gặp sự cố, TCP của nó sẽ mất tất cả thông tin về các kết nối đã tồn tại trước khi xảy ra sự cố. Do đó, máy chủ TCP phản hồi phân đoạn dữ liệu nhận được từ máy khách bằng RST.
5. Máy khách của chúng tôi bị chặn lệnh gọi đến `đường dẫn` khi nhận được RST, gây ra `readline` để trả về lỗi `ECONNRESET`.

Nếu điều quan trọng là máy khách của chúng tôi phải phát hiện sự cố của máy chủ máy chủ, ngay cả khi máy khách không chủ động gửi dữ liệu thì cần phải có một số kỹ thuật khác (chẳng hạn như tùy chọn ô cắm `SO_KEEPALIVE` hoặc một số chức năng nhịp tim của máy khách/máy chủ).

5.16 Tắt máy chủ máy chủ

Hai phần trước đã thảo luận về sự cố của máy chủ lưu trữ hoặc máy chủ lưu trữ không thể truy cập được trên mạng. Bây giờ chúng ta xem xét điều gì sẽ xảy ra nếu máy chủ của máy chủ bị nhà điều hành tắt trong khi quy trình máy chủ của chúng ta đang chạy trên đó chủ nhà.

Khi hệ thống Unix bị tắt, tiến trình `init` thư ứng gửi `SIGTERM` gửi tín hiệu đến tất cả các quy trình (chúng tôi có thể bắt được tín hiệu này), đợi một khoảng thời gian cố định (thường từ 5 đến 20 giây) rồi gửi tín hiệu `SIGKILL` (mà chúng tôi không thể bắt được) đến bất kỳ quy trình nào vẫn đang chạy. Điều này mang lại cho tất cả các tiến trình đang chạy một khoảng thời gian ngắn để dọn dẹp và chấm dứt. Nếu chúng tôi không bắt được `SIGTERM` và chấm dứt, máy chủ của chúng tôi sẽ bị chấm dứt bởi tín hiệu `SIGKILL`. Khi quá trình kết thúc, tắt cả

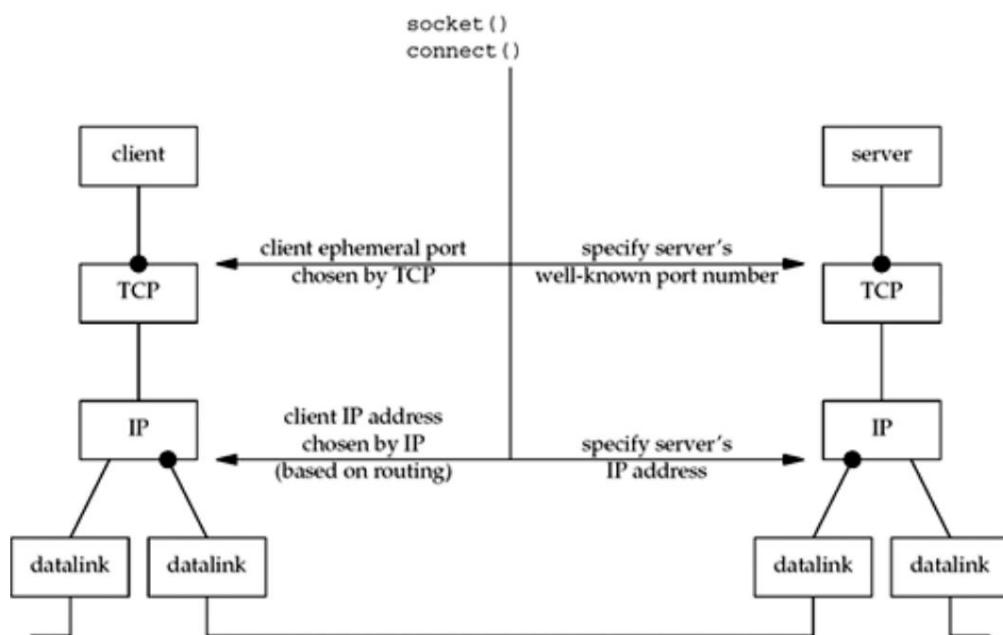
các bộ mô tả mở được đóng lại và sau đó chúng ta thực hiện theo trình tự các bước tương tự như đã thảo luận trong [Phần 5.12](#). Như đã nêu ở đó, chúng ta phải sử dụng chức năng `chọn` hoặc `thăm dò ý kiến` trong máy khách của mình để máy khách phát hiện việc chấm dứt quá trình máy chủ ngay khi nó xảy ra.

5.17 Tóm tắt ví dụ về TCP

Trước khi bắt kỳ máy khách và máy chủ TCP nào có thể giao tiếp với nhau, mỗi đầu phải chỉ định cặp ổ cắm cho kết nối: địa chỉ IP cục bộ, cổng cục bộ, địa chỉ IP nước ngoài và cổng nước ngoài. Trong [Hình 5.15](#), chúng tôi hiển thị bốn giá trị này dưới dạng dấu đầu dòng. Con số này là từ quản điểm của khách hàng. Địa chỉ IP nước ngoài và cổng nước ngoài phải được xác định trong cuộc gọi để `kết nối`. Hai giá trị cục bộ thường được kernel chọn như một phần của hàm `kết nối`. Máy khách có tùy chọn chỉ định một hoặc cả hai giá trị cục bộ bằng cách gọi `liên kết` trước khi `kết nối`, nhưng đây không phải là

chung.

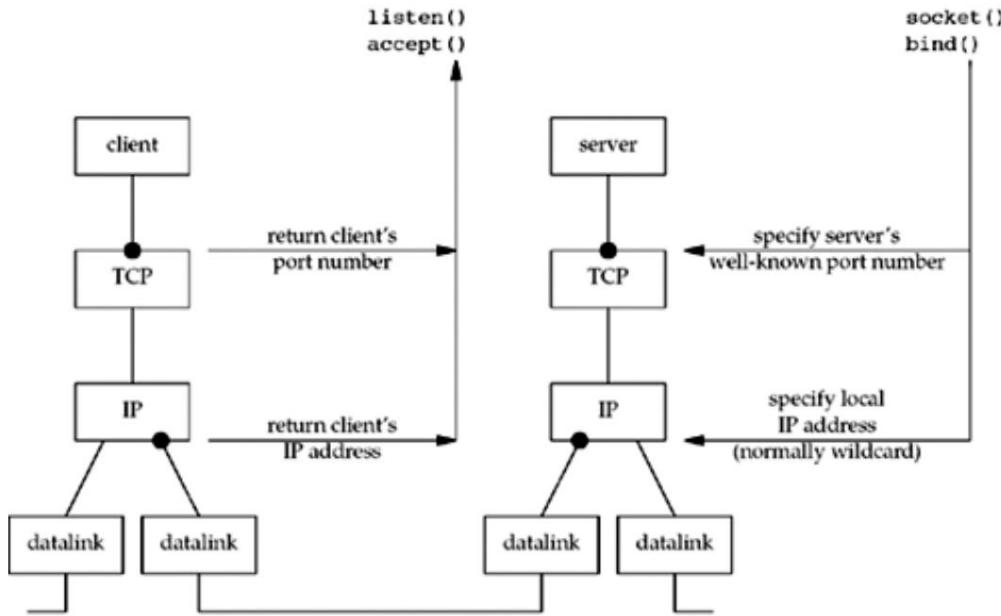
[Hình 5.15](#). Tóm tắt TCP client/server từ góc nhìn của client.



Như chúng tôi đã đề cập trong [Phần 4.10](#), máy khách có thể lấy hai giá trị cục bộ được kernel chọn bằng cách gọi `getsockname` sau khi kết nối được thiết lập.

[Hình 5.16](#) hiển thị bốn giá trị giống nhau, như từ góc nhìn của máy chủ.

[Hình 5.16](#). Tóm tắt về máy khách/máy chủ TCP từ góc nhìn của máy chủ.



Cổng cục bộ (cổng nổi tiếng của máy chủ) được chỉ định bằng [liên kết](#). Thông thường, máy chủ cũng chỉ định địa chỉ IP ký tự đại diện trong cuộc gọi này. Nếu máy chủ liên kết địa chỉ IP ký tự đại diện trên máy chủ nhiều trang, nó có thể xác định địa chỉ IP cục bộ bằng cách gọi `getsockname` sau khi kết nối được thiết lập ([Phần 4.10](#)). Hai giá trị ngoại đư ợc trả về máy chủ bằng cách [chấp nhận](#). Như chúng tôi đã đề cập [trong Phần 4.10](#), nếu một chương trình khác đư ợc [thực thi](#) bởi máy chủ gọi [chấp nhận](#), chương trình đó có thể gọi `getpeername` để xác định địa chỉ IP và cổng của máy khách, nếu cần.

5.18 Định dạng dữ liệu

Trong ví dụ của chúng tôi, máy chủ không bao giờ kiểm tra yêu cầu mà nó nhận đư ợc từ máy khách.

Máy chủ chỉ đọc tất cả dữ liệu xuyên suốt và bao gồm cả dòng mới và gửi lại cho máy khách, chỉ tìm kiếm dòng mới. Đây là một ngoại lệ, không phải quy tắc và thông thường chúng ta phải lo lắng về định dạng của dữ liệu đư ợc trao đổi giữa máy khách và máy chủ.

Ví dụ: Truyền chuỗi văn bản giữa Client và Server

Hãy sửa đổi máy chủ của chúng ta để nó vẫn đọc một dòng văn bản từ máy khách, như ng máy chủ hiện mong đợi dòng đó chứa hai số nguyên cách nhau bởi khoảng trắng và máy chủ trả về tổng của hai số nguyên đó. Chức năng [chính](#) của máy khách và máy chủ của chúng tôi vẫn giữ nguyên, cũng như hàm `str_cli` của chúng ta . Tất cả những thay đổi đó là `str_echo` của chúng tôi chức năng, mà chúng tôi thể hiện trong [hình 5.17](#).

[Hình 5.17](#) Hàm `str_echo` cộng hai số.

tcpcliserv/str_echo8.c

```

1 #bao gồm          "unp.h"
2 khoảng trắng
3 str_echo(int sockfd)
4 {
5     dài      arg1,      arg2;
6     cõ_t n;
7     dòng char[MAXLINE];
8
9     vì (      ; ; ) {
10         if ( (n = Readline(sockfd, line, MAXLINE)) == 0)
11             trở lại;           /*kết nối bị đóng bởi đầu bên kia */
12
13         if (sscanf(line, "%ld%ld", &arg1, &arg2) == 2)
14             snprintf(line, sizeof(line), "%ld\n", arg1 + arg2);
15         khác
16         snprintf(line, sizeof(line), "lỗi nhập\n");
17
18     }
19 }
```

11-14 Chúng ta gọi `sscanf` để chuyển đổi hai đối số từ chuỗi văn bản thành số nguyên dài, sau đó gọi `snprintf` để chuyển đổi kết quả thành chuỗi văn bản.

Máy khách và máy chủ mới này hoạt động tốt, bắt kể thứ tự byte của máy chủ máy khách và máy chủ.

Ví dụ: Truyền cấu trúc nhị phân giữa máy khách và máy chủ

Bây giờ chúng tôi sửa đổi máy khách và máy chủ của mình để chuyển các giá trị nhị phân qua ống cắm, thay vì các chuỗi văn bản. Chúng ta sẽ thấy rằng điều này không hoạt động khi máy khách và máy chủ chạy trên các máy chủ có thứ tự byte khác nhau hoặc trên các máy chủ không đồng ý về kích thước của một số nguyên dài ([Hình 1.17](#)).

Các chức năng **chính** của máy khách và máy chủ của chúng tôi không thay đổi. Chúng tôi xác định một cấu trúc cho hai đối số, một cấu trúc khác cho kết quả và đặt cả hai định nghĩa vào tiêu đề `sum.h` của chúng tôi, được hiển thị trong [Hình 5.18](#), [Hình 5.19](#) thể hiện hàm `str_cli`.

Hình 5.18 tiêu đề `sum.h`.

tcpcliserv/sum.h

```

1 đổi số cấu trúc {
2     arg1 dài;
3     arg2 dài;
4 };

```

```

5 kết quả cấu trúc {
6     dài         tổng;
7 };

```

Hình 5.19 Hàm str_cli gửi hai số nguyên nhị phân đến máy chủ.

tcpcliserv/str_cli09.c

```

1 #bao gồm      "unp.h"
2 #bao gồm      "tổng hợp.h"

3 khoảng trắng

4 str_cli(TẬP_TIN *fp, int sockfd)
5 {
6     ký tự      đư ờng dây gửi[MAXLINE];
7     struct lập luận;
8     kết quả kết quả cấu trúc;

9     while (Fgets(sendline, MAXLINE, fp) != NULL) {

10        if (sscanf(sendline, "%ld%ld", &args.arg1, &args.arg2) != 2) {
11            printf("đầu vào không hợp lệ: %s", sendline);
12            Tiếp tục;
13        }
14        Đã viết(sockfd, &args, sizeof(args));

15        if (Readn(sockfd, &result, sizeof(result)) == 0)
16            err_quit("str_cli: máy chủ đã kết thúc sớm");

17        printf("%ld\n", result.sum);
18    }
19 }

```

10-14 **sscanf** chuyển đổi hai đối số từ chuỗi văn bản thành chuỗi nhị phân và chúng tôi gọi **write** để gửi cấu trúc đến máy chủ.

15-17 Chúng ta gọi **readn** để đọc câu trả lời và in kết quả bằng **printf**.

[Hình 5.20](#) thể hiện hàm str_echo của chúng tôi .

Hình 5.20 hàm str_echo cộng hai số nguyên nhị phân.

tcpcliserv/str_ech09.c

```

1 #bao gồm           "unp.h"
2 #bao gồm           "tổng hợp.h"

3 khoảng trắng

4 str_echo(int sockfd)
5 {
6     cỡ_t n;
7     struct lập luận;
8     kết quả kết quả cấu trúc;

9     vì (      ;      ) {
10        if ( (n = Readn(sockfd, &args, sizeof(args))) == 0)
11            trở lại;           /*kết nối bị đóng bởi đầu bên kia */

12        result.sum = args.arg1 + args.arg2;
13        Đã viết(sockfd, &result, sizeof (kết quả));
14    }
15 }
```

9-14 Chúng ta đọc các đối số bằng cách gọi `readn`, tính toán và lưu trữ tổng, đồng thời gọi `ghi` để gửi lại cấu trúc kết quả.

Nếu chúng ta chạy máy khách và máy chủ trên hai máy có cùng kiến trúc, chẳng hạn như hai máy SPARC, thì mọi thứ đều hoạt động tốt. Đây là sự tương tác của khách hàng:

```

năng lưu lượng mặt trời % :tcpcli09 12.106.32.254
11 22
33
-11 -44
-55
```

chúng tôi gõ cái này

đây là câu trả lời của máy chủ

Nhưng khi máy khách và máy chủ ở trên hai máy có kiến trúc khác nhau (giả sử máy chủ nằm trên `freebsd` hệ thống SPARC cấp lớn và máy khách nằm trên hệ thống `linux Intel endian nhỏ`), thì nó không hoạt động.

```
linux% tcpccli09 206.168.112.96
1 2                                     chúng tôi gõ cái này
3                                     Và nó hoạt động
-22 -77                                     sau đó chúng ta gõ cái này
-16777314                                    và nó không hoạt động
```

Vấn đề là hai số nguyên nhị phân được máy khách gửi qua ô cắm ở định dạng endian nhỏ, như ng được máy chủ hiểu là số nguyên endian lớn. Chúng ta thấy rằng nó có vẻ hoạt động với số nguyên dương như ng không hoạt động với số nguyên âm (xem [Bài tập 5.8](#)). Thực sự có ba vấn đề tiềm ẩn với ví dụ này:

1. Các cách triển khai khác nhau lưu trữ số nhị phân ở các định dạng khác nhau. Các hầu hết các định dạng phổ biến là big-endian và little-endian, như chúng tôi đã mô tả trong [Phần 3.4](#).
2. Các cách triển khai khác nhau có thể lưu trữ cùng một kiểu dữ liệu C một cách khác nhau. Vì ví dụ, hầu hết các hệ thống Unix 32 bit sử dụng 32 bit cho thời gian dài như ng hệ thống 64 bit thường sử dụng 64 bit cho cùng một kiểu dữ liệu ([Hình 1.17](#)). Không có gì đảm bảo rằng short, int hoặc long có kích thước nhất định.
3. Các cách triển khai khác nhau sẽ đóng gói các cấu trúc khác nhau, tùy thuộc vào số bit được sử dụng cho các kiểu dữ liệu khác nhau và các hạn chế cẩn chỉnh của máy. Vì vậy, sẽ không bao giờ là khôn ngoan nếu gửi cấu trúc nhị phân qua một ô cắm.

Có hai giải pháp phổ biến cho vấn đề định dạng dữ liệu này:

1. Truyền tất cả dữ liệu số dưới dạng chuỗi văn bản. Đây là những gì chúng tôi đã làm [trong Hình 5.17](#). Cái này giả định rằng cả hai máy chủ đều có cùng một bộ ký tự.
2. Xác định rõ ràng định dạng nhị phân của các kiểu dữ liệu được hỗ trợ (số bit, big-hoặc little-endian) và chuyển đổi dữ liệu giữa máy khách và máy chủ ở định dạng này. Các gói RPC thường sử dụng kỹ thuật này. RFC 1832 [Srinivasan 1995] mô tả tiêu chuẩn Biểu diễn dữ liệu ngoài (XDR) được sử dụng với gói Sun RPC.

5.19 Tóm tắt

Phiên bản đầu tiên của máy khách/máy chủ echo của chúng tôi có tổng cộng khoảng 150 dòng (bao gồm cả chức năng [đọc](#) và [ghi](#)), nhưng cung cấp nhiều chi tiết để kiểm tra. Vấn đề đầu tiên chúng tôi gặp phải là zombie trẻ em và chúng tôi đã bắt được tín hiệu SIGCHLD để xử lý việc này. Sau đó, trình xử lý tín hiệu của chúng tôi được gọi là [waitpid](#) và chúng tôi đã chứng minh rằng chúng tôi phải gọi hàm này thay vì hàm [chờ](#) cũ hơn, vì tín hiệu Unix không được xếp hàng đợi. Điều này dẫn chúng ta đến một số chi tiết về xử lý tín hiệu POSIX (thông tin bổ sung về chủ đề này được cung cấp trong Chương 10 của APUE).

Vấn đề tiếp theo mà chúng tôi gặp phải là máy khách không được thông báo khi quá trình máy chủ kết thúc. Chúng tôi thấy rằng TCP của khách hàng của chúng tôi đã được thông báo, nhưng chúng tôi không nhận được thông báo đó vì chúng tôi đã bị chặn, đang chờ người dùng nhập dữ liệu. Chúng ta sẽ sử dụng [lựa chọn](#) hoặc hàm [thăm dò](#) trong [Chương 6](#) để xử lý tình huống này, bằng cách đợi bất kỳ một trong nhiều bộ mô tả sẵn sàng, thay vì chặn trên một bộ mô tả duy nhất.

Chúng tôi cũng phát hiện ra rằng nếu máy chủ lưu trữ gấp sự cố, chúng tôi sẽ không phát hiện ra điều này cho đến khi máy khách gửi dữ liệu đến máy chủ. Một số ứng dụng phải được biết về thực tế này sớm hơn; trong [Phần 7.5](#), chúng ta sẽ [xem xét tùy chọn](#) ở cẩm [SO_KEEPALIVE](#).

Ví dụ đơn giản của chúng tôi trao đổi các dòng văn bản, điều này không sao vì máy chủ không bao giờ nhìn vào các dòng nó lặp lại. Việc gửi dữ liệu số giữa máy khách và máy chủ có thể dẫn đến một loạt vấn đề mới, như được hiển thị.

Bài tập

[5.1](#) Xây dựng máy chủ TCP từ [Hình 5.2](#) và [5.3](#) và máy khách TCP từ [Hình 5.4](#) và [5.5](#). Khởi động máy chủ và sau đó [khởi động máy khách](#). Nhập một vài dòng để xác minh rằng máy khách và máy chủ có hoạt động không. Chấm dứt ứng dụng khách bằng cách nhập ký tự EOF của bạn và ghi lại thời gian. Sử dụng [netstat](#) trên máy chủ máy khách để xác minh rằng kết thúc kết nối của máy khách có chuyển sang trạng thái TIME_WAIT hay không. Thực thi [netstat](#) cứ sau năm giây hoặc lâu hơn để xem khi nào trạng thái TIME_WAIT kết thúc.

MSL cho việc triển khai này là gì?

[5.2](#) Điều gì xảy ra với máy khách/máy chủ echo của chúng tôi nếu chúng tôi chạy máy khách và chuyển hướng đầu vào tiêu chuẩn vào một tệp nhị phân?

[5.3](#) Sự khác biệt giữa máy khách/máy chủ echo của chúng tôi và việc sử dụng Telnet là gì client có thể liên lạc với máy chủ echo của chúng tôi không?

[5.4](#) Trong ví dụ ở [Phần 5.12](#), chúng tôi đã xác minh rằng hai phân đoạn đầu tiên của việc chấm dứt kết nối đã được gửi (FIN từ máy chủ sau đó được máy khách ACK) bằng cách xem xét trạng thái ở cẩm bằng [netstat](#). Hai phân đoạn cuối cùng có được trao đổi (FIN từ máy khách được máy chủ ACK) không? Nếu vậy thì khi nào, và nếu không thì tại sao?

[5.5](#) Điều gì xảy ra trong ví dụ được nêu ở [Phần 5.14](#) nếu giữa [Bước 2](#) và [3](#) chúng tôi khởi động lại ứng dụng máy chủ của mình trên máy chủ lưu trữ?

[5.6](#) Để xác minh những gì chúng tôi tuyên bố xảy ra với [SIGPIPE](#) trong [Phần 5.13](#), hãy sửa đổi [Hình 5.4](#) như sau: Viết bộ xử lý tín hiệu cho [SIGPIPE](#) chỉ in một thông báo và trả về. Thiết lập trình xử lý tín hiệu này trước khi gọi [connect](#).

Thay đổi số cổng của máy chủ thành 13, máy chủ ban ngày. Khi kết nối được thiết lập, **hãy ngủ** trong hai giây, **ghi** một vài byte vào ổ cắm, **ngủ** thêm hai giây nữa và **ghi** thêm vài byte vào ổ cắm. Chạy chương trình. Điều gì xảy ra?

5.7 Điều gì xảy ra trong [Hình 5.15](#) nếu địa chỉ IP của máy chủ được

được khách hàng chỉ định trong lệnh gọi kết **nối** có phải là địa chỉ IP được liên kết với liên kết dữ liệu ngoài cùng bên phải trên máy chủ, thay vì địa chỉ IP được liên kết với liên kết dữ liệu ngoài cùng bên trái trên máy chủ không?

5.8 Trong ví dụ đầu ra của chúng tôi từ [Hình 5.20](#), khi máy khách và máy chủ ở trên các hệ thống endian khác nhau, ví dụ này hoạt động với các số dư ơng nhỏ, nhưng không hoạt động với các số âm nhỏ. Tại sao? (Gợi ý: Vẽ hình các giá trị được trao đổi qua socket, tương tự như [Hình 3.9.](#))

5.9 Trong ví dụ ở [Hình 5.19](#) và [5.20](#), liệu chúng ta có thể giải quyết vấn đề thứ tự byte không? vấn đề bằng cách yêu cầu máy khách chuyển đổi hai đối số thành thứ tự byte mạng bằng cách sử dụng **htonl**, sau đó yêu cầu máy chủ gọi **ntohl** trên mỗi đối số trước khi thực hiện phép cộng và sau đó thực hiện chuyển đổi tương tự trên kết quả?

5.10 Điều gì xảy ra trong [Hình 5.19](#) và [5.20](#) nếu máy khách sử dụng SPARC lưu trữ giá trị **dài** ở dạng 32 bit, như máy chủ sử dụng Digital Alpha lưu trữ giá trị **dài** ở dạng 64 bit? Điều này có thay đổi nếu máy khách và máy chủ được hoán đổi giữa hai máy chủ này không?

5.11 Trong [Hình 5.15](#), chúng ta nói rằng địa chỉ IP của máy khách được IP chọn dựa trên lô trình. Điều đó có nghĩa là gì?

Chương 6. Ghép kênh I/O: 'chọn' và 'thăm dò ý kiến'

Chức năng

[Mục 6.1. Giới thiệu](#)

[Mục 6.2. Mô hình I/O](#)

[Mục 6.3. Chọn chức năng](#)

[Mục 6.4. Hàm str_cli \(Đã xem lại\)](#)

[Mục 6.5. Nhập và đêm hàng loạt](#)

[Mục 6.6. Chức năng tắt máy](#)

[Mục 6.7. Hàm str_cli \(Để đọc xem lại lần nữa\)](#)[Mục 6.8. Máy chủ TCP Echo \(Đã xem lại\)](#)[Mục 6.9. chức năng chọn lọc](#)[Mục 6.10. Chức năng thăm dò ý kiến](#)[Mục 6.11. Máy chủ TCP Echo \(Để đọc xem lại lần nữa\)](#)[Mục 6.12. Bản tóm tắt](#)[Bài tập](#)

6.1 Giới thiệu

Trong [Phần 5.12](#), chúng ta đã thấy máy khách TCP xử lý hai đầu vào cùng lúc: đầu vào tiêu chuẩn và ổ cảm TCP. Chúng tôi đã gặp phải sự cố khi máy khách bị chặn lệnh gọi tới `fgets` (trên đầu vào tiêu chuẩn) và quy trình máy chủ bị dừng. TCP máy chủ đã gửi FIN một cách chính xác đến TCP máy khách, nhưng vì quy trình máy khách bị chặn đọc từ đầu vào tiêu chuẩn nên nó không bao giờ nhìn thấy EOF cho đến khi đọc từ ổ cảm (có thể muộn hơn nhiều). Điều chúng ta cần là khả năng thông báo cho kernel biết rằng chúng ta muốn được thông báo nếu một hoặc nhiều điều kiện I/O sẵn sàng (tức là đầu vào đã sẵn sàng để đọc hoặc bộ mô tả có khả năng nhận nhiều đầu ra hơn). Khả năng này được gọi là ghép kênh I/O và được cung cấp bởi các chức năng `chọn` và `thăm dò`. Chúng tôi cũng sẽ đề cập đến một biến thể POSIX mới hơn của phiên bản trước, được gọi là `pselect`.

Một số hệ thống cung cấp những cách nâng cao hơn để các quy trình chờ danh sách sự kiện.

Thiết bị thăm dò ý kiến là một cơ chế để có thể cung cấp nhiều hình thức khác nhau bởi các nhà cung cấp khác nhau. Cơ chế này sẽ được mô tả trong [Chương 14](#).

Ghép kênh I/O thường được sử dụng trong các ứng dụng mạng sau đây
kịch bản:

- Khi máy khách đang xử lý nhiều bộ mô tả (thông thường là đầu vào tươn tác và ổ cảm mạng), nên sử dụng ghép kênh I/O. Đây là kịch bản chúng tôi đã mô tả trước đây.
- Có thể, nhưng hiếm, một client có thể xử lý nhiều socket cùng một lúc.
Chúng tôi sẽ trình bày một ví dụ về điều này bằng cách sử dụng [select](#) trong [Phần 16.5](#) trong ngữ cảnh của một máy khách Web.
- Nếu máy chủ TCP xử lý cả ổ cảm nghe và ổ cảm được kết nối của nó, I/O
ghép kênh thường được sử dụng, như chúng tôi sẽ trình bày trong [Phần 6.8](#).
- Nếu một máy chủ xử lý cả TCP và UDP, ghép kênh I/O thường được sử dụng. Chúng tôi
sẽ trình bày một ví dụ về điều này trong [Phần 8.15](#).

- Nếu một máy chủ xử lý nhiều dịch vụ và có lẽ nhiều giao thức (ví dụ, daemon `inetd` mà chúng tôi sẽ mô tả trong [Phần 13.5](#)), ghép kênh I/O thường được sử dụng.

Ghép kênh I/O không giới hạn trong lập trình mạng. Nhiều ứng dụng không cần thiết nhận thấy cần có những kỹ thuật này.

6.2 Mô hình I/O

Trước khi mô tả `select` và `poll`, chúng ta cần quay lại và nhìn vào bức tranh lớn hơn, xem xét những khác biệt cơ bản trong năm mô hình I/O có sẵn trong Unix:

- chặn I/O
- I/O không chặn
- Ghép kênh I/O ([chọn](#) và [thăm dò](#))
- I/O điều khiển bằng tín hiệu ([SIGIO](#))
- I/O không đồng bộ (hàm POSIX `aio_functions`)

Bạn có thể muốn đọc lướt phần này trong lần đọc đầu tiên và sau đó tham khảo lại khi gặp các mô hình I/O khác nhau được mô tả chi tiết hơn trong các chương sau.

Như chúng tôi trình bày trong tất cả các ví dụ trong phần này, thông thường có hai giai đoạn riêng biệt cho một thao tác đầu vào:

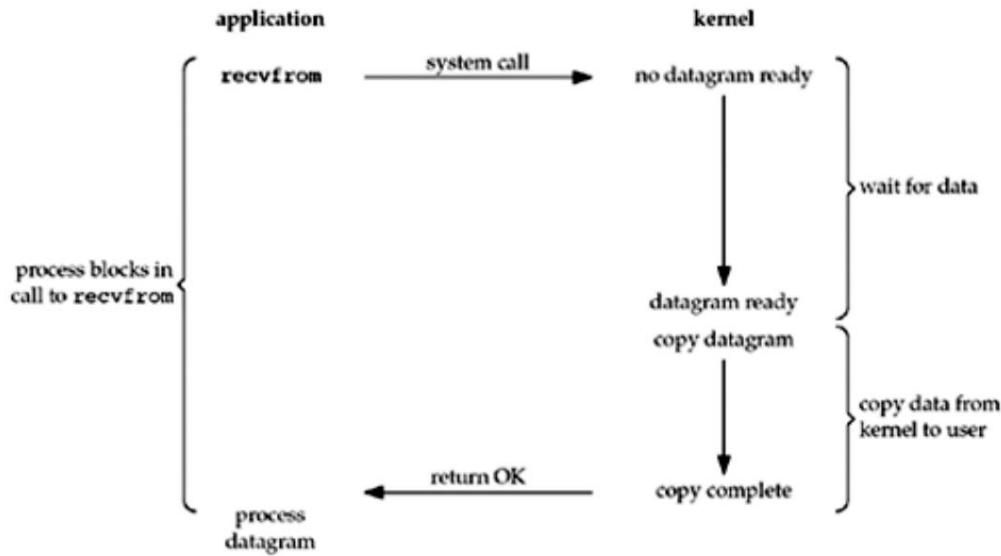
1. Chờ dữ liệu sẵn sàng
2. Sao chép dữ liệu từ kernel vào tiến trình

Đối với thao tác đầu vào trên ổ cảm, bước đầu tiên thường bao gồm việc chờ dữ liệu đến mạng. Khi gói đến, nó sẽ được sao chép vào bộ đệm trong kernel. Bước thứ hai là sao chép dữ liệu này từ bộ đệm của kernel vào bộ đệm ứng dụng của chúng ta.

Chặn mô hình I/O

Mô hình I/O phổ biến nhất là mô hình I/O chặn, mô hình này chúng tôi đã sử dụng cho tất cả các ví dụ cho đến nay trong văn bản. Theo mặc định, tất cả các ổ cảm đều bị chặn. Sử dụng ổ cảm datagram cho các ví dụ của chúng tôi, chúng tôi có kịch bản được gọi [hiển thị trong Hình 6.1](#).

Hình 6.1. Mô hình chặn I/O.



Chúng tôi sử dụng UDP cho ví dụ này thay vì TCP vì với UDP, khái niệm dữ liệu "sẵn sàng" để đọc rất đơn giản: toàn bộ gói dữ liệu đã được nhận hoặc chưa. Với TCP, mọi chuyện trở nên phức tạp hơn khi có các biến bổ sung như dấu mục nút ớc thấp của ổ cắm phát huy tác dụng.

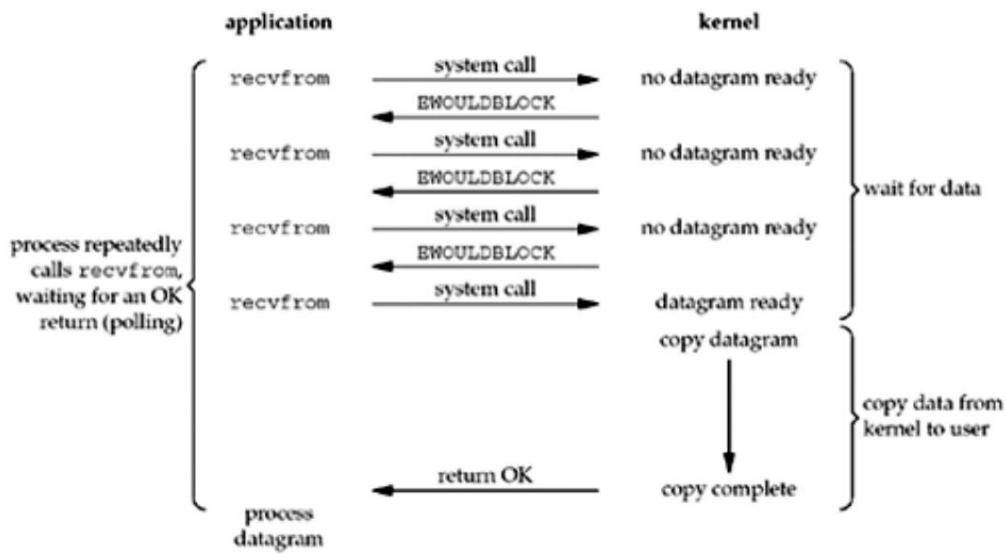
Trong các ví dụ ở phần này, chúng tôi cũng coi `recvfrom` là lệnh gọi hệ thống vì chúng tôi đang phân biệt giữa ứng dụng của mình và kernel. Bất kể `recvfrom` được triển khai như thế nào (như một lệnh gọi hệ thống trên hạt nhân có nguồn gốc từ Berkeley hoặc như một hàm gọi lệnh gọi hệ thống `getmsg` trên hạt nhân System V), thường có một sự chuyển đổi từ chạy trong ứng dụng sang chạy trong hạt nhân, một thời gian sau đó sẽ quay lại ứng dụng.

Trong [Hình 6.1](#), tiến trình gọi `recvfrom` và lệnh gọi hệ thống không quay trở lại cho đến khi datagram đến và được sao chép vào bộ đệm ứng dụng của chúng ta hoặc xảy ra lỗi. Lỗi phổ biến nhất là cuộc gọi hệ thống bị gián đoạn bởi tín hiệu, như chúng tôi đã mô tả trong [Phần 5.9](#). Chúng tôi nói rằng quy trình của chúng tôi bị chặn toàn bộ thời gian kể từ khi nó gọi `recvfrom` cho đến khi nó quay trở lại. Khi `recvfrom` trả về thành công, ứng dụng của chúng tôi sẽ xử lý datagram.

Mô hình I/O không chặn

Khi chúng tôi đặt một ổ cắm ở chế độ không chặn, chúng tôi sẽ thông báo cho kernel "khi một thao tác I/O mà tôi yêu cầu không thể hoàn thành nếu không đặt quy trình ở chế độ ngủ, dừng đặt quy trình ở chế độ ngủ mà thay vào đó hãy trả về lỗi." Chúng ta sẽ mô tả I/O không chặn trong [Chương 16](#), như [Hình 6.2](#) trình bày tóm tắt về ví dụ mà chúng ta đang xem xét.

Hình 6.2. Mô hình I/O không chặn.



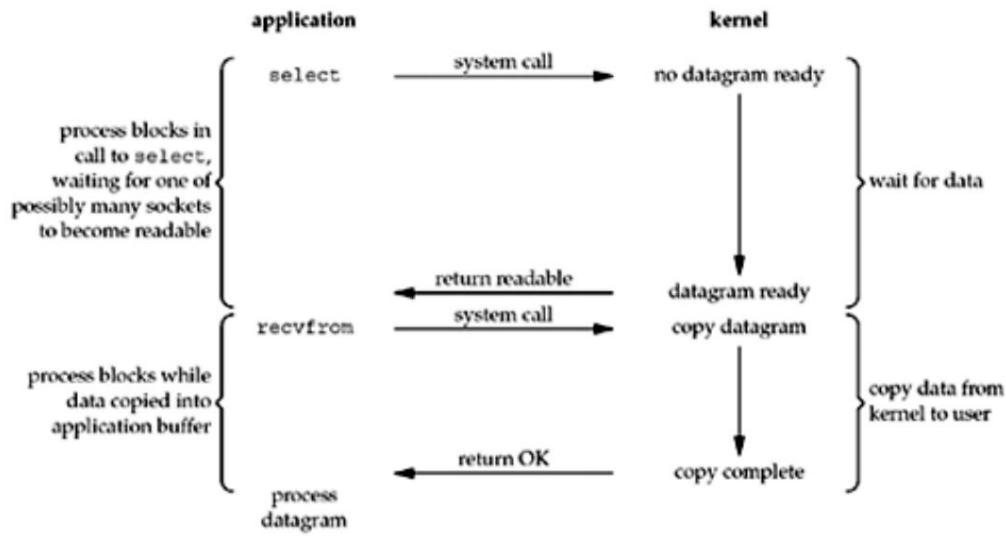
Ba lần đầu tiên chúng tôi gọi `recvfrom`, không có dữ liệu nào để trả về, vì vậy kernel ngay lập tức trả về lỗi `EWOULDBLOCK`. Lần thứ tư chúng ta gọi `recvfrom`, một datagram đã sẵn sàng, nó được sao chép vào bộ đệm ứng dụng của chúng ta và `recvfrom` trở lại thành công. Sau đó chúng tôi xử lý dữ liệu.

Khi một ứng dụng nằm trong một vòng lặp gọi `recvfrom` trên một bộ mô tả không chặn như thế này, thì nó được gọi là bô phiếu. Ứng dụng liên tục thăm dò kernel để xem liệu một số thao tác đã sẵn sàng chưa. Điều này thường gây lãng phí thời gian của CPU, nhưng mô hình này đôi khi cũng gấp phải, thường là trên các hệ thống dành riêng cho một chức năng.

Mô hình ghép kênh I/O

Với ghép kênh I/O, chúng tôi gọi `chọn` hoặc `thăm dò` và chặn một trong hai lệnh gọi hệ thống này, thay vì chặn trong lệnh gọi hệ thống I/O thực tế. [Hình 6.3](#) là bản tóm tắt mô hình ghép kênh I/O.

Hình 6.3. Mô hình ghép kênh I/O.



Chúng tôi chặn cuộc gọi để **chọn**, chờ ở cảm gói dữ liệu có thể đọc được. Khi **select** trả về rằng socket có thể đọc được, chúng ta sẽ gọi **recvfrom** để sao chép datagram vào bộ đệm ứng dụng của mình.

So sánh [Hình 6.3](#) với [Hình 6.1](#), dù ờng như không có bất kỳ lợi thế nào và trên thực tế, có một bất lợi nhỏ vì việc sử dụng **select** yêu cầu hai lệnh gọi hệ thống thay vì một. Nhưng ưu điểm của việc sử dụng **select**, mà chúng ta sẽ thấy ở phần sau của chương này, là chúng ta có thể đợi nhiều bộ mô tả sẵn sàng.

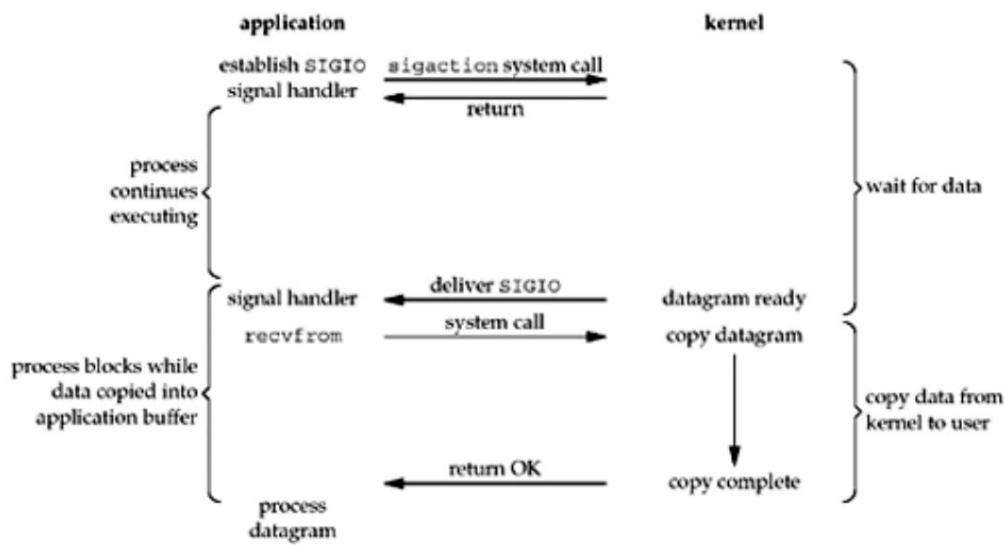
Một mô hình I/O có liên quan chặt chẽ khác là sử dụng đa luồng với việc chặn I/O. Mô hình đó rất giống với mô hình được mô tả ở trên, ngoại trừ việc thay vì sử dụng **lýa chọn** để chặn trên nhiều bộ mô tả tệp, chương trình sử dụng nhiều luồng (một luồng cho mỗi bộ mô tả tệp) và mỗi luồng sau đó có thể tự do gọi các lệnh gọi hệ thống chặn như **recvfrom**.

Mô hình I/O điều khiển bằng tín hiệu

Chúng tôi cũng có thể sử dụng tín hiệu, yêu cầu kernel thông báo cho chúng tôi bằng tín hiệu **SIGIO** khi bộ mô tả sẵn sàng.

Chúng tôi gọi đây là I/O điều khiển bằng tín hiệu và hiển thị tóm tắt về nó trong [Hình 6.4](#).

Hình 6.4. Mô hình I/O điều khiển bằng tín hiệu.



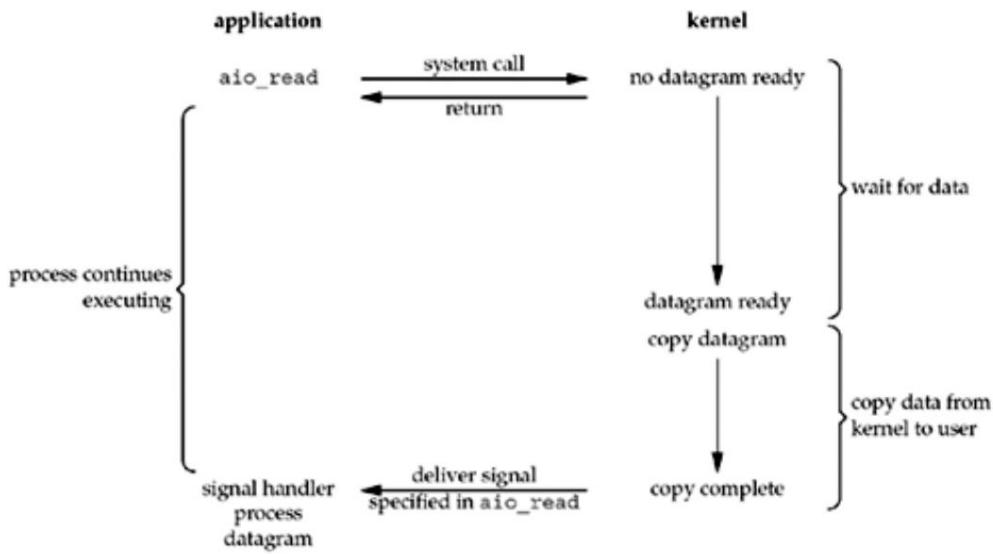
Trước tiên, chúng tôi kích hoạt ô cắm cho I/O điều khiển bằng tín hiệu (như chúng tôi sẽ mô tả trong [Phần 25.2](#)) và cài đặt bộ xử lý tín hiệu bằng cách sử dụng lệnh gọi hệ thống `sigaction`. Kết quả trả về từ lệnh gọi hệ thống này là ngay lập tức và quá trình của chúng tôi vẫn tiếp tục; nó không bị chặn. Khi datagram đã sẵn sàng để đọc, tín hiệu `SIGIO` sẽ được tạo cho quy trình của chúng tôi. Chúng ta có thể đọc datagram từ bộ xử lý tín hiệu bằng cách gọi `recvfrom` và sau đó thông báo cho vòng lặp chính rằng dữ liệu đã sẵn sàng để được xử lý (đây là những gì chúng ta sẽ làm trong [Phần 25.3](#)), hoặc chúng ta có thể thông báo cho vòng lặp chính và để nó đọc datagram.

Bất kể chúng ta xử lý tín hiệu như thế nào, ưu điểm của mô hình này là chúng ta không bị chặn trong khi chờ datagram đến. Vòng lặp chính có thể tiếp tục thực thi và chỉ chờ bộ xử lý tín hiệu thông báo rằng dữ liệu đã sẵn sàng để xử lý hoặc gói dữ liệu đã sẵn sàng để đọc.

Mô hình I/O không đồng bộ

I/O không đồng bộ được xác định bởi đặc tả POSIX và nhiều khác biệt trong các hàm thời gian thực xuất hiện trong các tiêu chuẩn khác nhau kết hợp với nhau để tạo thành đặc tả POSIX hiện tại đã được điều chỉnh. Nói chung, các chức năng này hoạt động bằng cách yêu cầu kernel bắt đầu hoạt động và thông báo cho chúng tôi khi toàn bộ hoạt động (bao gồm cả việc sao chép dữ liệu từ kernel vào bộ đệm của chúng tôi) hoàn tất. Sự khác biệt chính giữa mô hình này và mô hình I/O điều khiển bằng tín hiệu trong phần trước là với I/O điều khiển tín hiệu, kernel cho chúng ta biết khi nào một hoạt động I/O có thể được bắt đầu, nhưng với I/O không đồng bộ, kernel cho chúng ta biết khi nào thao tác I/O hoàn tất. Chúng tôi đưa ra một ví dụ trong [Hình 6.5](#).

Hình 6.5. Mô hình I/O không đồng bộ.



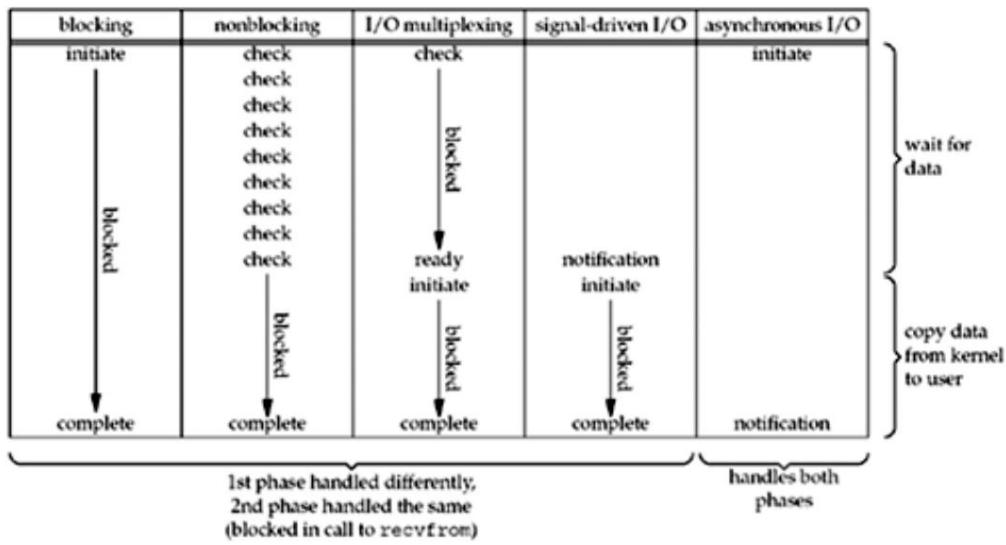
Chúng tôi gọi `aio_read` (các hàm I/O không đồng bộ POSIX bắt đầu bằng `aio_` hoặc `lio_`) và truyền cho kernel bộ mô tả, con trỏ bộ đệm, kích thước bộ đệm (ba đối số giống nhau cho **việc đọc**), offset tệp (tương tự như `lseek`) và cách thông báo chúng tôi khi toàn bộ hoạt động hoàn tất. Cuộc gọi hệ thống này sẽ trả về ngay lập tức và quy trình của chúng tôi không bị chặn trong khi chờ I/O hoàn tất. Trong ví dụ này, chúng tôi giả sử rằng chúng tôi yêu cầu kernel tạo ra một số tín hiệu khi thao tác hoàn tất. Tín hiệu này không được tạo cho đến khi dữ liệu được sao chép vào bộ đệm ứng dụng của chúng tôi, khác với mô hình I/O điều khiển bằng tín hiệu.

Theo văn bản này, rất ít hệ thống hỗ trợ I/O không đồng bộ POSIX. Ví dụ: chúng tôi không chắc chắn liệu hệ thống có hỗ trợ ở cắm hay không. Việc chúng tôi sử dụng nó ở đây là một ví dụ để so sánh với mô hình I/O điều khiển bằng tín hiệu.

So sánh các mô hình I/O

[Hình 6.6](#) là sự so sánh của năm mô hình I/O khác nhau. Nó cho thấy rằng sự khác biệt chính giữa bốn mô hình đầu tiên là giai đoạn đầu tiên, vì giai đoạn thứ hai trong bốn mô hình đầu tiên là như nhau: quá trình bị chặn trong lệnh gọi tới `recvfrom` trong khi dữ liệu được sao chép từ kernel sang bộ đệm của người gọi. Tuy nhiên, I/O không đồng bộ xử lý cả hai giai đoạn và khác với bốn giai đoạn đầu tiên.

Hình 6.6. So sánh năm mô hình I/O.



I/O đồng bộ so với I/O không đồng bộ

POSIX định nghĩa hai thuật ngữ này như sau:

- Thao tác I/O đồng bộ khiếu quá trình yêu cầu bị chặn cho đến khi thao tác I/O đó hoàn thành.
- Thao tác I/O không đồng bộ không làm cho quá trình yêu cầu bị dừng bị chặn.

Bằng cách sử dụng các định nghĩa này, bốn mô hình I/O đầu tiên-chặn, không chặn, ghép kênh I/O và I/O điều khiển bằng tín hiệu-tắt cả đều đồng bộ vì hoạt động I/O thực tế (`recvfrom`) chặn quá trình. Chỉ mô hình I/O không đồng bộ khớp với định nghĩa I/O không đồng bộ.

6.3 Chức năng 'chọn'

Hàm này cho phép tiến trình hứa ứng dẫn kernel đợi bất kỳ một trong nhiều sự kiện xảy ra và chỉ đánh thức tiến trình khi một hoặc nhiều sự kiện này xảy ra hoặc khi một khoảng thời gian xác định đã trôi qua.

Ví dụ: chúng ta có thể gọi `select` và yêu cầu kernel chỉ trả về khi:

- Bất kỳ bộ mô tả nào trong bộ {1, 4, 5} đều sẵn sàng để đọc
- Bất kỳ bộ mô tả nào trong tập {2, 7} đều sẵn sàng để ghi
- Bất kỳ bộ mô tả nào trong tập {1, 4} đều có điều kiện ngoại lệ đang chờ xử lý
- 10,2 giây đã trôi qua

Nghĩa là, chúng ta cho kernel biết bộ mô tả nào chúng ta quan tâm (để đọc, viết hoặc một điều kiện ngoại lệ) và thời gian chờ đợi. Các bộ mô tả mà chúng ta quan tâm không bị giới hạn ở các ô cắm; bất kỳ bộ mô tả nào cũng có thể được kiểm tra bằng cách sử dụng `select`.

Việc triển khai có nguồn gốc từ Berkeley luôn cho phép ghép kênh I/O với bất kỳ bộ mô tả nào. SVR3 ban đầu giới hạn việc ghép kênh I/O ở các bộ mô tả là thiết bị STREAMS ([Chương 31](#)), như ng hạn chế này đã được loại bỏ với SVR4.

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *ngoại trừset, const struct timeval *timeout);
Trả về: số lư ợng dư ợng của các bộ mô tả sẵn sàng, 0 khi hết thời gian chờ, -1 khi có lỗi
```

Chúng ta bắt đầu mô tả hàm này bằng đối số cuối cùng của nó, đối số này cho hạt nhân biết phải đợi bao lâu để một trong các bộ mô tả được chỉ định sẵn sàng. Một *khoảng thời gian* cấu trúc chỉ định số giây và micro giây.

```
cấu trúc thời gian {
    trong tv_sec;           /* giây */
    trong tv_usec; };       /* micro giây */
```

Có ba khả năng:

1. Đợi mãi— Chỉ trả về khi một trong các bộ mô tả được chỉ định sẵn sàng cho I/O. Đối với điều này, chúng tôi chỉ định đối số hết thời gian chờ là con trỏ null.
2. Đợi đến một khoảng thời gian cố định— Quay lại khi một trong các khoảng thời gian được chỉ định trong bộ mô tả đã sẵn sàng cho I/O, như ng không đợi vượt quá số giây và micro giây được chỉ định trong cấu trúc *thời gian* được chỉ ra bởi thời gian chờ lý lẽ.
3. Đừng chờ đợi— Hãy quay lại ngay sau khi kiểm tra phần mô tả. Điều này được gọi là bỏ phiếu. Để chỉ định điều này, đối số hết thời gian chờ phải trả đến *khoảng thời gian* cấu trúc và giá trị bộ định thời (số giây và micro giây do cấu trúc chỉ định) phải bằng 0.

Quá trình chờ đợi trong hai trường hợp đầu tiên thường bị gián đoạn nếu quá trình bắt đư ợc tín hiệu và trả về từ bộ xử lý tín hiệu.

Các hạt nhân có nguồn gốc từ Berkeley không bao giờ tự động khởi động lại **lựa chọn** (tr. 527 của TCPv2), trong khi SVR4 sẽ làm như vậy nếu cờ **SA_RESTART** được chỉ định khi cài đặt bộ xử lý tín hiệu. Điều này có nghĩa là để có tính di động, chúng tôi phải chuẩn bị sẵn sàng cho **việc chọn** trả về lỗi **EINTR** nếu chúng tôi đang bắt được tín hiệu.

Mặc dù cấu trúc **thời gian** cho phép chúng ta chỉ định độ phân giải tính bằng micro giây, nhưng độ phân giải thực tế được hạt nhân hỗ trợ thường thấp hơn. Ví dụ: nhiều nhân Unix làm tròn giá trị thời gian chờ lên bởi số của 10 ms. Ngoài ra còn có độ trễ lập lịch liên quan, nghĩa là phải mất một thời gian sau khi hết giờ trước khi kernel lên lịch cho quá trình này chạy.

Trên một số hệ thống, **việc chọn** sẽ không thành công với **EINVAL** nếu truwong **tv_sec** trong thời gian chờ vượt quá 100 triệu giây. Tuy nhiên, đó là khoảng thời gian chờ rất lớn (trên ba năm) và có thể không hữu ích lắm, như ng vẫn đề là cấu trúc **thời gian** có thể biểu thị các giá trị không được lựa chọn hỗ trợ.

Vòng loại **const** trên đối số hết thời gian chờ có nghĩa là nó không được sửa đổi bằng cách **chọn** khi trả về. Ví dụ: nếu chúng tôi chỉ định giới hạn thời gian là 10 giây và **chọn** trả về trước khi hết giờ với một hoặc nhiều bộ mô tả sẵn sàng hoặc có lỗi **EINTR**, thì cấu trúc **thời gian** không được cập nhật với số giây còn lại khi hàm trả lại. Nếu muốn biết giá trị này, chúng ta phải lấy thời gian hệ thống trước khi gọi **select**, sau đó lặp lại khi nó trả về và trừ đi hai (bất kỳ chương trình mạnh mẽ nào cũng sẽ tính đến việc thời gian hệ thống có thể được điều chỉnh bởi quản trị viên hoặc bởi thỉnh thoảng có một daemon như **ntpd**).

Một số phiên bản Linux sửa đổi cấu trúc **thời gian**. Do đó, để có tính di động, giả sử cấu trúc **thời gian** không được xác định khi trả về và khởi tạo nó trước mỗi lệnh gọi tới **select**. POSIX chỉ định vòng loại **const**.

Ba đối số ở giữa, **readset**, **writeset** và ngoại trừ, chỉ định các bộ mô tả mà chúng ta muốn kernel kiểm tra các điều kiện đọc, ghi và ngoại lệ. Hiện chỉ có hai điều kiện ngoại lệ được hỗ trợ:

1. Sự xuất hiện của dữ liệu ngoài băng tần cho ổ cẩm. Chúng tôi sẽ mô tả điều này nhiều hơn chi tiết ở [Chương 24](#).
2. Sự hiện diện của thông tin trạng thái điều khiển được đọc từ phía chính của thiết bị đầu cuối giả đã được đưa vào chế độ gói. Chúng tôi không nói về thiết bị đầu cuối giả trong cuốn sách này.

Vấn đề thiết kế là làm thế nào để xác định một hoặc nhiều giá trị mô tả cho mỗi đối số trong số ba đối số này. **select** sử dụng các bộ mô tả, thường là một mảng các số nguyên, với mỗi bit trong mỗi số nguyên tương ứng với một bộ mô tả. Ví dụ: sử dụng số nguyên 32 bit, phần tử đầu tiên của mảng tương ứng với các bộ mô tả từ 0 đến 31, phần tử thứ hai của mảng tương ứng với các bộ mô tả từ 32 đến 63, v.v. Tất cả

chi tiết triển khai không liên quan đến ứng dụng và bị ẩn trong kiểu dữ liệu `fd_set` và bốn macro sau:

```
void FD_ZERO(fd_set *fdset); /* xóa tất cả các bit trong fdset */

void FD_SET(int fd, fd_set *fdset); /* bật bit cho fd trong fdset */

void FD_CLR(int fd, fd_set *fdset); /* tắt bit cho fd trong fdset */

int FD_ISSET(int fd, fd_set *fdset); /* có phải là bit dành cho fd trong fdset không? */
```

Chúng tôi phân bổ một bộ mô tả của kiểu dữ liệu `fd_set`, chúng tôi đặt và kiểm tra các bit trong tập hợp bằng cách sử dụng các macro này và chúng tôi cũng có thể gán nó cho một bộ mô tả khác qua dấu bằng (=) trong C.

Những gì chúng tôi đang mô tả, một mảng các số nguyên sử dụng một bit cho mỗi bộ mô tả, chỉ là một cách khả thi để triển khai **lựa chọn**. Tuy nhiên, người ta thường coi các bộ mô tả riêng lẻ trong một bộ mô tả là các bit, như trong "bật bit cho bộ mô tả nghe trong tập đọc".

Chúng ta sẽ thấy [trong Phần 6.10](#) rằng hàm `thamd` sử dụng một cách biểu diễn hoàn toàn khác: một mảng các cấu trúc có độ dài thay đổi với một cấu trúc cho mỗi bộ mô tả.

Ví dụ: để xác định một biến kiểu `fd_set` và sau đó bật các bit cho các bộ mô tả 1, 4 và 5, chúng ta viết

```
fd_set đặt lại;

FD_ZERO(&rset); /* khởi tạo tập hợp: tắt tất cả các bit */

FD_SET(1, &rset); /* bật bit cho fd 1 */

FD_SET(4, &rset); /* bật bit cho fd 4 */

FD_SET(5, &rset); /* bật bit cho fd 5 */
```

Điều quan trọng là phải khởi tạo tập hợp, vì các kết quả không thể đoán trước có thể xảy ra nếu tập hợp chưa được phân bổ dưới dạng biến tự động và không được khởi tạo.

Bất kỳ đối số nào trong số ba đối số ở giữa để **chọn**, tập hợp đọc, tập ghi hoặc tập ngoại lệ đều có thể được chỉ định làm con trả null nếu chúng ta không quan tâm đến điều kiện đó. Thật vậy, nếu cả ba con trả đều rỗng thì chúng ta có bộ đếm thời gian có độ chính xác cao hơn chế độ ngủ Unix thông thường chức năng (ngủ trong bộ số của một giây). Chức năng `thamd` ý kiến cung cấp tính năng tự

chức năng. Hình C.9 và C.10 của APUE hiển thị hàm `sleep_us` được triển khai bằng cách sử dụng cả `select` và `poll` ở chế độ ngủ trong bộ số của một micro giây.

Đối số `maxfdp1` chỉ định số lượng bộ mô tả cần kiểm tra. Giá trị của nó là bộ mô tả tối đa được kiểm tra cộng với một (do đó tên của chúng tôi là `maxfdp1`). Các bộ mô tả 0, 1, 2, từ trên xuống và bao gồm `maxfdp1-1` đã được kiểm tra.

Hàng số `FD_SETSIZE`, được xác định bằng cách bao gồm `<sys/select.h>`, là số lượng bộ mô tả trong kiểu dữ liệu `fd_set`. Giá trị của nó thường là 1024, nhưng rất ít chương trình sử dụng nhiều bộ mô tả như vậy. Đối số `maxfdp1` buộc chúng ta phải tính toán bộ mô tả lớn nhất mà chúng ta quan tâm và sau đó cho kernel biết giá trị này. Ví dụ: với mã truy cập đó giá trị là 6. Lý do nó là 6 chứ không phải 5 là vì chúng tôi đang chỉ định số lượng bộ mô tả, không phải giá trị lớn nhất và bộ mô tả bắt đầu từ 0.

Lý do tồn tại của lập luận này, cùng với gánh nặng tính toán giá trị của nó, hoàn toàn là vì tính hiệu quả. Mặc dù mỗi `fd_set` có chỗ cho nhiều bộ mô tả, thường là 1.024, nhưng con số này nhiều hơn số lượng được sử dụng bởi một quy trình thông thường. Hạt nhân đạt được hiệu quả bằng cách không sao chép các phần không cần thiết của bộ mô tả giữa tiến trình và hạt nhân và bằng cách không kiểm tra các bit luôn bằng 0 (Phần 16.13 của TCPv2).

`select` sửa đổi các bộ mô tả được chỉ ra bởi `readset`, `writeset` và ngoại trừ con trỏ. Ba đối số này là đối số kết quả giá trị. Khi gọi hàm, chúng ta chỉ định giá trị của các bộ mô tả mà chúng ta quan tâm và khi trả về, kết quả sẽ cho biết những bộ mô tả nào đã sẵn sàng. Chúng tôi sử dụng macro `FD_ISSET` khi quay lại để kiểm tra bộ mô tả cụ thể trong cấu trúc `fd_set`. Bất kỳ bộ mô tả nào chưa sẵn sàng khi trả về sẽ bị xóa bit tương ứng trong bộ mô tả. Để xử lý vấn đề này, chúng tôi bật tắt cả các bit mà chúng tôi quan tâm trong tất cả các bộ mô tả mỗi khi chúng tôi gọi `select`.

Hai lỗi lập trình phổ biến nhất khi sử dụng `select` là quên thêm một vào số bộ mô tả lớn nhất và quên rằng bộ mô tả là đối số kết quả giá trị. Lỗi thứ hai dẫn đến `việc chọn` được gọi với bit được đặt thành 0 trong bộ mô tả, khi chúng tôi cho rằng bit đó là 1.

Giá trị trả về từ hàm này cho biết tổng số bit đã sẵn sàng trên tất cả các bộ mô tả. Nếu giá trị bộ định thời hết hạn truy cập khi bất kỳ bộ mô tả nào sẵn sàng thì giá trị 0 sẽ được trả về. Giá trị trả về là -1 cho biết có lỗi (điều này có thể xảy ra, ví dụ: nếu chức năng bị gián đoạn do tín hiệu bắt được).

Các bản phát hành đầu tiên của SVR4 có lỗi trong quá trình triển khai lựa chọn: Nếu cùng một bit được bật trong nhiều bộ, giả sử bộ mô tả đã sẵn sàng cho cả đọc và ghi, thì nó chỉ được tính một lần. Các bản phát hành hiện tại đã sửa lỗi này.

Trong những điều kiện nào thì bộ mô tả đã sẵn sàng?

Chúng ta đã nói về việc chờ đợi một bộ mô tả sẵn sàng cho I/O (đọc hoặc ghi) hoặc có một điều kiện ngoại lệ đang chờ xử lý trên nó (dữ liệu ngoài băng tần). Mặc dù khả năng đọc và ghi là rõ ràng đối với các bộ mô tả như các tệp thông thường, nhưng chúng ta phải cụ thể hơn về các điều kiện khiến [việc chọn trả về "sẵn sàng"](#) cho các ô cắm (Hình 16.52 của TCPv2).

1. Ô cắm sẵn sàng để đọc nếu bất kỳ điều kiện nào trong bốn điều kiện sau là đúng:

Một. Số byte dữ liệu trong bộ đệm nhận ô cắm lớn hơn hoặc bằng kích thước hiện tại của dấu mực nước thấp cho bộ đệm nhận ô cắm. Thao tác đọc trên ô cắm sẽ không bị chặn và sẽ trả về giá trị lớn hơn 0 (tức là dữ liệu đã sẵn sàng để đọc).

Chúng ta có thể đặt vạch nước thấp này bằng tùy chọn ô cắm [SO_RCVLOWAT](#).

Nó mặc định là 1 cho các ô cắm TCP và UDP.

b. Nửa đọc của kết nối bị đóng (tức là kết nối TCP

đã nhận được FIN). Thao tác đọc trên ô cắm sẽ không bị chặn và sẽ trả về 0 (tức là EOF).

c. Ô cắm là ô cắm nghe và số lượng hoàn thành

kết nối là khác không. Việc [chấp nhận](#) trên ô cắm nghe thưòng sẽ không bị chặn, mặc dù chúng tôi sẽ mô tả điều kiện định thời trong [Phần 16.6](#)
theo đó [chấp nhận](#) có thể chặn.

d. Một lỗi ô cắm đang chờ xử lý. Thao tác đọc trên ô cắm sẽ không chặn và sẽ trả về lỗi (-1) với [lỗi](#) được đặt thành tình trạng lỗi cụ thể. Những lỗi đang chờ xử lý này cũng có thể được tìm nạp và xóa bằng cách gọi [getsockopt](#) và chỉ định tùy chọn ô cắm [SO_ERROR](#).

2. Ô cắm sẵn sàng để ghi nếu bất kỳ điều kiện nào trong bốn điều kiện sau là đúng:

Một. Số byte trống có sẵn trong bộ đệm gửi ô cắm là

lớn hơn hoặc bằng kích thước hiện tại của dấu mực nước thấp đối với bộ đệm gửi ô cắm và : (i) ô cắm đã được kết nối hoặc (ii) ô cắm không yêu cầu kết nối (ví dụ: UDP). Điều này có nghĩa là nếu chúng ta đặt ô cắm ở chế độ không chặn ([Chú ý](#) 16), thao tác ghi sẽ không chặn và sẽ trả về giá trị dương (ví dụ: số byte được lớp vận [chuyển chấp nhận](#)). Chúng ta có thể đặt vạch nước thấp này bằng tùy chọn ô cắm [SO_SNDDLOWAT](#). Dấu hiệu mức nước thấp này thưòng được mặc định là 2048 đối với các ô cắm TCP và UDP.

b. Một nửa ghi của kết nối đã bị đóng. Một thao tác ghi trên

socket sẽ tạo [SIGPIPE](#) ([Phần 5.12](#)).

c. Ô cắm sử dụng [kết nối](#) không chặn đã hoàn thành kết nối,
hoặc [kết nối](#) không thành công.

d. Một lỗi ô cắm đang chờ xử lý. Thao tác ghi trên ô cắm sẽ không bị chặn và sẽ trả về lỗi (-1) với [lỗi](#) được đặt thành tình trạng lỗi cụ thể. Những lỗi đang chờ xử lý này cũng có thể được tìm nạp và xóa bằng cách gọi [getsockopt](#) với tùy chọn ô cắm [SO_ERROR](#).

3. Ô cắm có tình trạng ngoại lệ đang chờ xử lý nếu có dữ liệu ngoài băng tần cho ô cắm hoặc ô cắm vẫn ở mức ngoài băng tần. (Chúng tôi sẽ mô tả dữ liệu ngoài băng trong [Chú ý](#) 24.)

Định nghĩa của chúng tôi về "có thẻ đọc" và "có thẻ ghi" được lấy trực tiếp từ các macro `có thẻ đọc` và `có thẻ ghi` của kernel trên trang 530-531 của TCPv2.

Tương tự, định nghĩa của chúng tôi về "điều kiện ngoại lệ" cho ô cắm là từ chức năng `soo_select` trên cùng các trang này.

Lưu ý rằng khi xảy ra lỗi trên ô cắm, nó được đánh dấu là có thẻ đọc và ghi được bằng `cách chọn`.

Mục đích của việc nhận và gửi các dấu hiệu ít nút là cung cấp cho ứng dụng quyền kiểm soát lưu lượng dữ liệu phải có để đọc hoặc lưu lượng dung lượng trống phải có để ghi trừ khi `lựa chọn` trả về trạng thái có thẻ đọc hoặc ghi. Ví dụ: nếu chúng tôi biết rằng ứng dụng của chúng tôi không có tác dụng gì trừ khi có ít nhất 64 byte dữ liệu, chúng tôi có thể đặt dấu nhận ít nút thành 64 để ngăn `việc chọn`

khỏi việc đánh thức chúng tôi nếu có ít hơn 64 byte sẵn sàng để đọc.

Miễn là mức nút sắp hết cho ô cắm UDP nhỏ hơn kích thước bộ đệm gửi (phải luôn là mỗi quan hệ mặc định), ô cắm UDP luôn có thẻ ghi vì không cần kết nối.

Hình 6.7 tóm tắt các điều kiện vừa được mô tả khi triển khai sẵn sàng để chọn.

Hình 6.7. Tóm tắt các điều kiện khi triển khai sẵn sàng để chọn.

Condition	Readable?	Writable?	Exception?
Data to read	•		
Read half of the connection closed	•		
New connection ready for listening socket	•		
Space available for writing		•	
Write half of the connection closed		•	
Pending error	•	•	
TCP out-of-band data			•

Số lưu lượng mô tả tối đa để chọn

Chúng tôi đã nói trước đó rằng hầu hết các ứng dụng không sử dụng nhiều bộ mô tả. Chẳng hạn, rất hiếm khi tìm thấy một ứng dụng sử dụng hàng trăm bộ mô tả. Tuy nhiên, các ứng dụng như vậy vẫn tồn tại và chúng thường sử dụng `tính năng chọn` để ghép các bộ mô tả. Khi `lựa chọn` được thiết kế ban đầu, HĐH thường có giới hạn trên số lưu lượng mô tả tối đa cho mỗi quy trình (giới hạn 4.2BSD là 31) và `chọn` chỉ sử dụng cùng giới hạn này. Tuy nhiên, các phiên bản hiện tại của Unix cho phép số lưu lượng bộ mô tả hầu như không giới hạn trên mỗi quy trình (thường chỉ bị giới hạn bởi dung lượng bộ nhớ và bất kỳ giới hạn quản trị nào), vì vậy câu hỏi đặt ra là: Điều này ảnh hưởng đến `việc chọn` như thế nào?

Nhiều cách triển khai có các khai báo tương tự như sau, được lấy từ tiêu đề 4.4BSD `<sys/types.h>` :

```
/*
 * Chọn sử dụng mặt nạ bit của bộ mô tả tệp trong thời gian dài. Các macro này
 * thao tác các trang bit như vậy (macro hệ thống tập tin sử dụng ký tự).
 *
 * FD_SETSIZE có thể được người dùng xác định, nhưng giá trị mặc định ở đây sẽ
 * đủ cho hầu hết các mục đích sử dụng.
 */
#ifndef FD_SETSIZE
#define FD_SETSIZE      256
#endif
```

Điều này khiến chúng tôi nghĩ rằng chúng tôi chỉ có thể `#define FD_SETSIZE` thành một số giá trị lớn hơn trước khi đưa tiêu đề này vào để tăng kích thước của bộ mô tả được sử dụng bởi [phản chọn](#).
Thật không may, điều này thường không hoạt động.

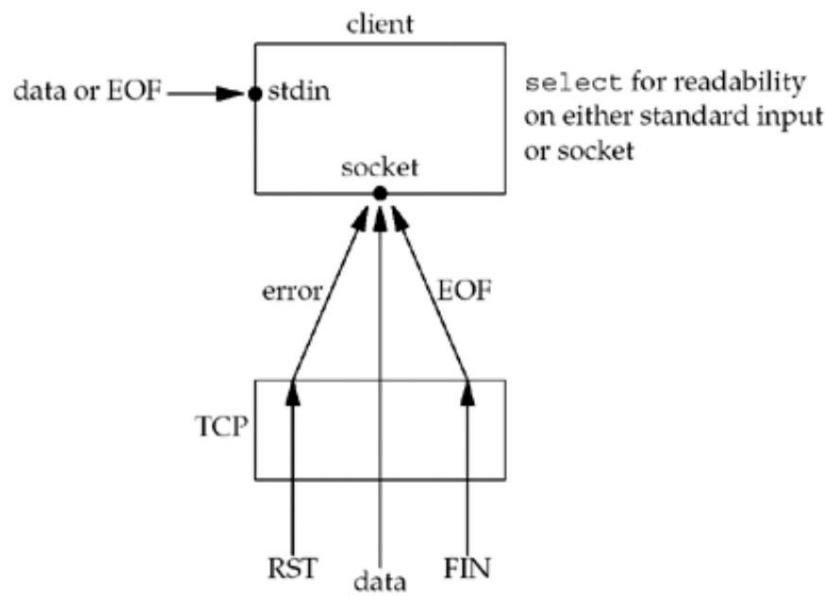
Để xem điều gì sai, hãy lưu ý rằng Hình 16.53 của TCPv2 khai báo ba bộ mô tả trong hạt nhân và cũng sử dụng định nghĩa `FD_SETSIZE` của hạt nhân làm giới hạn trên. Cách duy nhất để tăng kích thước của bộ mô tả là tăng giá trị của `FD_SETSIZE` và sau đó biên dịch lại kernel. Thay đổi giá trị mà không biên dịch lại kernel là không đủ.

Một số nhà cung cấp đang thay đổi cách triển khai [lựa chọn](#) để cho phép quy trình xác định `FD_SETSIZE` thành giá trị lớn hơn giá trị mặc định. BSD/OS đã thay đổi cách triển khai kernel để cho phép các bộ mô tả lớn hơn và nó cũng cung cấp bốn `FD_XXX` mới
macro để phân bổ và thao tác động các tập hợp lớn hơn này. Tuy nhiên, từ quan điểm về tính di động, hãy cẩn thận khi sử dụng các bộ mô tả lớn.

6.4 Hàm 'str_cli' (Đã xem lại)

Bây giờ chúng ta có thể viết lại hàm `str_cli` từ [Phần 5.5](#), ~~lần này bằng cách sử dụng `select`~~, để chúng ta được thông báo ngay khi quá trình máy chủ kết thúc. Vấn đề với phiên bản trước đó là chúng tôi có thể bị chặn lệnh gọi tới `fgets` khi có điều gì đó xảy ra trên ổ cảm. Thay vào đó, phiên bản mới của chúng tôi chặn lệnh gọi tới `select`, chờ đầu vào tiêu chuẩn hoặc ổ cảm có thể đọc được được. [Hình 6.8](#) cho thấy các điều kiện khác nhau được xử lý bằng lệnh gọi `select` của chúng ta.

[Hình 6.8](#). Các điều kiện được xử lý bằng cách chọn trong `str_cli`.



Ba điều kiện đư ợc xử lý với socket:

1. Nếu TCP ngang hàng gửi dữ liệu, socket sẽ có thể đọc đư ợc và kết quả **đọc** sẽ trả về lớn hơn 0 (tức là số byte dữ liệu).
2. Nếu TCP ngang hàng gửi FIN (quá trình ngang hàng kết thúc), thì ô cắm sẽ trở thành có thể đọc đư ợc và **đọc** trả về 0 (EOF).
3. Nếu TCP ngang hàng gửi RST (máy chủ ngang hàng đã gặp sự cố và khởi động lại), ô cắm sẽ có thể đọc đư ợc, **đọc** trả về -1 và **errno** chứa mã lỗi cụ thể.

Hình 6.9 hiển thị mã nguồn của phiên bản mới này.

Hình 6.9 Triển khai hàm str_cli bằng cách sử dụng select (đư ợc cải thiện trong Hình 6.13).

chọn/strcliselect01.c

```

1 #include "unp.h"

2 khoảng trống

3 str_cli(TÂP_TIN *fp, int sockfd)
4 {
5     int      maxfdp1;
6     fd_set đặt lại;
7     char sendline[MAXLINE], recvline[MAXLINE];

    FD_ZERO(&rset);

```

```

9     vì (      ;   )   {
10    FD_SET(fileno(fp), &rset);
11    FD_SET(sockfd, &rset);
12    maxfdp1 = max(fileno(fp), sockfd) + 1;
13    Chọn(maxfdp1, &rset, NULL, NULL, NULL);

14    if (FD_ISSET(sockfd, &rset)) { /* socket có thể đọc đư ợc */
15        if (Readline(sockfd, recvline, MAXLINE) == 0)
16            err_quit("str_cli: máy chủ đã kết thúc sớm");
17        Fputs(recvline, stdout);
18    }

19    if (FD_ISSET(fileno(fp), &rset)) { /* đầu vào có thể đọc đư ợc */
20        if (Fgets(sendline, MAXLINE, fp) == NULL)
21            trở lại;           /* tất cả đã đư ợc làm xong */
22        Đã viết(sockfd, sendline, strlen(sendline));
23    }
24}
25}

```

Chọn cuộc gọi

8-13 Chúng tôi chỉ cần một bộ mô tả để kiểm tra khả năng đọc. Bộ này đư ợc khởi tạo bởi `FD_ZERO` và sau đó hai bit đư ợc bật bằng cách sử dụng `FD_SET`: bit tương ứng với con trỏ tệp I/O tiêu chuẩn, `fp`, và bit tương ứng với socket, `sockfd`. Hàm `fileno` chuyển đổi một con trỏ tệp I/O tiêu chuẩn thành bộ mô tả tương ứng của nó. `select` (và `thêm dò ý kiến`) chỉ hoạt động với các bộ mô tả.

`select` đư ợc gọi sau khi tính toán mức tối đa của hai bộ mô tả. Trong cuộc gọi, con trỏ tập hợp ghi và con trỏ tập hợp ngoại lệ đều là con trỏ null. Đối số cuối cùng (giới hạn thời gian) cũng là một con trỏ rỗng vì chúng ta muốn chặn cuộc gọi cho đến khi có thứ gì đó sẵn sàng.

Xử lý ở cắm có thể đọc đư ợc

14-18 Nếu, khi quay về từ `phản chọn`, ở cắm có thể đọc đư ợc, dòng phản hồi đư ợc đọc với `dòng đọc` và đầu ra bằng `fput`.

Xử lý đầu vào có thể đọc đư ợc

19-23 Nếu đầu vào tiêu chuẩn có thể đọc đư ỢC, một dòng sẽ đư ỢC `fget` đọc và ghi vào ở cắm bằng cách `ghi`.

Lưu ý rằng bốn hàm I/O giống nhau đư ỢC sử dụng như trong [Hình 5.5](#), `fgets`, `write`, `readline` và `fputs`, như ng thứ tự luồng trong hàm đã thay đổi. Thay vì

của luồng hàm được điều khiển bởi lệnh gọi `fgets`, thì giờ đây nó được điều khiển bởi lệnh gọi `select`. Chỉ với một vài dòng mã bổ sung trong [Hình 6.9](#), so với [Hình 5.5](#), chúng tôi đã bổ sung rất nhiều vào độ mạnh mẽ của ứng dụng khách của mình.

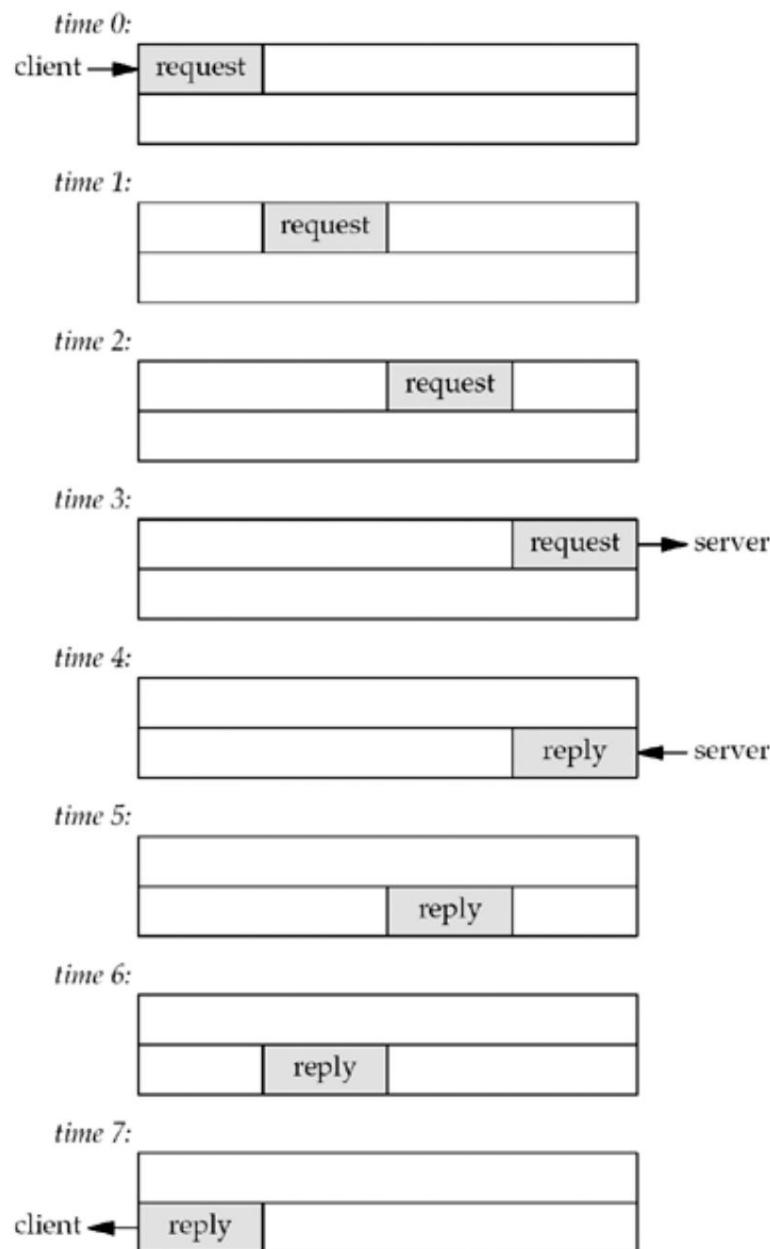
6.5 Nhập và lưu vào bộ đệm hàng loạt

Thật không may, hàm `str_cli` của chúng tôi vẫn chưa chính xác. Đầu tiên, hãy quay lại phiên bản gốc của chúng ta, [Hình 5.5](#). Nó hoạt động ở chế độ dừng và chờ, rất phù hợp cho việc sử dụng tương tác: Nó gửi một dòng đến máy chủ và sau đó chờ phản hồi. Khoảng thời gian này là một RTT cộng với thời gian xử lý của máy chủ (gần bằng 0 đối với máy chủ echo đơn giản). Do đó, chúng tôi có thể ước tính sẽ mất bao lâu để một số dòng nhất định được phản hồi nếu chúng tôi biết RTT giữa máy khách và máy chủ.

Chương trình `ping` là một cách dễ dàng để đo RTT. Nếu chúng tôi chạy `ping` đến máy chủ `connix.com` từ máy chủ `Solaris` thì RTT trung bình trên 30 lần đo là 175 mili giây. Trang 89 của TCPv1 cho thấy các phép đo `ping` này dành cho gói dữ liệu IP có độ dài là 84 byte. Nếu chúng tôi lấy 2.000 dòng đầu tiên của tệp `termcap` Solaris thì kích thước gói kết quả là 98.349 byte, trung bình là 49 byte trên mỗi dòng. Nếu chúng ta cộng kích thước của tiêu đề IP (20 byte) và tiêu đề TCP (20), phân đoạn TCP trung bình sẽ có kích thước khoảng 89 byte, gần bằng kích thước gói `ping`. Do đó, chúng tôi có thể ước tính rằng tổng thời gian đóng hộp sẽ vào khoảng 350 giây cho 2.000 dòng (2.000×0,175 giây). Nếu chúng ta chạy ứng dụng khách TCP echo từ [Chương 5](#), thời gian thực tế là khoảng 354 giây, rất gần với ước tính của chúng tôi.

Nếu chúng ta coi mạng giữa máy khách và máy chủ là một đường ống song công hoàn toàn, với các yêu cầu đi từ máy khách đến máy chủ và trả lời theo hướng ngược lại, thì [Hình 6.10](#) hiển thị chế độ dừng và chờ của chúng ta.

[Hình 6.10](#). Dòng thời gian của chế độ dừng và chờ: đầu vào tư ứng tác.



Một yêu cầu được khách hàng gửi vào thời điểm 0 và chúng tôi giả sử RTT là 8 đơn vị thời gian.

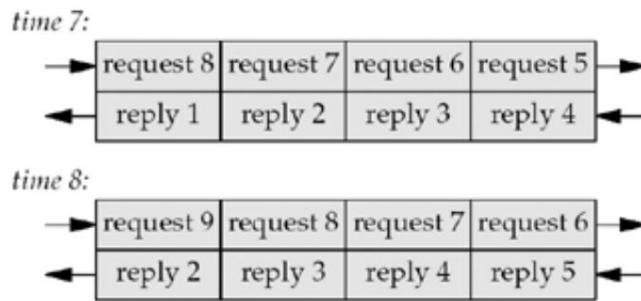
Phản hồi được gửi vào lúc 4 sẽ được nhận vào lúc 7. Chúng tôi cũng giả định rằng không có thời gian xử lý của máy chủ và kích thước của yêu cầu giống với phản hồi. Chúng tôi chỉ hiển thị các gói dữ liệu giữa máy khách và máy chủ, bỏ qua các xác nhận TCP cũng đang truyền qua mạng.

Vì có độ trễ giữa việc gửi gói và gói đó đến đầu kia của đường ống và vì đường ống ở chế độ song công hoàn toàn trong ví dụ này, chúng tôi chỉ sử dụng 1/8 công suất của đường ống. Chế độ dừng chờ này phù hợp với đầu vào tư duy tác, nhưng vì ứng dụng khách của chúng tôi đọc từ đầu vào tiêu chuẩn và ghi vào đầu ra tiêu chuẩn, và vì việc chuyển hư ống đầu vào và đầu ra trong hệ vỏ Unix là chuyện bình thường nên chúng tôi có thể dễ dàng chạy ứng dụng khách của mình trong một chế độ hàng loạt. Tuy nhiên, khi chúng ta chuyển hư ống đầu vào và đầu ra,

tệp đầu ra thu được luôn nhỏ hơn tệp đầu vào (và chúng phải giống hệt nhau đối với máy chủ echo).

Để xem điều gì đang xảy ra, hãy nhận ra rằng ở chế độ hàng loạt, chúng tôi có thể tiếp tục gửi yêu cầu nhanh nhất có thể mà mạng có thể chấp nhận chúng. Máy chủ xử lý chúng và gửi lại các câu trả lời với tốc độ như nhau. Điều này dẫn đến đường ống đầy ở thời điểm thứ 7, như trong [Hình 6.11.](#)

Hình 6.11. Làm đầy đường ống giữa máy khách và máy chủ: chế độ hàng loạt.



Ở đây, chúng tôi giả định rằng sau khi gửi yêu cầu đầu tiên, chúng tôi ngay lập tức gửi một yêu cầu khác, rồi một yêu cầu khác. Chúng tôi cũng giả định rằng chúng tôi có thể tiếp tục gửi yêu cầu nhanh nhất mà mạng có thể chấp nhận, cùng với việc xử lý các câu trả lời nhanh như mạng cung cấp chúng.

Có rất nhiều điểm phức tạp liên quan đến luồng dữ liệu số lượng lớn của TCP mà chúng tôi đang bỏ qua ở đây, chẳng hạn như thuật toán khởi động chậm, giới hạn tốc độ dữ liệu được gửi trên kết nối mới hoặc không hoạt động, cũng như các ACK trả về. Tất cả đều được đề cập trong Chương 20 của TCPv1.

Để xem vấn đề với hàm `str_cli` đã sửa đổi của chúng tôi trong [Hình 6.9](#), giả sử rằng tệp đầu vào chỉ chứa chín dòng. Dòng cuối cùng được gửi vào thời điểm thứ 8, như trong [Hình 6.11](#). Như ng chúng tôi không thể đóng kết nối sau khi viết yêu cầu này vì vẫn còn các yêu cầu và phản hồi khác trong đường ống. Nguyên nhân của vấn đề là do chúng ta xử lý EOF trên đầu vào: Hàm quay trở lại hàm `chính`, sau đó hàm này kết thúc.

Như ng ở chế độ hàng loạt, EOF trên đầu vào không có nghĩa là chúng tôi đã đọc xong từ ổ cắm; vẫn có thể có yêu cầu trên đường đến máy chủ hoặc phản hồi trên đường quay lại từ máy chủ.

Điều chúng ta cần là một cách để đóng một nửa kết nối TCP. Nghĩa là, chúng tôi muốn gửi FIN đến máy chủ, thông báo rằng chúng tôi đã gửi xong dữ liệu nhưng vẫn để bộ mô tả ổ cắm mở để đọc. Điều này được thực hiện bằng chức năng `tắt máy`, được mô tả trong phần tiếp theo.

Nói chung, việc đệm cho hiệu năng làm tăng thêm độ phức tạp cho ứng dụng mạng và mã trong [Hình 6.9](#) gập phải sự phức tạp này. Hãy xem xét trường hợp khi một số

dòng đầu vào có sẵn từ đầu vào tiêu chuẩn. `select` sẽ kiểm mã ở dòng 20 đọc đầu vào bằng `fgets` và điều đó sẽ đọc các dòng có sẵn vào bộ đệm được sử dụng bởi `stdio`. Tuy nhiên, `fgets` chỉ trả về một dòng duy nhất và để lại mọi dữ liệu còn lại trong bộ đệm `stdio`. Mã ở dòng 22 của [Hình 6.9](#) ghi dòng đơn đó vào máy chủ và sau đó `select` được gọi lại để chờ làm việc nhiều hơn, ngay cả khi có thêm dòng để sử dụng trong bộ đệm `stdio`. Nguyên nhân của việc này là `chọn`

không biết gì về bộ đệm được sử dụng bởi `stdio`-nó sẽ chỉ hiển thị khả năng đọc từ quan điểm của lệnh gọi hệ thống `đọc` chứ không phải các lệnh gọi như `fgets`. Vì lý do này, việc trộn `stdio` và `select` được coi là rất dễ xảy ra lỗi và chỉ nên thực hiện hết sức cẩn thận.

Vấn đề tương tự tồn tại với lệnh gọi `readline` trong ví dụ ở [Hình 6.9](#).

Thay vì dữ liệu bị ẩn khỏi `phản chọn` trong bộ đệm `stdio`, nó bị ẩn trong bộ đệm `của dòng đọc`. Hãy nhớ lại rằng trong [Phần 3.9](#), chúng tôi đã cung cấp `một hàm cho phép hiển thị bộ đệm của dòng đọc`, do đó, một giải pháp khả thi là sửa đổi mã của chúng tôi để sử dụng hàm đó trước khi gọi `select` để xem dữ liệu đã được đọc chưa như ngưng chưa được sử dụng hay chưa. Nhưng một lần nữa, sự phức tạp nhanh chóng vượt quá tầm kiểm soát khi chúng ta phải xử lý trường hợp bộ đệm `dòng đọc` chứa một phần dòng (có nghĩa là chúng ta vẫn cần đọc thêm) cũng như khi nó chưa một hoặc nhiều dòng hoàn chỉnh (mà chúng ta có thể sử dụng).).

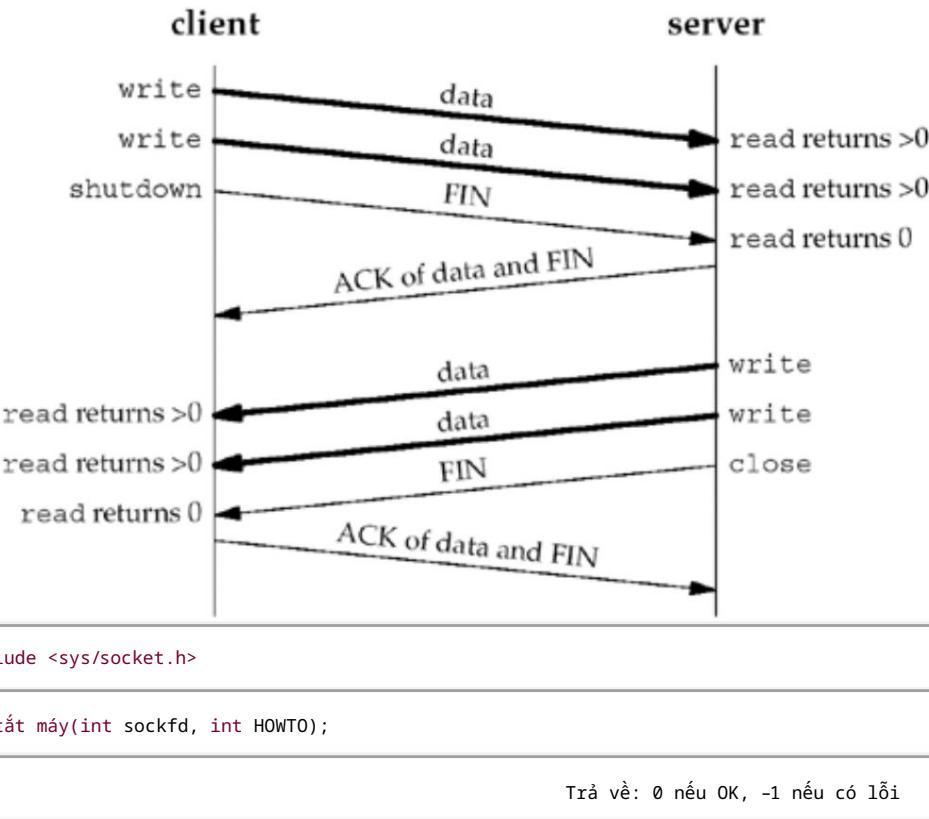
Chúng tôi sẽ giải quyết những vấn đề về bộ đệm này trong phiên bản cải tiến của `str_cli` được hiển thị trong [Mục 6.7](#).

6.6 Chức năng 'tắt máy'

Cách thông thường để chấm dứt kết nối mạng là gọi hàm `close`. Nhưng, có hai hạn chế khi đóng có thể tránh được khi tắt máy:

1. `close` giảm số lượng tham chiếu của bộ mô tả và chỉ đóng socket nếu số lượng đạt đến 0. Chúng ta đã nói về điều này trong [Phần 4.8](#). Khi tắt máy, chúng ta có thể bắt đầu trình tự `chấm dứt kết nối` thông thường của TCP (bốn phân đoạn bắt đầu bằng FIN trong [Hình 2.5](#)), bắt kè số lượng tham chiếu.
2. đóng chấm dứt cả hai hướng truyền dữ liệu, đọc và ghi. Vì kết nối TCP là kết nối song công hoàn toàn nên đôi khi chúng ta muốn thông báo với đầu bên kia rằng chúng ta đã gửi xong, mặc dù đầu đó có thể có nhiều dữ liệu hơn để gửi cho chúng ta. Đây là tình huống chúng ta gặp phải trong phần trước với dữ liệu đầu vào hàng loạt cho hàm `str_cli` của chúng ta. [Hình 6.12](#) cho thấy các lệnh gọi hàm diễn hình trong kịch bản này.

[Hình 6.12](#). Gọi tắt máy để đóng một nửa kết nối TCP.



Hành động của hàm phụ thuộc vào giá trị của đối số HOWTO .

SHUT_RD Nửa đọc của kết nối đã bị đóng– Không thể nhận thêm dữ liệu

trên ống cắm và mọi dữ liệu hiện có trong bộ đệm nhận ống cắm sẽ bị loại bỏ. Quá trình này không thể phát hành bất kỳ chức năng đọc nào trên ống cắm nữa. Mọi dữ liệu nhận được sau lệnh gọi ống cắm TCP này đều được xác nhận và sau đó bị loại bỏ một cách âm thầm.

Theo mặc định, mọi thứ được ghi vào ống cắm định tuyến ([Chương 18](#)) sẽ lặp lại dữ ống cắm đầu vào có thể có cho tất cả ống cắm định tuyến trên máy chủ. Một số chương trình gọi **tắt máy** với đối số thứ hai là **SHUT_RD** để ngăn chặn việc sao chép vòng lặp.

Một cách khác để ngăn chặn việc sao chép vòng lặp này là xóa tùy chọn ống cắm **SO_USELOOPBACK** .

SHUT_WR Nửa ghi của kết nối đã bị đóng– Trong trường hợp TCP, điều này được gọi là nửa đóng (Mục 18.5 của TCPv1). Mọi dữ liệu hiện có trong bộ đệm gửi ống cắm sẽ được gửi, theo sau là trình tự kết thúc kết nối thông thường của TCP. Như chúng tôi đã đề cập trước đó, việc đóng nửa ghi này được thực hiện bắt kè số tham chiếu của bộ mô tả ống cắm hiện có lớn hơn 0 hay không. Quá trình này không thể phát hành bất kỳ chức năng ghi nào trên ống cắm nữa.

SHUT_RDWR Nửa đọc và nửa ghi của kết nối đều bị đóng– Điều này tương đương với việc gọi **tắt máy** hai lần: đầu tiên với **SHUT_RD** và sau đó với

SHUT_RD Nửa đọc của kết nối đã bị đóng— Không thể nhận thêm dữ liệu trên ổ cắm và mọi dữ liệu hiện có trong bộ đệm nhận ở cắm sẽ bị loại bỏ. Quá trình này không thể phát hành bất kỳ chức năng đọc nào trên ổ cắm nữa. Mọi dữ liệu nhận được sau lệnh gọi ổ cắm TCP này đều được xác nhận và sau đó bị loại bỏ một cách âm thầm.

Theo mặc định, mọi thứ được ghi vào ổ cắm định tuyến ([Chương 18](#)) sẽ lặp lại dữ ới dạng đầu vào có thể có cho tất cả các ổ cắm định tuyến trên máy chủ. Một số chương trình gọi tắt máy với đối số thứ hai là **SHUT_RD** để ngăn chặn việc sao chép vòng lặp. Một cách khác để ngăn chặn việc sao chép vòng lặp này là xóa tùy chọn ổ cắm **SO_USELOOPBACK**.

SHUT_WR.

[Hình 7.12](#) sẽ tắt các khả năng khác nhau có sẵn cho tiến trình bằng cách gọi tắt máy và đóng.

Hoạt động đóng phụ thuộc vào giá trị của tùy chọn ổ cắm **SO_LINGER**.

Ba tên **SHUT_XXX** được xác định bởi đặc tả POSIX. Các giá trị điển hình cho đối số hibernating mà bạn sẽ gặp sẽ là 0 (đóng nửa đọc), 1 (đóng nửa đọc) và 2 (đóng nửa đọc và nửa ghi).

6.7 Hàm 'str_cli' (Được xem lại lần nữa)

[Hình 6.13](#) cho thấy phiên bản sửa đổi (và chính xác) của hàm **str_cli**. Phiên bản này sử dụng tính năng chọn và tắt máy. Cái trú ớc thông báo cho chúng tôi ngay khi máy chủ đóng phần cuối của kết nối và kết nối sau cho phép chúng tôi xử lý đầu vào hàng loạt một cách chính xác. Phiên bản này cũng loại bỏ mã tập trung vào dòng và thay vào đó hoạt động trên bộ đệm, loại bỏ những lo ngại về độ phức tạp được nêu trong [Phần 6.5](#).

Hình 6.13 Hàm str_cli sử dụng select để xử lý EOF một cách chính xác.

chọn/strcliselect02.c

```
1 #include "unp.h"
```

2 khoảng trắng

```
3 str_cli(TẬP_TIN *fp, int sockfd)
4 {
5     int      maxfdp1, stdineof;
6     fd_set đặt lại;
7     char buf[MAXLINE];
```

```

    int      N;

9     stdineof = 0;
10    FD_ZERO(&rset);
11    vì (      ;  ; ) {
12        nếu (stdineof == 0)
13            FD_SET(fileno(fp), &rset);
14            FD_SET(sockfd, &rset);
15            maxfdp1 = max(fileno(fp), sockfd) + 1;
16            Chọn(maxfdp1, &rset, NULL, NULL, NULL);

17        if (FD_ISSET(sockfd, &rset)) { /* socket có thể đọc đư ợc */
18            if ( (n = Đọc(sockfd, buf, MAXLINE)) == 0) {
19                nếu (stdineof == 1)
20                    trả lại;           /*kết thúc bình thư ờng */
21                khác
22                    err_quit("str_cli: máy chủ đã kết thúc sớm");
23            }
24            Write(fileno(stdout), buf, n);
25        }
26        if (FD_ISSET(fileno(fp), &rset)) { /* đầu vào có thể đọc đư ợc */
27            if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
28                stdineof = 1;
29                Tắt máy(sockfd, SHUT_WR); /* gửi FIN */
30                FD_CLR(fileno(fp), &rset);
31                Tiếp tục;
32            }
33            Đã viết(sockfd, buf, n);
34        }
35    }
36 }

```

5-8 `stdineof` là một cờ mới đư ợc khởi tạo thành 0. Miễn là cờ này bằng 0, mỗi lần xung quanh vòng lặp chính, chúng tôi `chọn` đầu vào tiêu chuẩn để dễ đọc.

17-25 Khi chúng tôi đọc EOF trên ổ cắm, nếu chúng tôi đã gặp EOF trên đầu vào tiêu chuẩn thì đây là kết thúc bình thư ờng và hàm sẽ trả về. Nhưng nếu chúng tôi chư a gặp EOF trên đầu vào tiêu chuẩn thì quá trình máy chủ đã kết thúc sớm. Bây giờ chúng tôi gọi `đọc` và `ghi` để hoạt động trên bộ đệm thay vì dòng và cho phép `chọn` hoạt động như mong đợi.

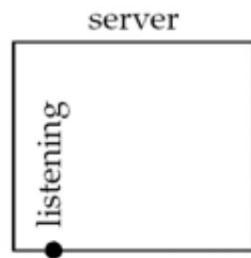
26-34 Khi chúng tôi gặp EOF trên đầu vào tiêu chuẩn, cờ mới của chúng tôi, `stdineof`, đư ợc đặt và chúng tôi gọi `tắt máy` với đối số thứ hai là `SHUT_WR` để gửi FIN. Ở đây cũng vậy, chúng tôi đã thay đổi sang hoạt động trên bộ đệm thay vì dòng, sử dụng `đọc` và `ghi`.

Chúng tôi chưa hoàn thành với hàm `str_cli` của mình. Chúng tôi sẽ phát triển một phiên bản sử dụng I/O không chặn trong Phần 16.2 và một phiên bản sử dụng các luồng trong Phần 26.3.

6.8 Máy chủ TCP Echo (Đã xem lại)

Chúng ta có thể truy cập lại máy chủ TCP echo của mình từ [Phần 5.2](#) và [5.3](#) và viết lại máy chủ như một quy trình duy nhất sử dụng [lựa chọn](#) để xử lý bất kỳ số lượng khách hàng nào, thay vì rèn một đứa trẻ cho mỗi khách hàng. Trước khi hiển thị mã, hãy xem cấu trúc dữ liệu mà chúng tôi sẽ sử dụng để theo dõi khách hàng. [Hình 6.14](#) thể hiện trạng thái của máy chủ trước khi khách hàng đầu tiên thiết lập kết nối.

Hình 6.14. Máy chủ TCP trước khi máy khách đầu tiên thiết lập kết nối.

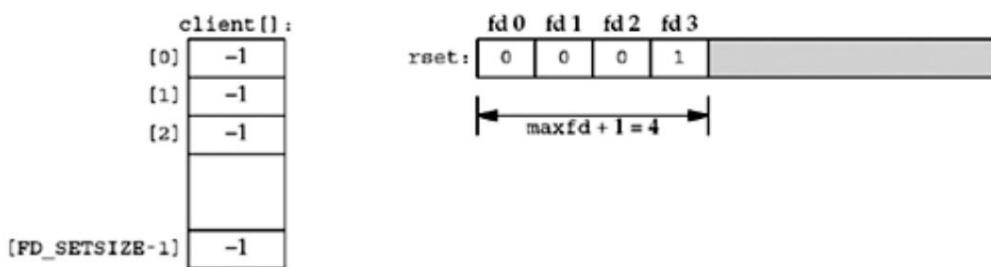


Máy chủ có một bộ mô tả lựu đạn nghe duy nhất mà chúng tôi hiển thị dưới dạng dấu đầu dòng.

Máy chủ chỉ duy trì một bộ mô tả đọc, được hiển thị trong [Hình 6.15](#). Chúng tôi giả sử rằng máy chủ được khởi động ở nền trước, do đó các bộ mô tả 0, 1 và 2 được đặt thành đầu vào, đầu ra và lỗi tiêu chuẩn. Do đó, bộ mô tả khả dụng đầu tiên cho ô cắm nghe là 3. Chúng tôi cũng hiển thị một mảng các số nguyên có tên `máy khách` chứa bộ mô tả ô cắm được kết nối cho mỗi máy khách. Tất cả các phần tử trong mảng này đều

được khởi tạo thành -1.

Hình 6.15. Cấu trúc dữ liệu cho máy chủ TCP chỉ với một ô cắm nghe.

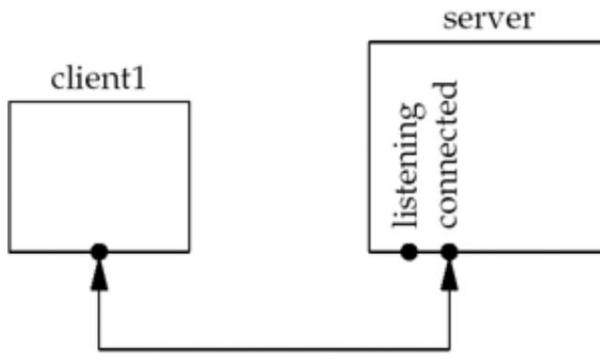


Mục nhập khác 0 duy nhất trong bộ mô tả là mục nhập dành cho ô cắm nghe và đối số đầu tiên cần [chọn](#) sẽ là 4.

Khi máy khách đầu tiên thiết lập kết nối với máy chủ của chúng tôi, bộ mô tả đang nghe sẽ có thể đọc được dữ liệu và các cuộc gọi máy chủ của chúng tôi sẽ chấp nhận. Bộ mô tả kết nối mới được trả về bởi chấp nhận sẽ là 4, dựa trên các giả định của ví dụ này. [Hình 6.16](#)

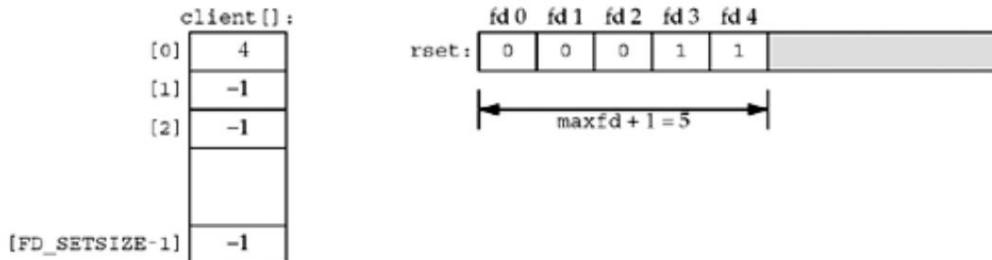
hiển thị kết nối từ máy khách đến máy chủ.

Hình 6.16. Máy chủ TCP sau khi máy khách đầu tiên thiết lập kết nối.



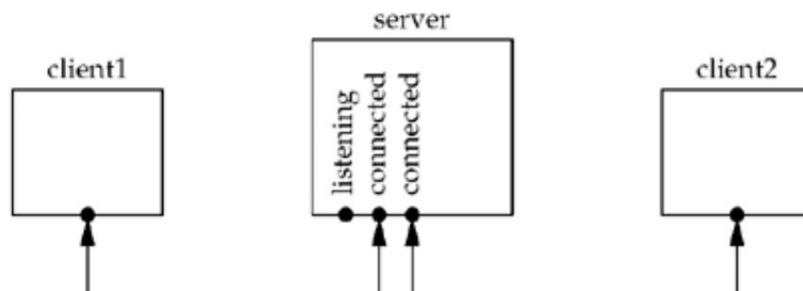
Kể từ thời điểm này, máy chủ của chúng tôi phải ghi nhớ ở cắm được kết nối mới trong [máy khách](#) của nó mang và ở cắm được kết nối phải được thêm vào bộ mô tả. Các cấu trúc dữ liệu cập nhật này được hiển thị trong [Hình 6.17](#).

Hình 6.17. Cấu trúc dữ liệu sau khi kết nối máy khách đầu tiên được thiết lập.



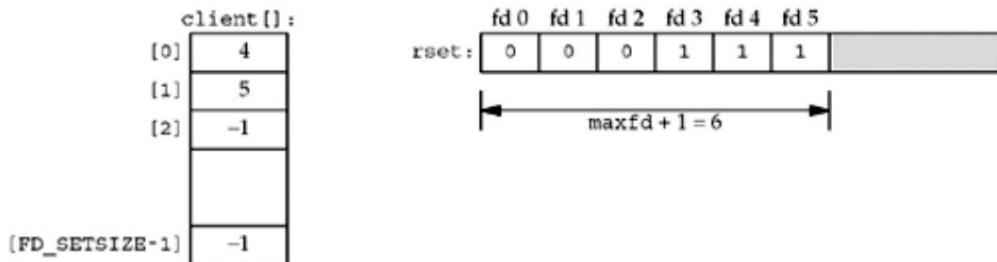
Một thời gian sau, khách hàng thứ hai thiết lập kết nối và chúng tôi có tình huống thể hiện trong [hình 6.18](#).

Hình 6.18. Máy chủ TCP sau khi kết nối máy khách thứ hai được thiết lập.



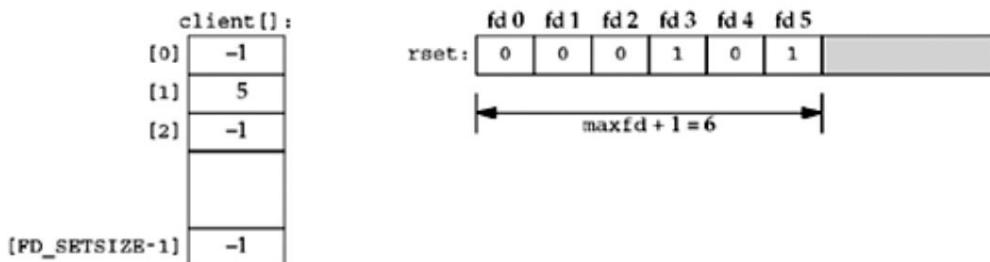
Ở cẩm đư ợc kết nối mới (mà chúng tôi giả sử là 5) phài đư ợc ghi nhớ, đư a ra cấu trúc dữ liệu đư ợc hiển thị trong [Hình 6.19](#).

Hình 6.19. Cấu trúc dữ liệu sau khi kết nối máy khách thứ hai đư ợc thiết lập.



Tiếp theo, chúng tôi giả sử máy khách đầu tiên chấm dứt kết nối của nó. Máy khách TCP gửi một FIN, làm cho bộ mô tả 4 trong máy chủ có thể đọc đư ợc. Khi máy chủ của chúng tôi đọc ở cẩm đư ợc kết nối này, `read` trả về 0. Sau đó, chúng tôi đóng ở cẩm này và cập nhật cấu trúc dữ liệu tương ứng. Giá trị của `client [0]` đư ợc đặt thành -1 và bộ mô tả 4 trong bộ mô tả đư ợc đặt thành 0. Điều này đư ợc hiển thị trong [Hình 6.20](#). Lưu ý rằng giá trị của `maxfd` không thay đổi.

Hình 6.20. Cấu trúc dữ liệu sau khi máy khách đầu tiên chấm dứt kết nối.



Tóm lại, khi khách hàng đến, chúng tôi ghi lại bộ mô tả ở cẩm đư ợc kết nối của họ trong mục nhập có sẵn đầu tiên trong mảng **máy khách** (tức là mục nhập đầu tiên có giá trị -1). Chúng ta cũng phài thêm ở cẩm đư ợc kết nối vào bộ mô tả đọc. Biến `maxi` là chỉ số cao nhất trong mảng **máy khách** hiện đang đư ợc sử dụng và biến `maxfd` (công mộng) là giá trị hiện tại của đối số đầu tiên cần **chọn**. Giới hạn duy nhất về số lượng máy khách mà máy chủ này có thể xử lý là giá trị tối thiểu của hai giá trị `FD_SETSIZE`

và số lượng bộ mô tả tối đa đư ợc nhân cho phép đối với quá trình này (mà chúng ta đã nói đến ở cuối [Phần 6.3](#)).

[Hình 6.21](#) thê hiên nữa đầu của phiên bản máy chủ này.

Hình 6.21 Máy chủ TCP sử dụng một tiến trình duy nhất và chọn: khởi tạo.

`tcpcliserv/tcpserveselect01.c`

```

1 #bao gồm          "unp.h"

2 số nguyên

3 chính(int argc, char **argv)
4 {
5     int      tôi, maxi, maxfd, listenfd, connfd, sockfd;
6     int      sẵn sàng, khách hàng[FD_SETSIZE];
7     cỡ_t n;
8     fd_set rset, allset;
9     char buf[MAXLINE];
10    socklen_t clilen;
11    struct sockaddr_in cliaddr, servaddr;

12    listenfd = Ở cắm(AF_INET, SOCK_STREAM, 0);

13    bzero(&servaddr, sizeof(servaddr));
14    servaddr.sin_family = AF_INET;
15    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
16    servaddr.sin_port = htons(SERV_PORT);

17    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

18    Nghe(listenfd, LISTENQ);

19    maxfd = listenfd;           /*khởi tạo */
20    cực đại = -1;             /* lập chỉ mục vào mảng client[] */
21    cho (i = 0; i < FD_SETSIZE; i++)
22        khách hàng[i] = -1;      /* -1 cho biết mục có sẵn */
23    FD_ZERO(&allset);
24    FD_SET(nghefd, &allset);

```

Tạo ở cắm nghe và khởi tạo để chọn

12-24 Các bước tạo ở cắm nghe cũng giống như đã thấy trước đó: Ở cắm, liên kết và lắng nghe. Chúng tôi khởi tạo cấu trúc dữ liệu của mình giả định rằng bộ mô tả duy nhất mà chúng tôi sẽ chọn ban đầu là ở cắm nghe.

Nửa cuối của hàm được hiển thị trong [Hình 6.22](#)

Hình 6.22 Máy chủ TCP sử dụng một vòng lặp chọn và xử lý đơn.

tcpcliserv/tcpservselect01.c

```

25    vì (      ;  ; ) {
26        rset = allset;           /* gán cấu trúc */

```

```

27         nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);

28         if (FD_ISSET(listenfd, &rset)) {                                /* kết nối máy khách mới
*/
29             clilen = sizeof(cliaddr);
30             connfd = Chấp nhận(listenfd, (SA *) &cliaddr, &clilen);

31             cho (i = 0; i < FD_SETSIZE; i++)
32                 nếu (khách hàng[i] < 0) {
33                     client[i] = connfd; /*lưu phần mô tả */
34                     phá vỡ;
35                 }
36             nếu (i == FD_SETSIZE)
37                 err_quit("quá nhiều khách hàng");
38             FD_SET(connfd, &allset);                                     /* thêm bộ mô tả mới để đặt
*/
39             nếu (connfd > maxfd)
40                 maxfd = connfd; /* để chọn */
41             nếu (i > maxi)
42                 tối đa = tối đa;           /* chỉ số tối đa trong mảng client[] */

43             nếu (--nready <= 0)
44                 Tiếp tục;          /* không còn bộ mô tả nào có thể đọc được nữa */
45             }
46             cho (i = 0; tối đa <= maxi; i++) {                         /* kiểm tra dữ liệu của tất cả các máy khách
*/
47                 nếu ( (sockfd = client[i]) < 0)
48                     Tiếp tục;
49                 if (FD_ISSET(sockfd, &rset)) {
50                     if ( (n = Đọc(sockfd, buf, MAXLINE)) == 0) {
51                         /*kết nối bị đóng bởi client */
52                         Đóng(sockfd);
53                         FD_CLR(sockfd, &allset);
54                         khách hàng[i] = -1;
55                     } khác
56                     Đã viết(sockfd, buf, n);
57                 nếu (--nready <= 0)
58                     phá vỡ;          /* không còn bộ mô tả nào có thể đọc được nữa */
59                 }
60             }
61         }
62     }

Chặn trong phần chọn

```

26-27 `select` chờ điều gì đó xảy ra: thiết lập kết nối máy khách mới hoặc dữ liệu đến, FIN hoặc RST trên kết nối hiện có.

Chấp nhận kết nối mới

28-45 Nếu ở cắm nghe có thể đọc được thì kết nối mới đã được thiết lập. Chúng tôi gọi **chấp nhận** và cập nhật cấu trúc dữ liệu của mình cho phù hợp. Chúng tôi sử dụng mục nhập không được sử dụng đầu tiên trong mảng **máy khách** để ghi lại ở cắm được kết nối. Số lượng bộ mô tả sẵn sàng giảm đi và nếu nó bằng 0, chúng ta có thể tránh vòng lặp `for` tiếp theo. Điều này cho phép chúng ta sử dụng giá trị trả về từ `select` để tránh việc kiểm tra các bộ mô tả chưa sẵn sàng.

Kiểm tra các kết nối hiện có

46-60 Một thử nghiệm được thực hiện cho mỗi kết nối máy khách hiện có để xem bộ mô tả của nó có nằm trong bộ mô tả được trả về bởi `select` hay không. Nếu vậy, một dòng sẽ được đọc từ máy khách và phản hồi lại máy khách. Nếu máy khách đóng kết nối, `read` trả về 0 và chúng tôi cập nhật cấu trúc dữ liệu ứng ứng.

Chúng tôi không bao giờ giảm giá trị của `maxi`, như ng chúng tôi có thể kiểm tra khả năng này mỗi khi khách hàng đóng kết nối.

Máy chủ này phức tạp hơn máy chủ được hiển thị trong [Hình 5.2](#) và [5.3](#), nhưng nó tránh được mọi chi phí cần thiết khi tạo một quy trình mới cho mỗi máy khách và đây là một ví dụ hay về [lựa chọn](#). Tuy nhiên, trong [Phần 16.6](#), chúng tôi sẽ mô tả một vấn đề với máy chủ này có thể dễ dàng khắc phục bằng cách làm cho ổ cắm nghe không bị chặn, sau đó kiểm tra và bỏ qua một số lỗi từ [việc chấp nhận](#).

Sự từ chối của dịch vụ tấn công

Thật không may, có một vấn đề với máy chủ mà chúng tôi vừa hiển thị. Hãy xem điều gì sẽ xảy ra nếu một máy khách đọc hại kết nối với máy chủ, gửi một byte dữ liệu (không phải dòng mới) và sau đó chuyển sang chế độ ngủ. Máy chủ sẽ gọi `read`, nó sẽ đọc

một byte dữ liệu từ máy khách rồi chặn lệnh gọi tiếp theo để [đọc](#), chờ thêm dữ liệu từ máy khách này. Sau đó, máy chủ sẽ bị chặn ("treo" có thể là một thuật ngữ tốt hơn) bởi một khách hàng này và sẽ không phục vụ bất kỳ khách hàng nào khác (kết nối khách hàng mới hoặc dữ liệu của khách hàng hiện tại) cho đến khi khách hàng đọc hại gửi dòng mới hoặc

chấm dứt.

Khái niệm cơ bản ở đây là khi một máy chủ đang xử lý nhiều máy khách, máy chủ đó không bao giờ có thể chặn lệnh gọi hàm liên quan đến một máy khách. Làm như vậy có thể treo máy chủ và từ chối dịch vụ cho tất cả các máy khách khác. Đây được gọi là một cuộc tấn công từ chối dịch vụ. Nó thực hiện điều gì đó với máy chủ để ngăn nó phục vụ các máy khách hợp pháp khác.

Các giải pháp khả thi là: (i) sử dụng I/O không chặn ([Chương 16](#)), (ii) yêu cầu mỗi máy khách được phục vụ bằng một luồng điều khiển riêng biệt (ví dụ: tạo ra một quy trình hoặc một luồng để phục vụ từng máy khách) hoặc (iii) đặt thời gian chờ cho các thao tác I/O ([Phần 14.2](#)).

6.9 Chức năng 'pselect'

Hàm `pselect` được POSIX phát minh và hiện được hỗ trợ bởi nhiều Các biến thể Unix.

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

#include <signal.h>

#include <time.h>

int pselect (int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *ngoại trừ, const struct timespec
*timeout, const sigset_t *sigmask);
```

`pselect` chứa hai thay đổi từ hàm `chọn` thông thư ờng :

1. `pselect` sử dụng cấu trúc `timespec`, một phát minh POSIX khác, thay vì
cấu trúc thời gian .
 - 2.
 - 3.
 - 4.
 5. cấu trúc thời gian {
 6. `time_t tv_sec; 7. tv_nsec` /* giây */
dài; /* nano giây */
 8. };
 - 9.

Sự khác biệt trong hai cấu trúc này là ở thành phần thứ hai:

Thành viên `tv_nsec` của cấu trúc mới hơn chỉ định nano giây, trong khi thành viên `tv_usec` của cấu trúc cũ chỉ định micro giây.

10. `pselect` thêm đối số thứ sáu: một con trỏ tới mặt nạ tín hiệu. Điều này cho phép chương trình để vô hiệu hóa việc phân phối các tín hiệu nhất định, kiểm tra một số biến chung được trình xử lý đặt cho các tín hiệu hiện đã bị tắt này, sau đó gọi `pselect`, yêu cầu nó đặt lại mặt nạ tín hiệu.

Về điểm thứ hai, hãy xem xét ví dụ sau (đư ợc thảo luận trên trang 308-309 của APUE). Trình xử lý tín hiệu của chương trình của chúng tôi cho `SIGINT` chỉ đặt `intr_flag` toàn cầu và trả về. Nếu quy trình của chúng tôi bị chặn trong lệnh gọi tới `chọn`, thì kết quả trả về từ trình xử lý tín hiệu sẽ khiến hàm trả về với `lỗi` đư ợc đặt thành `EINTR`. Như ng khi `select` đư ợc gọi, mã trông như sau:

```

nếu (intr_flag)
    xử lý_intr();           /*xử lý tín hiệu*/
if ( (nready = select( ... )) < 0) {
    if (errno == EINTR) {
        nếu (intr_flag)
            xử lý_intr();
    }
    ...
}

```

Vẫn đe là giữa quá trình kiểm tra `intr_flag` và lệnh gọi `select`, nếu tín hiệu xảy ra, nó sẽ bị mất nếu `select` chặn vĩnh viễn. Với `pselect`, bây giờ chúng ta có thể viết mã ví dụ này một cách đáng tin cậy như

```

sigset(SIG_BLOCK, oldmask, zeromask);

sigemptyset(&zeromask);
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* chặn SIGINT */
nếu (intr_flag)
    xử lý_intr();           /*xử lý tín hiệu*/
nếu ( (nready = pselect ( ... ,           &zeromask)) < 0) {
    if (errno == EINTR) {
        nếu (intr_flag)
            xử lý_intr ();
    }
    ...
}

```

Trước khi kiểm tra biến `intr_flag`, chúng tôi chặn `SIGINT`. Khi `pselect` được gọi, nó sẽ thay thế mặt nạ tín hiệu của quy trình bằng một tập trống (tức là `zeromask`) và sau đó kiểm tra các bộ mô tả, có thể chuyển sang chế độ ngủ. Như ng khi `pselect` trả về, mặt nạ tín hiệu của quy trình được đặt lại về giá trị của nó trước khi `pselect` được gọi (tức là `SIGINT` bị chặn).

Chúng ta sẽ nói nhiều hơn về `pselect` và đưa ra một ví dụ về nó trong [Phần 20.5](#). Chúng ta sẽ sử dụng `pselect` trong [Hình 20.7](#) và [hiển thị cách triển khai pselect](#) đơn giản, mặc dù không chính xác trong [Hình 20.8](#).

Có một sự khác biệt nhỏ khác giữa hai hàm `chọn`. Thành viên đầu tiên của cấu trúc `timeval` là một số nguyên dài có dấu, trong khi thành viên đầu tiên của cấu trúc `timespec` là `time_t`. Độ dài đã ký trư ớc đây cũng phải là `time_t`, nhưng không được thay đổi về trư ớc để tránh phá vỡ mã hiện có. Tuy nhiên, chức năng hoàn toàn mới có thể tạo ra sự thay đổi này.

6.10 Chức năng 'thăm dò ý kiến'

Chức năng `thăm dò` bắt nguồn từ SVR3 và ban đầu được giới hạn ở các thiết bị STREAMS ([Chương 31](#)). SVR4 đã loại bỏ giới hạn này, cho phép cuộc thăm dò hoạt động với bất kỳ bộ mô tả nào. `cuộc thăm dò` cung cấp chức năng tương tự như `cuộc chọn`, nhưng `cuộc thăm dò` cung cấp thông tin bổ sung khi xử lý các thiết bị STREAMS.

```
#include <poll.h>

int poll (struct pollfd *fdarray, nfds dài không dấu , int timeout);

Trả về: số lư ợng mô tả sẵn sàng, 0 khi hết thời gian chờ, -1 khi có lỗi
```

Đối số đầu tiên là một con trỏ tới phần tử đầu tiên của một mảng cấu trúc. Mỗi phần tử của mảng là một cấu trúc `pollfd` xác định các điều kiện cần kiểm tra cho một bộ mô tả nhất định, `fd`.

```
cấu trúc thăm dò ý kiến {
    int          fd;           /*mô tả để kiểm tra */
    sự kiện ngắn; /*sự kiện quan tâm trên fd */
    hồi lưu ngắn; /*các sự kiện xảy ra trên fd */
};
```

Các điều kiện cần kiểm tra được chỉ định bởi thành viên `sự kiện` và hàm trả về trạng thái cho bộ mô tả đó trong thành viên `revents` tương ứng. (Có hai biến cho mỗi bộ mô tả, một giá trị và một biến là kết quả, tránh các đối số kết quả-giá trị. Hãy nhớ rằng ba đối số ở giữa cho `select` là kết quả-giá trị.) Mỗi thành viên trong số hai thành viên này bao gồm một hoặc nhiều bit chỉ định một điều kiện nhất định.

[Hình 6.23](#) cho thấy các hàng số được sử dụng để chỉ định cờ `sự kiện` và để kiểm tra cờ `revents`.

Hình 6.23. Nhập sự kiện và trả lại kết quả cho cuộc thăm dò ý kiến.

Constant	Input to events ?	Result from events ?	Description
POLLIN	•	•	Normal or priority band data can be read
POLLRDNORM	•	•	Normal data can be read
POLLRDBAND	•	•	Priority band data can be read
POLLPRI	•	•	High-priority data can be read
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Normal data can be written
POLLWRBAND	•	•	Priority band data can be written
POLLERR		•	Error has occurred
POLLHUP		•	Hangup has occurred
POLLINVAL		•	Descriptor is not an open file

Chúng tôi đã chia hình này thành ba phần: Bốn hàng số đầu tiên xử lý đầu vào, ba hàng số tiếp theo xử lý đầu ra và ba hàng số cuối cùng xử lý lỗi. Lưu ý rằng ba phần cuối cùng không thể được đặt trong **các sự kiện** nhưng luôn được trả về trong **các lần quay lại** khi tồn tại điều kiện tương ứng.

Có ba loại dữ liệu được xác định bằng **cuộc thăm dò**: bình thường, dài ưu tiên và mức độ ưu tiên cao. Các thuật ngữ này đến từ việc triển khai dựa trên STREAMS ([Hình 31.5](#)).

POLLIN có thể được định nghĩa là OR logic của **POLLRDNORM** và **POLLRDBAND**. **POLLIN** hàng số tồn tại từ quá trình triển khai SVR3 có truớc các dài ưu tiên trong SVR4, do đó hàng số này vẫn giữ nguyên để tương thích ngược. Tương tự, **POLLOUT** tương đương với **POLLWRNORM**, với cái truớc có truớc cái sau.

Đối với các ô cắm TCP và UDP, các điều kiện sau đây khiếu **cuộc thăm dò** trả về kết quả đã chỉ định. Thực không may, POSIX để lại nhiều lỗ hổng (tức là các cách tùy chọn để trả về cùng một điều kiện) trong định nghĩa **thăm dò ý kiến**.

- Tất cả dữ liệu TCP thông thường và tất cả dữ liệu UDP được coi là bình thường.
- Dữ liệu ngoài băng tần của TCP ([Chương 24](#)) được coi là băng tần ưu tiên.
- Khi nửa đọc của kết nối TCP bị đóng (ví dụ: nhận được FIN), đây cũng được coi là dữ liệu bình thường và thao tác đọc tiếp theo sẽ trả về 0.
- Sự hiện diện của lỗi đối với kết nối TCP có thể được coi là dữ liệu bình thường hoặc lỗi (**POLLERR**). Trong cả hai trường hợp, **lần đọc tiếp theo** sẽ trả về -1 với **errno** được đặt thành giá trị thích hợp. Điều này xử lý các điều kiện như nhận RST hoặc thời gian chờ.
- Có thể xem xét tính khả dụng của kết nối mới trên ô cắm nghe dữ liệu bình thường hoặc dữ liệu ưu tiên. Hầu hết việc triển khai đều coi dữ liệu này là bình thường.
- Việc hoàn thành **kết nối** không chặn được coi là tạo thành một socket có thể ghi được.

Số phần tử trong mảng cấu trúc được chỉ định bởi đối số nfds .

Trong lịch sử, lập luận này đã có từ **lâu**, có vẻ quá đáng. Một **int không dấu** sẽ là đủ. Unix 98 định nghĩa một kiểu dữ liệu mới cho đối số này: **nfds_t**.

Đối số hết thời gian chờ chỉ định hàm phải đợi trong bao lâu trước khi quay lại. Giá trị đương chỉ định số mili giây chờ đợi. [Hình 6.24](#) hiển thị các giá trị có thể có cho đối số hết thời gian chờ .

Hình 6.24. giá trị thời gian chờ cho cuộc thăm dò.

<i>timeout</i> value	Description
INFTIM	Wait forever
0	Return immediately, do not block
> 0	Wait specified number of milliseconds

Hằng số **INFTIM** được xác định là giá trị âm. Nếu hệ thống không cung cấp bộ hẹn giờ với độ chính xác đến mili giây thì giá trị sẽ được làm tròn đến giá trị được hỗ trợ gần nhất giá trị.

Đặc tả POSIX yêu cầu **INFTIM** phải được xác định bằng cách bao gồm **<poll.h>**, nhưng nhiều hệ thống vẫn định nghĩa nó trong **<sys/stropts.h>**.

Giống như **lựa chọn**, bất kỳ thời gian chờ nào được đặt cho **cuộc thăm dò** đều bị giới hạn bởi độ phân giải đồng hồ của quá trình triển khai (thường là 10 mili giây).

Giá trị trả về từ **cuộc thăm dò** là -1 nếu xảy ra lỗi, 0 nếu không có bộ mô tả nào sẵn sàng trước khi hết giờ, nếu không thì đó là số lượng bộ mô tả có thành viên **trả về** khác 0 .

Nếu chúng ta không còn quan tâm đến một bộ mô tả cụ thể nữa, chúng ta chỉ cần đặt thành viên **fd** của cấu trúc **pollfd** thành giá trị âm. Sau đó, thành viên **sự kiện** bị bỏ qua và thành viên **trả lại** được đặt thành 0 khi trả về.

Hãy nhớ lại cuộc thảo luận của chúng ta ở cuối [Phần 6.3 về FD_SETSIZE](#) và số lượng bộ mô tả tối đa trên mỗi bộ mô tả so với số lượng bộ mô tả tối đa trên mỗi quy trình. Chúng tôi không gấp vấn đề đó với **poll** vì trách nhiệm của người gọi là phân bổ một mảng cấu trúc **pollfd** và sau đó cho kernel biết số lượng phần tử trong mảng. Không có kiểu dữ liệu có kích thước cố định tương tự như **fd_set**

mà hạt nhân biết về.

Đặc tả POSIX yêu cầu cả **chọn** và **thăm dò ý kiến**. Tuy nhiên, từ góc độ tính di động ngày nay, nhiều hệ thống hỗ trợ **lựa chọn** hơn là **thăm dò ý kiến**. Ngoài ra, POSIX định nghĩa **pselect**, một phiên bản nâng cao của **select** xử lý việc chặn tín hiệu và cung cấp độ phân giải thời gian tăng lên. Không có gì tương tự được xác định cho **cuộc thăm dò ý kiến**.

6.11 TCP Echo Server (Đã được xem lại lần nữa)

Bây giờ chúng tôi làm lại máy chủ TCP echo của mình từ [Phần 6.8](#) bằng cách sử dụng [cuộc thăm dò](#) thay vì [chọn](#). Trong phiên bản trước sử dụng [select](#), chúng tôi phải phân bổ một mảng [máy khách](#) cùng với bộ mô tả có tên [rset](#) ([Hình 6.15](#)). Với [poll](#), chúng ta phải phân bổ một mảng cấu trúc [pollfd](#) để duy trì thông tin khách hàng thay vì phân bổ một mảng khác. Chúng tôi xử lý thành viên [fd](#) của mảng này giống như cách chúng tôi xử lý [máy khách](#) mảng trong [Hình 6.15](#): giá trị -1 nghĩa là mục này không được sử dụng; mặt khác, nó là giá trị mô tả. Hãy nhớ lại từ phần trước rằng bất kỳ mục nào trong mảng Các cấu trúc [pollfd](#) được chuyển đến [cuộc thăm dò](#) có giá trị âm cho thành viên [fd](#) sẽ bị bỏ qua.

[Hình 6.25](#) hiển thị nửa đầu của máy chủ của chúng tôi.

Hình 6.25 Nửa đầu máy chủ TCP sử dụng thăm dò ý kiến.

tcpcliserv/tcpservpoll01.c

```

1 #bao gồm          "unp.h"
2 #bao gồm          <giới hạn.h>      /* cho OPEN_MAX */

3 int
4 chính(int argc, char **argv)
5 {
6     int      tôi, maxi, listenfd, connfd, sockfd;
7     int      đã sẵn sàng;
8     cở_t n;
9     char buf[MAXLINE];
10    socklen_t clilen;
11    cấu trúc máy khách thăm dò ý kiến [OPEN_MAX];
12    struct sockaddr_in cliaddr, servaddr;

13    listenfd = Ở cảm(AF_INET, SOCK_STREAM, 0);

14    bzero(&servaddr, sizeof(servaddr));
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
17    servaddr.sin_port = htons(SERV_PORT);

18    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

19    Nghe(listenfd, LISTENQ);

```

```

20     client[0].fd = listenfd;
21     client[0].events = POLLRDNORM;
22     cho (i = 1; tôi < OPEN_MAX; i++)
23         client[i].fd = -1;           /* -1 cho biết mục có sẵn */
24     cực đại = 0;                 /* chỉ mục tối đa vào mảng client[] */

```

Phân bổ mảng cấu trúc pollfd

11 Chúng ta khai báo các phần tử `OPEN_MAX` trong mảng cấu trúc `pollfd` của mình. Việc xác định số lượng mô tả tối đa mà một tiến trình có thể mở tại một thời điểm là rất khó. Chúng ta sẽ gặp lại vấn đề này trong [Hình 13.4](#). Một cách là gọi hàm POSIX `sysconf` với đối số `_SC_OPEN_MAX` (như [đã](#) được mô tả trên trang 42-44 của APUE) rồi phân bổ động một mảng có kích thước phù hợp. Nhưng một trong những kết quả có thể có từ `sysconf` là "không xác định", nghĩa là chúng ta vẫn phải đoán một giá trị. Ở đây, chúng tôi chỉ sử dụng hằng số POSIX `OPEN_MAX`.

Khởi tạo

20-24 Chúng tôi sử dụng mục nhập đầu tiên trong mảng `máy khách` cho socket nghe và đặt bộ mô tả cho các mục còn lại thành -1. Chúng tôi cũng đặt sự kiện `POLLRDNORM` cho bộ mô tả này để được thông báo bằng `cuộc thăm dò` khi kết nối mới sẵn sàng được chấp nhận. Biến `maxi` chứa chỉ mục lớn nhất của mảng `máy khách` hiện đang được sử dụng.

Nửa sau của hàm của chúng tôi được hiển thị trong [Hình 6.26](#).

[Hình 6.26](#) Nửa sau của máy chủ TCP sử dụng poll.

`tcpcliserv/tcpservpoll01.c`

```

25     vì (      ;   ) {
26         nready = Thăm dò ý kiến(client, maxi + 1, INFTIM);
27
28         if (client[0].revents & POLLRDNORM) { /* kết nối máy khách mới
29             */
30             clilen = sizeof(cliaddr);
31             connfd = Chấp nhận(listenfd, (SA *) &cliaddr, &clilen);
32
33             cho (i = 1; tôi < OPEN_MAX; i++)
34                 if (client[i].fd < 0) {
35                     client[i].fd = connfd; /* lưu phần mô tả */
36                     phá vỡ;
37                 }
38             nếu (i == OPEN_MAX)
39                 err_quit("quá nhiều khách hàng");
40             client[i].events = POLLRDNORM;

```

```

38             nếu (i > maxi)
39                 tối đa = tôi;           /* chỉ số tối đa trong mảng client[] */
40
41             nếu (--nready <= 0)
42                 Tiếp tục;           /* không còn bộ mô tả nào có thể đọc đư ợc nữa */
43             }
44
45             cho (i = 1; tôi <= maxi; i++) {           /* kiểm tra dữ liệu của tất cả các máy khách */
46
47                 if ( (sockfd = client[i].fd) < 0)
48                     Tiếp tục;
49                 if (client[i].revents & (POLLRDNORM | POLLERR)) {
50                     if ( (n = read(sockfd, buf, MAXLINE)) < 0) {
51                         if (errno == ECONNRESET) {
52                             /*thiết lập lại kết nối bởi client */
53                             Đóng(sockfd);
54                             client[i].fd = -1;
55                         } khác
56                         err_sys("lỗi đọc");
57                     } khác nếu (n == 0) {
58                         /*kết nối bị đóng bởi client */
59                         Đóng(sockfd);
60                         client[i].fd = -1;
61                     } khác
62                     Đã viết(sockfd, buf, n);
63                 nếu (--nready <= 0)
64                     phá vỡ;           /* không còn bộ mô tả nào có thể đọc đư ợc nữa */
65             }

```

Gọi thăm dò ý kiến, kiểm tra kết nối mới

26-42 Chúng tôi gọi **cuộc thăm dò ý kiến** để chờ kết nối mới hoặc dữ liệu về kết nối hiện có.

Khi một kết nối mới đư ợc chấp nhận, chúng tôi sẽ tìm thấy mục nhập có sẵn đầu tiên trong ứng dụng khách mảng bằng cách tìm kiếm mảng đầu tiên có bộ mô tả phủ định. Lưu ý rằng chúng tôi bắt đầu tìm kiếm với chỉ mục 1, vì **client[0]** đư ợc sử dụng cho ổ cẩm nghe. Khi tìm thấy một mục có sẵn, chúng tôi lưu bộ mô tả và đặt sự kiện **POLLRDNORM**.

Kiểm tra dữ liệu trên kết nối hiện có

43-63 Hai sự kiện trả về mà chúng tôi kiểm tra là **POLLRDNORM** và **POLLERR**. Các

thứ hai trong số này chúng tôi không đặt trong thành viên **sự kiện** vì nó luôn đư ợc trả về khi điều kiện đúng. Lý do chúng tôi kiểm tra **POLLERR** là vì một số

việc triển khai trả về sự kiện này khi nhận được RST cho một kết nối, trong khi những sự kiện khác chỉ trả về POLLRDNORM. Trong cả hai trường hợp, chúng ta gọi `read` và nếu xảy ra lỗi, nó sẽ trả về lỗi. Khi một kết nối hiện có bị khách hàng chấm dứt, chúng tôi chỉ đặt thành viên `fd` thành -1.

6.12 Tóm tắt

Có năm mô hình I/O khác nhau được cung cấp bởi Unix:

- Chặn
- Không chặn
- Ghép kênh I/O
- Vào/ra điều khiển bằng tín hiệu
- Vào/ra không đồng bộ

Mặc định là chặn I/O, đây cũng là cách được sử dụng phổ biến nhất. Chúng tôi sẽ đề cập đến I/O không chặn và I/O điều khiển bằng tín hiệu trong các chương sau và đã đề cập đến việc ghép kênh I/O trong chương này. I/O không đồng bộ thực sự được xác định bởi đặc tả POSIX, nhưng tồn tại rất ít triển khai.

Chức năng được sử dụng phổ biến nhất để ghép kênh I/O là `chọn`. Chúng tôi nói với người `để lựa chọn` chức năng những bộ mô tả mà chúng ta quan tâm (để đọc, viết và ngoại lệ), lựu lượng thời gian chờ đợi tối đa và số bộ mô tả tối đa (cộng một). Hầu hết các cuộc gọi tới `select` đều chỉ định khả năng đọc và chúng tôi đã lưu ý rằng điều kiện ngoại lệ duy nhất khi xử lý ở cẩm là sự xuất hiện của dữ liệu ngoài bẳng tần ([Chương 24](#)).

Vì `select` cung cấp giới hạn thời gian về thời gian chặn một hàm, nên chúng tôi sẽ sử dụng tính năng này trong [Hình 14.3](#) để đặt giới hạn thời gian cho một thao tác đầu vào.

Chúng tôi đã sử dụng ứng dụng khách echo của mình ở chế độ hàng loạt bằng cách sử dụng `tính năng chọn` và phát hiện ra rằng mặc dù gặp phải phần cuối của đầu vào của người dùng, dữ liệu vẫn có thể được truyền đến hoặc từ máy chủ. Để xử lý tình huống này cần có chức năng `tắt máy` và nó cho phép chúng ta tận dụng tính năng đóng một nửa của TCP.

Sự nguy hiểm của việc trộn lẫn bộ đệm stdio (cũng như bộ đệm `dòng đọc` của riêng chúng tôi) với `lựa chọn` đã khiến chúng tôi tạo ra các phiên bản của máy khách và máy chủ echo hoạt động trên bộ đệm thay vì dòng.

POSIX xác định hàm `pselect`, giúp tăng độ chính xác về thời gian từ micro giây lên nano giây và nhận một đối số mới là con trỏ tới bộ tín hiệu. Điều này cho phép chúng ta tránh được tình trạng cạnh tranh khi bắt được tín hiệu và chúng ta nói chuyện nhiều hơn về điều này trong [Phần 20.5](#).

Chức năng `thăm dò` từ Hệ thống V cung cấp chức năng tư ứng tự như `chọn` và cung cấp thông tin bổ sung trên các thiết bị STREAMS. POSIX yêu cầu cả hai `lựa chọn` và `thăm dò ý kiến`, nhưng cái trước được sử dụng thường xuyên hơn.

Bài tập

6.1 Chúng ta đã nói rằng một bộ mô tả có thể được gán cho một bộ mô tả khác qua dấu bằng trong C. Việc này được thực hiện như thế nào nếu một bộ mô tả là một mảng các số nguyên?

(Gợi ý: Hãy xem tiêu đề `<sys/select.h>` hoặc `<sys/types.h>` của hệ thống của bạn.)

6.2 Khi mô tả các điều kiện để `lựa chọn` trả về "có thể ghi" trong [Phản 6.3](#), tại sao chúng ta cần hạn định `rằng ở` `cắm` phải không bị chặn để thao tác ghi trả về giá trị đương?

6.3 Điều gì xảy ra trong [Hình 6.9](#) nếu chúng ta thêm từ "`else`" vào trư ớc từ đó "`nếu`" ở dòng 19?

6.4 Trong ví dụ của chúng tôi ở [Hình 6.21](#), hãy thêm mã để cho phép máy chủ có thể sử dụng nhiều bộ mô tả như được hặt nhân cho phép hiện tại. (Gợi ý: Hãy xem hàm `setrlimit`.)

6.5 Hãy xem điều gì sẽ xảy ra khi đổi số thứ hai của lệnh `tắt máy` là `SHUT_RD`.

Bắt đầu với máy khách TCP trong [Hình 5.4](#) và thực hiện các thay đổi sau: Thay đổi số cổng từ `SERV_PORT` thành 19, máy chủ tính phí ([Hình 2.18](#)); sau đó, thay thế lệnh gọi `str_cli` bằng lệnh `gọi hàm tạm dừng`. Chạy chương trình này chỉ định địa chỉ IP của máy chủ cục bộ chạy máy chủ được tính phí. Xem các gói bằng một công cụ như `tcpdump` ([Phản C.5](#)). Điều gì xảy ra?

6.6 Tại sao một ứng dụng gọi `tắt máy` với đổi số `SHUT_RDWR` thay vì chỉ gọi `đóng`?

6.7 Điều gì xảy ra trong [Hình 6.22](#) khi máy khách gửi RST để chấm dứt sự liên quan?

6.8 Mã hóa lại [Hình 6.25](#) để gọi `sysconf` nhằm xác định số lưu lượng bộ mô tả tối đa và phân bổ mảng `máy khách` tư ứng ứng.

Chương 7. Tùy chọn ô cắm

[Mục 7.1. Giới thiệu](#)

[Mục 7.2. Hàm `getsockopt` và `setsockopt`](#)

[Mục 7.3. Kiểm tra xem một tùy chọn có được hỗ trợ hay không và lấy mặc định](#)[Mục 7.4. Trạng thái ở cắm](#)[Mục 7.5. Tùy chọn ở cắm chung](#)[Mục 7.6. Tùy chọn ở cắm IPv4](#)[Mục 7.7. Tùy chọn ở cắm ICMPv6](#)[Mục 7.8. Tùy chọn ở cắm IPv6](#)[Mục 7.9. Tùy chọn ở cắm TCP](#)[Mục 7.10. Tùy chọn ở cắm SCTP](#)[Mục 7.11. Hàm fcntl](#)[Mục 7.12. Bản tóm tắt](#)[Bài tập](#)

7.1 Giới thiệu

Có nhiều cách khác nhau để lấy và đặt các tùy chọn ảnh hưởng đến ở cắm:

- Các hàm `getsockopt` và `setsockopt`
- Hàm `fcntl`
- Hàm `ioctl`

Chương này bắt đầu bằng cách đề cập đến các hàm `setsockopt` và `getsockopt`, tiếp theo là một ví dụ in giá trị mặc định của tất cả các tùy chọn và sau đó là mô tả chi tiết về tất cả các tùy chọn ở cắm. Chúng tôi chia các mô tả chi tiết thành các loại sau: chung, IPv4, IPv6, TCP và SCTP. Nội dung chi tiết này có thể được bỏ qua trong lần đọc đầu tiên của chương này và các phần riêng lẻ sẽ được tham chiếu khi cần thiết. Một số tùy chọn sẽ được thảo luận chi tiết trong chương sau, chẳng hạn như các tùy chọn phát đa hướng IPv4 và IPv6 mà chúng tôi sẽ mô tả về phát đa hướng trong [Phần 21.6](#).

Chúng tôi cũng mô tả hàm `fcntl`, vì đây là cách POSIX để đặt ở cắm cho I/O không chặn, I/O điều khiển bằng tín hiệu và để đặt chủ sở hữu của ở cắm. Chúng ta lưu hàm `ioctl` cho [Chương 17](#).

7.2 Hàm 'getsockopt' và 'setsockopt'

Hai chức năng này chỉ áp dụng cho ô cắm.

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

int getsockopt(int sockfd, int cấp, int optname, void *optval, socklen_t
*optlen);

int setsockopt(int sockfd, int cấp, int optname, const void *optval socklen_t
optlen);
```

Cả hai đều trả về: 0 nếu OK, -1 bị lỗi

sockfd phải tham chiếu đến bộ mô tả ô cắm mở. cấp độ chỉ định mã trong hệ thống diển giải tùy chọn: mã ô cắm chung hoặc một số mã dành riêng cho giao thức (ví dụ: IPv4, IPv6, TCP hoặc SCTP).

optval là một con trỏ tới một biến mà từ đó giá trị mới của tùy chọn được `setsockopt` tìm nạp hoặc giá trị hiện tại của tùy chọn được lưu trữ bởi `getsockopt`.

Kích thước của biến này được chỉ định bởi đối số cuối cùng, làm giá trị cho `setsockopt` và là kết quả giá trị cho `getsockopt`.

[Hình 7.1](#) và [7.2](#) tóm tắt các tùy chọn có thể được truy vấn bởi `getsockopt` hoặc được thiết lập bởi `setsockopt`. Cột "Datatype" hiển thị kiểu dữ liệu của optval con trỏ phải trả tới cho mỗi tùy chọn. Chúng ta sử dụng ký hiệu hai dấu ngoặc nhọn để biểu thị một cấu trúc, như trong `linger{}` có nghĩa là một `cấu trúc kéo dài`.

Hình 7.1. Tóm tắt các tùy chọn ô cắm và ô cắm lớp IP cho `getsockopt` và `setsockopt`.

level	optionname	get	set	Description	Flag	Datatype
SOL_SOCKET	SO_BROADCAST	•	•	Permit sending of broadcast datagrams	•	int
	SO_DEBUG	•	•	Enable debug tracing	•	int
	SO_DONTROUTE	•	•	Bypass routing table lookup	•	int
	SO_ERROR	•	•	Get pending error and clear	•	int
	SO_KEEPALIVE	•	•	Periodically test if connection still alive	•	int
	SO_LINGER	•	•	Linger on close if data to send	linger{}	linger{}
	SO_OOBINLINE	•	•	Leave received out-of-band data inline	•	int
	SO_RCVBUF	•	•	Receive buffer size	•	int
	SO_SNDBUF	•	•	Send buffer size	•	int
	SO_RCVLOWAT	•	•	Receive buffer low-water mark	•	int
	SO_SNDDLOWAT	•	•	Send buffer low-water mark	•	int
	SO_RCVTIMEO	•	•	Receive timeout	timeval{}	timeval{}
	SO_SNDDTIMEO	•	•	Send timeout	timeval{}	timeval{}
	SO_REUSEADDR	•	•	Allow local address reuse	•	int
	SO_REUSEPORT	•	•	Allow local port reuse	•	int
	SO_TYPE	•	•	Get socket type	•	int
	SO_USELOOPBACK	•	•	Routing socket gets copy of what it sends	•	int
IPPROTO_IP	IP_HDRINCL	•	•	IP header included with data	•	int
	IP_OPTIONS	•	•	IP header options	•	(see text)
	IP_RECVDSTADDR	•	•	Return destination IP address	•	int
	IP_RECVIF	•	•	Return received interface index	•	int
	IP_TOS	•	•	Type-of-service and precedence	•	int
	IP_TTL	•	•	TTL	•	int
	IP_MULTICAST_IF	•	•	Specify outgoing interface	•	in_addr{}
	IP_MULTICAST_TTL	•	•	Specify outgoing TTL	•	u_char
	IP_MULTICAST_LOOP	•	•	Specify loopback	•	u_char
	IP_ADD_MEMBERSHIP	•	•	Join or leave multicast group	•	ip_mreq{}
IPPROTO_ICMPV6	ICMP6_FILTER	•	•	Specify ICMPv6 message types to pass	•	icmp6_filter{}
	IPV6_CHECKSUM	•	•	Offset of checksum field for raw sockets	•	int
	IPV6_DONTFRAG	•	•	Drop instead of fragment large packets	•	int
	IPV6_NEXTHOP	•	•	Specify next-hop address	•	sockaddr_in6{}
	IPV6_PATHMTU	•	•	Retrieve current path MTU	•	ip6_mtuinfo{}
	IPV6_RECVDSTOPTS	•	•	Receive destination options	•	int
	IPV6_RECVHOPLIMIT	•	•	Receive unicast hop limit	•	int
	IPV6_RECVHOPOPTS	•	•	Receive hop-by-hop options	•	int
	IPV6_RECVPATHMTU	•	•	Receive path MTU	•	int
	IPV6_RECVPKTINFO	•	•	Receive packet information	•	int
IPPROTO_IPV6	IPV6_RECVRTHDR	•	•	Receive source route	•	int
	IPV6_RECVTCLASS	•	•	Receive traffic class	•	int
	IPV6_UNICAST_HOPS	•	•	Default unicast hop limit	•	int
	IPV6_USE_MIN_MTU	•	•	Use minimum MTU	•	int
	IPV6_V6ONLY	•	•	Disable v4 compatibility	•	int
	IPV6_XXX	•	•	Sticky ancillary data	•	(see text)
	IPV6_MULTICAST_IF	•	•	Specify outgoing interface	•	u_int
	IPV6_MULTICAST_HOPS	•	•	Specify outgoing hop limit	•	int
	IPV6_MULTICAST_LOOP	•	•	Specify loopback	•	u_int
	IPV6_JOIN_GROUP	•	•	Join multicast group	•	ipv6_mreq{}
	IPV6_LEAVE_GROUP	•	•	Leave multicast group	•	ipv6_mreq{}
IPPROTO_IP or IPPROTO_IPV6	MCAST_JOIN_GROUP	•	•	Join multicast group	group_req{}	group_source_req{}
	MCAST_LEAVE_GROUP	•	•	Leave multicast group	group_req{}	group_source_req{}
	MCAST_BLOCK_SOURCE	•	•	Block multicast source	group_req{}	group_source_req{}
	MCAST_UNBLOCK_SOURCE	•	•	Unblock multicast source	group_req{}	group_source_req{}
	MCAST_JOIN_SOURCE_GROUP	•	•	Join source-specific multicast	group_req{}	group_source_req{}
	MCAST_LEAVE_SOURCE_GROUP	•	•	Leave source-specific multicast	group_req{}	group_source_req{}

Hình 7.2. Tóm tắt các tùy chọn ô cắm của lớp vận chuyển.

level	optionname	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_MAXSEG	*	*	TCP maximum segment size	•	int
	TCP_NODELAY	*	*	Disable Nagle algorithm		int
IPPROTO_SCTP	SCTP_ADAPTION_LAYER	*	*	Adaption layer indication	•	sctp_setadaption()
	SCTP_ASSOCINFO	†	*	Examine and set association info		sctp_assocparam()
	SCTP_AUTOCLOSE	*	*	Autoclose operation		int
	SCTP_DEFAULT_SEND_PARAM	*	*	Default send parameters		sctp_sndrcvinfo()
	SCTP_DISABLE_FRAGMENTS	*	*	SCTP fragmentation		int
	SCTP_EVENTS	*	*	Notification events of interest		sctp_event_subscribe()
	SCTP_GET_PEER_ADDR_INFO	†	*	Retrieve peer address status		sctp_paddrinfo()
	SCTP_I_MAPPED_V4_ADDR	*	*	Mapped v4 addresses		int
	SCTP_INITMSG	*	*	Default INIT parameters		sctp_initmsg()
	SCTP_MAXBURST	*	*	Maximum burst size		int
	SCTP_MAXSEG	*	*	Maximum fragmentation size		int
	SCTP_NODELAY	*	*	Disable Nagle algorithm		int
	SCTP_PEER_ADDR_PARAMS	†	*	Peer address parameters		sctp_paddrparams()
	SCTP_PRIMARY_ADDR	†	*	Primary destination address		sctp_setprim()
	SCTP_RTOINFO	†	*	RTO information		sctp_rtoinfo()
	SCTP_SET_PEER_PRIMARY_ADDR	*	*	Peer primary destination address	•	sctp_setpeerprim()
	SCTP_STATUS	†	*	Get association status		sctp_status()

Có hai loại tùy chọn cơ bản: tùy chọn nhị phân bật hoặc tắt một tính năng nhất định (cờ) và tùy chọn tìm nạp và trả về các giá trị cụ thể mà chúng ta có thể đặt

hoặc kiểm tra (giá trị). Cột có nhãn "Cờ" chỉ định xem tùy chọn có phải là tùy chọn gắn cờ hay không.

Khi gọi `getsockopt` cho các tùy chọn cờ này, `*optval` là một số nguyên. Giá trị được trả về trong `*optval` bằng 0 nếu tùy chọn này bị tắt hoặc khác 0 nếu tùy chọn này được bật.

Tương tự, `setsockopt` yêu cầu giá trị `*optval` khác 0 để bật tùy chọn và giá trị 0 để tắt tùy chọn. Nếu cột "Cờ" không chứa "^" thì tùy chọn này được sử dụng để chuyển giá trị của kiểu dữ liệu được chỉ định giữa quy trình ngay ở cùng một hệ thống.

Các phần tiếp theo của chương này sẽ cung cấp thêm chi tiết về các tùy chọn mà ảnh hưởng đến ở cắm.

7.3 Kiểm tra xem một tùy chọn có được hỗ trợ hay không và lấy

Mặc định

Bây giờ chúng ta viết một chương trình để kiểm tra xem hầu hết các tùy chọn được xác định trong [Hình 7.1](#) có và [7.2](#) đều được hỗ trợ và nếu vậy, hãy in giá trị mặc định của chúng. [Hình 7.3](#) chứa các khai báo cho chương trình của chúng ta.

Khai báo hợp các giá trị có thể

[3-8 Liên minh](#) của chúng tôi chứa một thành viên cho mỗi giá trị trả về có thể có từ `getsockopt`.

Xác định nguyên mẫu hàm

[9-12](#) Chúng tôi xác định nguyên mẫu hàm cho bốn hàm được gọi để in giá trị cho một tùy chọn ở cắm nhất định.

Xác định cấu trúc và khởi tạo mảng

[13-52](#) Cấu trúc `sock_opts` của chúng tôi chứa tất cả thông tin cần thiết để gọi `getsockopt` cho mỗi tùy chọn socket và sau đó in giá trị hiện tại của nó. Thành viên cuối cùng, `opt_val_str`, là một con trỏ tới một trong bốn hàm của chúng tôi sẽ in giá trị tùy chọn.

Chúng tôi phân bổ và khởi tạo một mảng gồm các cấu trúc này, một phần tử cho mỗi tùy chọn ở cắm.

Không phải tất cả các triển khai đều hỗ trợ tất cả các tùy chọn ở cắm. Cách để xác định xem một tùy chọn nhất định có được hỗ trợ hay không là sử dụng `#ifdef` hoặc `#if` để xác định, như chúng tôi hiển thị cho `SO_REUSEPORT`. Để hoàn thiện, mọi phần tử của mảng phải được biên dịch tự như những gì chúng tôi hiển thị cho `SO_REUSEPORT`, nhưng chúng tôi bỏ qua những phần tử này vì `#ifdefs` chỉ kéo dài đoạn mã mà chúng tôi hiển thị và không thêm gì vào cuộc thảo luận.

[Hình 7.3](#) Các khai báo cho chương trình của chúng ta để kiểm tra các tùy chọn socket.

`sockopt/checkopts.c`

```
1 #include "unp.h"
2 #include <netinet/tcp.h>           /* cho TCP_xxx định nghĩa */
3 công đoàn val {
4 số nguyên          i_val;
5 dài              l_val;
6 cấu trúc nán lại      nán lại_val;
7 cấu trúc thời gian timeval_val;
8 } giá trị;
9 char tinh *sock_str_flag(union val *, int);
10 char tinh *sock_str_int(union val *, int);
11 char tinh *sock_str_linger(union val *, int);
12 char tinh *sock_str_timeval(union val *, int);
13 cấu trúc sock_opts {
ký tự 14 const          *opt_str;
15 int          opt_level;
16 int          opt_name;
17 char *(*opt_val_str) (union val *, int);
18 } sock_opts[] = {
19     { "SO_BROADCAST",           SOL_SOCKET, SO_BROADCAST,
  sock_str_flag },
20     { "SO_DEBUG",               SOL_SOCKET, SO_DEBUG,           sock_str_flag },
21     { "SO_DONTROUTE",           SOL_SOCKET, SO_DONTROUTE,
  sock_str_flag },
22     { "SO_ERROR",                SOL_SOCKET, SO_ERROR,           sock_str_int },
23     { "SO_KEEPALIVE",           SOL_SOCKET, SO_KEEPALIVE,
  sock_str_flag },
24     { "SO_LINGER",                 SOL_SOCKET, SO_LINGER,
  sock_str_linger },
25     { "SO_OOBINLINE",            SOL_SOCKET, SO_OOBINLINE,
  sock_str_flag },
26     { "SO_RCVBUF",                SOL_SOCKET, SO_RCVBUF,           sock_str_int },
27     { "SO_SNDBUF",                SOL_SOCKET, SO_SNDBUF,           sock_str_int },
28     { "SO_RCVLOWAT",              SOL_SOCKET, SO_RCVLOWAT,          sock_str_int },
29     { "SO SNDLOWAT",              SOL_SOCKET, SO SNDLOWAT,          sock_str_int },
30     { "SO_RCVTIMEO",              SOL_SOCKET, SO_RCVTIMEO,
  sock_str_timeval },
31     { "SO_SNDTIMEO",              SOL_SOCKET, SO SNDTIMEO,
  sock_str_timeval },
32     { "SO_REUSEADDR",             SOL_SOCKET, SO_REUSEADDR,
  sock_str_flag },
33 #ifdef SO_REUSEPORT
34     { "SO_REUSEPORT",             SOL_SOCKET, SO_REUSEPORT,
  sock_str_flag },
35 #khác
```

```

36     { "SO_REUSEPORT",           0,          0,           VÔ GIÁ TRỊ },
37 #endif
38     { "SO_TYPE",                SOL_SOCKET, SO_TYPE,           sock_str_int },
39     { "SO_USELOOPBACK",         SOL_SOCKET, SO_USELOOPBACK,
sock_str_flag },
40     { "IP_TOS",                 IPPROTO_IP, IP_TOS,           sock_str_int },
41     { "IP_TTL",                 IPPROTO_IP, IP_TTL,           sock_str_int },
42     { "IPV6_DONTFRAG",         IPPROTO_IPV6, IPV6_DONTFRAG,
sock_str_flag },
43     { "IPV6_UNICAST_HOPS",
IPPROTO_IPV6, IPV6_UNICAST_HOPS, sock_str_int },
44     { "CHỈ IPV6_V6",           IPPROTO_IPV6, CHỈ IPV6_V6,
sock_str_flag },
45     { "TCP_MAXSEG",             IPPROTO_TCP, TCP_MAXSEG,           sock_str_int },
46     { "TCP_NODELAY",            IPPROTO_TCP, TCP_NODELAY,
sock_str_flag },
47     { "SCTP_AUTOCLOSE",
IPPROTO_SCTP, SCTP_AUTOCLOSE, sock_str_int },
48     { "SCTP_MAXBURST",          IPPROTO_SCTP, SCTP_MAXBURST,
sock_str_int },
49     { "SCTP_MAXSEG",            IPPROTO_SCTP, SCTP_MAXSEG,
sock_str_int },
50     { "SCTP_NODELAY",           IPPROTO_SCTP, SCTP_NODELAY,
sock_str_flag },
51     { VÔ GIÁ TRỊ,               0,          0,           VÔ GIÁ TRỊ }
52 };

```

Hình 7.4 cho thấy chức năng chính của chúng tôi .

Hình 7.4 chức năng chính để kiểm tra tất cả các tùy chọn ở cắm.

sockopt/checkopts.c

```

53 int
54 chính(int argc, char **argv)
55 {
56     int      fd;
57     socklen_t len;
58     struct sock_opts *ptr;

59     for (ptr = sock_opts; ptr->opt_str != NULL; ptr++) {
60         printf("%s: ", ptr->opt_str);
61         nếu (ptr->opt_val_str == NULL)
62             printf("(không xác định)\n");

```

```

63         khác {
64             chuyển đổi (ptr->opt_level) {
65                 truờng hợp SOL_SOCKET:
66                 truờng hợp IPPROTO_IP:
67                 truờng hợp IPPROTO_TCP:
68                     fd = Ở cắm(AF_INET, SOCK_STREAM, 0);
69                     phá vỡ;
70 #ifdef IPV6
71                 truờng hợp IPPROTO_IPV6:
72                     fd = Ở cắm (AF_INET6, SOCK_STREAM, 0);
73                     phá vỡ;
74 #endif
75 #ifdef IPPROTO_SCTP
76                 truờng hợp IPPROTO_SCTP:
77                     fd = Ở cắm(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
78                     phá vỡ;
79 #endif
80             mặc định:
81                 err_quit("Không thể tạo fd cho cấp độ %d\n",
82                         ptr->opt_level);
83             }
84             len = sizeof(val);
85             if (getsockopt(fd, ptr->opt_level, ptr->opt_name,
86                             &val, &len) == -1) {
87                 err_ret("lỗi getsockopt");
88             } khác {
89                 printf("mặc định = %s\n", (*ptr->opt_val_str) (&val,
90                         chỉ một));
91             }
92         }
93         thoát (0);
94 }

```

Đi qua tất cả các tùy chọn

59-63 Chúng tôi xem xét tất cả các phần tử trong mảng của mình. Nếu con trả `opt_val_str` là null thì tùy chọn này không được xác định trong quá trình triển khai (mà chúng tôi đã hiển thị cho `SO_REUSEPORT`).

Tạo Ở cắm

63-82 Chúng tôi tạo một Ở cắm để thử tùy chọn. Để thử các tùy chọn Ở cắm lớp socket, IPv4 và TCP, chúng tôi sử dụng Ở cắm IPv4 TCP. Để thử các tùy chọn Ở cắm lớp IPv6, chúng tôi sử dụng Ở cắm IPv6 TCP và để thử các tùy chọn Ở cắm lớp SCTP, chúng tôi sử dụng SCTP IPv4 Ở cắm.

Gọi getsockopt

83-87 Chúng tôi gọi `getsockopt` như ng không kết thúc nếu trả về lỗi. Nhiều triển khai xác định một số tên tùy chọn ở cắm mặc dù chúng không hỗ trợ tùy chọn này. Các tùy chọn không được hỗ trợ sẽ gây ra lỗi `ENOPROTOOPT`.

Giá trị mặc định của tùy chọn in

88-89 Nếu `getsockopt` trả về thành công, chúng ta gọi hàm của mình để chuyển đổi giá trị tùy chọn thành một chuỗi và in chuỗi đó.

Trong [Hình 7.3](#), chúng tôi đã trình bày bốn nguyên mẫu hàm, một nguyên mẫu cho mỗi loại giá trị tùy chọn được trả về. [Hình 7.5](#) cho thấy một trong bốn hàm này, `sock_str_flag`, in giá trị của tùy chọn cờ. Ba chức năng còn lại tương tự.

Hình 7.5 Hàm `sock_str_flag`: chuyển đổi tùy chọn cờ thành một chuỗi.

`sockopt/checkopts.c`

```

95 chuỗi ký tự tinh[128];

96 ký tự tinh *

97 sock_str_flag(union val *ptr, int len)
98 {
99     if (len != sizeof(int))
100         snprintf(strres, sizeof(strres), "size (%d) không phải sizeof(int)",
chỉ một);
101     khác
102     snprintf(strres, sizeof(strres),
103             "%s", (ptr->i_val == 0) ? "tắt mở");
104     return(strres);
105 }
```

99-104 Hãy nhớ lại rằng đối số cuối cùng của `getsockopt` là đối số kết quả giá trị. Kiểm tra đầu tiên chúng tôi thực hiện là kích thước của giá trị được trả về bởi `getsockopt` có phải là kích thước dự kiến hay không. Chuỗi trả về tắt hoặc bật, tùy thuộc vào giá trị của tùy chọn cờ tương ứng là 0 hay khác 0.

Chạy chương trình này trong FreeBSD 4.8 với các bản vá KAME SCTP sẽ cho kết quả như sau:

kiểm tra % freebsd

SO_BROADCAST: mặc định = tắt
 SO_DEBUG: mặc định = tắt
 SO_DONTROUTE: mặc định = tắt
 SO_ERROR: mặc định = 0
 SO_KEEPALIVE: mặc định = tắt
 SO_LINGER: mặc định = l_onoff = 0, l_linger = 0
 SO_OOBINLINE: mặc định = tắt
 SO_RCVBUF: mặc định = 57344
 SO_SNDBUF: mặc định = 32768
 SO_RCVLOWAT: mặc định = 1
 SO SNDLOWAT: mặc định = 2048
 SO_RCVTIMEO: mặc định = 0 giây, 0 usec
 SO SNDTIMEO: mặc định = 0 giây, 0 usec
 SO_REUSEADDR: mặc định = tắt
 SO_REUSEPORT: mặc định = tắt
 SO_TYPE: mặc định = 1
 SO_USELOOPBACK: mặc định = tắt
 IP_TOS: mặc định = 0
 IP_TTL: mặc định = 64
 IPV6_DONTFRAG: mặc định = tắt
 IPV6_UNICAST_HOPS: mặc định = -1
 CHỈ IPV6_V6: mặc định = tắt
 TCP_MAXSEG: mặc định = 512
 TCP_NODELAY: mặc định = tắt
 SCTP_AUTOCLOSE: mặc định = 0
 SCTP_MAXBURST: mặc định = 4
 SCTP_MAXSEG: mặc định = 1408
 SCTP_NODELAY: mặc định = tắt

Giá trị 1 được trả về cho tùy chọn SO_TYPE tương ứng với SOCK_STREAM cho việc triển khai này.

7.4 Trạng thái ổ cắm

Một số tùy chọn ổ cắm có các cân nhắc về thời gian và thời điểm đặt hoặc tìm nạp tùy chọn so với trạng thái của ổ cắm. Chúng tôi đề cập đến những điều này với các tùy chọn bị ảnh hưởng.

Các tùy chọn ổ cắm sau được ổ cắm TCP đã kết nối kế thừa từ ổ cắm nghe (trang 462-463 của TCPv2):

SO_DEBUG, SO_DONTROUTE, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF, SO_RCVLOWAT, SO_SNDBUF, SO SNDLOWAT, TCP_MAXSEG và TCP_NODELAY. Điều này rất quan trọng với TCP vì ổ cắm đã kết nối không được trả về máy chủ bằng cách chấp nhận cho đến khi quá trình bắt tay ba chiều được hoàn thành bởi lớp TCP. Để đảm bảo rằng một trong các tùy chọn ổ cắm này được đặt cho

ở cắm được kết nối khi quá trình bắt tay ba chiều hoàn tất, chúng ta phải đặt tùy chọn đó cho ô cắm nghe.

7.5 Tùy chọn ô cắm chung

Chúng ta bắt đầu bằng việc thảo luận về các tùy chọn ô cắm chung. Các tùy chọn này độc lập với giao thức (nghĩa là chúng được xử lý bởi mã độc lập với giao thức trong kernel, không phải bởi một mô-đun giao thức cụ thể như IPv4), nhưng một số tùy chọn chỉ áp dụng cho một số loại ô cắm nhất định. Ví dụ: mặc dù tùy chọn ô cắm SO_BROADCAST được gọi là "chung", nhưng nó chỉ áp dụng cho ô cắm gói dữ liệu.

Tùy chọn ô cắm SO_BROADCAST

Tùy chọn này cho phép hoặc vô hiệu hóa khả năng gửi tin nhắn quảng bá của quy trình.

Việc phát sóng chỉ được hỗ trợ cho các ô cắm datagram và chỉ trên các mạng hỗ trợ khái niệm tin nhắn quảng bá (ví dụ: Ethernet, vòng mã thông báo, v.v.). Bạn không thể quảng bá trên liên kết điểm-diểm hoặc bất kỳ giao thức truyền tải dựa trên kết nối nào như SCTP hoặc TCP. Chúng ta sẽ nói nhiều hơn về việc phát sóng ở [Chương 20](#).

Vì ứng dụng phải đặt tùy chọn ô cắm này trước khi gửi gói dữ liệu quảng bá, nên nó sẽ ngăn quá trình gửi quảng bá khi ứng dụng chưa bao giờ được thiết kế để quảng bá. Ví dụ: ứng dụng UDP có thể lấy địa chỉ IP đích làm đối số dòng lệnh, nhưng ứng dụng đó không bao giờ có ý định cho người dùng nhập địa chỉ quảng bá. Thay vì buộc ứng dụng cố gắng xác định xem một địa chỉ nhất định có phải là địa chỉ quảng bá hay không, kiểm tra nằm trong kernel: Nếu địa chỉ đích là địa chỉ quảng bá và tùy chọn ô cắm này không được đặt, **EACCES** sẽ được trả về (tr. 233 của TCPv2).

Tùy chọn ô cắm SO_DEBUG

Tùy chọn này chỉ được hỗ trợ bởi TCP. Khi được bật cho ô cắm TCP, hạt nhân sẽ theo dõi thông tin chi tiết về tất cả các gói được gửi hoặc nhận bởi TCP cho ô cắm. Chúng được giữ trong một bộ đệm tròn trong kernel và có thể được kiểm tra bằng chương trình **trpt**. Trang 916-920 của TCPv2 cung cấp thêm chi tiết và ví dụ sử dụng tùy chọn này.

Tùy chọn ô cắm SO_DONTROUTE

Tùy chọn này chỉ định rằng các gói gửi đi sẽ bỏ qua các cơ chế định tuyến thông thường của giao thức cơ bản. Ví dụ: với IPv4, gói được chuyển hướng đến giao diện cục bộ thích hợp, như được chỉ định bởi các phần mạng và mạng con của địa chỉ đích. Nếu giao diện cục bộ không thể được xác định từ địa chỉ đích (ví dụ: đích không nằm ở đầu bên kia của liên kết điểm-diểm hoặc không nằm trên mạng chia sẻ), **ENETUNREACH** sẽ được trả về.

Tùy chọn tương ứng này cũng có thể được áp dụng cho các gói dữ liệu riêng lẻ bằng cách sử dụng cờ `MSG_DONTROUTE` với các chức năng `gửi`, `gửi tới` hoặc `gửi tin nhắn`.

Tùy chọn này thường được sử dụng bởi các daemon định tuyến (ví dụ: `định tuyến` và `kiểm soát`) để bỏ qua bảng định tuyến và buộc gói được gửi ra một giao diện cụ thể.

Tùy chọn ở cắm SO_ERROR

Khi xảy ra lỗi trên một ống cắm, mô-đun giao thức trong hạt nhân có nguồn gốc từ Berkeley sẽ đặt một biến có tên `so_error` cho ống cắm đó thành một trong các Unix Exxx tiêu chuẩn các giá trị. Đây được gọi là lỗi đang chờ xử lý đối với ống cắm. Quá trình có thể được thông báo ngay lập tức về lỗi theo một trong hai cách:

1. Nếu quá trình bị chặn trong cuộc gọi tới `select` trên ống cắm ([Phần 6.3](#)), đối với khả năng đọc hoặc khả năng ghi, `select` trả về với một hoặc cả hai điều kiện bô.
2. Nếu quy trình đang sử dụng I/O điều khiển bằng tín hiệu ([Chú ý](#) 25), tín hiệu `SIGIO` được tạo cho quy trình hoặc nhóm quy trình.

Sau đó, quá trình có thể lấy giá trị của `so_error` bằng cách tìm nạp tùy chọn ống cắm `SO_ERROR`. Giá trị số nguyên được trả về bởi `getsockopt` là lỗi đang chờ xử lý của ống cắm.

Giá trị của `so_error` sau đó được kernel đặt lại về 0 (tr. 547 của TCPv2).

Nếu `so_error` khác 0 khi tiến trình gọi `read` và không có dữ liệu nào để trả về, `read` trả về -1 với `errno` được đặt thành giá trị `so_error` (tr. 516 của TCPv2). Sau đó, giá trị của `so_error` được đặt lại về 0. Nếu có dữ liệu được xếp hàng đợi cho socket, dữ liệu đó sẽ được trả về bằng cách `đọc` thay vì điều kiện lỗi. Nếu `so_error` khác 0 khi quá trình ghi lệnh gọi , -1 được trả về với `errno` được đặt thành giá trị `so_error` (tr. 495 của TCPv2) và `so_error` được đặt lại về 0.

Có một lỗi trong mã hiển thị trên trang. 495 của TCPv2 trong đó `so_error` không được đặt lại về 0. Điều này đã được sửa trong hầu hết các bản phát hành hiện đại. Bắt đầu khi nào lỗi đang chờ xử lý của ống cắm được trả về, nó phải được đặt lại về 0.

Đây là tùy chọn ống cắm đầu tiên mà chúng tôi gặp phải có thể tìm nạp như ng không thể đặt.

Tùy chọn ống cắm SO_KEEPALIVE

Khi tùy chọn duy trì hoạt động được đặt cho ống cắm TCP và không có dữ liệu nào được trao đổi qua ống cắm theo một trong hai hướng trong hai giờ, TCP sẽ tự động gửi một đầu đờ duy trì hoạt động đến thiết bị ngang hàng. Thêm nữa là một phân đoạn TCP mà thiết bị ngang hàng phải phản hồi. Một trong ba kết quả kịch bản:

1. Máy ngang hàng phản hồi bằng ACK dự kiến. Ứng dụng không được thông báo (vì mọi thứ đều ổn). TCP sẽ gửi một thăm dò khác sau hai giờ không hoạt động.
2. Máy ngang hàng phản hồi bằng RST, thông báo cho TCP cục bộ rằng máy chủ ngang hàng đã gặp sự cố và đã khởi động lại. Lỗi đang chờ xử lý của ô cắm được đặt thành ECONNRESET và ô cắm đã đóng lại.
3. Không có phản hồi từ thiết bị ngang hàng đối với đầu dò duy trì hoạt động. Các TCP có nguồn gốc từ Berkeley gửi 8 đầu dò bổ sung, cách nhau 75 giây, cố gắng đưa ra phản hồi. TCP sẽ từ bỏ nếu không có phản hồi trong vòng 11 phút 15 giây sau khi gửi bản thăm dò đầu tiên.

HP-UX 11 xử lý các đầu dò duy trì theo cách tương tự như xử lý dữ liệu, gửi đầu dò thứ hai sau khi hết thời gian truyền lại và tăng gấp đôi thời gian chờ cho mỗi gói cho đến khoảng thời gian tối đa được định cấu hình, với giá trị mặc định trong 10 phút.

Nếu không có phản hồi nào đối với các đầu dò duy trì hoạt động của TCP thì lỗi đang chờ xử lý của ô cắm được đặt thành ETIMEDOUT và ô cắm bị đóng. Nhưng nếu ô cắm nhận được một lỗi ICMP phản ứng với một trong các đầu dò duy trì hoạt động, thay vào đó, lỗi tương ứng ([Hình A.15](#) và [A.16](#)) được trả về (và ô cắm vẫn đóng). Lỗi ICMP phổ biến trong trường hợp này là "không thể truy cập máy chủ", cho biết rằng máy chủ ngang hàng không thể truy cập được, trong trường hợp đó, lỗi đang chờ xử lý được đặt thành EHOSTUNREACH. Điều này có thể xảy ra do lỗi mạng hoặc do

bởi vì máy chủ từ xa đã gặp sự cố và bộ định tuyến bù ớc nhảy cuối cùng đã phát hiện ra sự cố.

Chú ý 23 của TCPv1 và trang 828-831 của TCPv2 chứa các chi tiết bổ sung về tùy chọn duy trì.

Chắc chắn câu hỏi phổ biến nhất liên quan đến tùy chọn này là liệu các tham số thời gian có thể được sửa đổi hay không (thường là để giảm khoảng thời gian hai giờ không hoạt động xuống một số giá trị ngắn hơn). Phụ lục E của TCPv1 thảo luận cách thay đổi các tham số thời gian này cho các hạt nhân khác nhau, nhưng lưu ý rằng hầu hết các hạt nhân duy trì các tham số này trên cơ sở từng hạt nhân chứ không phải trên cơ sở từng ô cắm, do đó thay đổi khoảng thời gian không hoạt động từ 2 giờ thành 15 phút, chẳng hạn, sẽ ảnh hưởng đến tất cả các ô cắm trên máy chủ kích hoạt tùy chọn này. Tuy nhiên, những câu hỏi như vậy thường xuất phát từ sự hiểu lầm về mục đích của lựa chọn này.

Mục đích của tùy chọn này là để phát hiện xem máy chủ ngang hàng có gặp sự cố hoặc không thể truy cập được hay không (ví dụ: mắt két nối modem quay số, mắt điện, v.v.). Nếu quá trình ngang hàng gặp sự cố, TCP của nó sẽ gửi FIN qua kết nối mà chúng ta có thể dễ dàng phát hiện bằng cách chọn. (Đây là lý do tại sao chúng tôi sử dụng tính năng chọn trong [Phần 6.4](#).) Cũng cần lưu ý rằng nếu không có phản hồi nào đối với bất kỳ thăm dò duy trì nào (kịch bản 3), chúng tôi không đảm bảo rằng máy chủ ngang hàng đã gặp sự cố và TCP có thể chấm dứt một kết nối hợp lệ. Có thể một số bộ định tuyến trung gian đã bị hỏng trong 15 phút,

và khoảng thời gian đó tình cờ trùng lặp hoàn toàn với khoảng thời gian thăm dò duy trì sự sống 11 phút 15 giây của máy chủ của chúng tôi. Trên thực tế, chức năng này có thể được gọi chính xác hơn là "làm chết" thay vì "giữ nguyên" vì nó có thể chấm dứt các kết nối trực tiếp.

Tùy chọn này thường được sử dụng bởi máy chủ, mặc dù máy khách cũng có thể sử dụng tùy chọn này.

Máy chủ sử dụng tùy chọn này vì chúng dành phần lớn thời gian bị chặn để chờ đầu vào qua kết nối TCP, nghĩa là chờ yêu cầu của máy khách. Nhưng nếu kết nối của máy chủ khách bị rớt, tắt nguồn hoặc gặp sự cố, quy trình máy chủ sẽ không bao giờ biết về điều đó và máy chủ sẽ liên tục chờ đợi đầu vào không bao giờ có thể đến được. Đây được gọi là kết nối nửa mở. Tùy chọn duy trì sẽ phát hiện các kết nối nửa mở này và chấm dứt chúng.

Một số máy chủ, đặc biệt là máy chủ FTP, cung cấp thời gian chờ ứng dụng, thường theo thứ tự vài phút. Điều này được thực hiện bởi chính ứng dụng, thường là xung quanh lệnh gọi `đọc`, đọc lệnh máy khách tiếp theo. Thời gian chờ này không liên quan đến tùy chọn ở cắm này.

Đây thường là phương pháp tốt hơn để loại bỏ kết nối tới các máy khách bị thiếu, vì ứng dụng có toàn quyền kiểm soát nếu nó tự triển khai thời gian chờ.

SCTP có cơ chế nhịp tim tương tự như cơ chế "giữ mạng" của TCP.

Cơ chế nhịp tim được điều khiển thông qua các tham số của tùy chọn ở cắm

`SCTP_SET_PEER_ADDR_PARAMS` được thảo luận sau trong chương này, thay vì tùy chọn ở cắm `SO_KEEPALIVE`. Các cài đặt do `SO_KEEPALIVE` thực hiện trên ở cắm SCTP sẽ bị bỏ qua và không ảnh hưởng đến cơ chế nhịp tim SCTP.

Hình 7.6 tóm tắt các phương pháp khác nhau mà chúng ta phải phát hiện khi có điều gì đó xảy ra ở đầu bên kia của kết nối TCP. Khi chúng tôi nói "sử dụng lựa chọn để có thể đọc được", chúng tôi muốn nói đến việc gọi `chọn` để kiểm tra xem ở cắm có thể đọc được hay không.

Hình 7.6. Các cách để phát hiện các điều kiện TCP khác nhau.

Scenario	Peer process crashes	Peer host crashes	Peer host is unreachable
Our TCP is actively sending data	Peer TCP sends a FIN, which we can detect immediately using <code>select</code> for readability. If TCP sends another segment, peer TCP responds with an RST. If the application attempts to write to the socket after TCP has received an RST, our socket implementation sends us <code>SIGPIPE</code> .	Our TCP will time out and our socket's pending error will be set to <code>ETIMEDOUT</code> .	Our TCP will time out and our socket's pending error will be set to <code>EHOSTUNREACH</code> .
Our TCP is actively receiving data	Peer TCP will send a FIN, which we will read as a (possibly premature) EOF.	We will stop receiving data.	We will stop receiving data.
Connection is idle, keep-alive set	Peer TCP sends a FIN, which we can detect immediately using <code>select</code> for readability.	Nine keep-alive probes are sent after two hours of inactivity and then our socket's pending error is set to <code>ETIMEDOUT</code> .	Nine keep-alive probes are sent after two hours of inactivity and then our socket's pending error is set to <code>EHOSTUNREACH</code> .
Connection is idle, keep-alive not set	Peer TCP sends a FIN, which we can detect immediately using <code>select</code> for readability.	(Nothing)	(Nothing)

Tùy chọn ở cắm SO_Linger

Tùy chọn này chỉ định cách hoạt động của chức năng đóng đối với giao thức hưu ứng kết nối (ví dụ: đối với TCP và SCTP, nhưng không dành cho UDP). Theo mặc định, lệnh đóng trả về ngay lập tức, nhưng nếu vẫn còn bất kỳ dữ liệu nào trong bộ đệm gửi ở cắm, hệ thống sẽ cố gắng phân phối dữ liệu đến thiết bị ngang hàng.

Tùy chọn ở cắm **SO_Linger** cho phép chúng tôi thay đổi mặc định này. Tùy chọn này yêu cầu cấu trúc sau được chuyển giữa tiến trình người dùng và kernel. Nó được xác định bằng cách bao gồm `<sys/socket.h>`.

```
cấu trúc nán lại {
    int l_onoff; int             /* 0=tắt, khác 0=bật */
    l_linger; };                  /* thời gian kéo dài, POSIX chỉ định đơn vị là giây */
```

Gọi `setsockopt` dẫn đến một trong ba trường hợp sau, tùy thuộc vào giá trị của hai thành viên cấu trúc:

1. Nếu `l_onoff` bằng 0, tùy chọn này sẽ bị tắt. Giá trị của `l_linger` bị bỏ qua và áp dụng mặc định TCP đã thảo luận trước đó: đóng trả về ngay lập tức.
2. Nếu `l_onoff` khác 0 và `l_linger` bằng 0, TCP sẽ hủy kết nối khi nó bị đóng (trang 1019-1020 của TCPv2). Nghĩa là, TCP loại bỏ mọi dữ liệu còn lại trong bộ đệm gửi ở cắm và gửi RST đến thiết bị ngang hàng, chứ không phải trình tự kết thúc kết nối bốn gói thông thường ([Phần 2.6](#)). Chúng ta sẽ trình bày một ví dụ về điều này trong [Hình 16.21](#). Điều này tránh trạng thái TIME_WAIT của TCP, nhưng khi làm nhu vây, sẽ để ngỏ khả năng một phiên bản khác của kết nối này được tạo trong vòng 2MSL giây ([Phần 2.7](#)) và có các phân đoạn trùng lặp cũ từ kết nối vừa kết thúc không chính xác

được chuyển giao cho sự tái sinh mới.

SCTP cũng sẽ thực hiện việc đóng ở cắm bị hủy bỏ bằng cách gửi một đoạn ABORT đến thiết bị ngang hàng (xem [Phần 9.2](#) của [Stewart và Xie 2001]) khi `l_onoff` khác 0 và `l_linger` bằng 0.

Các bài đăng thỉnh thoảng của USENET ủng hộ việc sử dụng tính năng này chỉ để tránh trạng thái TIME_WAIT và để có thể khởi động lại máy chủ đang lắng nghe ngay cả khi các kết nối vẫn đang được sử dụng với công phu biến của máy chủ. Điều này KHÔNG nên được thực hiện và có thể dẫn đến hỏng dữ liệu, như được nêu chi tiết trong RFC 1337 [Braden 1992]. Thay vào đó, tùy chọn ở cắm **SO_REUSEADDR** phải luôn được sử dụng trong máy chủ truy cập lệnh gọi liên kết, như chúng tôi sẽ mô tả sau. TIME_WAIT

state là bạn của chúng tôi và luôn sẵn sàng giúp đỡ chúng tôi (tức là để các phân đoạn trùng lặp cũ hết hạn trong mạng). Thay vì cố trốn tránh trạng thái, chúng ta nên hiểu nó ([Phần 2.7](#)).

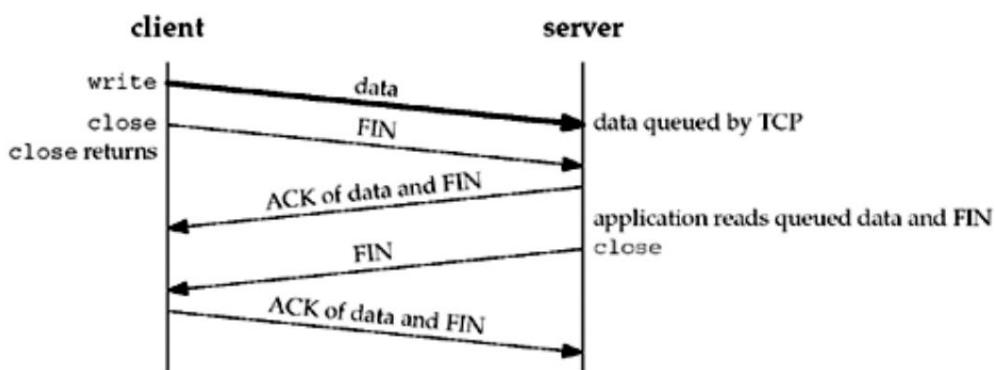
Có một số trường hợp nhất định cần sử dụng tính năng này để gửi một kết thúc hủy bỏ. Một ví dụ là máy chủ đầu cuối RS-232, có thể bị treo vĩnh viễn trong CLOSE_WAIT khi cố gắng phân phối dữ liệu đến cổng đầu cuối bị tấn công nhưng sẽ đặt lại cổng bị kẹt đúng cách nếu nó có RST để loại bỏ dữ liệu đang chờ xử lý.

3. Nếu `l_onoff` khác 0 và `l_linger` khác 0 thì kernel sẽ tồn tại

khi ở cắm được đóng (tr. 472 của TCPv2). Nghĩa là, nếu vẫn còn bất kỳ dữ liệu nào trong bộ đệm gửi ở cắm, quy trình sẽ ở chế độ ngủ cho đến khi: (i) tất cả dữ liệu được gửi và xác nhận bởi TCP ngang hàng hoặc (ii) thời gian kéo dài hết hạn. Nếu ở cắm đã được đặt thành không chặn ([Chu ơng 16](#)), nó sẽ không đợi quá `trình đóng hoàn tất`, ngay cả khi thời gian kéo dài khác 0. Khi sử dụng tính năng này của tùy chọn `SO_LINGER`, điều quan trọng là ứng dụng phải kiểm tra giá trị trả về `khi đóng`, vì nếu thời gian kéo dài hết trớn khi dữ liệu còn lại được gửi và xác nhận, thì lệnh `đóng` sẽ trả về `EWOULDBLOCK` và mọi dữ liệu còn lại trong bộ đệm gửi sẽ bị loại bỏ. bị loại bỏ.

Bây giờ chúng ta cần biết chính xác thời điểm đóng socket trở lại với các điều kiện khác nhau kịch bản chúng tôi đã xem xét. Chúng ta giả định rằng máy khách ghi dữ liệu vào socket và sau đó cuộc gọi `gắn gửi`. [Hình 7.7](#) thể hiện tình huống mặc định.

Hình 7.7. Hoạt động mặc định của đóng: nó trả về ngay lập tức.

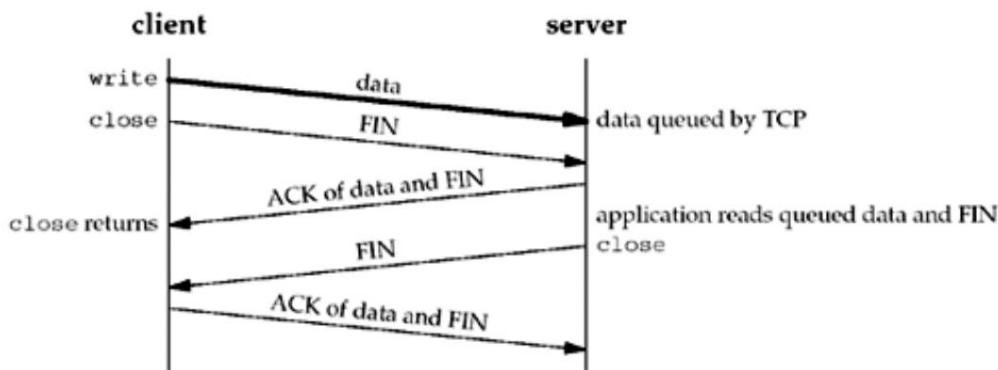


Chúng tôi giả định rằng khi dữ liệu của máy khách đến, máy chủ tạm thời bận, do đó dữ liệu sẽ được thêm vào bộ đệm nhận ở cắm bằng TCP của nó. Tương tự, phân đoạn tiếp theo, FIN của máy khách, cũng được thêm vào bộ đệm nhận ở cắm (theo bất kỳ cách nào việc triển khai ghi lại rằng FIN đã được nhận trên kết nối). Nhưng theo mặc định, lệnh `đóng` của khách hàng sẽ trả về ngay lập tức. Như chúng tôi trình bày trong kịch bản này, việc đóng của máy khách có thể quay lại trước khi máy chủ đọc dữ liệu còn lại trong bộ đệm nhận ở cắm của nó. Vì vậy, có thể máy chủ máy chủ gặp sự cố trước khi ứng dụng máy chủ đọc được dữ liệu còn lại này và ứng dụng máy khách sẽ không bao giờ biết được.

Máy khách có thể đặt tùy chọn ô cắm `SO_LINGER`, chỉ định một số thời gian kéo dài tích cực.

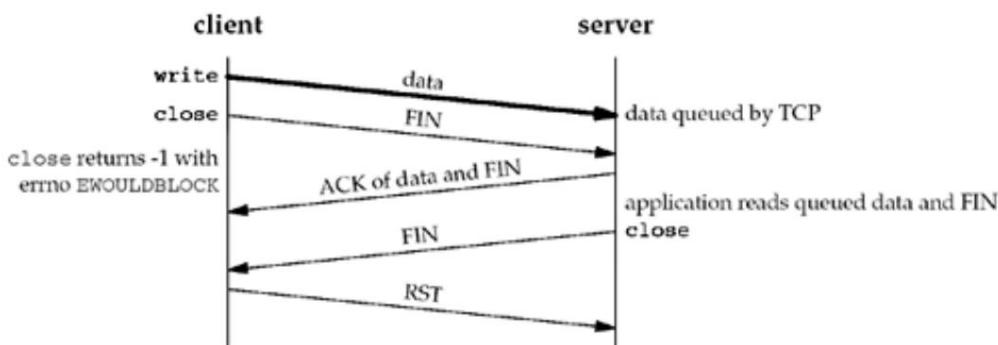
Khi điều này xảy ra, **việc đóng** của máy khách sẽ không quay trở lại cho đến khi tất cả dữ liệu của máy khách và FIN của nó đã được máy chủ TCP xác nhận. Chúng tôi thể hiện điều này trong [Hình 7.8](#).

Hình 7.8. đóng với tùy chọn ô cắm `SO_LINGER` được đặt và `l_linger` một giá trị dương.



Như ng chúng ta vẫn gặp vấn đề tương tự như trong [Hình 7.7](#): Máy chủ máy chủ có thể gặp sự cố trễ chờ khi ứng dụng máy chủ đọc dữ liệu còn lại của nó và ứng dụng khách sẽ không bao giờ biết được. Tôi tệ hơn, [Hình 7.9](#) cho thấy điều gì có thể xảy ra khi tùy chọn `SO_Linger` được đặt ở giá trị quá thấp.

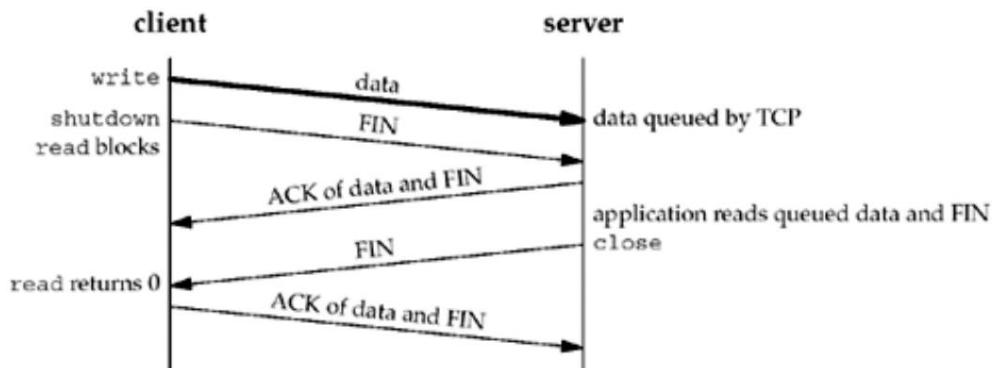
Hình 7.9. đóng bằng bộ tùy chọn ô cắm `SO_Linger` và `l_linger` một giá trị dương nhỏ.



Nguyên tắc cơ bản ở đây là việc quay lại thành công sau khi **đóng**, với `SO_Linger` bộ tùy chọn socket chỉ cho chúng tôi biết rằng dữ liệu chúng tôi đã gửi (và FIN của chúng tôi) đã được TCP ngang hàng xác nhận. Điều này không cho chúng tôi biết liệu ứng dụng ngang hàng đã đọc dữ liệu hay chưa. Nếu chúng tôi không đặt tùy chọn ô cắm `SO_Linger`, chúng tôi sẽ không biết liệu TCP ngang hàng có thừa nhận dữ liệu hay không.

Một cách để client biết rằng máy chủ đã đọc dữ liệu của nó là gọi lệnh **tắt máy** (với đối số thứ hai là `SHUT_WR`) thay vì **đóng** và đợi ngang hàng **đóng** kết thúc của nó. Chúng tôi hiển thị kịch bản này trong [Hình 7.10](#).

Hình 7.10. Sử dụng tắt máy để biết rằng thiết bị ngang hàng đó đã nhận được dữ liệu của chúng tôi.



So sánh hình này với [Hình 7.7](#) và [7.8](#), chúng ta thấy rằng khi chúng ta đóng phần cuối của kết nối, tùy thuộc vào chức năng được gọi ([đóng hoặc tắt](#)) và tùy chọn ô cắm `SO_LINGER` có được đặt hay không, việc quay lại có thể xảy ra ở ba thời điểm khác nhau:

1. đóng trả về [ngay](#) lập tức mà không cần chờ đợi (mặc định; [Hình 7.7](#)).
2. [đóng](#) lại cho đến khi nhận được ACK của FIN của chúng tôi ([Hình 7.7](#)).
3. [tắt](#) máy sau đó [đọc](#) sẽ đợi cho đến khi chúng tôi nhận được FIN của thiết bị ngang hàng ([Hình 7.10](#)).

Một cách khác để biết ứng dụng ngang hàng đã đọc dữ liệu của chúng ta là sử dụng xác nhận cấp ứng dụng hoặc ACK ứng dụng. Ví dụ: trong phần sau, máy khách gửi dữ liệu của mình đến máy chủ và sau đó gọi lệnh [đọc](#) cho một byte dữ liệu:

```

char ack;

Viết(sockfd, dữ liệu, nbyte);           /*dữ liệu từ máy khách đến máy chủ */
n = Đọc(sockfd, &ack, 1);                /*đợi ACK cấp ứng dụng */

```

Máy chủ đọc dữ liệu từ máy khách và sau đó gửi lại ACK cấp ứng dụng một byte:

```

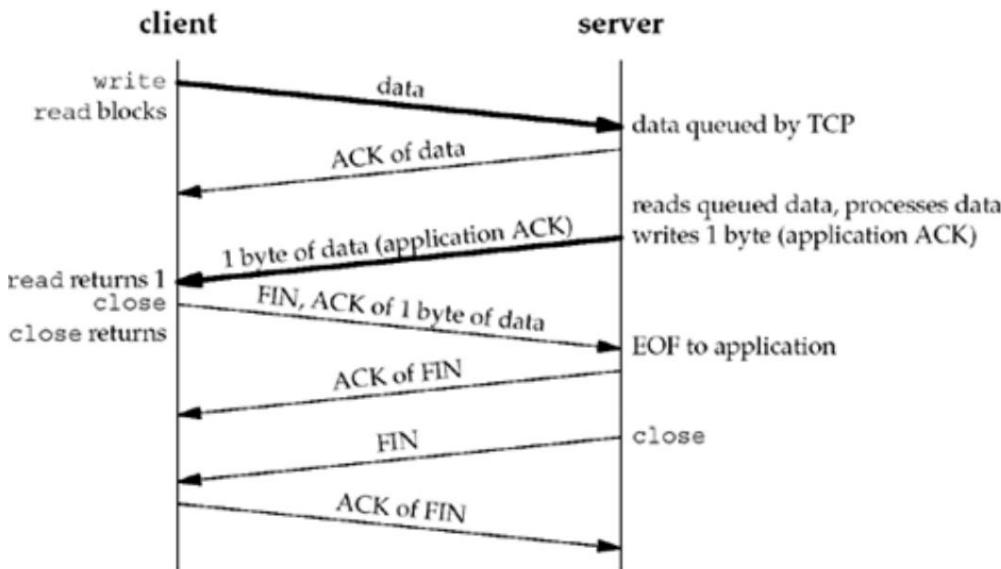
nbytes = Đọc(sockfd, buff, sizeof(buff)); /*dữ liệu từ máy khách*/
/* máy chủ xác minh nó đã nhận được đúng
lượng dữ liệu từ khách hàng */
Write(sockfd, "", 1);                    /* ACK của máy chủ quay lại máy khách */

```

Chúng tôi được đảm bảo rằng khi quá trình đọc trong máy khách trả về, quy trình máy chủ đã đọc dữ liệu chúng tôi đã gửi. (Điều này giả định rằng máy chủ biết lượng dữ liệu mà khách hàng đang gửi hoặc có một số điểm đánh dấu cuối bản ghi do ứng dụng xác định mà chúng tôi không hiển thị ở đây.) Ở đây, ACK cấp ứng dụng là một byte bằng 0, nhưng nội dung của byte này có thể được sử dụng để báo hiệu các điều kiện khác từ máy chủ đến máy khách.

[Hình 7.11 cho thấy khả năng trao đổi gói tin.](#)

Hình 7.11. Ứng dụng ACK.



[Hình 7.12](#) tóm tắt hai lệnh gọi có thể tắt máy và ba lệnh gọi có thể đóng và ảnh hưởng đến ỗ cảm TCP.

Hình 7.12. Tóm tắt các tình huống tắt máy và SO_Linger.

Function	Description
shutdown, SHUT_RD	No more receives can be issued on socket; process can still send on socket; socket receive buffer discarded; any further data received is discarded by TCP (Exercise 6.5); no effect on socket send buffer.
shutdown, SHUT_WR	No more sends can be issued on socket; process can still receive on socket; contents of socket send buffer sent to other end, followed by normal TCP connection termination (FIN); no effect on socket receive buffer.
close, l_onoff = 0 (default)	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer and socket receive buffer discarded.
close, l_onoff = 1 l_linger = 0	No more receives or sends can be issued on socket. If descriptor reference count becomes 0: RST sent to other end; connection state set to CLOSED (no TIME_WAIT state); socket send buffer and socket receive buffer discarded.
close, l_onoff = 1 l_linger != 0	No more receives or sends can be issued on socket; contents of socket send buffer sent to other end. If descriptor reference count becomes 0: normal TCP connection termination (FIN) sent following data in send buffer; socket receive buffer discarded; and if linger time expires before connection CLOSED, close returns EWOULDBLOCK.

Tùy chọn ô cắm SO_OOBINLINE

Khi tùy chọn này được đặt, dữ liệu ngoài băng tần sẽ được đặt trong hàng đợi đầu vào thông thư ờng (tức là nội tuyến). Khi điều này xảy ra, cờ [MSG_OOB](#) cho các chức năng nhận không thể được sử dụng để đọc dữ liệu ngoài băng tần. Chúng ta sẽ thảo luận chi tiết hơn về dữ liệu ngoài băng trong [Chương 24](#).

Tùy chọn ô cắm SO_RCVBUF và SO_SNDBUF

Mỗi socket đều có bộ đệm gửi và bộ đệm nhận. Chúng tôi đã mô tả hoạt động của bộ đệm gửi với TCP, UDP và SCTP trong [Hình 2.15, 2.16](#) và [2.17](#).

Bộ đệm nhận được TCP, UDP và SCTP sử dụng để giữ dữ liệu đã nhận cho đến khi ứng dụng đọc được. Với TCP, dung lượng trống trong socket nhận bộ đệm

giới hạn cửa sổ mà TCP có thể quảng cáo tới đầu bên kia. Ô cắm TCP nhận bộ đệm không thể tràn vì thiết bị ngang hàng không được phép gửi dữ liệu ngoài cửa sổ được quảng cáo. Đây là cơ chế kiểm soát luồng của TCP và nếu thiết bị ngang hàng bỏ qua cửa sổ được quảng cáo và gửi dữ liệu ra ngoài cửa sổ thì TCP nhận sẽ loại bỏ nó. Tuy nhiên, với UDP, khi một datagram đến không vừa với bộ đệm nhận socket, datagram đó sẽ bị loại bỏ. Hãy nhớ lại rằng UDP không có khả năng kiểm soát luồng: Người gửi nhanh có thể dễ dàng áp đảo người nhận chậm hơn, khiến các datagram bị UDP của người nhận loại bỏ, như chúng tôi sẽ trình bày trong [Phần 8.13](#). Trên thực tế, người gửi nhanh có thể làm quá tải giao diện mạng của chính họ, khiến các gói dữ liệu bị chính người gửi loại bỏ.

Hai tùy chọn ô cắm này cho phép chúng tôi thay đổi kích thước mặc định. Các giá trị mặc định rất khác nhau giữa các lần triển khai. Các triển khai cũ hơn có nguồn gốc từ Berkeley sẽ mặc định bộ đệm gửi và nhận TCP ở mức 4.096 byte, như những hệ thống mới hơn sử dụng các giá trị lớn hơn, từ 8.192 đến 61.440 byte. Kích thước bộ đệm gửi UDP thư ờng được mặc định là khoảng 9.000 byte nếu máy chủ hỗ trợ NFS và kích thước bộ đệm nhận UDP thư ờng được mặc định là giá trị khoảng 40.000 byte.

Khi thiết lập kích thước của bộ đệm nhận socket TCP, thứ tự của các lệnh gọi hàm rất quan trọng. Điều này là do tùy chọn tỷ lệ cửa sổ của TCP ([Phần 2.6](#)), được trao đổi với thiết bị ngang hàng trên các phân đoạn SYN kết nối được thiết lập.

Đối với máy khách, điều này có nghĩa là tùy chọn ô cắm [SO_RCVBUF](#) phải được đặt trước khi gọi [kết nối](#). Đối với máy chủ, điều này có nghĩa là tùy chọn ô cắm phải được đặt cho ô cắm nghe trước khi gọi [listen](#). Việc đặt tùy chọn này cho ô cắm được kết nối sẽ không có tác dụng gì đối với tùy chọn tỷ lệ cửa sổ có thể có vì [việc chấp nhận](#) không quay trở lại với ô cắm được kết nối cho đến khi quá trình bắt tay ba chiều của TCP hoàn tất. Đó là lý do tại sao tùy chọn này phải được đặt cho socket nghe. (Kích thước của bộ đệm ô cắm luôn được kế thừa từ ô cắm nghe bởi ô cắm được kết nối mới được tạo: trang 462-463 của TCPv2.)

Kích thước bộ đệm ô cắm TCP phải gấp ít nhất bốn lần MSS cho kết nối.

Nếu chúng ta đang xử lý việc truyền dữ liệu một chiều, chẳng hạn như truyền tệp trong một

hư ứng, khi chúng tôi nói "kích thư ớc bộ đệm ở cắm", chúng tôi muốn nói đến kích thư ớc bộ đệm gửi ở cắm trên máy chủ gửi và kích thư ớc bộ đệm nhận ở cắm trên máy chủ nhận. Đối với truyền dữ liệu hai chiều, chúng tôi muốn nói đến cả kích thư ớc bộ đệm ở cắm trên máy gửi và cả kích thư ớc bộ đệm ở cắm trên máy thu. Với kích thư ớc bộ đệm mặc định điển hình là 8.192 byte hoặc lớn hơn và MSS điển hình là 512 hoặc 1.460, yêu cầu này thường được đáp ứng.

Bội số MSS tối thiểu của 4 là kết quả của cách thức hoạt động của thuật toán khôi phục nhanh của TCP. Người gửi TCP sử dụng ba xác nhận trùng lặp để phát hiện gói tin bị mất (RFC 2581 [Allman, Paxson và Stevens 1999]). Người nhận gửi một bản xác nhận trùng lặp cho mỗi phân đoạn mà nó nhận được sau khi mất phân đoạn.

Nếu kích thư ớc của số nhỏ hơn bốn phân đoạn thì không thể có ba xác nhận trùng lặp, do đó không thể gọi thuật toán khôi phục nhanh.

Để tránh lãng phí không gian bộ đệm tiềm năng, kích thư ớc bộ đệm ở cắm TCP cũng phải là bội số chẵn của MSS cho kết nối. Một số triển khai xử lý chi tiết này cho ứng dụng, làm tròn kích thư ớc bộ đệm ở cắm sau khi kết nối được thiết lập (tr. 902 của TCPv2). Đây là một lý do khác để đặt hai tùy chọn ở cắm này trước khi thiết lập kết nối. Ví dụ: sử dụng kích thư ớc 4.4BSD mặc định là 8.192 và giả sử Ethernet có MSS là 1.460, cả hai bộ đệm ở cắm đều được làm tròn thành 8.760 (6×1.460) khi kết nối được thiết lập. Đây không phải là một yêu cầu quan trọng; không gian bổ sung trong bộ đệm ở cắm phía trên bội số của MSS đơn giản là không được sử dụng.

Một cân nhắc khác trong việc thiết lập kích thư ớc bộ đệm ở cắm liên quan đến hiệu suất.

Hình 7.13 hiển thị kết nối TCP giữa hai điểm cuối (mà chúng tôi gọi là đường ống) với dung lư ợng tám đoạn.

Hình 7.13. Kết nối TCP (ống) có dung lư ợng tám đoạn.



Điều quan trọng cần hiểu là khái niệm về đư ờng óng song công hoàn toàn, công suất của nó và điều đó liên quan như thế nào đến kích thư ớc bộ đệm ở cắm ở cả hai đầu của kết nối. Các công suất của đư ờng óng đư ợc gọi là tich độ trẽ băng thông và chúng tôi tính toán điều này bằng cách nhân băng thông (tính bằng bit/giây) với RTT (tính bằng giây), chuyển đổi kết quả từ bit sang byte. RTT có thể dễ dàng đo đư ợc bằng chương trình [ping](#).

Băng thông là giá trị tương ứng với liên kết chậm nhất giữa hai điểm cuối và phải đư ợc biết bằng cách nào đó. Ví dụ: một đư ờng T1 (1.536.000 bit/giây) với RTT là 60 ms sẽ tạo ra tich độ trẽ băng thông là 11.520 byte. Nếu kích thư ớc bộ đệm ở cắm nhỏ hơn kích thư ớc này, đư ờng óng sẽ không đầy và hiệu suất sẽ thấp hơn mong đợi. Cần có bộ đệm ở cắm lớn khi băng thông lớn hơn (ví dụ: đư ờng T3 ở tốc độ 45 Mbit/giây) hoặc khi RTT trở nên lớn (ví dụ: liên kết vệ tinh có RTT khoảng 500 ms). Khi sản phẩm có độ trẽ băng thông vư ợt quá kích thư ớc cửa sổ thông thư ờng tối đa của TCP (65.535 byte), cả hai điểm cuối cũng cần óng dẫn dài TCP

các tùy chọn mà chúng tôi đã đề cập ở [Phần 2.6](#).

Hầu hết các triển khai đều có giới hạn trên đối với kích thư ớc của bộ đệm gửi và nhận ở cắm và đôi khi quản trị viên có thể sửa đổi giới hạn này. Các triển khai cũ hơn có nguồn gốc từ Berkeley có giới hạn trên cứng là khoảng 52.000 byte, nhưng các triển khai mới hơn có giới hạn mặc định là 256.000 byte trở lên và quản trị viên thư ờng có thể tăng giới hạn này. Thật không may, không có cách đơn giản nào để ứng dụng xác định giới hạn này. POSIX xác định hàm [fpathconf](#) mà hầu hết các triển khai đều hỗ trợ và sử dụng hằng số [_PC_SOCK_MAXBUF](#) làm đối số thứ hai, chúng ta có thể truy xuất kích thư ớc tối đa của bộ đệm ở cắm.

Ngoài ra, một ứng dụng có thể thử đặt bộ đệm ở cắm thành giá trị mong muốn và nếu thất bại, hãy giảm giá trị xuống một nửa và thử lại cho đến khi thành công. Cuối cùng, ứng dụng phải đảm bảo rằng nó không thực sự làm cho bộ đệm ở cắm nhỏ hơn khi đặt nó thành giá trị "lớn" đư ợc cấu hình sẵn; trước tiên hãy gọi [getsockopt](#) để truy xuất giá trị mặc định của hệ thống và xem liệu nó có đủ lớn hay không thư ờng là một khởi đầu tốt.

Tùy chọn ở cắm SO_RCVLOWAT và SO SNDLOWAT

Mỗi ở cắm cũng có dấu hiệu nhận ít nư ớc và dấu gửi nhận ít nư ớc. Chúng đư ợc sử dụng bởi hàm [select](#), như chúng tôi đã mô tả trong [Phần 6.3](#). Hai tùy chọn ở cắm này, [SO_RCVLOWAT](#) và [SO SNDLOWAT](#), chúng ta hãy thay đổi hai vạch nư ớc thấp này.

Dấu nhận ít nư ớc là lư ợng dữ liệu phải có trong ở cắm nhận bộ đệm để [chọn](#) trả về "có thẻ đọc đư ợc". Nó mặc định là 1 cho các ở cắm TCP, UDP và SCTP. Dấu hiệu sắp hết nư ớc gửi là lư ợng không gian có sẵn phải tồn tại trong bộ đệm gửi ở cắm để [chọn](#) trả về "có thẻ ghi". Dấu hiệu mức nư ớc thấp này thư ờng đư ợc mặc định là 2.048 đối với ở cắm TCP. Với UDP, dấu mức nư ớc thấp đư ợc sử dụng, vì chúng tôi đư ợc mô tả trong [Phần 6.3](#), như ng do số byte dung lư ợng trống trong bộ đệm gửi cho ở cắm UDP không bao giờ thay đổi (vì UDP không giữ bản sao của các gói dữ liệu đư ợc gửi bởi ứng dụng), miễn là kích thư ớc bộ đệm gửi của ở cắm UDP là

lớn hơn vạch mục nút thấp của ô cắm, ô cắm UDP luôn có thẻ ghi đư ợc. Nhờ lại [Hình 2.16](#) rằng UDP không có bộ đêm gửi; nó chỉ có kích thư ớc bộ đệm gửi.

Tùy chọn ô cắm SO_RCVTIMEO và SO_SNDFTIMEO

Hai tùy chọn ô cắm này cho phép chúng ta đặt thời gian chờ khi nhận và gửi ô cắm.

Lưu ý rằng đối số của hai hàm `sockopt` là một con trỏ tới một **khoảng thời gian** cấu trúc tương tự như cấu trúc đư ợc sử dụng với [select \(Phần 6.3\)](#). Điều này cho phép chúng tôi chỉ định thời gian chờ tính bằng giây và micro giây. Chúng tôi vô hiệu hóa thời gian chờ bằng cách đặt giá trị của nó thành 0 giây và 0 micro giây. Cả hai thời gian chờ đều bị tắt theo mặc định.

Thời gian chờ nhận ảnh hưởng đến năm chức năng đầu vào: `read`, `readv`, `recv`, `recvfrom` và `recvmsg`. Thời gian chờ gửi ảnh hưởng đến năm chức năng đầu ra: `write`, `writev`, `send`, `sendto` và `sendmsg`. Chúng ta sẽ nói nhiều hơn về thời gian chờ của socket trong [Phần 14.2](#).

Hai tùy chọn ô cắm này và khái niệm về thời gian chờ vốn có khi nhận và gửi ô cắm đã đư ợc thêm vào 4.3BSD Reno.

Trong các triển khai bắt nguồn từ Berkeley, hai giá trị này thực sự triển khai bộ hẹn giờ không hoạt động chứ không phải bộ hẹn giờ tuyệt đối cho lệnh gọi hệ thống đọc hoặc ghi. Trang 496 và 516 của TCPv2 nói về điều này chi tiết hơn.

Tùy chọn ô cắm SO_REUSEADDR và SO_REUSEPORT

Tùy chọn ô cắm `SO_REUSEADDR` phục vụ bốn mục đích khác nhau:

1. `SO_REUSEADDR` cho phép máy chủ nghe khởi động và **liên kết** cổng nổi tiếng của nó, ngay cả khi tồn tại các kết nối đư ợc thiết lập trước đó sử dụng cổng này làm cổng cục bộ. Tình trạng này thư ờng gặp như sau:
 - a. Một máy chủ lắng nghe đư ợc bắt đầu.
 - b. Một yêu cầu kết nối đến và một tiến trình con đư ợc sinh ra để xử lý khách hàng đó.
 - c. Máy chủ lắng nghe chấm dứt, như ng máy chủ con vẫn tiếp tục phục vụ máy khách trên kết nối hiện có.
 - d. Máy chủ nghe đư ợc khởi động lại.

Theo mặc định, khi máy chủ nghe đư ợc khởi động lại trong (d) bằng cách gọi `socket`, **liên kết** và **lắng nghe**, lệnh gọi **liên kết** không thành công vì máy chủ nghe đang cố gắng liên kết một cổng là một phần của kết nối hiện có (cổng đư ợc xử lý bởi đứa trẻ đư ợc sinh ra trước đó). Như ng nếu máy chủ đặt tùy chọn ô cắm `SO_REUSEADDR` giữa các lệnh gọi đến ô cắm và **liên kết** thì chức năng sau sẽ thành công. Tất cả các máy chủ TCP phải chỉ định tùy chọn ô cắm này để cho phép máy chủ đư ợc khởi động lại trong trường hợp này.

2. Kịch bản này là một trong những câu hỏi thư ờng gặp nhất trên USENET.

3. `SO_REUSEADDR` cho phép khởi động một máy chủ mới trên cùng một cổng với máy chủ hiện có liên kết với địa chỉ tự đại diện, miễn là mỗi phiên bản liên kết với một địa chỉ IP khác nhau. Điều này thường xảy ra đối với một trang web lưu trữ nhiều máy chủ HTTP sử dụng kỹ thuật bí danh IP ([Phần A.4](#)). Giả sử địa chỉ IP chính của máy chủ cục bộ là 198.69.10.2 nhưng nó có hai bí danh: 198.69.10.128 và 198.69.10.129. Ba máy chủ HTTP được khởi động. Máy chủ HTTP đầu tiên sẽ gọi **liên kết** với ký tự đại diện làm địa chỉ IP cục bộ và cổng cục bộ là 80 (cổng phổ biến cho HTTP). Máy chủ thứ hai sẽ gọi **liên kết** với địa chỉ IP cục bộ là 198.69.10.128 và cổng cục bộ là 80. Tuy nhiên, lệnh gọi **liên kết** thứ hai này không thành công trừ khi `SO_REUSEADDR` được đặt trước cuộc gọi. Máy chủ thứ ba sẽ **liên kết** 198.69.10.129 và cổng 80. Một lần nữa, cần có `SO_REUSEADDR` để lệnh gọi cuối cùng này thành công. Giả sử `SO_REUSEADDR` được đặt và ba máy chủ được khởi động, các yêu cầu kết nối TCP đến có địa chỉ IP đích là 198.69.10.128 và cổng đích là 80 sẽ được gửi đến máy chủ thứ hai, các yêu cầu đến có địa chỉ IP đích là 198.69.10.129 và một cổng đích 80 được gửi đến máy chủ thứ ba và tất cả các yêu cầu kết nối TCP khác có cổng đích 80 sẽ được gửi đến máy chủ đầu tiên. Máy chủ "mặc định" này xử lý các yêu cầu dành cho 198.69.10.2 bên cạnh bất kỳ bí danh IP nào khác mà máy chủ có thể đã định cấu hình. Ký tự đại diện có nghĩa là "mọi thứ không phù hợp hơn (cụ thể hơn)".

Lưu ý rằng kịch bản cho phép nhiều máy chủ cho một dịch vụ nhất định được xử lý tự động nếu máy chủ luôn đặt tùy chọn ô cẩm `SO_REUSEADDR` (như chúng tôi khuyến nghị).

Với TCP, chúng tôi không bao giờ có thể khởi động nhiều máy chủ **liên kết** cùng một địa chỉ IP và cùng một cổng: một liên kết hoàn toàn trùng lặp. Nghĩa là, chúng tôi không thể khởi động một máy chủ liên kết 198.69.10.2 cổng 80 và khởi động một máy chủ khác cũng liên kết 198.69.10.2 cổng 80, ngay cả khi chúng tôi đặt tùy chọn ô cẩm `SO_REUSEADDR` cho máy chủ thứ hai.

Vì lý do bảo mật, một số hệ điều hành ngăn chặn mọi liên kết "cụ thể hơn" với một cổng đã được liên kết với địa chỉ ký tự đại diện, nghĩa là chuỗi liên kết được mô tả ở đây sẽ không hoạt động khi có hoặc không có `SO_REUSEADDR`. Trên hệ thống như vậy, máy chủ thực hiện liên kết ký tự đại diện phải được khởi động sau cùng.

Điều này là để tránh sự cố máy chủ giả mạo liên kết với địa chỉ IP và cổng đang được dịch vụ hệ thống phục vụ và chặn các yêu cầu hợp pháp. Đây là một vấn đề cụ thể đối với NFS, thư ờng không sử dụng cổng đặc quyền.

4. `SO_REUSEADDR` cho phép một tiến trình liên kết cùng một cổng với nhiều tiến trình socket, miễn là mỗi liên kết chỉ định một địa chỉ IP cục bộ khác nhau. Đây là phổ biến cho các máy chủ UDP cần biết địa chỉ IP đích của máy khách yêu cầu trên các hệ thống không cung cấp tùy chọn ô cẩm `IP_RECVDSTADDR`. Kỹ thuật này thường không được sử dụng với máy chủ TCP vì máy chủ TCP luôn có thể xác định địa chỉ IP đích bằng cách gọi `getsockname` sau

kết nối được thiết lập. Tuy nhiên, máy chủ TCP muốn phục vụ các kết nối tới một số, chứ không phải tất cả, các địa chỉ thuộc về máy chủ đa địa chỉ nên sử dụng kỹ thuật này.

5. **SO_REUSEADDR** cho phép các liên kết trùng lặp hoàn toàn: liên kết của một địa chỉ IP và cổng, khi cùng một địa chỉ IP và cổng đó đã được liên kết với một ô cấm khác, nếu giao thức truyền tải hỗ trợ nó. Thông thường tính năng này chỉ được hỗ trợ cho các socket UDP.

Tính năng này được sử dụng với tính năng phát đa hướng để cho phép cùng một ứng dụng được chạy nhiều lần trên cùng một máy chủ. Khi một gói dữ liệu UDP được nhận cho một trong các ô cấm được liên kết nhiều lần này, quy tắc là nếu gói dữ liệu đó được dành cho địa chỉ quảng bá hoặc địa chỉ multicast, thì một bản sao của gói dữ liệu sẽ được gửi đến mỗi ô cấm phù hợp. Như ng nếu datagram được dành cho một địa chỉ unicast thì datagram đó chỉ được phân phối tới một socket. Trong trường hợp gói dữ liệu unicast, nếu có nhiều ô cấm phù hợp với gói dữ liệu đó thì việc lựa chọn ô cấm nào nhận gói dữ liệu phụ thuộc vào việc triển khai.

Trang 777-779 của TCPv2 nói thêm về tính năng này. Chúng ta sẽ nói nhiều hơn về phát sóng và phát đa hướng trong [Chương 20](#) và [21](#).

[Bài tập 7.5](#) và [7.6](#) đưa ra một số ví dụ về tùy chọn socket này.

4.4BSD đã giới thiệu tùy chọn ô cấm **SO_REUSEPORT** khi hỗ trợ phát đa hướng được thêm vào. Thay vì làm quá tải **SO_REUSEADDR** với ngữ nghĩa phát đa hướng mong muốn cho phép các liên kết trùng lặp hoàn toàn, tùy chọn ô cấm mới này đã được giới thiệu với ngữ nghĩa sau:

1. Tùy chọn này cho phép các liên kết trùng lặp hoàn toàn, nhưng chỉ khi mỗi ô cấm đó muốn liên kết cùng một địa chỉ IP và cổng, hãy chỉ định tùy chọn ô cấm này.
2. **SO_REUSEADDR** được coi là tương đương với **SO_REUSEPORT** nếu địa chỉ IP bị ràng buộc là địa chỉ multicast (tr. 731 của TCPv2).

Vấn đề với tùy chọn ô cấm này là không phải tất cả các hệ thống đều hỗ trợ nó và trên những hệ thống không hỗ trợ tùy chọn này như ng hỗ trợ phát đa hướng, **SO_REUSEADDR** được sử dụng thay vì **SO_REUSEPORT** để cho phép các liên kết trùng lặp hoàn toàn khi có ý nghĩa (ví dụ: máy chủ UDP có thể được chạy nhiều lần trên cùng một máy chủ cùng một lúc và dự kiến sẽ nhận được các gói dữ liệu quảng bá hoặc đa hướng).

Chúng tôi có thể tóm tắt cuộc thảo luận của chúng tôi về các tùy chọn ô cấm này với các khuyến nghị sau:

1. Đặt tùy chọn ô cấm **SO_REUSEADDR** trước khi gọi **liên kết** trong tất cả các máy chủ TCP.
2. Khi viết một ứng dụng multicast có thể chạy nhiều lần trên cùng một máy chủ cùng lúc, đặt tùy chọn ô cấm **SO_REUSEADDR** và liên kết địa chỉ multicast của nhóm làm địa chỉ IP cụb bộ.

Chương 22 của TCPv2 nói chi tiết hơn về hai tùy chọn socket này.

Có một vấn đề bảo mật tiềm ẩn với `SO_REUSEADDR`. Nếu tồn tại một ô cấm được liên kết với địa chỉ tự đại diện và cổng 5555, nếu chúng tôi chỉ định `SO_REUSEADDR`, thì chúng tôi có thể liên kết cùng một cổng đó với một địa chỉ IP khác, chẳng hạn như địa chỉ IP chính của máy chủ. Bất kỳ gói dữ liệu nào trong tương lai đến cổng 5555 và địa chỉ IP mà chúng tôi liên kết với ô cấm của mình đều được gửi đến ô cấm của chúng tôi chứ không phải đến ô cấm khác được liên kết với địa chỉ tự đại diện. Đây có thể là các phân đoạn TCP SYN, các khối SCTP INIT hoặc các gói dữ liệu UDP. ([Bài tập 11.9](#) thể hiện tính năng này với UDP.) Ví dụ, đối với hầu hết các dịch vụ phổ biến, HTTP, FTP và Telnet, đây không phải là vấn đề vì các máy chủ này đều liên kết với một cổng dành riêng. Do đó, bất kỳ quá trình nào xảy ra sau đó và cố gắng liên kết một phiên bản cụ thể hơn của cổng đó (tức là lấy cấp cổng) đều cần có đặc quyền siêu người dùng. Tuy nhiên, NFS có thể là một vấn đề vì cổng thông thường của nó (2049) không được bảo lưu.

Một vấn đề cơ bản với API ô cấm là việc cài đặt cặp ô cấm được thực hiện bằng hai lệnh gọi hàm (liên kết và kết nối) thay vì một. [Torek 1994] đề xuất một hàm duy nhất có thể giải quyết vấn đề này.

```
int bind_connect_listen(int sockfd, const struct sockaddr *laddr, int
laddrlen, const struct sockaddr *faddr, int faddrlen, int listen);
```

`laddr` chỉ định địa chỉ IP cục bộ và cổng cục bộ, `faddr` chỉ định địa chỉ IP ngoài và cổng ngoài, và `listen` chỉ định một máy khách (không) hoặc một máy chủ (khác 0; giống như đối số tồn đọng để nghe). Sau đó, `bind` sẽ là một hàm thư viện gọi hàm này với `faddr` là con trả null và `faddrlen` 0, và `connect` sẽ là hàm thư viện gọi hàm này với `laddr` là con trả null và `laddrlen` 0. Có một vài ứng dụng, đặc biệt là TFTP, cần chỉ định cả cặp cục bộ và cặp ngoại và họ có thể gọi trực tiếp `bind_connect_listen`. Với chức năng như vậy, nhu cầu về `SO_REUSEADDR` sẽ biến mất, ngoại trừ đối với các máy chủ UDP phát đa hống rõ ràng cần cho phép các liên kết trùng lặp hoàn toàn của cùng một địa chỉ IP và cổng.

Một lợi ích khác của chức năng mới này là máy chủ TCP có thể tự hạn chế phục vụ các yêu cầu kết nối đến từ một địa chỉ IP và cổng cụ thể, điều mà RFC 793 [Postel 1981c] chỉ định như không thể triển khai với API ô cấm hiện có.

Tùy chọn ô cấm SO_TYPE

Tùy chọn này trả về loại ô cấm. Giá trị số nguyên được trả về là giá trị như `SOCK_STREAM` hoặc `SOCK_DGRAM`. Tùy chọn này thường được sử dụng bởi một tiến trình kế thừa một socket khi nó được khởi động.

Tùy chọn ô cấm SO_USELOOPBACK

Tùy chọn này chỉ áp dụng cho các ô cắm trong miền định tuyến ([AF_ROUTE](#)). Tùy chọn này được đặt mặc định là BẬT cho các ô cắm này (tùy chọn duy nhất trong số các tùy chọn ô cắm SO_XXX được đặt mặc định là BẬT thay vì TẮT). Khi tùy chọn này được bật, ô cắm sẽ nhận được bản sao của mọi thứ được gửi trên ô cắm.

Một cách khác để vô hiệu hóa các bản sao vòng lặp này là gọi lệnh [tắt máy](#) với đối số thứ hai là [SHUT_RD](#).

7.6 Tùy chọn ô cắm IPv4

Các tùy chọn ô cắm này được xử lý bằng IPv4 và có cấp độ [IPPROTO_IP](#). Chúng tôi trì hoãn việc thảo luận về các tùy chọn ô cắm phát đa hướng cho đến [Phần 21.6](#).

Tùy chọn ô cắm IP_HDRINCL

Nếu tùy chọn này được đặt cho ô cắm IP thô ([Chú ý 28](#)), chúng ta phải xây dựng tiêu đề IP của riêng mình cho tất cả các datagram chúng ta gửi trên ô cắm thô. Thông thường, kernel xây dựng tiêu đề IP cho các datagram được gửi trên ô cắm thô, nhưng có một số ứng dụng (đặc biệt là [traceroute](#)) xây dựng tiêu đề IP của riêng chúng để ghi đè các giá trị mà IP sẽ đặt vào.

trường tiêu đề nhất định.

Khi tùy chọn này được đặt, chúng tôi sẽ xây dựng một tiêu đề IP hoàn chỉnh, với các ngoại lệ sau:

- IP luôn tính toán và lưu trữ tổng kiểm tra tiêu đề IP.
- Nếu chúng ta đặt trường nhận dạng IP thành 0, kernel sẽ đặt trường này.
- Nếu địa chỉ IP nguồn là [INADDR_ANY](#), IP sẽ đặt nó thành địa chỉ IP chính của giao diện gửi đi.

- Việc thiết lập các tùy chọn IP phụ thuộc vào việc thực hiện. Một số triển khai sử dụng bất kỳ tùy chọn IP nào được đặt bằng tùy chọn ô cắm [IP_OPTIONS](#) và nối các tùy chọn này vào tiêu đề mà chúng tôi xây dựng, trong khi các tùy chọn triển khai khác yêu cầu tiêu đề của chúng tôi cũng chứa bất kỳ tùy chọn IP mong muốn nào.
- Một số trường phải theo thứ tự byte máy chủ và một số trường phải theo thứ tự byte mạng. Điều này phụ thuộc vào việc triển khai, khiến cho việc ghi các gói thô bằng [IP_HDRINCL](#) không thể di chuyển được như chúng tôi mong muốn.

Chúng tôi đưa ra một ví dụ về tùy chọn này trong [Phần 29.7](#). Trang 1056-1057 của TCPv2 cung cấp thêm chi tiết về tùy chọn ô cắm này.

Tùy chọn ô cắm IP_OPTIONS

Đặt tùy chọn này cho phép chúng tôi đặt tùy chọn IP trong tiêu đề IPv4. Điều này đòi hỏi kiến thức sâu sắc về định dạng của các tùy chọn IP trong tiêu đề IP. Chúng ta sẽ thảo luận về lựa chọn này liên quan đến các tuyến nguồn IPv4 trong [Phần 27.3](#).

Tùy chọn ô cắm IP_RECVSTADDR

Tùy chọn ô cắm này khiền địa chỉ IP đích của gói dữ liệu UDP đã nhận được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Chúng tôi sẽ trình bày một ví dụ về tùy chọn này trong [Phần 22.2](#).

Tùy chọn ô cắm IP_RECVIF

Tùy chọn ô cắm này làm cho chỉ mục của giao diện mà gói dữ liệu UDP được nhận được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Chúng tôi sẽ trình bày một ví dụ về tùy chọn này trong [Phần 22.2](#).

Tùy chọn ô cắm IP_TOS

Tùy chọn này cho phép chúng ta đặt trường loại dịch vụ (TOS) (chứa các trường DSCP và ECN, [Hình A.1](#)) trong tiêu đề IP cho ô cắm TCP, UDP hoặc SCTP. Nếu chúng ta gọi `getsockopt` cho tùy chọn này, giá trị hiện tại sẽ được đặt vào các trường DSCP và ECN trong tiêu đề IP (mặc định là 0) sẽ được trả về. Không có cách nào để lấy giá trị từ gói dữ liệu IP đã nhận.

Ứng dụng có thể đặt DSCP thành giá trị được thương lượng với nhà cung cấp dịch vụ mạng để nhận các dịch vụ được sắp xếp trước, ví dụ: độ trễ thấp đối với điện thoại IP hoặc thông thương lượng cao hơn để truyền dữ liệu số thương lượng lớn. Kiến trúc diffserv, được định nghĩa trong RFC 2474 [Nichols et al. 1998], chỉ cung cấp khả năng tương thích ngay lập tức với định nghĩa trường TOS lịch sử (từ RFC 1349 [Almquist 1992]). Ứng dụng đặt `IP_TOS` thành một trong các nội dung từ `<netinet/ip.h>`, chẳng hạn như `IPTOS_LOWDELAY` hoặc `IPTOS_THROUGHPUT`, thay vào đó nên sử dụng giá trị DSCP do người dùng chỉ định. Các giá trị TOS duy nhất mà Diffserv giữ lại là mức độ ưu tiên 6 ("điều khiển mạng") và 7 ("điều khiển mạng"); điều này có nghĩa là các ứng dụng đặt `IP_TOS` thành `IPTOS_PREC_NETControl` hoặc `IPTOS_PREC_INTERNETControl` sẽ hoạt động ở chế độ khác mạng.

RFC 3168 [Ramakrishnan, Floyd và Black 2001] chứa định nghĩa về trường ECN. Các ứng dụng thương lượng nên để cài đặt trường ECN cho kernel và phải chỉ định các giá trị 0 ở hai bit thấp của giá trị được đặt bằng `IP_TOS`.

Tùy chọn ô cắm IP_TTL

Với tùy chọn này, chúng ta có thể thiết lập và tìm nạp TTL mặc định ([Hình A.1](#)) mà hệ thống sẽ sử dụng cho các gói unicast được gửi trên một socket nhất định. (TTL multicast được đặt bằng cách sử dụng tùy chọn ô cắm `IP_MULTICAST_TTL`, được mô tả trong [Phần 21.6](#).) Ví dụ: 4.4BSD sử dụng giá trị mặc định là 64 cho cả hai ô cắm TCP và UDP (được chỉ định trong số đăng ký "Số tùy chọn IP" của IANA [IANA]) và 255 cho ô cắm thôn. Giống như trường TOS, việc gọi `getsockopt` trả về giá trị mặc định của trường mà hệ thống sẽ sử dụng trong các gói dữ liệu gửi đi-không có cách nào để lấy giá trị từ một gói dữ liệu đã nhận.

Chúng ta sẽ thiết lập tùy chọn socket này bằng chương trình `traceroute` trong [Hình 28.19](#).

Tùy chọn ô cắm 7.7 ICMPv6

Tùy chọn này cho phép chúng tôi tìm nạp và đặt cấu trúc `icmp6_filter` chỉ định loại thông báo nào trong số 256 loại thông báo ICMPv6 có thể sẽ được chuyển đến quy trình trên một ô cắm thô. Chúng ta sẽ thảo luận về lựa chọn này [trong Phần 28.4](#).

7.8 Tùy chọn ô cắm IPv6

Các tùy chọn ô cắm này được xử lý bằng IPv6 và có cấp độ IPPROTO_IPV6. Chúng tôi trì hoãn việc thảo luận về các tùy chọn ô cắm phát đa hứa ứng cho đến [Phần 21.6](#). Chúng tôi lưu ý rằng nhiều tùy chọn trong số này sử dụng dữ liệu phụ trợ với hàm `recvmsg` và chúng tôi sẽ mô tả điều này trong [Phần 14.6](#). Tất cả các tùy chọn ô cắm IPv6 được xác định trong RFC 3493 [Gilligan et al. 2003] và RFC 3542 [Stevens và cộng sự. 2003].

Tùy chọn ô cắm IPV6_CHECKSUM

Tùy chọn ô cắm này chỉ định độ lệch byte vào dữ liệu gửi đi dùng nơi đặt truthor tổng kiểm tra. Nếu giá trị này không âm, kernel sẽ: (i) tính toán và lưu trữ tổng kiểm tra cho tất cả các gói gửi đi và (ii) xác minh tổng kiểm tra nhận được ở đầu vào, loại bỏ các gói có tổng kiểm tra không hợp lệ. Tùy chọn này ảnh hưởng đến tất cả các ô cắm thô IPv6, ngoại trừ ô cắm thô ICMPv6. (Hạt nhân luôn tính toán và lưu trữ tổng kiểm tra cho các ô cắm thô ICMPv6.) Nếu giá trị -1 được chỉ định (mặc định), hạt nhân sẽ không tính toán và lưu trữ tổng kiểm tra cho các gói gửi đi trên ô cắm thô này và sẽ không xác minh tổng kiểm tra cho các gói nhận được.

Tất cả các giao thức sử dụng IPv6 phải có tổng kiểm tra trong tiêu đề giao thức riêng của chúng. Các tổng kiểm tra này bao gồm một tiêu đề giả (RFC 2460 [Deering và Hinden 1998]) bao gồm địa chỉ IPv6 nguồn như một phần của tổng kiểm tra (khác với tất cả các giao thức khác thường được triển khai bằng ô cắm thô có IPv4).

Thay vì buộc ứng dụng sử dụng socket thô thực hiện lựa chọn địa chỉ nguồn, kernel sẽ thực hiện việc này, sau đó tính toán và lưu trữ tổng kiểm tra kết hợp với tiêu đề giả IPv6 tiêu chuẩn.

Tùy chọn ô cắm IPV6_DONTFRAG

Việc đặt tùy chọn này sẽ tắt tính năng tự động chèn tiêu đề phân đoạn cho UDP và ô cắm thô. Khi tùy chọn này được đặt, các gói đầu ra lớn hơn MTU của giao diện gửi đi sẽ bị loại bỏ. Không cần trả lại lỗi từ lệnh gọi hệ thống gửi gói vì gói có thể vượt quá đường dẫn MTU trên đường đi. Thay vào đó, ứng dụng nên kích hoạt tùy chọn `IPV6_RECVPATHMTU` ([Phần 22.9](#)) để tìm hiểu về các thay đổi MTU đường dẫn.

Tùy chọn ô cắm IPV6_NEXTHOP

Tùy chọn này chỉ định địa chỉ bù ớc nhảy tiếp theo cho một datagram dưới dạng cấu trúc địa chỉ socket và là một hoạt động đặc quyền. Chúng tôi sẽ nói nhiều hơn về tính năng này ở [Phần 22.8](#).

Tùy chọn ô cắm IPV6_PATHMTU

Tùy chọn này không thể được đặt, chỉ được truy xuất. Khi tùy chọn này được truy xuất, MTU hiện tại được xác định bằng khám phá đường dẫn-MTU sẽ được trả về (xem [Phần 22.9](#)).

Tùy chọn ô cắm IPV6_RECVDSTOPTS

Việc đặt tùy chọn này chỉ định rằng mọi tùy chọn đích IPv6 đã nhận sẽ được `recvmsg` trả về dưới dạng dữ liệu **phụ trợ**. Tùy chọn này mặc định là TẮT. Chúng tôi sẽ mô tả các chức năng được sử dụng để xây dựng và xử lý các tùy chọn này trong [Phần 27.5](#).

Tùy chọn ô cắm IPV6_RECVHOPLIMIT

Việc đặt tùy chọn này chỉ định rằng trung giới hạn số bù ớc nhảy đã nhận sẽ được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Tùy chọn này mặc định là TẮT. Chúng tôi sẽ mô tả tùy chọn này trong [Mục 22.8](#).

Không có cách nào với IPv4 để có được trung giới TTL đã nhận.

Tùy chọn ô cắm IPV6_RECHOPOPTS

Việc đặt tùy chọn này chỉ định rằng mọi tùy chọn từng chặng IPv6 đã nhận sẽ được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Tùy chọn này mặc định là TẮT. Chúng tôi sẽ mô tả các chức năng được sử dụng để xây dựng và xử lý các tùy chọn này trong [Phần 27.5](#).

Tùy chọn ô cắm IPV6_RECVPATHMTU

Việc đặt tùy chọn này chỉ định rằng MTU đường dẫn của một đường dẫn sẽ được `recvmsg` trả về dưới dạng dữ liệu phụ trợ (không có bất kỳ dữ liệu đi kèm nào) khi nó thay đổi. Chúng tôi sẽ mô tả tùy chọn này trong [Phần 22.9](#).

Tùy chọn ô cắm IPV6_RECVPKTINFO

Việc đặt tùy chọn này chỉ định rằng hai phần thông tin sau đây về gói dữ liệu IPv6 đã nhận sẽ được `recvmsg` trả về dưới dạng dữ liệu phụ trợ: địa chỉ IPv6 đích và chỉ mục giao diện đến. Chúng tôi sẽ mô tả tùy chọn này trong [Phần 22.8](#).

Tùy chọn ô cắm IPV6_RECVRTHDR

Việc đặt tùy chọn này chỉ định rằng tiêu đề định tuyến IPv6 đã nhận sẽ được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Tùy chọn này mặc định là **TẮT**. Chúng tôi sẽ mô tả các chức năng được sử dụng để xây dựng và xử lý tiêu đề định tuyến IPv6 trong [Phần 27.6](#).

Tùy chọn ô cắm IPV6_RECVTCLASS

Việc đặt tùy chọn này chỉ định rằng lớp lưu lượng đã nhận (chứa các trường DSCP và ECN) sẽ được trả về dưới dạng dữ liệu phụ trợ bởi `recvmsg`. Tùy chọn này mặc định là **TẮT**.

Chúng tôi sẽ mô tả tùy chọn này trong [Phần 22.8](#).

Tùy chọn ô cắm IPV6_UNICAST_HOPS

Tùy chọn IPv6 này tương tự như tùy chọn ô cắm IPv4 `IP_TTL`. Việc đặt tùy chọn ô cắm chỉ định giới hạn số bù ớc nhảy mặc định cho các gói dữ liệu gửi đi trên ô cắm, trong khi tìm nạp tùy chọn ô cắm sẽ trả về giá trị cho giới hạn số bù ớc nhảy mà kernel sẽ sử dụng cho ô cắm. Trường giới hạn bù ớc nhảy thực tế từ gói dữ liệu IPv6 đã nhận được lấy bằng cách sử dụng tùy chọn ô cắm `IPV6_RECVHOPLIMIT`.

Chúng ta sẽ thiết lập tùy chọn socket này bằng chương trình `traceroute` trong [Hình 28.19](#).

Tùy chọn ô cắm IPV6_USE_MIN_MTU

Đặt tùy chọn này thành 1 chỉ định rằng việc phát hiện MTU đường dẫn sẽ không được thực hiện và các gói được gửi bằng MTU IPv6 tối thiểu để tránh phân mảnh. Đặt nó thành 0 sẽ khiến việc phát hiện MTU đường dẫn xảy ra đối với tất cả các đích. Đặt nó thành-1 chỉ định rằng việc phát hiện MTU đường dẫn được thực hiện cho các đích unicast nhưng MTU tối thiểu được sử dụng khi gửi đến các đích multicast. Tùy chọn này mặc định là -1. Chúng tôi sẽ mô tả tùy chọn này trong [Phần 22.9](#).

Tùy chọn ô cắm IPV6_V6ONLY

Việc đặt tùy chọn này trên ô cắm `AF_INET6` sẽ chỉ giới hạn ở khả năng giao tiếp IPv6.

Tùy chọn này mặc định là **TẮT**, mặc dù một số hệ thống có tùy chọn **BẬT** tùy chọn này theo mặc định. Chúng tôi sẽ mô tả giao tiếp IPv4 và IPv6 bằng cách sử dụng ô cắm `AF_INET6` trong [Phần 12.2](#) và [12.3](#).

Tùy chọn ô cắm IPV6_XXX

Hầu hết các tùy chọn IPv6 để sửa đổi tiêu đề đều giả định một ô cắm UDP với thông tin được truyền giữa kernel và ứng dụng bằng cách sử dụng dữ liệu phụ trợ với `recvmsg` và `sendmsg`. Thay vào đó, ô cắm TCP tìm nạp và lưu trữ các giá trị này bằng cách sử dụng `getsockopt` và `setsockopt`. Tùy chọn ô cắm giống với loại dữ liệu phụ trợ và bộ đệm chứa cùng thông tin như có trong dữ liệu phụ trợ. Chúng tôi sẽ mô tả điều này trong [Phần 27.7](#).

7.9 Tùy chọn ô cắm TCP

Có hai tùy chọn ô cắm cho TCP. Chúng tôi chỉ định cấp độ là `IPPROTO_TCP`.

Tùy chọn ô cắm TCP_MAXSEG

Tùy chọn ô cắm này cho phép chúng tôi tìm nạp hoặc đặt MSS cho kết nối TCP. Giá trị được trả về là lượng dữ liệu tối đa mà TCP của chúng ta sẽ gửi đến đầu bên kia; thông thường, đó là MSS được đầu bên kia công bố với SYN của nó, trừ khi TCP của chúng tôi chọn sử dụng giá trị nhỏ hơn MSS được công bố của đối phuơng. Nếu giá trị này được tìm nạp trước khi ô cắm được kết nối, giá trị được trả về là giá trị mặc định sẽ được sử dụng nếu không nhận được tùy chọn MSS từ đầu bên kia. Ngoài ra, hãy lưu ý rằng một giá trị nhỏ hơn giá trị được trả về thực sự có thể được sử dụng cho kết nối nếu tùy chọn dấu thời gian chặng hạn đang được sử dụng, vì tùy chọn này chiếm 12 byte tùy chọn TCP trong mỗi phân đoạn.

Lượng dữ liệu tối đa mà TCP của chúng tôi sẽ gửi cho mỗi phân đoạn cũng có thể thay đổi trong suốt thời gian kết nối nếu TCP hỗ trợ khám phá MTU đường dẫn. Nếu đường đến thiết bị ngang hàng thay đổi, giá trị này có thể tăng hoặc giảm.

Chúng tôi lưu ý [trong Hình 7.1](#) rằng tùy chọn ô cắm này cũng có thể được thiết lập bởi ứng dụng. Điều này không thể thực hiện được trên tất cả các hệ thống; ban đầu nó là một tùy chọn chỉ đọc. 4.4BSD giới hạn ứng dụng giảm giá trị: Chúng tôi không thể tăng giá trị (tr. 1023 của TCPv2).

Vì tùy chọn này kiểm soát lượng dữ liệu mà TCP gửi trên mỗi phân đoạn nên việc cấm ứng dụng tăng giá trị là điều hợp lý. Sau khi kết nối được thiết lập, giá trị này là tùy chọn MSS được thiết bị ngang hàng thông báo và chúng tôi không thể vượt quá giá trị đó. Tuy nhiên, TCP của chúng tôi luôn có thể gửi ít hơn TCP của bạn

MSS đã công bố.

Tùy chọn ô cắm TCP_NODELAY

Nếu được đặt, tùy chọn này sẽ tắt thuật toán Nagle của TCP (Phần 19.4 của TCPv1 và trang 858-859 của TCPv2).

Theo mặc định, thuật toán này được kích hoạt.

Mục đích của thuật toán Nagle là giảm số lượng gói nhỏ trên mạng WAN. Thuật toán nêu rõ rằng nếu một kết nối nhất định có dữ liệu nối bật (nghĩa là dữ liệu mà TCP của chúng tôi đã gửi và hiện đang chờ xác nhận), thì sẽ không có gói nhỏ nào được gửi trên kết nối để phản hồi thao tác ghi của người dùng cho đến khi dữ liệu hiện có được thừa nhận. Định nghĩa gói "nhỏ" là bất kỳ gói nào nhỏ hơn MSS. TCP sẽ luôn gửi gói có kích thước đầy đủ nếu có thể; Mục đích của thuật toán Nagle là ngăn chặn kết nối có nhiều gói nhỏ chưa được xử lý bắt cứ lúc nào.

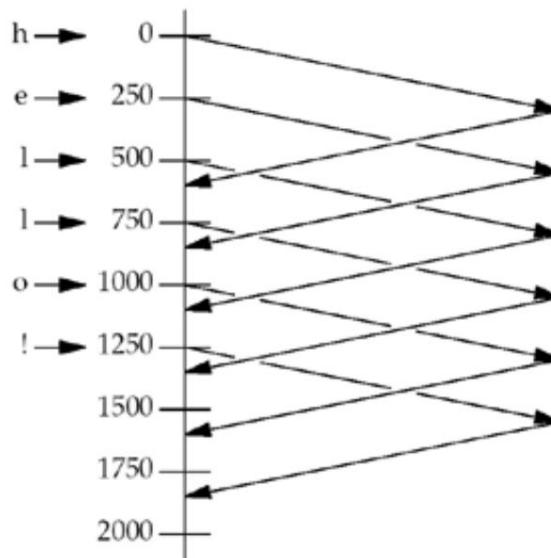
Hai trình tạo phô biến của các gói nhỏ là máy khách Rlogin và Telnet, vì chúng thường gửi mỗi lần nhấn phím dữ ới dạng một gói riêng biệt. Trên mạng LAN nhanh, chúng tôi thường không chú ý đến thuật toán Nagle với các máy khách này vì thời gian cần thiết để xác nhận một gói nhỏ thường là vài mili giây-ít hơn nhiều so với thời gian giữa hai ký tự liên tiếp mà chúng tôi nhập. Như ng trên mạng WAN, nơi có thể mất một giây để xác nhận một gói nhỏ, chúng ta có thể nhận thấy độ trễ trong tiếng vang của ký tự và độ trễ này thường bị thuật toán Nagle phóng đại.

Hãy xem xét ví dụ sau: Chúng ta gửi chuỗi sáu ký tự "hello!" tới máy khách Rlogin hoặc Telnet, với thời gian chính xác là 250 mili giây giữa mỗi ký tự. RTT tới máy chủ là 600 ms và máy chủ ngay lập tức gửi lại tiếng vang của từng ký tự.

Chúng tôi giả sử ACK của ký tự của máy khách được gửi lại cho máy khách cùng với tiếng vang của ký tự và chúng tôi bỏ qua các ACK mà máy khách gửi cho tiếng vang của máy chủ.

(Chúng ta sẽ nói về ACK bị trì hoãn ngay sau đây.) Giả sử thuật toán Nagle bị vô hiệu hóa, chúng ta có 12 gói được hiển thị trong [Hình 7.14](#).

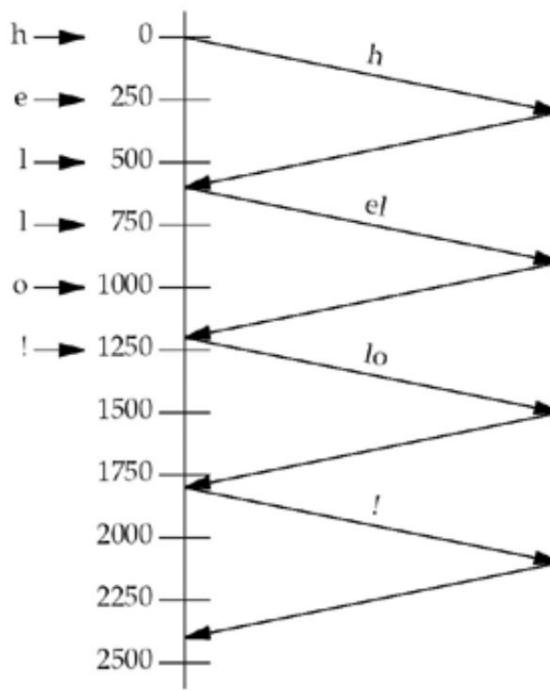
Hình 7.14. Sáu ký tự được lặp lại bởi máy chủ khi thuật toán Nagle bị tắt.



Mỗi ký tự được gửi trong một gói riêng: các phân đoạn dữ liệu từ trái sang phải và ACK từ phải sang trái.

Nếu thuật toán Nagle được bật (mặc định), chúng ta có tám gói được hiển thị trong [Hình 7.15](#). Ký tự đầu tiên được gửi dữ ới dạng gói riêng, như ng hai ký tự tiếp theo không được gửi vì kết nối có một gói nhỏ chưa được xử lý. Tại thời điểm 600, khi nhận được ACK của gói đầu tiên, cùng với tiếng vang của ký tự đầu tiên, hai ký tự này sẽ được gửi đi. Cho đến khi gói này được ACK vào thời điểm 1200 thì không có gói nhỏ nào được gửi nữa.

Hình 7.15. Sáu ký tự được lặp lại bởi máy chủ có bật thuật toán Nagle.



Thuật toán Nagle thư ờng tư ơng tác với một thuật toán TCP khác: ACK bị trì hoãn. Thuật toán này khiến TCP không gửi ACK ngay khi nhận dữ liệu; thay vào đó, TCP sẽ đợi một khoảng thời gian nhỏ (thu ờng là 50-200 ms) và chỉ sau đó mới gửi ACK. Hy vọng là trong khoảng thời gian ngắn ngủi này, sẽ có dữ liệu để gửi lại cho thiết bị ngang hàng và ACK có thể mang theo dữ liệu đó, lưu một phân đoạn TCP. Điều này thư ờng xảy ra với các máy khách Rlogin và Telnet, vì các máy chủ thư ờng lặp lại từng ký tự đư ợc máy khách gửi, do đó ACK của ký tự của máy khách sẽ gắn liền với tiếng vọng của máy chủ về ký tự đó.

Vấn đề xảy ra với các máy khách khác có máy chủ không tạo ra lưu lư ợng truy cập theo hư ớng ngư ợc lại mà ACK có thể công. Những máy khách này có thể phát hiện độ trễ đáng chú ý vì máy khách TCP sẽ không gửi bất kỳ dữ liệu nào đến máy chủ cho đến khi bộ đếm thời gian ACK bị trì hoãn của máy chủ hết hạn. Những máy khách này cần một cách để vô hiệu hóa thuật toán Nagle, do đó có tùy chọn [TCP_NODELAY](#).

Một loại máy khách khác tư ơng tác không tốt với thuật toán Nagle và ACK bị trễ của TCP là máy khách gửi một yêu cầu logic duy nhất đến máy chủ của nó theo từng phần nhỏ. Ví dụ: giả sử một máy khách gửi yêu cầu 400 byte đến máy chủ của nó, nhưng đây là loại yêu cầu 4 byte, theo sau là 396 byte dữ liệu yêu cầu. Nếu máy khách thực hiện [ghi](#) 4 byte, sau đó là ghi 396 byte, lần ghi thứ hai sẽ không đư ợc gửi bởi TCP máy khách cho đến khi TCP máy chủ xác nhận ghi 4 byte. Ngoài ra, do ứng dụng máy chủ không thể hoạt động trên 4 byte dữ liệu cho đến khi nhận đư ợc 396 byte dữ liệu còn lại, TCP máy chủ sẽ trì hoãn ACK của 4 byte dữ liệu (tức là sẽ không có bất kỳ dữ liệu nào từ máy chủ tới máy khách sẽ mang theo ACK). Có ba cách để khắc phục loại khách hàng này:

1. Sử dụng `writenv` ([Phần 14.4](#)) thay vì hai lệnh gọi `write`. Một cuộc gọi tới `writenv` kết thúc bằng một lệnh gọi đến đầu ra TCP thay vì hai lệnh gọi, dẫn đến một phân đoạn TCP cho ví dụ của chúng tôi. Đây là giải pháp ưa thích.
2. Sao chép 4 byte dữ liệu và 396 byte dữ liệu vào một bộ đệm duy nhất và gọi `viết` một lần cho bộ đệm này.
3. Đặt tùy chọn ô cắm `TCP_NODELAY` và tiếp tục gọi `ghi` hai lần. Đây là giải pháp ít được mong muốn nhất và có hại cho mạng, do đó, nó thường không được xem xét.

[Bài tập 7.8 và 7.9 tiếp tục ví dụ này.](#)

Tùy chọn ô cắm SCTP 7.10

Số lượng tùy chọn ô cắm tương đối lớn cho SCTP (17 tùy chọn hiện tại) phản ánh mức độ kiểm soát tốt hơn mà SCTP cung cấp cho nhà phát triển ứng dụng. Chúng tôi chỉ định cấp độ là `IPPROTO_SCTP`.

Một số tùy chọn được sử dụng để lấy thông tin về SCTP yêu cầu dữ liệu đó phải được chuyển vào kernel (ví dụ: ID liên kết và/hoặc địa chỉ ngang hàng). Mặc dù một số triển khai của `getsockopt` hỗ trợ truyền dữ liệu vào và ra khỏi kernel, như không phải tất cả đều làm được. API SCTP xác định hàm `sctp_opt_info` ([Phần 9.11](#)) để che giấu sự khác biệt này. Trên các hệ thống mà `getsockopt` hỗ trợ điều này, nó chỉ đơn giản là một trình bao bọc xung quanh `getsockopt`. Ngoài ra, nó sẽ thực hiện hành động được yêu cầu, có thể bằng cách sử dụng `ioctl` tùy chỉnh hoặc lệnh gọi hệ thống mới. Chúng tôi khuyên bạn nên luôn sử dụng `sctp_opt_info` khi truy xuất các tùy chọn này để có tính di động tối đa. Các tùy chọn này được đánh dấu bằng dấu gạch () trong [Hình 7.2](#) và bao gồm `SCTP_ASSOCINFO`, `SCTP_GET_PEER_ADDR_INFO`, `SCTP_PEER_ADDR_PARAMS`, `SCTP_PRIMARY_ADDR`, `SCTP_RTOINFO` và `SCTP_STATUS`.



Tùy chọn ô cắm SCTP_ADAPTATION_LAYER

Trong quá trình khởi tạo liên kết, một trong hai điểm cuối có thể chỉ định chỉ báo lớp thích ứng. Chỉ báo này là số nguyên không dấu 32 bit có thể được hai ứng dụng sử dụng để phối hợp bất kỳ lớp thích ứng ứng dụng cụ bộ nào. Tùy chọn này cho phép người gọi tìm nạp hoặc đặt chỉ báo lớp thích ứng mà điểm cuối này sẽ cung cấp cho

đồng nghiệp.

Khi tìm nạp giá trị này, người gọi sẽ chỉ truy xuất giá trị mà ô cắm cục bộ sẽ cung cấp cho tất cả các đồng nghiệp trong tương lai. Để truy xuất chỉ báo lớp thích ứng của thiết bị ngang hàng, ứng dụng phải đăng ký các sự kiện của lớp thích ứng.

Tùy chọn ô cắm SCTP_ASSOCINFO

Tùy chọn ô cắm `SCTP_ASSOCINFO` có thể được sử dụng cho ba mục đích: (i) để lấy thông tin về một liên kết hiện có, (ii) để thay đổi các tham số của một liên kết hiện có.

liên kết và/hoặc (iii) để đặt giá trị mặc định cho các liên kết trong tương lai. Khi truy xuất thông tin về một liên kết hiện có, nên sử dụng hàm `sctp_opt_info` thay vì `getsockopt`. Tùy chọn này lấy đầu vào là `sctp_assocparams` kết cấu.

```
cấu trúc sctp_assocparams {
    sctp_assoc_t sasoc_assoc_id;
    u_int16_t sasoc_asocmaxrxt;
    u_int16_t sasoc_number_peer_destination;
    u_int32_t sasoc_peer_rwnd;
    u_int32_t sasoc_local_rwnd;
    u_int32_t sasoc_cookie_life;
};
```

Các trường này có ý nghĩa như sau:

- `sasoc_assoc_id` giữ thông tin nhận dạng cho mỗi quan tâm. Nếu điều này giá trị được đặt thành 0 khi gọi hàm `setsockopt`, sau đó `sasoc_asocmaxrxt` và `sasoc_cookie_life` đại diện cho các giá trị được đặt làm mặc định trên ô cắm. Việc gọi `getsockopt` sẽ trả về thông tin cụ thể của liên kết nếu ID liên kết được cung cấp; mặt khác, nếu trường này bằng 0 thì cài đặt điểm cuối mặc định sẽ được trả về.
- `sasoc_asocmaxrxt` chứa số lần truyền lại tối đa liên kết sẽ thực hiện mà không có sự thừa nhận trước khi từ bỏ, báo cáo thiết bị ngang hàng không sử dụng được và đóng liên kết.
- `sasoc_number_peer_destinations` chứa số lượng đích ngang hàng địa chỉ. Nó không thể được thiết lập, chỉ lấy được.
- `sasoc_peer_rwnd` giữ cửa sổ nhận được tính toán hiện tại của máy ngang hàng. Cái này value đại diện cho tổng số byte dữ liệu có thể được gửi. Trường này rất năng động; khi điểm cuối cục bộ gửi dữ liệu, giá trị này sẽ giảm. Khi ứng dụng từ xa đọc dữ liệu đã nhận được, giá trị này sẽ tăng lên. Giá trị này không thể thay đổi bằng lệnh gọi tùy chọn ô cắm này.
- `sasoc_local_rwnd` đại diện cho cửa sổ nhận cục bộ mà ngăn xếp SCTP là hiện đang báo cáo cho đồng nghiệp. Giá trị này cũng động và bị ảnh hưởng bởi tùy chọn ô cắm `SO_SNDBUF`. Giá trị này không thể thay đổi bằng lệnh gọi tùy chọn ô cắm này.
- `sasoc_cookie_life` đại diện cho số mili giây mà một cookie được cấp cho một thiết bị ngang hàng ở xa là hợp lệ. Mỗi cookie trạng thái được gửi đến một thiết bị ngang hàng có thời gian tồn tại được liên kết với nó để ngăn chặn các cuộc tấn công lặp lại. Giá trị mặc định 60.000 mili giây có thể được thay đổi bằng cách đặt tùy chọn này bằng `sasoc_assoc_id` giá trị 0.

Chúng tôi sẽ đưa ra lời khuyên về việc điều chỉnh giá trị `sasoc_asocmaxrxt` để đạt hiệu suất trong [Phần 23.11](#). Có thể giảm `sasoc_cookie_life` để bảo vệ tốt hơn trước các cuộc tấn công phát lại cookie như ng kém mạnh mẽ hơn đối với độ trễ mạng trong quá trình bắt đầu liên kết. Các giá trị khác hữu ích cho việc gỡ lỗi.

Tùy chọn ô cắm SCTP_AUTOCLLOS

Tùy chọn này cho phép chúng tôi tìm nạp hoặc đặt thời gian tự động đóng cho điểm cuối SCTP. Thời gian tự động đóng là số giây mà liên kết SCTP sẽ vẫn mở khi không hoạt động. Nhìn rõi được xác định bởi ngăn xếp SCTP vì không phải người dùng gửi hoặc nhận điểm cuối dữ liệu. Mặc định là chức năng tự động đóng sẽ bị tắt.

Tùy chọn tự động đóng được thiết kế để sử dụng trong giao diện SCTP kiểu một-nhiều ([Chương 9](#)). Khi tùy chọn này được đặt, số nguyên được truyền cho tùy chọn là số giây trước khi đóng kết nối không hoạt động; giá trị 0 sẽ vô hiệu hóa tính năng tự động đóng. Chỉ các liên kết trong tương lai được tạo bởi điểm cuối này mới bị ảnh hưởng bởi tùy chọn này; các hiệp hội hiện tại vẫn giữ nguyên thiết lập hiện tại của họ.

Máy chủ có thể sử dụng tính năng tự động đóng để buộc đóng các liên kết nhàn rỗi mà không cần máy chủ duy trì trạng thái bổ sung. Máy chủ sử dụng tính năng này cần đánh giá cẩn thận thời gian nhàn rỗi dài nhất dự kiến trên tất cả các liên kết của nó. Việc đặt giá trị tự động đóng nhỏ hơn mức cần thiết sẽ dẫn đến việc đóng liên kết sớm.

Tùy chọn ô cắm SCTP_DEFAULT_SEND_PARAM

SCTP có nhiều tham số gửi tùy chọn thư ờng được truyền dưới dạng dữ liệu phụ trợ hoặc được sử dụng với lệnh gọi hàm `sctp_sendmsg` (thư ờng được triển khai dưới dạng lệnh gọi thư viện để truyền dữ liệu phụ trợ cho người dùng). Một ứng dụng muốn gửi một số lượng lớn tin nhắn, tất cả đều có cùng tham số, có thể sử dụng tùy chọn này để thiết lập các tham số mặc định và do đó tránh sử dụng dữ liệu phụ trợ hoặc lệnh gọi `sctp_sendmsg`. Tùy chọn này lấy cấu trúc `sctp_sndrcvinfo` làm đầu vào.

```
cấu trúc sctp_sndrcvinfo {
    u_int16_t sininfo_stream;
    u_int16_t sininfo_ssn;
    u_int16_t sininfo_flags;
    u_int32_t sininfo_ppid;
    u_int32_t sininfo_context;
    u_int32_t sininfo_timetolive;
    u_int32_t sininfo_tsn;
    u_int32_t sininfo_cumtsn;
    sctp_assoc_t sininfo_assoc_id;
```

```
};
```

Các truờng này được định nghĩa như sau:

- **sinfo_stream** chỉ định luồng mặc định mới mà tất cả các tin nhắn sẽ được đưa vào đã gửi.
- **sinfo_ssn** bị bỏ qua khi thiết lập các tùy chọn mặc định. Khi nhận được một thông báo có hàm `recvmsg` hoặc hàm `sctp_recvmsg`, truờng này sẽ giữ giá trị mà thiết bị ngang hàng được đặt trong truờng số thứ tự luồng (SSN) trong đoạn dữ liệu SCTP.
- **sinfo_flags** quy định các cờ mặc định để áp dụng cho tất cả các tin nhắn được gửi trong tương lai. Các giá trị cờ cho phép có thể được tìm thấy trong [Hình 7.16](#).

Hình 7.16. Giá trị cờ SCTP được phép cho truờng `sinfo_flags`.

Constant	Description
<code>MSG_ABORT</code>	Invoke ABORTIVE termination of the association
<code>MSG_ADDR_OVER</code>	Specify that SCTP should override the primary address and use the provided address instead
<code>MSG_EOF</code>	Invoke graceful termination after the sending of this message
<code>MSG_PR_BUFFER</code>	Enable the buffer-based profile of the partial reliability feature (if available)
<code>MSG_PR_SCTP</code>	Enable the partial reliability (if available) feature on this message
<code>MSG_UNORDERED</code>	Specify that this message uses the unordered message service

- **sinfo_pid** cung cấp giá trị mặc định để sử dụng khi thiết lập tài trọng SCTP định danh giao thức trong tất cả các lần truyền dữ liệu.
- **sinfo_context** chỉ định giá trị mặc định để đặt trong truờng `sinfo_context`, được cung cấp dưới dạng thê cục bộ khi các tin nhắn không thể gửi đến thiết bị ngang hàng được lấy ra.
- **sinfo_timetolive** quy định thời gian tồn tại mặc định sẽ được áp dụng cho tất cả tin nhắn gửi đi. Truờng thời gian tồn tại được ngăn xếp SCTP sử dụng để biết khi nào nên loại bỏ tin nhắn gửi đi do độ trễ quá mức (trừ 1 lần truyền đầu tiên). Nếu hai điểm cuối hỗ trợ tùy chọn độ tin cậy một phần thì thời gian tồn tại cũng được sử dụng để chỉ định khoảng thời gian hợp lệ của tin nhắn sau lần truyền đầu tiên.
- **sinfo_tsn** bị bỏ qua khi thiết lập các tùy chọn mặc định. Khi nhận được một thông báo có hàm `recvmsg` hoặc hàm `sctp_recvmsg`, truờng này sẽ giữ giá trị mà thiết bị ngang hàng đặt trong truờng số thứ tự truyền tải (TSN) trong đoạn SCTP DATA.
- **sinfo_cumtsn** bị bỏ qua khi thiết lập các tùy chọn mặc định. Khi nhận được một tin nhắn có hàm `recvmsg` hoặc hàm `sctp_recvmsg`, truờng này sẽ chứa TSN tích lũy hiện tại mà ngăn xếp SCTP cục bộ đã liên kết với nó ngang hàng từ xa.
- **sinfo_assoc_id** chỉ định nhận dạng liên kết mà người yêu cầu muốn các tham số mặc định được thiết lập theo. Đối với các ô cấm một-một, truờng này bị bỏ qua.

Lưu ý rằng tất cả cài đặt mặc định sẽ chỉ ảnh hưởng đến các thư được gửi mà không có cấu trúc `sctp_sndrcvinfo` riêng. Bất kỳ lệnh gửi nào cung cấp cấu trúc này (ví dụ: chức năng `sctp_sendmsg` hoặc `sendmsg` với dữ liệu phụ trợ) sẽ ghi đè cài đặt mặc định. Bên cạnh việc thiết lập các giá trị mặc định, tùy chọn này có thể được sử dụng để truy xuất các tham số mặc định hiện tại bằng cách sử dụng hàm `sctp_opt_info`.

Tùy chọn ở cắm SCTP_DISABLE_FRAGMENTS

SCTP thư ờng phân mảnh bất kỳ tin nhắn người gửi dùng nào không vừa với một gói SCTP thành nhiều khối DATA. Việc đặt tùy chọn này sẽ vô hiệu hóa hành vi này đối với người gửi. Khi tắt tùy chọn này, SCTP sẽ trả về lỗi `EMSGSIZE` và không gửi tin nhắn. Hành vi mặc định là tắt tùy chọn này; SCTP thư ờng sẽ phân đoạn tin nhắn của người gửi.

Tùy chọn này có thể được sử dụng bởi các ứng dụng muốn kiểm soát kích thước tin nhắn, đảm bảo rằng mọi tin nhắn ứng dụng của người gửi dùng vừa với một gói IP duy nhất. Ứng dụng kích hoạt tùy chọn này phải được chuẩn bị để xử lý trường hợp lỗi (tức là thông báo của nó quá lớn) bằng cách cung cấp phân mảnh thông báo ở lớp ứng dụng hoặc thông báo nhỏ hơn.

Tùy chọn ở cắm SCTP_EVENTS

Tùy chọn ở cắm này cho phép người gửi gọi `tim nạp`, bật hoặc tắt các thông báo SCTP khác nhau. Thông báo SCTP là thông báo mà ngăn xếp SCTP sẽ gửi đến ứng dụng. Tin nhắn được đọc dưới dạng dữ liệu bình thường, với trường `msg_flags` của hàm `recvmsg` đang được đặt thành `MSG_NOTIFICATION`. Ứng dụng chưa được chuẩn bị để sử dụng `recvmsg` hoặc `sctp_recvmsg` sẽ không kích hoạt sự kiện. Tám loại sự kiện khác nhau có thể được đăng ký bằng cách sử dụng tùy chọn này và chuyển cấu trúc `sctp_event_subscribe`. Bất kỳ giá trị nào bằng 0 đều biểu thị trạng thái không đăng ký và giá trị 1 biểu thị trạng thái đăng ký.

Cấu trúc `sctp_event_subscribe` có dạng sau:

```
cấu trúc sctp_event_subscribe {
    u_int8_t sctp_data_io_event;
    u_int8_t sctp_association_event;
    u_int8_t sctp_address_event;
    u_int8_t sctp_send_failure_event;
    u_int8_t sctp_peer_error_event;
    u_int8_t sctp_shutdown_event;
    u_int8_t sctp_partial_delivery_event;
    u_int8_t sctp_adaption_layer_event;
```

```
};
```

Hình 7.17 tóm tắt các sự kiện khác nhau. Thông tin chi tiết hơn về thông báo có thể được
đư ợc tìm thấy trong [Phản 9.14](#).

Hình 7.17. Đăng ký sự kiện SCTP.

Constant	Description
sctp_data_io_event	Enable/disable <code>sctp_sndrcvinfo</code> to come with each <code>recvmsg</code>
sctp_association_event	Enable/disable association notifications
sctp_address_event	Enable/disable address notifications
sctp_send_failure_event	Enable/disable message send failure notifications
sctp_peer_error_event	Enable/disable peer protocol error notifications
sctp_shutdown_event	Enable/disable shutdown notifications
sctp_partial_delivery_event	Enable/disable partial-delivery API notifications
sctp_adaption_layer_event	Enable/disable adaption layer notification

Tùy chọn ở cẩm SCTP_GET_PEER_ADDR_INFO

Tùy chọn này truy xuất thông tin về địa chỉ ngang hàng, bao gồm cửa sổ tắc nghẽn, RTT và MTU
đư ợc làm mịn. Tùy chọn này chỉ có thể đư ợc sử dụng để lấy thông tin về một địa chỉ
ngang hàng cụ thể. Ngư ời gọi cung cấp `sctp_paddrinfo`
cấu trúc có trường `spinfo_address` đư ợc điền địa chỉ ngang hàng quan tâm và nên sử dụng
`sctp_opt_info` thay vì `getsockopt` để có tính di động tối đa. Cấu trúc `sctp_paddrinfo` có định
dạng sau:

```
cấu trúc sctp_paddrinfo {
    sctp_assoc_t spinfo_assoc_id;
    struct sockaddr_storage spinfo_address;
    int32_t spinfo_state;
    u_int32_t spinfo_cwnd;
    u_int32_t spinfo_srtt;
    u_int32_t spinfo_rto;
    u_int32_t spinfo_mtu;
};
```

Dữ liệu đư ợc trả về cho ngư ời gọi cung cấp những thông tin sau:

- `spinfo_assoc_id` chứa thông tin nhận dạng liên kết, đồng thời
đư ợc cung cấp trong thông báo "kết nối" (`SCTP_COMM_UP`). Giá trị duy nhất này có thể đư ợc
sử dụng như một phư ơng pháp tốc ký để thể hiện sự liên kết cho hầu hết các hoạt động
SCTP.

- **spinfo_address** được gọi ngay khi thiết lập để thông báo cho ổ cắm SCTP về địa chỉ nào sẽ trả về thông tin. Khi trả lại, giá trị của nó sẽ không thay đổi.
- **spinfo_state** giữ một hoặc nhiều giá trị trong [Hình 7.18](#).

Hình 7.18. Trạng thái địa chỉ ngang hàng SCTP.

Constant	Description
SCTP_ACTIVE	Address is active and reachable
SCTP_INACTIVE	Address cannot currently be reached
SCTP_ADDR_UNCONFIRMED	No heartbeat or data has confirmed this address

Địa chỉ chưa được xác nhận là địa chỉ mà thiết bị ngang hàng đã liệt kê là địa chỉ hợp lệ, như ngay điểm cuối SCTP cục bộ không thể xác nhận rằng thiết bị ngang hàng có giữ địa chỉ đó hay không. Điểm cuối SCTP xác nhận một địa chỉ khi nhịp tim hoặc dữ liệu ngay dùng gửi đến địa chỉ đó được xác nhận. Lưu ý rằng địa chỉ chưa được xác nhận cũng sẽ không có giá trị thời gian chờ truyền lại (RTO) hợp lệ. Địa chỉ hoạt động đại diện cho các địa chỉ được coi là có sẵn để sử dụng.

- **spinfo_cwnd** đại diện cho cửa sổ tắc nghẽn hiện tại được ghi lại cho địa chỉ ngang hàng. Bạn có thể tìm thấy mô tả về cách quản lý giá trị **cwnd** ở trang 177 của [Stewart và Xie 2001].
- **spinfo_srtt** thể hiện ước tính hiện tại của RTT được làm mìn cho việc này Địa chỉ.
- **spinfo_rto** thể hiện thời gian chờ truyền lại hiện tại được sử dụng cho việc này Địa chỉ.
- **spinfo_mtu** đại diện cho đường dẫn MTU hiện tại được phát hiện bởi đường dẫn MTU khám phá.

Một cách sử dụng thú vị cho tùy chọn này là chuyển cấu trúc địa chỉ IP thành nhận dạng liên kết có thể được sử dụng trong các cuộc gọi khác. Chúng tôi sẽ minh họa việc sử dụng tùy chọn ổ cắm này trong [Chú ý 23](#). Một khả năng khác là ứng dụng có thể theo dõi hiệu suất đối với từng địa chỉ của một máy ngang hàng nhiều nhà và cập nhật địa chỉ chính của liên kết thành địa chỉ tốt nhất của máy ngang hàng. Những giá trị này cũng hữu ích cho việc ghi nhật ký và gỡ lỗi.

Tùy chọn ổ cắm SCTP_I_WANT_MAPPED_V4_ADDR

Còn này có thể được sử dụng để bật hoặc tắt các địa chỉ được ánh xạ IPv4 trên ổ cắm **loại AF_INET6**. Lưu ý rằng khi được bật (là hành vi mặc định), tắt cả địa chỉ IPv4 sẽ được ánh xạ tới địa chỉ IPv6 trước khi gửi tới ứng dụng. Nếu tùy chọn này bị tắt, ổ cắm SCTP sẽ không ánh xạ các địa chỉ IPv4 và thay vào đó sẽ chuyển chúng dưới dạng cấu trúc **sockaddr_in**.

Tùy chọn ổ cắm SCTP_INITMSG

Tùy chọn này có thể được sử dụng để nhận hoặc đặt các tham số ban đầu mặc định được sử dụng trên ô cắm SCTP khi gửi tin nhắn INIT. Tùy chọn sử dụng `sctp_initmsg` cấu trúc được xác định như sau:

```
cấu trúc sctp_initmsg {
    uint16_t allow_num_ostreams;
    uint16_t allow_max_instreams;
    uint16_t allow_max_attempts;
    uint16_t allow_max_init_time;
};
```

Các trường này được định nghĩa như sau:

- `sinit_num_ostreams` thể hiện số lượng luồng SCTP gửi đi mà ứng dụng muốn yêu cầu. Giá trị này không được xác nhận cho đến khi liên kết kết thúc quá trình bắt tay ban đầu và có thể được thư ứng lượng xuống dưới thông qua các giới hạn điểm cuối ngang hàng.
- `sinit_max_instreams` thể hiện số lượng luồng vào tối đa mà ứng dụng được chuẩn bị để cho phép. Giá trị này sẽ bị ngăn xếp SCTP ghi đè nếu nó lớn hơn luồng tối đa cho phép mà ngăn xếp SCTP hỗ trợ.
- `sinit_max_attempts` biểu thị số lần ngăn xếp SCTP sẽ gửi thông báo INIT ban đầu trước khi coi điểm cuối ngang hàng là không thể truy cập được.
- `sinit_max_init_timeo` đại diện cho giá trị RTO tối đa cho bộ đếm thời gian INIT. Trong thời gian lùi theo cấp số nhân của bộ định thời ban đầu, giá trị này thay thế `RTO.max` làm mức tràn cho các lần truyền lại. Giá trị này được biểu thị bằng mili giây.

Lưu ý rằng khi thiết lập các trường này, mọi giá trị được đặt thành 0 sẽ bị ô cắm SCTP bỏ qua. Người dùng ô cắm kiểu một-nhiều (được mô tả trong [Phần 9.2](#)) cũng có thể chuyển `cấu trúc sctp_initmsg` trong dữ liệu phụ trợ trong quá trình thiết lập liên kết ngầm.

Tùy chọn ô cắm SCTP_MAXBURST

Tùy chọn ô cắm này cho phép ứng dụng tìm nạp hoặc đặt kích thước cụm tối đa được sử dụng khi gửi gói. Khi triển khai SCTP gửi dữ liệu đến một thiết bị ngang hàng, không quá gói `SCTP_MAXBURST` được gửi cùng một lúc để tránh làm tràn ngập mạng với các gói. Việc triển khai có thể áp dụng giới hạn này bằng cách: (i) giảm cửa sổ tắc nghẽn của nó xuống kích thước chuyến bay hiện tại cộng với kích thước cụm tối đa nhân với đường dẫn MTU hoặc (ii) sử dụng giá trị này làm điều khiển vi mô riêng biệt, gửi tối đa cụm tối đa gói tin ở bất kỳ cơ hội gửi nào.

Tùy chọn ở cắm SCTP_MAXSEG

Tùy chọn ở cắm này cho phép ứng dụng tìm nạp hoặc đặt kích thư ớc phân đoạn tối đa đư ợc sử dụng trong quá trình phân mảnh SCTP. Tùy chọn này tương tự như tùy chọn TCP [TCP_MAXSEG](#) đư ợc mô tả trong [Phần 7.9.](#)

Khi ngư ời gửi SCTP nhận đư ợc tin nhắn từ một ứng dụng lớn hơn giá trị này, ngư ời gửi SCTP sẽ chia tin nhắn thành nhiều phần để vận chuyển đến điểm cuối ngang hàng. Kích thư ớc mà ngư ời gửi SCTP thư ờng sử dụng là MTU nhỏ nhất trong tất cả các địa chỉ đư ợc liên kết với thiết bị ngang hàng. Tùy chọn này ghi đè giá trị này xuống giá trị đư ợc chỉ định. Lưu ý rằng ngăn xếp SCTP có thể phân đoạn tin nhắn ở ranh giới nhỏ hơn so với yêu cầu của tùy chọn này. Sự phân mảnh nhỏ hơn này sẽ xảy ra khi một trong các đường dẫn đến điểm cuối ngang hàng có MTU nhỏ hơn giá trị đư ợc yêu cầu trong tùy chọn [SCTP_MAXSEG](#).

Giá trị này là cài đặt trên toàn điểm cuối và có thể ảnh hưởng đến nhiều liên kết trong kiểu giao diện một-nhiều.

Tùy chọn ở cắm SCTP_NODELAY

Nếu đư ợc đặt, tùy chọn này sẽ tắt thuật toán Nagle của SCTP. Tùy chọn này đư ợc TẮT theo mặc định (nghĩa là thuật toán Nagle đư ợc BẬT theo mặc định). Thuật toán Nagle của SCTP hoạt động giống hệt với TCP ngoại trừ việc nó đang cố gắng kết hợp nhiều khối DATA thay vì chỉ kết hợp các byte trên một luồng. Để thảo luận thêm về thuật toán Nagle, hãy xem [TCP_MAXSEG](#).

Tùy chọn ở cắm SCTP_PEER_ADDR_PARAMS

Tùy chọn ở cắm này cho phép ứng dụng tìm nạp hoặc đặt các tham số khác nhau trên một liên kết. Ngư ời gọi cung cấp cấu trúc [sctp_paddrparams](#), điền thông tin nhận dạng liên kết. Cấu trúc [sctp_paddrparams](#) có định dạng sau:

```
cấu trúc sctp_paddrparams {
    sctp_assoc_t spp_assoc_id;
    cấu trúc sockaddr_storage spp_address;
    u_int32_t spp_hbinterval;
    u_int16_t spp_pathmaxrxt;
};
```

Các trường này đư ợc định nghĩa như sau:

- **spp_assoc_id** giữ nhận dạng liên kết cho thông tin đang đư ợc
đư ợc yêu cầu hoặc thiết lập. Nếu giá trị này đư ợc đặt thành 0 thì các giá trị mặc định của điểm
cuối sẽ đư ợc đặt hoặc truy xuất thay vì các giá trị liên kết cụ thể.
- **spp_address** chỉ định địa chỉ IP mà các tham số này đang đư ợc yêu cầu hoặc đặt. Nếu tru ờng
spp_assoc_id đư ợc đặt thành 0 thì tru ờng này sẽ bị bỏ qua.
- **spp_hbinterval** là khoảng thời gian giữa các nhịp tim. Giá trị **SCTP_NO_HB**
vô hiệu hóa nhịp tim. Giá trị **SCTP_ISSUE_HB** yêu cầu nhịp tim theo yêu cầu. Bất kỳ
giá trị nào khác sẽ thay đổi khoảng nhịp tim thành giá trị này tính bằng mili giây. Khi
thiết lập các tham số mặc định, giá trị **SCTP_ISSUE_HB** không đư ợc phép.
- **spp_hbpathmaxrxt** chứa số lần truyền lại sẽ đư ợc thực hiện
đã thử đến đích này tru ớc khi nó đư ợc tuyên bố KHÔNG HOẠT ĐỘNG. Khi một địa chỉ
đư ợc khai báo KHÔNG HOẠT ĐỘNG, nếu đó là địa chỉ chính thì địa chỉ thay thế sẽ đư ợc
chọn làm địa chỉ chính.

Tùy chọn ở cắm SCTP_PRIMARY_ADDR

Tùy chọn ở cắm này tìm nạp hoặc đặt địa chỉ mà điểm cuối cục bộ đang sử dụng làm địa chỉ chính.
Theo mặc định, địa chỉ chính đư ợc sử dụng làm địa chỉ đích cho tất cả các tin nhắn đư ợc gửi đến
một máy ngang hàng. Để đặt giá trị này, người gọi sẽ điền thông tin nhận dạng liên kết
và địa chỉ ngang hàng sẽ đư ợc sử dụng làm địa chỉ chính. Người gọi chuyển thông tin này theo cấu
trúc **sctp_setprim**, đư ợc định nghĩa là:

```
cấu trúc sctp_setprim {
    sctp_assoc_t           spp_assoc_id;
    cấu trúc sockaddr_storage spp_addr;
};
```

Các tru ờng này đư ợc định nghĩa như sau:

- **spp_assoc_id** chỉ định nhận dạng liên kết mà người yêu cầu muốn đặt hoặc truy xuất địa chỉ
chính hiện tại. Đối với kiểu một-một, tru ờng này bị bỏ qua.
- **spp_addr** chỉ định địa chỉ chính, địa chỉ này phải là địa chỉ thuộc về mạng ngang hàng. Nếu
hoạt động là lệnh gọi hàm **setsockopt** thì giá trị trong tru ờng này biểu thị địa chỉ ngang
hàng mới mà người yêu cầu muốn chuyển thành địa chỉ đích chính.

Lưu ý rằng việc truy xuất giá trị của tùy chọn này trên ở cắm một-một chỉ có một địa chỉ cục bộ
đư ợc liên kết với nó cũng giống như gọi **getsockname**.

Tùy chọn ở cắm SCTP_RTOINFO

Tùy chọn ở cắm này có thể được sử dụng để tìm nạp hoặc đặt nhiều thông tin RTO khác nhau trên một liên kết cụ thể hoặc các giá trị mặc định được điểm cuối này sử dụng. Khi tìm nạp, người gọi nên sử dụng `sctp_opt_info` thay vì `getsockopt` để có tính di động tối đa. Người gọi cung cấp cấu trúc `sctp_rtoinfo` có dạng sau:

```
cấu trúc sctp_rtoinfo {
    sctp_assoc          srto_assoc_id;
    uint32_t            srto_initial;
    uint32_t            srto_max;
    uint32_t            srto_min;
};
```

Các trường này được định nghĩa như sau:

- `srto_assoc_id` chứa liên kết quan tâm cụ thể hoặc 0. Nếu điều này trống chứa giá trị 0 thì các tham số mặc định của hệ thống sẽ bị ảnh hưởng bởi lệnh gọi.
- `srto_initial` chứa giá trị RTO ban đầu được sử dụng cho địa chỉ ngang hàng. Các RTO ban đầu được sử dụng khi gửi đoạn INIT tới thiết bị ngang hàng. Giá trị này tính bằng mili giây và có giá trị mặc định là 3.000.
- `srto_max` chứa giá trị RTO tối đa sẽ được sử dụng khi thực hiện cập nhật cho bộ định thời truyềnlại. Nếu giá trị cập nhật lớn hơn mức tối đa RTO thì mức tối đa RTO sẽ được sử dụng làm RTO thay vì giá trị được tính toán. Giá trị mặc định cho trường này là 60.000 mili giây.
- `srto_min` chứa giá trị RTO tối thiểu sẽ được sử dụng khi khởi động bộ định thời truyềnlại. Bất cứ khi nào một bản cập nhật được thực hiện cho bộ định thời RTO, giá trị tối thiểu RTO sẽ được kiểm tra so với giá trị mới. Nếu giá trị mới nhỏ hơn giá trị tối thiểu thì giá trị tối thiểu sẽ thay thế giá trị mới. Giá trị mặc định cho trường này là 1.000 mili giây.

Giá trị 0 cho `srto_initial`, `srto_max` hoặc `srto_min` cho biết rằng không nên thay đổi giá trị mặc định hiện được đặt. Tất cả các giá trị thời gian được biểu thị bằng mili giây. Chúng tôi cung cấp hướng dẫn về cách đặt các bộ hẹn giờ này để thực hiện trong [Phần 23.11](#).

Tùy chọn ở cắm SCTP_SET_PEER_PRIMARY_ADDR

Việc đặt tùy chọn này sẽ khiến một thông báo được gửi yêu cầu thiết bị ngang hàng đặt địa chỉ cụ bộ được chỉ định làm địa chỉ chính của nó. Người gọi cung cấp một

cấu trúc `sctp_setpeerprim` và phải điền cả thông tin nhận dạng liên kết và địa chỉ cục bộ để yêu cầu nhãn hiệu ngang hàng làm nhãn hiệu chính. Địa chỉ được cung cấp phải là một trong những địa chỉ bị ràng buộc của điểm cuối cục bộ. Cấu trúc `sctp_setpeerprim` là như sau:

```
cấu trúc sctp_setpeerprim {
    sctp_assoc_t           sspp_assoc_id;
    cấu trúc sockaddr sspp_addr;
};
```

Các trường này được định nghĩa như sau:

- `sspp_assoc_id` chỉ định nhận dạng liên kết mà người yêu cầu muốn đặt địa chỉ chính. Đối với kiểu một-một, trường này bị bỏ qua.
- `sspp_addr` giữ địa chỉ cục bộ mà người yêu cầu muốn hỏi ngang hàng hệ thống để đặt làm địa chỉ chính.

Tính năng này là tùy chọn và phải được cả hai điểm cuối hỗ trợ để hoạt động. Nếu điểm cuối cục bộ không hỗ trợ tính năng này, lỗi `EOPNOTSUPP` sẽ được trả về cho người gọi. Nếu điểm cuối từ xa không hỗ trợ tính năng này thì sẽ xảy ra lỗi `EINVAL`.
sẽ được trả lại cho người gọi. Lưu ý rằng giá trị này chỉ có thể được đặt và không thể truy xuất được.

Tùy chọn ô cảm SCTP_STATUS

Tùy chọn ô cảm này sẽ truy xuất trạng thái hiện tại của liên kết SCTP. Người gọi nên sử dụng `sctp_opt_info` thay vì `getaddrinfo` để có tính di động tối đa. Người gọi cung cấp cấu trúc `sctp_status`, điền vào trường nhận dạng liên kết, `sstat_assoc_id`. Cấu trúc sẽ được trả về với đầy đủ thông tin liên quan đến liên kết được yêu cầu. Cấu trúc `sctp_status` có định dạng sau:

```
cấu trúc sctp_status {
    sctp_assoc_t sstat_assoc_id;
    int32_t sstat_state;
    u_int32_t sstat_rwnd;
    u_int16_t sstat_unackdata;
```

```

u_int16_t sstat_penddata;
u_int16_t sstat_instrms;
u_int16_t sstat_outstrms;
u_int32_t sstat_fragmentation_point;
struct sctp_paddrinfo sstat_primary;
};

```

Các trường này được định nghĩa như sau:

- `sstat_assoc_id` giữ nhận dạng liên kết.
- `sstat_state` giữ một trong các giá trị được tìm thấy trong [Hình 7.19](#) và cho biết trạng thái tổng thể của liên kết. Có thể tìm thấy mô tả chi tiết về các trạng thái mà điểm cuối SCTP trải qua trong quá trình thiết lập liên kết hoặc tắt máy có thể được tìm thấy [trong Hình 2.8](#).

Hình 7.19. trạng thái SCTP.

Constant	Description
<code>SCTP_CLOSED</code>	A closed association
<code>SCTP_COOKIE_WAIT</code>	An association that has sent an INIT
<code>SCTP_COOKIE_ECHOED</code>	An association that has echoed the COOKIE
<code>SCTP_ESTABLISHED</code>	An established association
<code>SCTP_SHUTDOWN_PENDING</code>	An association pending sending the shutdown
<code>SCTP_SHUTDOWN_SENT</code>	An association that has sent a shutdown
<code>SCTP_SHUTDOWN_RECEIVED</code>	An association that has received a shutdown
<code>SCTP_SHUTDOWN_ACK_SENT</code>	An association that is waiting for a SHUTDOWN-COMPLETE

- `sstat_rwnd` giữ ước tính hiện tại của điểm cuối về mức nhận của thiết bị ngang hàng cửa sổ.
- `sstat_unackdata` chứa số khối DATA chưa được xác nhận đang chờ xử lý cho thiết bị ngang hàng.
- `sstat_penddata` chứa số khối DATA chưa đọc mà điểm cuối cục bộ đang giữ để ứng dụng đọc.
- `sstat_instrms` chứa số luồng mà thiết bị ngang hàng đang sử dụng để gửi dữ liệu tới điểm cuối này.
- `sstat_outstrms` chứa số luồng cho phép mà điểm cuối này có thể sử dụng để gửi dữ liệu đến ngang hàng.
- `sstat_fragmentation_point` chứa giá trị hiện tại của SCTP cục bộ điểm cuối đang sử dụng làm điểm phân mảnh cho tin nhắn của người dùng. Giá trị này thường là MTU nhỏ nhất trong tất cả các đích hoặc có thể là giá trị nhỏ hơn do ứng dụng cục bộ đặt với `SCTP_MAXSEG`.
- `sstat_primary` giữ địa chỉ chính hiện tại. Địa chỉ chính là địa chỉ mặc định được sử dụng khi gửi dữ liệu đến điểm cuối ngang hàng.

Những giá trị này hữu ích cho việc chẩn đoán và xác định các đặc điểm của phiên; ví dụ: hàm `sctp_get_no_strms` trong [Phần 10.2](#) sẽ sử dụng

`sstat_outstrms` thành viên để xác định có bao nhiêu luồng có sẵn để sử dụng bên ngoài. Giá trị `sstat_rwnd` thấp và/hoặc giá trị `sstat_unackdata` cao có thể được sử dụng để xác định rằng bộ đệm ở cắm nhận của thiết bị ngang hàng sắp đầy. Giá trị này có thể được sử dụng làm tín hiệu để ứng dụng giảm tốc độ truyền nếu có thể. Một số ứng dụng có thể sử dụng `sstat_fragmentation_point` để giảm số lưu lượng đoạn mà SCTP phải tạo bằng cách gửi ứng dụng nhỏ hơn tin nhắn.

7.11 Hàm 'fcntl'

`fcntl` là viết tắt của "điều khiển tệp" và hàm này thực hiện các hoạt động điều khiển mô tả khác nhau. Trước khi mô tả chức năng và cách nó ảnh hưởng đến ở cắm, chúng ta cần nhìn vào bức tranh toàn cảnh hơn. [Hình 7.20](#) tóm tắt các hoạt động khác nhau được thực hiện bởi các ô cắm `fcntl`, `ioctl` và định tuyến.

Hình 7.20. Tóm tắt các hoạt động của `fcntl`, `ioctl` và định tuyến socket.

Operation	fcntl	ioctl	Routing socket	POSIX
Set socket for nonblocking I/O	<code>F_SETFL, O_NONBLOCK</code>	<code>FIONBIO</code>		<code>fcntl</code>
Set socket for signal-driven I/O	<code>F_SETFL, O_ASYNC</code>	<code>FIOASYNC</code>		<code>fcntl</code>
Set socket owner	<code>F_SETOWN</code>	<code>SIOCSPGRP or FIOSETOWN</code>		<code>fcntl</code>
Get socket owner	<code>F_GETOWN</code>	<code>SIOCGPGRP or FIOGETOWN</code>		<code>fcntl</code>
Get # bytes in socket receive buffer		<code>FIONREAD</code>		
Test for socket at out-of-band mark		<code>SIOCATMARK</code>		<code>socketmark</code>
Obtain interface list		<code>SIOCGIFCONF</code>	<code>sysctl</code>	
Interface operations		<code>SIOC[GS] IFXXX</code>		
ARP cache operations		<code>SIOCGARP</code>	<code>RTM_XXX</code>	
Routing table operations		<code>SIOCxxxRT</code>	<code>RTM_XXX</code>	

Sáu thao tác đầu tiên có thể được áp dụng cho ô cắm bằng bất kỳ quy trình nào; hai cái thứ hai (thao tác giao diện) ít phổ biến hơn, như ng vẫn có mục đích chung; và hai cái cuối cùng (ARP và bảng định tuyến) được cấp bởi các chương trình quản trị như `ifconfig` và `lộ trình`. Chúng ta sẽ nói nhiều hơn về các hoạt động `ioctl` khác nhau trong [Chương 17](#) và ở cắm định tuyến trong [Chương 18](#).

Có nhiều cách để thực hiện bốn thao tác đầu tiên, như ng chúng tôi lưu ý trong cột cuối cùng rằng POSIX chỉ định rằng `fcntl` là cách ưu tiên. Chúng tôi cũng lưu ý rằng POSIX cung cấp chức năng `socketmark` ([Phần 24.3](#)) làm cách ưu tiên để kiểm tra dấu ngoài dài. Các thao tác còn lại, với cột cuối cùng trống, chưa được POSIX chuẩn hóa.

Chúng tôi cũng lưu ý rằng hai thao tác đầu tiên, thiết lập ô cắm cho I/O không chặn và cho I/O điều khiển tín hiệu, đã được đặt trước đây bằng cách sử dụng `FNDELAY` và `FASYNC`.
lệnh với `fcntl`. POSIX xác định hằng số `O_XXX`.

Hàm `fcntl` cung cấp các tính năng sau liên quan đến lập trình mạng:

- I/O không chặn– Chúng ta có thể đặt cờ trạng thái tệp `O_NONBLOCK` bằng lệnh `F_SETFL` để đặt ở cắm ở trạng thái không chặn. Chúng ta sẽ mô tả I/O không chặn trong [Chương 16](#).
- I/O điều khiển bằng tín hiệu– Chúng ta có thể đặt cờ trạng thái tệp `O_ASYNC` bằng `F_SETFL` lệnh, khiến tín hiệu `SIGIO` được tạo khi trạng thái của ổ cắm thay đổi. Chúng ta sẽ thảo luận điều này ở [Chương 25](#).
- Lệnh `F_SETOWN` cho phép chúng ta thiết lập chủ sở hữu socket (ID tiến trình hoặc ID nhóm xử lý) để nhận tín hiệu `SIGIO` và `SIGURG`. Tín hiệu trứ ớc được tạo khi I/O điều khiển bằng tín hiệu được bật cho ổ cắm ([Chương 25](#)) và tín hiệu sau được tạo khi dữ liệu ngoài băng tần mới đến ổ cắm

([Chương 24](#)). Lệnh `F_GETOWN` trả về chủ sở hữu hiện tại của ổ cắm.

Thuật ngữ "chủ sở hữu ổ cắm" được xác định bởi POSIX. Trong lịch sử, các triển khai có nguồn gốc từ Berkeley đã gọi đây là "ID nhóm quy trình của ổ cắm" vì biến lưu trữ ID này là thành viên `so_pgid` của cấu trúc `đo cắm` (tr. 438 của TCPv2).

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Trả về: phụ thuộc vào cmd nếu OK, -1 nếu có lỗi

Mỗi bộ mô tả (bao gồm một ổ cắm) có một tập hợp các cờ tệp được tìm nạp bằng lệnh `F_GETFL` và được thiết lập bằng lệnh `F_SETFL`. Hai cờ ảnh hưởng đến ổ cắm là

- `O_NONBLOCK`–I/O không chặn
- `O_ASYNC`–I/O điều khiển bằng tín hiệu

Chúng tôi sẽ mô tả cả hai tính năng này chi tiết hơn sau. Hiện tại, chúng tôi lưu ý rằng mã điển hình để bắt I/O không chặn, sử dụng `fcntl`, sẽ là:

```
int      cờ;
/* Đặt ổ cắm ở chế độ không chặn */
nếu ( (flags = fcntl (fd, F_GETFL, 0)) < 0)
    err_sys ("Lỗi F_GETFL");
```

```
cờ |= O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("Lỗi F_SETFL");
```

Hãy cẩn thận với mã mà bạn có thể gặp phải khi chỉ đặt cờ mong muốn.

```
/* Cách đặt sai ở chế độ không chặn */
nếu (fcntl(fd, F_SETFL, O_NONBLOCK) < 0)
    err_sys("Lỗi F_SETFL");
```

Trong khi điều này đặt cờ không chặn, nó cũng xóa tất cả các cờ trạng thái tệp khác. Cách chính xác duy nhất để đặt một trong các cờ trạng thái tệp là tìm nạp các cờ hiện tại, một cách logic HỌC trong cờ mới, sau đó đặt cờ.

Đoạn mã sau tắt cờ không chặn, giả sử **các cờ** được đặt bằng lệnh gọi tới **fcntl** được hiển thị ở trên:

```
cờ &= ~O_NONBLOCK;
if (fcntl(fd, F_SETFL, flags) < 0)
    err_sys("Lỗi F_SETFL");
```

Hai tín hiệu **SIGIO** và **SIGURG** khác với các tín hiệu khác ở chỗ chúng chỉ được tạo cho ở cắm nếu ở cắm đã được gán chủ sở hữu bằng lệnh **F_SETOWN**. Giá trị đối số nguyên cho lệnh **F_SETOWN** có thể là số nguyên dương, chỉ định ID tiến trình để nhận tín hiệu hoặc số nguyên âm có giá trị tuyệt đối là ID nhóm quy trình để nhận tín hiệu. Lệnh **F_GETOWN** trả về chủ sở hữu socket làm giá trị trả về từ **fcntl**

hàm, ID tiến trình (giá trị trả về dương) hoặc ID nhóm quy trình (giá trị âm khác với -1). Sự khác biệt giữa việc chỉ định một quy trình hoặc một nhóm quy trình nhận tín hiệu là quy trình truớc chỉ khiến một quy trình duy nhất nhận tín hiệu, trong khi quy trình sau khiến tất cả quy trình trong nhóm quy trình (có thể nhiều hơn một) nhận tín hiệu.

Khi một ô cắm mới được tạo bởi **để cắm**, nó không có chủ sở hữu. Như ng khi một ô cắm mới được tạo từ ô cắm nghe, chủ sở hữu ô cắm đó sẽ được kế thừa từ ô cắm nghe.

ở cắm bằng ở cắm được kết nối (cũng như nhiều tùy chọn ở cắm [trang 462-463 của TCPv2]).

7.12 Tóm tắt

Các tùy chọn ở cắm chạy phạm vi từ rất chung chung (`SO_ERROR`) đến rất cụ thể (tùy chọn tiêu đề IP). Các tùy chọn được sử dụng phổ biến nhất mà chúng ta có thể gắp là `SO_KEEPALIVE`, `SO_RCVBUF`, `SO_SNDBUF` và `SO_REUSEADDR`. Cái sau phải luôn được đặt cho máy chủ TCP trước khi nó gọi [liên kết](#) ([Hình 11.12](#)). Tùy chọn `SO_BROADCAST` và 10 tùy chọn [ở cắm phát đa](#) hứa ứng chỉ dành cho các ứng dụng phát sóng hoặc phát đa hứa ứng tư ứng ứng.

Tùy chọn ở cắm `SO_KEEPALIVE` được nhiều máy chủ TCP đặt và tự động chấm dứt kết nối nửa mở. Tính năng hay của tùy chọn này là nó được xử lý bởi lớp TCP mà không yêu cầu bộ đếm thời gian không hoạt động ở cấp ứng dụng; như điểm của nó là không thể phân biệt được sự khác biệt giữa máy chủ máy khách bị lỗi và việc mất kết nối tạm thời với máy khách. SCTP cung cấp 17 tùy chọn socket được ứng dụng sử dụng để điều khiển việc vận chuyển. `SCTP_NODELAY` và `SCTP_MAXSEG` tư ứng tự như `TCP_NODELAY` và `TCP_MAXSEG` và thực hiện các chức năng tư ứng đương. 15 tùy chọn khác giúp ứng dụng kiểm soát tốt hơn ngăn xếp SCTP; chúng ta sẽ thảo luận về việc sử dụng nhiều tùy chọn ở cắm này trong [Chu ơng 23](#).

Tùy chọn ở cắm `SO_Linger` cho phép chúng tôi kiểm soát nhiều hơn khi đóng trả lại và cũng cho phép chúng tôi buộc gửi RST thay vì trình tự chấm dứt kết nối bốn gói của TCP. Chúng ta phải cẩn thận khi gửi RST vì điều này tránh được trạng thái `TIME_WAIT` của TCP. Trong nhiều trường hợp, tùy chọn ở cắm này không cung cấp thông tin mà chúng ta cần, trong trường hợp đó cần có ACK cấp ứng dụng.

Mỗi ở cắm TCP và SCTP đều có bộ đệm gửi và bộ đệm nhận và mọi ở cắm UDP đều có bộ đệm nhận. Các tùy chọn ở cắm `SO_SNDBUF` và `SO_RCVBUF` cho phép chúng tôi thay đổi kích thước của các bộ đệm này. Cách sử dụng phổ biến nhất của các tùy chọn này là để truyền dữ liệu số lưu lượng lớn qua các đường ống dài: kết nối TCP có băng thông cao hoặc độ trễ dài, thường sử dụng phần mở rộng RFC 1323. Mặt khác, các ở cắm UDP có thể muốn tăng kích thước của bộ đệm nhận để cho phép hắt nhân xếp hàng nhiều gói dữ liệu hơn nếu ứng dụng đang bận.

Bài tập

- 7.1 Viết chương trình in TCP, UDP và SCTP gửi và nhận mặc định kích thước bộ đệm và chạy nó trên các hệ thống mà bạn có quyền truy cập.

7.2 Sửa đổi [Hình 1.5](#) như sau: Trừ ớc khi gọi `connect`, gọi `getsockopt` để lấy kích thư ớc bộ đệm nhận socket và MSS. In cả hai giá trị. Sau đó `connect` trả về thành công, tìm nạp hai tùy chọn ở cẩm này và in giá trị của chúng. Các giá trị có thay đổi không? Tại sao? Chạy chương trình kết nối với máy chủ trên mạng cục bộ của bạn và cũng chạy chương trình kết nối với máy chủ trên mạng từ xa. MSS có thay đổi không? Tại sao? Bạn cũng nên chạy chương trình trên bát kỳ máy chủ nào khác mà bạn có quyền truy cập.

7.3 Bắt đầu với máy chủ TCP của chúng tôi từ [Hình 5.2](#) và [5.3](#) và máy khách TCP của chúng tôi từ [Hình 5.4](#) và [5.5](#). Sửa đổi chức năng `main` của máy khách để đặt `SO_Linger`

tùy chọn socket trư ớc khi gọi `exit`, đặt `l_onoff` thành 1 và `l_linger` thành 0.

Khởi động máy chủ và sau đó khởi động máy khách. Nhập một hoặc hai dòng vào máy khách để xác minh thao tác, sau đó kết thúc máy khách bằng cách nhập ký tự EOF của bạn. Điều gì xảy ra? Sau khi bạn chấm dứt máy khách, hãy chạy `netstat` trên máy chủ máy khách và xem liệu ở cẩm có chuyển sang trạng thái `TIME_WAIT` hay không.

7.4 Giả sử hai máy khách TCP khởi động cùng lúc. Cả hai đều đặt

Tùy chọn ở cẩm `SO_REUSEADDR` và sau đó gọi `listen` với cùng một địa chỉ IP cục bộ và cùng một cổng cục bộ (giả sử 1500). Tuy nhiên, một khách hàng `kết nối` với cổng 198.69.10.2 7000 và khách hàng thứ hai `kết nối` với 198.69.10.2 (cùng địa chỉ IP ngang hàng) như ng cổng 8000. Mô tả tình trạng tranh xảy ra.

7.5 Lấy mã nguồn cho các ví dụ trong cuốn sách này (xem Lời nói đầu) và biên dịch chương trình `sock` ([Phần C.3](#)). Trước tiên, hãy phân loại máy chủ của bạn là (a) không hỗ trợ phát đa hường, (b) hỗ trợ phát đa hường như ng `SO_REUSEPORT` không đư ợc cung cấp hoặc (c) hỗ trợ phát đa hường và `SO_REUSEPORT` đư ợc cung cấp. Hãy thử khởi động nhiều phiên bản của chương trình `sock` dưới dạng máy chủ TCP (tùy chọn dòng lệnh `-s`) trên cùng một cổng, liên kết địa chỉ ký tự đại diện, một trong các địa chỉ giao diện máy chủ của bạn và địa chỉ vòng lặp. Bạn có cần chỉ định tùy chọn `SO_REUSEADDR` (tùy chọn dòng lệnh `-A`) không? Sử dụng `netstat` để xem các ở cẩm nghe.

7.6 Tiếp tục ví dụ trư ớc, như ng khởi động máy chủ UDP (dòng lệnh `-u option`) và thử khởi động hai phiên bản, cả hai đều liên kết cùng một địa chỉ IP và cổng cục bộ. Nếu việc triển khai của bạn hỗ trợ `SO_REUSEPORT`, hãy thử sử dụng nó (`-T` tùy chọn dòng lệnh).

7.7 Nhiều phiên bản của chương trình `ping` có cờ `-d` để bật `SO_DEBUG` tùy chọn ở cẩm. Cái này làm gì?

7.8 Tiếp tục ví dụ ở cuối phần thảo luận của chúng ta về `TCP_NODELAY`

tùy chọn socket, giả sử rằng máy khách thực hiện hai **lần ghi**: lần ghi đầu tiên là 4 byte và lần ghi thứ hai là 396 byte. Cũng giả sử rằng thời gian ACK bị trễ của máy chủ là 100 ms, RTT giữa máy khách và máy chủ là 100 ms và thời gian xử lý yêu cầu của máy khách là 50 ms. Vẽ một dòng thời gian cho thấy

sự tương tác của thuật toán Nagle với các ACK bị trì hoãn.

7.9 Làm lại bài tập trư ớc, giả sử tùy chọn socket `TCP_NODELAY` đư ợc đặt.

7.10 Làm lại bài tập 7.8 giả sử tiến trình gọi `writenv` một lần, cho cả hai bộ đệm 4 byte và bộ đệm 396 byte.

7.11 Đọc RFC 1122 [Braden 1989] để xác định khoảng thời gian đư ợc khuyến nghị cho ACK bị trì hoãn.

7.12 Máy chủ của chúng tôi trong Hình 5.2 và 5.3 dành phần lớn thời gian ở đâu? Giả sử máy chủ đặt tùy chọn ô cắm `SO_KEEPALIVE`, không có dữ liệu nào đư ợc trao đổi qua kết nối và máy chủ khách gặp sự cố và không khởi động lại. Điều gì xảy ra?

7.13 Khách hàng của chúng ta trong Hình 5.4 và 5.5 dành phần lớn thời gian ở đâu? Giả sử máy khách đặt tùy chọn ô cắm `SO_KEEPALIVE`, không có dữ liệu nào đư ợc trao đổi qua kết nối và máy chủ lưu trữ gặp sự cố và không khởi động lại. Điều gì xảy ra?

7.14 Khách hàng của chúng ta trong Hình 5.4 và 6.13 dành phần lớn thời gian ở đâu? Giả sử máy khách đặt tùy chọn ô cắm `SO_KEEPALIVE`, không có dữ liệu nào đư ợc trao đổi qua kết nối và máy chủ lưu trữ gặp sự cố và không khởi động lại. Điều gì xảy ra?

7.15 Giả sử cả máy khách và máy chủ đều đặt tùy chọn ô cắm `SO_KEEPALIVE`. Kết nối đư ợc duy trì giữa hai thiết bị ngang hàng, nhưng không có dữ liệu ứng dụng nào đư ợc trao đổi qua kết nối. Khi bộ đếm thời gian duy trì hết hai giờ một lần, có bao nhiêu phân đoạn TCP đư ợc trao đổi trên kết nối?

7.16 Hầu hết tất cả các triển khai đều xác định hằng số `SO_ACCEPTCONN` trong tiêu đề `<sys/socket.h>`, nhưng chúng tôi chưa mô tả tùy chọn này. Hãy đọc [Lanciani 1996] để tìm hiểu lý do tại sao lựa chọn này tồn tại.

Chương 8. Ô cắm UDP cơ bản

Mục 8.1. Giới thiệu

Mục 8.2. Chức năng `recvfrom` và `sendto`

Mục 8.3. Máy chủ Echo UDP: Chức năng chính

Mục 8.4. Máy chủ Echo UDP: Chức năng `dg_echo`

[Mục 8.5. Máy khách UDP Echo: Chức năng chính](#)[Mục 8.6. Máy khách UDP Echo: Hàm dg_cli](#)[Mục 8.7. Datagram bị mất](#)[Mục 8.8. Xác minh phản hồi đã nhận](#)[Mục 8.9. Máy chủ không chạy](#)[Mục 8.10. Tóm tắt ví dụ UDP](#)[Mục 8.11. Kết nối chức năng với UDP](#)[Mục 8.12. Hàm dg_cli \(Đã xem lại\)](#)[Mục 8.13. Thiếu kiểm soát luồng với UDP](#)[Mục 8.14. Xác định giao diện gửi đi với UDP](#)[Mục 8.15. Máy chủ Echo TCP và UDP Sử dụng chọn](#)[Mục 8.16. Bản tóm tắt](#)[Bài tập](#)

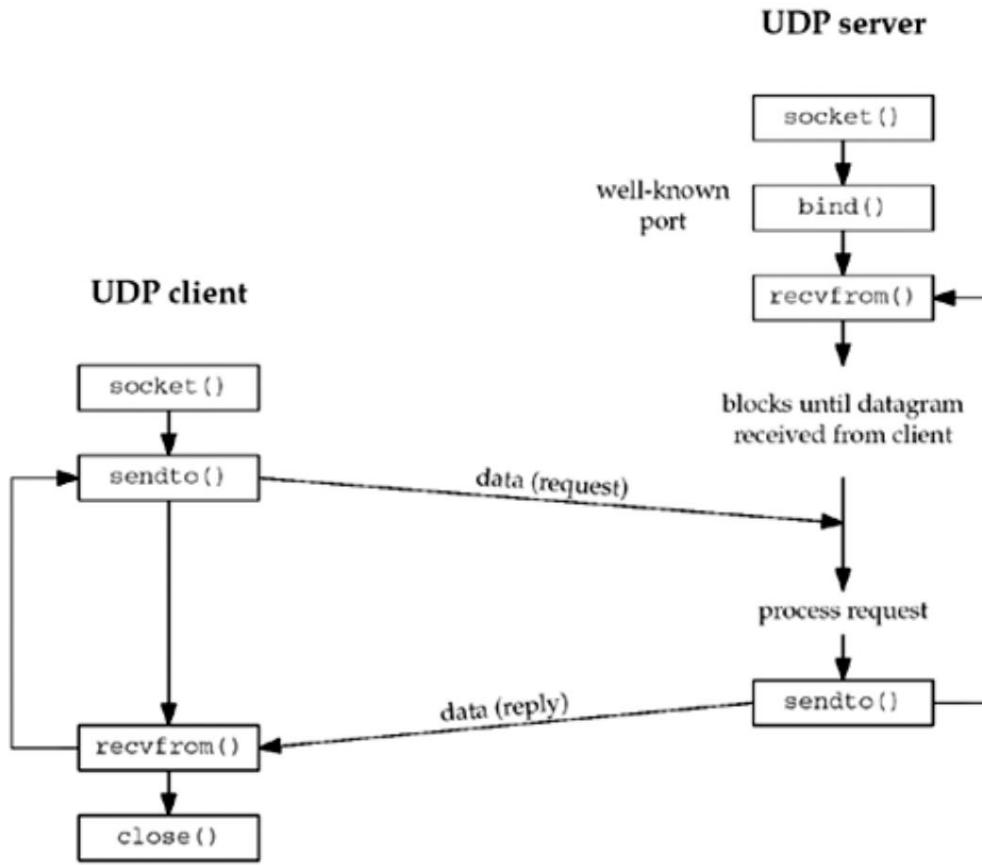
8.1 Giới thiệu

Có một số khác biệt cơ bản giữa các ứng dụng được viết bằng TCP và các ứng dụng sử dụng UDP. Điều này là do sự khác biệt trong hai lớp truyền tải: UDP là một giao thức datagram không có kết nối, không đáng tin cậy, hoàn toàn không giống với luồng byte đáng tin cậy, hống kết nối được cung cấp bởi TCP. Tuy nhiên, có những trường hợp sử dụng UDP thay vì TCP là hợp lý và chúng ta sẽ xem xét lựa chọn thiết kế này trong [Phần 22.4](#). Ví dụ: một số ứng dụng phổ biến được xây dựng bằng UDP: DNS, [NFS](#) và [SNMP](#).

[Hình 8.1](#) cho thấy các lệnh gọi hàm dành cho máy khách/máy chủ UDP điển hình. Máy khách không thiết lập kết nối với máy chủ. Thay vào đó, máy khách chỉ gửi một datagram đến máy chủ bằng hàm `sendto` (được mô tả trong phần tiếp theo), hàm này yêu cầu địa chỉ đích (máy chủ) làm tham số. Tương tự, máy chủ không chấp nhận kết nối từ máy khách. Thay vào đó, máy chủ chỉ gọi `recvfrom`

chức năng chờ cho đến khi dữ liệu đến từ một số máy khách. `recvfrom` trả về địa chỉ giao thức của máy khách, cùng với datagram, để máy chủ có thể gửi phản hồi đến đúng máy khách.

Hình 8.1. Chức năng socket cho máy khách/máy chủ UDP.



[Hình 8.1](#) cho thấy dòng thời gian của kịch bản diễn hình diễn ra trong trao đổi máy khách/máy chủ UDP. Chúng ta có thể so sánh điều này với trao đổi TCP diễn hình được thể hiện trong [Hình 4.1](#).

Trong chương này, chúng tôi sẽ mô tả các chức năng mới mà chúng tôi sử dụng với UDP socket, `recvfrom` và `sendto`, đồng thời làm lại máy khách/máy chủ echo của chúng tôi để sử dụng UDP. Chúng tôi cũng sẽ mô tả việc sử dụng chức năng **kết nối** với ổ cắm UDP và khái niệm về lối không đồng bộ.

8.2 Hàm 'recvfrom' và 'sendto'

Hai hàm này tương tự như các hàm **đọc** và **ghi** tiêu chuẩn, nhưng cần có ba đối số bổ sung.

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from,
socklen_t *addrlen);
```

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

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const struct sockaddr
*to, socklen_t addrlen);
```

Cả hai đều trả về: số byte được đọc hoặc ghi nếu OK, -1 bị lỗi

Ba đối số đầu tiên, sockfd, buff và nbytes, giống hệt với ba đối số đầu tiên để **đọc** và **ghi**: bộ mô tả, con trỏ tới bộ đệm để đọc vào hoặc ghi từ đó và số byte để đọc hoặc ghi.

Chúng ta sẽ mô tả đối số flags trong [Chương 14 khi thảo luận](#) về các hàm **recv**, **send**, **recvmsg** và **sendmsg**, vì chúng ta không cần chúng với ví dụ máy khách/máy chủ UDP đơn giản trong chương này. Hiện tại, chúng tôi sẽ luôn đặt cờ thành 0.

Đối số to cho **sendto** là cấu trúc địa chỉ ở cắm chứa địa chỉ giao thức (ví dụ: địa chỉ IP và số cổng) nơi dữ liệu sẽ được gửi. Kích thước của cấu trúc địa chỉ socket này được chỉ định bởi addrlen. Hàm **recvfrom** điền vào cấu trúc địa chỉ socket được trả đến bởi địa chỉ giao thức của người đã gửi datagram. Số byte được lưu trữ trong cấu trúc địa chỉ socket này cũng được trả về cho người gọi dưới dạng số nguyên được trả bởi addrlen. Lưu ý rằng đối số cuối cùng của **sendto** là một giá trị số nguyên, trong khi đối số cuối cùng của **recvfrom** là một con trỏ tới một giá trị số nguyên (đối số giá trị-kết quả).

Hai đối số cuối cùng của **recvfrom** tương tự như hai đối số cuối cùng cần **chấp nhận**: Nội dung của cấu trúc địa chỉ socket khi trả về cho chúng ta biết ai đã gửi datagram (trong trường hợp UDP) hoặc ai đã khởi tạo kết nối (trong trường hợp TCP) .

Hai đối số cuối cùng của **sendto** tương tự như hai đối số cuối cùng để **kết nối**: Chúng ta điền vào cấu trúc địa chỉ socket địa chỉ giao thức về nơi gửi datagram (trong trường hợp UDP) hoặc với ai để thiết lập kết nối (trong trường hợp TCP).

Cả hai hàm đều trả về độ dài của dữ liệu được đọc hoặc ghi dưới dạng giá trị của hàm. Trong cách sử dụng điển hình của **recvfrom**, với giao thức datagram, giá trị trả về là lượng dữ liệu người dùng trong datagram nhận được.

Việc viết một datagram có độ dài 0 là có thể chấp nhận được. Trong trường hợp UDP, điều này dẫn đến một gói dữ liệu IP chứa tiêu đề IP (thường là 20 byte cho IPv4 và 40 byte cho IPv6), tiêu đề UDP 8 byte và không có dữ liệu. Điều này cũng có nghĩa là giá trị trả về 0 từ **recvfrom** có thể được chấp nhận đối với giao thức datagram: Điều đó không có nghĩa là thiết bị ngang hàng đã đóng kết nối, cũng như giá trị trả về 0 khi **đọc** trên ổ cắm TCP. Vì UDP không có kết nối nên không có chuyện đóng kết nối UDP.

Nếu đối số from tới `recvfrom` là con trỏ null thì đối số độ dài tự ứng ứng (`addrlen`) cũng phải là con trỏ null và điều này cho thấy rằng chúng ta không quan tâm đến việc biết địa chỉ giao thức của người đã gửi dữ liệu cho chúng ta.

Cả `recvfrom` và `sendto` đều có thể được sử dụng với TCP, mặc dù thông thường không có lý do gì để làm như vậy.

8.3 UDP Echo Server: Chức năng 'chính'

Bây giờ chúng ta sẽ làm lại máy khách/máy chủ echo đơn giản của mình từ [Chương 5](#) bằng UDP. Các chương trình máy khách và máy chủ UDP của chúng tôi tuân theo luồng lệnh gọi hàm mà chúng tôi đã sơ đồ hóa trong [Hình 8.1](#). [Hình 8.2](#) mô tả các chức năng được sử dụng. [Hình 8.3](#) hiển thị máy chủ chính — chức năng.

Hình 8.2. Máy khách/máy chủ echo đơn giản sử dụng UDP.



Hình 8.3 Máy chủ tiếng vang UDP.

`udpciserv/udpserv01.c`

```

1 #bao gồm           "unp.h"
2 số nguyên
3 chinh(int argc, char **argv)
4 {
5     int      sockfd;
6     struct sockaddr_in servaddr, cliaddr;

7     sockfd = Ô cắm(AF_INET, SOCK_DGRAM, 0);

8     bzero(&servaddr, sizeof(servaddr));
9     servaddr.sin_family = AF_INET;
10    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11    servaddr.sin_port = htons(SERV_PORT);

12    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
  
```

```

13     dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }

```

Tạo socket UDP, liên kết cổng well-known của server

7-12 Chúng tôi tạo ô cắm UDP bằng cách chỉ định đối số thứ hai cho ô cắm là **SOCK_DGRAM** (ô cắm datagram trong giao thức IPv4). Như với ví dụ về máy chủ TCP, địa chỉ IPv4 cho **liên kết** được chỉ định là **INADDR_ANY** và cổng phỗ biến của máy chủ là hằng số **SERV_PORT** từ tiêu đề **unp.h**.

13 Hàm **dg_echo** được gọi để thực hiện xử lý máy chủ.

8.4 Máy chủ UDP Echo: Chức năng 'dg_echo'

[Hình 8.4](#) thể hiện hàm **dg_echo**.

Hình 8.4 Hàm dg_echo: các dòng echo trên ô cắm datagram.

`lib/dg_echo.c`

```

1 #bao gồm          "unp.h"
2 khoảng trắng

3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
4 {
5     int      N;
6     socklen_t len;
7     char mesg[MAXLINE];

8     vì (      ; ; ) {
9         len = clilen;
10        n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12    }
13 }

```

Đọc datagram, phản hồi lại người gửi

8-12 Hàm này là một vòng lặp đơn giản đọc datagram tiếp theo đến cổng của máy chủ bằng **recvfrom** và gửi lại bằng **sendto**.

Mặc dù chức năng này đơn giản như ng có rất nhiều chi tiết cần xem xét. Đầu tiên, chức năng này không bao giờ chấm dứt. Vì UDP là một giao thức không kết nối nên không có gì giống như EOF như chúng ta có với TCP.

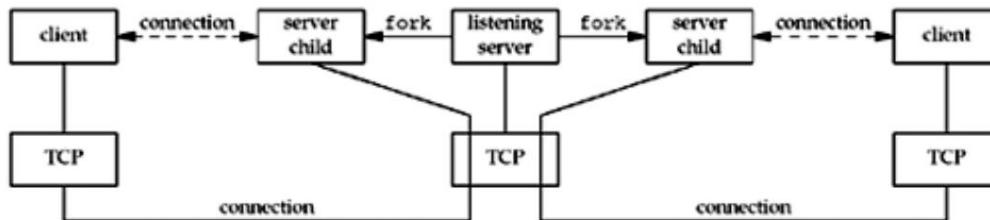
Tiếp theo, chức năng này cung cấp một máy chủ lặp chứ không phải máy chủ đồng thời như chúng ta đã có với TCP. Không có lệnh gọi `fork`, vì vậy một quy trình máy chủ duy nhất sẽ xử lý bất kỳ và tất cả các máy khách. Nói chung, hầu hết các máy chủ TCP đều hoạt động đồng thời và hầu hết các máy chủ UDP đều hoạt động đồng thời. lặp đi lặp lại.

Có ngụ ý xếp hàng diễn ra trong lớp UDP cho ổ cảm này. Thật vậy, mỗi ổ cảm UDP có một bộ đệm nhận và mỗi datagram đến ổ cảm này được đặt trong bộ đệm nhận ổ cảm đó. Khi tiến trình gọi `recvfrom`, datagram tiếp theo từ bộ đệm sẽ được trả về tiến trình theo thứ tự vào trự ớc, ra trự ớc (FIFO).

Bằng cách này, nếu có nhiều datagram đến socket trự ớc khi tiến trình có thể đọc những gì đã được xếp hàng đợi cho socket, thi các datagram đến sẽ chỉ được thêm vào bộ đệm nhận của socket. Tuy nhiên, bộ đệm này có kích thư ớc hạn chế. Chúng ta đã thảo luận về kích thư ớc này và cách tăng nó bằng tùy chọn ổ cảm `SO_RCVBUF` trong [Phần 7.5](#).

[Hình 8.5](#) tóm tắt máy khách/máy chủ TCP của chúng ta từ [Chương 5](#) khi hai máy khách thiết lập kết nối với máy chủ.

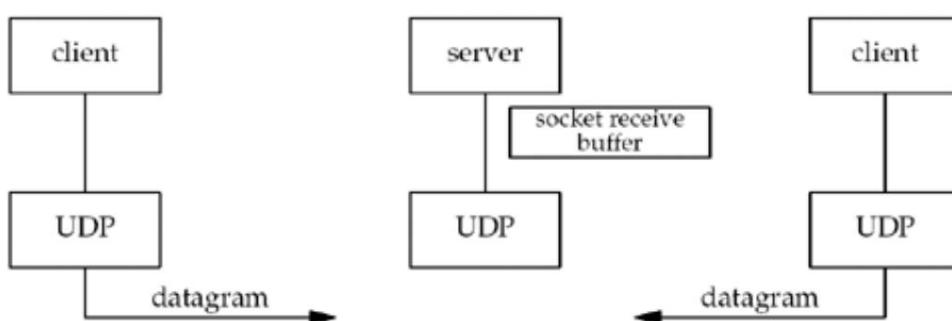
Hình 8.5. Tóm tắt TCP client/server với hai client.



Có hai ổ cảm được kết nối và mỗi ổ cảm trong số hai ổ cảm được kết nối trên máy chủ máy chủ có bộ đệm nhận ổ cảm riêng.

[Hình 8.6](#) hiển thị kích bản khi hai máy khách gửi datagram đến máy chủ UDP của chúng tôi.

Hình 8.6. Tóm tắt máy khách/máy chủ UDP với hai máy khách.



Chỉ có một quy trình máy chủ và nó có một ô cắm duy nhất để nhận tất cả các gói dữ liệu đến và gửi tất cả các phản hồi. Ở cắm đó có bộ đệm nhận để đặt tất cả các gói dữ liệu đến.

Hàm **chính** trong [Hình 8.3](#) phụ thuộc vào giao thức (nó tạo ra một ô cắm giao thức **AF_INET** và phân bổ cũng như khởi tạo cấu trúc địa chỉ ô cắm IPv4), nhưng hàm **dg_echo** không phụ thuộc vào giao thức. Lý do **dg_echo** không phụ thuộc vào giao thức là vì người gọi (hàm **chính** trong trường hợp của chúng ta) phải phân bổ cấu trúc địa chỉ ô cắm có kích thước chính xác và một con trỏ tới cấu trúc này, cùng với kích thước của nó, để được chuyển làm đối số cho **dg_echo**. Hàm **dg_echo** không bao giờ nhìn vào bên trong cấu trúc phụ thuộc vào giao thức này: Nó chỉ chuyển một con trỏ tới cấu trúc tới **recvfrom** và **sendto**. **recvfrom** điền vào cấu trúc này bằng địa chỉ IP và số cổng của máy khách, và vì cùng một con trỏ (**pcliaddr**) sau đó được chuyển đến **sendto** làm địa chỉ đích, đây là cách gói dữ liệu được phản hồi trả lại máy khách đã gửi gói dữ liệu đó.

8.5 UDP Echo Client: Chức năng 'chính'

Chức năng **chính** của máy khách UDP được hiển thị trong [Hình 8.7](#).

Hình 8.7 Máy khách tiếng vang UDP.

`udpcliserv/udpcli01.c`

```

1 #bao gồm          "unp.h"
2 số nguyên
3 chính(int argc, char **argv)
4 {
5     int      sockfd;
6     struct sockaddr_in servaddr;
7
8     nếu(argc != 2)
9         err_quit("cách sử dụng: udpcli <IPaddress>");
10
11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15
16    sockfd = Ô cắm(AF_INET, SOCK_DGRAM, 0);

```

```

14     dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));
15     thoát (0);
16 }

```

Điền vào cấu trúc địa chỉ socket bằng địa chỉ của máy chủ

9-12 Cấu trúc địa chỉ ở cắm IPv4 đư ợc điền bằng địa chỉ IP và số cổng của máy chủ. Cấu trúc này sẽ đư ợc chuyển tới `dg_cli`, chỉ định nơi gửi datagram.

13-14 Ở cắm UDP đư ợc tạo và hàm `dg_cli` đư ợc gọi.

8.6 Máy khách UDP Echo: Chức năng 'dg_cli'

Hình 8.8 cho thấy hàm `dg_cli`, hàm này thực hiện hầu hết quá trình xử lý máy khách.

Hình 8.8 Hàm `dg_cli`: vòng lặp xử lý máy khách.

lib/dg_cli.c

```

1 #bao gồm          "unp.h"
2 khoảng trắng

3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      N;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];

7     while (Fgets(sendline, MAXLINE, fp) != NULL) {

8        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr,
9         servlen);

10    n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

11    recvline[n] = 0;           /* null kết thúc */
12    Fputs(recvline, stdout);
13 }

```

7-12 Có bốn bước trong vòng xử lý máy khách: đọc một dòng từ đầu vào tiêu chuẩn bằng `fgets`, gửi dòng đó đến máy chủ bằng `sendto`, đọc lại tiếng vang của máy chủ bằng `recvfrom` và in dòng phản hồi ra đầu ra tiêu chuẩn bằng `fputs`.

Khách hàng của chúng tôi chưa yêu cầu kernel gán một cổng tạm thời cho ô cắm của nó. (Với máy khách TCP, chúng tôi đã nói lệnh gọi kết nối là nơi điều này diễn ra.) Với ô cắm UDP, lần đầu tiên quy trình gọi `sendto`, nếu ô cắm chưa có cổng cục bộ được liên kết với nó, đó là khi một cổng tạm thời được hạt nhân chọn cho ô cắm. Như với TCP, máy khách có thể gọi **liên kết** một cách rõ ràng, nhưng điều này hiếm khi được thực hiện.

Lưu ý rằng lệnh gọi `recvfrom` chỉ định một con trả null làm đối số thứ năm và thứ sáu. Điều này cho kernel biết rằng chúng ta không quan tâm đến việc biết ai đã gửi phản hồi. Có nguy cơ là bất kỳ quy trình nào, trên cùng một máy chủ hoặc một số máy chủ khác, đều có thể gửi một datagram đến địa chỉ IP và cổng của máy khách, và datagram đó sẽ được đọc bởi máy khách, người sẽ nghĩ đó là phản hồi của máy chủ. Chúng tôi sẽ giải quyết vấn đề này trong [Phần 8,8](#).

Giống như hàm máy chủ `dg_echo`, hàm máy khách `dg_cli` không phụ thuộc vào giao thức, nhưng chức năng chính của máy khách phụ thuộc vào giao thức. chính Hàm phân bổ và khởi tạo cấu trúc địa chỉ ô cắm của một số loại giao thức và sau đó chuyển một con trả tới cấu trúc này, cùng với kích thước của nó, tới `dg_cli`.

8.7 Datagram bị mất

Ví dụ máy khách/máy chủ UDP của chúng tôi không đáng tin cậy. Nếu một gói dữ liệu máy khách bị mất (giả sử nó bị bộ định tuyến nào đó giữa máy khách và máy chủ loại bỏ), máy khách sẽ chặn vĩnh viễn lệnh gọi tới `recvfrom` trong hàm `dg_cli`, chờ đợi phản hồi của máy chủ nhưng sẽ không bao giờ đến. Tương tự, nếu datagram máy khách đến máy chủ nhưng phản hồi của máy chủ bị mất, máy khách sẽ lại chặn vĩnh viễn lệnh gọi tới `recvfrom` của nó. Một cách điển hình để ngăn chặn điều này là đặt thời gian chờ cho lệnh gọi `recvfrom` của khách hàng. Chúng ta sẽ thảo luận điều này trong [Mục 14.2](#).

Chỉ đặt thời gian chờ trên `recvfrom` không phải là toàn bộ giải pháp. Ví dụ: nếu chúng tôi hết thời gian chờ, chúng tôi không thể biết liệu datagram của chúng tôi có bao giờ được gửi đến máy chủ hay không hoặc liệu phản hồi của máy chủ có bao giờ gửi lại hay không. Nếu yêu cầu của khách hàng giống như "chuyển một số tiền nhất định từ tài khoản A sang tài khoản B" (thay vì máy chủ echo đơn giản của chúng tôi), thì sẽ tạo ra sự khác biệt lớn về việc yêu cầu bị mất hay phản hồi bị mất. Chúng ta sẽ nói nhiều hơn về việc tăng độ tin cậy cho máy khách/máy chủ UDP trong [Mục 22.5](#).

8.8 Xác minh phản hồi đã nhận

Ở cuối [Phần 8.6](#), chúng tôi đã đề cập rằng bất kỳ quy trình nào biết số cổng tạm thời của máy khách đều có thể gửi các gói dữ liệu đến máy khách của chúng tôi và những gói này sẽ được trộn lẫn với các phản hồi thông thường của máy chủ. Những gì chúng ta có thể làm là thay đổi cuộc gọi [tới recvfrom](#) trong [Hình 8.8](#) để trả về địa chỉ IP và cổng của người gửi đã gửi phản hồi và bỏ qua mọi datagram đã nhận không phải từ máy chủ mà chúng ta đã gửi datagram đến. Tuy nhiên, có một số cạm bẫy với điều này, như chúng ta sẽ thấy.

Đầu tiên, chúng ta thay đổi chức năng [chính của máy khách \(Hình 8.7\)](#) để sử dụng máy chủ echo tiêu chuẩn ([Hình 2.18](#)). Chúng tôi chỉ thay thế bài tập

```
servaddr.sin_port = htons(SERV_PORT);
```

với

```
servaddr.sin_port = htons(7);
```

Chúng tôi làm điều này để có thể sử dụng bất kỳ máy chủ nào chạy máy chủ echo tiêu chuẩn với máy khách của chúng tôi.

Sau đó, chúng tôi mã hóa lại hàm [dg_cli](#) để phân bổ một cấu trúc địa chỉ ở cẩm khác nhằm giữ cấu trúc được trả về bởi [recvfrom](#). Chúng tôi thể hiện điều này trong [Hình 8.9](#).

Phân bổ cấu trúc địa chỉ ở cẩm khác

[9](#) Chúng tôi phân bổ một cấu trúc địa chỉ socket khác bằng cách gọi [malloc](#). Lưu ý rằng hàm [dg_cli](#) vẫn độc lập với giao thức; bởi vì chúng tôi không quan tâm đến loại cấu trúc địa chỉ ở cẩm mà chúng tôi đang xử lý, chúng tôi chỉ sử dụng kích thước của nó trong lệnh gọi tới [malloc](#).

So sánh địa chỉ trả về

[12-18](#) Trong lệnh gọi [recvfrom](#), chúng ta yêu cầu kernel trả về địa chỉ của người gửi datagram. Trước tiên, chúng tôi so sánh độ dài được [recvfrom](#) trả về trong đối số kết quả giá trị, sau đó so sánh chính cấu trúc địa chỉ socket bằng [memcmp](#).

Phản 3.2 nói rằng ngay cả khi cấu trúc địa chỉ ở cắm có chứa trường độ dài, chúng ta không bao giờ cần thiết lập hoặc kiểm tra nó. Tuy nhiên, `memcmp` so sánh từng byte dữ liệu trong hai cấu trúc địa chỉ socket và trường độ dài được đặt trong cấu trúc địa chỉ socket mà kernel trả về; vì vậy trong trường hợp này chúng ta phải thiết lập nó khi xây dựng `sockaddr`. Nếu không, `memcmp` sẽ so sánh số 0 (vì chúng tôi không đặt số này) với số 16 (giả sử `sockaddr_in`) và sẽ không khớp.

Hình 8.9 Phiên bản của `dg_cli` xác minh địa chỉ socket được trả về.

udpcliserv/dgcliaddr.c

```

1 #bao gồm          "unp.h"

2 khoảng trắng

3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      N;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];
7     socklen_t len;
8     struct sockaddr *preply_addr;

9     preply_addr = Malloc(servlen);

10    while (Fgets(sendline, MAXLINE, fp) != NULL) {

11        Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr,
servlen);

12        len = servlen;
13        n = Recvfrom(sockfd, recvline, MAXLINE, 0, preply_addr, &len);
14        if (len != servlen || memcmp(pservaddr, preply_addr, len) != 0)
{
15            printf("trả lời từ %s (bỏ qua)\n", Sock_ntop(preply_addr,
chỉ một));
16            Tiếp tục;
17        }

18        recvline[n] = 0;           /* null kết thúc */
19        Fputs(recvline, stdout);
20    }
21 }
```

Phiên bản mới này của ứng dụng khách của chúng tôi hoạt động tốt nếu máy chủ nằm trên một máy chủ chỉ có một địa chỉ IP duy nhất. Nhưng chương trình này có thể thất bại nếu máy chủ có nhiều máy chủ. Chúng tôi chạy chương trình này trên máy chủ **freebsd4**, có hai giao diện và hai địa chỉ IP.

```
macosx % máy chủ freebsd4
freebsd4.unpbook.com có địa chỉ 172.24.37.94
freebsd4.unpbook.com có địa chỉ 135.197.17.100
macosx% udpcli02 135.197.17.100
Xin chào
trả lời từ 172.24.37.94:7 (bỏ qua)
tạm biệt
trả lời từ 172.24.37.94:7 (bỏ qua)
```

Chúng tôi đã chỉ định địa chỉ IP không chia sẻ cùng mạng con với máy khách.

Điều này thường được cho phép. Hầu hết các triển khai IP đều chấp nhận gói dữ liệu IP đến được dành cho bất kỳ địa chỉ IP nào của máy chủ, bất kể giao diện mà gói dữ liệu đến (trang 217-219 của TCPv2). RFC 1122 [Braden 1989] gọi đây là mô hình hệ thống đầu cuối yếu. Nếu một hệ thống triển khai cái mà RFC này gọi là mô hình hệ thống đầu cuối mạnh, nó sẽ chỉ chấp nhận một gói dữ liệu đến nếu gói dữ liệu đó đến trên giao diện mà nó được đánh địa chỉ.

Địa chỉ IP được trả về bởi **recvfrom** (địa chỉ IP nguồn của gói dữ liệu UDP) không phải là địa chỉ IP mà chúng tôi đã gửi gói dữ liệu. Khi máy chủ gửi phản hồi, địa chỉ IP đích là 172.24.37.78. Chức năng định tuyến trong kernel trên **freebsd4** chọn 172.24.37.94 làm giao diện gửi đi. Vì máy chủ chưa liên kết địa chỉ IP với ổ cắm của nó (máy chủ đã liên kết địa chỉ ký tự đại diện với ổ cắm của nó, đây là điều mà chúng tôi có thể xác minh bằng cách chạy **netstat** trên **freebsd**), hệ nhân chọn địa chỉ nguồn cho gói dữ liệu IP. Nó được chọn làm địa chỉ IP chính của giao diện gửi đi (trang 232-233 của TCPv2). Ngoài ra, vì nó là địa chỉ IP chính của giao diện, nếu chúng ta gửi datagram của mình đến một địa chỉ IP không chính của giao diện (tức là bí danh), điều này cũng sẽ khiến thử nghiệm của chúng ta trong [Hình 8.9](#) thất bại.

Một giải pháp là máy khách xác minh tên miền của máy chủ phản hồi thay vì địa chỉ IP của nó bằng cách tra cứu tên máy chủ trong DNS ([Chương 11](#)), dựa trên địa chỉ IP được trả về bởi **recvfrom**. Một giải pháp khác là máy chủ UDP tạo một ổ cắm cho mỗi địa chỉ IP được định cấu hình trên máy chủ, **liên kết** địa chỉ IP đó với ổ cắm, sử dụng **chọn** trên tất cả các ổ cắm này (chờ bất kỳ địa chỉ nào có thể đọc được), sau đó trả lời từ socket có thể đọc được. Vì socket được sử dụng để trả lời được liên kết với địa chỉ IP là địa chỉ đích của yêu cầu của khách hàng (hoặc datagram sẽ không được gửi đến socket), điều này đảm bảo rằng

địa chỉ nguồn của phản hồi giống với địa chỉ đích của yêu cầu. Chúng ta sẽ trình bày một ví dụ về điều này trong [Phản 22.6](#).

Trên hệ thống Solaris multihomed, địa chỉ IP nguồn cho phản hồi của máy chủ là địa chỉ IP đích trong yêu cầu của khách hàng. Kịch bản dưới đây mô tả trong phần này dành cho các triển khai có nguồn gốc từ Berkeley chọn địa chỉ IP nguồn dựa trên giao diện gửi đi.

Máy chủ 8.9 không chạy

Tình huống tiếp theo cần kiểm tra là khởi động máy khách mà không khởi động máy chủ. Nếu chúng ta làm như vậy và gõ một dòng duy nhất cho máy khách thì sẽ không có gì xảy ra. Máy khách chặn vĩnh viễn lệnh gọi tới `recvfrom`, chờ phản hồi của máy chủ nhưng sẽ không bao giờ xuất hiện. Tuy nhiên, đây là một ví dụ mà chúng ta cần hiểu thêm về các giao thức cơ bản để hiểu điều gì đang xảy ra với ứng dụng mạng của mình.

Trước tiên, chúng tôi khởi động `tcpdump` trên máy chủ `macosx`, sau đó chúng tôi khởi động máy khách trên cùng một máy chủ, chỉ định máy chủ `freebsd4` làm máy chủ lưu trữ. Sau đó chúng ta gõ một dòng duy nhất nhưng dòng đó không bị lặp lại.

```
macosx% udpcli01 172.24.37.94
```

Chào thế giới

chúng tôi gõ dòng này nhưng không có gì được phản hồi lại

[Hình 8.10 hiển thị](#) đầu ra `tcpdump`.

Hình 8.10 đầu ra `tcpdump` khi quá trình máy chủ không được khởi động trên máy chủ.

```
1 0,0      arp ai có freebsd4 nói với macosx
2 0,003576 ( 0,0036)      arp trả lời freebsd4 là lúc 0:40:5:42:d6:de

3 0,003601 ( 0,0000)      macosx.51139 > freebsd4.9877: udp 13
4 0,009781 ( 0,0062)      freebsd4 > macosx: icmp: freebsd4 công udp 9877
không thể truy cập được
```

Trước tiên, chúng tôi nhận thấy rằng cần có yêu cầu và trả lời ARP trước khi máy khách có thể gửi gói dữ liệu UDP đến máy chủ. (Chúng tôi để lại trao đổi này ở đầu ra để nhắc lại khả năng trả lời yêu cầu ARP trước khi gói dữ liệu IP có thể được gửi đến máy chủ hoặc bộ định tuyến khác trên mạng cục bộ.)

Ở dòng 3, chúng ta thấy gói dữ liệu máy khách đã được gửi như ng máy chủ lưu trữ phản hồi ở dòng 4 với thông báo "port unreachable" ICMP. (Độ dài 13 chiếm 12 ký tự và dòng mới.) Tuy nhiên, lỗi ICMP này không được trả về quy trình máy khách vì những lý do mà chúng tôi sẽ mô tả sau. Thay vào đó, máy khách sẽ chặn vĩnh viễn lệnh gọi tới `recvfrom`

trong [Hình 8.8](#). Chúng tôi cũng lưu ý rằng ICMPv6 có lỗi "port unreachable", tương tự như ICMPv4 ([Hình A.15](#) và [A.16](#)), do đó kết quả được mô tả ở đây cũng tương tự đối với IPv6.

Chúng tôi gọi lỗi ICMP này là lỗi không đồng bộ. Lỗi do `sendto` gây ra như ng `sendto` đã trả về thành công. Hãy nhớ lại [Phản 2.11](#) rằng việc hoàn trả thành công từ hoạt động đầu ra UDP chỉ có nghĩa là có chỗ cho gói dữ liệu IP kết quả trên hàng đợi đầu ra giao diện. Lỗi ICMP không được trả về cho đến sau này (4 ms sau trong [Hình 8.10](#)), đó là lý do tại sao nó được gọi là không đồng bộ.

Nguyên tắc cơ bản là lỗi không đồng bộ không được trả về cho ô cắm UDP trừ khi ô cắm đã được kết nối. Chúng tôi sẽ mô tả cách gọi kết nối cho ô cắm UDP trong [Mục 8.11](#). Tại sao quyết định thiết kế này được đưa ra khi các socket được triển khai lần đầu tiên vẫn chưa được hiểu rõ. (Ý nghĩa triển khai được thảo luận ở trang 748-749 của TCPv2.)

Hãy xem xét một máy khách UDP gửi ba gói dữ liệu liên tiếp đến ba máy chủ khác nhau (tức là ba địa chỉ IP khác nhau) trên một ô cắm UDP. Sau đó, máy khách sẽ vào vòng lặp gọi `recvfrom` để đọc các câu trả lời. Hai trong số các gói dữ liệu được phân phối chính xác (nghĩa là máy chủ đang chạy trên hai trong số ba máy chủ) nhưng máy chủ thứ ba không chạy máy chủ. Máy chủ thứ ba này phản hồi với cồng ICMP không thể truy cập được.

Thông báo lỗi ICMP này chứa tiêu đề IP và tiêu đề UDP của datagram gây ra lỗi. (Thông báo lỗi ICMPv4 và ICMPv6 luôn chứa tiêu đề IP và tất cả tiêu đề UDP hoặc một phần của tiêu đề TCP để cho phép người nhận lỗi ICMP xác định ô cắm nào gây ra lỗi. Chúng tôi sẽ hiển thị điều này trong [Hình 28.21](#)

và [28.22](#).) Máy khách đã gửi ba datagram cần biết đích đến của datagram gây ra lỗi để phân biệt datagram nào trong ba datagram gây ra lỗi. Nhưng làm thế nào kernel có thể trả lại thông tin này cho tiến trình? Phản thông tin duy nhất mà `recvfrom` có thể trả về là một giá trị sai sót ; `recvfrom` không có cách nào trả lại địa chỉ IP đích và số cổng UDP đích của datagram bị lỗi. Do đó, quyết định đã được đưa ra là chỉ trả lại các lỗi không đồng bộ này cho quy trình nếu quy trình đó kết nối ô cắm UDP với chính xác một thiết bị ngang hàng.

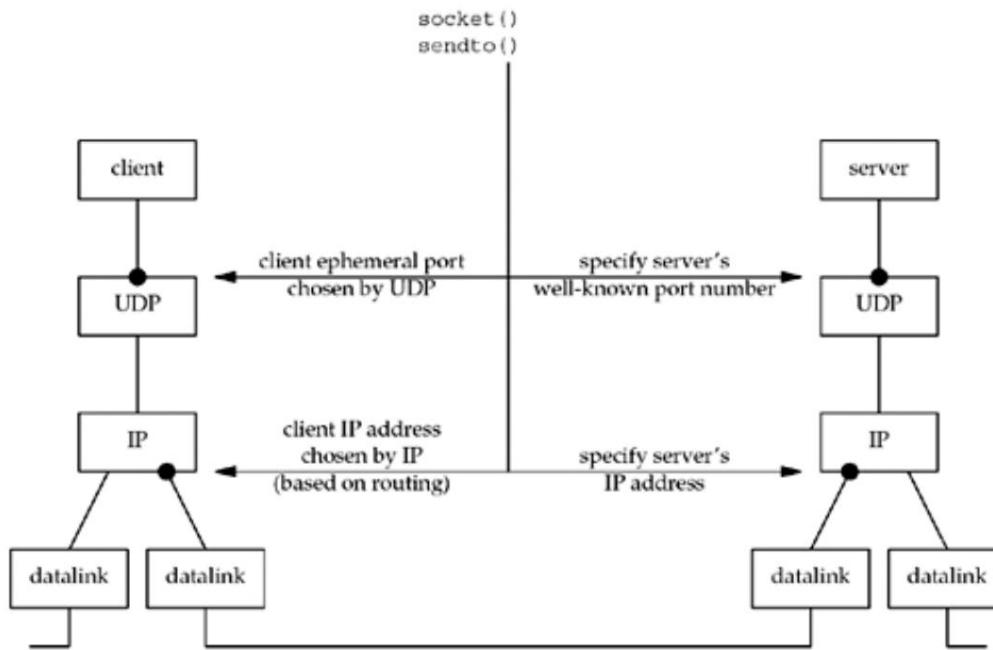
Linux trả về hầu hết các lỗi ICMP "không thể truy cập đích" ngay cả khi chưa kết nối ô cắm, miễn là tùy chọn ô cắm `SO_BSDCOMPAT` không được bật. Tất cả các lỗi ICMP "không thể truy cập đích" từ [Hình A.15](#) đều được trả về, ngoại trừ các mã 0, 1, 4, 5, 11 và 12.

Chúng ta quay lại vấn đề lỗi không đồng bộ với socket UDP ở [Mục 28.7](#) và chỉ ra cách dễ dàng để khắc phục những lỗi này trên các ô cắm không được kết nối bằng cách sử dụng trình nền của riêng chúng tôi.

8.10 Tóm tắt ví dụ UDP

[Hình 8.11](#) hiển thị dữ ới dạng dấu đầu dòng bốn giá trị phải đư ợc chỉ định hoặc chọn khi máy khách gửi gói dữ liệu UDP.

Hình 8.11. Tóm tắt về máy khách/máy chủ UDP từ góc nhìn của máy khách.



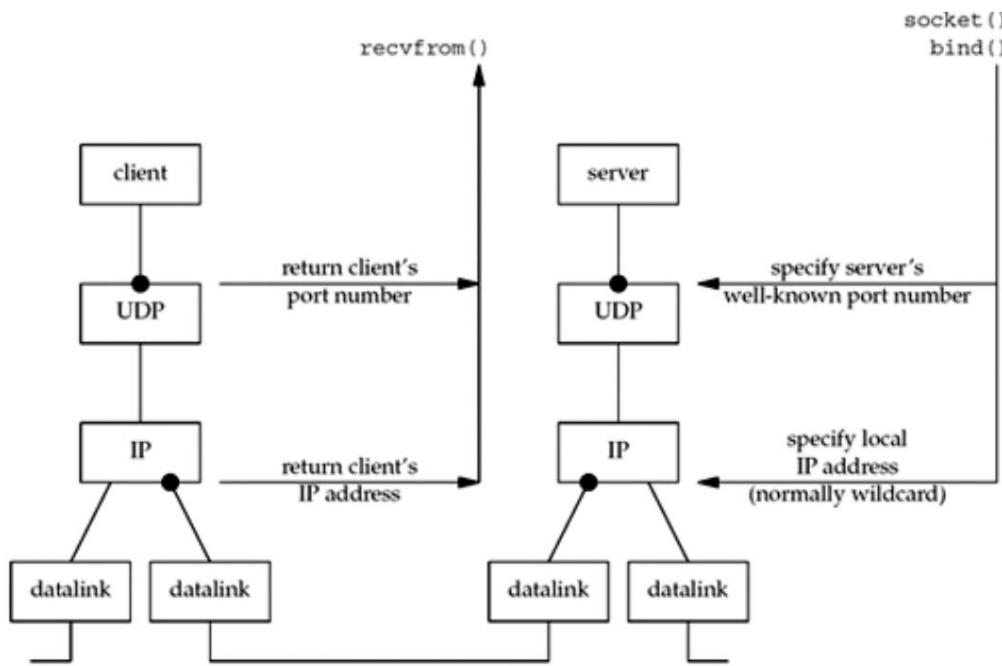
Máy khách phải chỉ định địa chỉ IP và số cổng của máy chủ cho cuộc gọi tới `sendto`.

Thông thường, địa chỉ IP và cổng của máy khách đư ợc kernel chọn tự động, mặc dù chúng tôi đã đề cập rằng máy khách có thể gọi **liên kết** nếu nó chọn như vậy. Nếu hai giá trị này cho máy khách đư ợc hạt nhân chọn thì chúng tôi cũng đã đề cập rằng cổng tạm thời của máy khách đư ợc chọn một lần, trong lần **gửi** đầu tiên và sau đó nó không bao giờ thay đổi. Tuy nhiên, địa chỉ IP của máy khách có thể thay đổi đối với mỗi gói dữ liệu UDP mà máy khách gửi, giả sử máy khách không **liên kết** một địa chỉ IP cụ thể với ổ cắm. Lý do đư ợc hiển thị trong [Hình 8.11](#): Nếu máy khách là multihomed, máy khách có thể luôn **phiên giữa hai** dịch, một di ra ngoài liên kết dữ liệu ở bên trái và một di ra liên kết dữ liệu ở bên phải. Trong trường hợp xấu nhất này, địa chỉ IP của máy khách, do kernel chọn dựa trên liên kết dữ liệu gửi đi, sẽ thay đổi đối với mỗi datagram.

Điều gì xảy ra nếu máy khách **liên kết** một địa chỉ IP với socket của nó, nhưng nhân quyết định rằng một datagram gửi đi phải đư ợc gửi đi một số liên kết dữ liệu khác? Trong trường hợp này, datagram IP sẽ chứa địa chỉ IP nguồn khác với địa chỉ IP của liên kết dữ liệu đi (xem [Bài tập 8.6](#)).

[Hình 8.12](#) hiển thị bốn giá trị giống nhau, như ng từ góc nhìn của máy chủ.

Hình 8.12. Tóm tắt về máy khách/máy chủ UDP từ góc nhìn của máy chủ.



Có ít nhất bốn thông tin mà máy chủ có thể muốn biết từ gói dữ liệu IP đến: địa chỉ IP nguồn, địa chỉ IP đích, số cổng nguồn và số cổng đích. Hình 8.13 hiển thị các lệnh gọi hàm trả về thông tin này cho máy chủ TCP và máy chủ UDP.

Hình 8.13. Thông tin có sẵn cho máy chủ từ datagram IP đến.

From client's IP datagram	TCP server	UDP server
Source IP address	accept	recvfrom
Source port number	accept	recvfrom
Destination IP address	getsockname	recvmsg
Destination port number	getsockname	getsockname

Máy chủ TCP luôn có quyền truy cập dễ dàng vào tất cả bốn thông tin của ô cảm được kết nối và bốn giá trị này không đổi trong suốt thời gian kết nối. Tuy nhiên, với ô cảm UDP, chỉ có thể lấy địa chỉ IP đích bằng cách đặt tùy chọn ô cảm `IP_RECVSTADDR` cho IPv4 hoặc tùy chọn ô cảm `IPV6_PKTINFO` cho IPv6, sau đó gọi `recvmsg` thay vì `recvfrom`. Vì UDP không có kết nối nên địa chỉ IP đích có thể thay đổi đối với mỗi datagram được gửi đến máy chủ. Máy chủ UDP cũng có thể nhận các gói dữ liệu dành cho một trong các địa chỉ quảng bá của máy chủ hoặc cho một địa chỉ multicast, như chúng ta sẽ thảo luận trong [Chương 20](#) và [21](#). Chúng tôi sẽ trình bày cách xác định địa chỉ đích của gói dữ liệu UDP trong [Phần 22.2](#), sau khi chúng tôi trình bày cách xác định địa chỉ đích của gói dữ liệu UDP trong [Phần 22.2](#), bao gồm chức năng `recvmsg`.

8.11 Chức năng 'kết nối' với UDP

Chúng tôi đã đề cập ở cuối [Phần 8.9](#) rằng lỗi không đồng bộ không được trả về trên ô cắm UDP trừ khi ô cắm đó đã được kết nối. Thật vậy, chúng ta có thể gọi [connect](#) ([Phần 4.3](#)) cho ô cắm UDP. Nhưng điều này không dẫn đến kết quả gì giống như [kết nối TCP: Không](#) có sự bắt tay ba chiều. Thay vào đó, kernel chỉ kiểm tra bất kỳ lỗi tức thời nào (ví dụ: đích rõ ràng là không thể truy cập được), ghi lại địa chỉ IP và số cổng của thiết bị ngang hàng (từ cấu trúc địa chỉ ô cắm được truyền để [kết nối](#)) và quay lại ngay quy trình gọi.

Việc quá tải chức năng [kết nối](#) bằng khả năng này cho các ô cắm UDP là điều khó hiểu. Nếu quy ước `sockname` là địa chỉ giao thức cục bộ và [ngang hàng](#) là địa chỉ giao thức nằm ngoài được sử dụng thì tên tốt hơn sẽ là `setpeername`.

Tương tự, tên hay hơn cho hàm [liên kết](#) sẽ là `setsockname`.

Với khả năng này, bây giờ chúng ta phải phân biệt giữa

- Ô cắm UDP chưa được kết nối, mặc định khi chúng ta tạo ô cắm UDP
- Ô cắm UDP được kết nối, kết quả của cuộc gọi [kết nối](#) trên ô cắm UDP

Với ô cắm UDP được kết nối, có ba điều thay đổi so với ô cắm UDP không được kết nối mặc định:

1. Chúng tôi không còn có thể chỉ định địa chỉ IP đích và cổng cho đầu ra hoạt động. Tức là chúng ta không sử dụng `sendto` mà thay vào đó [viết](#) hoặc [gửi](#). Mọi thứ được ghi vào ô cắm UDP được kết nối sẽ tự động được gửi đến địa chỉ giao thức (ví dụ: địa chỉ IP và cổng) được chỉ định bởi [kết nối](#).

Tương tự như TCP, chúng ta có thể gọi `sendto` cho ô cắm UDP được kết nối, nhưng chúng ta không thể chỉ định địa chỉ đích. Đổi số thứ năm của `sendto` (con trỏ tới cấu trúc địa chỉ ô cắm) phải là con trỏ rỗng và đổi số thứ sáu (kích thước của cấu trúc địa chỉ ô cắm) phải là 0. Đặc tả POSIX nêu rõ rằng khi đổi số thứ năm là con trỏ rỗng, đổi số thứ sáu bị bỏ qua.

2. Chúng ta không cần sử dụng `recvfrom` để tìm hiểu người gửi datagram mà thay vào đó [hãy đọc](#), `recv` hoặc `recvmsg`.

Các gói dữ liệu duy nhất được hạt nhân trả về cho thao tác đầu vào trên ô cắm UDP được kết nối là những gói đến từ địa chỉ giao thức được chỉ định trong `connect`. Các gói dữ liệu được gửi đến địa chỉ giao thức cục bộ của ô cắm UDP được kết nối (ví dụ: địa chỉ IP và cổng) nhưng đến từ một địa chỉ giao thức khác với địa chỉ mà ô cắm được kết nối sẽ không được chuyển đến ô cắm được kết nối. Điều này giới hạn ô cắm UDP được kết nối trong việc trao đổi gói dữ liệu với một và chỉ một thiết bị ngang hàng.

Về mặt kỹ thuật, ô cắm UDP được kết nối sẽ trao đổi các gói dữ liệu chỉ với một địa chỉ IP, vì có thể kết nối với địa chỉ multicast hoặc quảng bá.

3. Các lỗi không đồng bộ được trả về quy trình dành cho các ô cắm UDP được kết nối.

Hệ quả tất yếu, như chúng tôi đã mô tả trước đây, là các ô cắm UDP không được kết nối sẽ không nhận được lỗi không đồng bộ.

[Hình 8.14](#) tóm tắt điểm đầu tiên trong danh sách liên quan đến 4.4BSD.

Hình 8.14. Ô cắm TCP và UDP: địa chỉ giao thức đích có thể được chỉ định không?

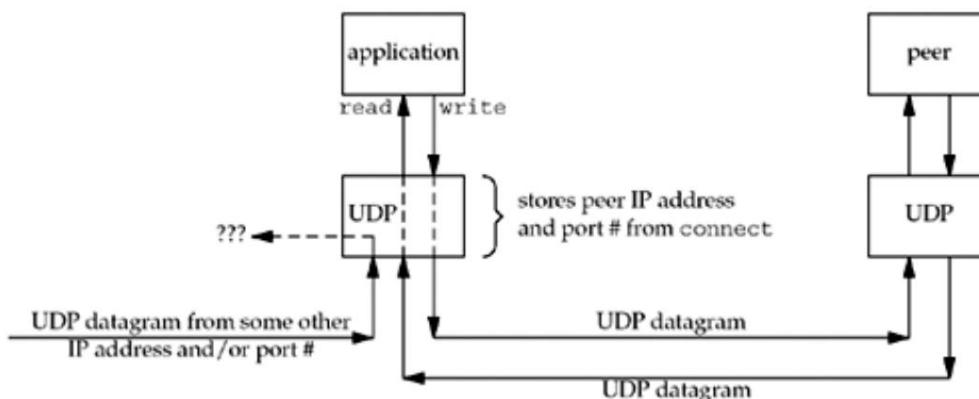
Type of socket	write or send	sendto that does not specify a destination	sendto that specifies a destination
TCP socket	OK	OK	EISCONN
UDP socket, connected	OK	OK	EISCONN
UDP socket, unconnected	EDESTADDRREQ	EDESTADDRREQ	OK

Đặc tả POSIX nêu rõ rằng thao tác đầu ra không chỉ định địa chỉ đích trên ô cắm UDP chưa được kết nối sẽ trả về **ENOTCONN**, không phải

ĐỊA CHỈ TIỀN ĐỘ.

[Hình 8.15](#) tóm tắt ba điểm mà chúng tôi đã đưa ra về ô cắm UDP được kết nối.

Hình 8.15. Ô cắm UDP được kết nối.



Các cuộc gọi ứng dụng **kết nối**, chỉ định địa chỉ IP và số cổng ngang hàng của nó.

Sau đó nó sử dụng chức năng **đọc** và **ghi** để trao đổi dữ liệu với thiết bị ngang hàng.

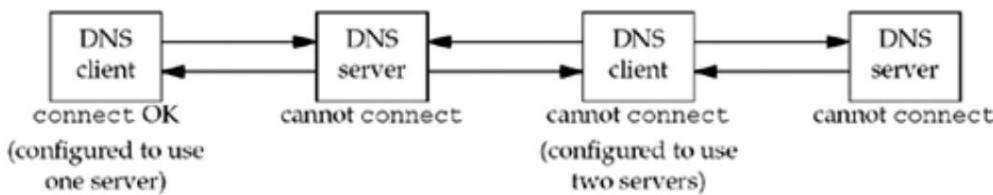
Các gói dữ liệu đến từ bất kỳ địa chỉ IP hoặc cổng nào khác (mà chúng tôi hiển thị là "???" trong [Hình 8.15](#)) không **được chuyển đến** ô cắm được kết nối vì địa chỉ IP nguồn hoặc cổng UDP nguồn không khớp với địa chỉ giao thức mà ô cắm đó kết nối. đã **kết nối**. Các datagram này có thể được gửi đến một số ô cắm UDP khác trên

chủ nhà. Nếu không có ô cắm phù hợp nào khác cho gói dữ liệu đến, UDP sẽ loại bỏ nó và tạo ra lỗi ICMP "không thể truy cập cổng".

Tóm lại, chúng ta có thể nói rằng máy khách hoặc máy chủ UDP chỉ có thể gọi **kết nối** nếu quá trình đó sử dụng ô cắm UDP để giao tiếp với chính xác một thiết bị ngang hàng. Thông thường, máy khách UDP gọi **kết nối**, nhưng có những ứng dụng trong đó máy chủ UDP giao tiếp với một máy khách trong thời gian dài (ví dụ: TFTP); trong trường hợp này, cả máy khách và máy chủ đều có thể gọi **connect**.

DNS cung cấp một ví dụ khác, như trong [Hình 8.16](#).

Hình 8.16. Ví dụ về máy khách và máy chủ DNS và chức năng kết nối.



Máy khách DNS có thể được cấu hình để sử dụng một hoặc nhiều máy chủ, thông thường bằng cách liệt kê địa chỉ IP của máy chủ trong tệp `/etc/resolv.conf`. Nếu một máy chủ được liệt kê (ở ngoài cùng bên trái trong hình), máy khách có thể gọi **connect**, nhưng nếu nhiều máy chủ được liệt kê (ở thứ hai từ bên phải trong hình), máy khách không thể gọi **connect**.

Ngoài ra, máy chủ DNS thường xử lý mọi yêu cầu của khách hàng, do đó máy chủ không thể gọi **kết nối**.

Gọi kết nối nhiều lần cho ô cắm UDP

Một tiến trình có ô cắm UDP được kết nối có thể gọi lại **kết nối** cho ô cắm đó vì một trong hai lý do:

- Để chỉ định địa chỉ IP và cổng mới
- Để ngắt kết nối ô cắm

Trường hợp đầu tiên, chỉ định một thiết bị ngang hàng mới cho ô cắm UDP được kết nối, khác với việc sử dụng **kết nối** với ô cắm TCP: **kết nối** chỉ có thể được gọi một lần cho ô cắm TCP.

Để ngắt kết nối ô cắm UDP, chúng tôi gọi **kết nối** nhưng đặt thành viên gia đình của cấu trúc địa chỉ ô cắm (`sin_family` cho IPv4 hoặc `sin6_family` cho IPv6) thành `AF_UNSPEC`. Điều này có thể trả về lỗi `EAFNOSUPPORT` (tr. 736 của TCPv2), nhưng điều đó có thể chấp nhận được. Chính quá trình gọi **kết nối** trên ô cắm UDP đã được kết nối sẽ khiến ô cắm không được kết nối (trang 787-788 của TCPv2).

Các biến thể Unix thường khác nhau về cách ngắt kết nối ô cắm một cách chính xác và bạn có thể gặp các phuơng pháp hoạt động trên một số hệ thống chứ không phải trên các hệ thống khác. Ví dụ: gọi **kết nối** bằng `NULL` cho địa chỉ chỉ hoạt động trên một số hệ thống (và trên một số,

nó chỉ hoạt động nếu đối số thứ ba, độ dài, khác 0). Đặc tả POSIX và các trang man BSD không giúp ích nhiều ở đây, chỉ đề cập rằng địa chỉ null

nen đư ợc sử dụng và không đề cập đến việc trả lại lỗi (ngay cả khi thành công). Giải pháp di động nhất là loại bỏ cấu trúc địa chỉ, đặt họ thành AF_UNSPEC như đã đề cập ở trên và chuyển nó để **kết nối**.

Một lĩnh vực bất đồng khác là xung quanh việc ràng buộc cục bộ của ô cắm trong quá trình ngắt kết nối. AIX giữ cả địa chỉ IP cục bộ đã chọn và cảng, ngay cả từ một liên kết ngầm. FreeBSD và Linux đặt địa chỉ IP cục bộ trở về tất cả số 0, ngay cả khi trước đó bạn đã gọi **liên kết** như vẫn giữ nguyên số cảng. Solaris đặt địa chỉ IP cục bộ trở về tất cả số 0 nếu nó đư ợc gán bởi một liên kết ngầm; như ng nếu chương trình đư ợc gọi là **liên kết** một cách rõ ràng thì địa chỉ IP vẫn không thay đổi.

Hiệu suất

Khi một ứng dụng gọi **sendto** trên ô cắm UDP chưa đư ợc kết nối, các hạt nhân có nguồn gốc từ Berkeley sẽ tạm thời kết nối ô cắm, gửi datagram và sau đó ngắt kết nối ô cắm (trang 762-763 của TCPv2). Việc gọi **sendto** cho hai datagram trên ô cắm UDP chưa đư ợc kết nối sau đó bao gồm sáu bước sau của kernel:

- Kết nối ô cắm
- Xuất datagram đầu tiên
- Ngắt kết nối ô cắm
- Kết nối ô cắm
- Xuất datagram thứ hai
- Ngắt kết nối ô cắm

Một vấn đề cần cân nhắc khác là số lượng tìm kiếm trong bảng định tuyến. Kết nối tạm thời đầu tiên tìm kiếm bảng định tuyến để tìm địa chỉ IP đích và lưu (lưu trữ) thông tin đó. Kết nối tạm thời thứ hai thông báo rằng địa chỉ đích bằng với đích của thông tin bảng định tuyến đư ợc lưu trong bộ nhớ đệm (chúng tôi giả sử có hai **sendto** đến cùng một đích) và chúng tôi không cần tìm kiếm lại bảng định tuyến (trang 737-738 của TCPv2).

Khi ứng dụng biết nó sẽ gửi nhiều datagram đến cùng một thiết bị ngang hàng, việc kết nối ô cắm một cách rõ ràng sẽ hiệu quả hơn. Gọi **kết nối** và sau đó gọi **viết** hai lần bao gồm các bước sau của kernel:

- Kết nối ô cắm
- Xuất datagram đầu tiên
- Xuất datagram thứ hai

Trong trường hợp này, kernel chỉ sao chép cấu trúc địa chỉ socket chứa địa chỉ IP đích và cảng một lần, so với hai lần khi **sendto** đư ợc gọi hai lần. [Partridge và Pink 1993] lưu ý rằng việc kết nối tạm thời của một

socket UDP không được kết nối chiếm gần 1/3 giá thành của mỗi UDP quá trình lây truyền.

8.12 Hàm 'dg_cli' (Đã xem lại)

Bây giờ chúng ta quay lại hàm `dg_cli` từ [Hình 8.8](#) và mã hóa lại nó để gọi `connect`.

[Hình 8.17](#) thể hiện chức năng mới.

Hình 8.17 hàm `dg_cli` gọi `connect`.

`udpcliserv/dgcliconnect.c`

```

1 #bao gồm          "unp.h"

2 khoảng trắng

3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
4 {
5     int      N;
6     char sendline[MAXLINE], recvline[MAXLINE + 1];

7     Connect(sockfd, (SA *) pservaddr, servlen);

8     while (Fgets(sendline, MAXLINE, fp) != NULL) {

9         Write(sockfd, sendline, strlen(sendline));

10        n = Đọc(sockfd, recvline, MAXLINE);

11        recvline[n] = 0;           /* null kết thúc */
12        Fputs(recvline, stdout);
13    }
14 }
```

Những thay đổi này là lệnh gọi mới để `kết nối` và thay thế lệnh gọi tới `sendto` và `recvfrom` bằng lệnh gọi `viết` và `đọc`. Hàm này vẫn đọc lập với giao thức vì nó không nhìn vào bên trong cấu trúc địa chỉ socket được truyền để `kết nối`.

Chức năng `chính` của máy khách của [chúng tôi](#), [Hình 8.7](#), vẫn giữ nguyên.

Nếu chúng tôi chạy chương trình này trên máy chủ `macosx`, chỉ định địa chỉ IP của máy chủ `freebsd4` (không chạy máy chủ của chúng tôi trên cổng 9877), chúng tôi có kết quả đầu ra sau:

```
macosx% udpcli04 172.24.37.94
Chào thế giới
lỗi đọc: Kết nối bị từ chối
```

Điểm đầu tiên chúng tôi nhận thấy là chúng tôi không nhận được lỗi khi bắt đầu quá trình máy khách. Lỗi chỉ xảy ra sau khi chúng tôi gửi datagram đầu tiên đến máy chủ. Việc gửi datagram này sẽ gây ra lỗi ICMP từ máy chủ. Nhưng khi máy khách TCP gọi **kết nối**, chỉ định máy chủ lưu trữ không chạy quy trình máy chủ, **kết nối** sẽ trả về lỗi vì lệnh gọi kết nối khiến quá trình bắt tay ba chiều TCP xảy ra và gói đầu tiên của quá trình bắt tay đó sẽ tạo ra RST từ máy chủ TCP ([Phần 4.3](#)).

[Hình 8.18 cho thấy](#) đầu ra **tcpdump**.

Hình 8.18 đầu ra **tcpdump** khi chạy Hình 8.17.

```
macosx% tcpdump
1 0,0          macosx.51139 > freebsd4.9877: udp 13
2 0,006180 ( 0,0062)      freebsd4 > macosx: icmp: freebsd4 công udp 9877
không thể truy cập đư ợc
```

Chúng ta cũng thấy [trong Hình A.15](#) rằng lỗi ICMP này được kernel ánh xạ vào lỗi **ECONNREFUSED**, tương ứng với đầu ra chuỗi thông báo bởi **err_sys** của chúng ta

chức năng: "Kết nối bị từ chối."

Thật không may, không phải tất cả các hạt nhân đều trả về các thông báo ICMP tới ô cảm UDP được kết nối, như chúng tôi đã trình bày trong phần này. Thông thường, các hạt nhân có nguồn gốc từ Berkeley trả về lỗi, trong khi các hạt nhân System V thì không. Ví dụ: nếu chúng tôi chạy cùng một máy khách trên máy chủ Solaris 2.4 và **kết nối** với máy chủ không chạy máy chủ của chúng tôi, chúng tôi có thể xem bằng **tcpdump** và xác minh rằng lỗi "port unreachable" ICMP được máy chủ trả về nhưng lệnh gọi **đọc** của máy khách không bao giờ quay trở lại. Lỗi này đã được sửa trong Solaris 2.5. UnixWare không trả về lỗi, trong khi AIX, Digital Unix, HP-UX và Linux đều trả về lỗi.

8.13 Thiếu kiểm soát luồng với UDP

Bây giờ chúng ta kiểm tra tác động của việc UDP không có bất kỳ điều khiển luồng nào. Đầu tiên, chúng tôi sửa đổi hàm **dg_cli** để gửi một số lượng datagram cố định. Nó không còn đọc từ đầu vào tiêu chuẩn nữa. [Hình 8.19](#) hiển thị phiên bản mới. Chức năng **này ghi 2.000** gói dữ liệu UDP 1.400 byte vào máy chủ.

Tiếp theo chúng tôi sửa đổi máy chủ để nhận datagram và đếm số lượng nhận được. Máy chủ này không còn gửi lại datagram cho máy khách nữa. Hình 8.20 thể hiện hàm dg_echo mới. Khi chúng tôi chấm dứt máy chủ bằng khóa ngắt đầu cuối ([SIGINT](#)), nó sẽ in số lượng datagram đã nhận và chấm dứt.

Hình 8.19 Hàm dg_cli ghi một số datagram cố định vào máy chủ.

udpcliserv/dgcliloop1.c

```

1 #bao gồm          "unp.h"

2 #định nghĩa NDG      2000           /* datagram cần gửi */
3 #xác định DGLEN 1400           /* độ dài của mỗi datagram */

4 khoảng trắng

5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
6 {
7     int      Tôi;
8     dòng gửi char[DGLEN];

9     vì (i = 0; i < NDG; i++) {
10        Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11    }
12 }
```

Hình 8.20 hàm dg_echo đếm các datagram đã nhận.

udpcliserv/dgecholoop1.c

```

1 #bao gồm          "unp.h"

2 khoảng trắng tinh recvfrom_int(int);
3 số int tinh;

4 khoảng trắng

5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     socklen_t len;
8     char mesg[MAXLINE];

9     Tín hiệu(SIGINT, recvfrom_int);

10    vì (      ;      ) {
11        len = clilen;
```

```

12         Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13         dém++;
14     }
15 }

16 khoảng trống tĩnh

17 recvfrom_int(int signo)
18 {
19     printf("\ndã nhận %d datagram\n", count);
20     thoát (0);
21 }

```

Bây giờ chúng tôi chạy máy chủ trên máy chủ freebsd, một SPARCStation chậm. Chúng tôi chạy máy khách trên hệ thống RS/6000 aix, được kết nối trực tiếp với Ethernet 100Mbps. Ngoài ra, chúng tôi chạy netstat -s trên máy chủ, cả trước và sau, vì số liệu thống kê xuất ra cho chúng tôi biết có bao nhiêu datagram bị mất. Hình 8.21 cho thấy đầu ra trên

máy chủ.

Hình 8.21 Đầu ra trên máy chủ.

```

freebsd % netstat -s -p udp
udp:
    Đã nhận đư ợc 71208 datagram
    0 với tiêu đề không đầy đủ
    0 với trự ờng độ dài dữ liệu sai
    0 với tổng kiểm tra sai
    0 không có tổng kiểm tra
    832 rớt do không có socket
    16 datagram quảng bá/đa hứ ờng bị rớt do không có ổ cắm
    1971 bị loại bỏ do bộ đệm ổ cắm đầy
    0 không dành cho pcb băm
    68389 đã giao
    Đầu ra 137685 datagram
freebsd % udpserv06
                                         khởi động máy chủ của chúng tôi
                                         chúng tôi điều hành
khách hàng ở đây
    ^C                                         chúng tôi gõ khóa ngắt sau máy khách
đã hoàn thành
đã nhận đư ợc 30 datagram
freebsd % netstat -s -p udp
udp:
    Đã nhận đư ợc 73208 datagram
    0 với tiêu đề không đầy đủ
    0 với trự ờng độ dài dữ liệu sai

```

```

0 với tổng kiểm tra sai
0 không có tổng kiểm tra
832 rớt do không có socket
16 datagram quảng bá/đa hống bị rớt do không có ô cắm
3941 bị rớt do bộ đệm ô cắm đầy
0 không dành cho pcb băm
68419 đã giao
Đầu ra 137685 datagram

```

Máy khách đã gửi 2.000 gói dữ liệu như ứng dụng máy chủ chỉ nhận được 30 gói dữ liệu trong số này, với tỷ lệ thất thoát là 98%. Không có dấu hiệu nào cho thấy ứng dụng máy chủ hoặc ứng dụng khách cho biết các gói dữ liệu này đã bị mất. Như chúng tôi đã nói, UDP không có khả năng kiểm soát luồng và nó không đáng tin cậy. Như chúng tôi đã chỉ ra, việc người gửi UDP vứt qua người nhận là chuyện bình thường.

Nếu chúng ta nhìn vào đầu ra `netstat`, tổng số datagram mà máy chủ máy chủ (không phải ứng dụng máy chủ) nhận được là 2.000 (73.208 - 71.208). Bộ đếm "bị rớt do bộ đệm ô cắm đầy" cho biết UDP đã nhận được bao nhiêu gói dữ liệu như bị loại bỏ do hàng đợi nhận của ô cắm nhận đã đầy (tr. 775 của TCPv2). Giá trị này là 1.970 (3.491 - 1.971), giá trị này khi được ứng dụng thêm vào đầu ra bộ đếm (30), bằng 2.000 datagram mà máy chủ nhận được.

Thật không may, bộ đếm `netstat` của số bị giảm do bộ đệm ô cắm đầy trên toàn hệ thống. Không có cách nào để xác định ứng dụng nào (ví dụ: cổng UDP nào) bị ảnh hưởng.

Số lưu lượng datagram mà máy chủ nhận được trong ví dụ này là không thể dự đoán được. Nó phụ thuộc vào nhiều yếu tố, chẳng hạn như tải mạng, tải xử lý trên máy chủ khách và tải xử lý trên máy chủ máy chủ.

Nếu chúng ta chạy cùng một máy khách và máy chủ, như lần này với máy khách trên Sun chậm và máy chủ trên RS/6000 nhanh hơn, không có datagram nào bị mất.

```

aix% udpserv06
^?
chúng tôi gõ phím ngắt sau khi khách hàng kết thúc
đã nhận được 2000 datagram
Bộ đệm nhận ô cắm UDP

```

Số lưu lượng gói dữ liệu UDP được UDP xếp hàng cho một ô cắm nhất định bị giới hạn bởi kích thước bộ đệm nhận của ô cắm đó. Chúng ta có thể thay đổi điều này bằng tùy chọn ô cắm `SO_RCVBUF`, như chúng ta đã mô tả trong [Phần 7.5](#). Kích thước mặc định của bộ đệm nhận ô cắm UDP trong FreeBSD là 42.080 byte, chỉ cho phép chứa 30 trong số 1.400 byte datagram của chúng tôi. Nếu chúng ta tăng kích thước của bộ đệm nhận ô cắm, chúng ta hy vọng máy chủ sẽ nhận được các gói dữ liệu bổ sung. [Hình 8.22](#) cho thấy một sửa đổi đối với hàm `dg_echo` từ [Hình 8.20](#), đặt bộ đệm nhận ô cắm thành 240 KB.

Hình 8.22 Hàm dg_echo làm tăng kích thư ớc của hàng đợi nhận socket.

udpcliserv/dgecholoop2.c

```

1 #include "unp.h"

2 khoảng trống tĩnh recvfrom_int(int);
3 só int tĩnh;

4 khoảng trống

5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
6 {
7     int      N;
8     socklen_t len;
9     char mesg[MAXLINE];

10    Tín hiệu(SIGINT, recvfrom_int);

11    n = 220 * 1024;
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13    vì (    ;    ) {
14        len = clilen;
15        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16        đếm++;
17    }
18 }

19 khoảng trống tĩnh

20 recvfrom_int(int signo)
21 {
22     printf("\nđã nhận %d datagram\n", count);
23     thoát (0);
24 }
```

Nếu chúng ta chạy máy chủ này trên Sun và máy khách trên RS/6000 thì số lư ợng datagram đã nhận hiện là 103. Mặc dù điều này tốt hơn một chút so với ví dụ trước đó với bộ đệm nhận ở cắm mặc định, như ng nó không phải là thuốc chữa bách bệnh.

Tại sao chúng ta đặt kích thư ớc bộ đệm ở cắm nhận là 220×1.024 trong [Hình 8.22](#)? Kích thư ớc tối đa của bộ đệm nhận ở cắm trong FreeBSD 5.1 mặc định là 262.144 byte (256×1.024), như ng do chính sách phân bổ bộ đệm (đư ợc mô tả trong Chương 2 của

TCPv2), giới hạn thực tế là 233.016 byte. Nhiều hệ thống trước đây dựa trên 4.3BSD đã giới hạn kích thước của bộ đệm ở cắm ở mức khoảng 52.000 byte.

8.14 Xác định giao diện gửi đi bằng UDP

Ở cắm UDP được kết nối cũng có thể được sử dụng để xác định giao diện gửi đi sẽ được sử dụng cho một đích cụ thể. Điều này là do tác dụng phụ của việc **kết nối** chức năng khi được áp dụng cho ổ cắm UDP: Hạt nhân chọn địa chỉ IP cục bộ (giả sử tiến trình chưa được gọi **liên kết** để gán rõ ràng địa chỉ này). Địa chỉ IP cục bộ này được chọn bằng cách tìm kiếm trong bảng định tuyến để tìm địa chỉ IP đích, sau đó sử dụng địa chỉ IP chính cho giao diện kết quả.

Hình 8.23 cho thấy một chương trình UDP đơn giản kết nối với một địa chỉ IP được chỉ định và sau đó gọi **gethostname**, in địa chỉ IP và cổng cục bộ.

Hình 8.23 Chương trình UDP sử dụng kết nối để xác định giao diện gửi đi.

udpcliserv/udpcli09.c

```

1 #bao gồm          "unp.h"

2 số nguyên

3 chính(int argc, char **argv)
4 {
5     int      sockfd;
6     socklen_t len;
7     struct sockaddr_in cliaddr, servaddr;

    nếu (argc != 2)
9         err_quit("cách sử dụng: udpcli <IPaddress>");

10    sockfd = Ở cắm(AF_INET, SOCK_DGRAM, 0);

11    bzero(&servaddr, sizeof(servaddr));
12    servaddr.sin_family = AF_INET;
13    servaddr.sin_port = htons(SERV_PORT);
14    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

15    Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));

16    len = sizeof(cliaddr);

```

```

17     Getsockname(sockfd, (SA *) &cliaddr, &len);
18     printf("địa chỉ cục bộ %s\n", Sock_ntop((SA *) &cliaddr, len));
19     thoát (0);
20 }

```

Nếu chúng tôi chạy chương trình trên máy chủ multihomed `freebsd`, chúng tôi có kết quả đầu ra sau:

```
freebsd % udpcli09 206.168.112.96
địa chỉ cục bộ 12.106.32.254:52329
```

```
freebsd% udpcli09 192.168.42.2
địa chỉ địa phương 192.168.42.1:52330
```

```
freebsd% udpcli09 127.0.0.1
địa chỉ cục bộ 127.0.0.1:52331
```

Lần đầu tiên chúng ta chạy chương trình, đối số dòng lệnh là một địa chỉ IP đi theo lô trình mặc định. Hạt nhân gán địa chỉ IP cục bộ cho địa chỉ chính của giao diện mà tuyến đường mặc định trả tới. Lần thứ hai, đối số là địa chỉ IP của hệ thống được kết nối với giao diện Ethernet thứ hai, do đó kernel sẽ gán địa chỉ IP cục bộ cho địa chỉ chính của giao diện thứ hai này.

Gọi **kết nối** trên ổ cắm UDP không gửi bất cứ thứ gì đến máy chủ đó; nó hoàn toàn là một hoạt động cục bộ nhằm lưu địa chỉ IP và cổng của thiết bị ngang hàng. Chúng tôi cũng thấy rằng **kết nối** cuộc gọi trên ổ cắm UDP không liên kết cũng chỉ định một cổng tạm thời cho ổ cắm.

Thật không may, kỹ thuật này không hoạt động trên tất cả các triển khai, chủ yếu là các hạt nhân có nguồn gốc từ SVR4. Ví dụ: tính năng này không hoạt động trên Solaris 2.5 nhưng nó hoạt động trên AIX, HP-UX 11, MacOS X, FreeBSD, Linux và Solaris 2.6 trở lên.

8.15 Máy chủ Echo TCP và UDP Sử dụng 'select'

Bây giờ chúng ta kết hợp máy chủ tiếng vang TCP đồng thời của chúng ta từ [Chương 5](#) với máy chủ tiếng vang UDP lặp lại của chúng ta từ chương này thành một máy chủ duy nhất sử dụng **tính năng chọn** để ghép kênh ổ cắm TCP và UDP. [Hình 8.24](#) là nửa đầu của máy chủ này.

Tạo ổ cắm TCP nghe

14-22 Ở cẩm TCP nghe đư ợc tạo và liên kết với cổng phô biến của máy chủ.

Chúng tôi đặt tùy chọn ở cẩm **SO_REUSEADDR** trong trường hợp có kết nối trên cổng này.

Tạo ở cẩm UDP

23-29 Ở cẩm UDP cũng đư ợc tạo và liên kết với cùng một cổng. Mặc dù sử dụng cùng một cổng cho ở cẩm TCP và UDP nhưng không cần thiết lập **SO_REUSEADDR**

tùy chọn socket trước lệnh gọi **liên kết này**, vì các cổng TCP độc lập với các cổng UDP.

[Hình 8.25 hiển thị](#) nữa sau của máy chủ của chúng tôi.

Thiết lập bộ xử lý tín hiệu cho SIGCHLD

30 Trình xử lý tín hiệu đư ợc thiết lập cho **SIGCHLD** vì các kết nối TCP sẽ đư ợc xử lý bởi một tiến trình con. Chúng tôi đã trình bày bộ xử lý tín hiệu này trong [Hình 5.11](#).

Chuẩn bị chọn lọc

31-32 Chúng tôi khởi tạo bộ mô tả để **chọn** và tính toán mức tối đa của hai bộ mô tả mà chúng tôi sẽ đợi.

[Hình 8.24](#) Nửa đầu của máy chủ echo xử lý TCP và UDP bằng cách sử dụng select.

udpcliserv/udpservselect01.c

```

1 #bao gồm          "unp.h"
2 số nguyên
3 chính(int argc, char **argv)
4 {
5     int      listenfd, connfd, udpfd, nready, maxfdp1;
6     char mesg[MAXLINE];
7     pid_t trẻ con;
8     fd_set đặt lại;
9     cõ_t n;
10    socklen_t len;
11    const int trên = 1;
12    struct sockaddr_in cliaddr, servaddr;
13    void sig_chld(int);

14    /* tạo socket TCP nghe */
15    listenfd = Ở cẩm(AF_INET, SOCK_STREAM, 0);

16    bzero(&servaddr, sizeof(servaddr));

```

```

17     servaddr.sin_family = AF_INET;
18     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
19     servaddr.sin_port = htons(SERV_PORT);

20     Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

22     Nghe(listenfd, LISTENQ);

23     /* tạo ô cắm UDP */
24     udpfd = Ở cắm(AF_INET, SOCK_DGRAM, 0);

25     bzero(&servaddr, sizeof(servaddr));
26     servaddr.sin_family = AF_INET;
27     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
28     servaddr.sin_port = htons(SERV_PORT);

29     Bind(udpfd, (SA *) &servaddr, sizeof(servaddr));

```

Chọn cuộc gọi

34-41 Chúng tôi gọi **select**, chỉ chờ khả năng đọc trên ô cắm TCP đang nghe hoặc khả năng đọc trên ô cắm UDP. Vì trình xử lý **sig_chld** của chúng tôi có thể làm gián đoạn cuộc gọi để **chọn** nên chúng tôi xử lý lỗi **EINTR**.

Xử lý kết nối máy khách mới

42-51 Chúng tôi **chấp nhận** một kết nối máy khách mới khi ô cắm TCP đang nghe có thể đọc đư ợc, **phân tách** một phần tử con và gọi hàm **str_echo** của chúng tôi trong phần tử con. Đây là trình tự các bước tư ơng tự mà chúng tôi đã sử dụng trong [Chương 5](#).

Hình 8.25 Nửa sau của máy chủ echo xử lý TCP và UDP bằng cách sử dụng **select**.

udpcliserv/udpservselect01.c

```

30     Tín hiệu(SIGCHLD, sig_chld);           /* phải gọi waitpid() */

31     FD_ZERO(&rset);
32     maxfdp1 = max(listenfd, udpfd) + 1;
33     vì (    ;    ) {
34         FD_SET(nghefd, &rset);
35         FD_SET(udpfd, &rset);
36         if ( (nready = select(maxfdp1, &rset, NULL, NULL, NULL)) < 0) {
37             nếu (errno == EINTR)
38                 Tiếp tục;          /* quay lại for() */

```

```

39             khác
40             err_sys("lỗi chọn");
41         }

42         if (FD_ISSET(listenfd, &rset)) {
43             len = sizeof(cliaddr);
44             connfd = Chấp nhận(listenfd, (SA *) &cliaddr, &len);

45             if ( (childpid = Fork()) == 0) { /* tiến trình con */
46                 Đóng(nghefd);           /* đóng socket nghe */
47                 str_echo(connfd);    thoát      /* xử lý yêu cầu */
48                 (0);
49             }
50             Đóng(connfd);          /* cha mẹ đóng socket đã kết nối */
51         }

52         if (FD_ISSET(udpfds, &rset)) {
53             len = sizeof(cliaddr);
54             n = Recvfrom(udpfds, mesg, MAXLINE, 0, (SA *) &cliaddr, &len);

55             Sendto(udpfds, mesg, n, 0, (SA *) &cliaddr, len);
56         }
57     }
58 }
```

Xử lý sự kiện của datagram

52-57 Nếu ở cắm UDP có thể đọc được thì có nghĩa là gói dữ liệu đã đến. Chúng tôi đọc nó bằng `recvfrom` và gửi lại cho khách hàng bằng `sendto`.

8.16 Tóm tắt

Việc chuyển đổi máy khách/máy chủ echo của chúng tôi sang sử dụng UDP thay vì TCP rất đơn giản. Tuy nhiên, rất nhiều tính năng do TCP cung cấp bị thiếu: phát hiện các gói bị mất và truyền lại, xác minh các phản hồi là từ đúng thiết bị ngang hàng, v.v. Chúng ta sẽ quay lại chủ đề này trong [Phần 22.5](#) và xem cần những gì để tăng thêm độ tin cậy cho ứng dụng UDP.

Ở cắm UDP có thể tạo ra lỗi không đồng bộ, nghĩa là lỗi được báo cáo một thời gian sau khi gói được gửi. Các socket TCP luôn báo cáo những lỗi này cho ứng dụng, nhưng với UDP, socket phải được kết nối mới nhận được những lỗi này.

UDP không có điều khiển luồng và điều này rất dễ chứng minh. Thông thường, đây không phải là vấn đề vì nhiều ứng dụng UDP được xây dựng bằng mô hình yêu cầu-trả lời chứ không phải để truyền dữ liệu số lượng lớn.

Vẫn còn nhiều điểm cần xem xét khi viết ứng dụng UDP, như ng chúng ta sẽ lưu lại những điểm này cho đến [Chương 22](#), sau khi đã cập đến [các chức năng giao diện](#), phát sóng và phát đa hống.

Bài tập

[8.1](#) Chúng tôi có hai ứng dụng, một ứng dụng sử dụng TCP và ứng dụng kia sử dụng UDP. 4.096 byte nằm trong bộ đệm nhận cho ô cắm TCP và hai gói dữ liệu 2.048 byte nằm trong bộ đệm nhận cho ô cắm UDP. Cuộc gọi ứng dụng TCP [đư ợc đọc](#) với đối số thứ ba là 4.096 và ứng dụng UDP gọi [recvfrom](#) với đối số thứ ba là 4.096. Có sự khác biệt nào không?

[8.2](#) Điều gì xảy ra trong [Hình 8.4](#) nếu chúng ta thay thế đối số cuối cùng thành [sendto](#) (mà chúng ta hiển thị là [len](#)) bằng [clilen](#)?

8.3 Biên dịch và chạy máy chủ UDP trong [Hình 8.3](#) và [8.4](#) và sau đó là máy khách UDP trong [Hình 8.7](#) và [8.8](#). Xác minh rằng máy khách và máy chủ hoạt động cùng nhau.

[8.4](#) Chạy chương trình [ping](#) trong một cửa sổ, chỉ định tùy chọn [-i 60](#) (gửi một gói cứ sau 60 giây; một số hệ thống sử dụng [-I](#) thay vì [-i](#)), tùy chọn [-v](#) (in tất cả các lỗi ICMP nhận được) và địa chỉ loopback (Thông thư ờng là 127.0.0.1). Chúng tôi sẽ sử dụng chương trình này để xem cổng ICMP không thể truy cập được máy chủ trả về. Tiếp theo, chạy ứng dụng khách của chúng ta từ bài tập trước trong một cửa sổ khác, chỉ định địa chỉ IP của một số máy chủ không chạy máy chủ.

Điều gì xảy ra?

[8.5](#) Chúng ta đã nói trong [Hình 8.5](#) rằng mỗi socket TCP được kết nối có bộ đệm nhận socket riêng. Còn ô cắm nghe thì sao; bạn có nghĩ rằng nó có bộ đệm nhận ô cắm riêng không?

[8.6](#) Sử dụng chương trình [sock](#) ([Phần C.3](#)) và một công cụ như [tcpdump](#) ([Phần C.5](#)) để kiểm tra những [giá chúng tôi đã tuyên bố](#) trong [Phần 8.10](#): Nếu máy khách [liên kết](#) một địa chỉ IP với địa chỉ IP của nó socket như ng gửi một datagram đi ra ngoài một số giao diện khác, datagram IP kết quả vẫn chứa địa chỉ IP được liên kết với socket, mặc dù địa chỉ này không tương ứng với giao diện đi.

[8.7](#) Biên dịch các chương trình từ [Phần 8.13](#) và chạy máy khách và máy chủ trên các máy chủ khác nhau. Đặt một [printf](#) vào máy khách mỗi khi một datagram được ghi vào

ở cắm. Điều này có thay đổi tỷ lệ phần trăm gói nhận được không? Tại sao? Đặt một `printf` vào máy chủ mỗi khi datagram được đọc từ socket. Điều này có thay đổi tỷ lệ phần trăm gói nhận được không? Tại sao?

8.8 Độ dài lớn nhất mà chúng ta có thể chuyển tới `sendto` cho ô cắm UDP/IPv4 là bao nhiêu, tức là lượng dữ liệu lớn nhất có thể chứa vừa trong một datagram UDP/IPv4 là bao nhiêu? Điều gì thay đổi với UDP/IPv6?

Sửa đổi Hình 8.8 để gửi một gói dữ liệu UDP có kích thước tối đa, đọc lại và in số byte được trả về bởi `recvfrom`.

8.9 Sửa đổi Hình 8.25 để phù hợp với RFC 1122 bằng cách sử dụng `IP_RECVSTADDR` cho ô cắm UDP.

Chương 9. Ô cắm SCTP cơ bản

[Mục 9.1. Giới thiệu](#)

[Mục 9.2. Mô hình giao diện](#)

[Mục 9.3. Hàm `sctp_bindx`](#)

[Mục 9.4. Chức năng `sctp_connectx`](#)

[Mục 9.5. Chức năng `sctp_getpaddrs`](#)

[Mục 9.6. Chức năng `sctp_freepaddrs`](#)

[Mục 9.7. hàm `sctp_getladdrs`](#)

[Mục 9.8. Chức năng `sctp_freeladdrs`](#)

[Mục 9.9. Chức năng `sctp_sendmsg`](#)

[Mục 9.10. Chức năng `sctp_recvmsg`](#)

[Mục 9.11. Chức năng `sctp_opt_info`](#)

[Mục 9.12. Chức năng `sctp_peeloff`](#)

[Mục 9.13. chức năng tắt máy](#)

[Mục 9.14. Thông báo](#)