CS110: Principles of Computer Systems

Autumn 2021
Jerry Cain
PDF

Lecture 22: Network System Calls, Library Functions

- The three data structures presented below are in place to model IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **`sockaddr_in`** is used to model IPv4 address/port pairs.

- The **`sin_family`** field should always be initialized to be **`AF_INET`**, which is a constant used when IPv4 addresses are being used. If it feels redundant that a record dedicated to IPv4 needs to store a constant saying everything is IPv4, then stay tuned.
- The **`sin_port`** field stores a port number in network byte (i.e. big endian) order.
- The **`sin_addr`** field stores an IPv4 address as a packed, big endian **`int`**, as you saw with **`gethostbyname`** and the **`struct hostent`**.
- The **`sin_zero`** field is generally ignored (though it's often set to store all zeroes). It exists to pad the record up to 16 bytes.

Lecture 22: Network System Calls, Library Functions

- The three data structures presented below are in place to model IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **sockaddr_in6** is used to model IPv6 address/port pairs.

- The **sin6_family** field should always be set to **AF_INET6**. As with the **sin_family** field, **sin6_family** field occupies the first two bytes of surrounding record.
- The **sin6_port** field holds a two-byte, network-byte-ordered port number, just like sin_port does.
- A **struct in6_addr** is also wedged in there to manage a 128-bit IPv6 address.
- **sin6_flowinfo** and **sin6_scope_id** are beyond the scope of what we need, so we'll ignore them.

Lecture 22: Network System Calls, Library Functions

- The three data structures presented below are in place to model IP address/port pairs:

```
struct sockaddr { // generic socket
    unsigned short sa_family; // protocol family for socket
    char sa_data[14];
    // address data (and defines full size to be 16 bytes)
};
```

```
struct sockaddr_in { // IPv4 socket address record
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};
```

```
struct sockaddr_in6 { // IPv6 socket address record
    unsigned short sin6_family;
    unsigned short sin6_port;
    unsigned int sin6_flowinfo;;
    struct in6_addr sin6_addr;
    unsigned int sin6_scope_id;
};
```

The **`struct sockaddr`** is the best C can do to emulate an abstract base class.

- You rarely if ever declare variables of type **`struct sockaddr`**, but many system calls will accept parameters of type **`struct sockaddr *`**.
- Rather than define a set of network system calls for IPv4 addresses and a second set of system calls for IPv6 addresses, Linux defines one set for both.
- If a system call accepts a parameter of type **`struct sockaddr *`**, it really accepts the address of either a **`struct sockaddr_in`** or a **`struct sockaddr_in6`**. The system call relies on the value within the first two bytes—the **`sa_family`** field—to determine what the true record type is.

Lecture 22: Network System Calls, Library Functions

At this point, we know most of the directives needed to implement and understand how to implement **`createClientSocket`** and **`createServerSocket`**.

- **`createClientSocket`** is the easier of the two, so we'll implement that one first. (For simplicity, we'll confine ourselves to an IPv4 world.)
- Fundamentally, **createClientSocket** needs to:
  - Confirm the host of interest is really on the net by checking to see if it has an IP address. **`gethostbyname`** does this for us.
  - Allocate a new descriptor and configure it to be a socket descriptor. We'll rely on the **`socket`** system call to do this.
  - Construct an instance of a **`struct sockaddr_in`** that packages the host and port number we're interested in connecting to.
  - Associate the freshly allocated socket descriptor with the host/port pair. We'll rely on an aptly named system call called **`connect`** to do this.
  - Return the fully configured client socket.
- The full implementation of createClientSocket is on the next slide (and right here).

Lecture 15: Network System Calls, Library Functions

Here is the full implementation of **`createClientSocket`**:

```
int createClientSocket(const string& host, unsigned short port) {
    struct hostent *he = gethostbyname(host.c_str());
    if (he == NULL) return -1;

    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) return -1;

    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = htons(port);

    // h_addr is #define for h_addr_list[0]
    address.sin_addr = *((struct in_addr *)he->h_addr);
    if (connect(s, (struct sockaddr *) &address, sizeof(address)) == 0) return s;

    close(s);
    return -1;
}
```

Lecture 15: Network System Calls, Library Functions

Here are key details about my **`createClientSocket`** implementation worth calling out:

- We call **`gethostbyname`** first before we call **`socket`**, because we want to confirm the host has a registered IP address—which means it's reachable—before we allocate any system resources.
- Recall that **`gethostbyname`** is intrinsically IPv4. If we wanted to involve IPv6 addresses instead, we would need to use **`gethostbyname2`**.
- The call to **`socket`** finds, claims, and returns an unused descriptor. **`AF_INET`** configures it to be compatible with an IPv4 address, and **`SOCK_STREAM`** configures it to provide reliable data transport, which basically means the socket will reorder data packets and requests missing or garbled data packets to be resent so as to give the impression that data that is received in the order it's sent.
  - The first argument could have been **`AF_INET6`** had we decided to use IPv6 addresses instead. (Other arguments are possible, but they're less common.)
  - The second argument could have been **`SOCK_DGRAM`** had we preferred to collect data packets in the order they just happen to arrive and manage missing and garbled data packets ourselves. (Other arguments are possible, though they're less common.)

Lecture 15: Network System Calls, Library Functions

Here are a few more details:

- **address** is declared to be of type **struct sockaddr_in**, since that's the data type specifically set up to model IPv4 addresses. Had we been dealing with IPv6 addresses, we'd have declared a **struct sockaddr_in6** instead.
  - It's important to embed **AF_INET** within **sin_family**, since those two bytes are examined by system calls to determine the type of socket address structure.
  - The **sin_port** field is, not surprisingly, designed to hold the port of interest. **htons**—that's an abbreviation for **h**ost-**to**-**n**etwork-**s**hort—is there to ensure the port is stored in network byte order (which is big endian order). On big endian machines, **htons** is implemented to return the provided short without modification. On little endian machines (like the **myth**s), **htons** returns a figure constructed by exchanging the two bytes of the incoming **short**. In addition to **htons**, Linux also provided **htonl** for four-byte **long**s, and it also provides **ntohs** and **ntohl** to restore host byte order from network byte ordered figures.
- The call to **connect** associates the descriptor **s** with the host/IP address pair modeled by the supplied **struct sockaddr_in ***. The second argument is downcast to a **struct sockaddr ***, since **connect** must accept a pointer to **any** type within the entire **struct sockaddr** family, not just **struct sockaddr_in**s. **connect** will return -1 (with **errno** set to **ECONNREFUSED**) if the server of interest isn't running.

Lecture 15: Network System Calls, Library Functions

Here is the full implementation of **`createServerSocket`** (and online right here):

```
int createServerSocket(unsigned short port, int backlog) {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    if (s < 0) return -1;

    struct sockaddr_in address;
    memset(&address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl(INADDR_ANY);
    address.sin_port = htons(port);

    if (bind(s, (struct sockaddr *)&address, sizeof(address)) == 0 &&
        listen(s, backlog) == 0) return s;

    close(s);
    return -1;
}
```

Lecture 15: Network System Calls, Library Functions

Here are key details about my implementation of **`createServerSocket`**:

- The call to **`socket`** is precisely the same here as it was in **`createClientSocket`**. It allocates a descriptor and configures it to be a socket descriptor within the **`AF_INET`** namespace.
- The address of type **`struct sockaddr_in`** here is configured in much the same way it was in **`createClientSocket`**, except that the **`sin_addr.s_addr`** field should be set to a local IP address, not a remote one. The constant **`INADDR_ANY`** is used to state that address should represent all local addresses.
- The **`bind`** call simply assigns the set of local IP addresses represented by **`address`** to the provided socket **`s`**. Because we embedded **`INADDR_ANY`** within **`address`**, **`bind`** associates the supplied socket with all local IP addresses. That means once **`createServerSocket`** has done its job, clients can connect to any of the machine's IP addresses via the specified port.
- The **`listen`** call is what converts the socket to be one that's willing to accept connections via **`accept`**. The second argument is a queue size limit, which states how many pending connection requests can accumulate and wait their turn to be **`accept`**ed. If the number of outstanding requests is at the limit, additional requests are simply refused.