4. System Calls or Bust

This is the section where we get into the system calls that allow you to access the network functionality of a Unix box. When you call one of these functions, the kernel takes over and does all the work for you automagically.

The place most people get stuck around here is what order to call these things in. In that, the **`man`** pages are no use, as you've probably discovered. Well, to help with that dreadful situation, I've tried to lay out the system calls in the following sections in *exactly* (approximately) the same order that you'll need to call them in your programs.

That, coupled with a few pieces of sample code here and there, some milk and cookies (which I fear you will have to supply yourself), and some raw guts and courage, and you'll be beaming data around the Internet like the Son of Jon Postel!

4.1. `socket()`--Get the File Descriptor!

I guess I can put it off no longer--I have to talk about the `socket()` system call. Here's the breakdown:

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

int socket(int domain, int type, int protocol);
```

But what are these arguments? First, *`domain`* should be set to "**`PF_INET`**". Next, the *`type`* argument tells the kernel what kind of socket this is: **`SOCK_STREAM`** or **`SOCK_DGRAM`**. Finally, just set *`protocol`* to "**`0`**" to have `socket()` choose the correct protocol based on the *`type`*. (Notes: there are many more *`domain`*s than I've listed. There are many more *`type`*s than I've listed. See the `socket()` man page. Also, there's a "better" way to get the *`protocol`*, but specifying **`0`** works in 99.9% of all cases. See the `getprotobyname()` man page if you're curious.)

`socket()` simply returns to you a socket descriptor that you can use in later system calls, or **`-1`** on error. The global variable *`errno`* is set to the error's value (see the `perror()` man page.)

(This **`PF_INET`** thing is a close relative of the **`AF_INET`** that you used when initializing the **`sin_family`** field in your `struct sockaddr_in`. In fact, they're so closely related that they actually have the same value, and many programmers will call `socket()` and pass **`AF_INET`** as the first argument instead of `PF_INET`. Now, get some milk and cookies, because it's times for a story. Once upon a time, a long time ago, it was thought that maybe a address family (what the "AF" in "**`AF_INET`**" stands for) might support several protocols that were referred to by their protocol family (what the "PF" in "**`PF_INET`**" stands for). That didn't happen. And they all lived happily ever after, The End. So the most correct thing to do is to use **`AF_INET`** in your `struct sockaddr_in` and **`PF_INET`** in your call to `socket()`.)

Fine, fine, fine, but what good is this socket? The answer is that it's really no good by itself, and you need to read on and make more system calls for it to make any sense.

4.2. `bind()`--What port am I on?

Once you have a socket, you might have to associate that socket with a port on your local machine. (This is commonly done if you're going to `listen()` for incoming connections on a specific port--MUDs do this when they tell you to "telnet to x.y.z port 6969".) The port number is used by the kernel to match an incoming packet to a certain process's socket descriptor. If you're going to only be doing a `connect()`, this may be unnecessary. Read it anyway, just for kicks.

Here is the synopsis for the `bind()` system call:

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

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd is the socket file descriptor returned by `socket()`. *my_addr* is a pointer to a `struct sockaddr` that contains information about your address, namely, port and IP address. *addrlen* can be set to `sizeof(struct sockaddr)`.

Whew. That's a bit to absorb in one chunk. Let's have an example:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(PF_INET, SOCK_STREAM, 0); // do some error checking!

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    // don't forget your error checking for bind():
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

There are a few things to notice here: *my_addr.sin_port* is in Network Byte Order. So is *my_addr.sin_addr.s_addr*. Another thing to watch out for is that the header files might differ from system to system. To be sure, you should check your local **man** pages.

Lastly, on the topic of `bind()`, I should mention that some of the process of getting your own IP address and/or port can be automated:

```
my_addr.sin_port = 0; // choose an unused port at random
my_addr.sin_addr.s_addr = INADDR_ANY;  // use my IP address
```

See, by setting *my_addr.sin_port* to zero, you are telling `bind()` to choose the port for you. Likewise, by setting *my_addr.sin_addr.s_addr* to **INADDR_ANY**, you are telling it to automatically fill in the IP address of the machine the process is running on.

If you are into noticing little things, you might have seen that I didn't put **INADDR_ANY** into Network Byte Order! Naughty me. However, I have inside info: **INADDR_ANY** is really zero! Zero still has zero on bits even if you rearrange the bytes. However, purists will point out that there could be a parallel dimension where **INADDR_ANY** is, say, 12 and that my code won't work there. That's ok with me:

```
my_addr.sin_port = htons(0); // choose an unused port at random
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // use my IP address
```

Now we're so portable you probably wouldn't believe it. I just wanted to point that out, since most of the code you come across won't bother running **INADDR_ANY** through `htonl()`.

`bind()` also returns **-1** on error and sets *errno* to the error's value.

Another thing to watch out for when calling `bind()`: don't go underboard with your port numbers. All ports below 1024 are RESERVED (unless you're the superuser)! You can have any port number above that, right up to 65535 (provided they aren't already being used by another program.)

Sometimes, you might notice, you try to rerun a server and `bind()` fails, claiming "Address already in use." What does that mean? Well, a little bit of a socket that was connected is still hanging around in the kernel, and it's hogging the port. You can either wait for it to clear (a minute or so), or add code to your program allowing it to reuse the port, like this:

```
int yes=1;
//char yes='1'; // Solaris people use this

// lose the pesky "Address already in use" error message
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

One small extra final note about `bind()`: there are times when you won't absolutely have to call it. If you are `connect()`ing to a remote machine and you don't care what your local port is (as is the case with **telnet** where you only care about the remote port), you can simply call `connect()`, it'll check to see if the socket is unbound, and will `bind()` it to an unused local port if necessary.

4.3. `connect()`--Hey, you!

Let's just pretend for a few minutes that you're a telnet application. Your user commands you (just like in the movie *TRON*) to get a socket file descriptor. You comply and call `socket()`. Next, the user tells you to connect to "`10.12.110.57`" on port "`23`" (the standard telnet port.) Yow! What do you do now?

Lucky for you, program, you're now perusing the section on `connect()`--how to connect to a remote host. So read furiously onward! No time to lose!

The `connect()` call is as follows:

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

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

`sockfd` is our friendly neighborhood socket file descriptor, as returned by the `socket()` call, *`serv_addr`* is a `struct sockaddr` containing the destination port and IP address, and *`addrlen`* can be set to `sizeof(struct sockaddr)`.

Isn't this starting to make more sense? Let's have an example:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEST_IP   "10.12.110.57"
#define DEST_PORT 23

main()
{
    int sockfd;
    struct sockaddr_in dest_addr;   // will hold the destination addr

    sockfd = socket(PF_INET, SOCK_STREAM, 0); // do some error checking!

    dest_addr.sin_family = AF_INET;          // host byte order
    dest_addr.sin_port = htons(DEST_PORT);   // short, network byte order
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    memset(&(dest_addr.sin_zero), '\0', 8);  // zero the rest of the struct

    // don't forget to error check the connect()!
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    .
    .
    .
```

Again, be sure to check the return value from `connect()`--it'll return **`-1`** on error and set the variable *`errno`*.

Also, notice that we didn't call `bind()`. Basically, we don't care about our local port number; we only care where we're going (the remote port). The kernel will choose a local port for us, and the site we connect to will automatically get this information from us. No worries.

4.4. `listen()`--Will somebody please call me?

Ok, time for a change of pace. What if you don't want to connect to a remote host. Say, just for kicks, that you want to wait for incoming connections and handle them in some way. The process is two step: first you `listen()`, then you `accept()` (see below.)

The listen call is fairly simple, but requires a bit of explanation:

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

`sockfd` is the usual socket file descriptor from the `socket()` system call. *`backlog`* is the number of connections allowed on the incoming queue. What does that mean? Well, incoming connections are going to wait in this queue until you `accept()` them (see below) and this is the limit on how many can queue up. Most systems silently limit this number to about 20; you can probably get away with setting it to **`5`** or **`10`**.

Again, as per usual, `listen()` returns **-1** and sets *errno* on error.

Well, as you can probably imagine, we need to call `bind()` before we call `listen()` or the kernel will have us listening on a random port. Bleah! So if you're going to be listening for incoming connections, the sequence of system calls you'll make is:

```
socket();
bind();
listen();
/* accept() goes here */
```

I'll just leave that in the place of sample code, since it's fairly self-explanatory. (The code in the `accept()` section, below, is more complete.) The really tricky part of this whole sha-bang is the call to `accept()`.

4.5. `accept()`--"Thank you for calling port 3490."

Get ready--the `accept()` call is kinda weird! What's going to happen is this: someone far far away will try to `connect()` to your machine on a port that you are `listen()`ing on. Their connection will be queued up waiting to be `accept()`ed. You call `accept()` and you tell it to get the pending connection. It'll return to you a *brand new socket file descriptor* to use for this single connection! That's right, suddenly you have *two socket file descriptors* for the price of one! The original one is still listening on your port and the newly created one is finally ready to `send()` and `recv()`. We're there!

The call is as follows:

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

sockfd is the `listen()`ing socket descriptor. Easy enough. *addr* will usually be a pointer to a local `struct sockaddr_in`. This is where the information about the incoming connection will go (and with it you can determine which host is calling you from which port). *addrlen* is a local integer variable that should be set to `sizeof(struct sockaddr_in)` before its address is passed to `accept()`. Accept will not put more than that many bytes into *addr*. If it puts fewer in, it'll change the value of *addrlen* to reflect that.

Guess what? `accept()` returns **-1** and sets *errno* if an error occurs. Betcha didn't figure that.

Like before, this is a bunch to absorb in one chunk, so here's a sample code fragment for your perusal:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 3490    // the port users will be connecting to

#define BACKLOG 10     // how many pending connections queue will hold

main()
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr;    // my address information
    struct sockaddr_in their_addr; // connector's address information
```

```
    int sin_size;

    sockfd = socket(PF_INET, SOCK_STREAM, 0); // do some error checking!

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // auto-fill with my IP
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    // don't forget your error checking for these calls:
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

    sin_size = sizeof(struct sockaddr_in);
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    .
    .
    .
```

Again, note that we will use the socket descriptor *new_fd* for all `send()` and `recv()` calls. If you're only getting one single connection ever, you can `close()` the listening *sockfd* in order to prevent more incoming connections on the same port, if you so desire.

4.6. `send()` and `recv()`--Talk to me, baby!

These two functions are for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets, you'll need to see the section on `sendto() and recvfrom()`, below.

The `send()` call:

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd is the socket descriptor you want to send data to (whether it's the one returned by `socket()` or the one you got with `accept()`.) *msg* is a pointer to the data you want to send, and *len* is the length of that data in bytes. Just set *flags* to **0**. (See the `send()` man page for more information concerning flags.)

Some sample code might be:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

`send()` returns the number of bytes actually sent out--*this might be less than the number you told it to send!* See, sometimes you tell it to send a whole gob of data and it just can't handle it. It'll fire off as much of the data as it can, and trust you to send the rest later. Remember, if the value returned by `send()` doesn't match the value in *len*, it's up to you to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will *probably* manage to send the whole thing all in one go. Again, **`-1`** is returned on error, and *`errno`* is set to the error number.

The `recv()` call is similar in many respects:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

`sockfd` is the socket descriptor to read from, *`buf`* is the buffer to read the information into, *`len`* is the maximum length of the buffer, and *`flags`* can again be set to **`0`**. (See the `recv()` man page for flag information.)

`recv()` returns the number of bytes actually read into the buffer, or **`-1`** on error (with *`errno`* set, accordingly.)

Wait! `recv()` can return **`0`**. This can mean only one thing: the remote side has closed the connection on you! A return value of **`0`** is `recv()`'s way of letting you know this has occurred.

There, that was easy, wasn't it? You can now pass data back and forth on stream sockets! Whee! You're a Unix Network Programmer!

4.7. `sendto()` and `recvfrom()`--Talk to me, DGRAM-style

"This is all fine and dandy," I hear you saying, "but where does this leave me with unconnected datagram sockets?" No problemo, amigo. We have just the thing.

Since datagram sockets aren't connected to a remote host, guess which piece of information we need to give before we send a packet? That's right! The destination address! Here's the scoop:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

As you can see, this call is basically the same as the call to `send()` with the addition of two other pieces of information. *`to`* is a pointer to a `struct sockaddr` (which you'll probably have as a `struct sockaddr_in` and cast it at the last minute) which contains the destination IP address and port. *`tolen`*, an `int` deep-down, can simply be set to `sizeof(struct sockaddr)`.

Just like with `send()`, `sendto()` returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or **`-1`** on error.

Equally similar are `recv()` and `recvfrom()`. The synopsis of `recvfrom()` is:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

Again, this is just like `recv()` with the addition of a couple fields. *`from`* is a pointer to a local `struct sockaddr` that will be filled with the IP address and port of the originating machine. *`fromlen`* is a pointer to a local `int` that should be initialized to `sizeof(struct sockaddr)`. When the function returns, *`fromlen`* will contain the length of the address actually stored in *`from`*.

`recvfrom()` returns the number of bytes received, or **`-1`** on error (with *`errno`* set accordingly.)

Remember, if you `connect()` a datagram socket, you can then simply use `send()` and `recv()` for all your transactions. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you.

4.8. `close()` and `shutdown()`--Get outta my face!

Whew! You've been `send()`ing and `recv()`ing data all day long, and you've had it. You're ready to close the connection on your socket descriptor. This is easy. You can just use the regular Unix file descriptor `close()` function:

```
close(sockfd);
```

This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error.

Just in case you want a little more control over how the socket closes, you can use the `shutdown()` function. It allows you to cut off communication in a certain direction, or both ways (just like `close()` does.) Synopsis:

```
int shutdown(int sockfd, int how);
```

`sockfd` is the socket file descriptor you want to shutdown, and *`how`* is one of the following:

- **`0`** -- Further receives are disallowed
- **`1`** -- Further sends are disallowed
- **`2`** -- Further sends and receives are disallowed (like `close()`)

`shutdown()` returns **`0`** on success, and **`-1`** on error (with *`errno`* set accordingly.)

If you deign to use `shutdown()` on unconnected datagram sockets, it will simply make the socket unavailable for further `send()` and `recv()` calls (remember that you can use these if you `connect()` your datagram socket.)

It's important to note that `shutdown()` doesn't actually close the file descriptor--it just changes its usability. To free a socket descriptor, you need to use `close()`.

Nothing to it.

4.9. `getpeername()`--Who are you?

This function is so easy.

It's so easy, I almost didn't give it it's own section. But here it is anyway.

The function `getpeername()` will tell you who is at the other end of a connected stream socket. The synopsis:

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

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

`sockfd` is the descriptor of the connected stream socket, *`addr`* is a pointer to a `struct sockaddr` (or a `struct sockaddr_in`) that will hold the information about the other side of the connection, and *`addrlen`* is a pointer to an `int`, that should be initialized to `sizeof(struct sockaddr)`.

The function returns **-1** on error and sets *errno* accordingly.

Once you have their address, you can use `inet_ntoa()` or `gethostbyaddr()` to print or get more information. No, you can't get their login name. (Ok, ok. If the other computer is running an ident daemon, this is possible. This, however, is beyond the scope of this document. Check out [RFC-1413](https://www.gta.ufrj.br/ensino/eel878/sockets/syscalls.html) for more info.)

4.10. `gethostname()`--Who am I?

Even easier than `getpeername()` is the function `gethostname()`. It returns the name of the computer that your program is running on. The name can then be used by `gethostbyname()`, below, to determine the IP address of your local machine.

What could be more fun? I could think of a few things, but they don't pertain to socket programming. Anyway, here's the breakdown:

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

The arguments are simple: *hostname* is a pointer to an array of chars that will contain the hostname upon the function's return, and *size* is the length in bytes of the *hostname* array.

The function returns **0** on successful completion, and **-1** on error, setting *errno* as usual.

4.11. DNS--You say "whitehouse.gov", I say "63.161.169.137"

In case you don't know what DNS is, it stands for "Domain Name Service". In a nutshell, you tell it what the human-readable address is for a site, and it'll give you the IP address (so you can use it with `bind()`, `connect()`, `sendto()`, or whatever you need it for.) This way, when someone enters:

```
$ telnet whitehouse.gov
```

telnet can find out that it needs to `connect()` to "63.161.169.137".

But how does it work? You'll be using the function `gethostbyname()`:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

As you see, it returns a pointer to a `struct hostent`, the layout of which is as follows:

```
struct hostent {
    char    *h_name;
    char    **h_aliases;
    int     h_addrtype;
    int     h_length;
    char    **h_addr_list;
```

```
};
#define h_addr h_addr_list[0]
```

And here are the descriptions of the fields in the `struct hostent`:

- *`h_name`* -- Official name of the host.
- *`h_aliases`* -- A NULL-terminated array of alternate names for the host.
- *`h_addrtype`* -- The type of address being returned; usually *`AF_INET`*.
- *`h_length`* -- The length of the address in bytes.
- *`h_addr_list`* -- A zero-terminated array of network addresses for the host. Host addresses are in Network Byte Order.
- *`h_addr`* -- The first address in *`h_addr_list`*.

`gethostbyname()` returns a pointer to the filled `struct hostent`, or NULL on error. (But *`errno`* is *not* set-- *`h_errno`* is set instead. See `herror()`, below.)

But how is it used? Sometimes (as we find from reading computer manuals), just spewing the information at the reader is not enough. This function is certainly easier to use than it looks.

[Here's an example program](#):

```
/*
** getip.c -- a hostname lookup demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    struct hostent *h;

    if (argc != 2) {  // error check the command line
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
        exit(1);
    }

    printf("Host name  : %s\n", h->h_name);
    printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));

   return 0;
}
```

With `gethostbyname()`, you can't use `perror()` to print error message (since *`errno`* is not used). Instead, call `herror()`.

It's pretty straightforward. You simply pass the string that contains the machine name ("whitehouse.gov") to `gethostbyname()`, and then grab the information out of the returned `struct hostent`.

The only possible weirdness might be in the printing of the IP address, above. *`h->h_addr`* is a `char*`, but `inet_ntoa()` wants a `struct in_addr` passed to it. So I cast *`h->h_addr`* to a `struct in_addr*`, then dereference it to get at the data.

Prev Contents Next