NETWORK PROGRAMMING LINUX SOCKET PART 4: THE APPLICATION PROGRAMMING INTERFACES (APIs)

Menu

Network Story 1
Network Story 2
Network Story 3
Network Story 4
Network Story 5
Network Story 6
Socket Example 1
Socket Example 2
Socket Example 3
Socket Example 4
Socket Example 5
Socket Example 6
Socket Example 7
Advanced TCP/IP 1
Advanced

My Training Period: xx hours

Note: Program examples if any, compiled using gcc on **Linux Fedora Core 3** machine with several update, as normal user. The Fedora machine used for the testing having the "No Stack Execute" disabled and the **SELinux** set to default configuration.

listen()

```
NAME
       listen() - listen for connections on a socket
SYNOPSIS
       #include <sys/socket.h>
       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.
- As an example, for the server, if you want to wait for incoming connections and handle them in some way, the steps are: first you listen(), then you accept().
- The incoming connections are going to wait in this queue until you accept() (explained later) them and this is the limit on how many can queue up.
- Again, as per usual, listen() returns -1 and sets errno on error.
- We need to call bind() before we call listen() or the kernel will have us listening on a random port.
- So if you're going to be listening for incoming connections, the sequence of system calls you'll make is something like this:

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

accept()

```
NAME
       accept() - accept a connection on a socket
SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>
       int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

TCP/IP 2
Advanced TCP/IP 3
Advanced TCP/IP 4
Advanced TCP/IP 5

- sockfd is the listen()ing socket descriptor.
- 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.
- accept() returns -1 and sets errno if an error occurs.
- Basically, after listen(), a server calls accept() to wait for the next client to connect. accept() will create a new socket to be used for I/O with the new client. The server then will continue to do further accepts with the original sockfd.
- When someone try to connect() to your machine on a port that you are listen()ing on, their connection will be queued up waiting to be accepted. You call accept() and you tell it to get the pending connection.
- It'll return to you a new socket file descriptor to use for this single connection.
- Then, you will have two socket file descriptors where the original one is still listening on your port and the newly created one is finally ready to send() and recv().
- The following is a program example that demonstrates the use of the previous functions.

```
[bodo@bakawali testsocket]$ cat test3.c
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/* the port users will be connecting to */
#define MYPORT 3440
/* how many pending connections queue will hold */
#define BACKLOG 10

int main()
{
/* listen on sock_fd, new connection on new_fd */
int sockfd, new_fd;
/* my address information, address where I run this program */
struct sockaddr_in my_addr;
/* remote address information */
struct sockaddr_in their_addr;
int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0);
if(sockfd == -1)
{
  perror("socket() error lol!");
```

```
    exit(1);
  }
  else
    printf("socket() is OK...\n");

  /* host byte order */
  my_addr.sin_family = AF_INET;
  /* short, network byte order */
  my_addr.sin_port = htons(MYPORT);
  /* auto-fill with my IP */
  my_addr.sin_addr.s_addr = INADDR_ANY;

  /* zero the rest of the struct */
  memset(&(my_addr.sin_zero), 0, 8);

  if(bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
  == -1)
  {
    perror("bind() error lol!");
    exit(1);
  }
  else
    printf("bind() is OK...\n");

  if(listen(sockfd, BACKLOG) == -1)
  {
    perror("listen() error lol!");
    exit(1);
  }
  else
    printf("listen() is OK...\n");

  /* ...other codes to read the received data... */

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

  if(new_fd == -1)
    perror("accept() error lol!");
  else
    printf("accept() is OK...\n");

  /*.....other codes.......*/

  close(new_fd);
  close(sockfd);
  return 0;
  }
```

```
[bodo@bakawali testsocket]$ gcc test3.c -o test3
[bodo@bakawali testsocket]$ ./test3
socket() is OK...
bind() is OK...
listen() is OK...
```

- 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.

send()

```
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 new one you got with accept()).

- msg is a pointer to the data you want to send.
- 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 = "I 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 and this might be less than the number you told it to send.
- 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.
- 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.

recv()

- 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 and len is the maximum length of the buffer.
- 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).
- If recv() return 0, this can mean only one thing that is 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.
- At this stage you can now pass data back and forth on stream sockets.
- These two functions send() and recv() are for communicating over stream sockets or connected datagram sockets.
- If you want to use regular unconnected datagram sockets (UDP), you need to use the sendto() and recvfrom().
- Or you can use more general, the normal file system functions, write() and read().

write()

```
NAME
       write() - write to a file descriptor
SYNOPSIS
       #include <unistd.h>
       ssize_t write(int fd, const void *buf, size_t count);
```

- Writes to files, devices, sockets etc.
- Normally data is copied to a system buffer and write occurs asynchronously.
- If buffers are full, write can block.

read()

```
NAME
       read() - read from a file descriptor
SYNOPSIS
       #include <unistd.h>
       ssize_t read(int fd, void *buf, size_t count);
```

- Reads from files, devices, sockets etc.
- If a socket has data available up to count bytes are read.
- If no data is available, the read blocks.
- If less than count bytes are available, read returns what it can without blocking.
- For UDP, data is read in whole or partial datagrams. If you read part of a datagram, the rest is discarded.

close() and **shutdown()**

```
NAME
       close() - close a file descriptor
SYNOPSIS
       #include <unistd.h>
       int close(int sockfd);
```

- 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.
- UNIX keeps a count of the number of uses for an open file or device.
- Close decrements the use count. If the use count reaches 0, it is closed.
- 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.
- The prototype:

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

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

  1. 0 – Further receives are disallowed.
  2. 1 – Further sends are disallowed.
  3. 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 change its usability.
- To free a socket descriptor, you need to use close().

*Continue on next Module…*More in-depth discussion about TCP/IP suite is given in Advanced TCP/IP Programming.

Further reading and digging:

1. Check the best selling C/C++, Networking, Linux and Open Source books at Amazon.com.
2. Protocol sequence diagram examples.
3. Another site for protocols information.
4. RFCs.
5. GCC, GDB and other related tools.

| Winsock & .NET | Winsock | < TCP/IP Socket Programming Interfaces (APIs) | Linux Socket Index | More GNU C Socket APIs & Library > |