eecs489 COMPUTER NETWORKS

Lecture 3:
Sockets Programming (TCP Server)

Initialize (TCP Server `bind` addr)

```
int sd;
struct sockaddr_in sin;

if ((sd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
  perror("opening TCP socket");
  abort();
}

memset(&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = htons(server_port);

if (bind(sd, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
  perror(“bind");
  printf("Cannot bind socket to address\n");
  abort();
}
```

Initialize (Server `bind` addr)

`bind()` used to “label” a socket with an IP address and/or port#

- why do we need to label a socket with a port#?
- must each service have a well-known port?
- why do we need to label a socket with IP address?
- what if we want to receive packets from all network interfaces of the server machine?
- why not always receive from all interfaces?
- what defines a connection?
- mainly used by TCP, but may be used by UDP also

If we call `bind()` with server port `0`, the kernel will assign an ephemeral port# to the socket

Initialize (TCP Server `listen`)

```
if (listen(sd, qlen) < 0) {
  perror(“error listening");
  abort();
}
```

- specifies max number of pending TCP connections allowed to wait to be accepted (by `accept()`)

Establish (TCP Server `accept`)

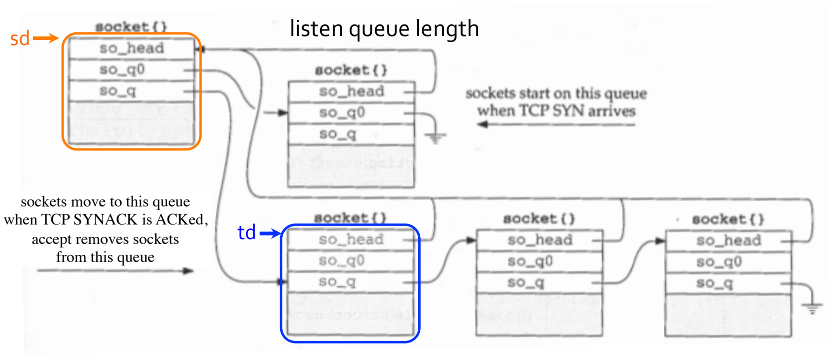
```
int addr_len = sizeof(addr);
int td;

td = accept(sd, (struct sockaddr *) &addr,
  &addr_len);

if (td < 0) {
  perror("error accepting connection");
  abort();
}
```

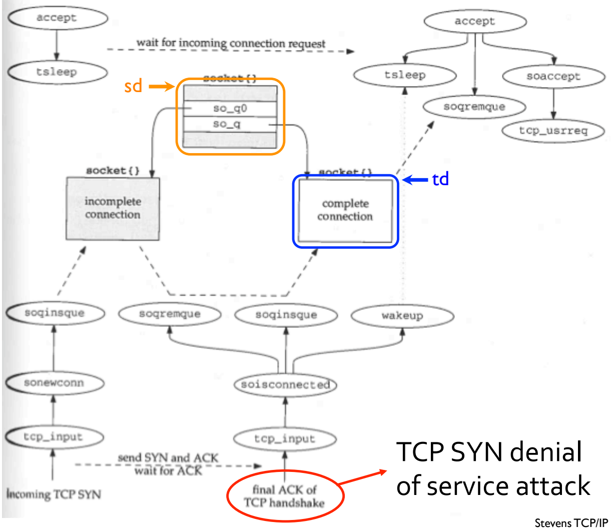
- waits for incoming client connection
- returns a connected socket ← different from the `listen`ed to socket

Socket Connection Queues



Stevens TCP/IP Illustrated v. 2 pp. 441, 461

Socket Connection Queues



TCP SYN denial of service attack

Stevens TCP/IP Illustrated v. 2 pp. 441, 461

Socket API Design Questions

Why separate `listen()` and `accept()`?

Why separate `bind()` and `listen()`?

Receiving Data Stream (TCP Server)

```
int
receive_packets(char *buffer, int blen, int *bytes)
{
  int left = blen - *bytes;
  received = recv(td, buffer + *bytes, left, 0);
  if (received > 0) *bytes += received;
  return received;
}
```

- returns the number of bytes actually received
  - 0 if connection is closed, −1 on error
- if non-blocking: −1 if no data, with errno set to EAGAIN (or EWOULDBLOCK)
- must loop to ensure all data is received
  - (in this example, receive_packets() itself is called in a loop, see later slide)

Data Stream vs. Datagram

SOCK_STREAM treats data as one stream, not chopped up into chunks (above the transport layer!)

Calls to recv() simply return however much data is available or requested (size of provided buffer)

To receive requested amount may require multiple calls

How do you know you have received everything sent?

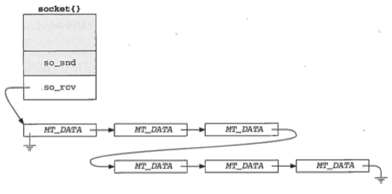


Figure 16.36 so_rcv buffer for TCP.

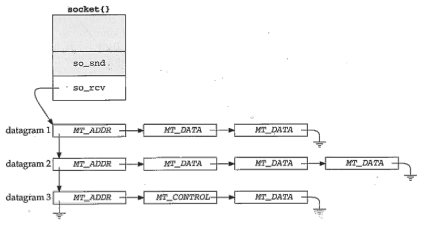


Figure 16.35 UDP receive buffer consisting of three datagrams.

Stevens

MSG_PEEK

```
recv(sd, buffer1, 1, MSG_PEEK);
```

Return data from the beginning of the receive queue without removing that data from the queue

A subsequent call to recv() will return the same data:

```
recv(sd, buffer2, 1, 0);
```

buffer1 and buffer2 contain the same byte (if the byte was there by the first call)

When is MSG_PEEK useful?

Connection close

Called by both client and server

close() marks socket unusable

- actual tear down depends on TCP:
  - when a previous binding has closed, but TCP hasn't released the port, TCP is said to be in TIME_WAIT state
- socket option SO_LINGER can be used to specify whether close() should
  - return immediately,
  - wait for termination, or
  - abort connection

Socket Options

The APIs `getsockopt()` and `setsockopt()` are used to query and set socket options

Some useful options:

- `SO_LINGER`
- `SO_RCVBUF` and `SO_SNDBUF` used to set buffer sizes
- `SO_KEEPALIVE` tells server to ping client periodically
- `SO_REUSEADDR` and `SO_REUSEPORT`

SO_REUSEADDR

When TCP is in `TIME_WAIT` state and a socket tries to `bind` to the same address and port:
`bind: Address already in use`

`SO_REUSEADDR` allows the `bind` to proceed

```
int sd;
int optval = 1;
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("opening TCP socket");
  abort();
}
if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
               &optval, sizeof(optval)) <0) {
  perror(“reuse address");
  abort();
}
```

SO_REUSEPORT

Cases when we want to bind multiple sockets to the same address and port# outside `TIME_WAIT` state:

1. peers accepting and initiating connections on the same port#, and
2. IP multicast applications

Implementation:

- on Mac OS X and Winsocks, `SO_REUSEADDR` is sufficient but only if all sockets of the same port have set the option
- on Linux, `SO_REUSEPORT` must be used; again, all sockets of the same port must set this option
- Mac OS X recognizes `SO_REUSEPORT`, Winsocks doesn't

Multiple I/O Streams

Where does a process get its input from?

- device (keyboard, mouse, touch, mic, sensors)
- network sockets

Input arrives asynchronously, a process doesn't know when its data will arrive

Alternatives for handling asynchronous I/O:

- multithreading: each thread handles one I/O stream (482)
- I/O multiplexing: a single thread handles multiple I/O streams

I/O Multiplexing

Two stages of blocking:

1. waiting for device availability (e.g., queueing for copy machine)
2. waiting for job completion (e.g., making copies)

Flavors:

- blocking I/O (default): wait in line, wait while copies are made
  - put process to sleep until I/O is ready
  - blocking for device availability and I/O completion
  - by calling `select()` or `poll()`
- non-blocking I/O: continue to check the line, wait while copying
  - only non-blocking during checks for device availability
  - by manual polling or signal driven (not covered)
  - I/O completion (device use) is still blocking
- asynchronous I/O: give job to copy shop, delivered when ready
  - process is notified when I/O is completed (not covered)

Non-Blocking I/O: Polling

```
int nonblock=1;
if (ioctl(sd, FIONBIO, &nonblock) < 0) {
  perror("ioctl(FIONBIO)");
  abort();
}
while (1) {
  // both sd and stdin can be read from,
  // without one blocking the other
  if (receive_packets(buffer, blen, &bytes)
      != /* full_amount or closed */) {
      break;
  }
  if (read_stdin(in_buf, in_len, &in_bytes)
      != 0) {
      break;
  }
}
```

set socket option non-blocking

get socket data

get user input

Why is this code not efficient?

Blocking I/O: `select()`

`select(maxfd, readset, writeset, exceptset, timeout)`

- waits on multiple file descriptors/sockets or timeout
- application does not consume CPU cycles while waiting
- `maxfd` is the maximum file descriptor number `+1`
  - if you have only one descriptor, number `5`, `maxfd` is `6`
- descriptors provided as bitmask
  - use `FD_ZERO`, `FD_SET`, `FD_ISSET`, and `FD_CLR` to manipulate the bitmasks
- ready descriptors returned on the same bitmask
- returns as soon as one of the specified sockets is ready to be read or written, or an error occurred, or timeout exceeded
  - returns # of ready sockets, `-1` on error, `0` if timed out and no device is ready (what for?)

Blocking I/O: `select()`

```
fd_set read_set;
struct timeval time_out;
while (1) {
  FD_ZERO(read_set);
  FD_SET(stdin, read_set); /* not on Windows */
  FD_SET(sd, read_set);
  time_out.tv_usec = 100000; time_out.tv_sec = 0;

  err = select(MAX(stdin, sd) + 1, &read_set,
               NULL, NULL, &time_out);

  if (err < 0) {
       perror ("select");
       abort ();
  } else if (err > 0) {
       if (FD_ISSET(sd, read_set))      // get socket data
           if (receive_packets(buffer, blen, &bytes)
                != /* full_amount or closed */)
               break;
       if (FD_ISSET(stdin, read_set)) // get user input
           if (read_user(in_buf, in_len, &in_bytes) != 0)
               break;
  } else {
       /* process time out */
  }
}
```

set up parameters for `select()`

run `select()`

interpret result

MSG_WAITALL

```
recv(sd, buffer, len, MSG_WAITALL);
```

Blocks until `len` amount of data received or process interrupted by a signal or an error or disconnect occurs (no effect on non-blocking socket)

Name three disadvantages of using `MSG_WAITALL`? Or, why is `recv()` not designed to block until the full `len` amount of data has arrived?

A blocking socket may similarly be used in non-blocking mode per-call with `MSG_DONTWAIT`, but only on Linux (≥ 2.2) and Mac OS X, not Winsocks

Use of both is discouraged

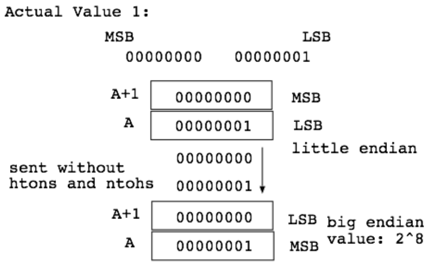
Byte Ordering Problem

```
struct sockaddr_in sin;

memset(&sin, 0, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = IN_ADDR;
sin.sin_port = htons(server_port);
```

Little-endian:
Most Significant Byte (MSB) in high address (sent/arrives later) (Intel x86)

Big-endian: MSB in low address (sent/arrives first)



Bi-endian: switchable endians (ARM, SPARC V9)

Byte Ordering Solution

To ensure interoperability, ALWAYS translate integers (`short`, `long`, `int`, `uint16`, `uint32`) to/from "network byte order" before/after transmission

Use these macros (note: 32-bit only):
`htons()`: host to network short
`htonl()`: host to network long
`ntohs()`: network to host short
`ntohl()`: network to host long

Do we have to be concerned about byte ordering for `char` type?

How about `float` and `double`? See XDR (RFC4506)

Naming and Addressing

Example fully-qualified domain name (FQDN) in character string: www.eecs.umich.edu

Its IP address in dotted-decimal (dd) character string: `141.212.113.110`

Its IP address in 32-bit binary:
`10001101 11010100 01110001 01101110`

Why do we need names?
Why not just use addresses directly?

Why do we need addresses in addition to names?

Name and Address Manipulation

APIs to map name to/from address:

- FQDN to binary: `gethostbyname()`
- binary to FQDN: `gethostbyaddress()`
  - `gethostbyname()` and `gethostbyaddr()` both return `struct hostent` that contains both FQDN & binary

APIs to change representation:

- dd to binary: `inet_aton()`
- binary to dd: `inet_ntoa()`

To map FQDN to dd:
`gethostbyname()` then `inet_ntoa()`

Name and Address Manipulation

Other useful APIs:

- `gethostname()`: returns FQDN of current host
- `getsockname()`: returns IP address bound to socket (in binary): used when address and/or port is not specified (`INADDR_ANY`), to find out the actual address and/or port in use
- `getpeername()`: returns IP address of peer (in binary)

RTFM: `http://web.eecs.umich.edu/~sugih/courses/eecs489/links.html`