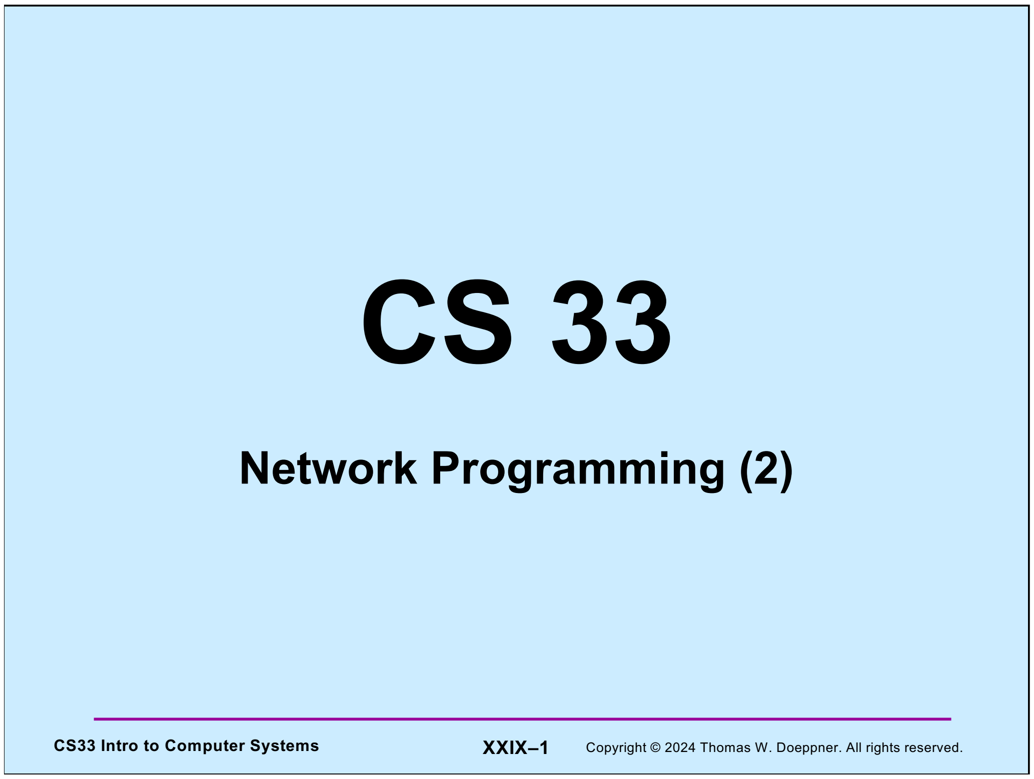


CS 33

Network Programming (2)

The source code used in this lecture, as well as some additional related source code, is on the course web page.

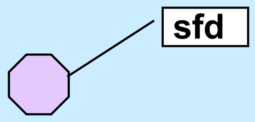
Reliable Communication

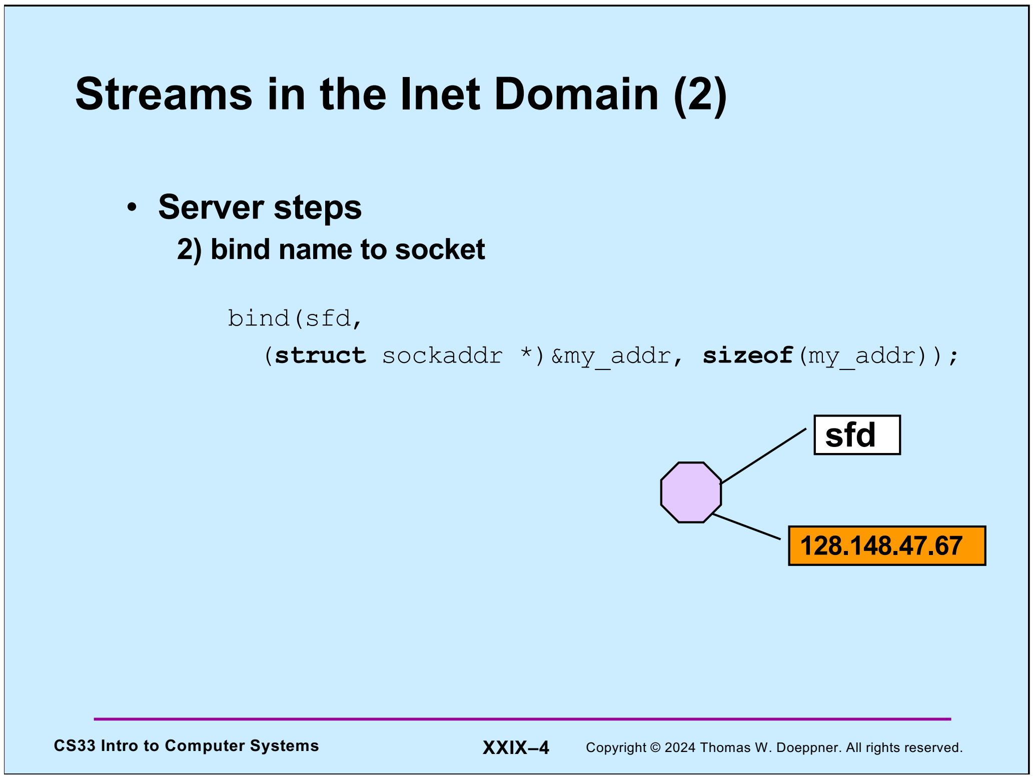
- The promise …
  - what is sent is received
  - order is preserved
- Set-up is required
  - two parties agree to communicate
  - within the implementation of the protocol:
    - » each side keeps track of what is sent, what is received
    - » received data is acknowledged
    - » unack'd data is re-sent
- The standard scenario
  - server receives connection requests
  - client makes connection requests

Streams in the Inet Domain (1)

- **Server steps**
  - **1) create socket**

```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```

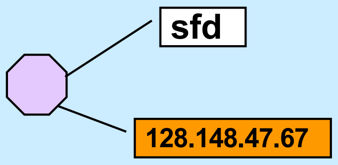


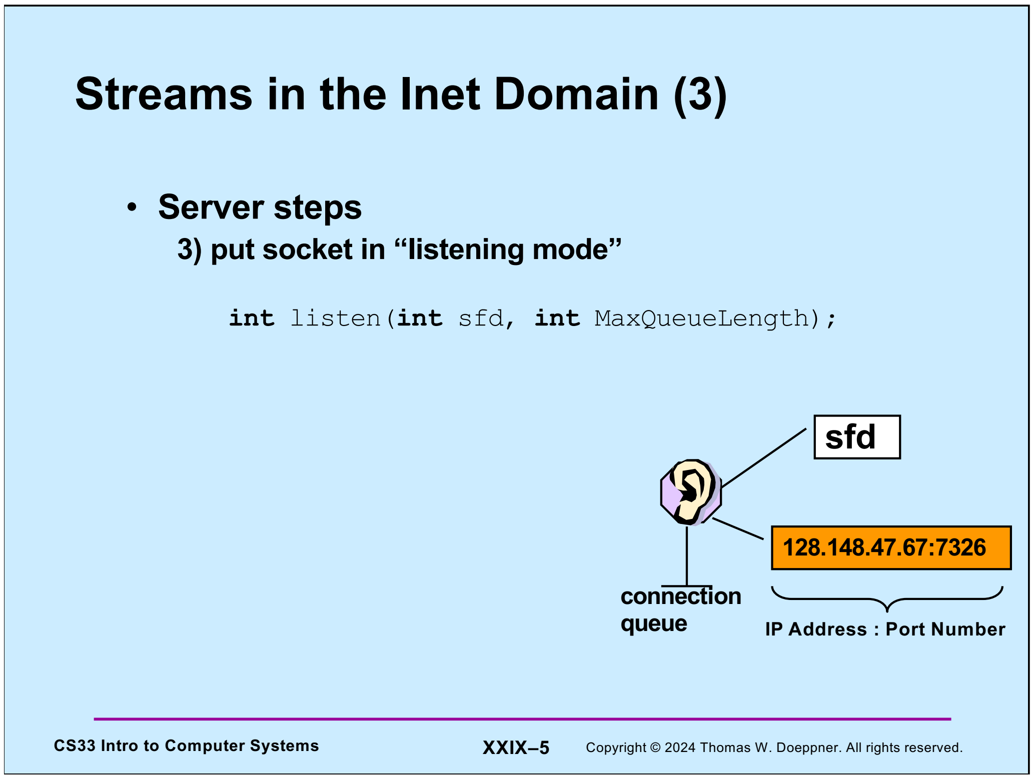


Streams in the Inet Domain (2)

- **Server steps**
  - **2) bind name to socket**

```
bind(sfd,
   (struct sockaddr *)&my_addr, sizeof(my_addr));
```

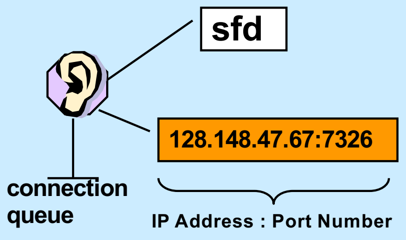




Streams in the Inet Domain (3)

- **Server steps**
  - 3) put socket in "listening mode"

```
int listen(int sfd, int MaxQueueLength);
```

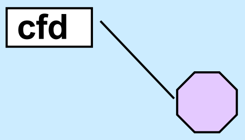


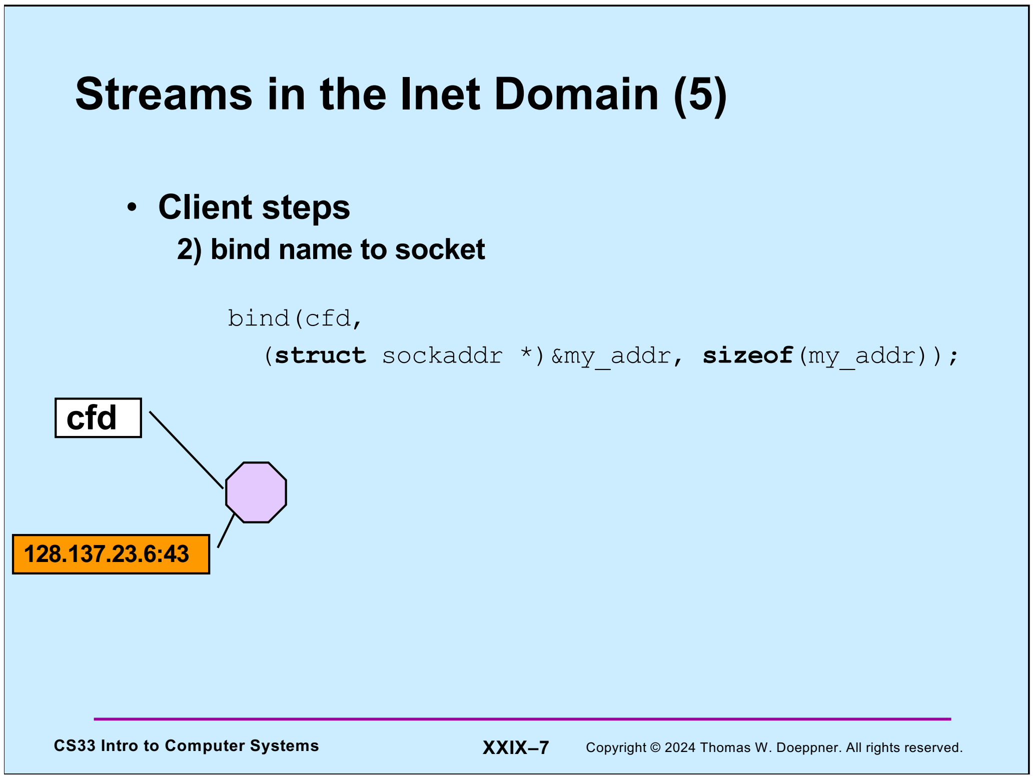
The **listen** system call tells the OS that the process would like to receive connections from clients via the indicated socket. The **MaxQueueLength** argument is the maximum number of connections that may be queued up, waiting to be accepted. Its maximum value is in /proc/sys/net/core/somaxconn (and is currently 128).

Streams in the Inet Domain (4)

- **Client steps**
  - **1) create socket**

```
cfd = socket(AF_INET, SOCK_STREAM, 0);
```

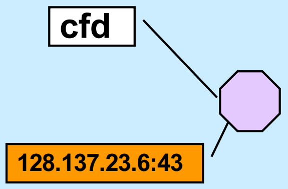




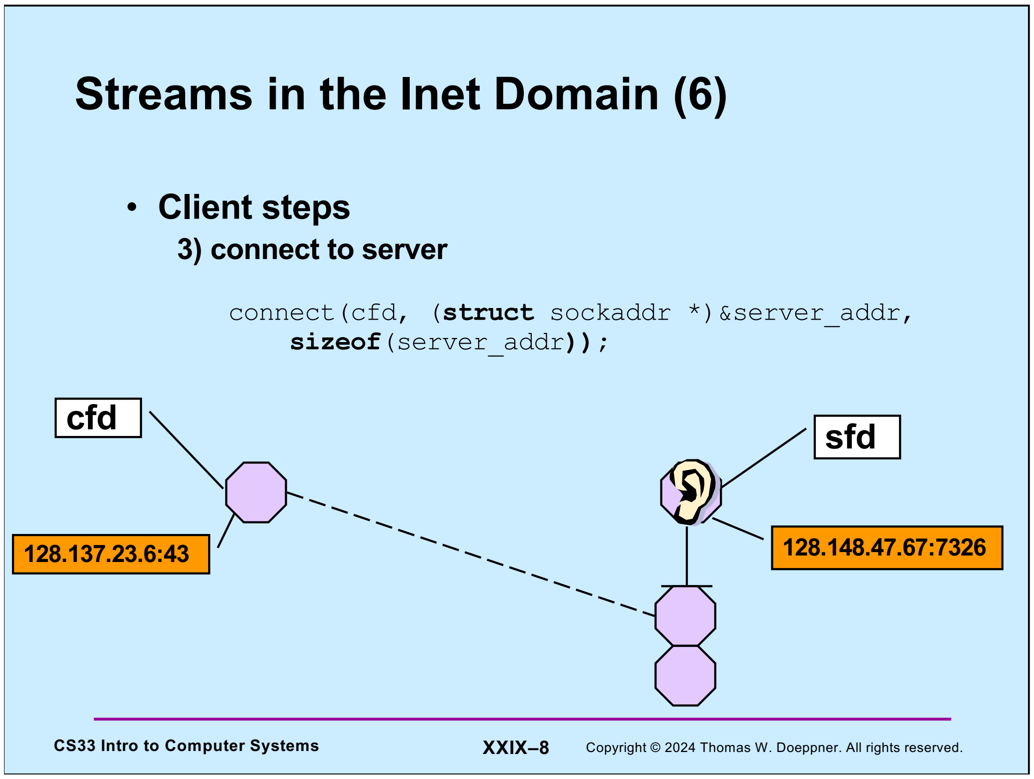
Streams in the Inet Domain (5)

- **Client steps**
  - **2) bind name to socket**

```
bind(cfd,
  (struct sockaddr *)&my_addr, sizeof(my_addr));
```



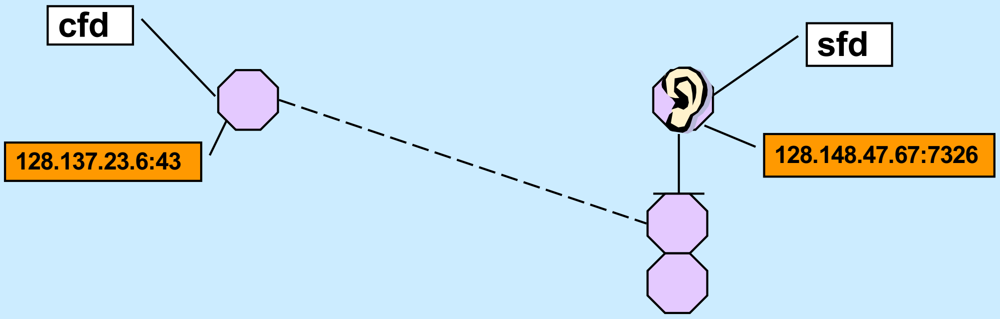
This step is optional – if not done, the OS does it automatically, supplying some available port number.



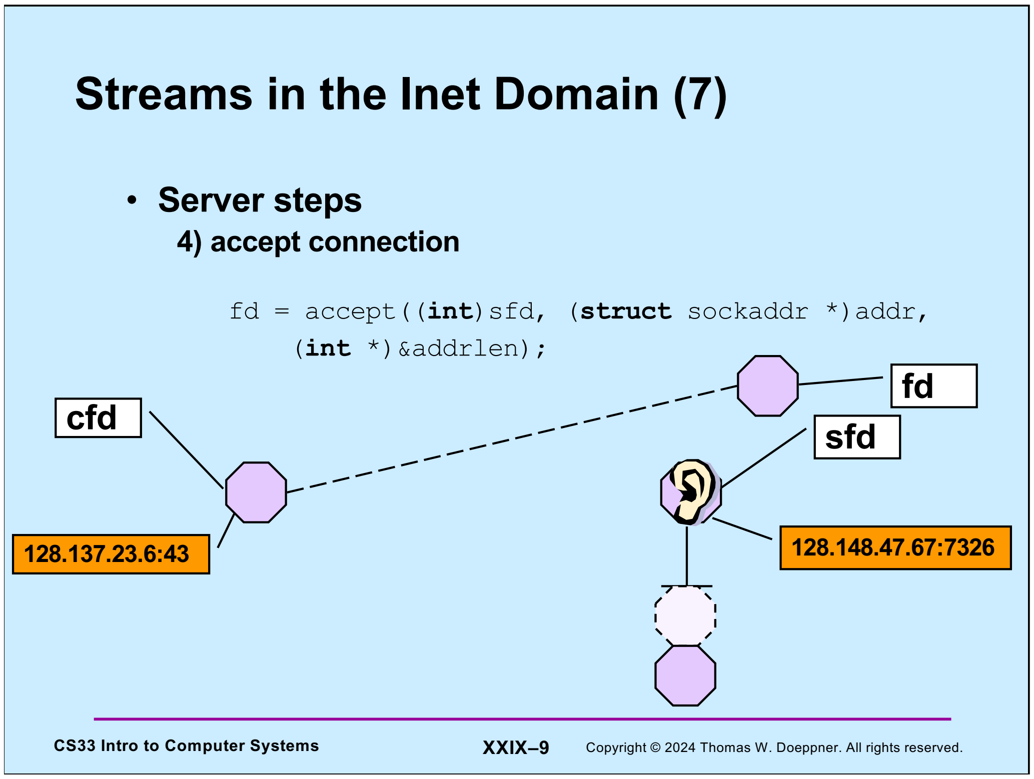
Streams in the Inet Domain (6)

- **Client steps**
  - 3) connect to server

```
connect(cfd, (struct sockaddr *)&server_addr,
    sizeof(server_addr));
```



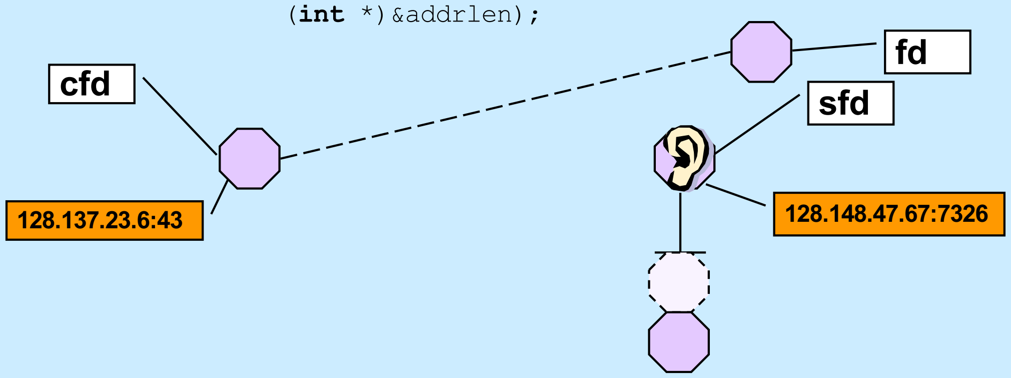
The client issues the **connect** system call to initiate a connection with the server. The first argument is a file descriptor referring to the client's socket. Ultimately this socket will be connected to a socket on the server. Behind the scenes the client OS communicates with the server OS via a protocol-specific exchange of messages. Eventually a connection is established and a new socket is created on the server to represent its end of the connection. This socket is queued on the server's listening socket, where it stays until the server process accepts the connection (as shown in the next slide).



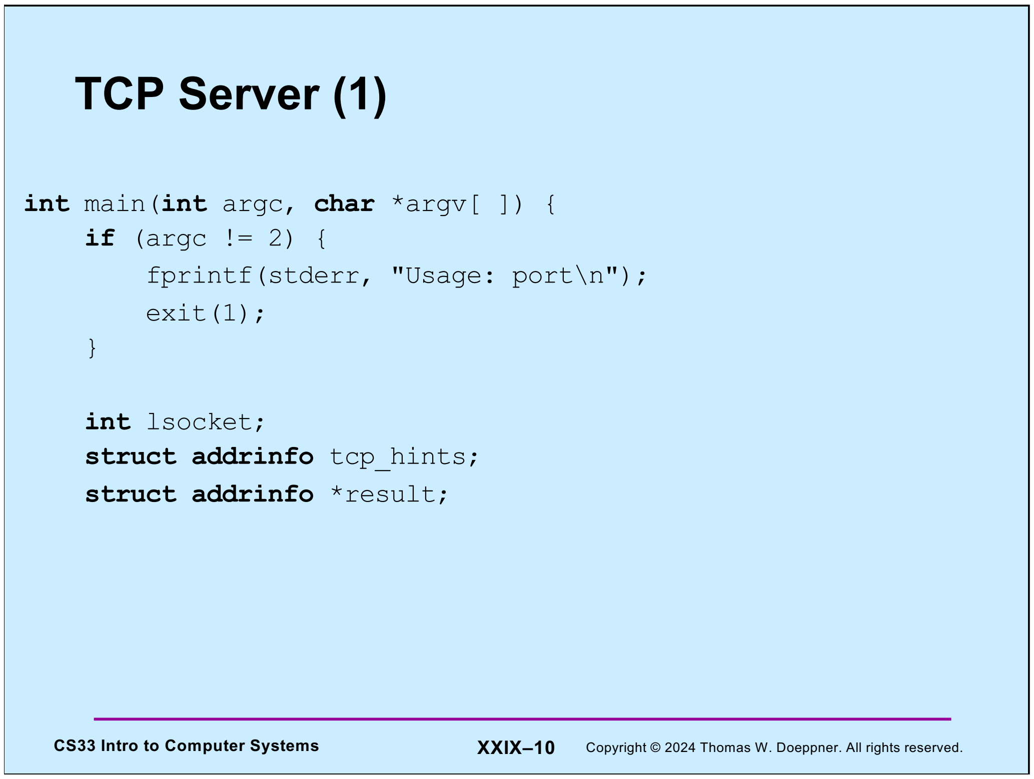
Streams in the Inet Domain (7)

- **Server steps**
  - **4) accept connection**

```
fd = accept((int)sfd, (struct sockaddr *)addr,
    (int *)&addrlen);
```



The server process issues an **accept** system which waits if necessary for a connected socked to appear on the listening socket's queue, then pulls the first such socket from the queue. This socket is the server end of a connection from a client. A file descriptor is returned that refers to that socket, allowing the process to now communicate with the client.

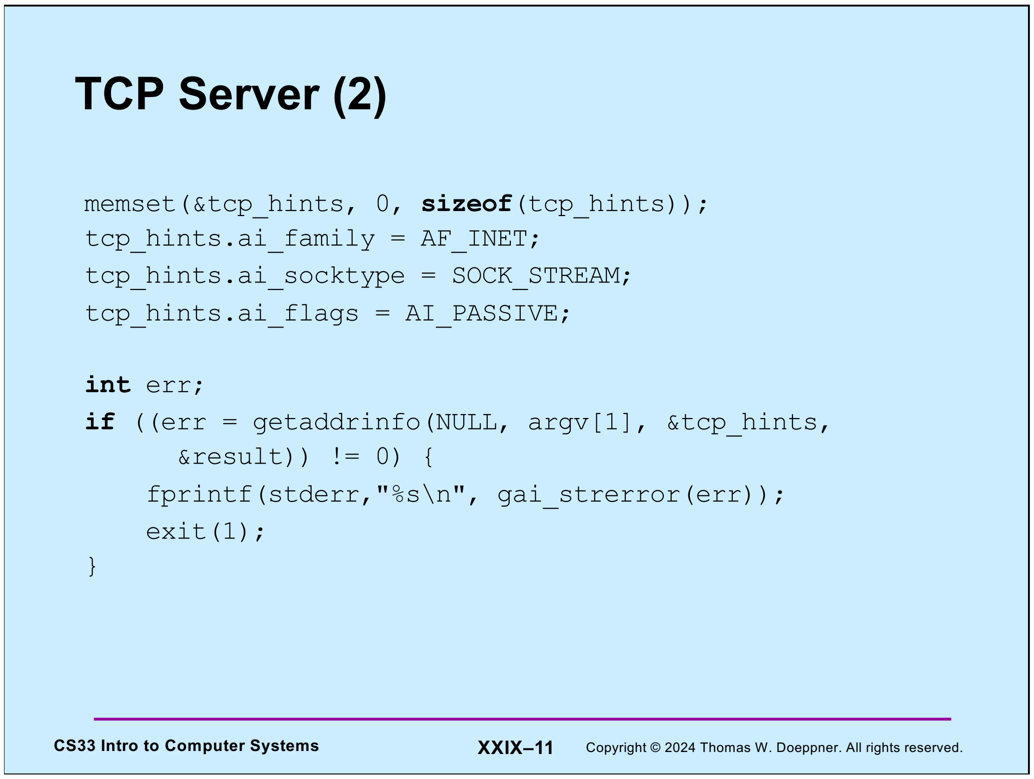


TCP Server (1)

```
int main(int argc, char *argv[ ]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: port\n");
        exit(1);
    }

    int lsocket;
    struct addrinfo tcp_hints;
    struct addrinfo *result;
```

We begin looking at a TCP example similar to our UDP example. Clients will contact our server, which prints everything its clients send to it.

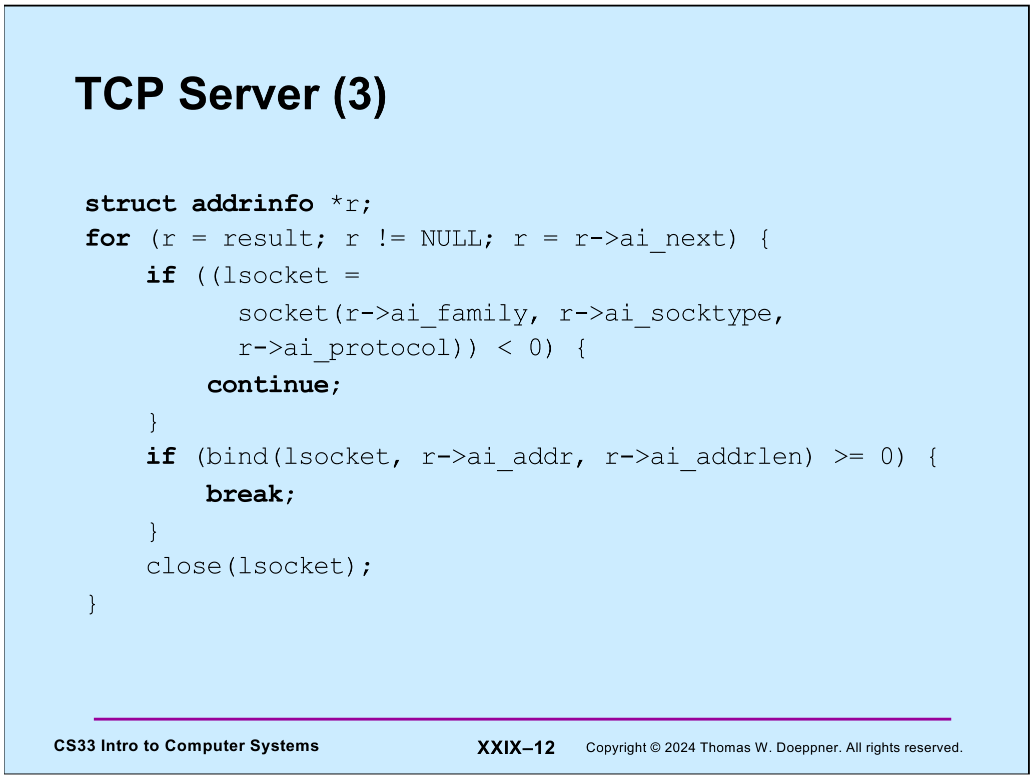


TCP Server (2)

```
memset(&tcp_hints, 0, sizeof(tcp_hints));
tcp_hints.ai_family = AF_INET;
tcp_hints.ai_socktype = SOCK_STREAM;
tcp_hints.ai_flags = AI_PASSIVE;

int err;
if ((err = getaddrinfo(NULL, argv[1], &tcp_hints,
      &result)) != 0) {
    fprintf(stderr,"%s\n", gai_strerror(err));
    exit(1);
}
```

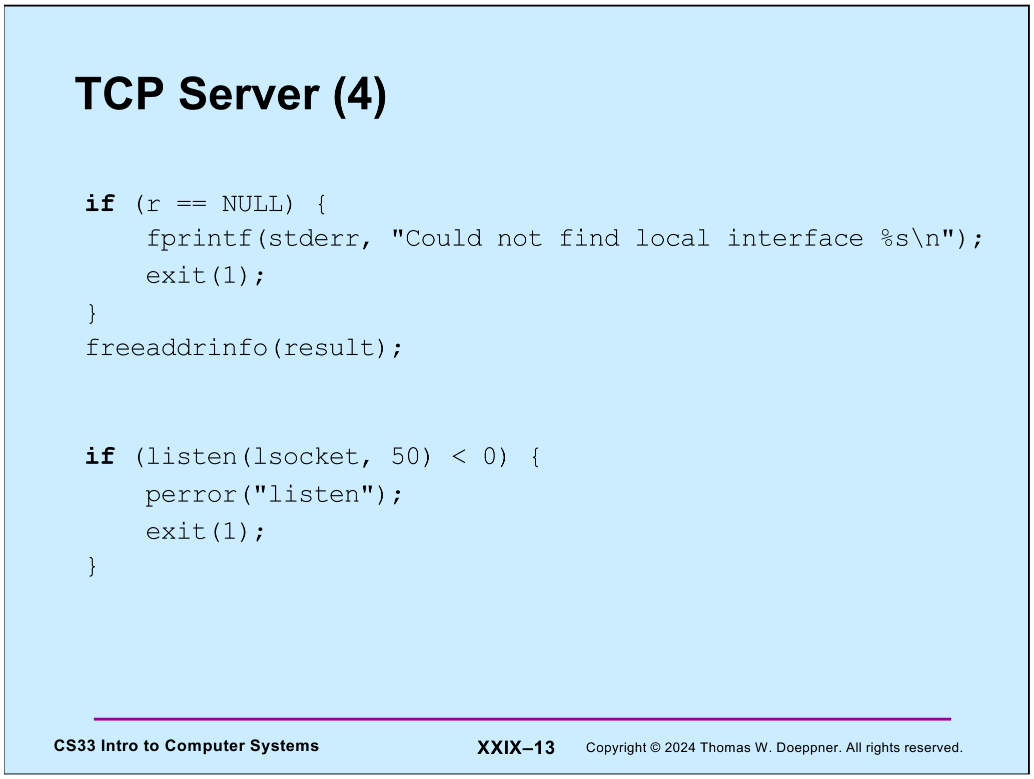
The server starts by using **getaddrinfo** to obtain information about its interfaces. Via **tcp_hints**, we request information about IPv4 (AF_INET) interfaces supporting TCP (SOCK_STREAM). The value to which **ai_flags** is set (AI_PASSIVE) indicates that our socket will be put into listening mode and its address will be set to allow it to receive connections on any network it's attached to.



TCP Server (3)

```
struct addrinfo *r;
for (r = result; r != NULL; r = r->ai_next) {
    if ((lsocket =
          socket(r->ai_family, r->ai_socktype,
          r->ai_protocol)) < 0) {
        continue;
    }
    if (bind(lsocket, r->ai_addr, r->ai_addrlen) >= 0) {
        break;
    }
    close(lsocket);
}
```

Here we look at the list of **addrinfo** structures returned by **getaddrinfo** and use the first for which we can create a socket and bind to (as usual with this, it will probably be the first and only item in the list).



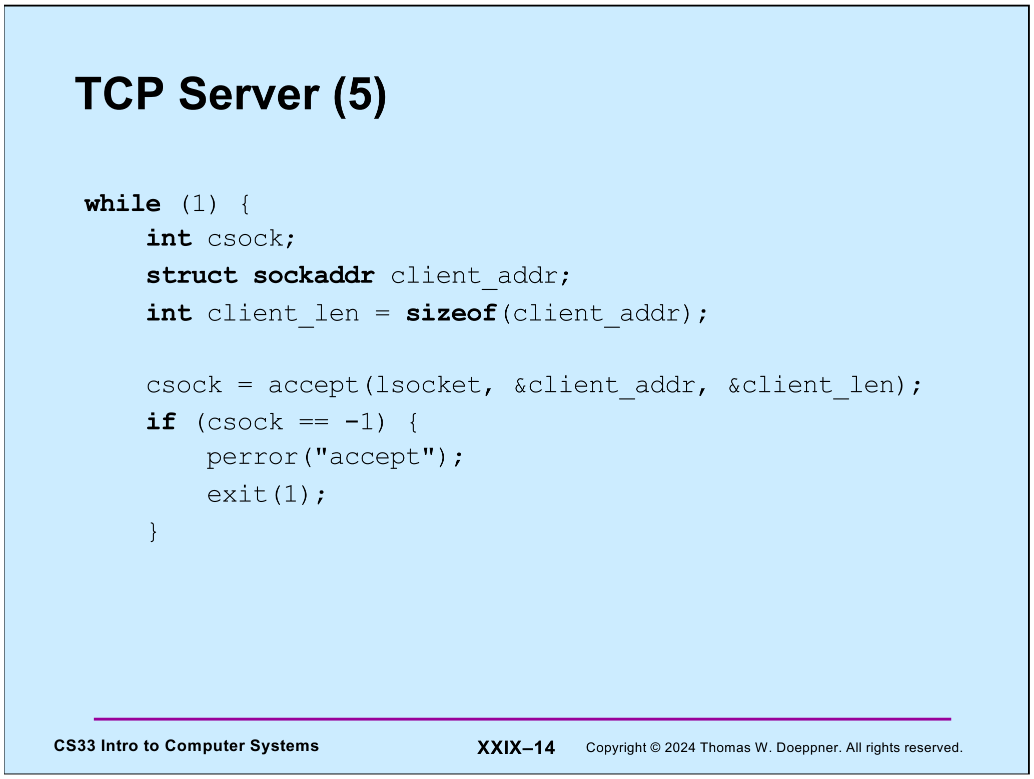
TCP Server (4)

```
if (r == NULL) {
    fprintf(stderr, "Could not find local interface %s\n");
    exit(1);
}
freeaddrinfo(result);


if (listen(lsocket, 50) < 0) {
    perror("listen");
    exit(1);
}
```

We check to make sure we found a suitable local address. Assuming we did, we free list of addresses, since we don't need them anymore.

Now that we have a socket, we put it in listening mode, indicating a maximum queue length of 50 (an arbitrarily chosen value).

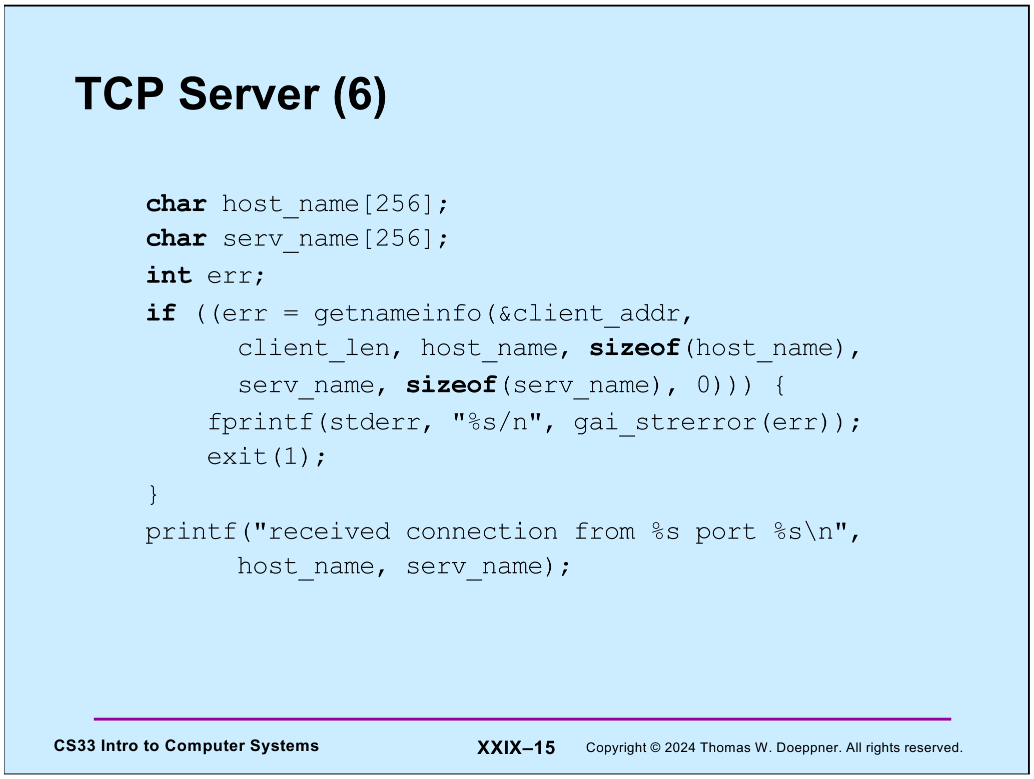


TCP Server (5)

```
while (1) {
    int csock;
    struct sockaddr client_addr;
    int client_len = sizeof(client_addr);

    csock = accept(lsocket, &client_addr, &client_len);
    if (csock == -1) {
        perror("accept");
        exit(1);
    }
```

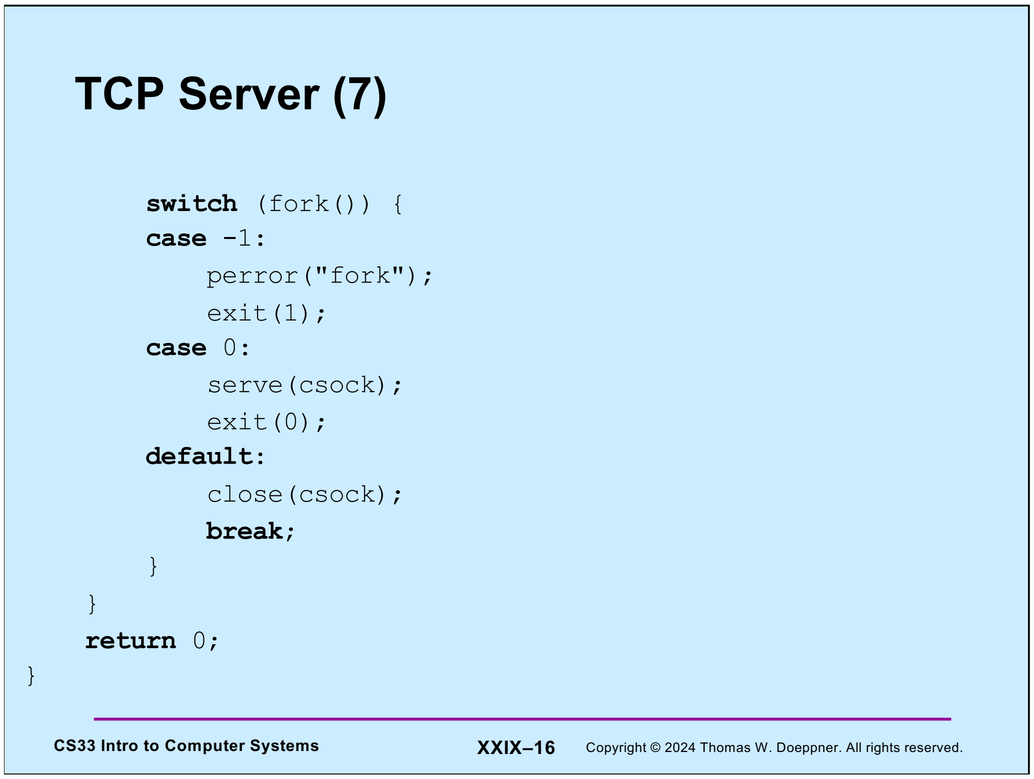
The server now begins a loop, accepting incoming connection requests from clients. Each time *accept* returns (assuming no errors), we have a file descriptor (**csock**) for the new client connection.



TCP Server (6)

```
char host_name[256];
char serv_name[256];
int err;
if ((err = getnameinfo(&client_addr,
      client_len, host_name, sizeof(host_name),
      serv_name, sizeof(serv_name), 0))) {
    fprintf(stderr, "%s/n", gai_strerror(err));
    exit(1);
}
printf("received connection from %s port %s\n",
      host_name, serv_name);
```

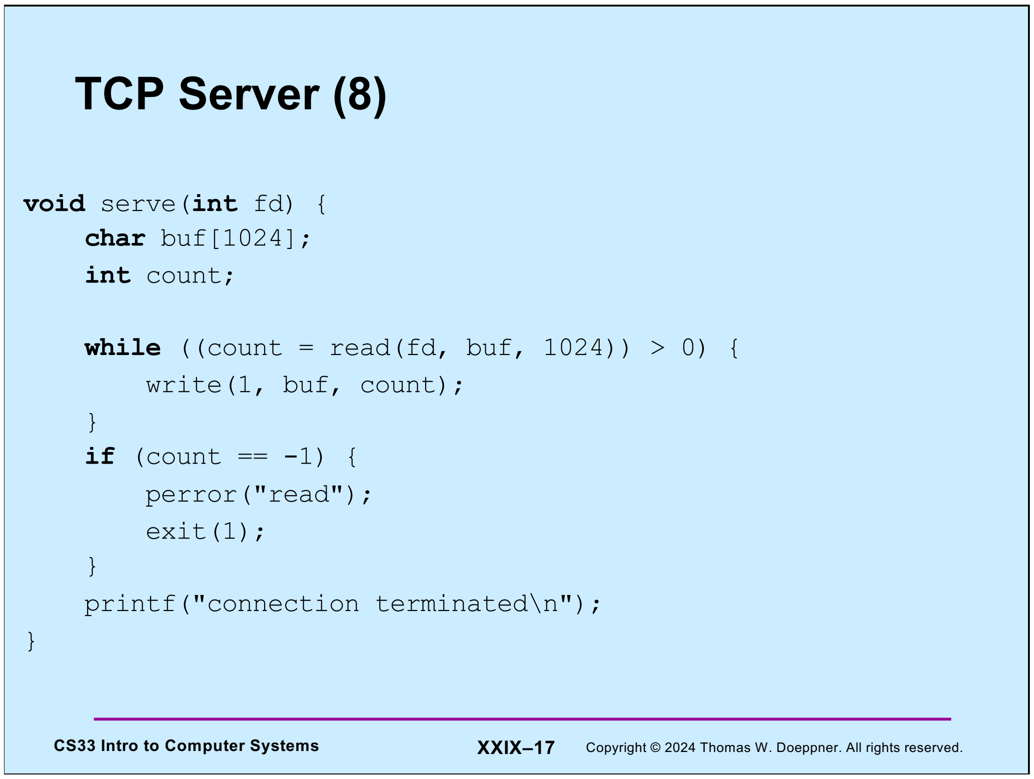
We figure how who the client is, based on the information returned by accept. We use **getnameinfo** to decode the host name and the service name (port number). Note the use of **gai_strerror** to deal with errors.



TCP Server (7)

```
        switch (fork()) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            serve(csock);
            exit(0);
        default:
            close(csock);
            break;
        }
    }
    return 0;
}
```

The server, having just received a connection from the client, creates a new process to handle that client's connection. The new (child) process calls serve, passing it the file descriptor for the connected socket. The parent has no further use for that file descriptor, so it closes it.

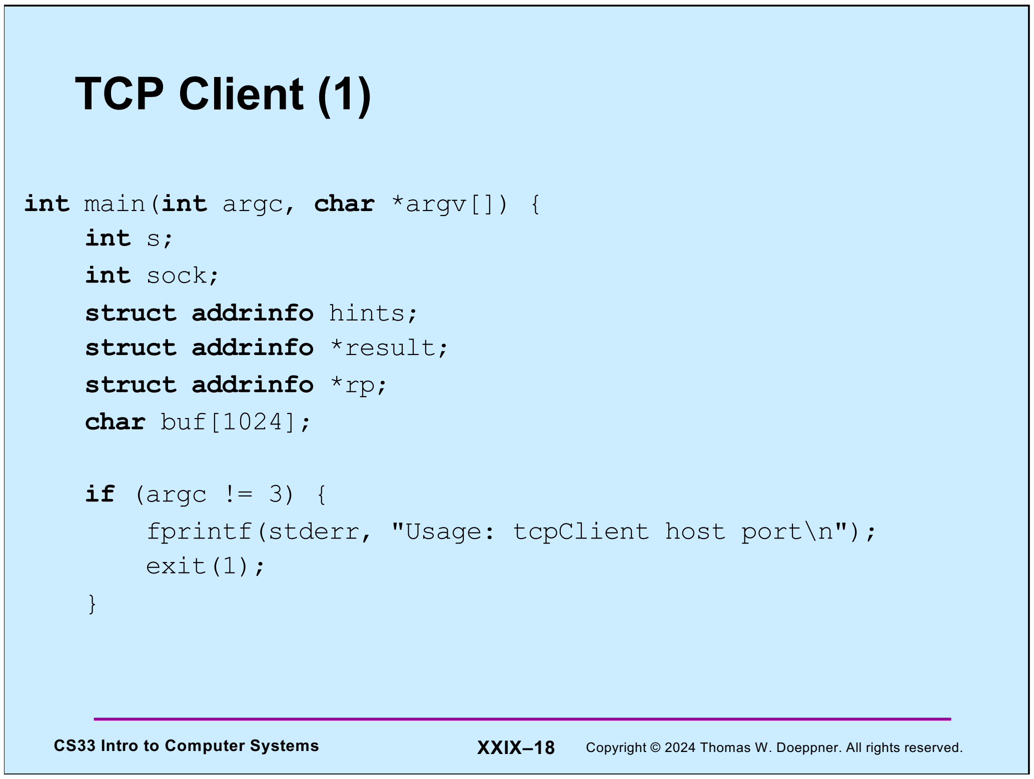


TCP Server (8)

```
void serve(int fd) {
    char buf[1024];
    int count;

    while ((count = read(fd, buf, 1024)) > 0) {
        write(1, buf, count);
    }
    if (count == -1) {
        perror("read");
        exit(1);
    }
    printf("connection terminated\n");
}
```

Finally, we have the *serve* function, which reads incoming data from the client and write it to file descriptor 1.

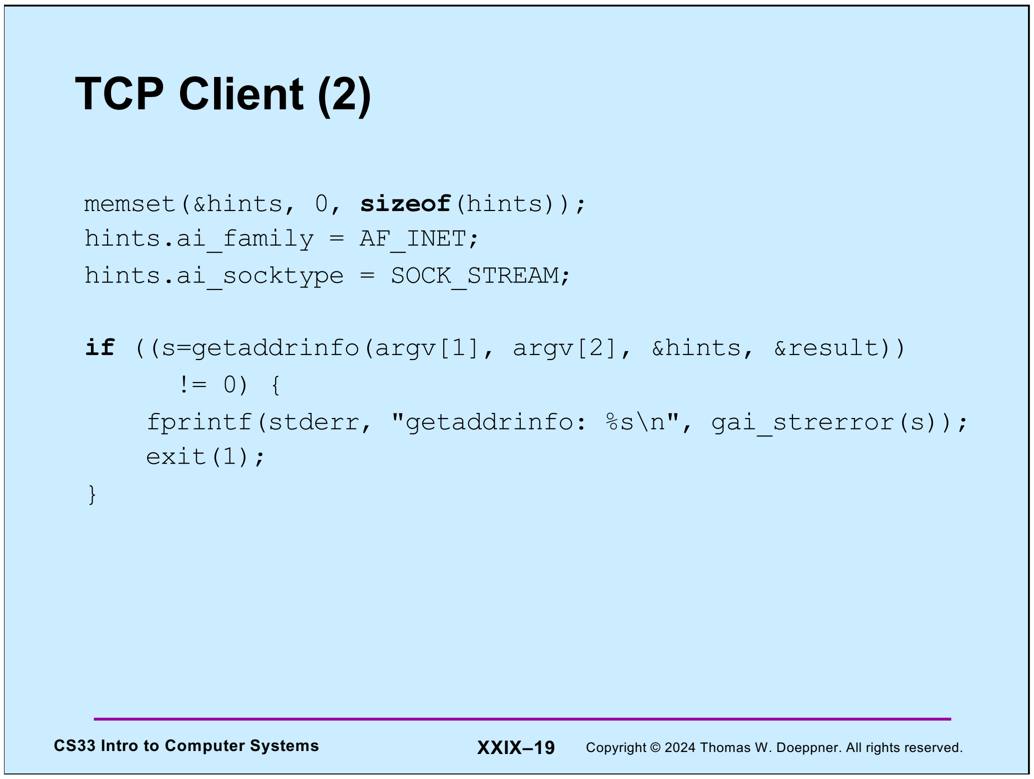


TCP Client (1)

```
int main(int argc, char *argv[]) {
    int s;
    int sock;
    struct addrinfo hints;
    struct addrinfo *result;
    struct addrinfo *rp;
    char buf[1024];

    if (argc != 3) {
        fprintf(stderr, "Usage: tcpClient host port\n");
        exit(1);
    }
```

And lastly, we have the code for our TCP client.

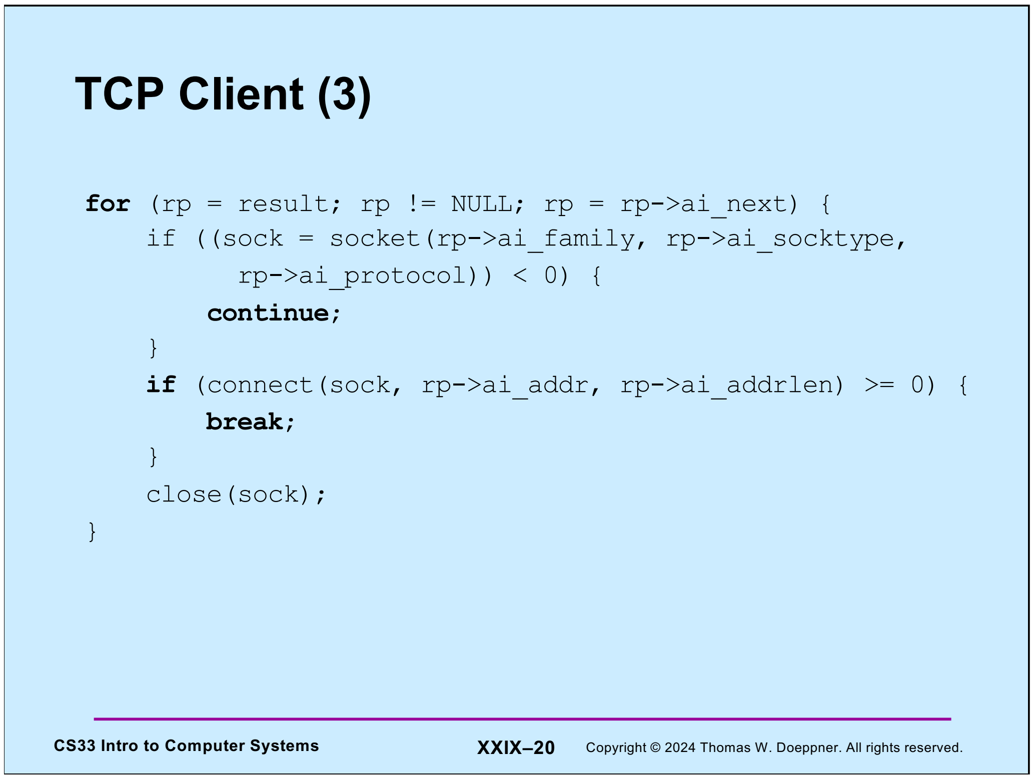


TCP Client (2)

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if ((s=getaddrinfo(argv[1], argv[2], &hints, &result))
      != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(1);
}
```

The client begins by looking up, via **getaddrinfo**, possible addresses for the server.

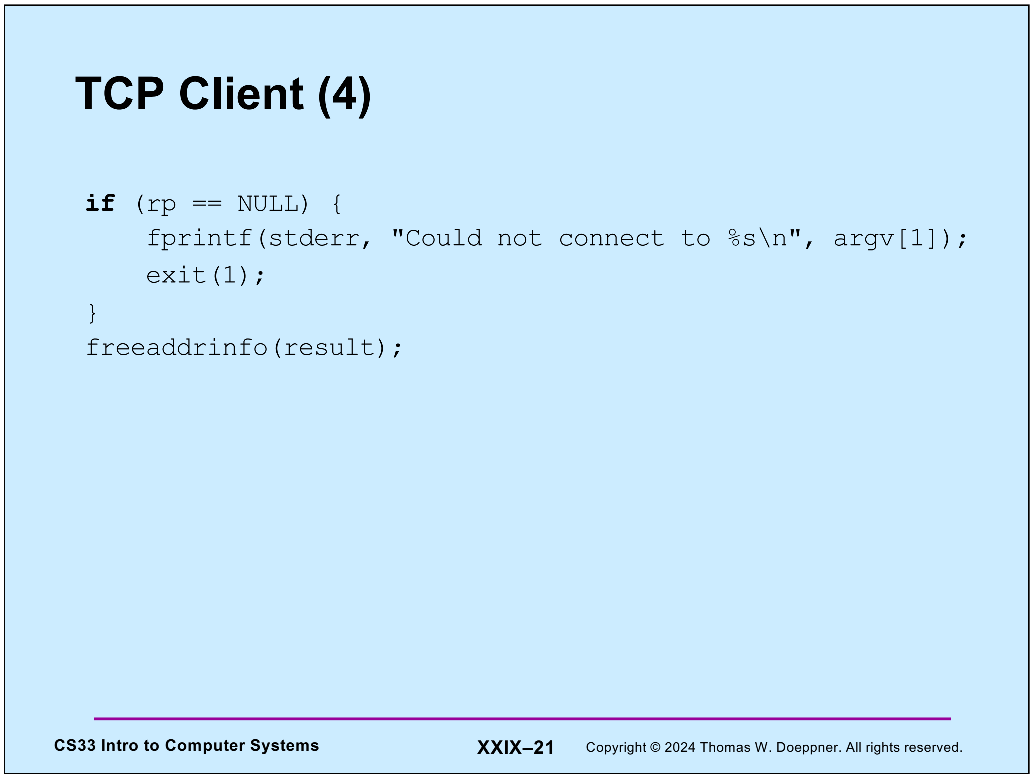


TCP Client (3)

```
for (rp = result; rp != NULL; rp = rp->ai_next) {
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
          rp->ai_protocol)) < 0) {
        continue;
    }
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) >= 0) {
        break;
    }
    close(sock);
}
```

The client chooses an address for which it can create a socket and connect to. Thus, if this code completes successfully, the client is now connected to the server via *sock*.

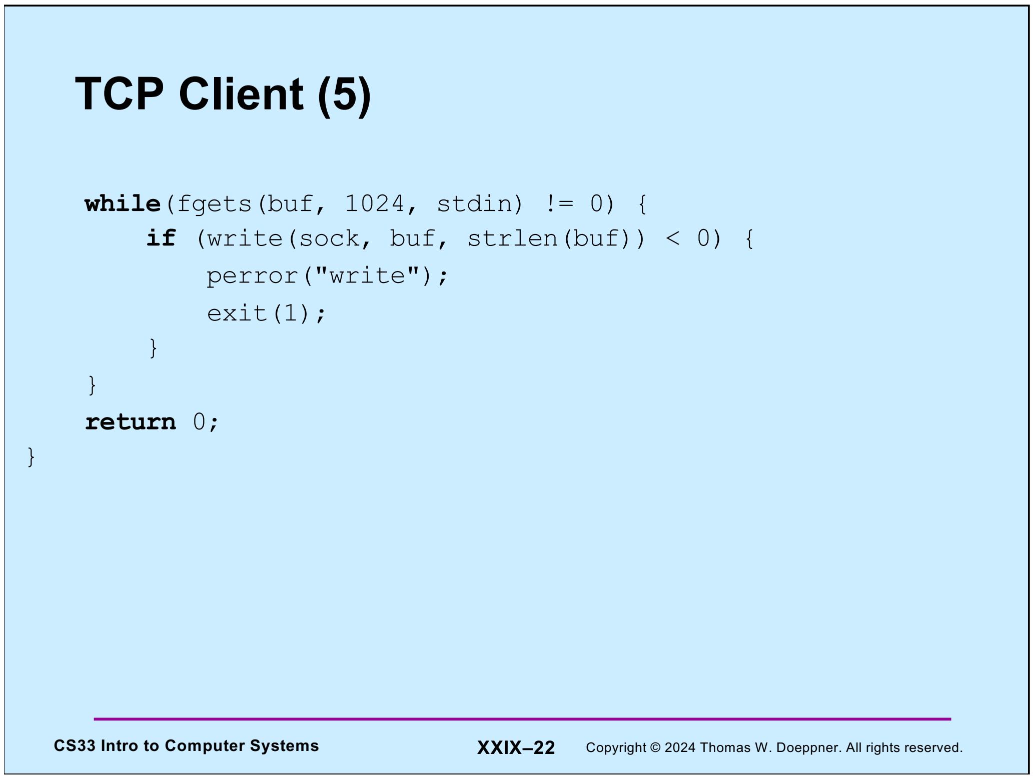
Note that no port number (or service) is associated with the client's socket. Usually, what port the client is using is unimportant and one is assigned arbitrarily when the client calls connect. If it's important that the client's socket have a particularly port associated with it, **bind** can be called on the socket before its used for communication.



TCP Client (4)

```
if (rp == NULL) {
    fprintf(stderr, "Could not connect to %s\n", argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

If no satisfactory address was found, the client terminates. Otherwise, it frees up the no-longer-needed list of addresses.



TCP Client (5)

```
    while(fgets(buf, 1024, stdin) != 0) {
        if (write(sock, buf, strlen(buf)) < 0) {
            perror("write");
            exit(1);
        }
    }
    return 0;
}
```

Finally, the clients reads from stdin and sends whatever it reads to the server.

Quiz 1

The previous slide contains
`write(sock, buf, strlen(buf))`

If data is lost and must be retransmitted

a) write returns an error so the caller can retransmit the data.

b) nothing happens as far as the application code is concerned, the data is retransmitted automatically.

c) the receiving application has to tell the sending application to retransmit.

Quiz 2

A previous slide contains
`write(sock, buf, strlen(buf))`

We lose the connection to the other party (perhaps a network cable is cut).

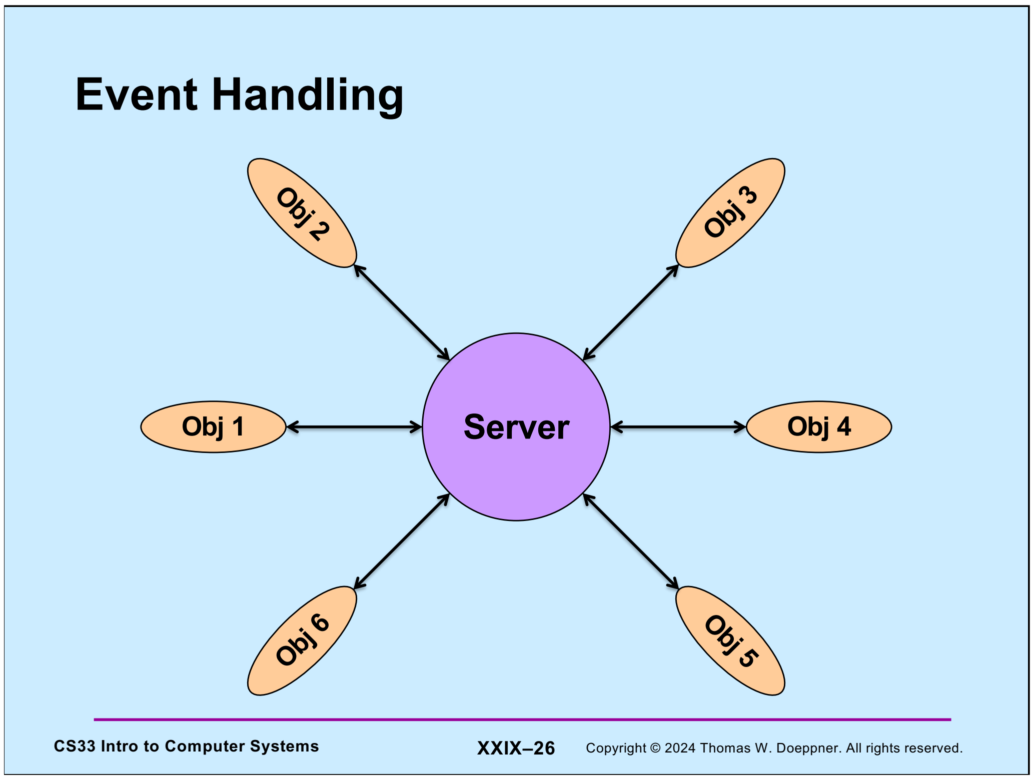
a) write returns an error so the caller can reconnect, if desired.

b) nothing happens as far as the application code is concerned, the connection is reestablished automatically.

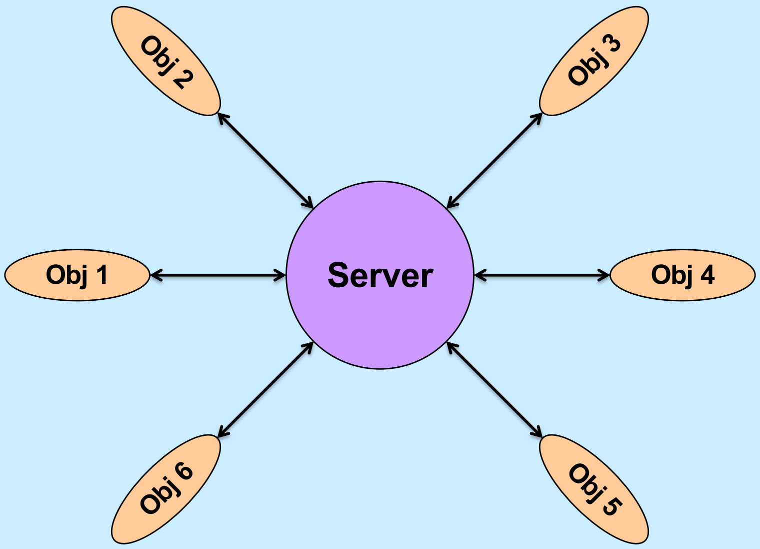
c) the receiving application has to tell the sending application to reconnect.

CS 33

Event-Based Programming

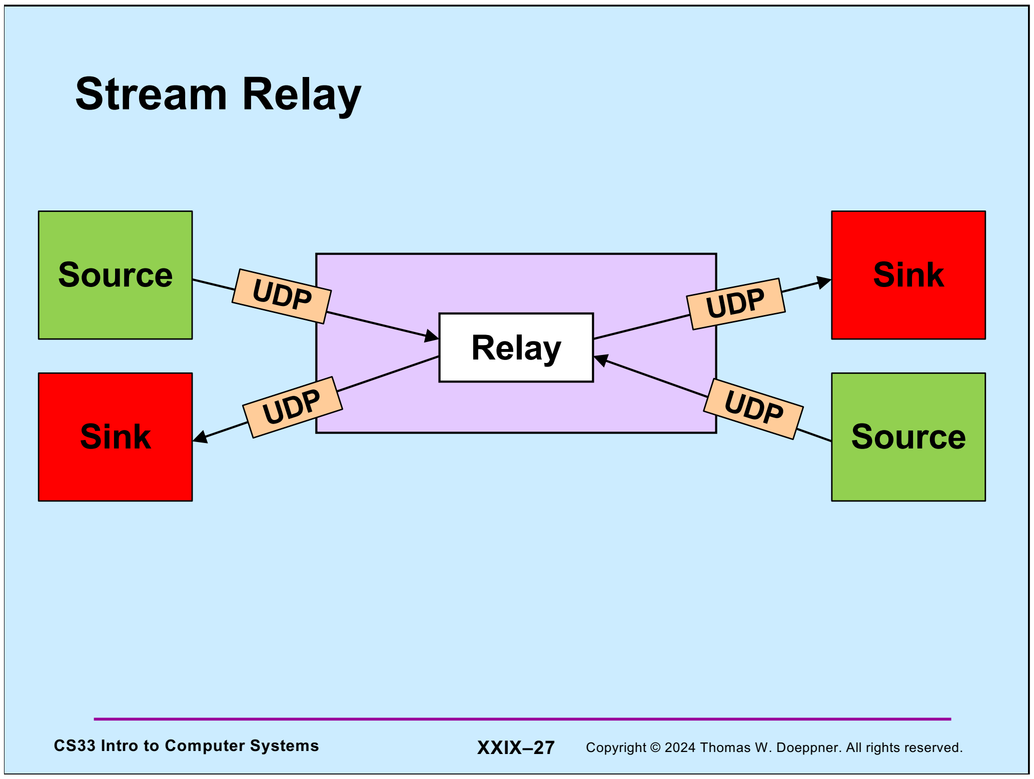


Event Handling

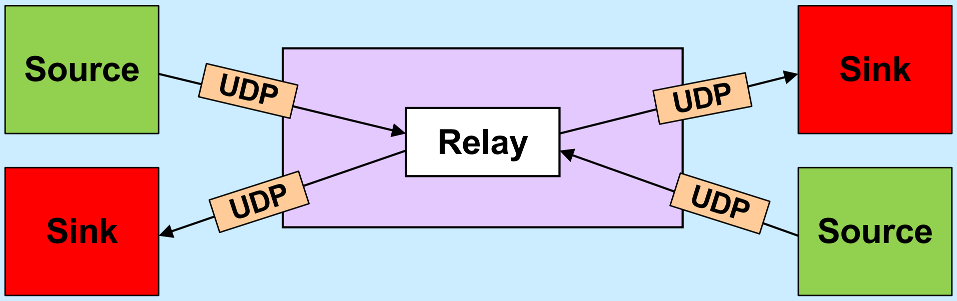


Here we have a server that is dealing with a number of external objects. These objects are independent of one another and, somewhat randomly, seek the attention of the server, which must process input from them and send them output.

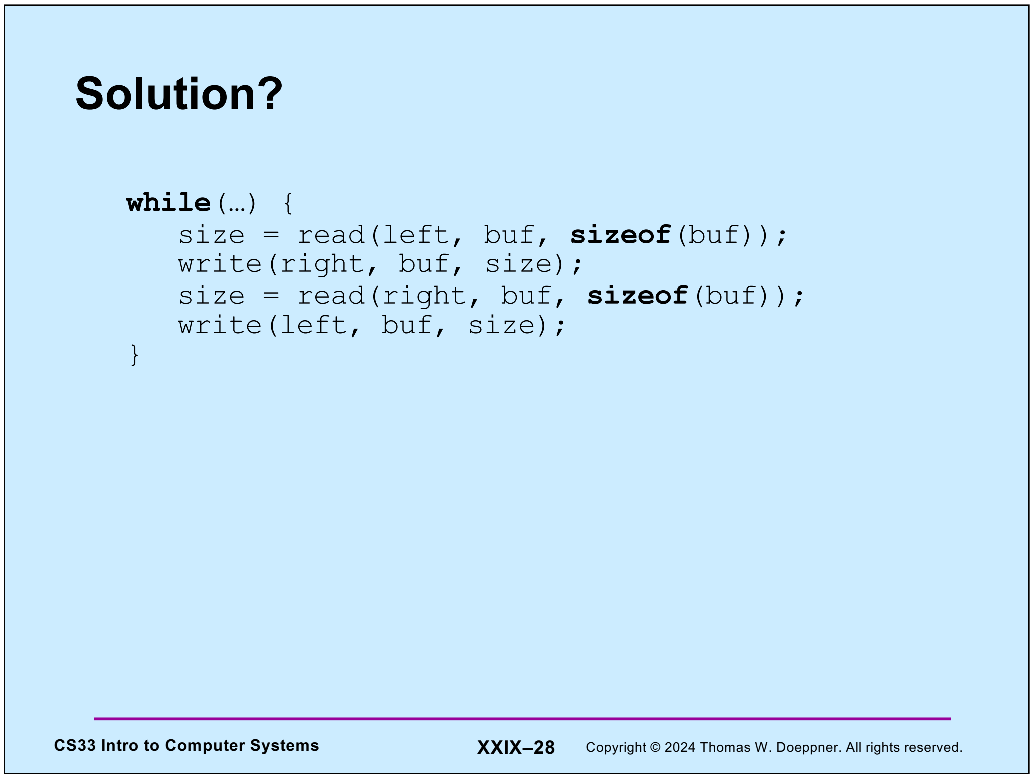
This is known as **event-based programming**: we write code that responds to events coming from a number of sources.



Stream Relay



As a more concrete example we examine a simple relay: we want to write a program that takes data received via UDP from a source on the left and forwards it (via UDP) to a sink on the right. At the same time, it's taking data received from a source on the right and forwards it (via UDP) to a sink on the left.

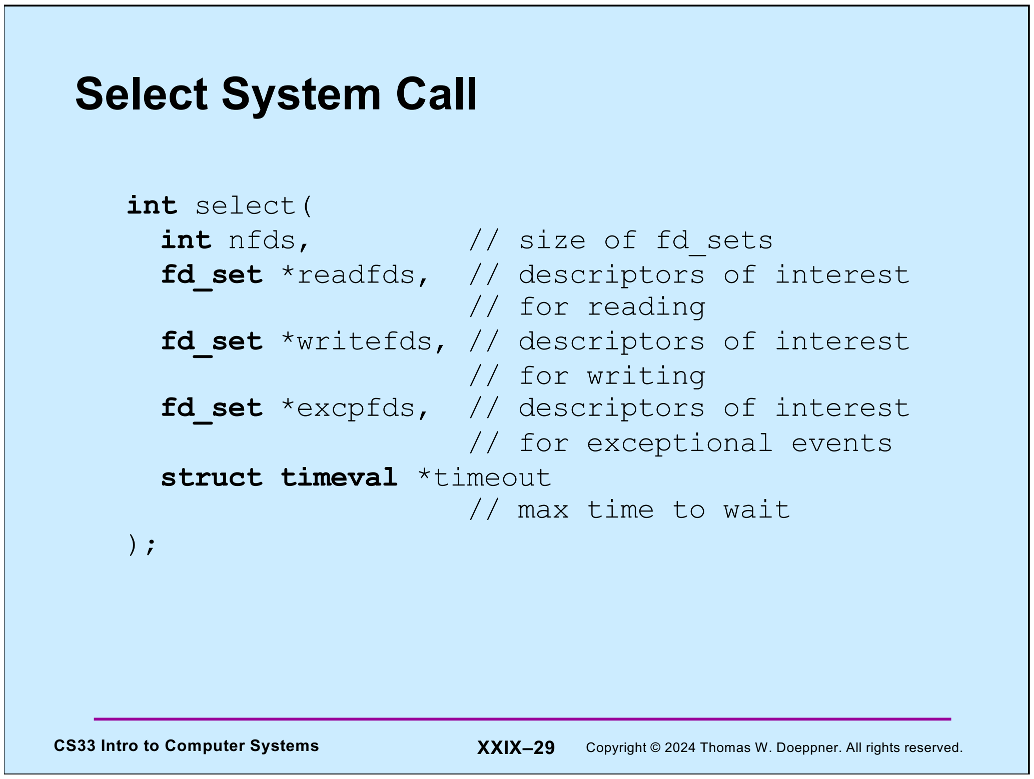


Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```

This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

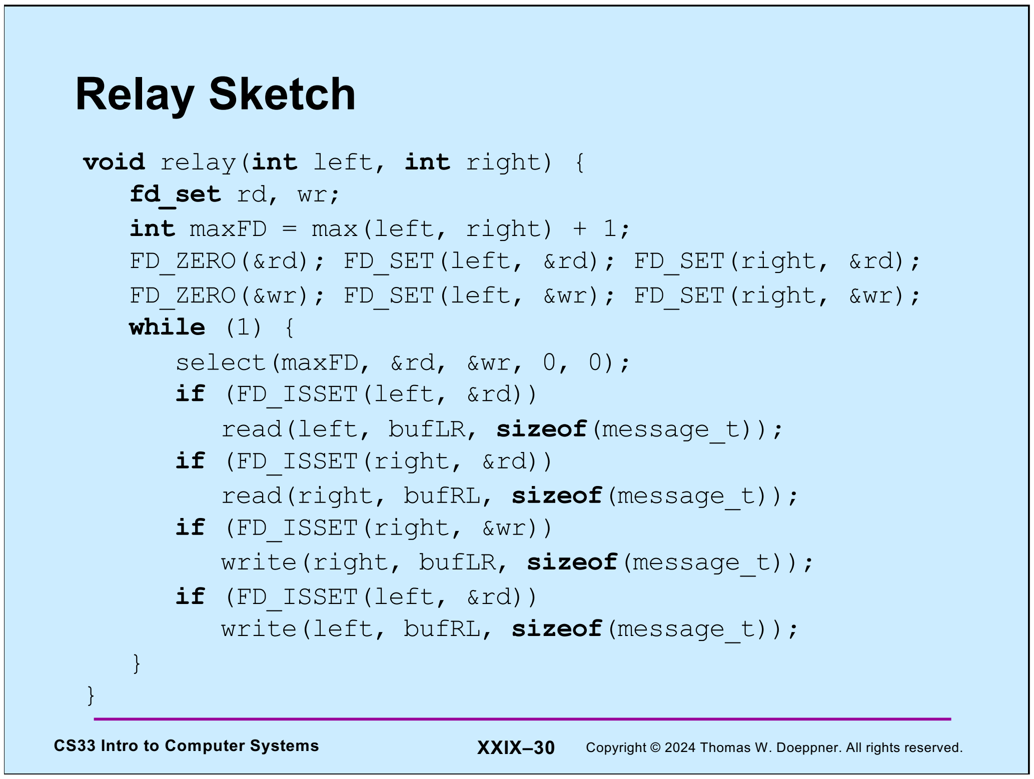
Note that to simply the slides a bit, even though we're using UDP, we'll use read and write system calls – the source and destination are assumed in each case.



Select System Call

```
int select(
  int nfds,          // size of fd_sets
  fd_set *readfds,   // descriptors of interest
                     // for reading
  fd_set *writefds,  // descriptors of interest
                     // for writing
  fd_set *excpfds,   // descriptors of interest
                     // for exceptional events
  struct timeval *timeout
                     // max time to wait
);
```

The **select** system call operates on three sets of file descriptors: one of fie descriptors we're interested in reading from, one of file descriptors we're interested in writing to, and one of file descriptors that might have exceptional conditions pending (we haven't covered any examples of such things – they come up as a peculiar feature of TCP known as out-of-band data, which is beyond the scope of this course). A call to **select** waits until at least one of the file descriptors in the given sets has something of interest. In particular, for a file descriptor in the read set, it's possible to read data from it; for a file descriptor in the write set, it's possible to write data to it. The **nfds** parameter indicates the maximum file descriptor number in any of the sets. The **timeout** parameter may be used to limit how long **select** waits. If set to zero, select waits indefinitely.

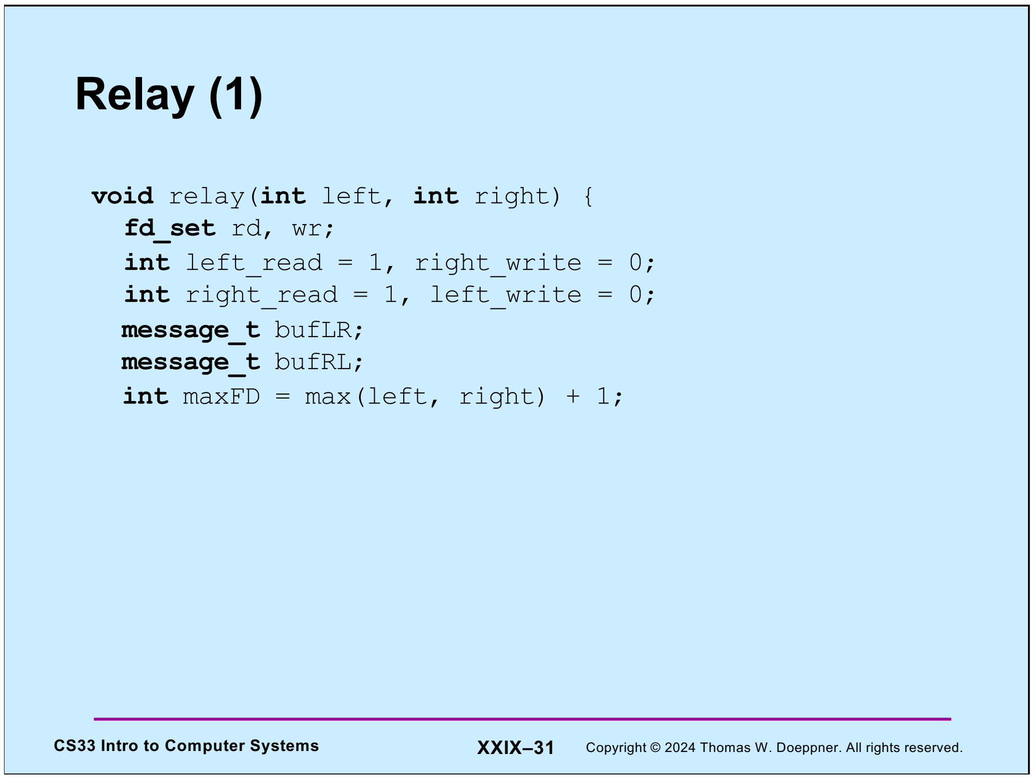


Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, sizeof(message_t));
      if (FD_ISSET(right, &rd))
         read(right, bufRL, sizeof(message_t));
      if (FD_ISSET(right, &wr))
         write(right, bufLR, sizeof(message_t));
      if (FD_ISSET(left, &rd))
         write(left, bufRL, sizeof(message_t));
   }
}
```

Here a simplified version of a program to handle the relay problem using *select*. An **fd_set** is a data type that represents a set of file descriptors. FD_ZERO, FD_SET, and FD_ISSET are macros for working with fd_sets; the first makes such a set represent the null set, the second sets a particular file descriptor to be included in the set, the last checks to see if a particular file descriptor is included in the set.

This sketch doesn't quite work because it doesn't take into account the fact that we have limited buffer space: we can't read two messages in a row from one side without writing the first to the other side before reading the second. Furthermore, even though select may say it's possible to write to either the left or the right side, we can't do so until we're read in some data from the other side. Also, the fd_sets that are select's arguments are modified on return from select to indicate if it's now possible to read or write on the associated file descriptor. Thus if, on return from select, it's not possible to use that file descriptor, its associated bit will be zero. We need to explicitly set it to one for the next call so that select knows we're still interested.



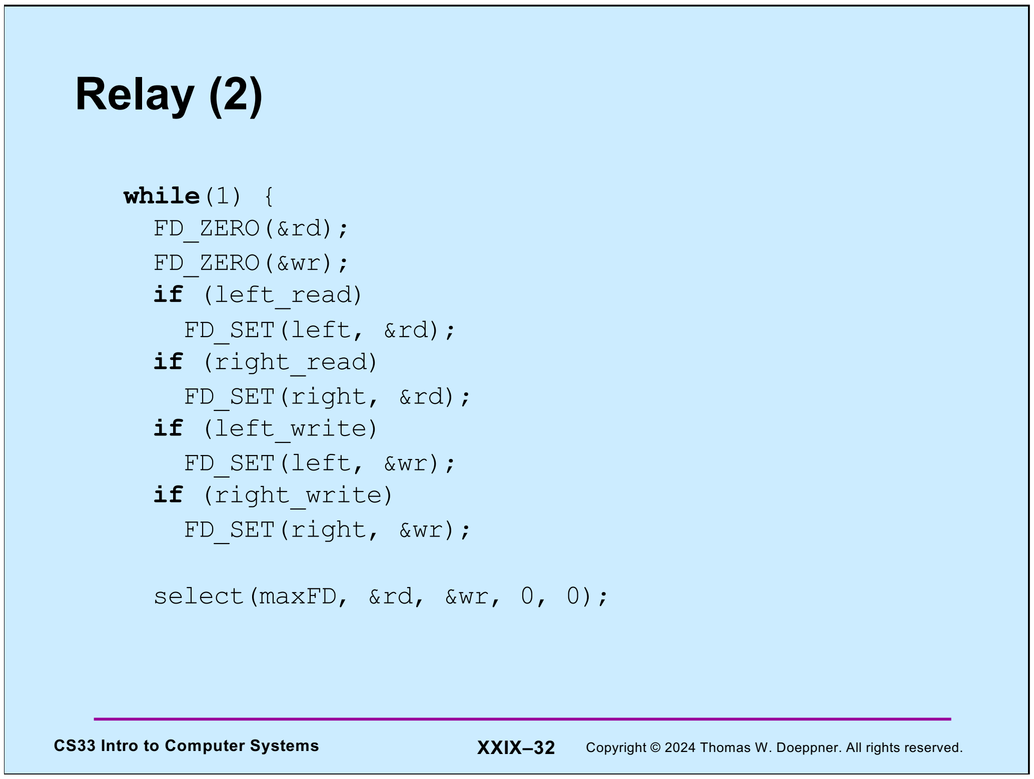
Relay (1)

```
void relay(int left, int right) {
  fd_set rd, wr;
  int left_read = 1, right_write = 0;
  int right_read = 1, left_write = 0;
  message_t bufLR;
  message_t bufRL;
  int maxFD = max(left, right) + 1;
```

This and the next three slides give a more complete version of the relay program.

Initially our program is prepared to read from either the left or the right side, but it's not prepared to write, since it doesn't have anything to write. The variables **left_read** and **right_read** are set to one to indicate that we want to read from the left and right sides. The variables **right_write** and **left_write** are set to zero to indicate that we don't yet want to write to either side.

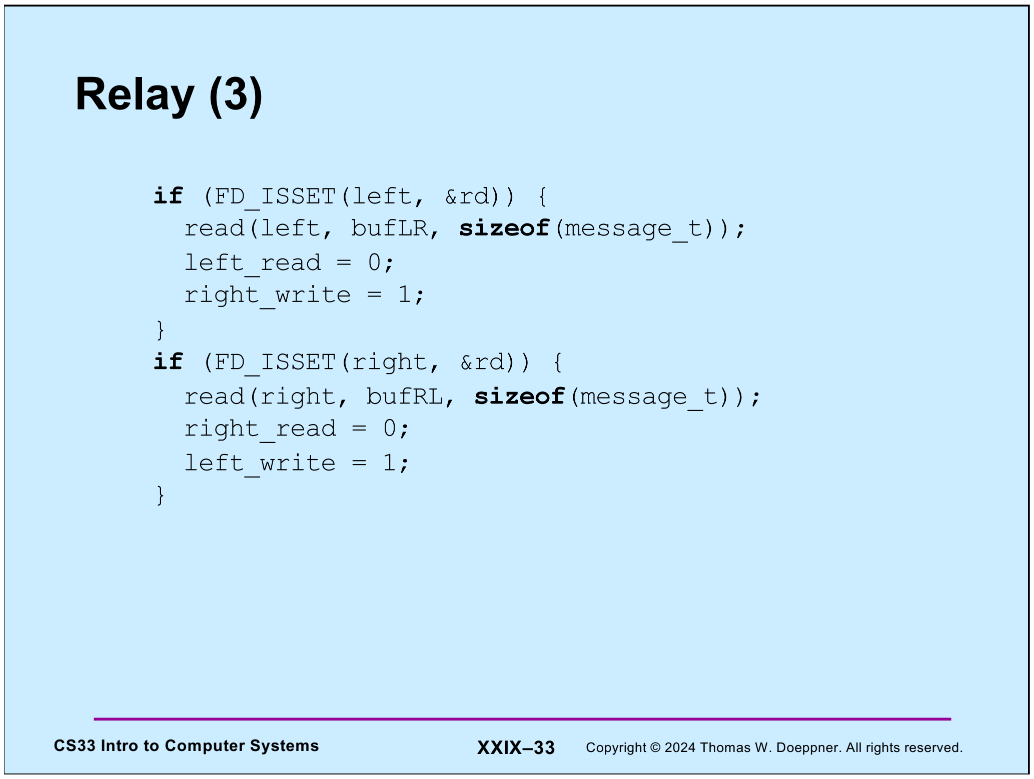
The two variables of type **message_t** are used as buffers to hold a messages received from the left and to be written to the right, or vice versa.


Relay (2)

```
while(1) {
  FD_ZERO(&rd);
  FD_ZERO(&wr);
  if (left_read)
    FD_SET(left, &rd);
  if (right_read)
    FD_SET(right, &rd);
  if (left_write)
    FD_SET(left, &wr);
  if (right_write)
    FD_SET(right, &wr);

  select(maxFD, &rd, &wr, 0, 0);
```

We set up the fd_sets **rd** and **wr** to indicate what we are interested in reading from and writing to (initially we have no interest in writing, but are interested in reading from either side).

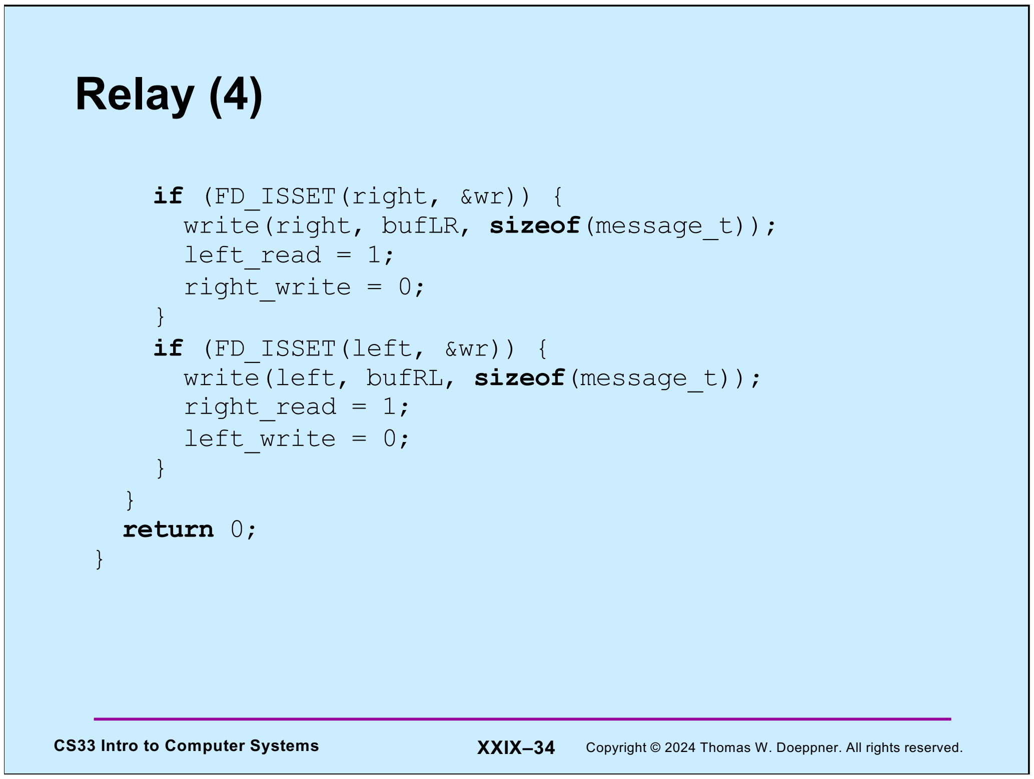


Relay (3)

```
if (FD_ISSET(left, &rd)) {
  read(left, bufLR, sizeof(message_t));
  left_read = 0;
  right_write = 1;
}
if (FD_ISSET(right, &rd)) {
  read(right, bufRL, sizeof(message_t));
  right_read = 0;
  left_write = 1;
}
```

If there is something to read from the left side, we read it. Having read it, we're temporarily not interested in reading anything further from the left side, but now want to write to the right side.

In a similar fashion, if there is something to read from the right side, we read it.



Relay (4)

```
    if (FD_ISSET(right, &wr)) {
      write(right, bufLR, sizeof(message_t));
      left_read = 1;
      right_write = 0;
    }
    if (FD_ISSET(left, &wr)) {
      write(left, bufRL, sizeof(message_t));
      right_read = 1;
      left_write = 0;
    }
  }
  return 0;
}
```

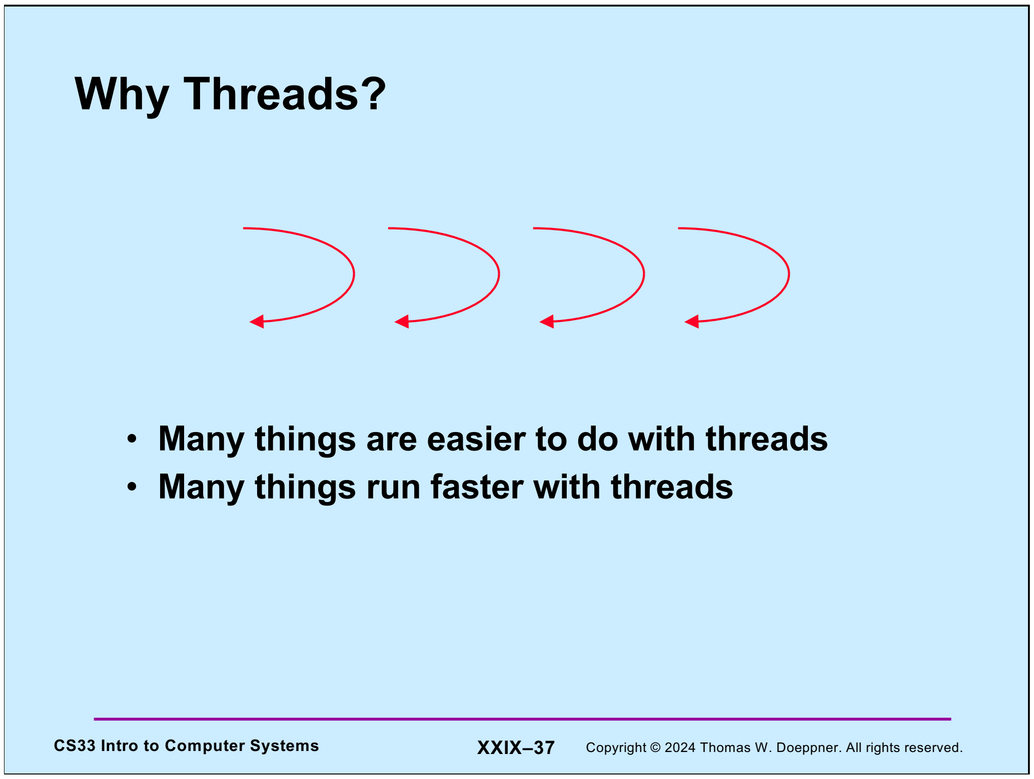
Similarly for writing: if we've written something to one side, we have nothing more to write to that side, but are now interested in reading from the other side.

CS 33

Multithreaded Programming (1)

Multithreaded Programming

- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions

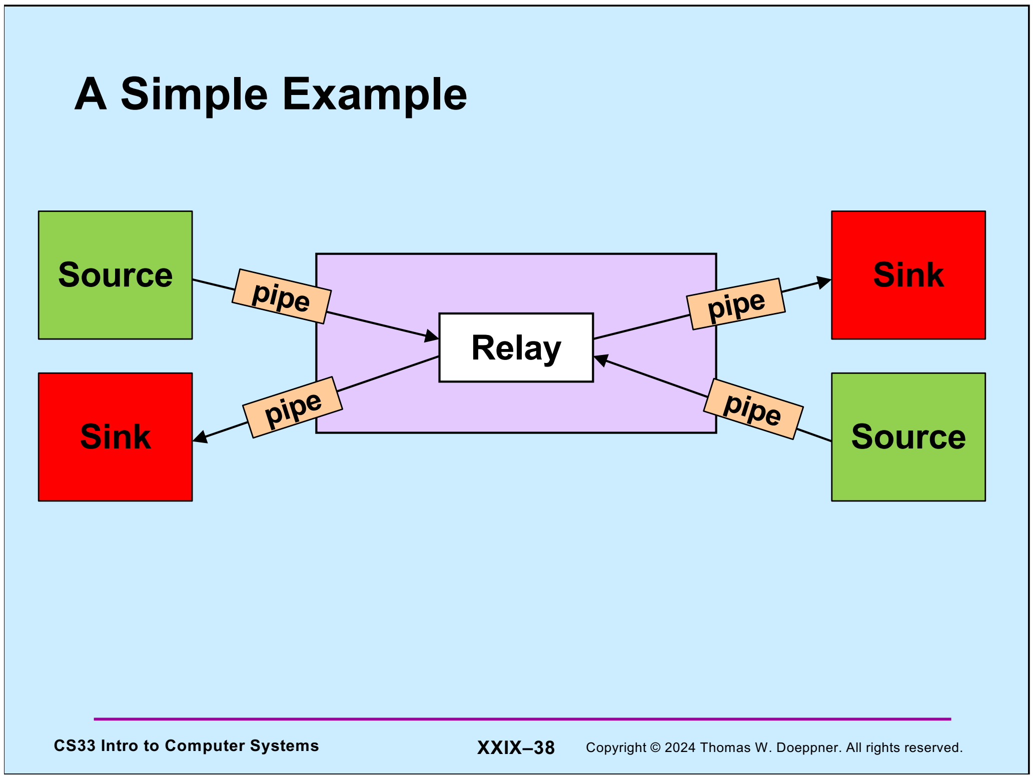


Why Threads?

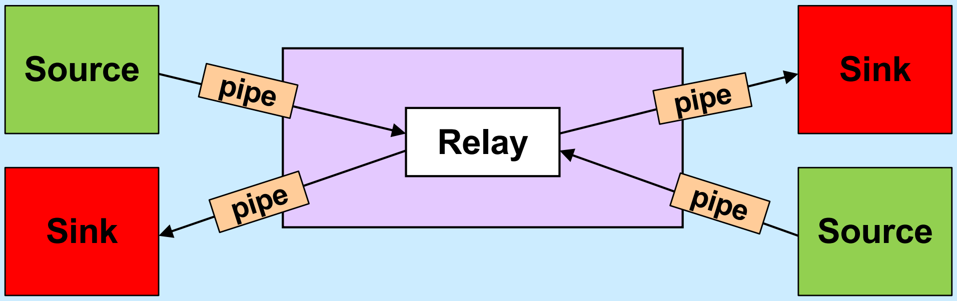
- Many things are easier to do with threads
- Many things run faster with threads

A **thread** is the abstraction of a processor — it is a **thread of control**. We are accustomed to writing single-threaded programs and to having multiple single-threaded programs running on our computers. Why does one want multiple threads running in the same program? Putting it only somewhat over-dramatically, programming with multiple threads is a powerful paradigm.

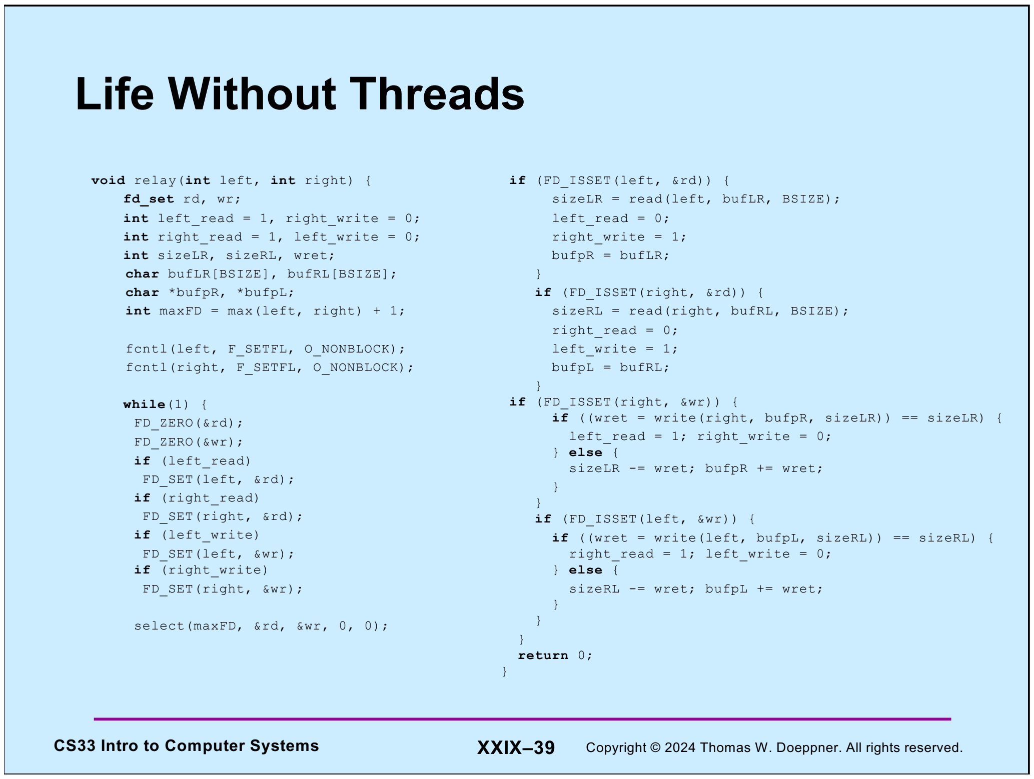
So, what is so special about this paradigm? Programming with threads is a natural means for dealing with **concurrency**. As we will see, concurrency comes up in numerous situations. A common misconception is that it is a useful concept only on multiprocessors. Threads do allow us to exploit the features of a multiprocessor, but they are equally useful on uniprocessors — in many instances a multithreaded solution to a problem is simpler to write, simpler to understand, and simpler to debug than a single-threaded solution to the same problem.



A Simple Example



For a simple example of a problem that is more easily solved with threads than without, let's look at the stream relay example from the previous lecture.



Life Without Threads

```
void relay(int left, int right) {
    fd_set rd, wr;
    int left_read = 1, right_write = 0;
    int right_read = 1, left_write = 0;
    int sizeLR, sizeRL, wret;
    char bufLR[BSIZE], bufRL[BSIZE];
    char *bufpR, *bufpL;
    int maxFD = max(left, right) + 1;

    fcntl(left, F_SETFL, O_NONBLOCK);
    fcntl(right, F_SETFL, O_NONBLOCK);

    while(1) {
     FD_ZERO(&rd);
     FD_ZERO(&wr);
     if (left_read)
      FD_SET(left, &rd);
     if (right_read)
      FD_SET(right, &rd);
     if (left_write)
      FD_SET(left, &wr);
     if (right_write)
      FD_SET(right, &wr);

     select(maxFD, &rd, &wr, 0, 0);

     if (FD_ISSET(left, &rd)) {
        sizeLR = read(left, bufLR, BSIZE);
        left_read = 0;
        right_write = 1;
        bufpR = bufLR;
      }
      if (FD_ISSET(right, &rd)) {
        sizeRL = read(right, bufRL, BSIZE);
        right_read = 0;
        left_write = 1;
        bufpL = bufRL;
      }
     if (FD_ISSET(right, &wr)) {
        if ((wret = write(right, bufpR, sizeLR)) == sizeLR) {
          left_read = 1; right_write = 0;
        } else {
          sizeLR -= wret; bufpR += wret;
        }
      }
      if (FD_ISSET(left, &wr)) {
        if ((wret = write(left, bufpL, sizeRL)) == sizeRL) {
          right_read = 1; left_write = 0;
        } else {
          sizeRL -= wret; bufpL += wret;
        }
      }
    }
    return 0;
  }
```

Here’s the event-oriented solution we devised earlier that uses **select** (and is rather complicated).

Life With Threads

```
 void copy(int source, int destination) {
  struct args *targs = args;
  char buf[BSIZE];

  while(1) {
    int len = read(source, buf, BSIZE);
    write(destination, buf, len);
  }
}
```

Here's an essentially equivalent solution that uses threads rather than select. We've left out the code that creates the threads (we'll see that pretty soon), but what's shown is executed by each of two threads. One has source set to the left side and destination to the right side, the other vice versa.

Quiz 3

The multi-threaded program, compared to the single-threaded program that uses *select*, is

a) always faster

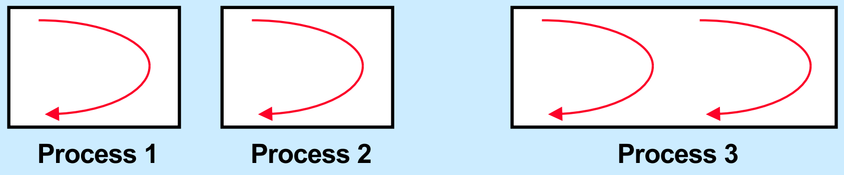
b) always faster if there is more than one processor

c) about the same for one processor; faster for more than one processor

d) often slower

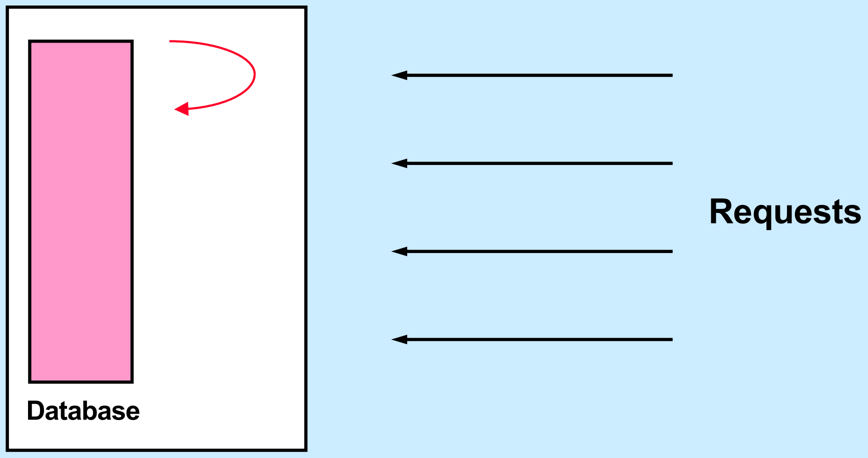
e) always slower

Processes vs. Threads

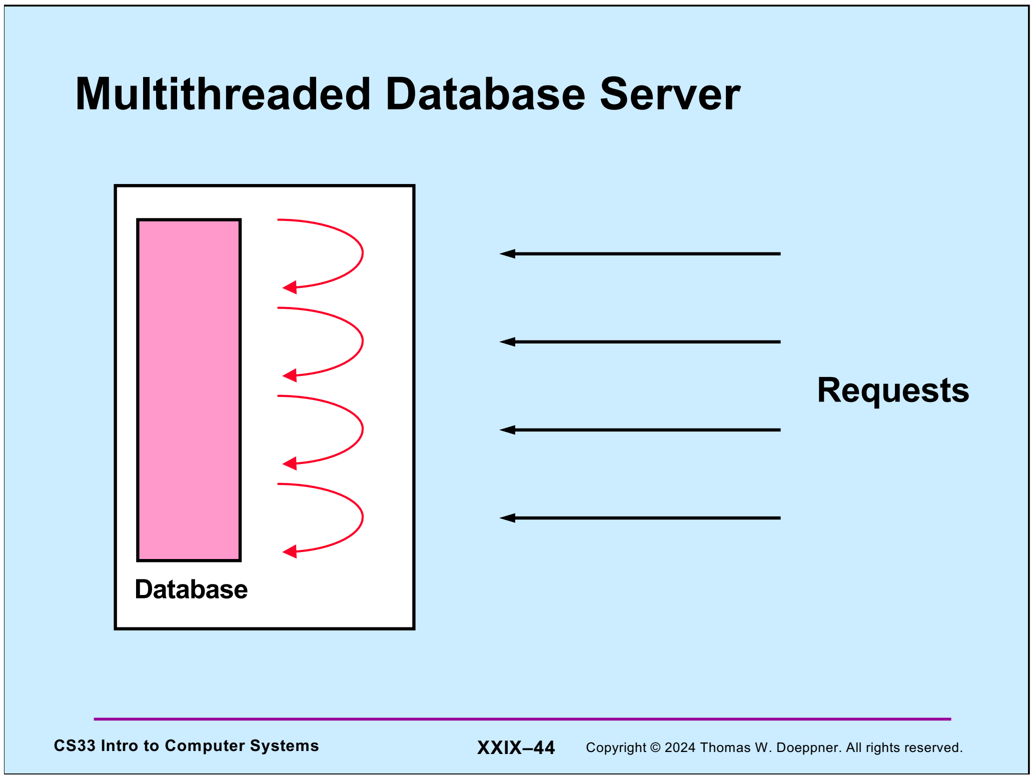


Threads provide concurrency, but so do processes. So, what is the difference between two single-threaded processes and one two-threaded process? First of all, if one process already exists, it is much cheaper to create another thread in the existing process than to create a new process. Switching between the contexts of two threads in the same process is also often cheaper than switching between the contexts of two threads in different processes. Finally, two threads in one process share everything — both address space and open files; the two can communicate without having to copy data. Though two different processes can share memory in modern Unix systems, the most common forms of interprocess communication are far more expensive.

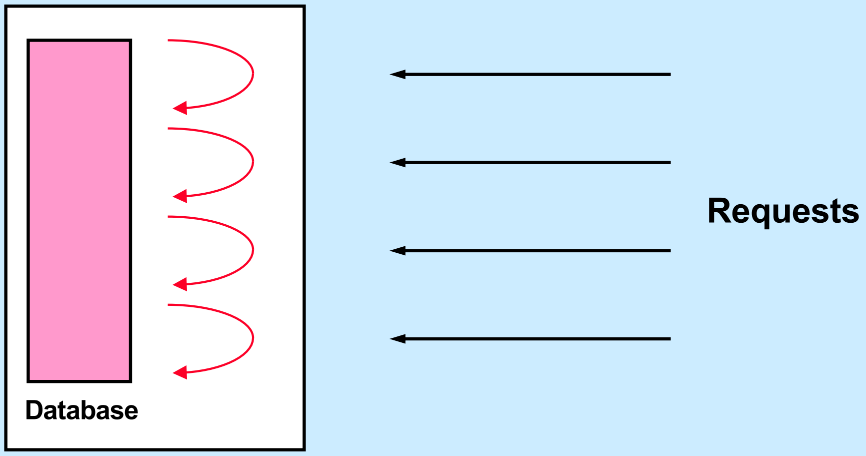
Single-Threaded Database Server



Here is another server example, a database server handling multiple clients. The single-threaded approach to dealing with these requests is to handle them sequentially or to multiplex them explicitly. The former approach would be unfair to quick requests occurring behind lengthy requests, and the latter would require fairly complex and error-prone code.

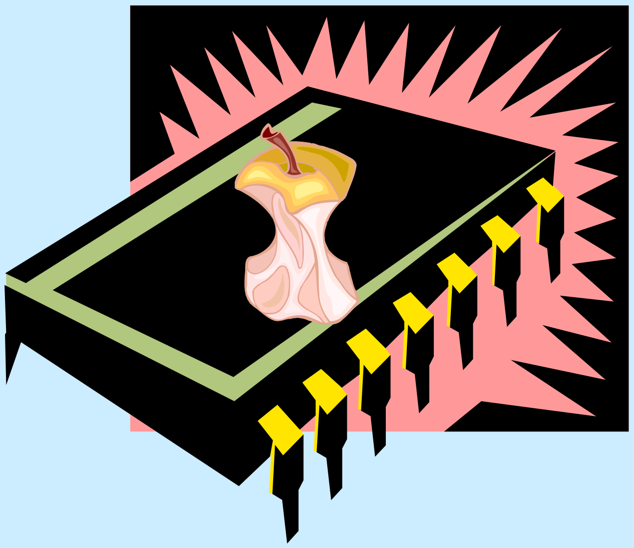


Multithreaded Database Server

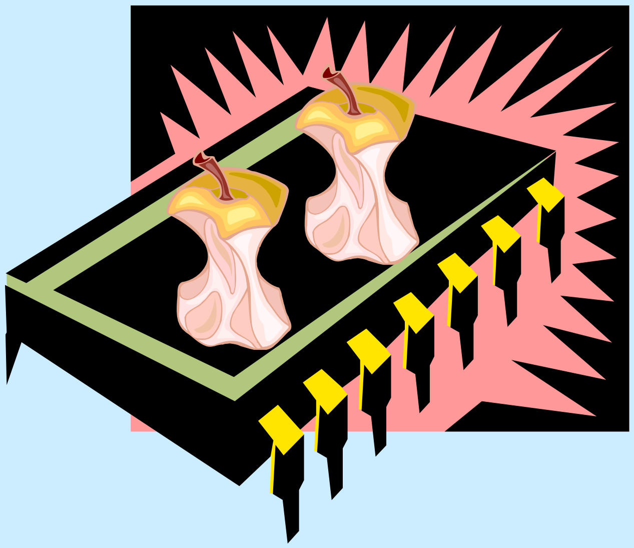


We now rearchitect our server to be multithreaded, assigning a separate thread to each request. The code is as simple as in the sequential approach and as fair as in the multiplexed approach. Some synchronization of access to the database is required, a topic we will discuss soon.

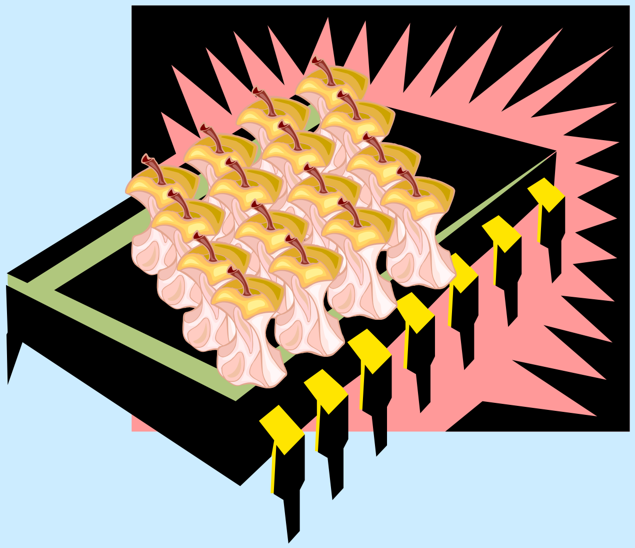
Single-Core Chips



Dual-Core Chips



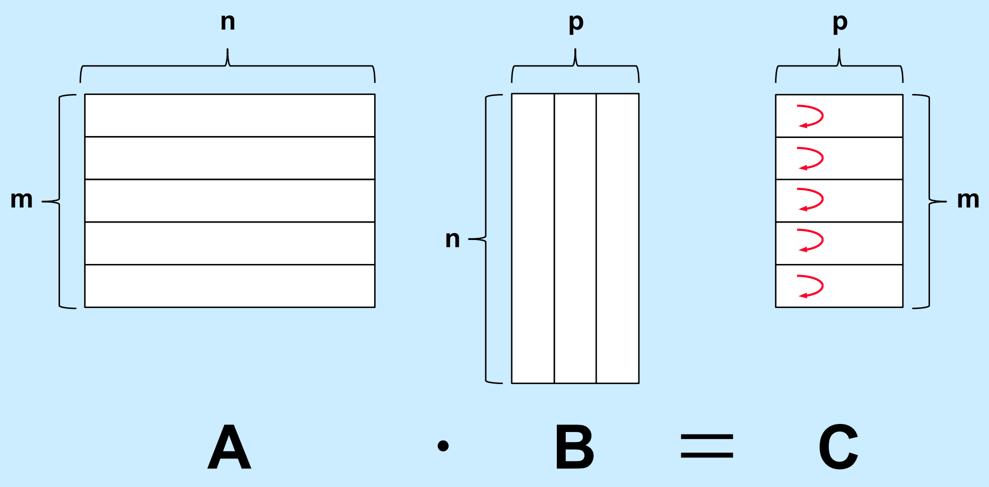
Multi-Core Chips



Good News/Bad News

- ☺ **Good news**
  - – multi-threaded programs can take advantage of multi-core chips (single-threaded programs cannot)
- ☹ **Bad news**
  - – it’s not easy
    - » must have parallel algorithm
      - • employing at least as many threads as processors
      - • threads must keep processors busy
        - – doing useful work

Matrix Multiplication Revisited



Standards

- POSIX 1003.4a → 1003.1c → 1003.1j
- Microsoft
  - Win32/64

Despite the long-known advantages of programming with threads, only relatively recently have standard APIs for multithreaded programming been developed. The most important of these APIs, at least in the Unix world, is the one developed by the group known as POSIX 1003.4a. This effort took a number of years and in the summer of 1995 resulted in an approved standard, which is now known by the number 1003.1c. In 2000, the POSIX advanced realtime standard, 1003.1j, was approved. It contains a number of additional features added to POSIX threads.

Microsoft, characteristically, produced a threads package whose interface has little in common with those of the Unix world. Moreover, there are significant differences between the Microsoft and POSIX approaches — some of the constructs of one cannot be easily implemented in terms of the constructs of the other, and vice versa. Despite this, both approaches are equally useful for multithreaded programming.