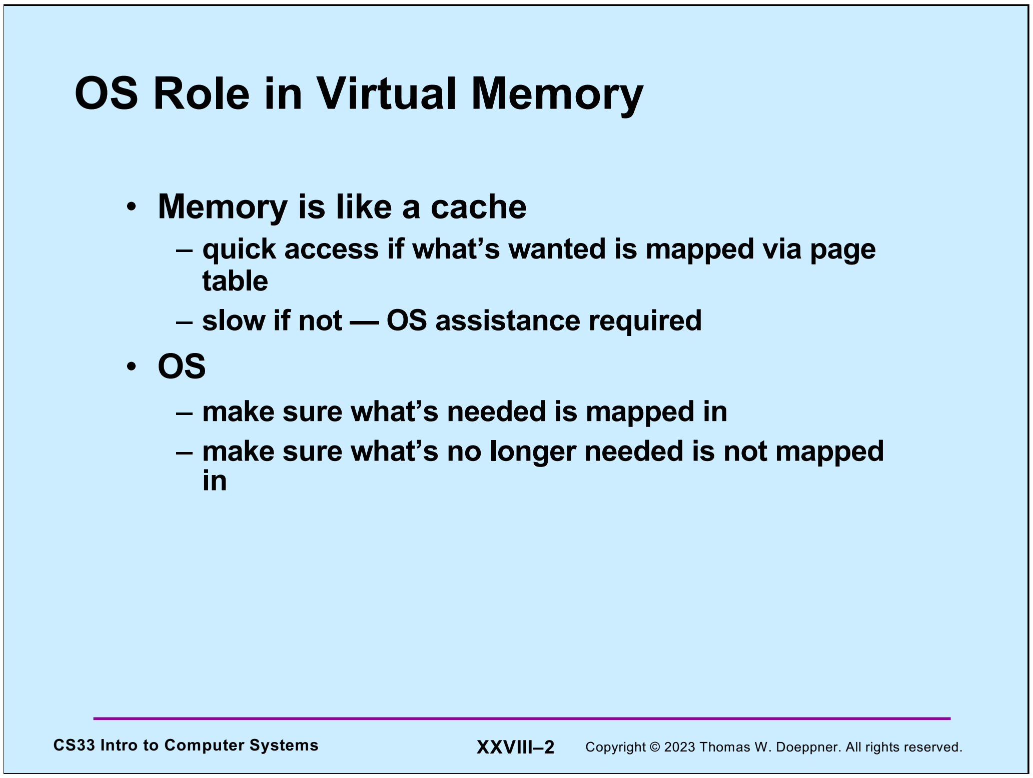
CS 33

Virtual Memory (2)

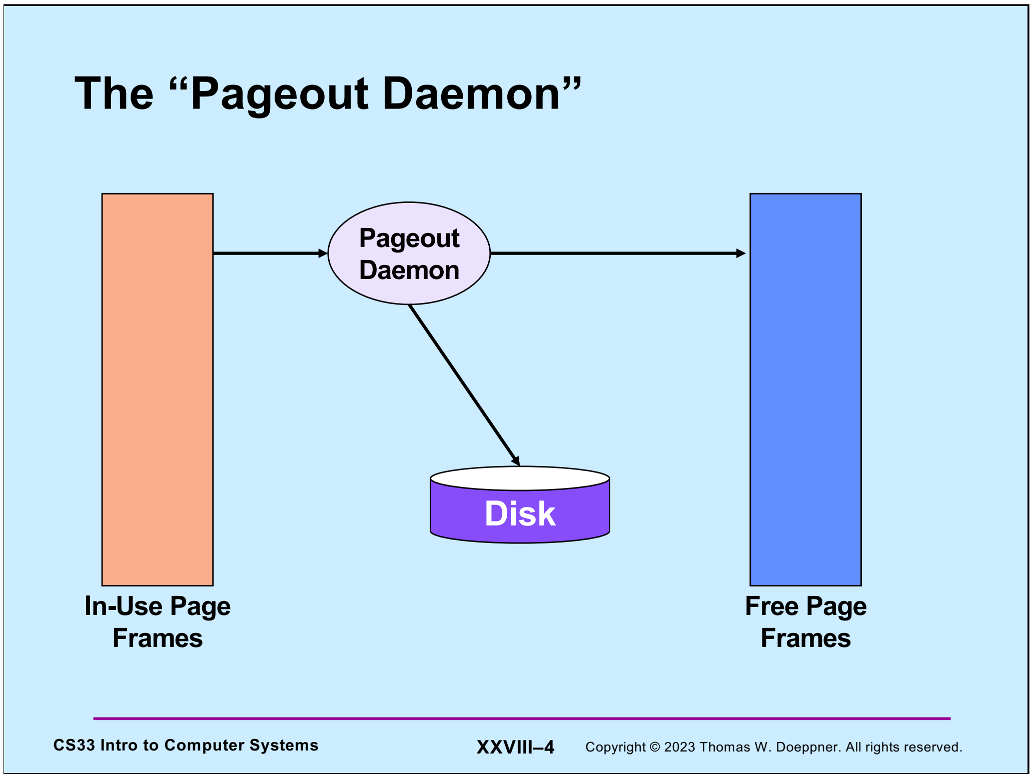


OS Role in Virtual Memory

- Memory is like a cache
  - quick access if what’s wanted is mapped via page table
  - slow if not — OS assistance required
- OS
  - make sure what’s needed is mapped in
  - make sure what’s no longer needed is not mapped in

Mechanism

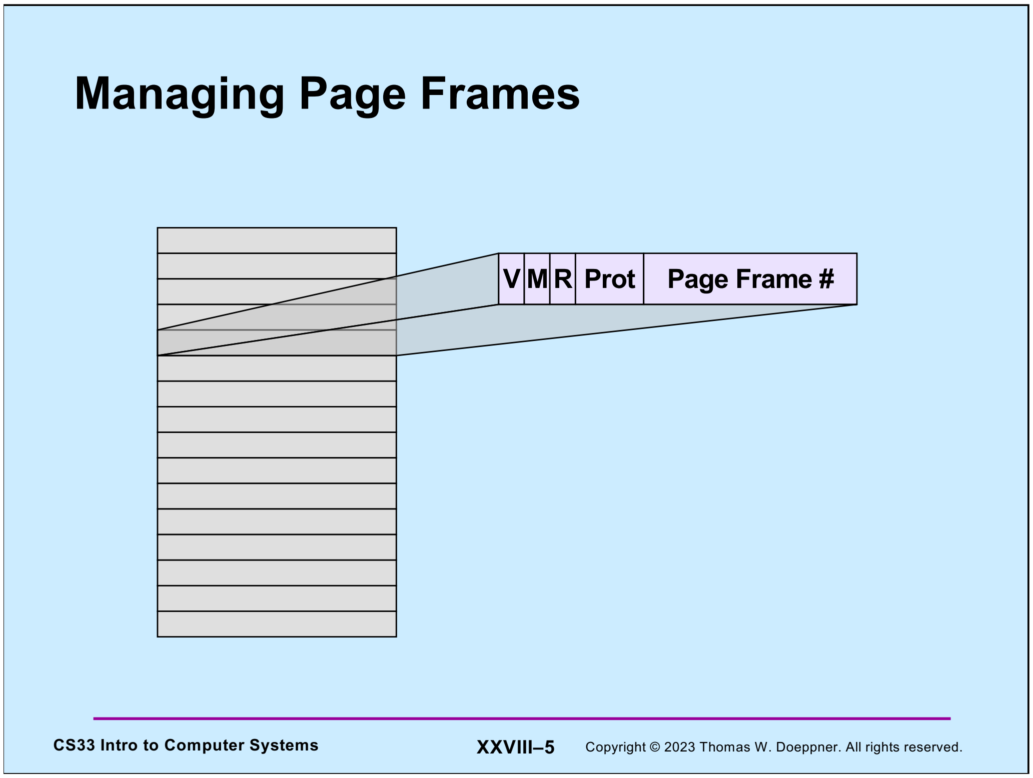
- Program references memory
  - if reference is mapped, access is quick
    - » even quicker if translation in TLB and referent in on-chip cache
  - if not, page-translation fault occurs and OS is invoked
    - » determines desired page
    - » maps it in, if legal reference



The “Pageout Daemon”

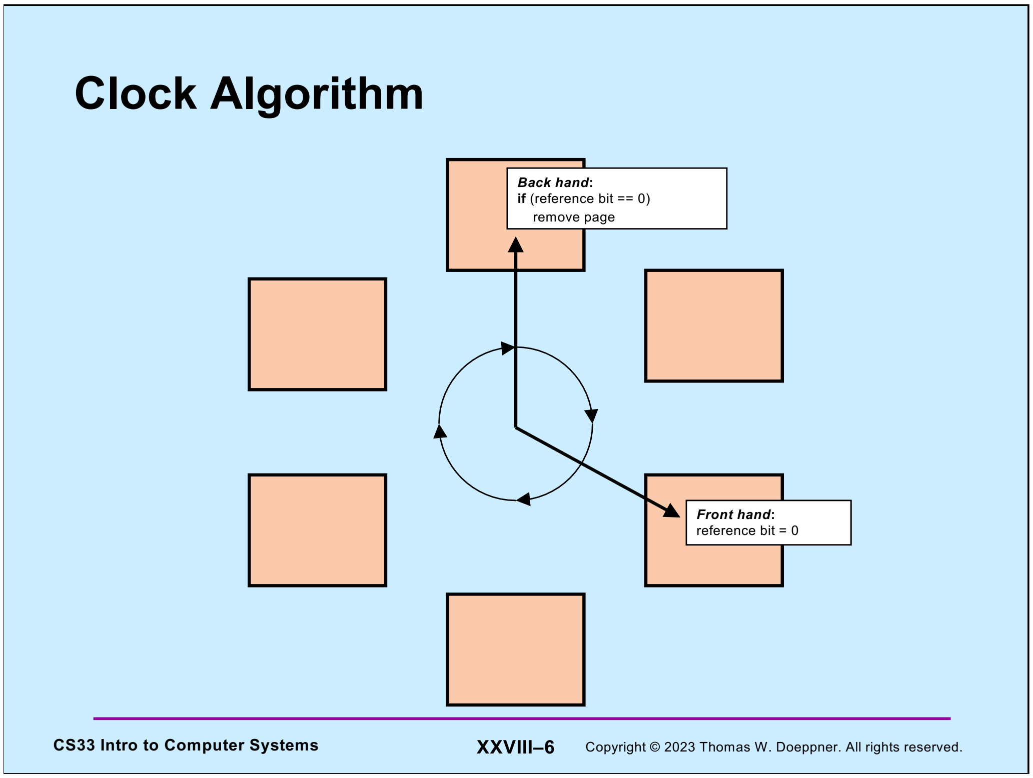
The (kernel) thread that maintains the free page-frame list is typically called the **pageout daemon**. Its job is to make certain that the free page-frame list has enough page frames on it. If the size of the list drops below some threshold, then the pageout daemon examines those page frames that are being used and selects a number of them to be freed. Before freeing a page, it must make certain that a copy of the current contents of the page exists on secondary storage. So, if the page has been modified since it was brought into primary storage (easily determined by the hardware-supported **modified bit**), it must first be written out to secondary storage. In many systems, the pageout daemon groups such pageouts into batches, so that a number of pages can be written out in a single operation, thus saving disk time. Unmodified, selected pages are transferred directly to the free page-frame list, modified pages are put there after they have been written out.

In most systems, pages in the free list get a “second chance” — if a thread in a process references such a page, there is a page fault (the page frame has been freed and could be used to hold another page), but the page-fault handler checks to see if the desired page is still in primary storage, but in the free list. If it is in the free list, it is removed and given back to the faulting process. We still suffer the overhead of a trap, but there is no wait for I/O.



Managing Page Frames

The OS can keep track of the history of page frame by use of two bits in each page-table entry: the *modify* bit, which is set by hardware whenever the associated page frame is modified, and the *referenced* bit, which is set by hardware whenever the associated page is accessed (via either a load or a store).



Clock Algorithm

Back hand:
if (reference bit == 0)
  remove page

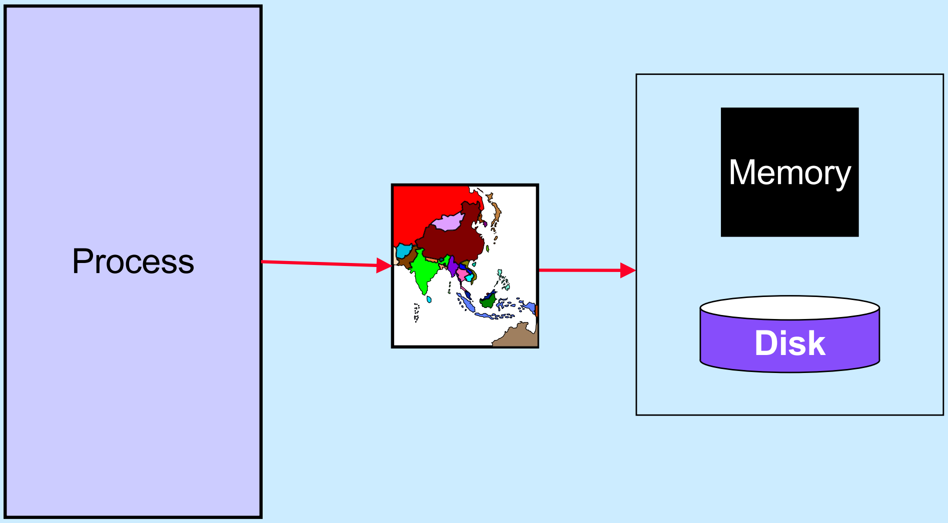
Front hand:
reference bit = 0

A common approach for determining which page frames are not in use is known as the clock algorithm. All active page frames are conceptually arranged in a circularly linked list. The page-out thread slowly traverses the list. The “one-handed” version of the clock algorithm, each time it encounters a page, checks the reference bit in the corresponding translation entry: if the bit is set, it clears it. If the bit is clear, it adds the page to the free list (writing it back to secondary storage first, if necessary).

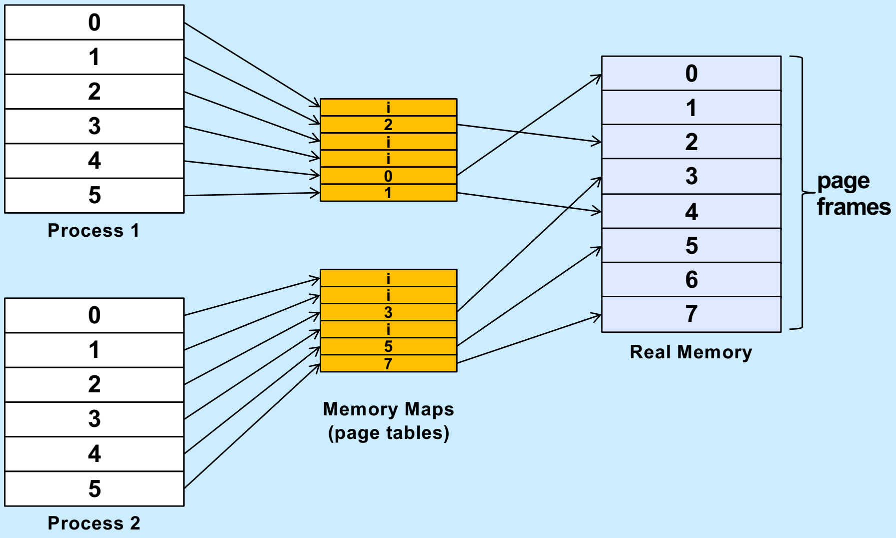
A problem with the one-handed version is that, in systems with large amounts of primary storage, it might take too long for the page-out thread to work its way all around the list of page frames before it can recognize that a page has not been recently referenced. In the two-handed version of the clock algorithm, the page-out thread implements a second hand some distance behind the first. The front hand simply clears reference bits. The second (back) hand removes those pages whose reference bits have not been set to one by the time the hand reaches the page frame.

Why is virtual memory used?

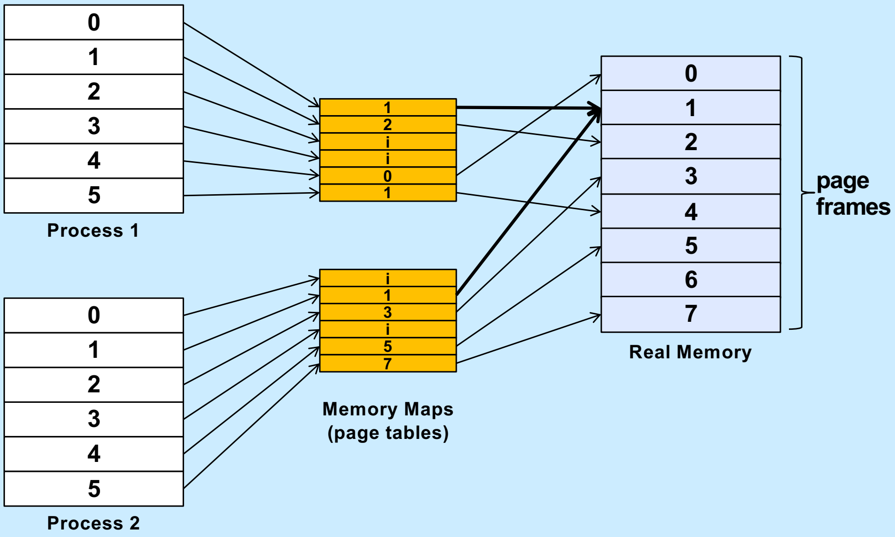
More VM than RM



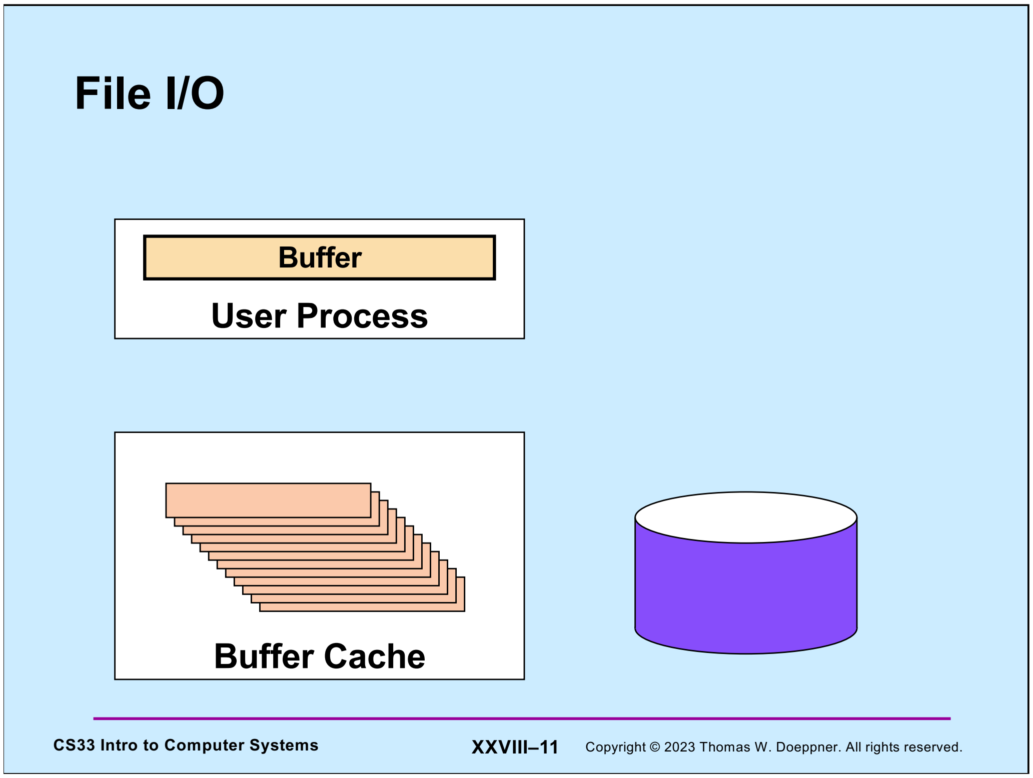
Isolation


0
1
2
3
4
5
Process 1
i
2
i
i
0
1
0
1
2
3
4
5
Process 2
i
i
3
i
5
7
Memory Maps
(page tables)
0
1
2
3
4
5
6
7
Real Memory
page
frames

Sharing



Virtual Memory



File I/O

Buffer

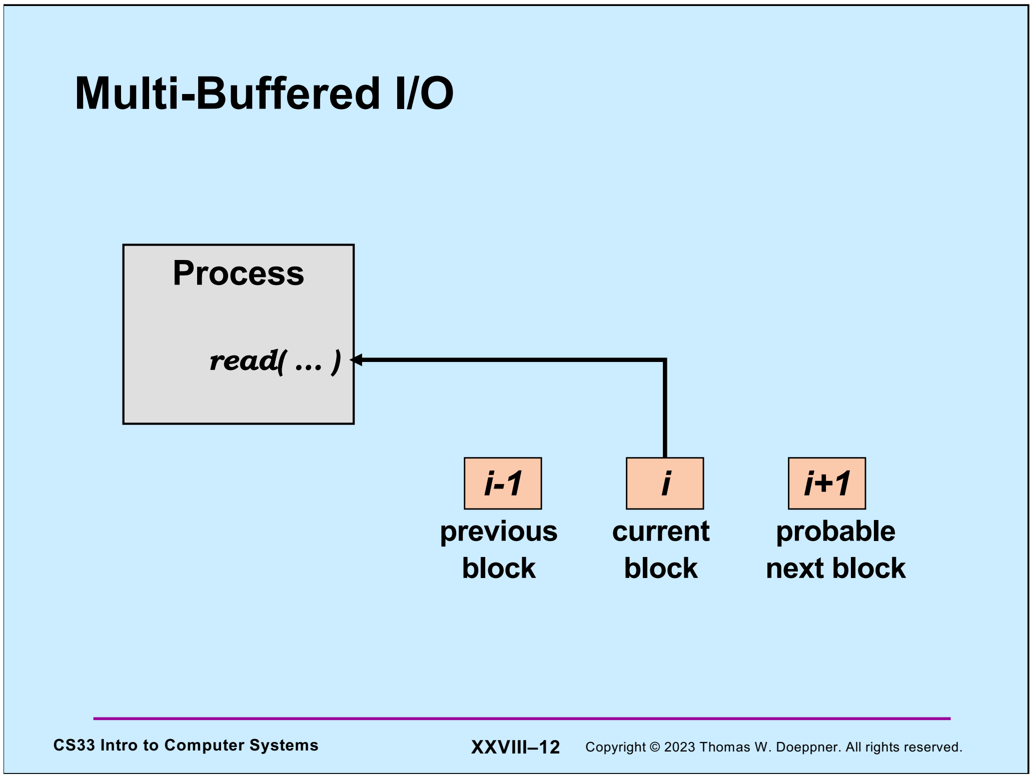
User Process

Buffer Cache

File I/O in Unix, and in most operating systems, is not done directly to the disk drive, but through intermediary buffers, known as the buffer cache, in the operating system's address space. This cache has two primary functions. The first, and most important, is to make possible concurrent I/O and computation within a Unix process. The second is to insulate the user from physical disk-block boundaries.

From a user process's point of view, I/O is **synchronous**. By this we mean that when the I/O system call returns, the system no longer needs the user-supplied buffer. For example, after a write system call, the data in the user buffer has either been transmitted to the device or copied to a kernel buffer — the user can now scribble over the buffer without affecting the data transfer. Because of this synchronization, from a user process's point of view, no more than one I/O operation can be in progress at a time.

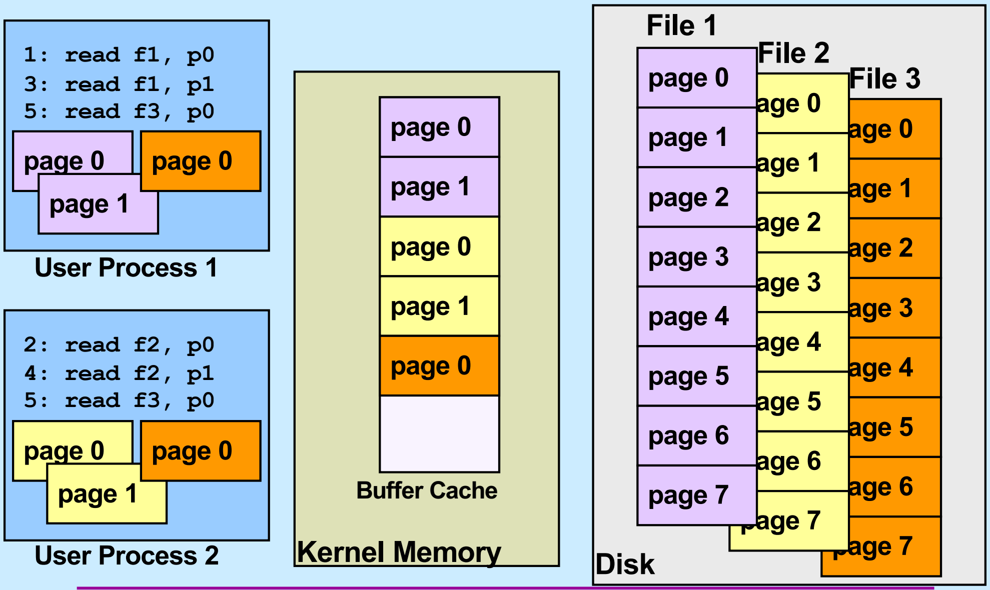
The buffer cache provides a kernel implementation of multibuffered I/O, and thus concurrent I/O and computation are made possible.



Multi-Buffered I/O

The use of **read-aheads** and **write-behinds** makes possible concurrent I/O and computation: if the block currently being fetched is block *i* and the previous block fetched was block *i-1*, then block *i+1* is also fetched. Modified blocks are normally written out not synchronously but instead sometime after they were modified, asynchronously.

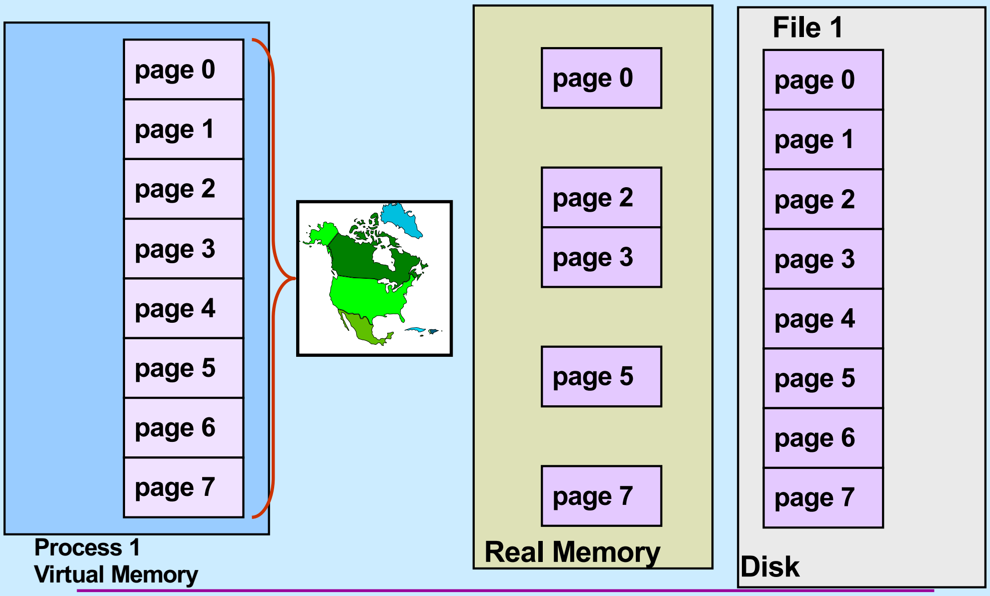
Traditional I/O



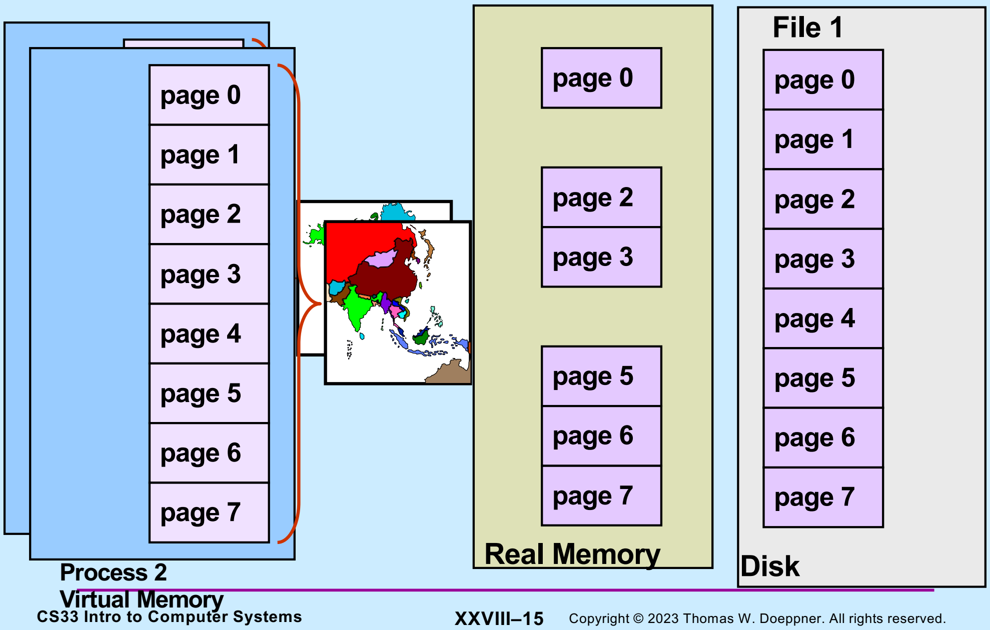
```
1: read f1, p0
3: read f1, p1
5: read f3, p0
```

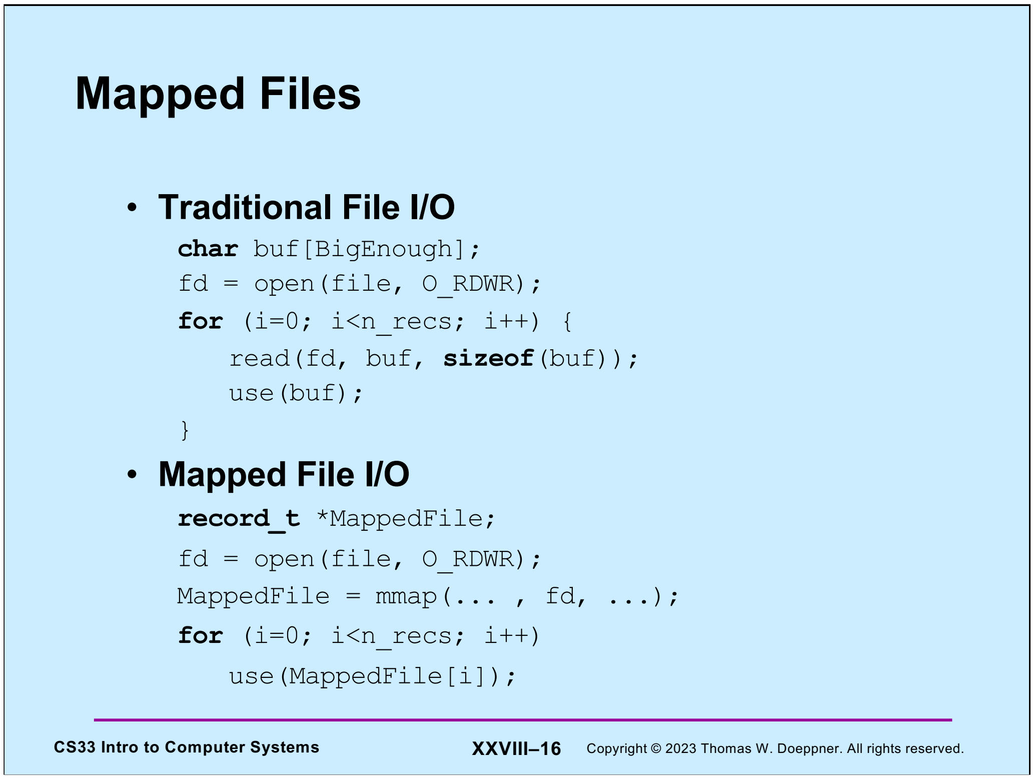
```
2: read f2, p0
4: read f2, p1
5: read f3, p0
```

Mapped File I/O



Multi-Process Mapped File I/O





Mapped Files

- **Traditional File I/O**

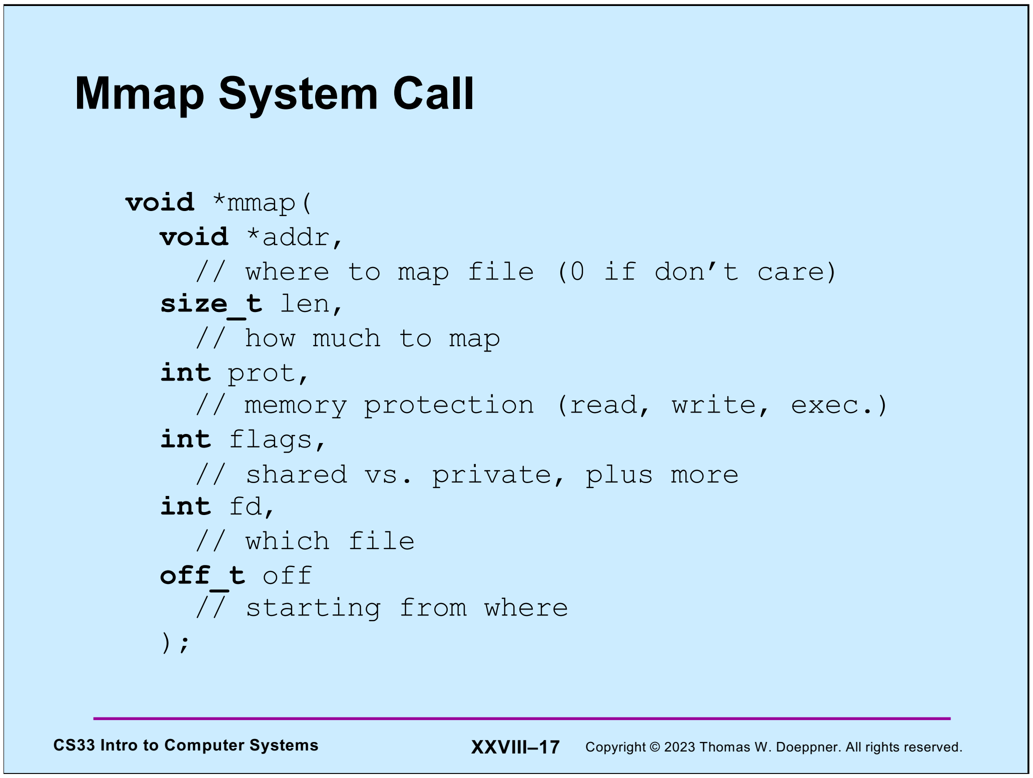
```
char buf[BigEnough];
fd = open(file, O_RDWR);
for (i=0; i<n_recs; i++) {
    read(fd, buf, sizeof(buf));
    use(buf);
}
```

- **Mapped File I/O**

```
record_t *MappedFile;
fd = open(file, O_RDWR);
MappedFile = mmap(... , fd, ...);
for (i=0; i<n_recs; i++)
    use(MappedFile[i]);
```

Traditional I/O involves explicit calls to read and write, which in turn means that data is accessed via a buffer; in fact, two buffers are usually employed: data is transferred between a user buffer and a kernel buffer, and between the kernel buffer and the I/O device.

An alternative approach is to ***map*** a file into a process's address space: the file provides the data for a portion of the address space and the kernel's virtual-memory system is responsible for the I/O. A major benefit of this approach is that data is transferred directly from the device to where the user needs it; there is no need for an extra system buffer.



Mmap System Call

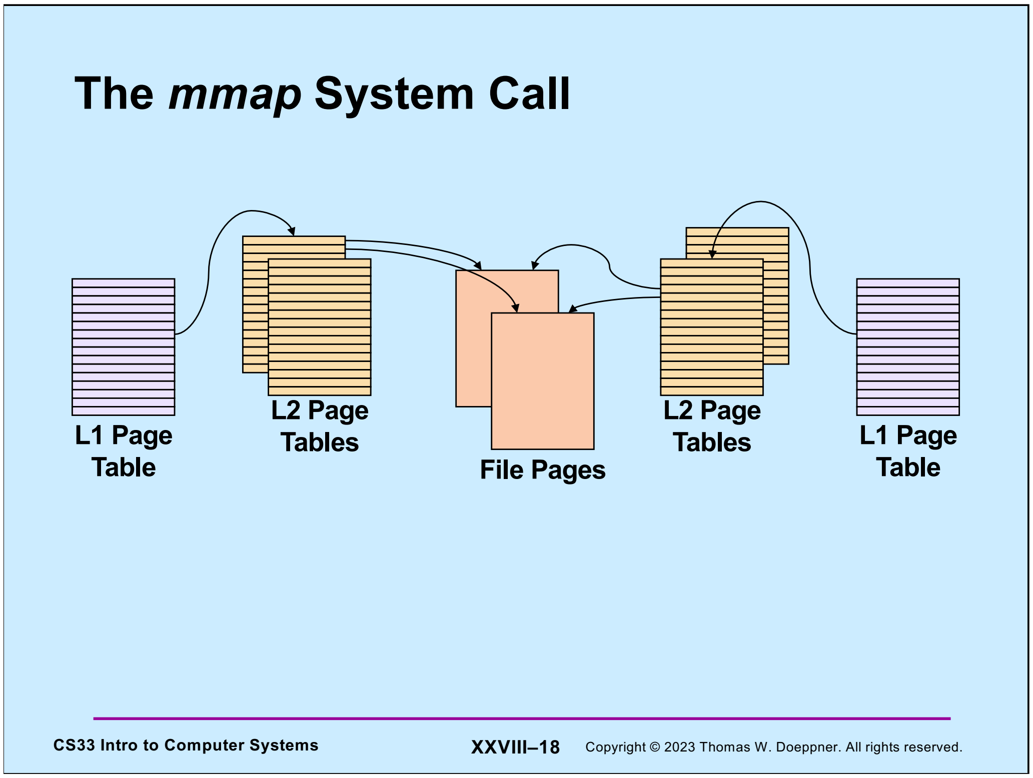
```
void *mmap(
  void *addr,
    // where to map file (0 if don’t care)
  size_t len,
    // how much to map
  int prot,
    // memory protection (read, write, exec.)
  int flags,
    // shared vs. private, plus more
  int fd,
    // which file
  off_t off
    // starting from where
  );
```

Mmap maps the file given by **fd**, starting at position **off**, for **len** bytes, into the caller’s address space starting at location **addr**

- **len** is rounded up to a multiple of the page size
- **off** must be page-aligned
- if **addr** is zero, the kernel assigns an address
- if **addr** is positive, it is a suggestion to the kernel as to where the mapped file should be located (it usually will be aligned to a page). However, if **flags** includes MAP_FIXED, then **addr** is not modified by the kernel (and if its value is not reasonable, the call fails)
- the call returns the address of the beginning of the mapped file

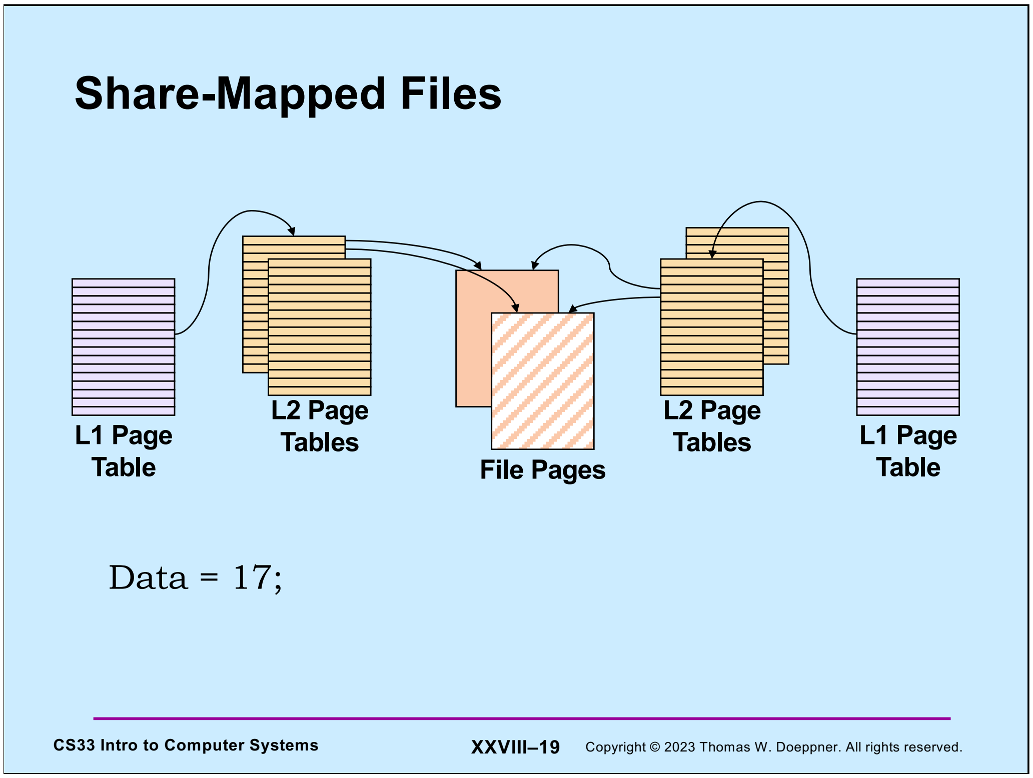
The **flags** argument must include either MAP_SHARED or MAP_PRIVATE (but not both). If it’s MAP_SHARED, then the mapped portion of the caller’s address space contains the current contents of the file; when the mapped portion of the address space is modified by the process, the corresponding portion of the file is modified.

However, if **flags** includes MAP_PRIVATE, then the idea is that the mapped portion of the address space is initialized with the contents of the file, but that changes made to the mapped portion of the address space by the process are private and not written back to the file. The details are a bit complicated: as long as the mapping process does not modify any of the mapped portion of the address space, the pages contained in it contain the current contents of the corresponding pages of the file. However, if the process modifies a page, then that particular page no longer contains the current contents of the corresponding file page, but contains whatever modifications are made to it by the process. These changes are not written back to the file and not shared with any other process that has mapped the file. It’s unspecified what the situation is for other pages in the mapped region after one of them is modified. Depending on the implementation, they might continue to contain the current contents of the corresponding pages of the file until they, themselves, are modified. Or they might also be treated as if they’d just been written to and thus no longer be shared with others.



The *mmap* System Call

The **mmap** system call maps a file into a process's address space. All processes mapping the same file can share the pages of the file.



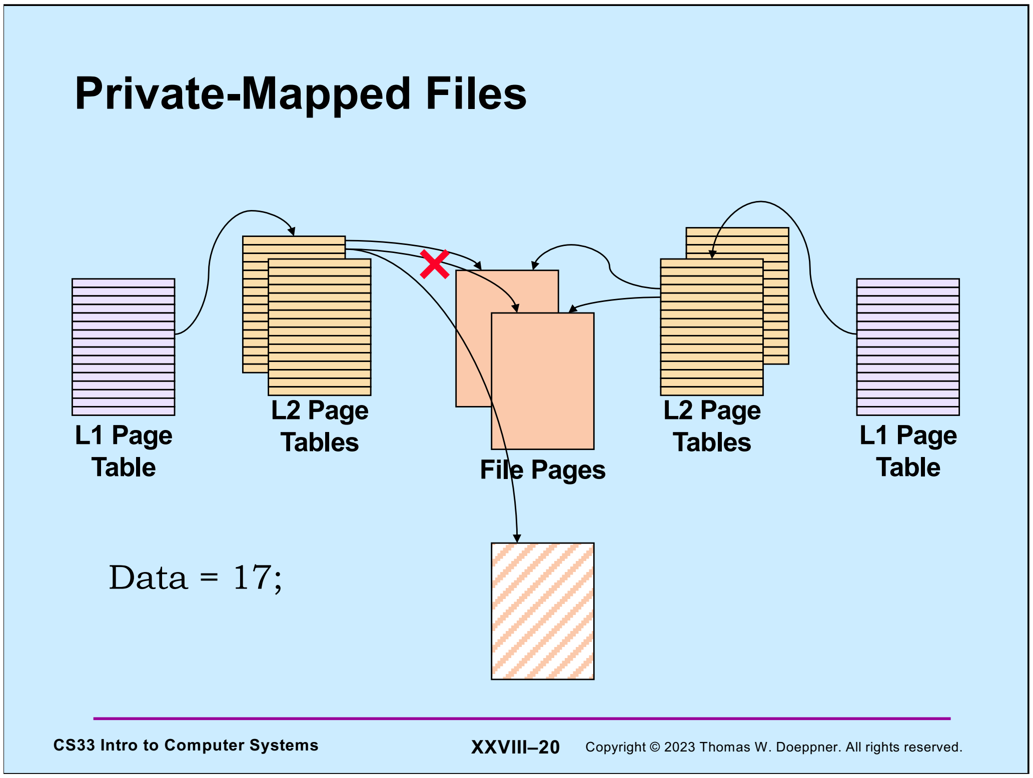
Share-Mapped Files

Data = 17;

Here, **Data** is a variable located in the highlighted file page.

There are a couple options for how modifications to mmapped files are dealt with. The most straightforward is the **share** option in which changes to mmapped file pages modify the file and hence the changes are seen by the other processes who have share-mapped the file.

Hence, the change to **Data** is seen by both processes mapping the file.

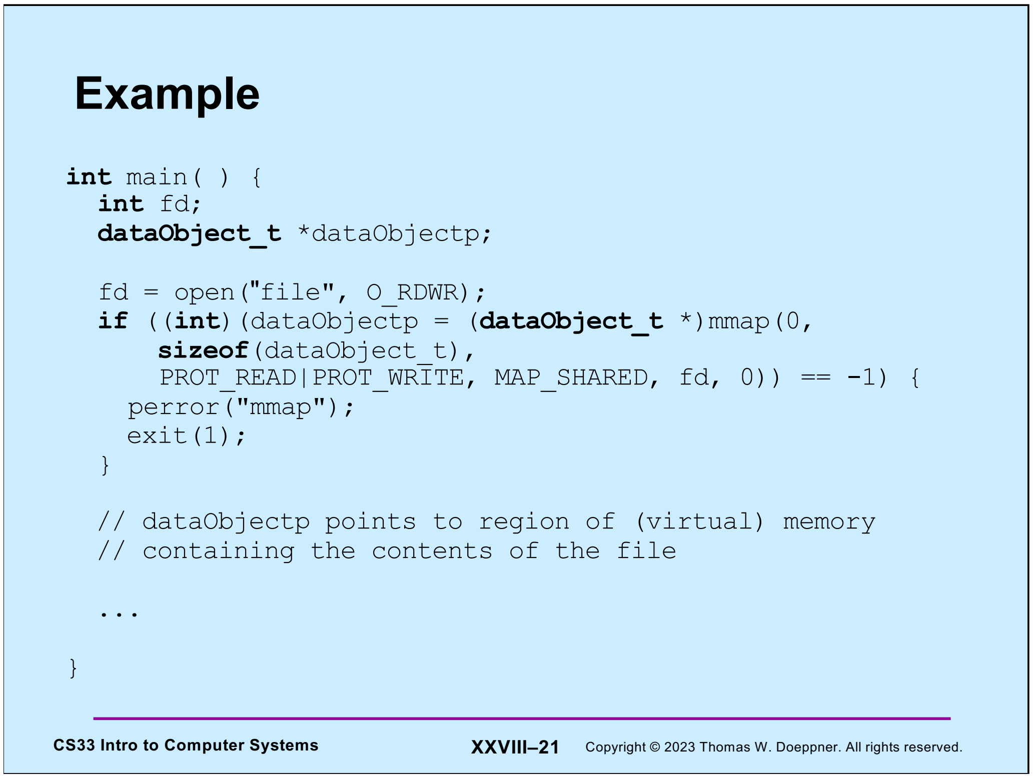


Private-Mapped Files

Data = 17;

The other option is to **private**-map the file: changes made to mmapped file pages do not modify the file. Instead, when a page of a file is first modified via a private mapping, a copy of just that page is made for the modifying process, but this copy is not seen by other processes, nor does it appear in the file.

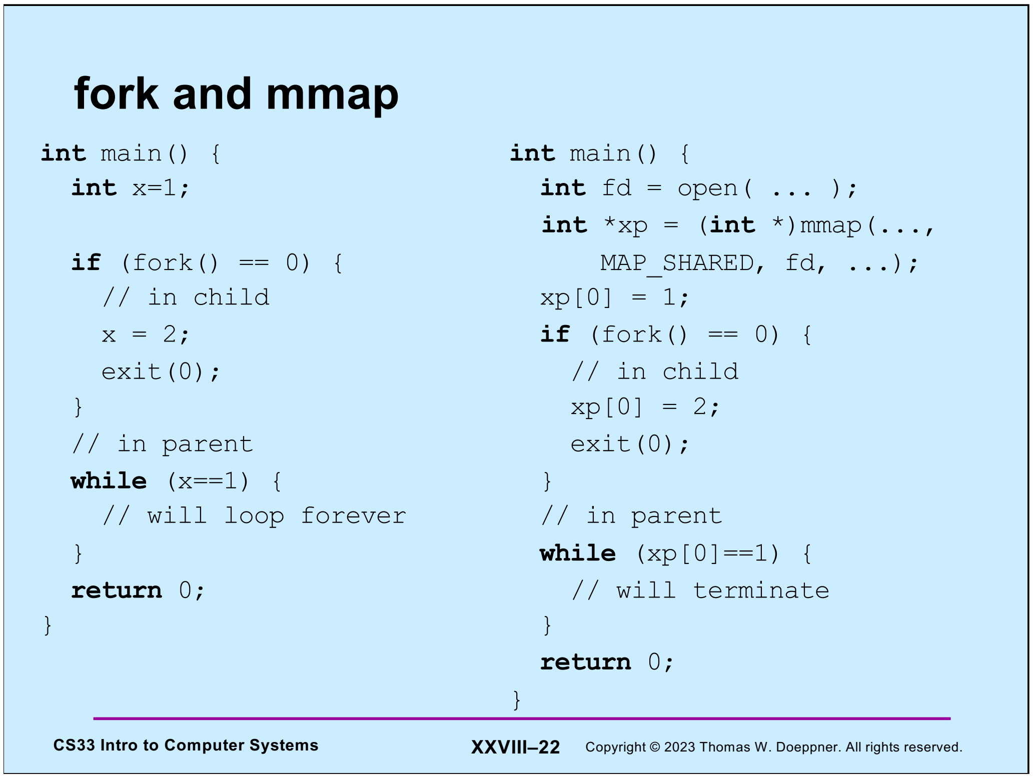
In the slide, the process on the left has private-mapped the file. Thus, its changes to **Data** (in the private-mapped portion of the address space) are made to a copy of the page containing Data. Thus, the other process will continue to see the original Data.



Example

```
int main( ) {
  int fd;
  dataObject_t *dataObjectp;

  fd = open("file", O_RDWR);
  if ((int)(dataObjectp = (dataObject_t *)mmap(0,
      sizeof(dataObject_t),
      PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)) == -1) {
    perror("mmap");
    exit(1);
  }

  // dataObjectp points to region of (virtual) memory
  // containing the contents of the file

  ...

}
```

Here we map the contents of a file containing a dataObject_t into the caller’s address space, allowing it both read and write access. Note mapping the file into memory does not cause any immediate I/O to take place. The operating system will perform the I/O when necessary, according to its own rules.

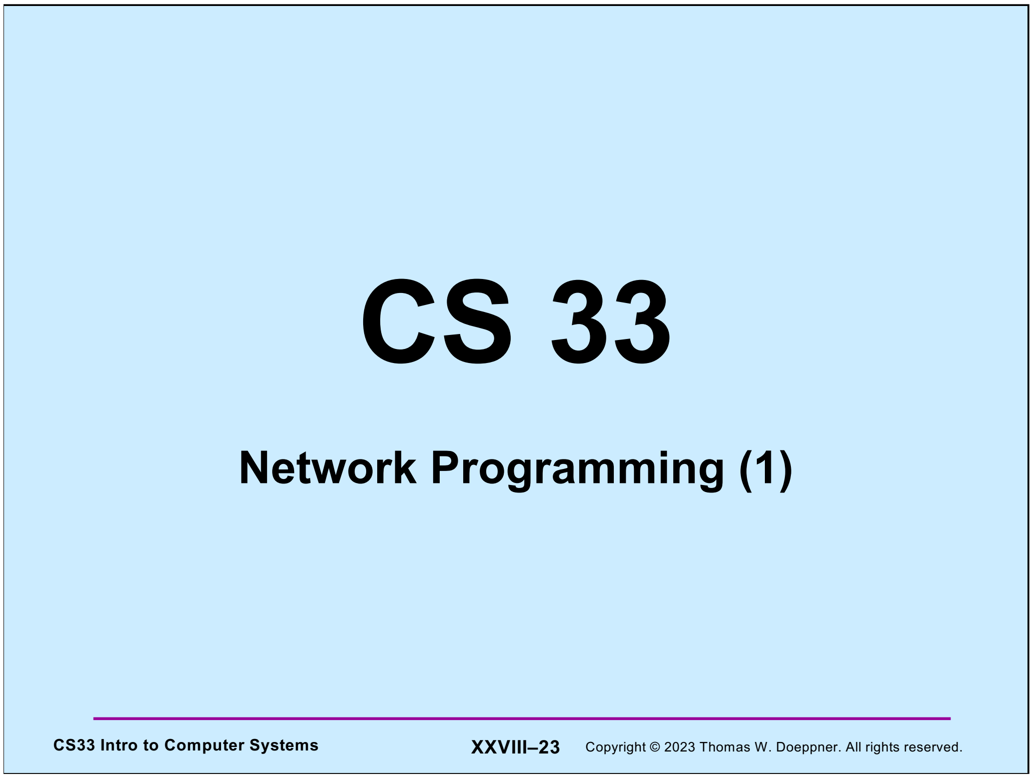


fork and mmap

```
int main() {
  int x=1;

  if (fork() == 0) {
    // in child
    x = 2;
    exit(0);
  }
  // in parent
  while (x==1) {
    // will loop forever
  }
  return 0;
}
```

```
int main() {
  int fd = open( ... );
  int *xp = (int *)mmap(...,
      MAP_SHARED, fd, ...);
  xp[0] = 1;
  if (fork() == 0) {
    // in child
    xp[0] = 2;
    exit(0);
  }
  // in parent
  while (xp[0]==1) {
    // will terminate
  }
  return 0;
}
```

When a process calls fork and creates a child, the child's address space is normally a copy of the parent's. Thus changes made by the child to its address space will not be seen in the parent's address space (as shown in the left-hand column). However, if there is a region in the parent's address space that has been mmapped using the MAP_SHARED flag, and subsequently the parent calls fork and creates a child, the mmapped region is not copied but is shared by parent and child. Thus changes to the region made by the child will be seen by the parent (and vice versa).

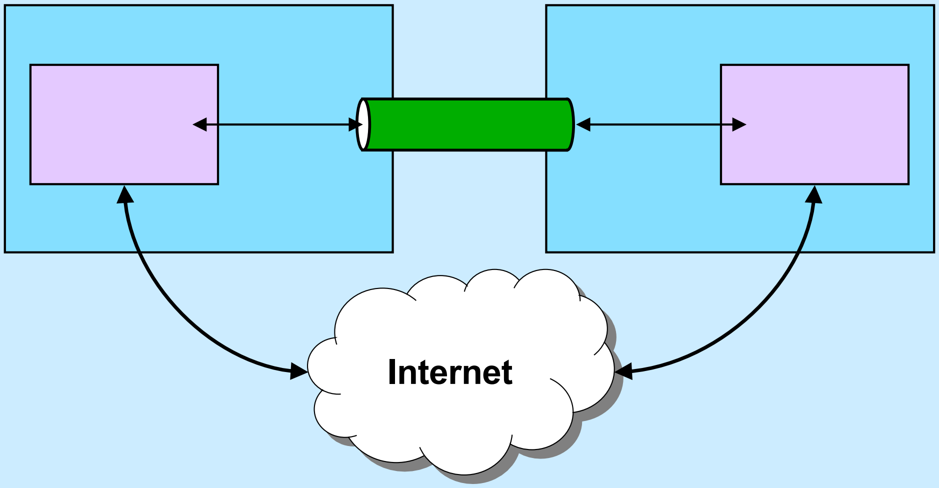


CS 33

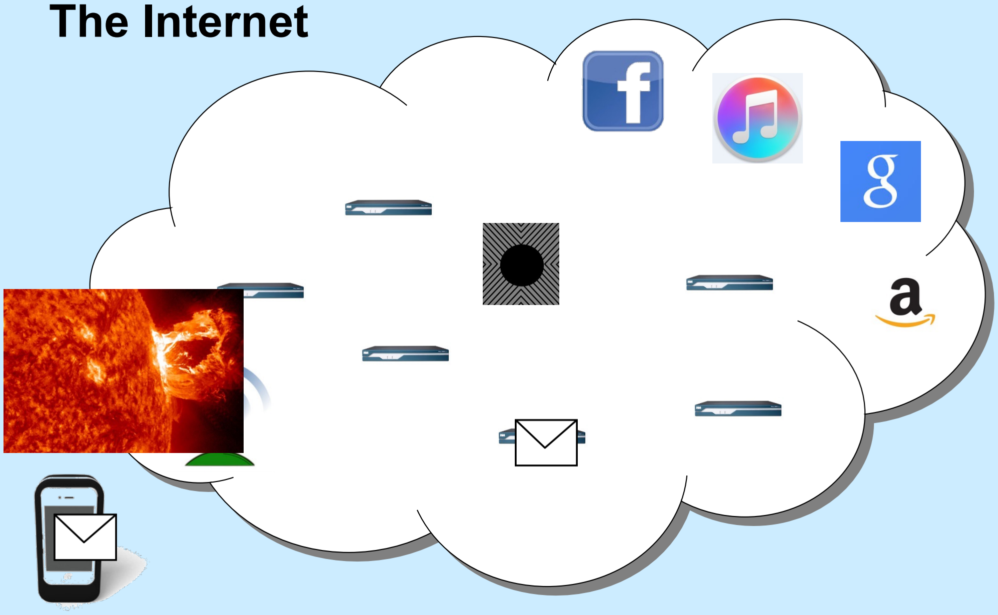
Network Programming (1)

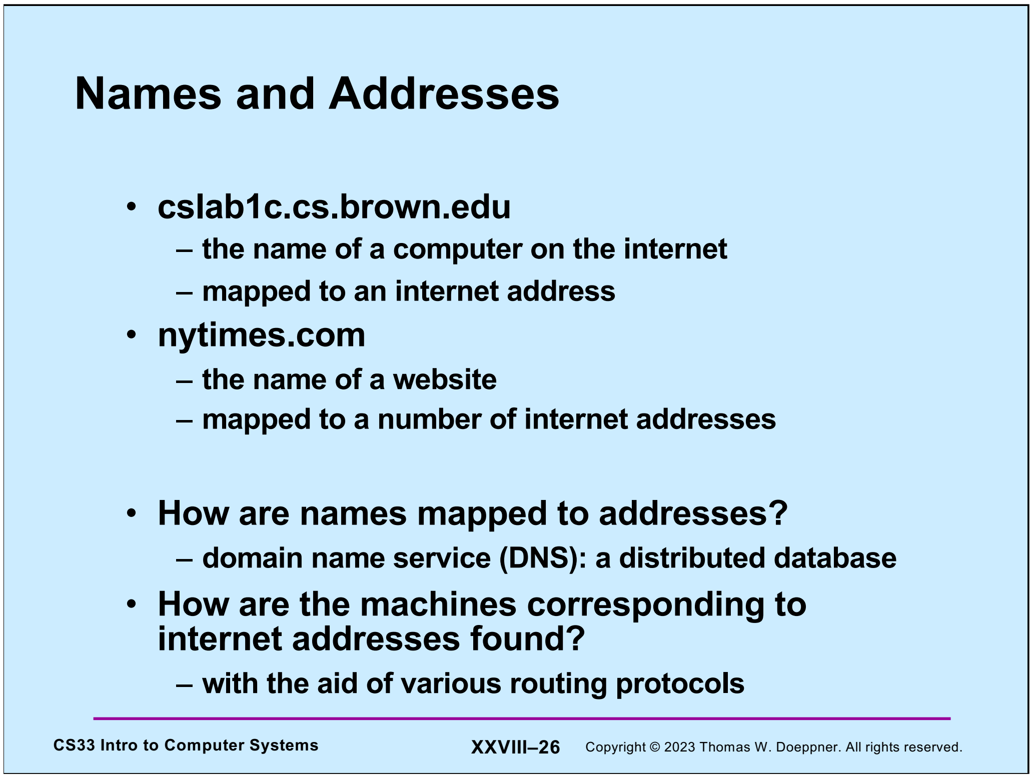
The source code used in this lecture, as well as some additional related source code, is on the course web page.

Communicating Over the Internet



The Internet





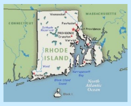
Names and Addresses

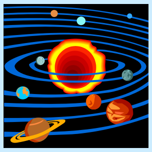
- cslab1c.cs.brown.edu
  - – the name of a computer on the internet
  - – mapped to an internet address
- nytimes.com
  - – the name of a website
  - – mapped to a number of internet addresses

- How are names mapped to addresses?
  - – domain name service (DNS): a distributed database
- How are the machines corresponding to internet addresses found?
  - – with the aid of various routing protocols

Internet Addresses

- IP (internet protocol) address
  - one per network interface
  - 32 bits (IPv4)
    - » 5527 per acre of RI
    - » 25 per acre of Texas
  - 128 bits (IPv6)
    - » 1.6 billion per cubic mile of a sphere whose radius is the mean distance from the Sun to the (former) planet Pluto





- Port number
  - one per service instance per machine
  - 16 bits
    - » port numbers less than 1024 are reserved for privileged applications

Notation

- Addresses (assume IPv4: 32-bit addresses)
  - written using dot notation
    - 128.48.37.1
      - dots separate bytes
  - address plus port (1426):
    - 128.48.37.1:1426

Reliability

- Two possibilities
  - – don’t worry about it
    - » just send it
      - • if it arrives at its destination, that’s good!
        - – no verification
  - – worry about it
    - » keep track of what’s been successfully communicated
      - • receiver “acks”
    - » retransmit until
      - • data is received

      or

      - • it appears that “the network is down”

Reliability vs. Unreliability

- Reliable communication
  - good for
    - email
    - texting
    - distributed file systems
    - web pages
  - bad for
    - streaming audio
    - streaming video

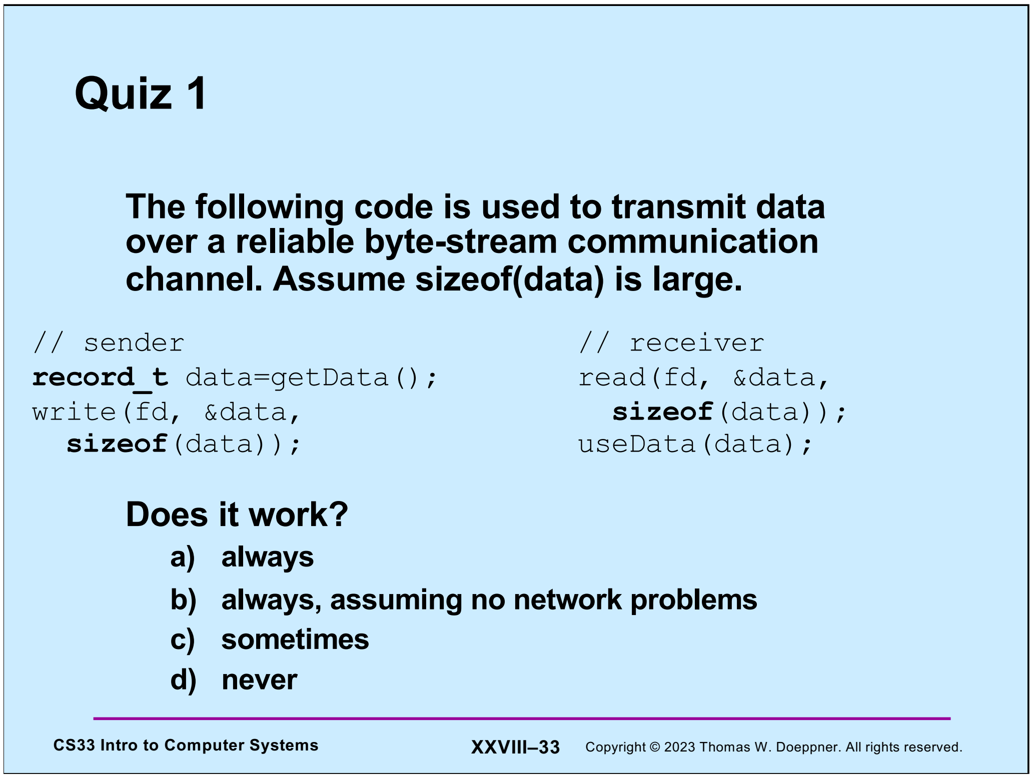
    a little noise is better than a long pause

The Data Abstraction

- Byte stream
  - sequence of bytes
    - as in pipes
  - any notion of a larger data aggregate is the responsibility of the programmer
- Discrete records
  - sequence of variable-size “records”
  - boundaries between records maintained
  - receiver receives discrete records, as sent by sender

What’s Supported

- Stream
  - byte-stream data abstraction
  - reliable transmission
- Datagram
  - discrete-record data abstraction
  - unreliable transmission



Quiz 1

The following code is used to transmit data over a reliable byte-stream communication channel. Assume sizeof(data) is large.

```
// sender                         // receiver
record_t data=getData();          read(fd, &data,
write(fd, &data,                    sizeof(data));
  sizeof(data));                  useData(data);
```

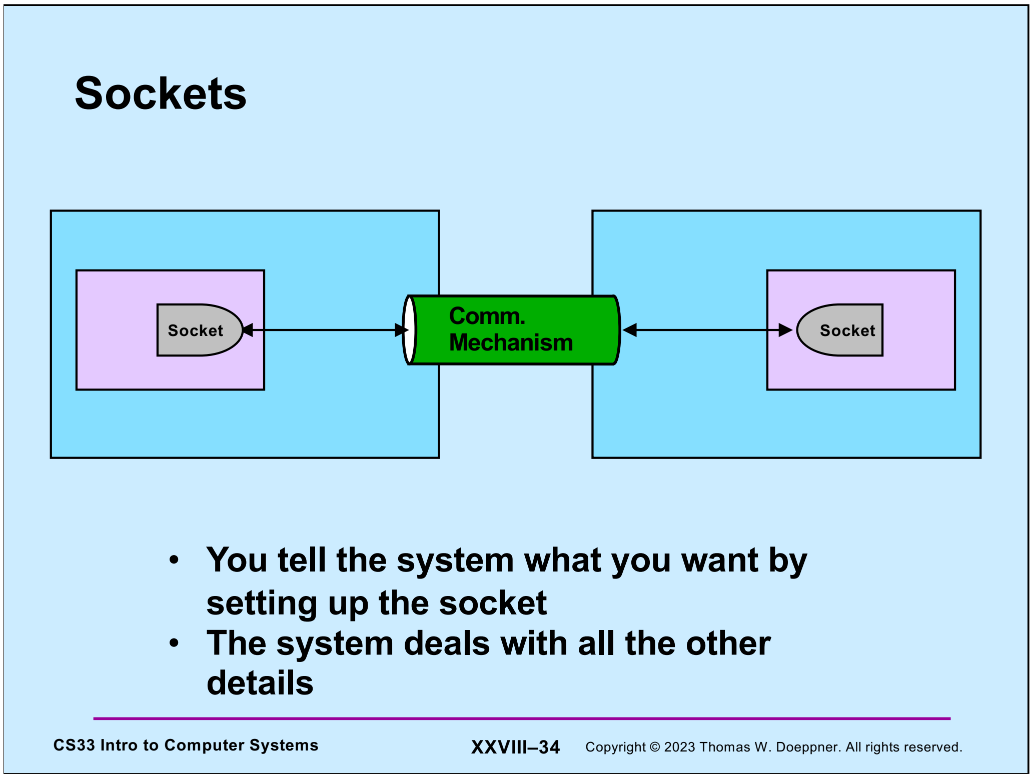
Does it work?

a) always

b) always, assuming no network problems

c) sometimes

d) never



Sockets

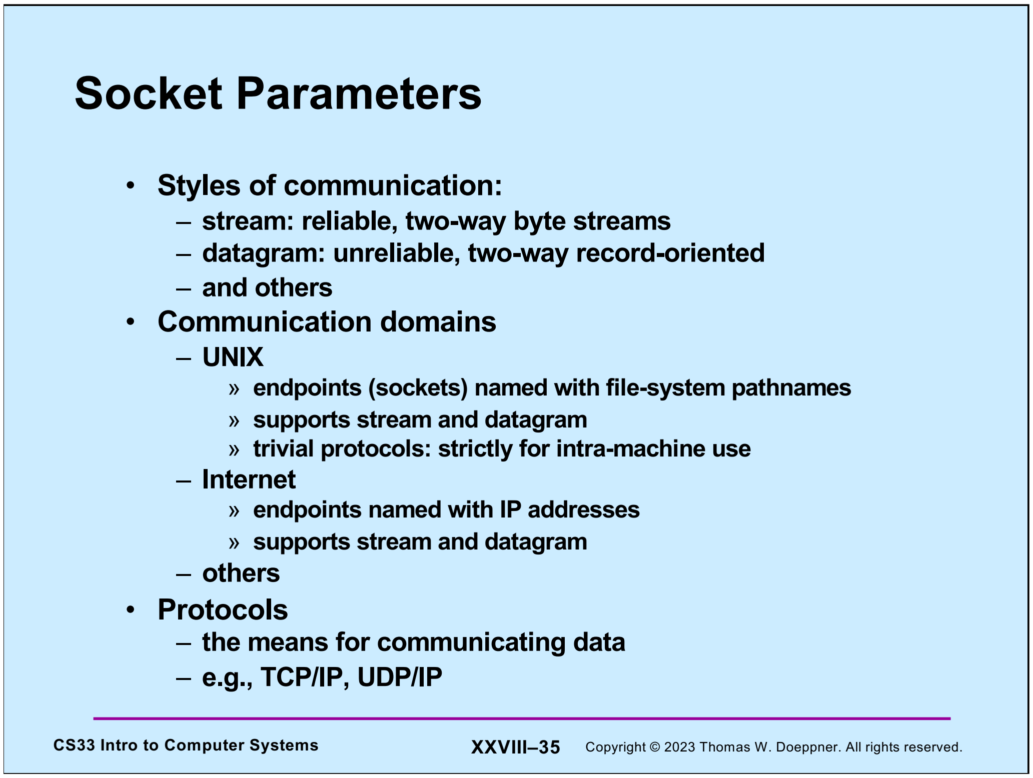
Socket

Comm. Mechanism

Socket

- You tell the system what you want by setting up the socket
- The system deals with all the other details

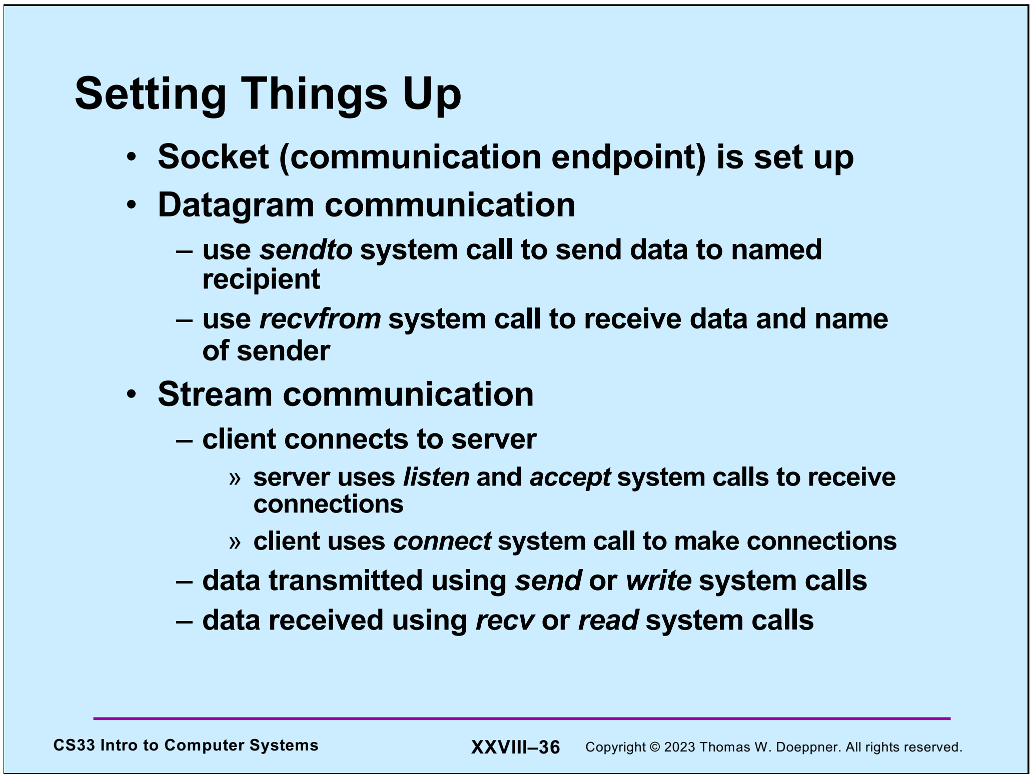
Sockets are the abstraction of the communication path. An application sets up a socket as the basis for communication. It refers to it via a file descriptor.



Socket Parameters

- Styles of communication:
  - stream: reliable, two-way byte streams
  - datagram: unreliable, two-way record-oriented
  - and others
- Communication domains
  - UNIX
    - endpoints (sockets) named with file-system pathnames
    - supports stream and datagram
    - trivial protocols: strictly for intra-machine use
  - Internet
    - endpoints named with IP addresses
    - supports stream and datagram
  - others
- Protocols
  - the means for communicating data
  - e.g., TCP/IP, UDP/IP

We focus strictly on the internet domain.

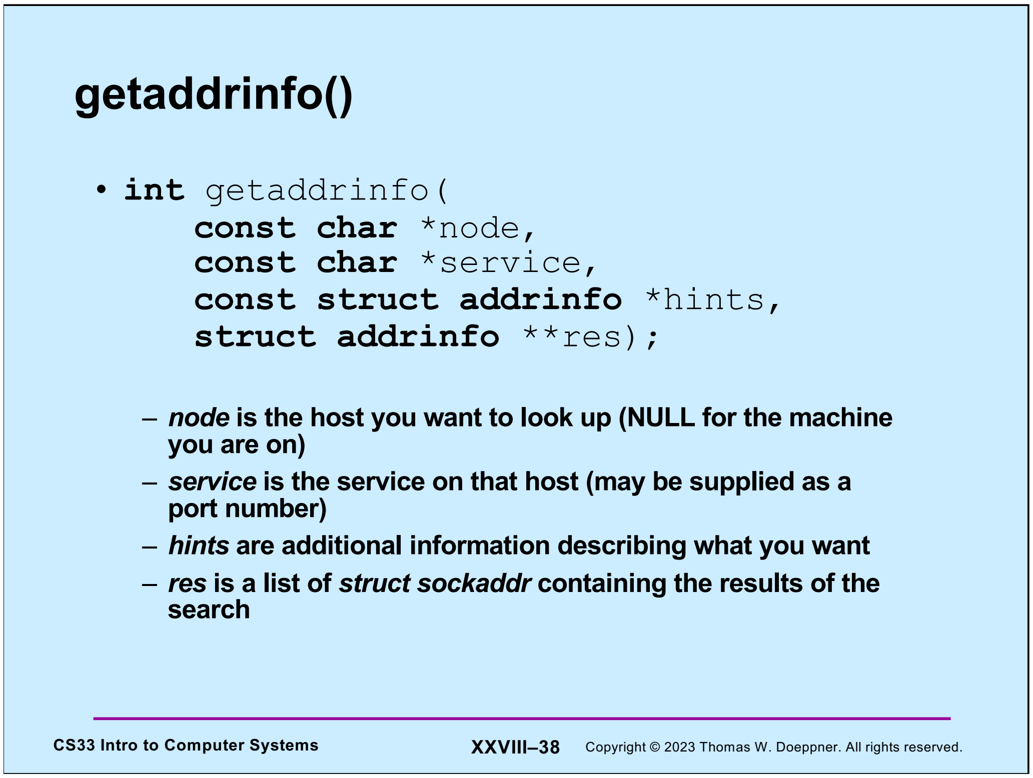


Setting Things Up

- Socket (communication endpoint) is set up
- Datagram communication
  - use *sendto* system call to send data to named recipient
  - use *recvfrom* system call to receive data and name of sender
- Stream communication
  - client connects to server
    - server uses *listen* and *accept* system calls to receive connections
    - client uses *connect* system call to make connections
  - data transmitted using *send* or *write* system calls
  - data received using *recv* or *read* system calls

Socket Addresses

- `struct sockaddr`
  - represents a network address
  - many sorts
    - » we use struct *sockaddr_in*
  - we can ignore the details
    - » embedded in layers of software
- getaddrinfo()
  - function used to obtain `struct sockaddr`’s

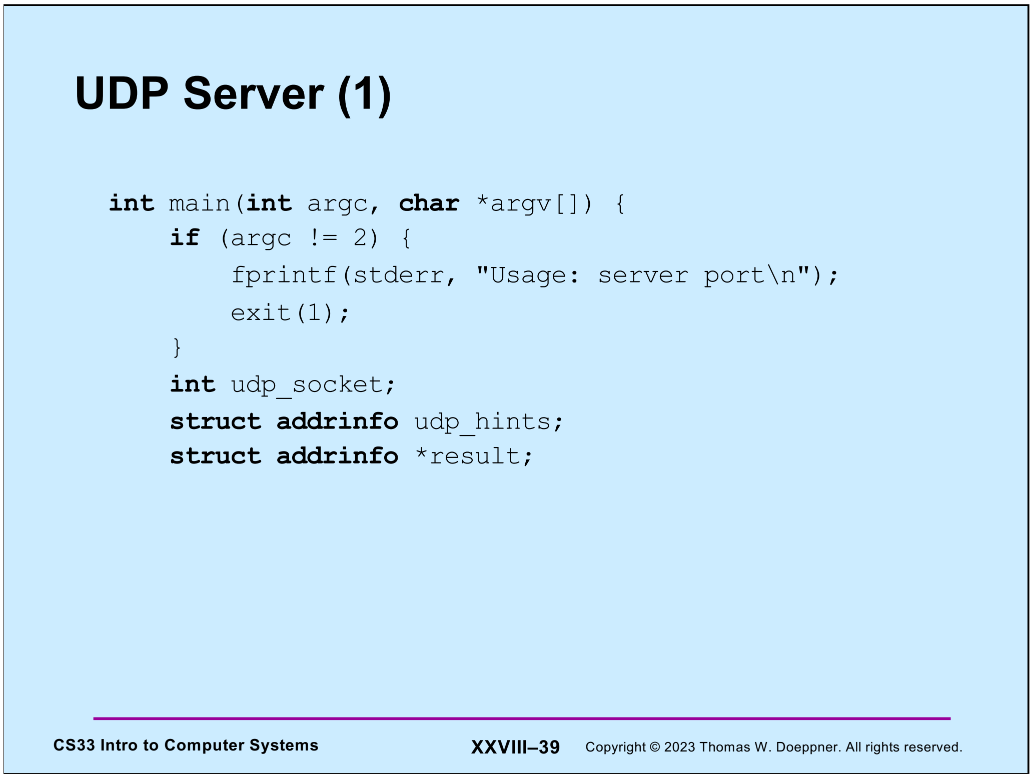


getaddrinfo()

```
• int getaddrinfo(
      const char *node,
      const char *service,
      const struct addrinfo *hints,
      struct addrinfo **res);
```

- *node* is the host you want to look up (NULL for the machine you are on)
- *service* is the service on that host (may be supplied as a port number)
- *hints* are additional information describing what you want
- *res* is a list of *struct sockaddr* containing the results of the search

The general idea of using **getaddrinfo** is that you supply the name of the host you'd like to contact (*node*), which service on that host (*service*), and a description of how you'd like to communicate (**hints**). It returns a list of possible means for contacting the server in the form of a list of addrinfo structures (**res**). If the node argument is neither NULL nor the name of the local machine, **getaddrinfo** looks up what it needs in the domain name service (DNS) – the internet-wide distributed name service.

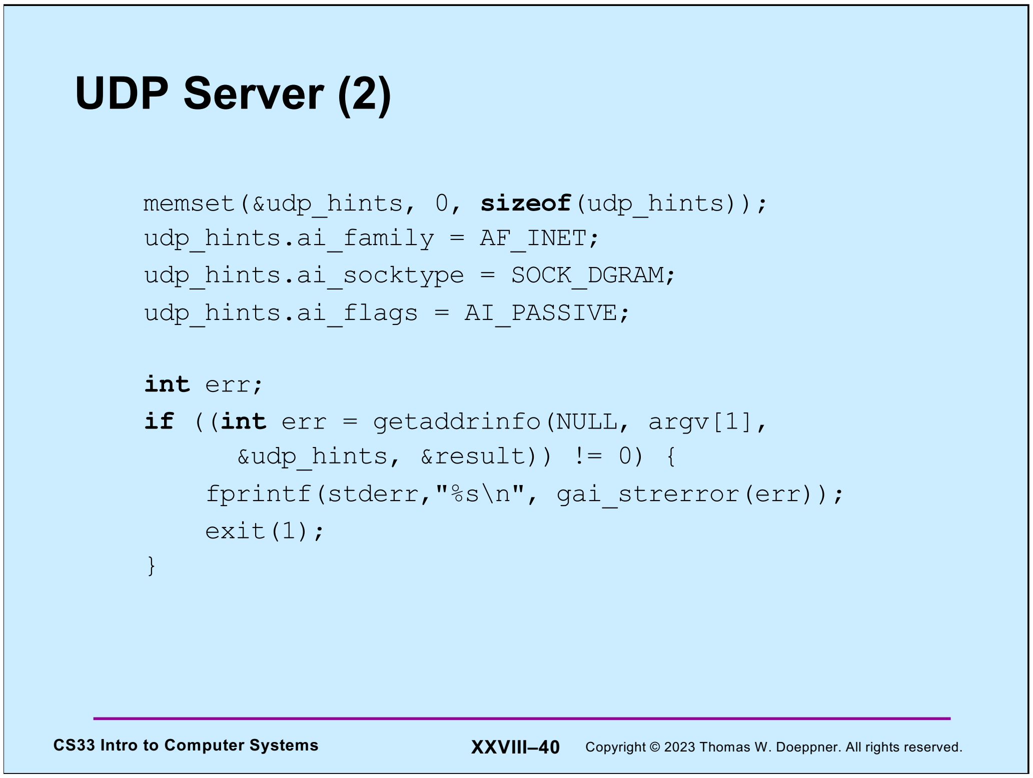


UDP Server (1)

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

Here we begin an example of a simple UDP server that receives messages from clients, prints them along with an indication of who sent the message, and politely responds.

In this first slide we check that we're invoked correctly (the command line should include the port number we're expecting to receive messages on) and have some initial declarations.

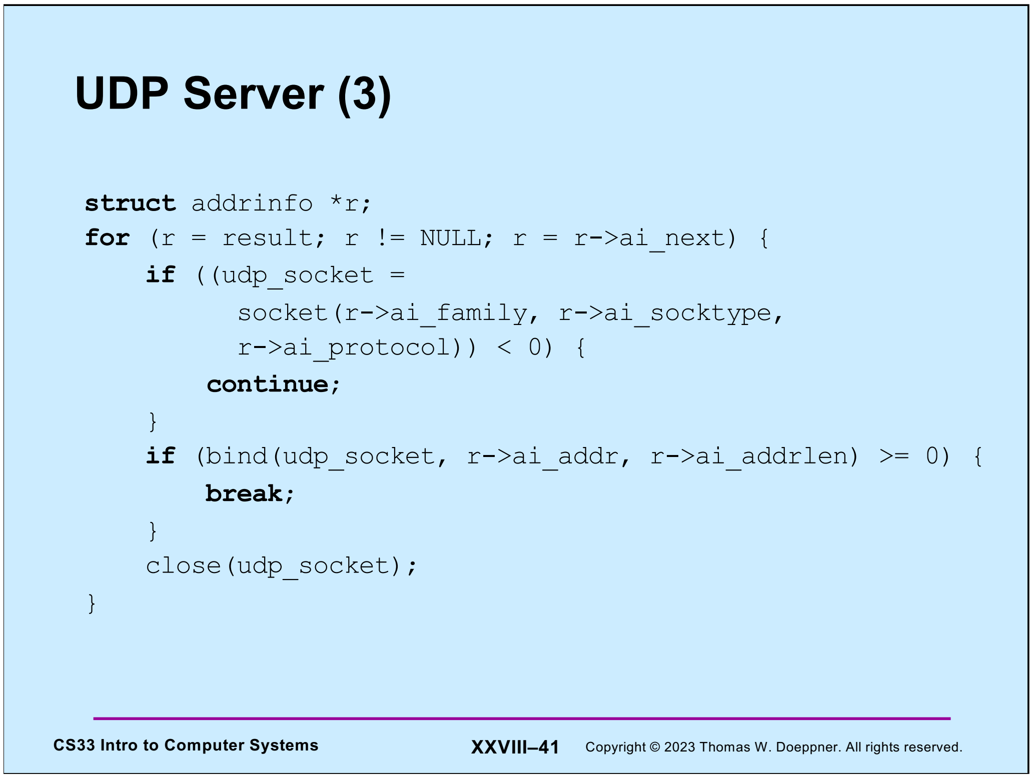


UDP Server (2)

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

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

The next step is to set up an address for our socket so that clients can contact us. In the *hints* structure, which we initialize to zeroes so that components we don't set are zero, we specify that we're using IPv4 (AF_INET), that we are using datagrams (which, over IPv4, means UDP). Setting the flags to AI_PASSIVE is a bit of magic that allows the server to receive messages from multiple sources.

We call **getaddrinfo** to get an appropriate address to bind to our socket (next slide). Its first (name) argument is NULL, which means that we want the address of the machine we're on. Note the use of **gai_strerror** to produce an error message given an error return from **getaddrinfo**.

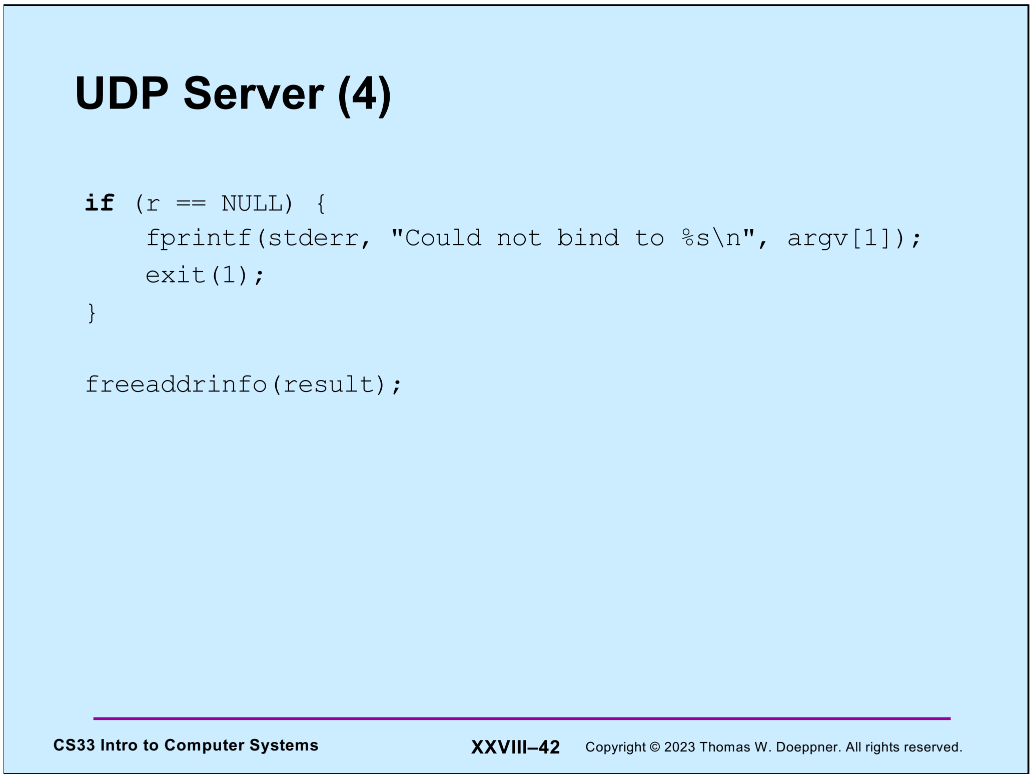


UDP Server (3)

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

Next, we iterate over the output of **getaddrinfo** (the list pointed to by its *result* argument). Though the length of this list is normally exactly one, it could be greater than one if our computer has multiple network interfaces. (The length could also be zero if it has no network interfaces, or none of the right sort.)

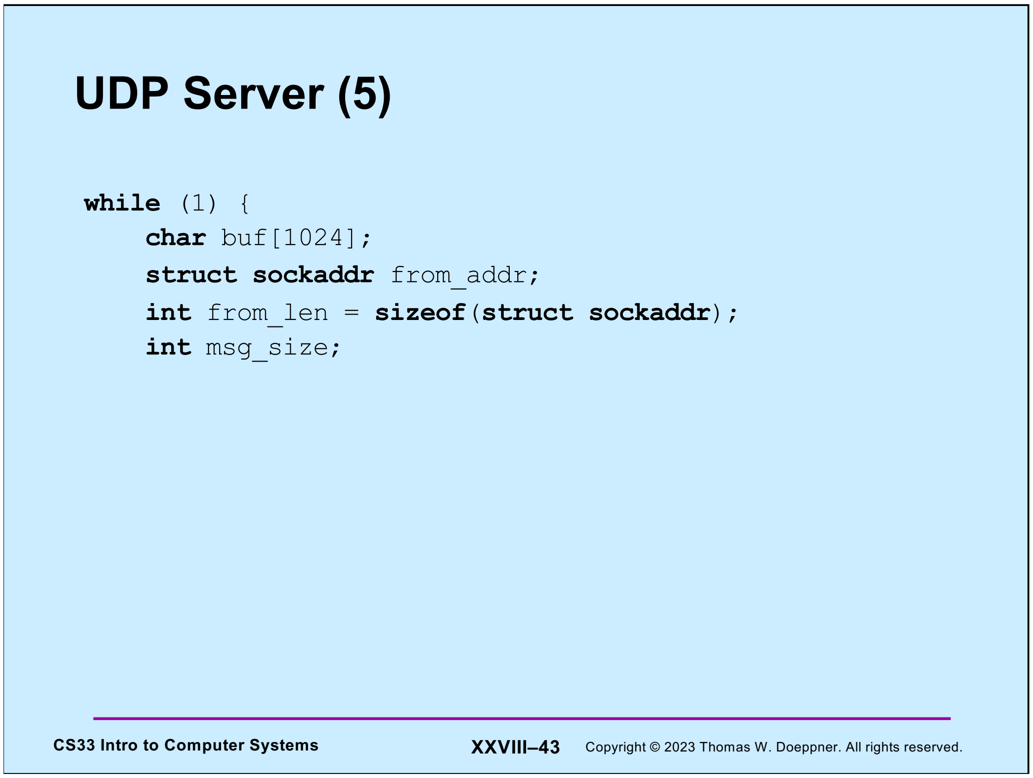
We try to create a socket that matches our desired socket type. Assuming we get the socket (which is referred to by the file descriptor **udp_socket**), we then try to bind it to the address returned by **getaddrinfo**. If all this works, we assume we're good to go. Otherwise, we try the next address in the list, if there are any more.



UDP Server (4)

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

freeaddrinfo(result);
```

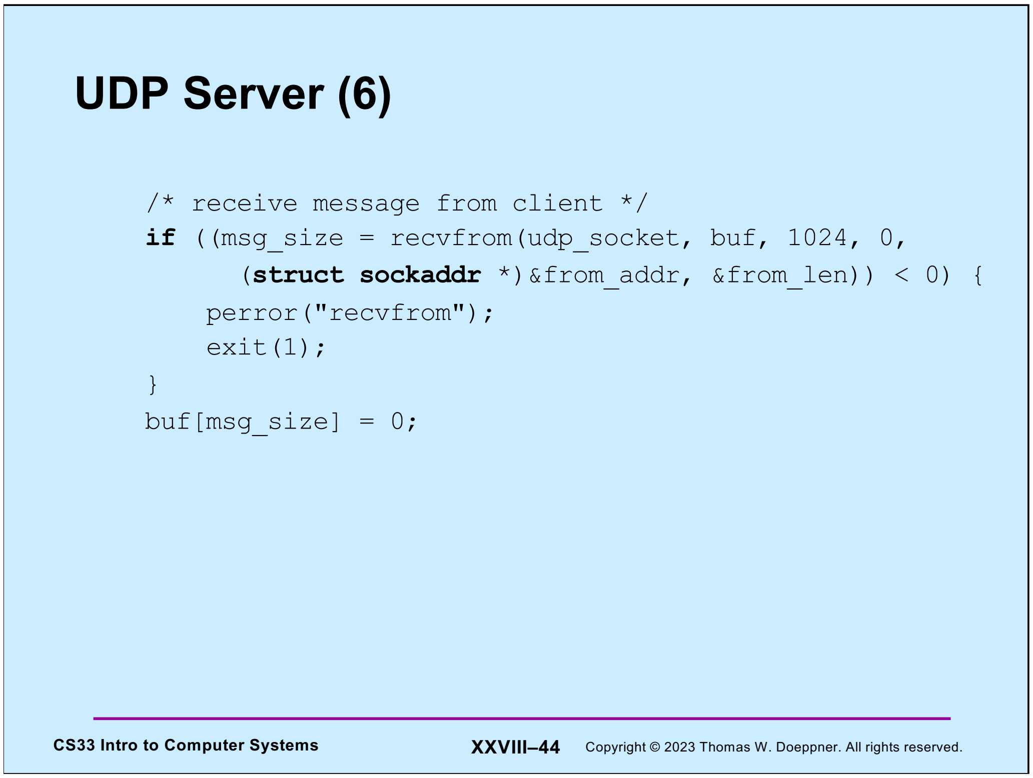
If we couldn't find anything that worked, we terminate the program. Otherwise, we free up the list of addresses, since we don't need them anymore. Note the use of **freeaddrinfo** for this purpose.



UDP Server (5)

```
while (1) {
    char buf[1024];
    struct sockaddr from_addr;
    int from_len = sizeof(struct sockaddr);
    int msg_size;
```

Now that we've set up a socket and bound it to an address that clients can send messages to, we enter a loop to deal with all the incoming messages.

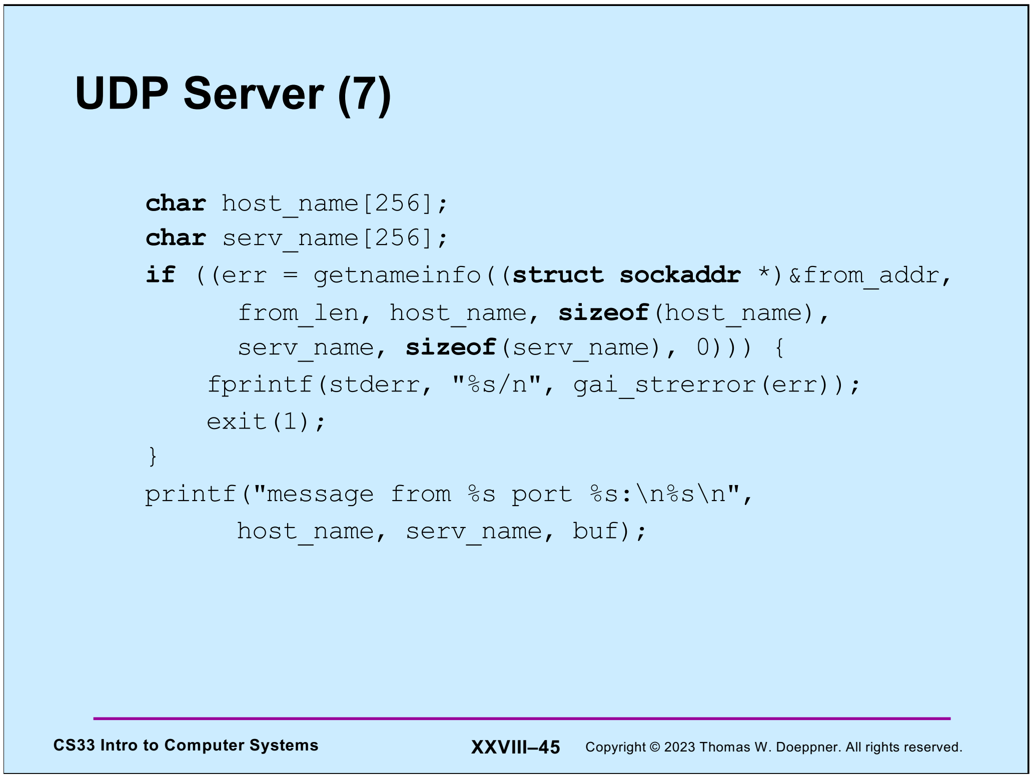


UDP Server (6)

```
/* receive message from client */
if ((msg_size = recvfrom(udp_socket, buf, 1024, 0,
      (struct sockaddr *)&from_addr, &from_len)) < 0) {
    perror("recvfrom");
    exit(1);
}
buf[msg_size] = 0;
```

We call **recfrom** (which is just like read, but with extra arguments) to get the next message from a client. The fourth argument could specify some flags, but we don't need any here (or in the networking lab). The fourth and fifth arguments, if not zeroes, give an address of memory to receive the network name of the caller, as well as its length. The length argument serves two purposes: on entry to the function, it indicates how much memory we have to receive the network address. On return from the function, it tells us how many bytes were actually used.

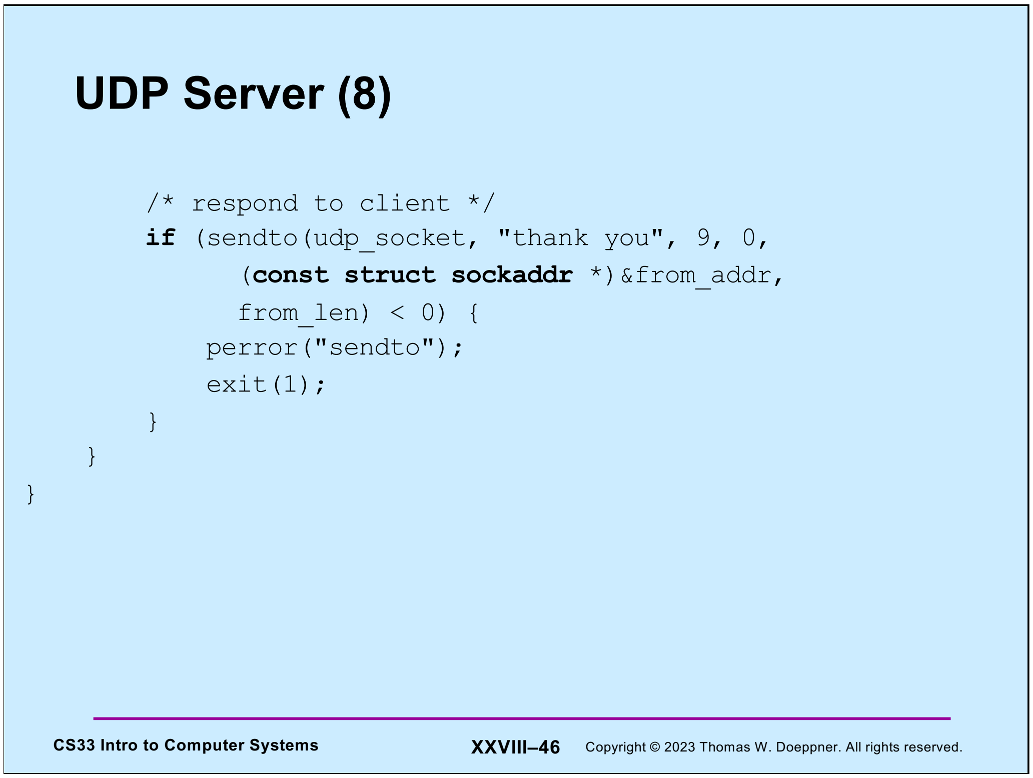
Note that we put a zero at the end of buf, so we can safely print it (next slide).



UDP Server (7)

```
char host_name[256];
char serv_name[256];
if ((err = getnameinfo((struct sockaddr *)&from_addr,
      from_len, host_name, sizeof(host_name),
      serv_name, sizeof(serv_name), 0))) {
    fprintf(stderr, "%s/n", gai_strerror(err));
    exit(1);
}
printf("message from %s port %s:\n%s\n",
      host_name, serv_name, buf);
```

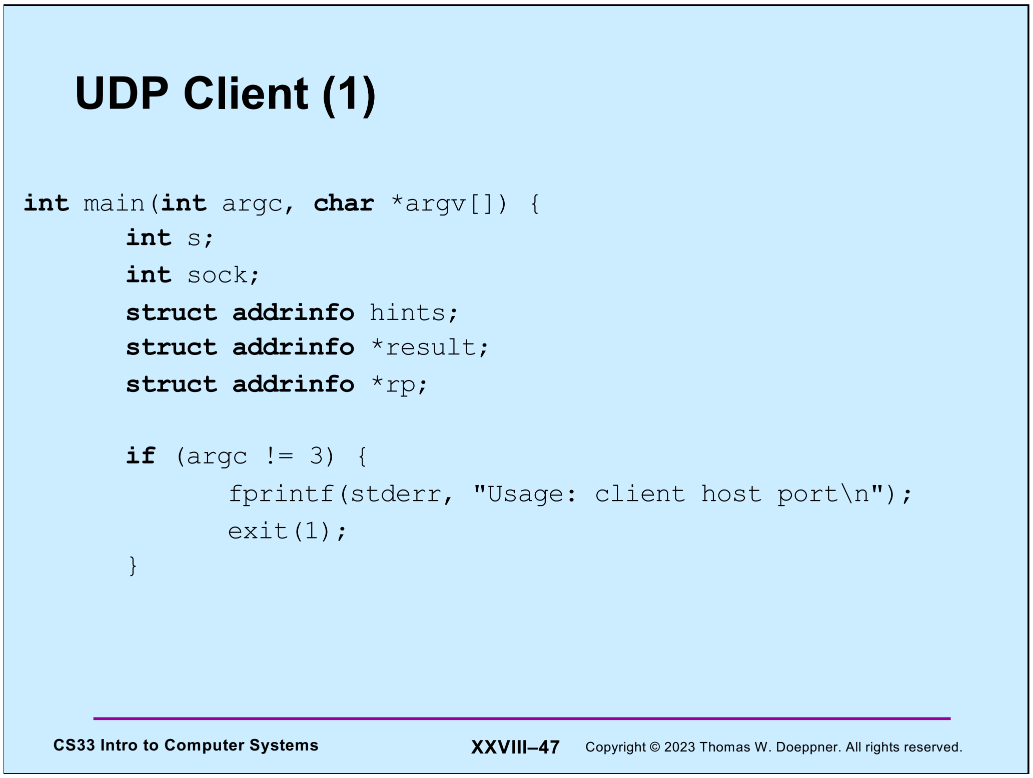
Next we print out who the client was and what its message was. The function **getnameinfo** is sort of the inverse of **getaddrinfo**: given a struct sockaddr (as produced by **recvfrom**), it tells us the name of the machine and the service requested (or port number). We then print the name of the machine, the service name (or port number), and the message itself. Note the use of **gai_strerror** for interpreting an error return from **getnameinfo**.



UDP Server (8)

```
        /* respond to client */
        if (sendto(udp_socket, "thank you", 9, 0,
              (const struct sockaddr *)&from_addr,
              from_len) < 0) {
            perror("sendto");
            exit(1);
        }
    }
}
```

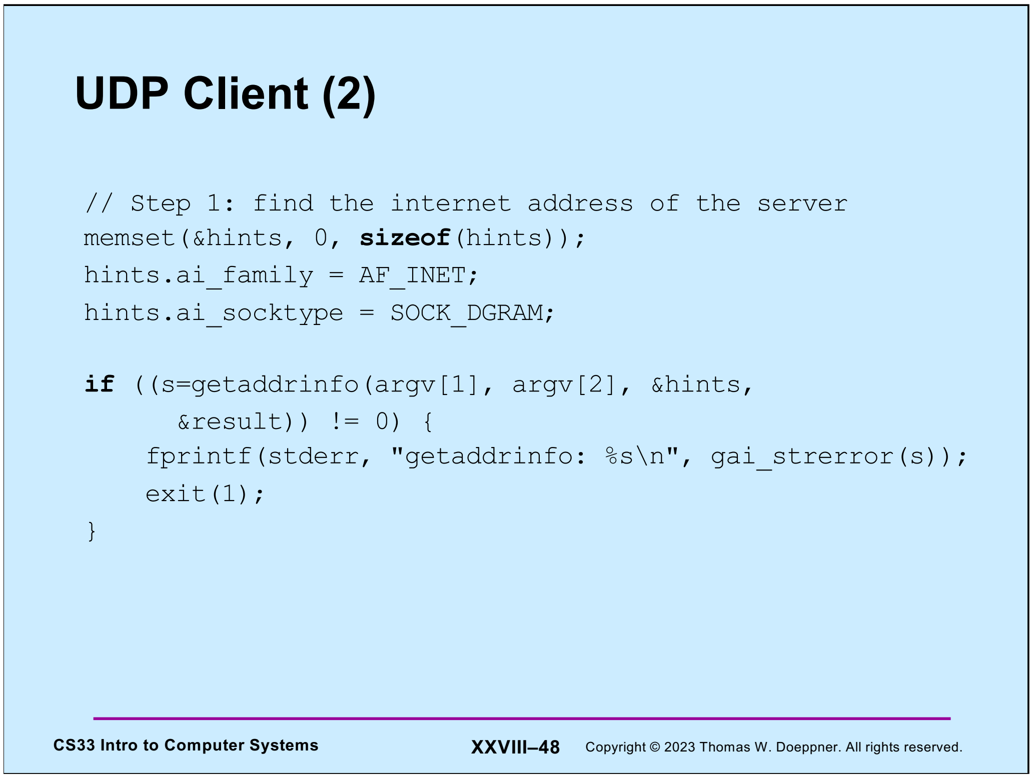
Finally, to be polite, we send a response to the client, thanking it for its message. The function **sendto** is like write, but with extra arguments. As with **recvfrom**, we set the flags argument (4th) to zero, but the next two arguments indicate whom we're sending the message to (the client, in this case).



UDP Client (1)

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

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

Now we look at the code for a client that communicates with our UDP server. Note that the command line of the client specifies both the host the server is on, as well as the port number. If the server is on the same host as the client, host may be specified as "localhost".

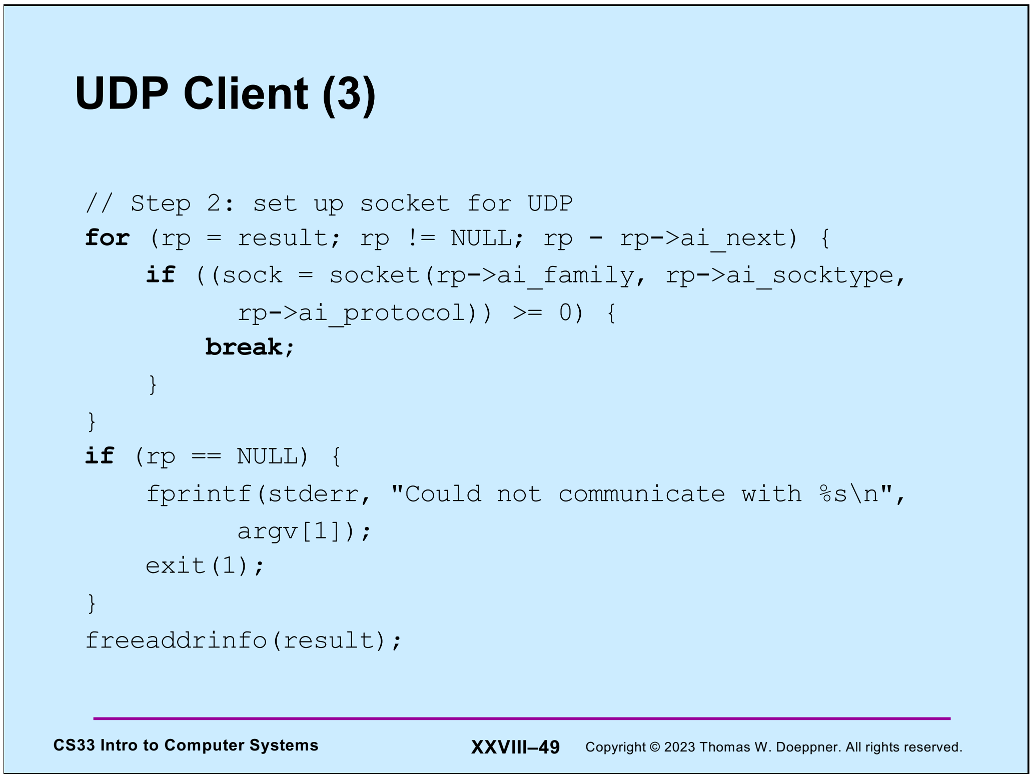


UDP Client (2)

```
// Step 1: find the internet address of the server
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_DGRAM;

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

We start by looking up the internet address of the server. To do this, we first fill in the hints structure to make it clear that we want a server with an internet (IPv4) interface and that we want UDP (datagrams). We call **getaddrinfo** to get a list of addresses. Again, note the use of **gai_strerror** to give us an error message.

Unlike what we did for the server code, we supply a non-null first argument to **getaddrinfo**, indicating which server we want to communicate with.

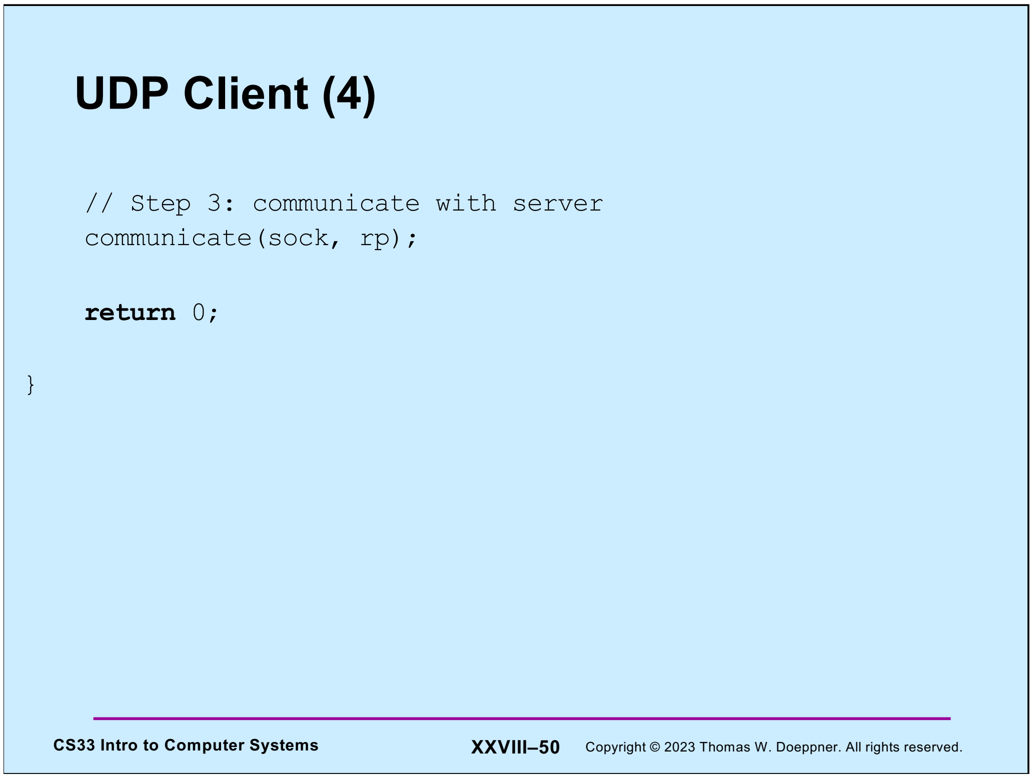


UDP Client (3)

```
// Step 2: set up socket for UDP
for (rp = result; rp != NULL; rp - rp->ai_next) {
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
          rp->ai_protocol)) >= 0) {
        break;
    }
}
if (rp == NULL) {
    fprintf(stderr, "Could not communicate with %s\n",
          argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

Next, we go through the addresses returned by **getaddrinfo** and use the first one for which we can successfully set up a socket. The list's length is usually one, and that one usually works.

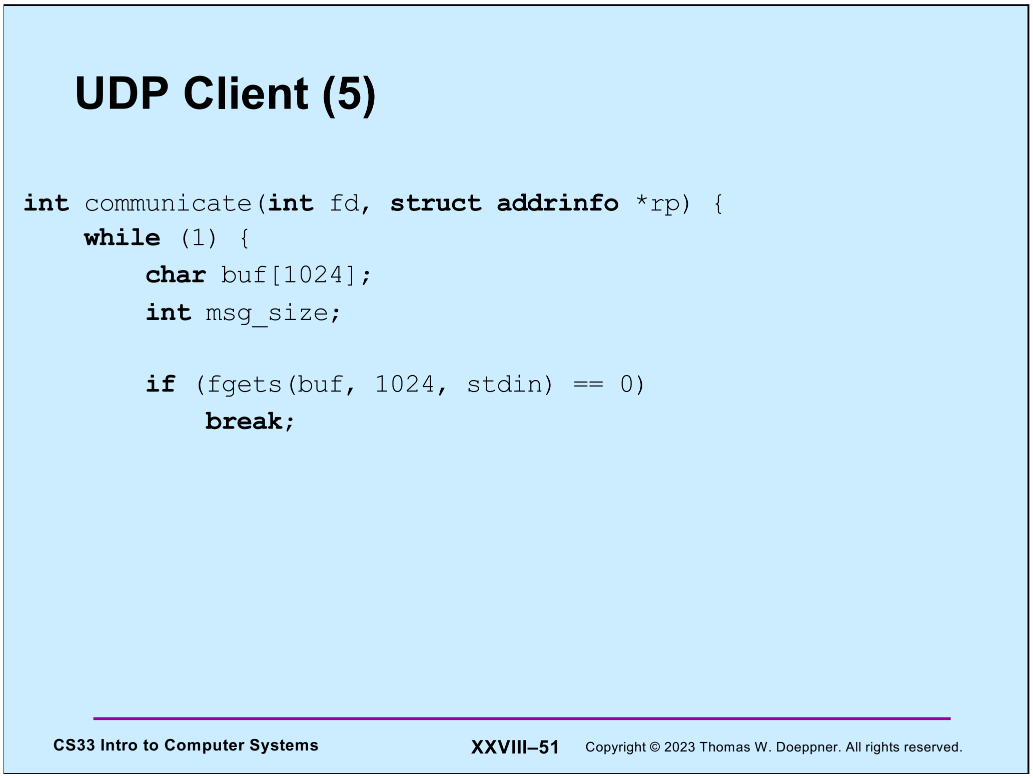
We free up list (by calling **freeaddrinfo**) since we no longer need it.



UDP Client (4)

```
    // Step 3: communicate with server
    communicate(sock, rp);

    return 0;

}
```

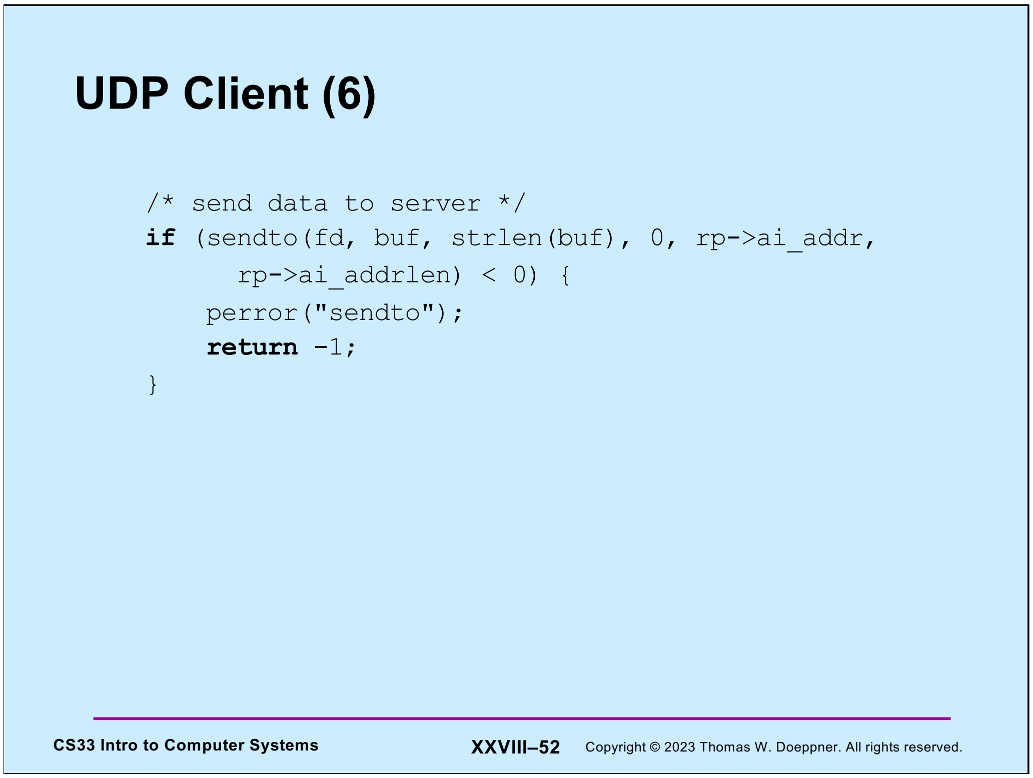
Next, we call our communicate function that will exchange messages with the server (although we don't know yet whether the server is up and running).



UDP Client (5)

```
int communicate(int fd, struct addrinfo *rp) {
    while (1) {
        char buf[1024];
        int msg_size;

        if (fgets(buf, 1024, stdin) == 0)
            break;
```

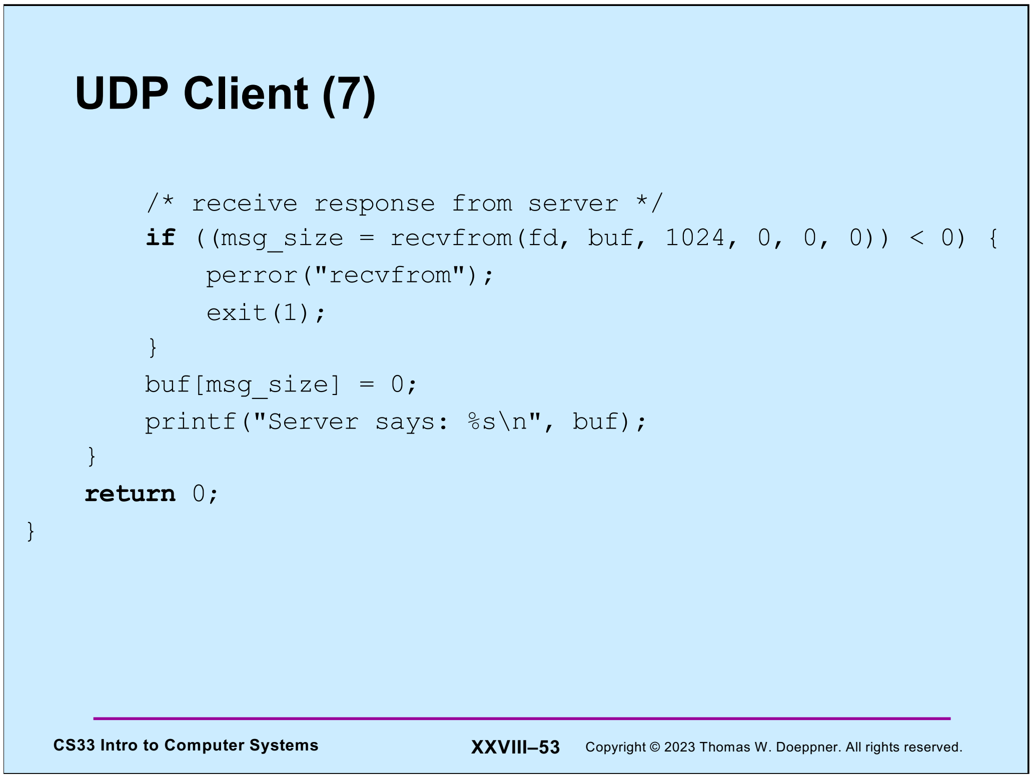
In our **communicate** function, we first read a line from stdin (which will be sent to the server).



UDP Client (6)

```
/* send data to server */
if (sendto(fd, buf, strlen(buf), 0, rp->ai_addr,
      rp->ai_addrlen) < 0) {
    perror("sendto");
    return -1;
}
```

The client sends to the server what was just read from stdin.



UDP Client (7)

```
        /* receive response from server */
        if ((msg_size = recvfrom(fd, buf, 1024, 0, 0, 0)) < 0) {
            perror("recvfrom");
            exit(1);
        }
        buf[msg_size] = 0;
        printf("Server says: %s\n", buf);
    }
    return 0;
}
```

The client receives the server's response, makes sure it's null-terminated, and prints it out.

Quiz 2

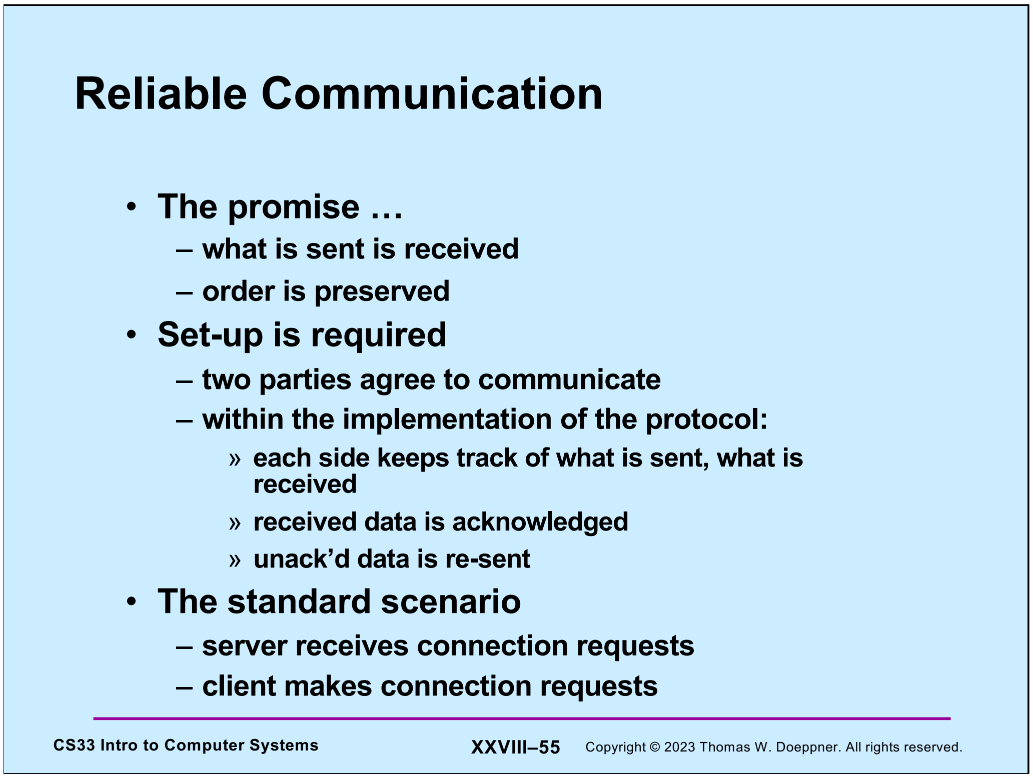
Suppose a process on one machine sends a datagram to a process on another machine. The sender uses *sendto* and the receiver uses *recvfrom*. There's a momentary problem with the network and the datagram doesn't make it to the receiving process. Its call to *recvfrom*

a) returns –1 (indicating an error)

b) returns 0

c) returns some other value

d) doesn't return



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