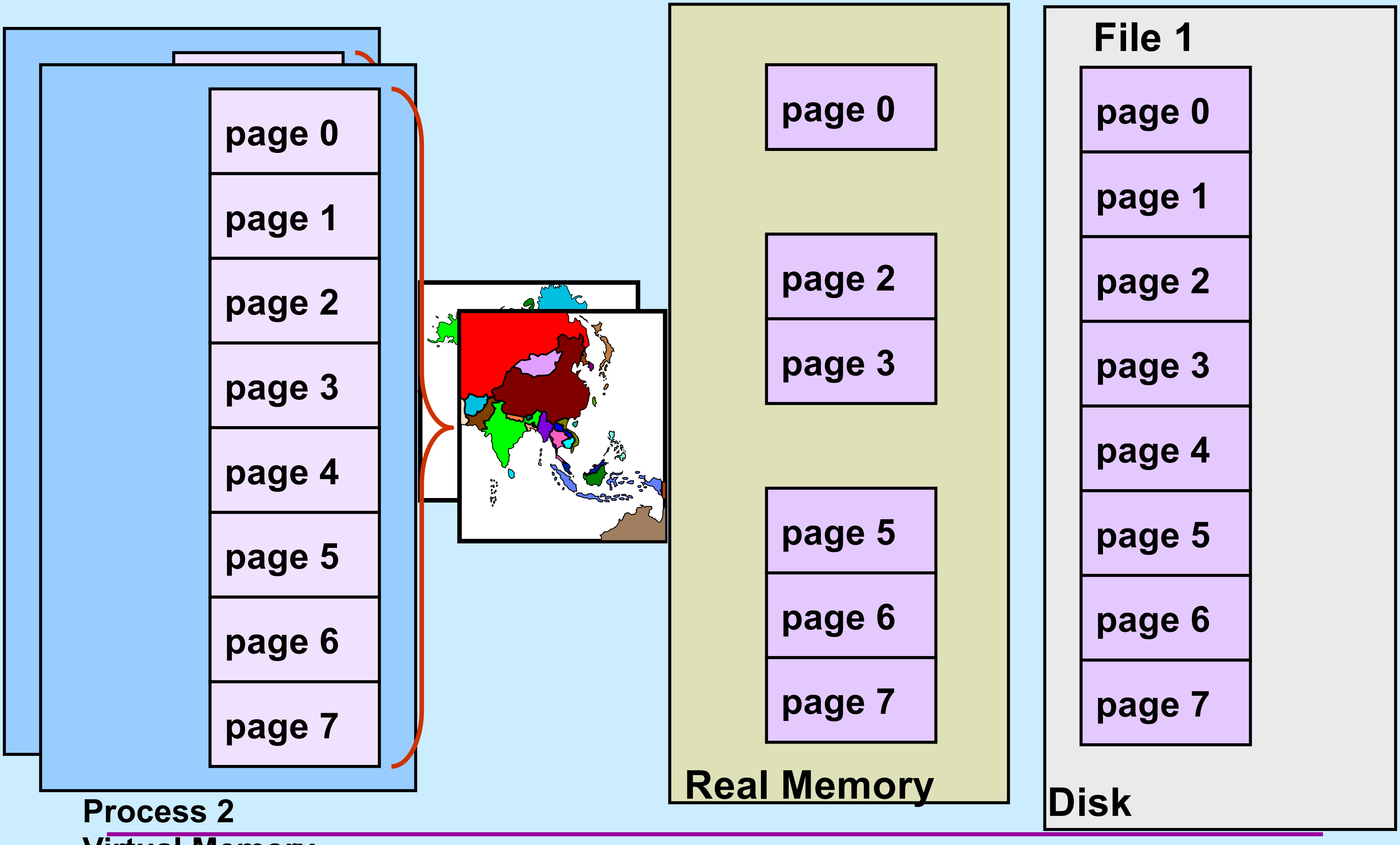
CS 33

Virtual Memory (2)

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);
}
```

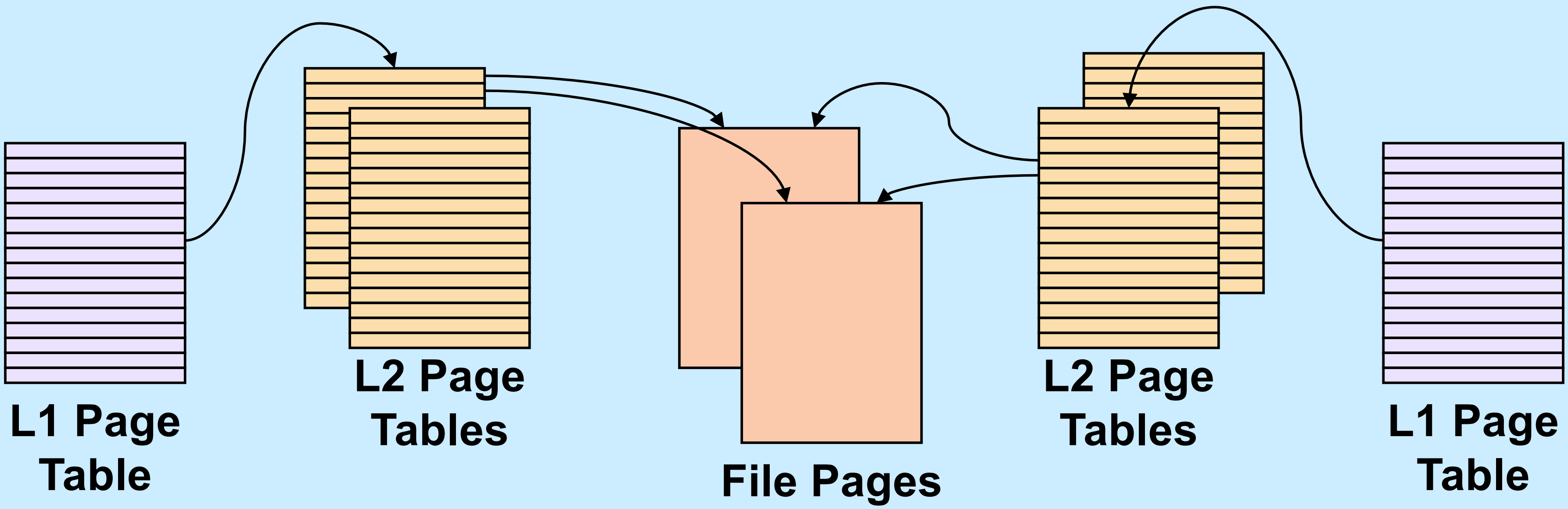
- **Mapped File I/O**

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

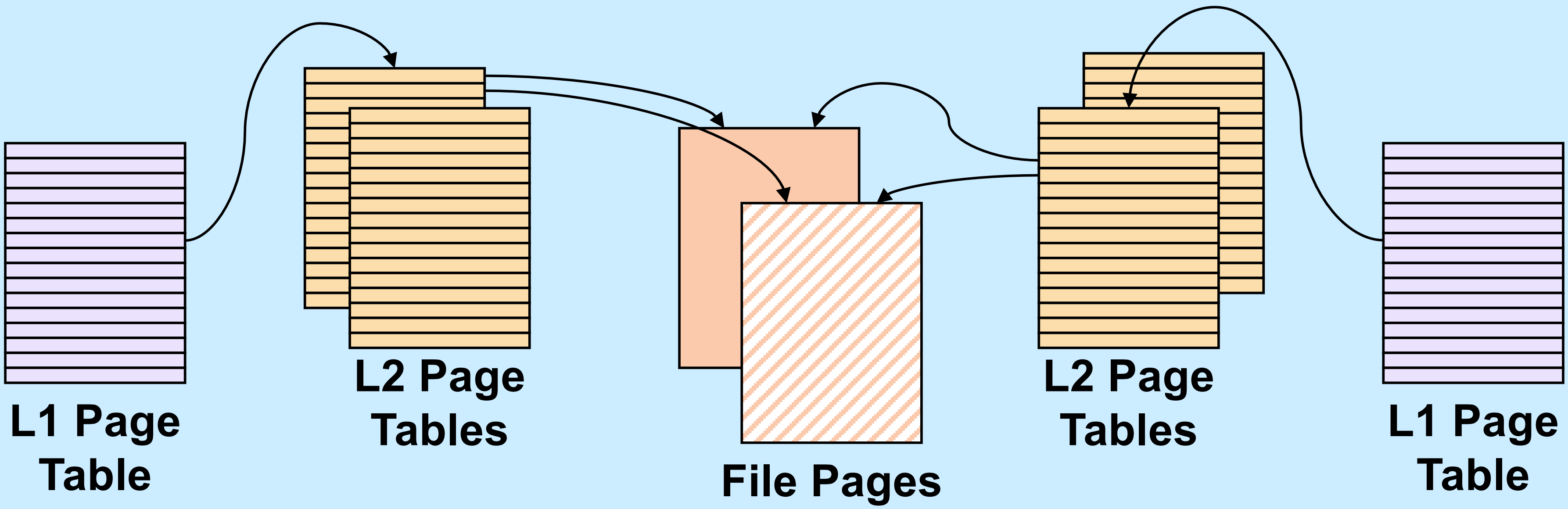
Mmap System Call

```
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
  );
```


The *mmap* System Call

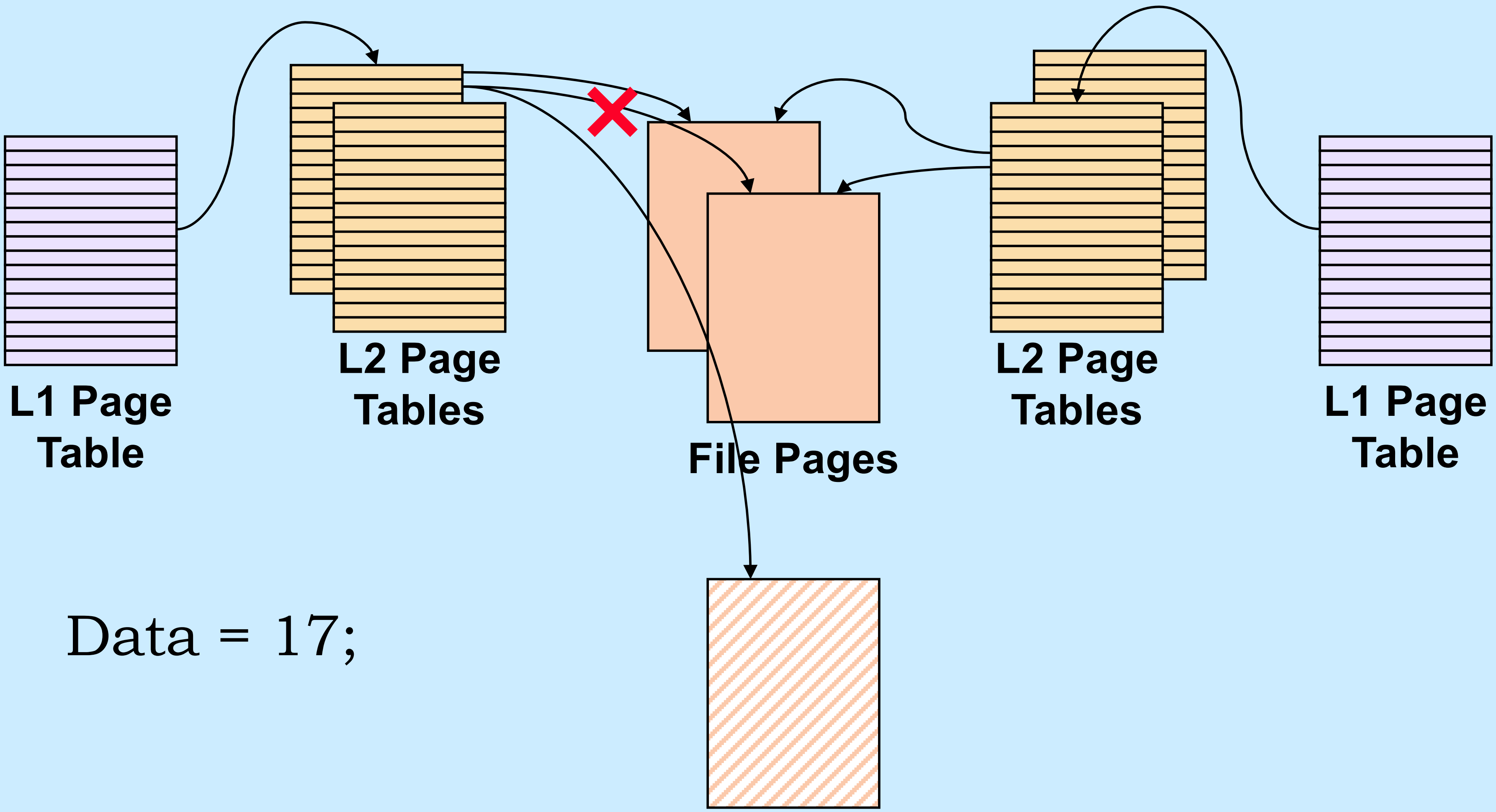


Share-Mapped Files



Data = 17;

Private-Mapped Files



Data = 17;

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

  ...

}
```


Quiz 1

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

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

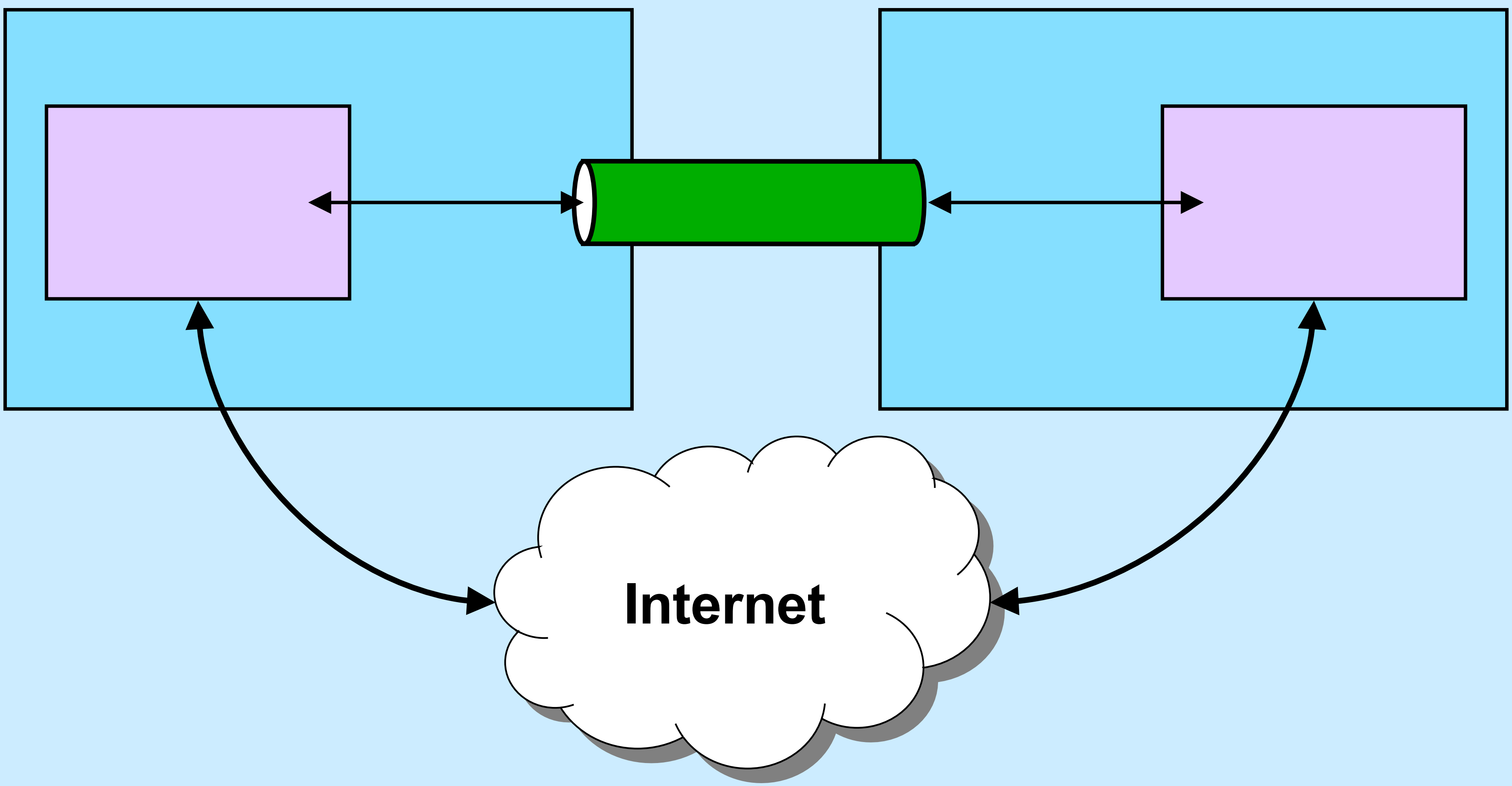
```
int main() {
  int fd = open( ... );
  int *xp = (int *)mmap(...,
      MAP_SHARED, fd, ...);
  xp[0] = 1;
  if (fork() == 0) {
    xp[0] = 2;
    exit(0);
  }
while (xp[0]==1) {
    // will loop forever?
  }
  return 0;
}
```

a) **Both loop forever**
b) **Both terminate**
c) **Left side loops forever, right side terminates**
d) **Right side loops forever, left side terminates**

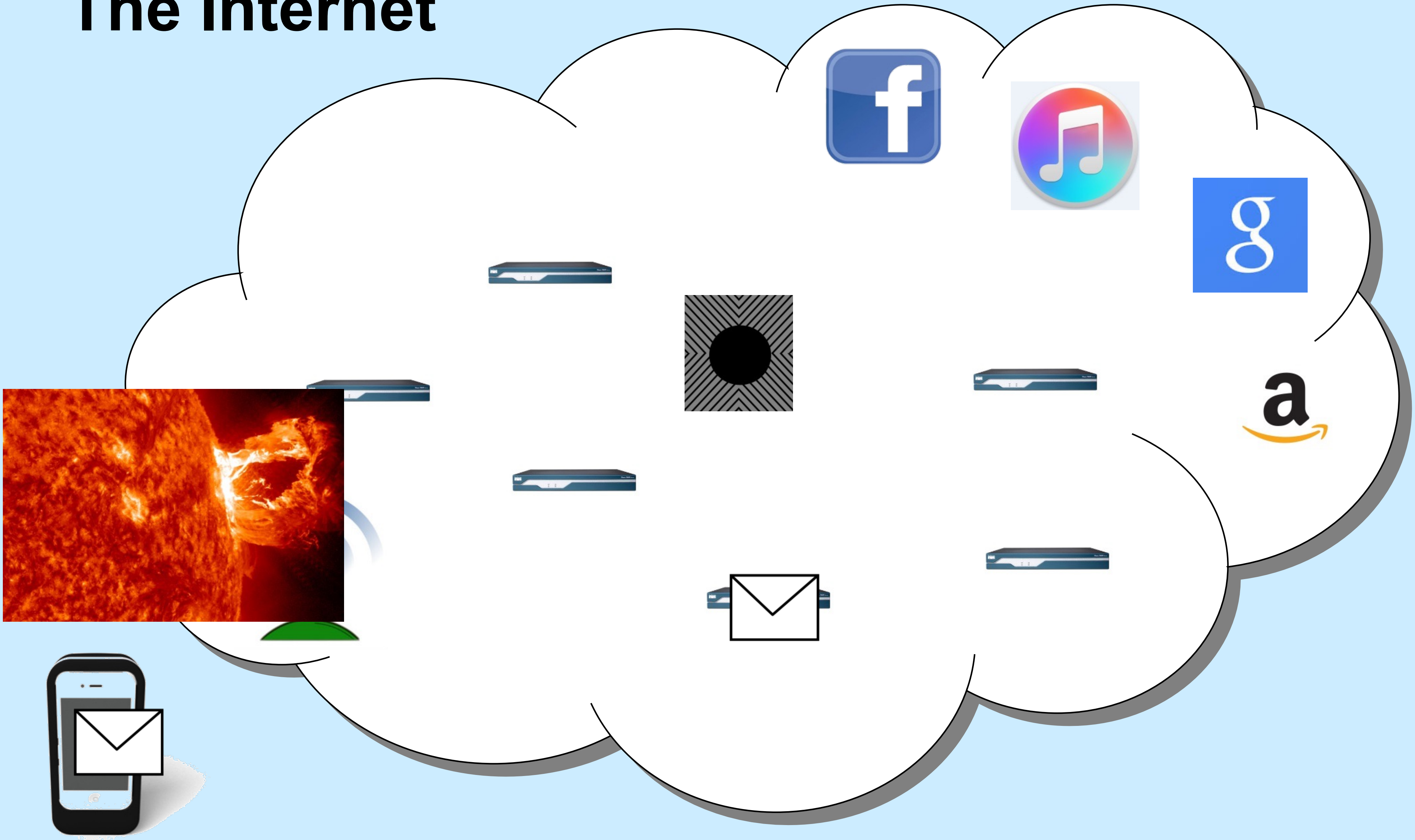
CS 33

Network Programming (1)

Communicating Over the Internet



The Internet



Names and Addresses

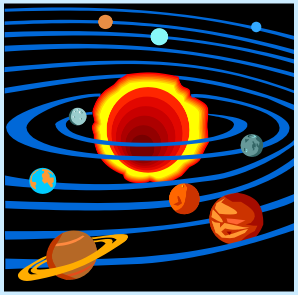
- **cslab1c.cs.brown.edu**
  - the name of a computer on the internet
  - mapped to an internet address
- **nytimes.com**
  - the name of a website
  - mapped to a number of internet addresses

- **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

  (streaming audio and 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 2

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

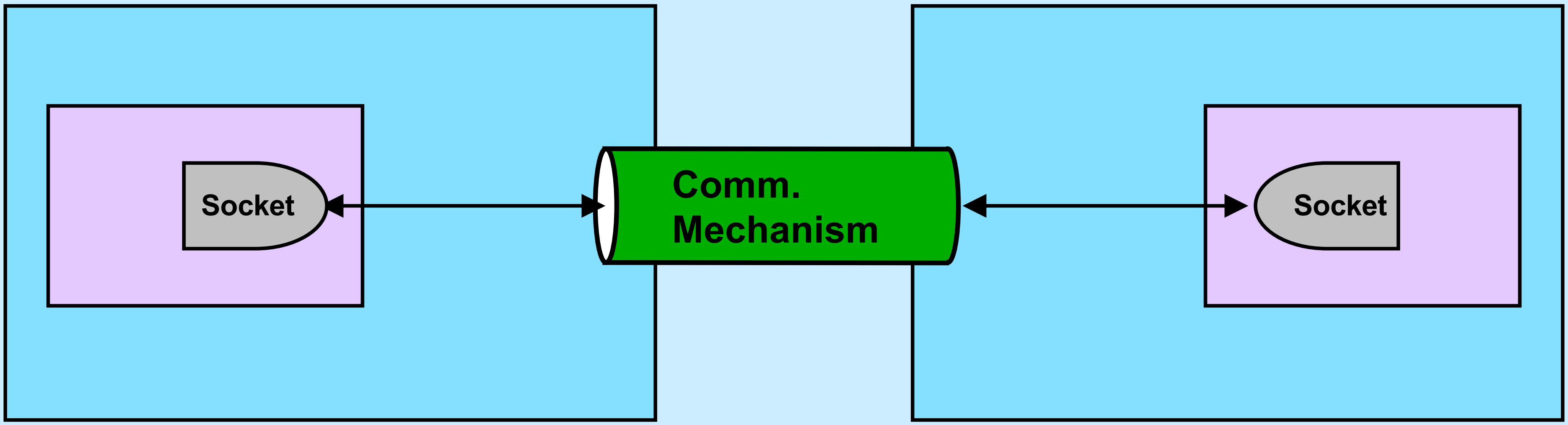
```
// sender
record_t data=getData();
write(fd, &data,
  sizeof(data));
```

```
// receiver
read(fd, &data,
  sizeof(data));
useData(data);
```

Does it work?

a) never
b) sometimes
c) always, assuming no network problems
d) always

Sockets



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

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

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)

    » port numbers <1024 are reserved for privileged servers

  – *hints* are additional information describing what you want

  – *res* is a list of *struct sockaddr* containing the results of the search

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;
```


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);
}
```


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);
}
```


UDP Server (4)

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

freeaddrinfo(result);
```

UDP Server (5)

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

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;
```


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);
```


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);
        }
    }
}
```


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);
	}
```


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);
}
```


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);
```

UDP Client (4)

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

    return 0;

}
```


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;
```


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;
}
```

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;
}
```

Quiz 3

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) doesn't return

b) returns –1 (indicating an error)

c) returns 0

d) returns some other value

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

