

CS 33

Virtual Memory (2)

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);  
}
```

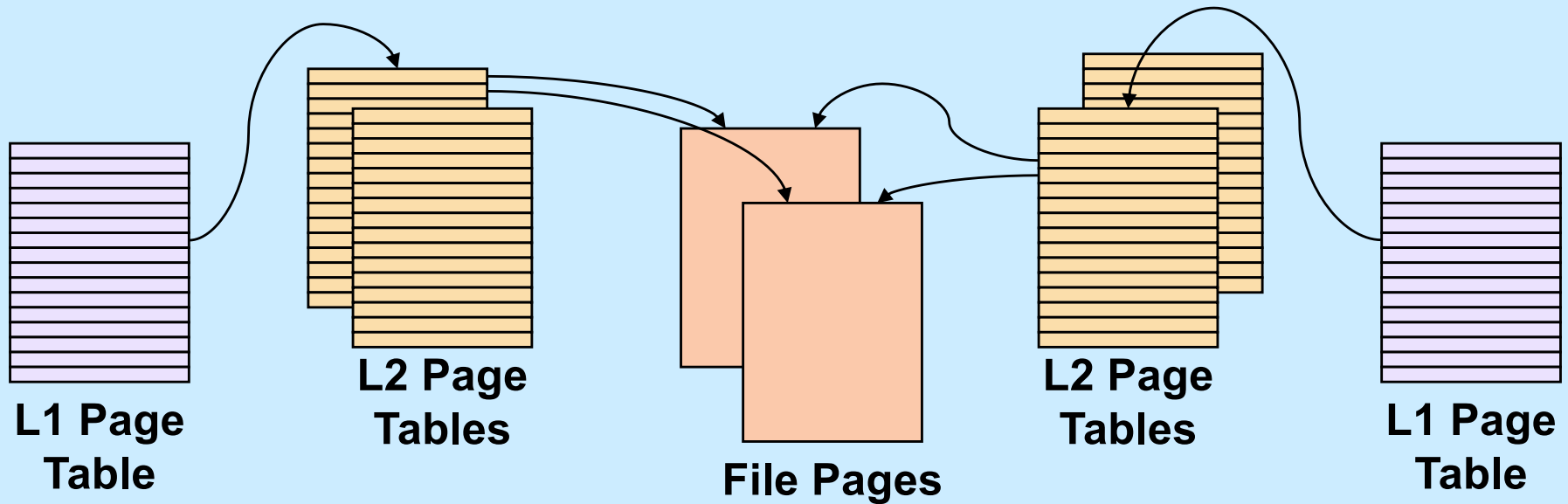
- **Mapped File I/O**

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

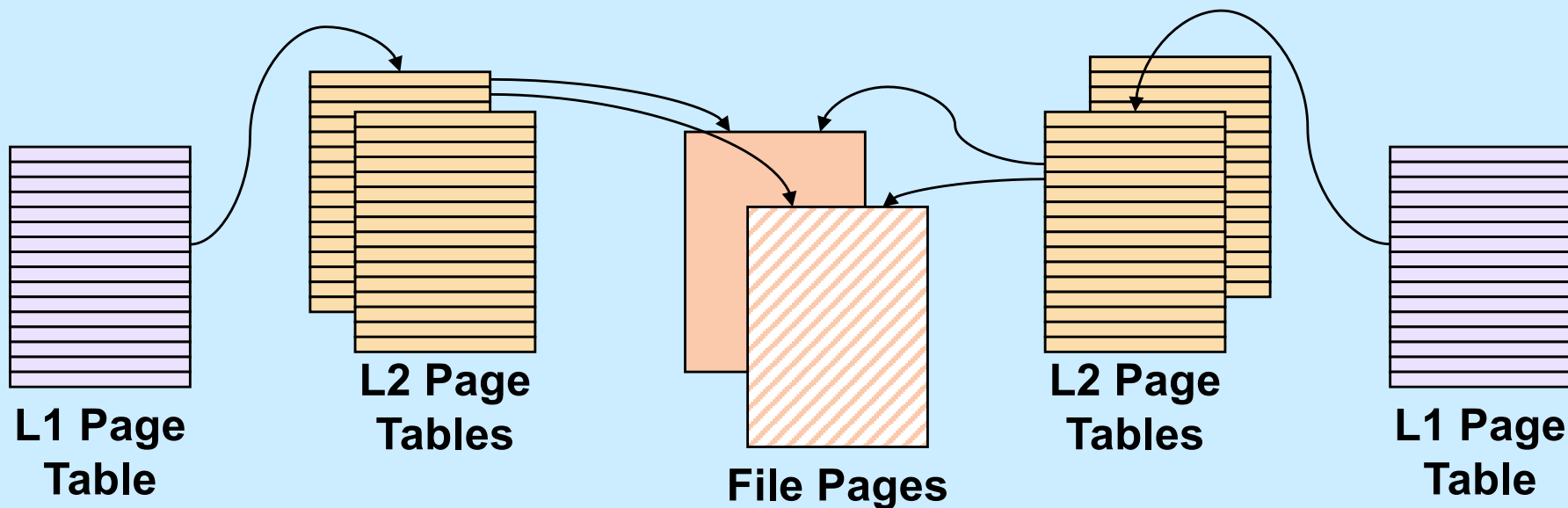
Mmap System Call

```
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

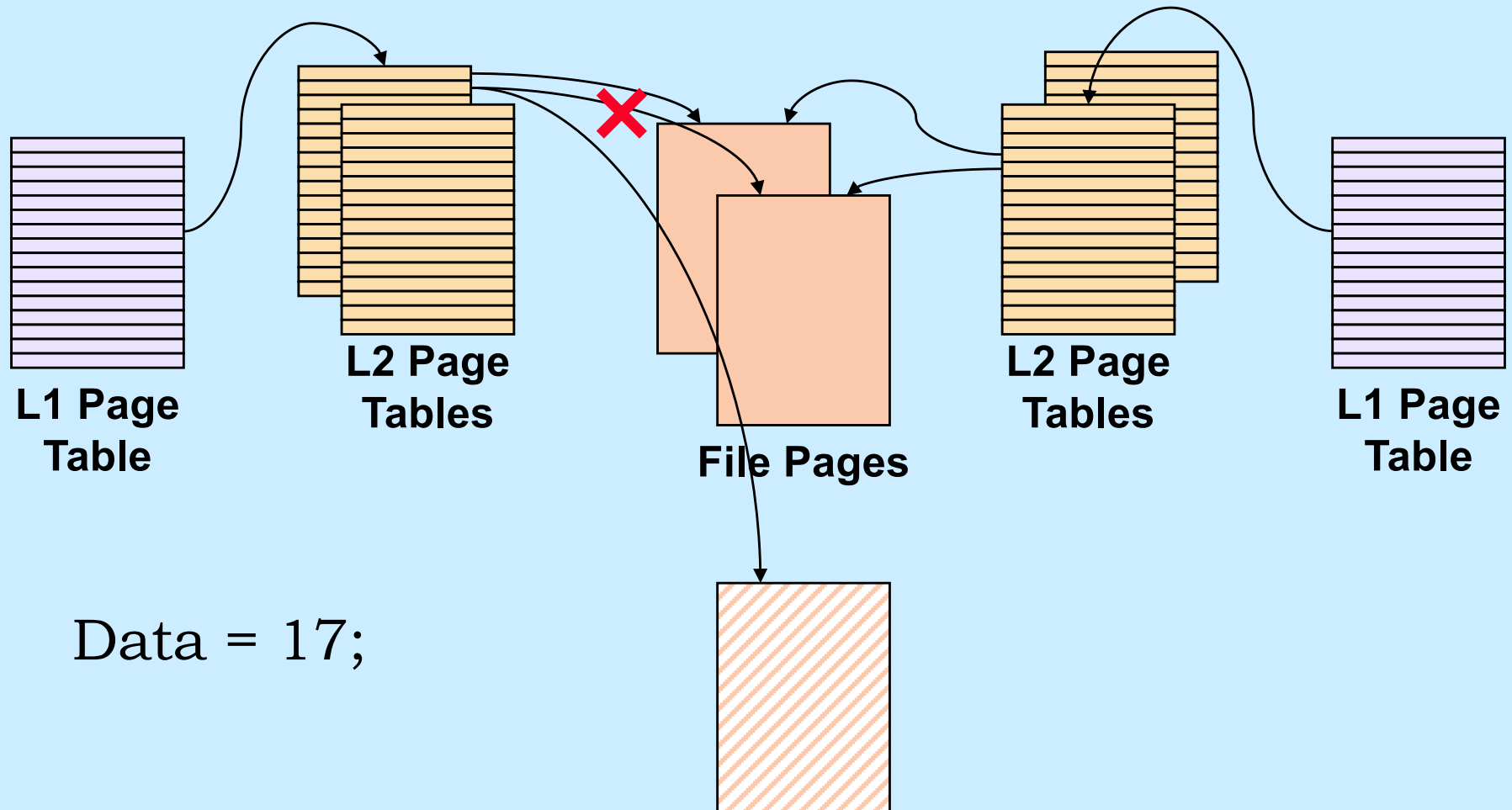


Share-Mapped Files



Data = 17;

Private-Mapped Files



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

    ...

}
```

fork and mmap

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

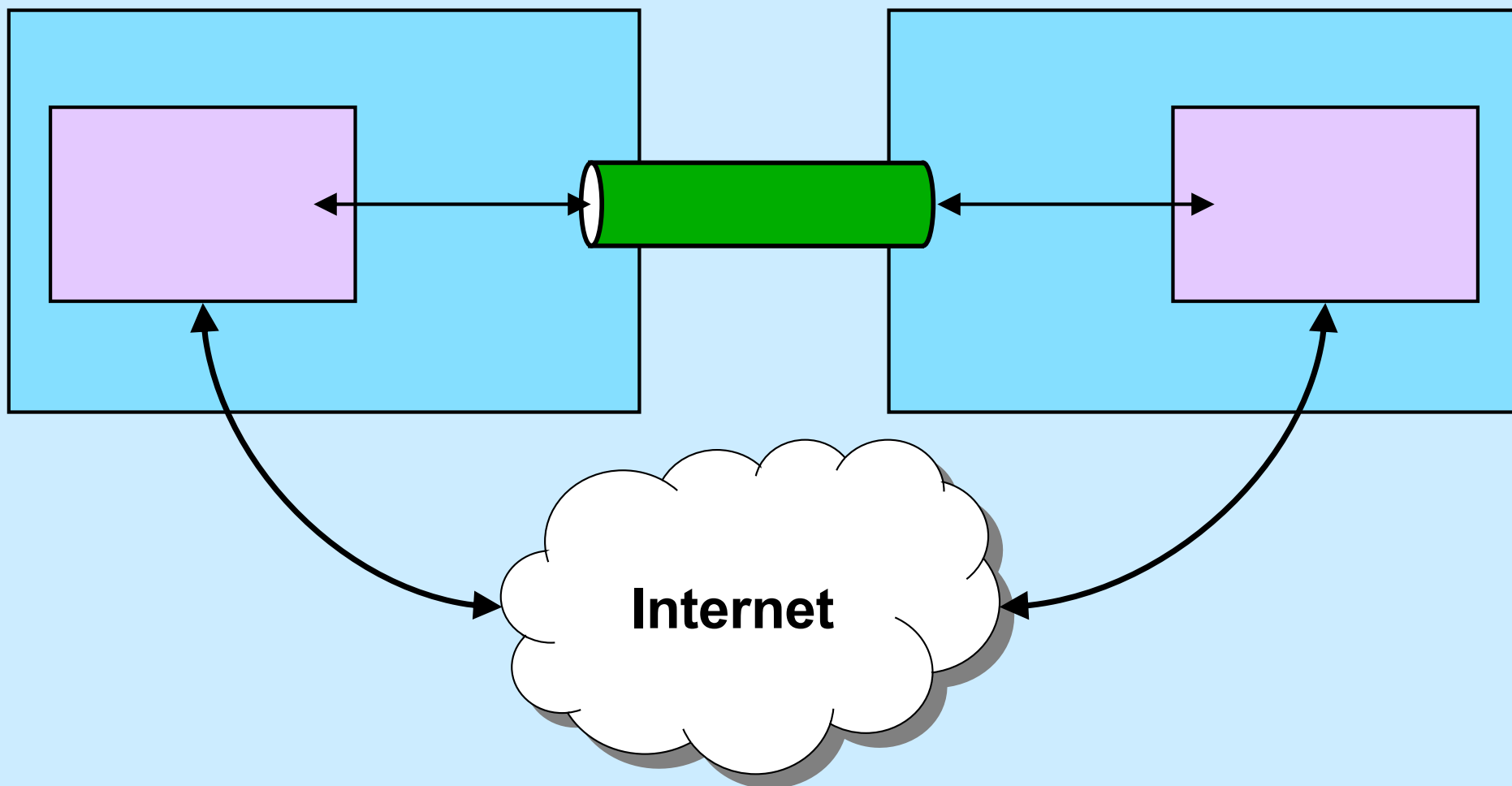
    if (fork() == 0) {
        // in child
        x = 2;
        exit(0);
    }
    // in parent
    while (x==1) {
        // will loop forever
    }
    return 0;
}
```

```
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;
}
```

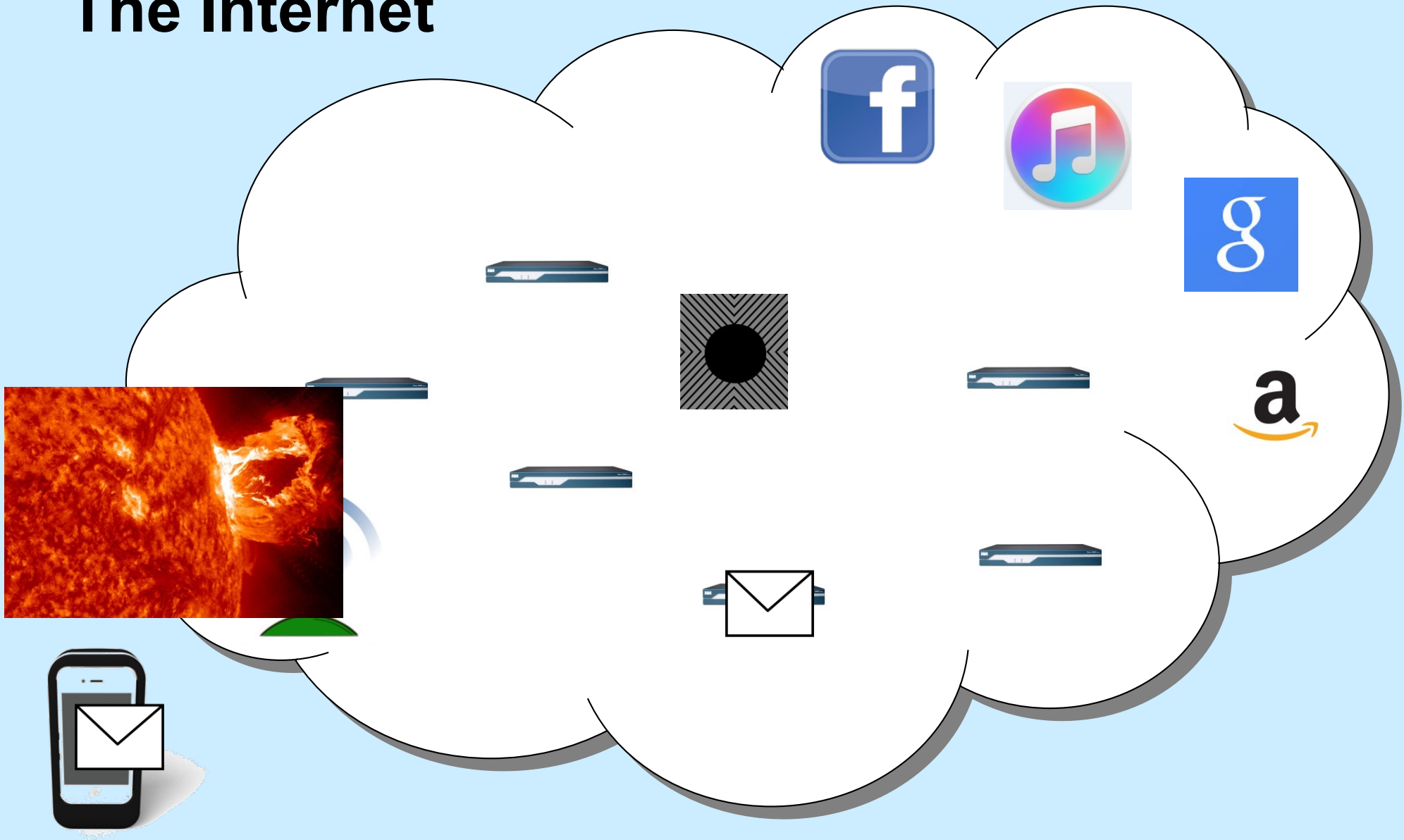

CS 33

Network Programming (1)

Communicating Over the Internet



The Internet

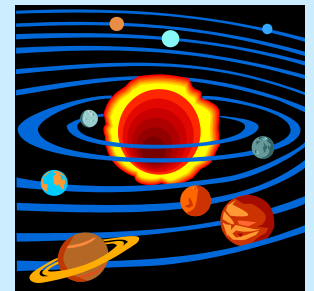


Names and Addresses

- **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
- } 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.

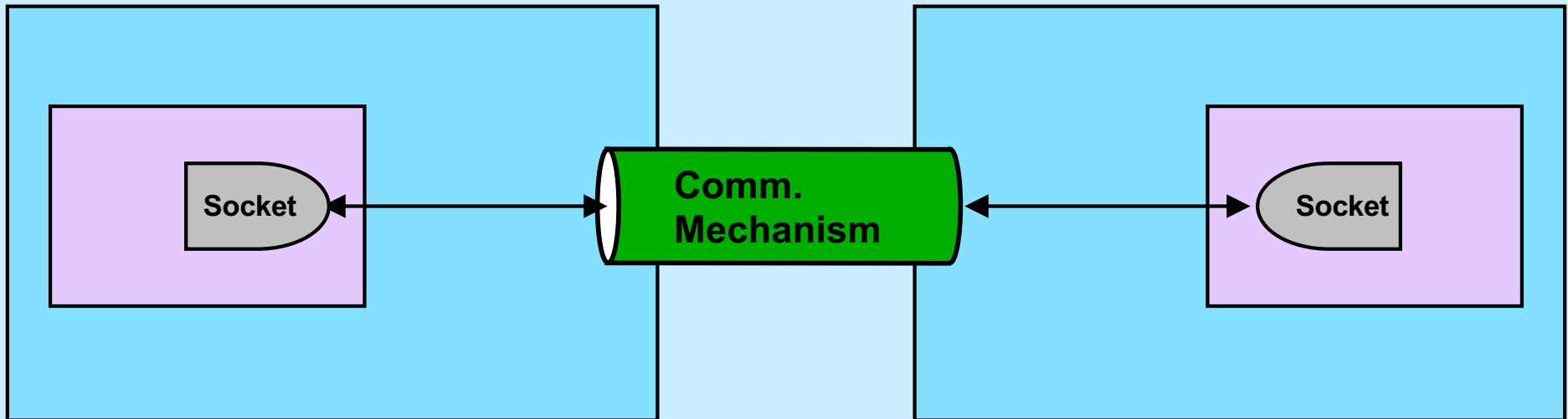
```
// sender
record_t data=getData();
write(fd, &data,
    sizeof(data));
```

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

Does it work?

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

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)
 - *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;  
  
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 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

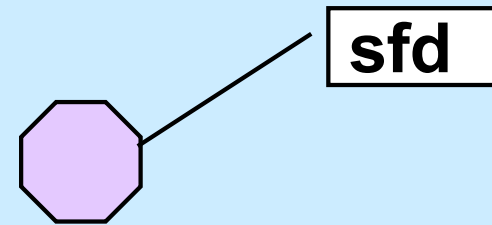
Reliable Communication

- **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

Streams in the Inet Domain (1)

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

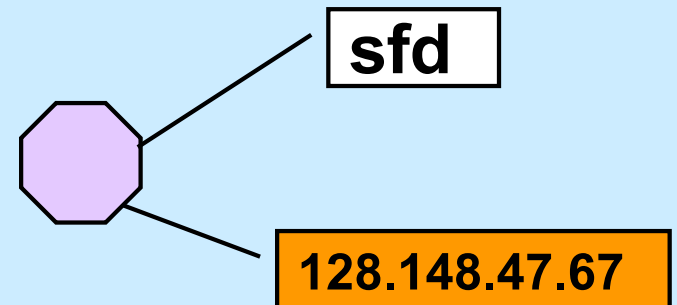
```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (2)

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

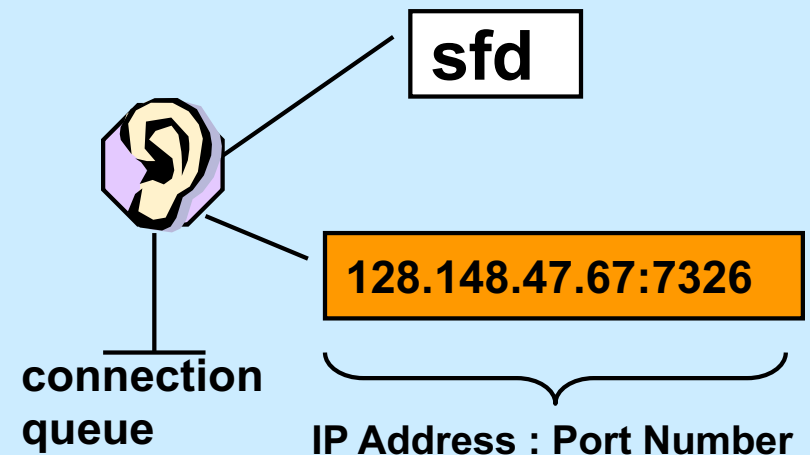
```
bind(sfd,  
    (struct sockaddr *) &my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (3)

- **Server steps**
 - 3) put socket in “listening mode”

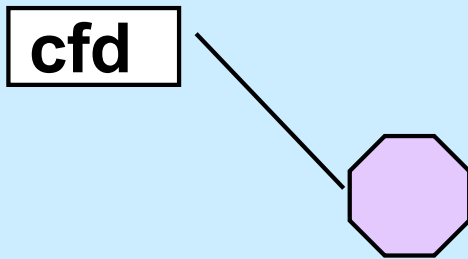
```
int listen(int sfd, int MaxQueueLength);
```



Streams in the Inet Domain (4)

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

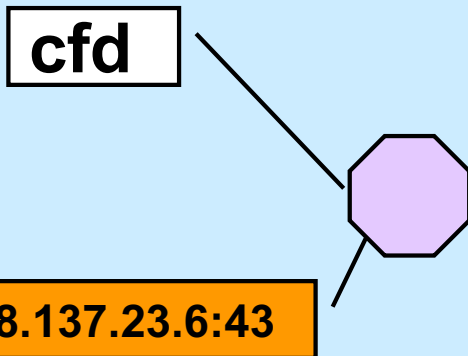
```
cfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (5)

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

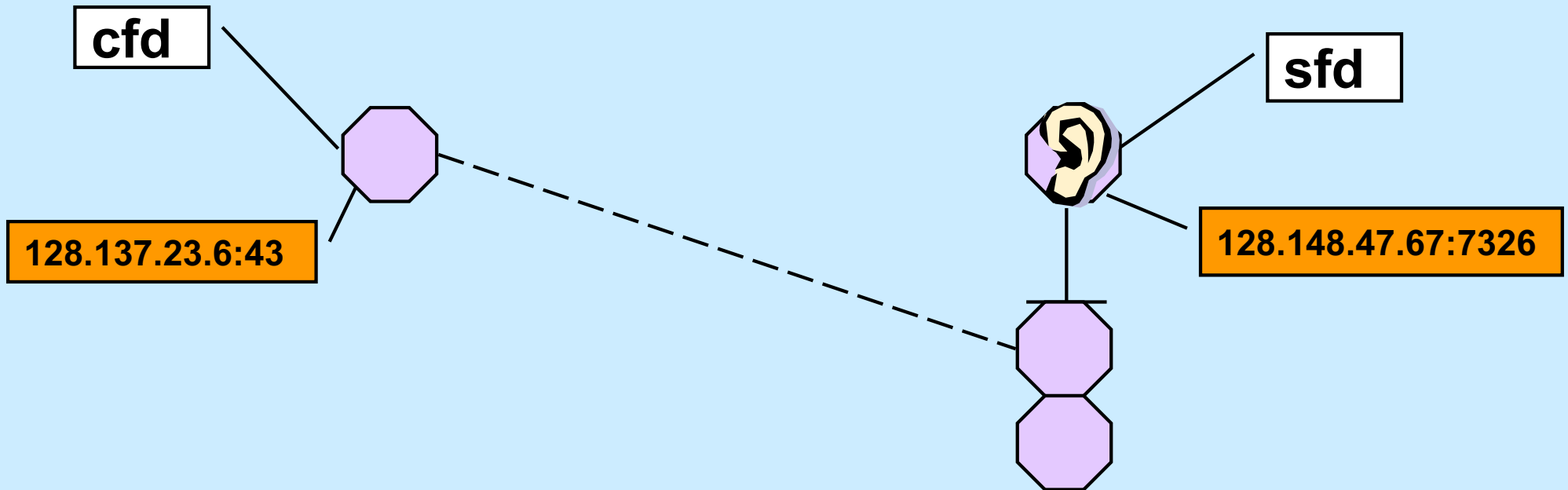
```
bind(cfd,  
    (struct sockaddr *) &my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (6)

- Client steps
 - 3) connect to server

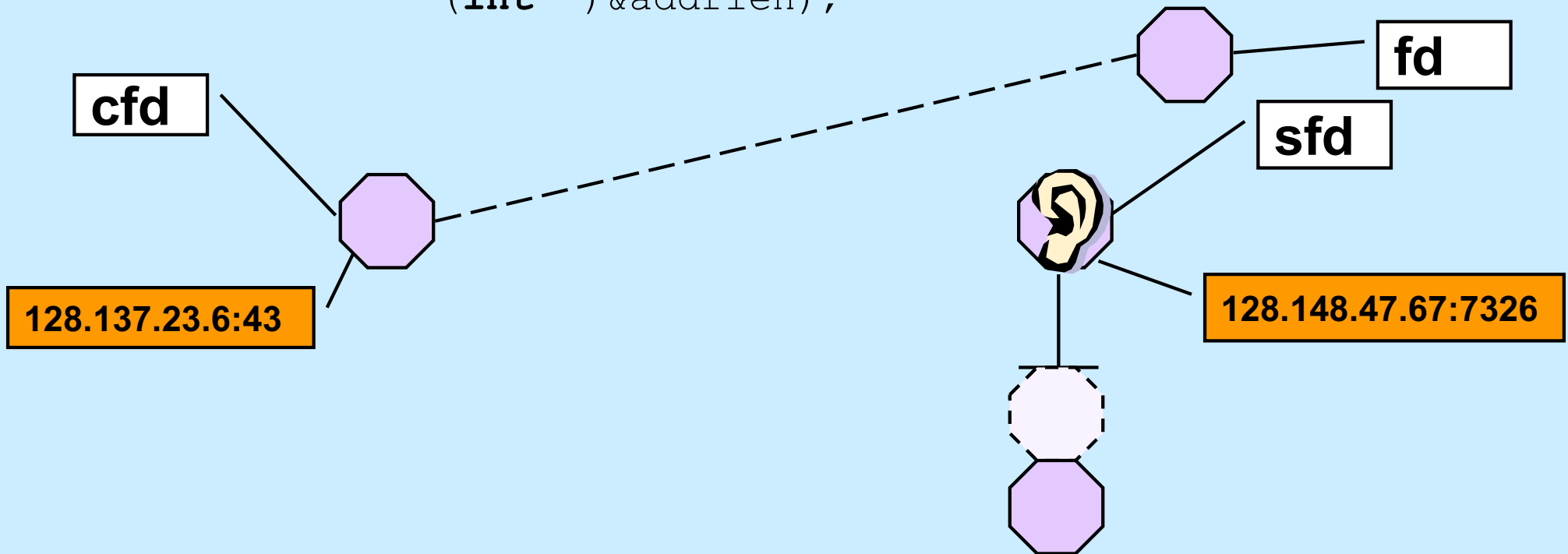
```
connect(cfd, (struct sockaddr *)&server_addr,  
        sizeof(server_addr));
```



Streams in the Inet Domain (7)

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

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



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;
```

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);
}
```

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);
}
```

TCP Server (4)

```
if (r == NULL) {  
    fprintf(stderr, "Could not find local interface %s\n");  
    exit(1);  
}  
freeaddrinfo(result);  
  
if (listen(lsocket, 5) < 0) {  
    perror("listen");  
    exit(1);  
}
```

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);  
    }  
}
```

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);
```

TCP Server (7)

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

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");  
}
```


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);
    }
```

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);  
}
```

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);  
}
```

TCP Client (4)

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

TCP Client (5)

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

Quiz 3

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.**

Quiz 4

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.**