CS 33
More Network Programming
Stream Relay

- Source
- Sink
- Relay
- Pipe

Source → Relay → Sink
Source → Relay → Source
Sink → Relay → Sink
Sink → Relay → Source
Solution?

```c
while(...) {
    size = read(left, buf, sizeof(buf));
    write(right, buf, size);
    size = read(right, buf, sizeof(buf));
    write(left, buf, size);
}
```
Select System Call

```c
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
);
```
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, BSIZE);
        if (FD_ISSET(right, &rd))
            read(right, bufRL, BSIZE);
        if (FD_ISSET(right, &wr))
            write(right, bufLR, BSIZE);
        if (FD_ISSET(left, &rd))
            write(left, bufRL, BSIZE);
    }
}
40 bytes have been read from the left-hand source. Select reports that it is ok to write to the right-hand sink.

a) You’re guaranteed you can immediately write all 40 bytes to the right-hand sink

b) All that’s guaranteed is that you can immediately write at least one byte to the right-hand sink

c) Nothing is guaranteed
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;
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);
Relay (3)

```c
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;
}
```
Relay (4)

```c
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;
```
A Really Simple Protocol

• Transfer a file
  – layered on top of TCP
    » reliable
    » indicates if connection is closed

• To send a file
  P<null-terminated pathname><contents of file>

• To retrieve a file
  G<null-terminated pathname><contents of file>
Server State Machine

- **GOOD IN**: recv: P<pathname> -> send: G
- **GOOD OUT**: recv: G<pathname> -> send: G
- **BAD**: recv: !(P||G) || invalid pathname -> send: B
- **IN**: recv: data
- **OUT**: send: data
- **EOF**: recv: data
- **Done**: send: data

RDY transitions:
- **GOOD IN**: recv: P<pathname> -> send: G
- **GOOD OUT**: recv: G<pathname> -> send: G
- **BAD**: recv: !(P||G) || invalid pathname -> send: B

Copyright © 2017 Thomas W. Doeppner. All rights reserved.
Keeping Track of State

typedef struct client {
    int fd;       // file descriptor of local file being transferred
    int size;     // size of out-going data in buffer
    char buf[BSIZE];
    enum state {RDY, BAD, GOOD, TRANSFER} state;
    /*
     * states:
     *   RDY: ready to receive client's command (P or G)
     *   BAD: client's command was bad, sending B response + error msg
     *   GOOD: client's command was good, sending G response
     *   TRANSFER: transferring data
     */
    enum dir {IN, OUT} dir;
    /*
     * IN: client has issued P command
     * OUT: client has issued G command
     */
} client_t;
Keeping Track of Clients

```c
client_t clients[MAX_CLIENTS];
for (i=0; i < MAX_CLIENTS; i++)
    clients[i].fd = -1; // illegal value
```
Main Server Loop

```c
while(1) {
    select(maxfd, &trd, &twr, 0, 0);
    if (FD_ISSET(lsock, &trd)) {
        // a new connection
        new_client(lsock);
    }
    for (i=lsock+1; i<maxfd; i++) {
        if (FD_ISSET(i, &trd)) {
            // ready to read
            read_event(i);
        }
        if (FD_ISSET(i, &twr)) {
            // ready to write
            write_event(i);
        }
    }
    trd = rd; twr = wr;
}
```
New Client

// Accept a new connection on listening socket // fd. Return the connected file descriptor

```c
int new_client(int fd) {
    int cfd = accept(fd, 0, 0);
    clients[cfd].state = RDY;
    FD_SET(cfd, &rd);
    return cfd;
}
```
Read Event (1)

// File descriptor fd is ready to be read. Read it, then handle
// the input

void read_event(int fd) {
    client_t *c = &clients[fd];
    int ret = read(fd, c->buf, BSIZE);
    switch (c->state) {
    case RDY:
        if (c->buf[0] == 'G') {
            // GET request (to fetch a file)
            c->dir = OUT;
            if ((c->fd = open(&c->buf[1], O_RDONLY)) == -1) {
                // open failed; send negative response and error message
                c->state = BAD;
                c->buf[0] = 'B';
                strncpy(&c->buf[1], strerror(errno), BSIZE-2);
                c->buf[BSIZE-1] = 0;
                c->size = strlen(c->buf)+1;
            }
        }
    }
else {
    // open succeeded; send positive response
    c->state = GOOD;
    c->size = 1;
    c->buf[0] = 'G';
}
// prepare to send response to client
FD_SET(fd, &wr);
FD_CLR(fd, &rd);
break;
}
if (c->buf[0] == 'P') {
    // PUT request (to create a file)
    c->dir = IN;
    if ((c->fd = open(&c->buf[1],
            O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
        // open failed; send negative response and error message
        ...
    } else {
        // open succeeded; send positive response
        ...
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
}
Read Event (4)

case TRANSFER:
    // should be in midst of receiving file contents from client
    if (ret == 0) {
        // eof: all done
        close(c->fd);
        close(fd);
        FD_CLR(fd, &rd);
        break;
    }
    if (write(c->fd, c->buf, ret) == -1) {
        // write to file failed: terminate connection to client
        ...
        break;
    }
    // continue to read more data from client
    break;
}
Write Event (1)

// File descriptor fd is ready to be written to. Write to it, then, // depending on current state, prepare for the next action.

void write_event(int fd) {
    client_t *c = &clients[fd];
    int ret = write(fd, c->buf, c->size);
    if (ret == -1) {
        // couldn't write to client; terminate connection
        close(c->fd);
        close(fd);
        FD_CLR(fd, &wr);
        c->fd = -1;
        perror("write to client");
        return;
    }
    switch (c->state) {

Write Event (2)

case BAD:
    // finished sending error message; now terminate client connection
    close(c->fd);
    close(fd);
    FD_CLR(fd, &wr);
    c->fd = -1;
    break;
Write Event (3)

case GOOD:
    c->state = TRANSFER;
    if (c->dir == IN) {
        // finished response to PUT request
        FD_SET(fd, &rd);
        FD_CLR(fd, &wr);
        break;
    }
    // otherwise finished response to GET request, so proceed
Write Event (4)

case TRANSFER:
    // should be in midst of transferring file contents to client
    if ((c->size = read(c->fd, c->buf, BSIZE)) == -1) {
        ...
        break;
    } else if (c->size == 0) {
        // no more file to transfer; terminate client connection
        close(c->fd);
        close(fd);
        close(fd);
        FD_CLR(fd, &wr);
        c->fd = -1;
        break;
    }
    // continue to write more data to client
    break;
}
Problems

• Works fine as long as the protocol is followed correctly
  – can client (malicious or incompetent) cause server to misbehave?
• How can the server limit the number of clients?
• How does server limit file access?