Computer Networks

EDA387/DIT663

Socket API

Part 4

Today

Preview

- One can show that when the echo client handles two inputs at the same time, i.e., the standard input and a TCP socket.
- The program encounters a problem when the client is blocked in a call to `fgets` (on standard input) and the server process is killed.
  - The server TCP correctly sends a FIN to the client TCP, but since the client process is blocked reading from standard input, it never sees the EOF until it read from the socket (possibly much later)

Today

- We explain how can the process tell the kernel that we want to be notified if one or more I/O conditions are ready
  - input is ready to be read, or the descriptor is capable of taking more output
- This capability is called I/O multiplexing and is provided by the `select`

I/O Multiplexing

- Typically used in networking applications:
  - When a client is handling multiple descriptors, I/O multiplexing should be used
  - Client handling multiple sockets at the same time
  - If a TCP server handles both a listening socket and its connected sockets, I/O multiplexing is normally used
  - If a server handles both TCP and UDP
  - If a server handles multiple services and perhaps multiple protocols

I/O multiplexing is not limited to network programming

I/O Models

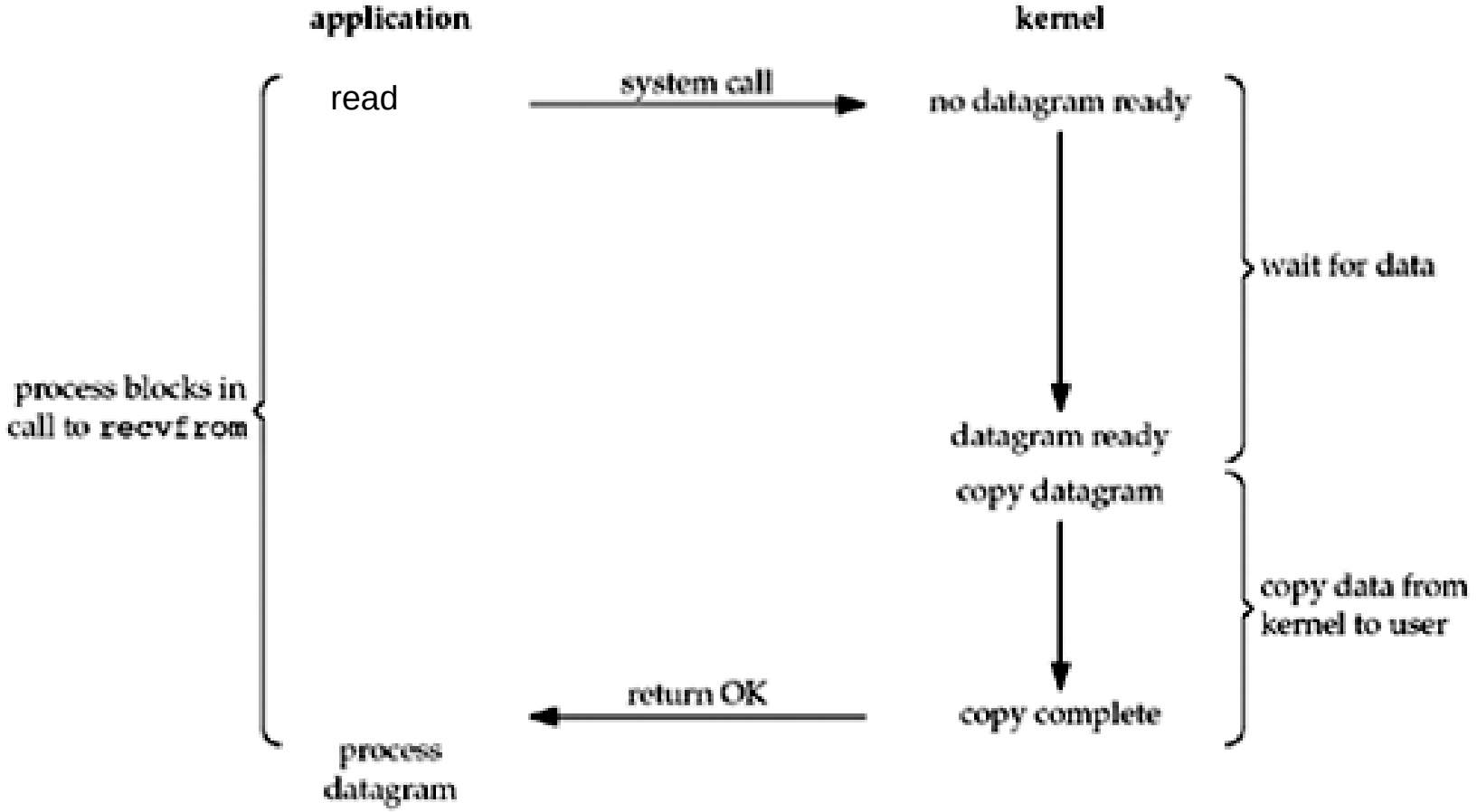
Outline

- blocking I/O
- nonblocking I/O
- I/O multiplexing (select and poll)
- ~~signal driven I/O~~ (not in this course)
- ~~asynchronous I/O~~ (not in this course)

Blocking I/O Model

Simplified

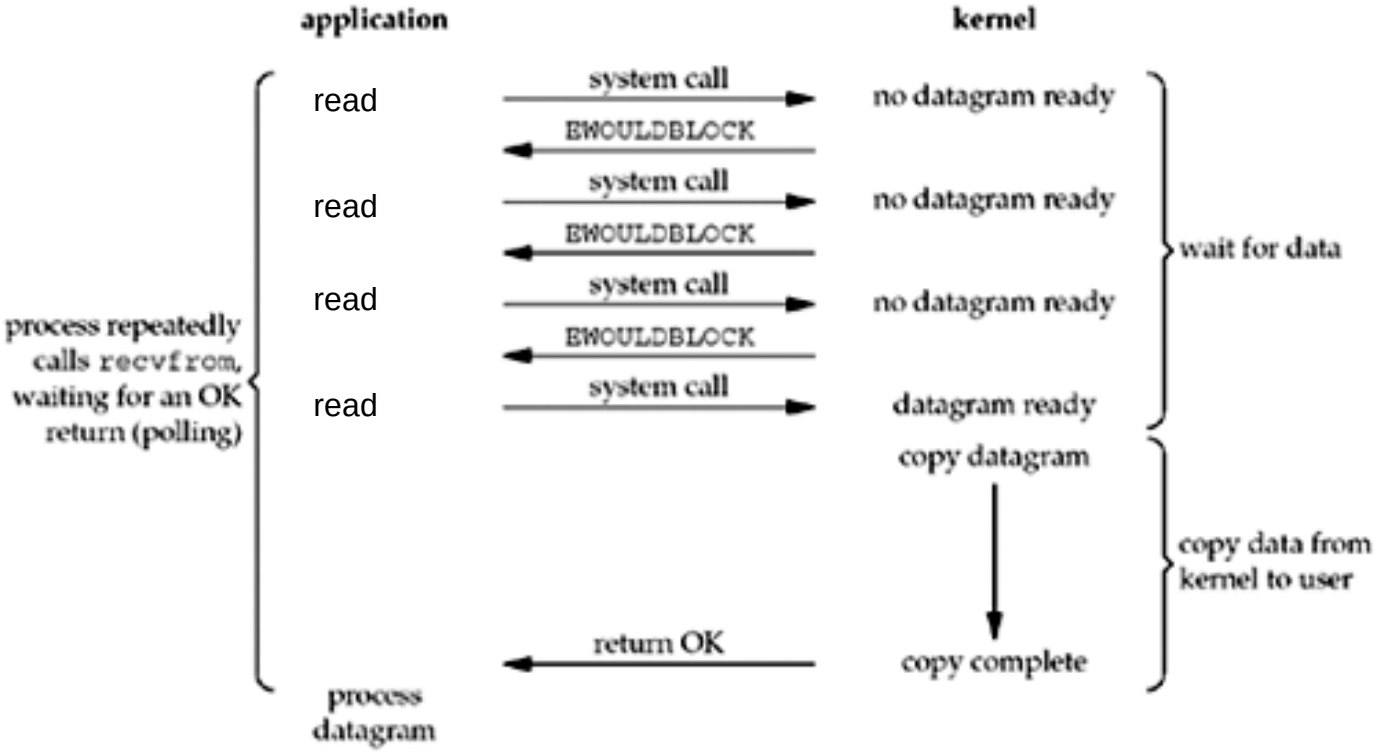
- By default, all sockets are blocking



Nonblocking I/O Model

Simplified

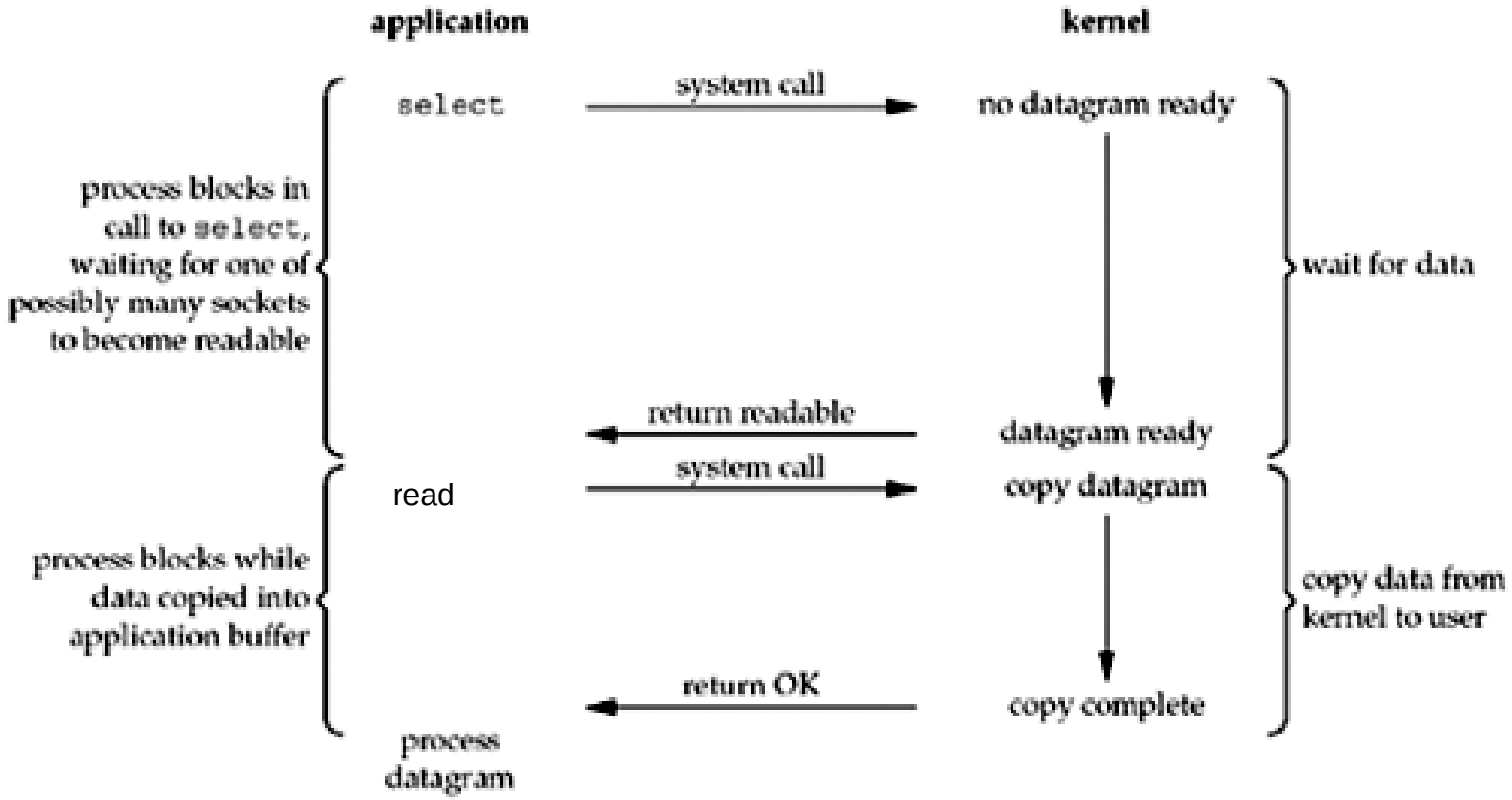
- Can set a socket to be nonblocking
  - telling the kernel "when an I/O operation that I request cannot be completed without putting the process to sleep, do not put the process to sleep, but return an error instead"



I/O Multiplexing Model

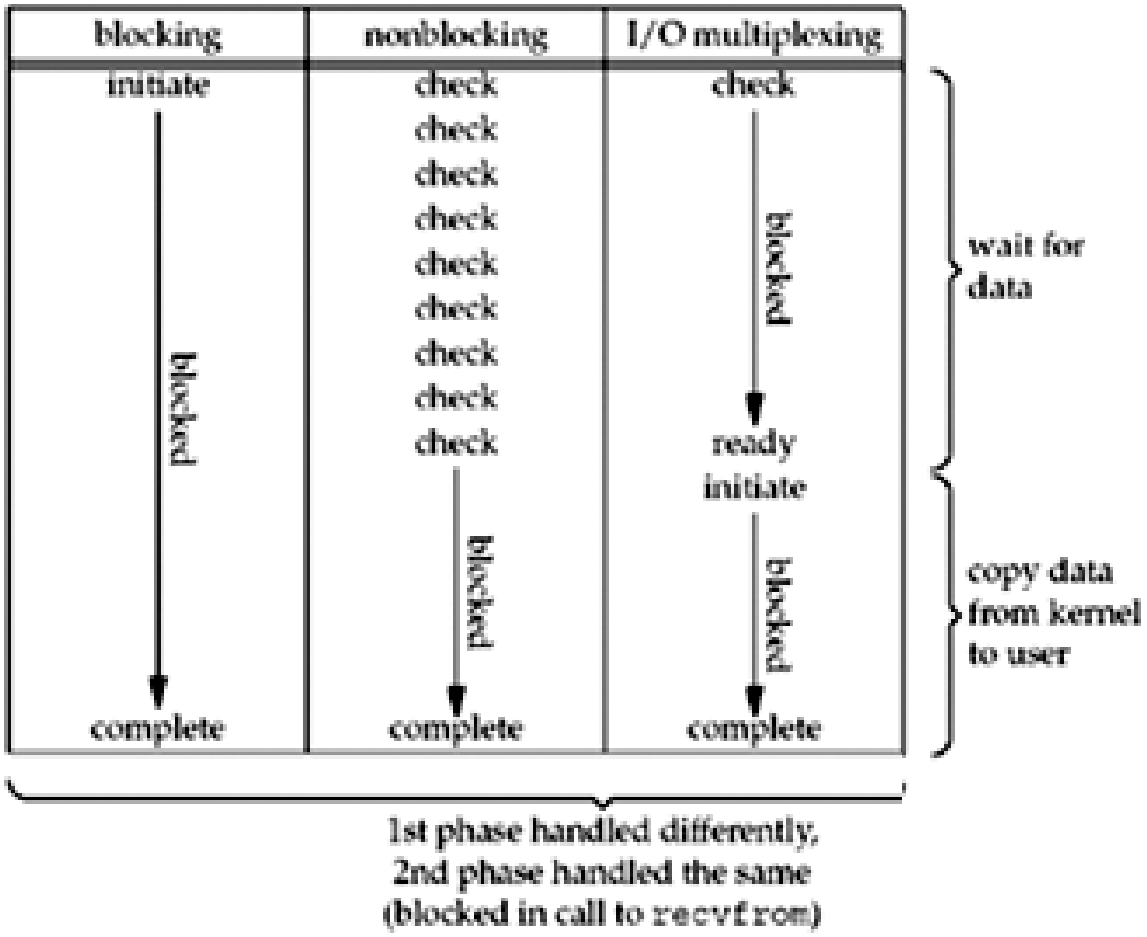
Simplified

- Call select and block, instead of blocking in the actual I/O system call
  - When returned, one of the file descriptors is ready



Comparison of the I/O Models

Simplified



`select` Function

- TCP client establishes a connection with its server

```
int select(int maxfdp1, fd_set *readset,
  fd_set *writeset, fd_set *exceptset, const
  struct timeval *timeout);
```

- E.g., call select and tell the kernel to return only when:
  - Any of the descriptors in the set {1, 4, 5} are ready for reading
  - Any of the descriptors in the set {2, 7} are ready for writing
  - Any of the descriptors in the set {1, 4} have an exception condition pending
  - 10.2 seconds have elapsed

`select` Function

- Tells the kernel which descriptors to test (reading, writing, or an exception condition) and how long to wait
  - not restricted to sockets; any file descriptor can be tested

```
struct timeval  {
   long    tv_sec;//seconds
   long    tv_usec;//microseconds
};
```

FD_* Functions

`void FD_ZERO(fd_set *fdset);` clear all bits in `fdset`

`void FD_SET(int fd,fd_set *fdset);` turn on the bit for `fd` in `fdset`

`void FD_CLR(int fd,fd_set *fdset);` turn off the bit for `fd` in `fdset`

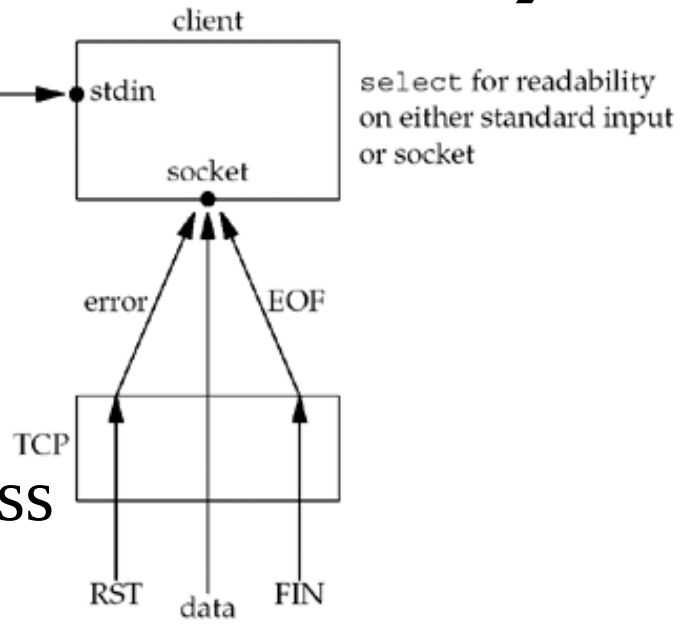
`int FD_ISSET(int fd,fd_set *fdset);` is the bit for `fd` on in `fdset`?

```
fd_set rset;
FD_ZERO(&rset);  initialize the set: all bits off
FD_SET(1, &rset);  turn on bit for fd 1
FD_SET(4, &rset);  turn on bit for fd 4
FD_SET(5, &rset);  turn on bit for fd 5
```

- `select` modifies the descriptor sets pointed to by these arguments

`str_cli` Function (Revisited)

- Peer TCP sends data, the socket becomes readable and read returns greater than 0
  - number of bytes of data
- Peer TCP sends a FIN (the peer process terminates), the socket becomes readable and read returns 0 (EOF)
- Peer TCP sends an RST (the peer host has crashed and rebooted), the socket becomes readable, read returns -1, and `errno` contains the specific error code



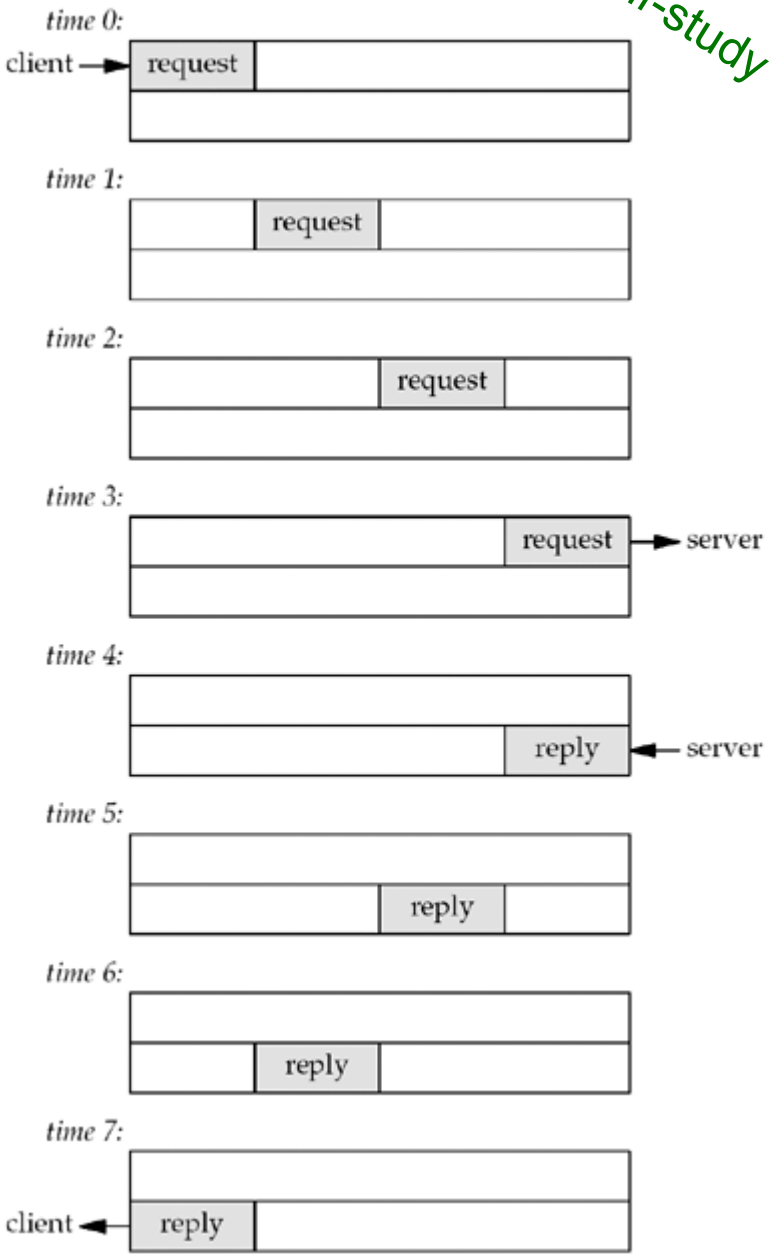
str_cli Function (Revisited)

```
void str_cli(FILE *fp, int sockfd){
int maxfdp1; fd_set rset; char sendline[MAXLINE], recvline[MAXLINE];
 FD_ZERO(&rset);
 for ( ; ; )  {
  FD_SET(fileno(fp), &rset);  FD_SET(sockfd, &rset);
  maxfdp1 = max(fileno(fp), sockfd)  +  1;
  Select(maxfdp1,  &rset,  NULL,  NULL,  NULL);
  if (FD_ISSET(sockfd,  &rset))  {//socket is readable
   if (Readline(sockfd, recvline, MAXLINE) == 0) err_quit("str_cli: server
   terminated prematurely");
   Fputs(recvline, stdout);
  }
  if (FD_ISSET(fileno(fp), &rset))  {//input is readable
   if (Fgets(sendline, MAXLINE, fp) == NULL) return;//all done
   Writen(sockfd, sendline, strlen(sendline));
}}}
```

Batch Input and Buffering

- Since there is a delay between sending a packet and that packet arriving at the other end of the pipe, and since the pipe is full-duplex, the iterative server uses only a small fraction of the pipe's capacity

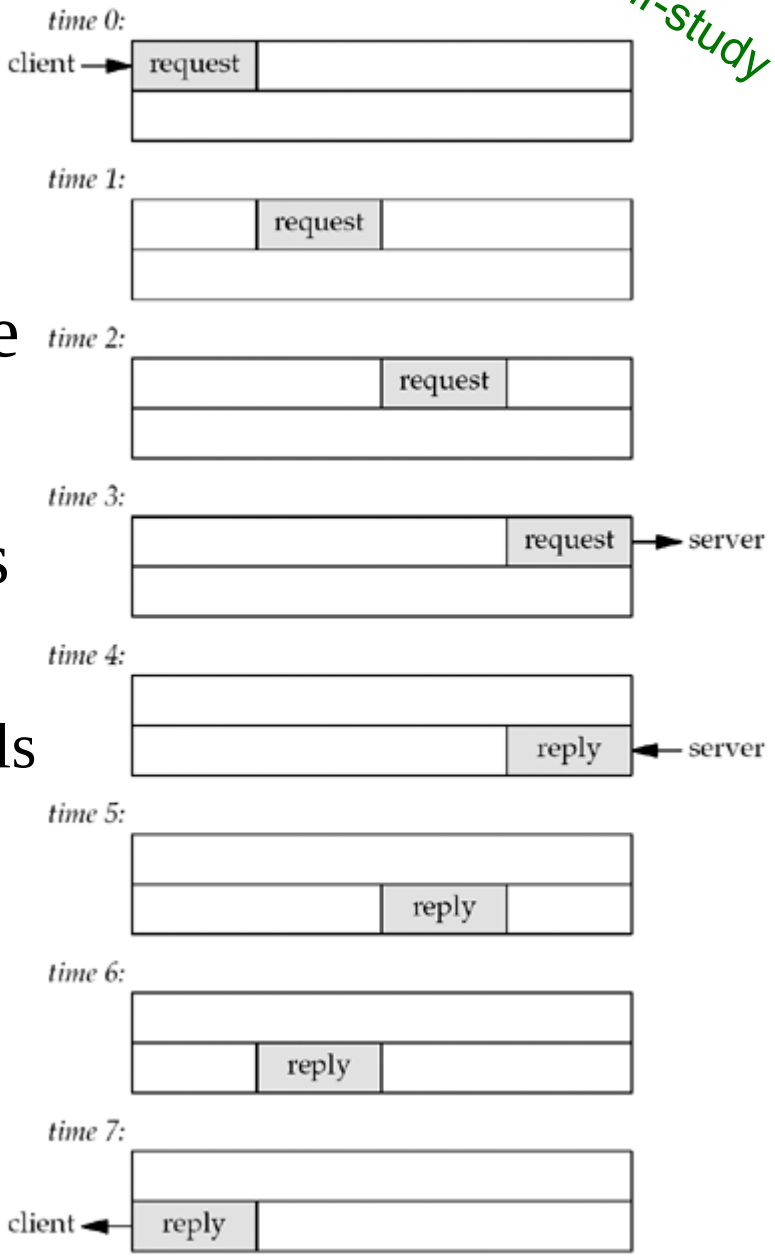
Self-study

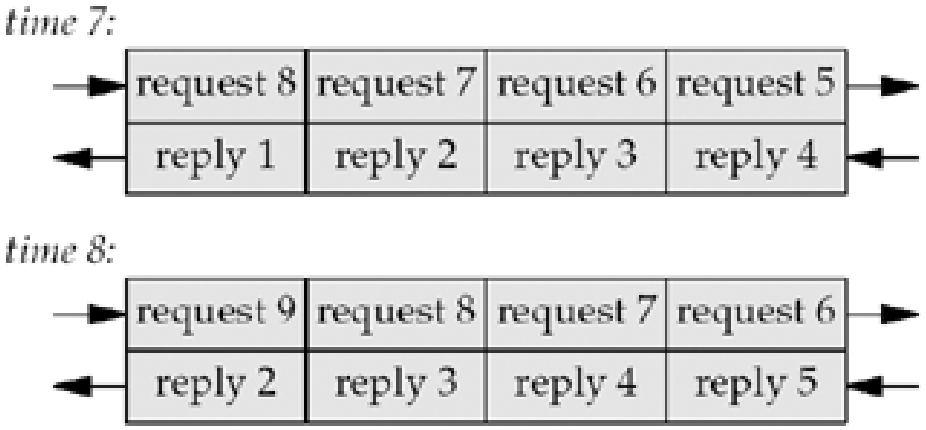


Batch Input and Buffering

Self-study

- Can run our client in a batch mode
  - Redirect the input and output,
- However, the output file is always smaller than the input file
  - should be identical for echo protocols





Batch Input and Buffering

Self-study

- Assume that the input file contains only nine lines
  - The last line is sent at time 8
- Cannot close the connection after writing this request
  - there are still other requests and replies in the pipe
- Problem cause is our handling of an EOF on input:
  - `str_cli` returns to the main function, which then terminates
- But in a batch mode, an EOF on input does not imply that we have finished reading from the socket;
  - might still be requests on the way to the server, or replies on the way back from the server

Batch Input and Buffering

Self-study

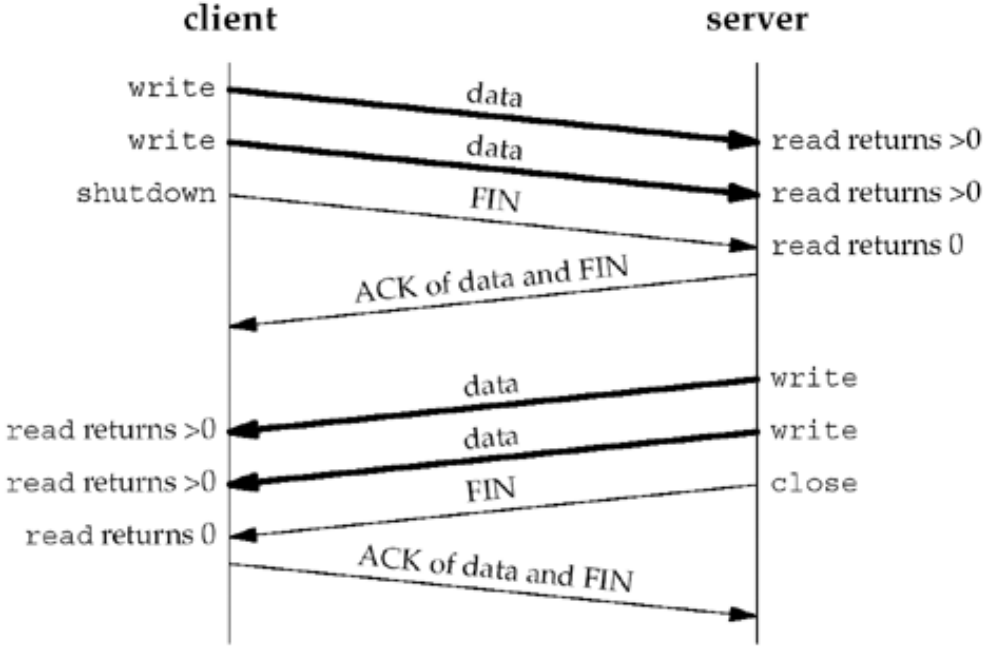
- Solution:
  - send a FIN to the server, telling it we have finished sending data, but leave the socket descriptor open for reading

`shutdown` Function

Self-study

```
int shutdown(int sockfd, int howto);
```

- `SHUT_RD` read half of the connection is closed
- `SHUT_WR` write half of the connection is closed
- `SHUT_RDWR` read half and the write half of the connection are both closed



str_cli (Revisited Again)

Self-study

```
void str_cli(FILE *fp, int sockfd){
int maxfdp1, stdineof; fd_set  rset;  char buf[MAXLINE]; int n;
 stdineof = 0; FD_ZERO(&rset);
 for ( ; ; ) {
 if (stdineof == 0) FD_SET(fileno(fp), &rset);
 FD_SET(sockfd, &rset);
 maxfdp1 = max(fileno(fp), sockfd) + 1;
 Select(maxfdp1, &rset, NULL, NULL, NULL);
 if (FD_ISSET(sockfd, &rset)) {//socket is readable
  if ( (n = Read(sockfd, buf, MAXLINE)) == 0)
   if (stdineof == 1) return;//normal termination
   else err_quit("str_cli: server terminated prematurely");
  Write(fileno(stdout), buf, n);
 }
```

str_cli (Revisited Again)

Self-study

```
 if (FD_ISSET(fileno(fp), &rset)) {//input is readable
  if ( (n = Read(fileno(fp), buf, MAXLINE)) == 0) {
   stdineof = 1;
   Shutdown(sockfd, SHUT_WR);//send FIN
   FD_CLR(fileno(fp), &rset);
   continue;
  }
  Writen(sockfd, buf, n);
  }
 }
}
```

TCP Echo Server (Revisited)

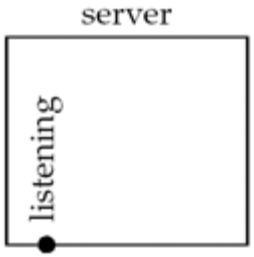
Self-study

Has a listening descriptor

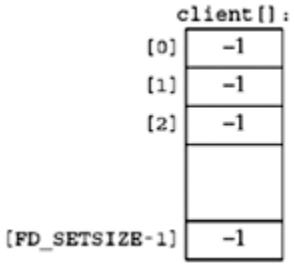
- maintains 1 read set
- starts in the foreground, so descriptors 0, 1, and 2 are set to input, output, and error
- first available descriptor for the listening socket is 3

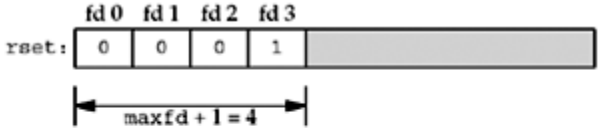
integer array contains the connected socket

- initialized to -1 expect the listening sockets
- 1st argument to `select` will be 4



TCP server before first client has established a connection



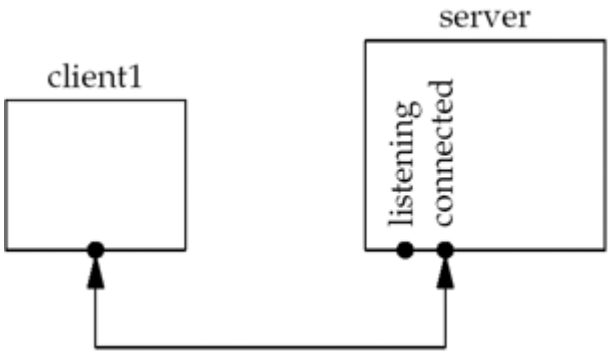


Data structures for TCP server with just a listening socket

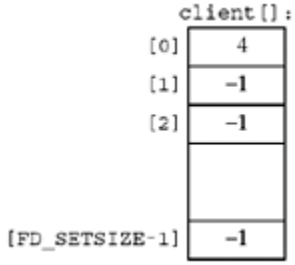
TCP Echo Server (Revisited)

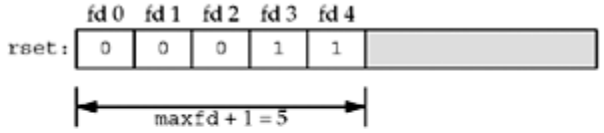
Self-study

- 1st client establishes a connection with the server
  - the listening descriptor becomes readable and the server calls accept
  - new connected descriptor returned by accept will be 4
- The server must remember the new connected socket in its client array, and the connected socket must be added to the descriptor set



TCP server after first client establishes connection



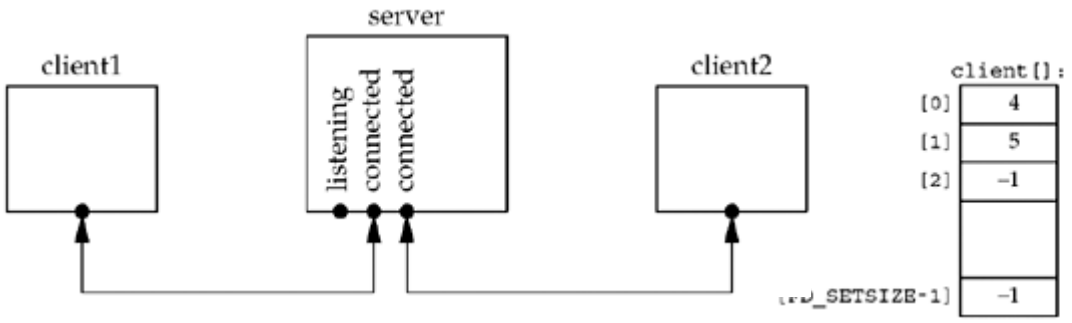


Data structures after first client connection is established

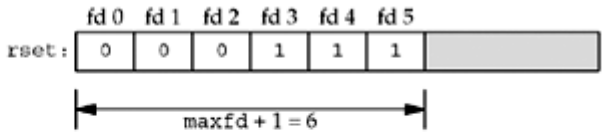
TCP Echo Server (Revisited)

Self-study

- Sometime later a 2nd client establishes a connection
- The new connected socket (which we assume is 5) must be remembered



TCP server after second client connection is established

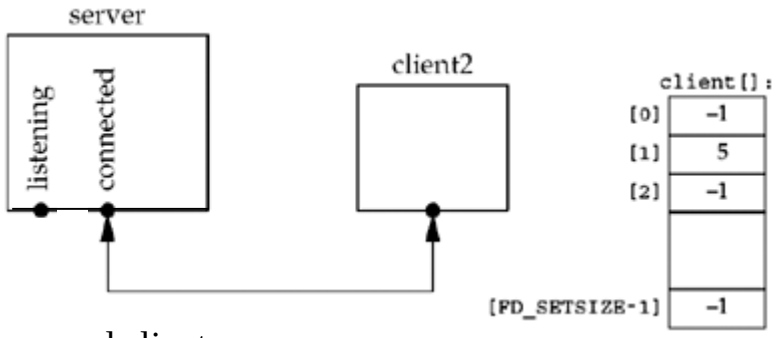


Data structures after second client connection is established

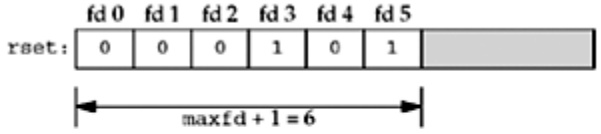
TCP Echo Server (Revisited)

Self-study

- 1st client terminates its connection
- Client sends a FIN; makes descriptor 4 is readable
  - server reads this connected socket, read returns 0
  - close this socket and update our data structures
  - `client[0]` is set to -1 and descriptor 4 in the set is set to 0



TCP server after second client connection is established



Data structures after first client terminates its connection

TCP Echo Server (Revisited)

Self-study

```
int main(int argc, char **argv){
int i, maxi, maxfd, listenfd, connfd, sockfd;
int nready, client[FD_SETSIZE];
ssize_t n;
fd_set rset, allset;
char buf[MAXLINE];
socklen_t  clilen;
struct sockaddr_in cliaddr, servaddr;
```

TCP Echo Server (Revisited)

Self-study

```
listenfd = Socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin_family = AF_INET;

servaddr.sin_addr.s_addr = htonl(INADDR_ANY);

servaddr.sin_port = htons(SERV_PORT);

Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

Listen(listenfd, LISTENQ);

maxfd = listenfd;                  //initialize

maxi = -1;                         //index into client[] array

for (i = 0; i < FD_SETSIZE;  i++) client[i] = -1;//-1 indicates
  available entry

FD_ZERO(&allset);

FD_SET(listenfd, &allset);
```

TCP Echo Server (Revisited)

Self-study

```
for ( ; ; ) {
  rset = allset;           // structure assignment
  nready = Select(maxfd + 1, &rset, NULL, NULL, NULL);
  if (FD_ISSET(listenfd, &rset)) {       // new client connection
   clilen = sizeof(cliaddr);
   connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
   for (i = 0; i < FD_SETSIZE; i++)
    if (client[i] < 0) {
     client[i] = connfd; /* save descriptor */
     break;
   }
```

TCP Echo Server (Revisited)

Self-study

```
   if (i == FD_SETSIZE) err_quit("too many clients");
   FD_SET(connfd, &allset);//add new descriptor to set
   if (connfd > maxfd) maxfd = connfd;// for select
   if (i > maxi) maxi = i;// max index in client[] array
   if (--nready <= 0) continue;//no more readable descriptors
  }
  for (i = 0; i <= maxi; i++) {//check all clients for data
   if ( (sockfd = client[i]) < 0) continue;
   if (FD_ISSET(sockfd, &rset)) {
    if ( (n = Read(sockfd, buf, MAXLINE)) == 0) { //client closed
     Close(sockfd); FD_CLR(sockfd, &allset); client[i] = -1;
    } else Writen(sockfd, buf, n);
    if (--nready <= 0) break;//no more readable descriptors
}}}}
```

Denial-of-Service Attacks

Self-study

- What happens if a malicious client connects to the server, sends one byte of data (other than '\n'), and then goes to sleep?
  - Server will call read, which will read the single byte of data from the client and then block in the next call to read, waiting for more data from this client
  - The server is then blocked by this one client and will not service any other clients
  - either new client connections or existing clients' data
  - until the malicious client either sends a newline or terminates

Denial-of-Service Attacks

Self-study

- Servers can never block in a function call related to a single client might deny service to all other clients
- Solutions:
  - nonblocking I/O
  - each client serviced by a separate thread
  - place a timeout on the I/O operations

Summary

- The default is blocking I/O, which is also the most commonly used
- The most commonly used function for I/O multiplexing is `select`
  - We tell the `select` function what descriptors we are interested in (for reading, writing, and exceptions), the maximum amount of time to wait, and the maximum descriptor number (+1)

Summary

- We used our echo client in a batch mode using `select` and discovered that even though the end of the user input is encountered, data can still be in the pipe to or from the server
  - To handle this scenario requires the `shutdown` function, and it lets us take advantage of TCP's half-close feature
- The dangers of mixing stdio buffering (as well as our own `readline` buffering) with select caused us to produce versions of the echo client and server that operated on buffers instead of lines

GDB

Self-study

- Tools to help make programming task simpler
  - GDB helps the debugging task
  - DDD: GDB's visual version

  - Debugging
    - Program defensively – check error returns, buffer sizes
    - Build program in a modular fashion
    - Print sensible error messages
      - May want to do better than the built-in wrapper functions

Debugging with GDB

Self-study

- Prepare program for debugging
  - Compile with "`-g`" (keep full symbol table)
  - Don't use compiler optimization ("`-O`", "`-O2`", …)
- Two main ways to run gdb
  - On program directly
    - `gdb progname`
    - Once gdb is executing we can execute the program with:
      - `run args`
      - Can use shell-style redirection e.g. `run < infile > /dev/null`
  - On a core (post-mortem)
    - `gdb progname core`
    - Useful for examining program state at the point of crash
- Extensive in-program documentation exists
  - `help` (or `help <topic>` or `help <command>`)

Self-study

Controlling Your Program with GDB

- Stopping your program with breakpoints
  - Program will run until it encounters a breakpoint
    - To start running again: `cont`
  - Break command format
    - `break foo.c:4` stops at the 4th source line of `foo.c`
    - `break 16` stops at the 16th source line of the current source file
    - `break main` stops upon entry to the `main()` function
- Stop your program with SIGINT (CTRL-C)
  - Useful if program hangs (sometimes)
- Stepping through your program
  - `step N` command: steps through N source lines (default 1)
  - `next` is like `step` but it treats function calls as a single line
- Hint: avoid writing mega-expressions
  - Hard to step through `foo(bar(tmp = baz(), tmp2 = baz2()))`

Self-study

Examining the State of your Program

- `backtrace` (`bt` for short)
  - Shows stack trace (navigate procedures using `up` and `down`)
  - `bt full` prints out all available local variables as well
- `print EXP`
  - Print the value of expression
  - Can print out values in variables
- `x/<count><format><size> ADDR`
  - Examine a memory region at ADDR
  - Count is the number of items to display (default: 1)
  - Format is a single letter code
    - o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string)
  - Size is the size of the items (single letter code)
    - b(byte), h(halfword), w(word), g(giant, 8 bytes)

More information…

Self-study

- GDB
  - Official GDB homepage: http://www.gnu.org/software/gdb/gdb.html
  - GDB primer: http://www.cs.pitt.edu/~mosse/gdb-note.html

Testing Servers Using `telnet`

Self-study

- The `telnet` program is invaluable for testing servers that transmit ASCII strings over Internet connections
  - Our simple echo server
  - Web servers
  - Mail servers
- Usage:
  - `unix> telnet <host> <portnumber>`
  - Creates a connection with a server running on *`<host>`* and listening on port *`<portnumber>`*.

Socket Programming Help

Self-study

- man is your friend (aka RTFM)
  - man accept
  - man select
  - Etc.
- The manual page will tell you:
  - What #include<> directives you need at the top of your source code
  - The type of each argument
  - The possible return values
  - The possible errors (in errno)

Simplified

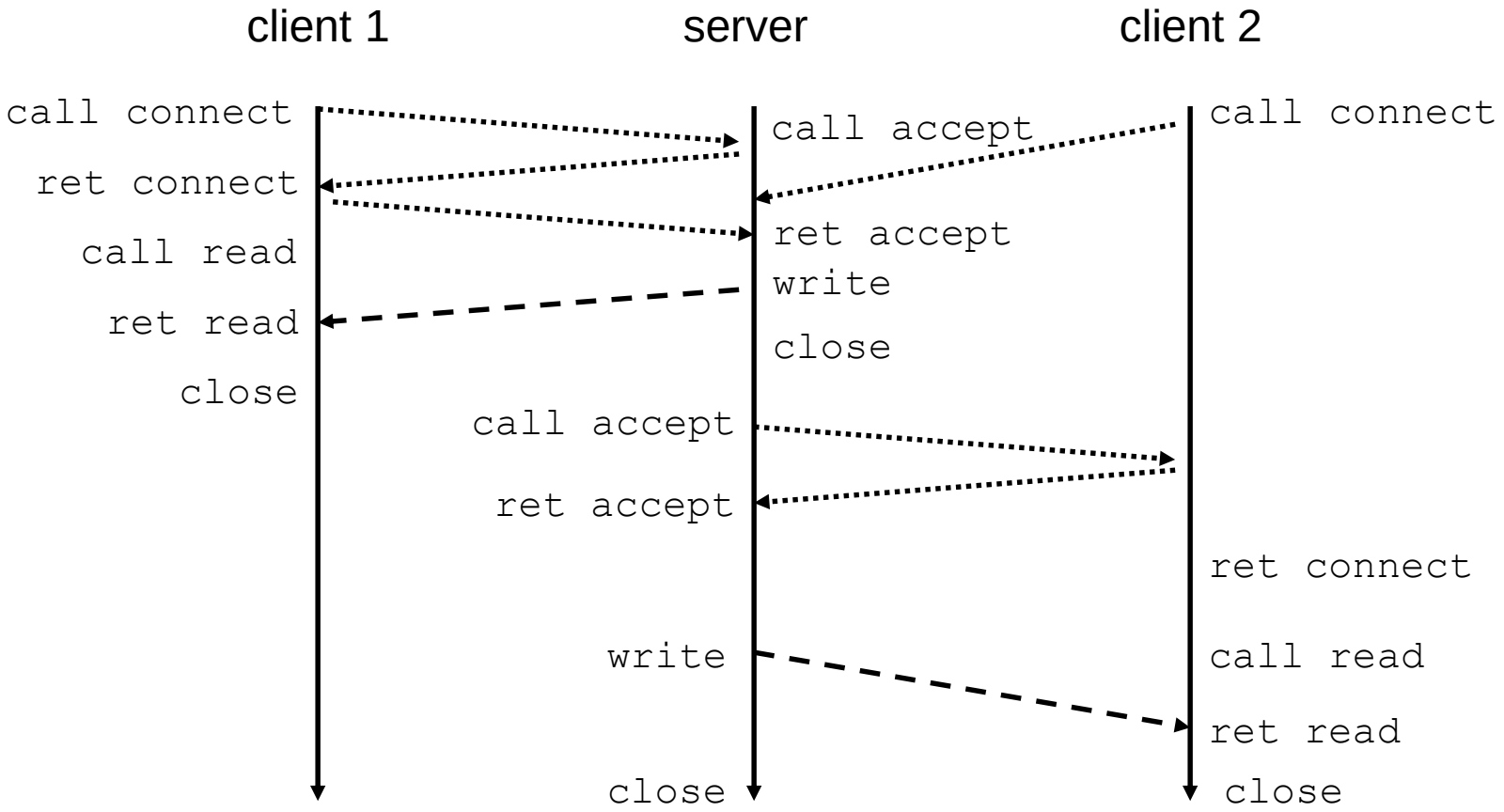
Self-study

Types of Server Implementations

Iterative Servers

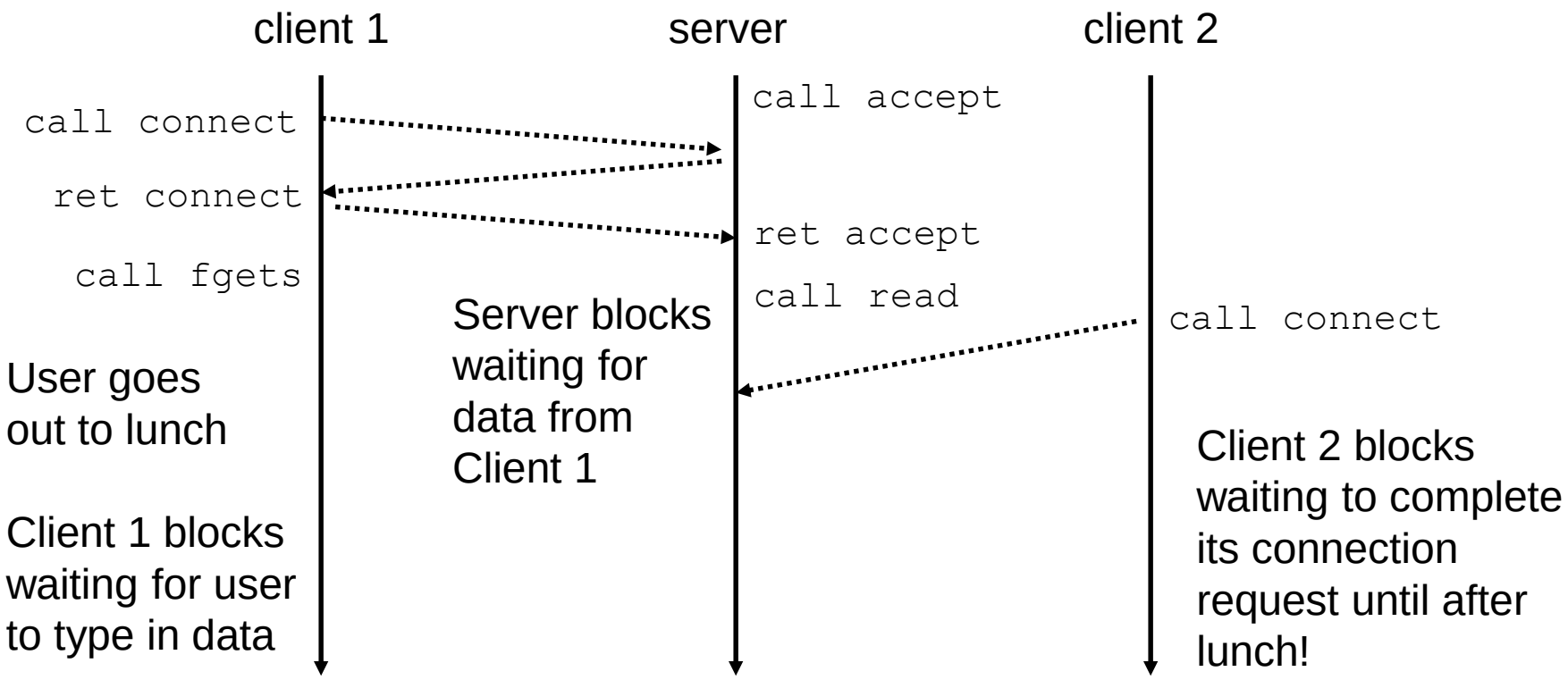
Self-study

- Iterative servers process one request at a time.



Flaw of Iterative Servers

Self-study

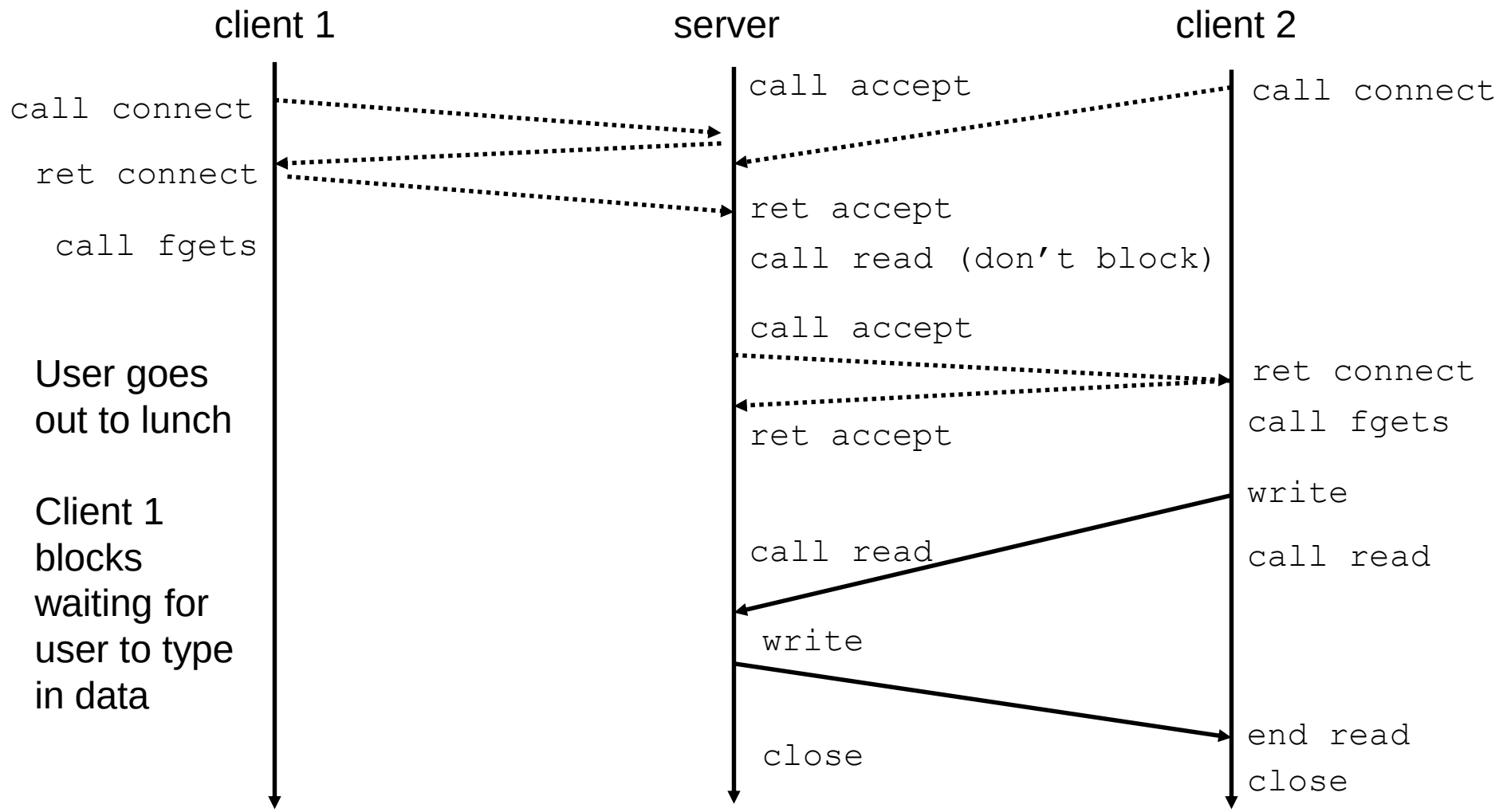


- Solution: use *concurrent servers* instead.
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.

Concurrent Servers

Self-study

- Concurrent servers handle multiple requests concurrently.



Possible Mechanisms for Creating Concurrent Flows

Self-study

- 1. Processes
  - Kernel automatically interleaves multiple logical flows.
  - Each flow has its own private address space.
- 2. I/O multiplexing with `select()` Our Focus
  - User manually interleaves multiple logical flows.
  - Each flow shares the same address space.
  - Popular for high-performance server designs.
- 3. Threads
  - Kernel automatically interleaves multiple logical flows.
  - Each flow shares the same address space.

Event-Based Concurrent Servers Using I/O Multiplexing

Self-study

- Maintain a pool of connected descriptors.
- Repeat the following forever:
  - Use the Unix `select f`unction to block until:
    - (a) New connection request arrives on the listening descriptor.
    - (b) New data arrives on an existing connected descriptor.
  - If (a), add the new connection to the pool of connections.
  - If (b), read any available data from the connection
    - Close connection on EOF and remove it from the pool.

Self-study

Pro and Cons of Event-Based Designs

- + One logical control flow.
- + Can single-step with a debugger.
- + No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines.
- - Significantly more complex to code than process- or thread-based designs.
- - Can be vulnerable to denial of service attacks
  - ?

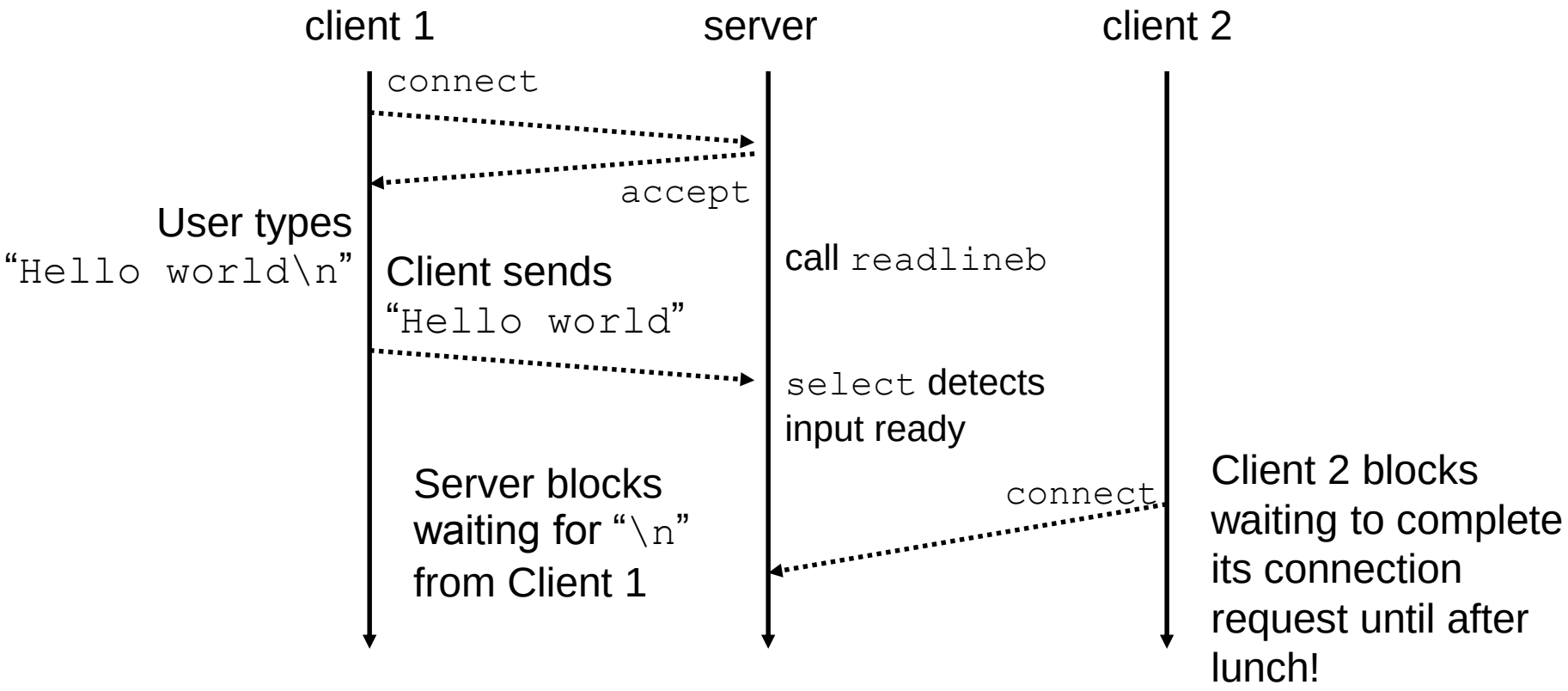
Attack #1

Self-study

- Overwhelm Server with Connections
  - Limited to FD_SETSIZE – 4 (typically 1020) connections
- Defenses?

Attack #2: Partial Lines

Self-study



- Client gets attention of server by sending partial line
- Server blocks until line completed

Flaky Client

Self-study

```
    while (Fgets(buf, MAXLINE, stdin) != NULL) {
        Rio_writen(clientfd, buf, strlen(buf)-1);
        Fgets(buf, MAXLINE, stdin);  /* Read & ignore line */
        Rio_writen(clientfd, "\n", 1);
        Rio_readlineb(&rio, buf, MAXLINE);
        Fputs(buf, stdout);
    }
```

- Sends everything up to newline
- Doesn't send newline until user types another line
- Meanwhile, server will block

Self-study

Implementing a Robust Server

- Break Up Reading Line into Multiple Partial Reads
  - Every time connection selected, read as much as is available
  - Construct line in separate buffer for each connection
- Must Use Unix Read

```
read(int fd, void *buf, size_t maxn)
```

  - Read as many bytes as are available from file `fd` into buffer `buf`.
  - Up to maximum of `maxn` bytes

Self-study

NOTE (Just to complicate things…)

- If a client sends x bytes of data in one write() call, it is NOT guaranteed that all x bytes will be received in a single read() call by the server.
- i.e., the following scenario is possible:
  - Client writes “Hello world\n” to server
  - Server’s select() notices & unblocks, server then calls read()
  - read() returns “Hell”
  - A subsequent call to read() returns “o w”
  - A subsequent call to read() returns “orld\n”
- Server’s solution: maintain a buffer for each of your clients, and only process the buffer’s contents when a message has been received in full (note: the type of application determines what a ‘message’ is, and what indicates it has been fully received)

Conceptual Model

Self-study

- Maintain State Machine for Each Connection
  - First Version: State is just identity of connfd
  - Second Version: State includes partial line + count of characters
- `select` Determines Which State Machine to Update
  - First Version: Process entire line
  - Second Version: Process as much of line as is available
- Design Issue
  - Must set granularity of state machine to avoid server blocking