香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Operating Systems
Lecture 6

System programming: sockets, pipes

Prof. Yunming XIAO

School of Data Science

Acknowledgments: Ion Stoica and Matei Zaharia, Berkeley CS 162

Recall: what's in an open file description?

- Inside kernel!
- For our purpose, the two most important things are:
  - Where to find the file data on disk
  - The current position within the file

Linux:
https://github.com/torvalds/linux/blob/master/include/linux/fs.h

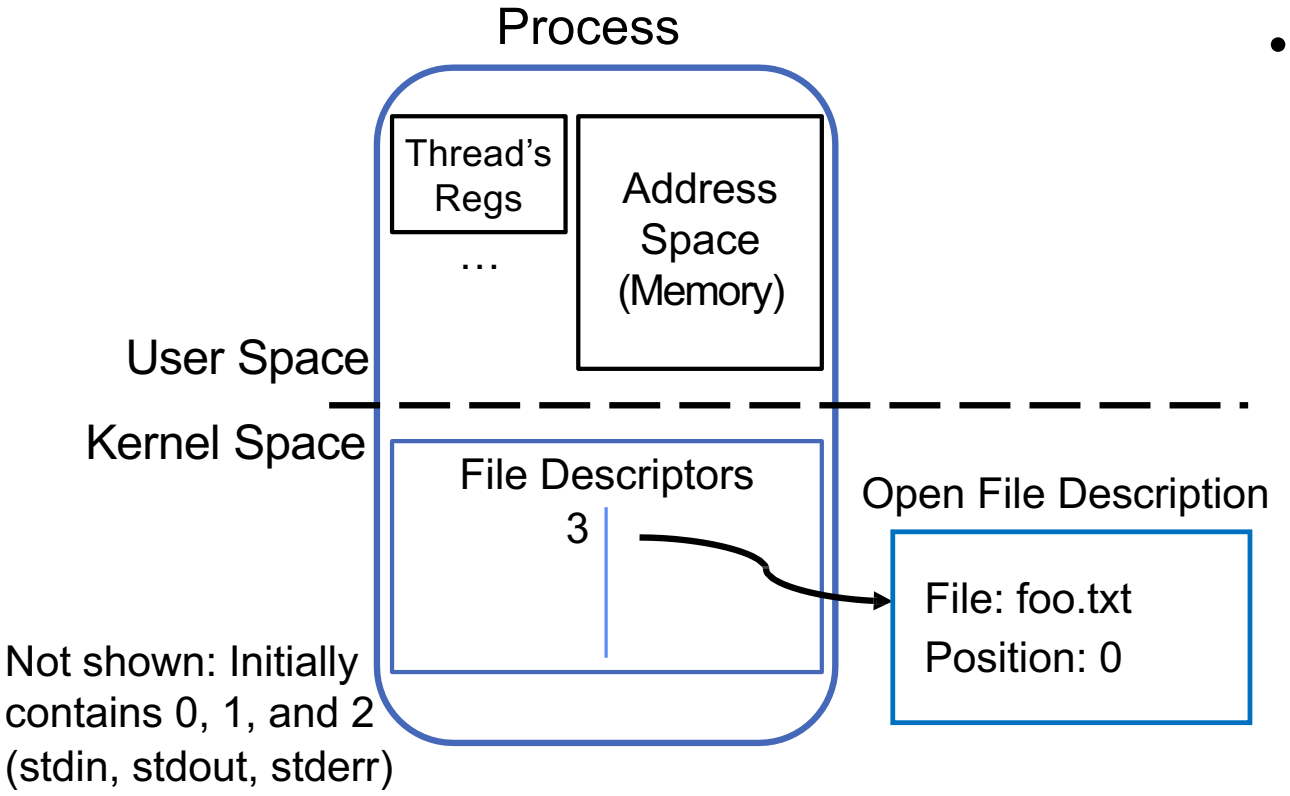
xv6:
https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/file.h

linux / include / linux / fs.h

Code | Blame | 3609 lines (3186 loc) · 117 KB

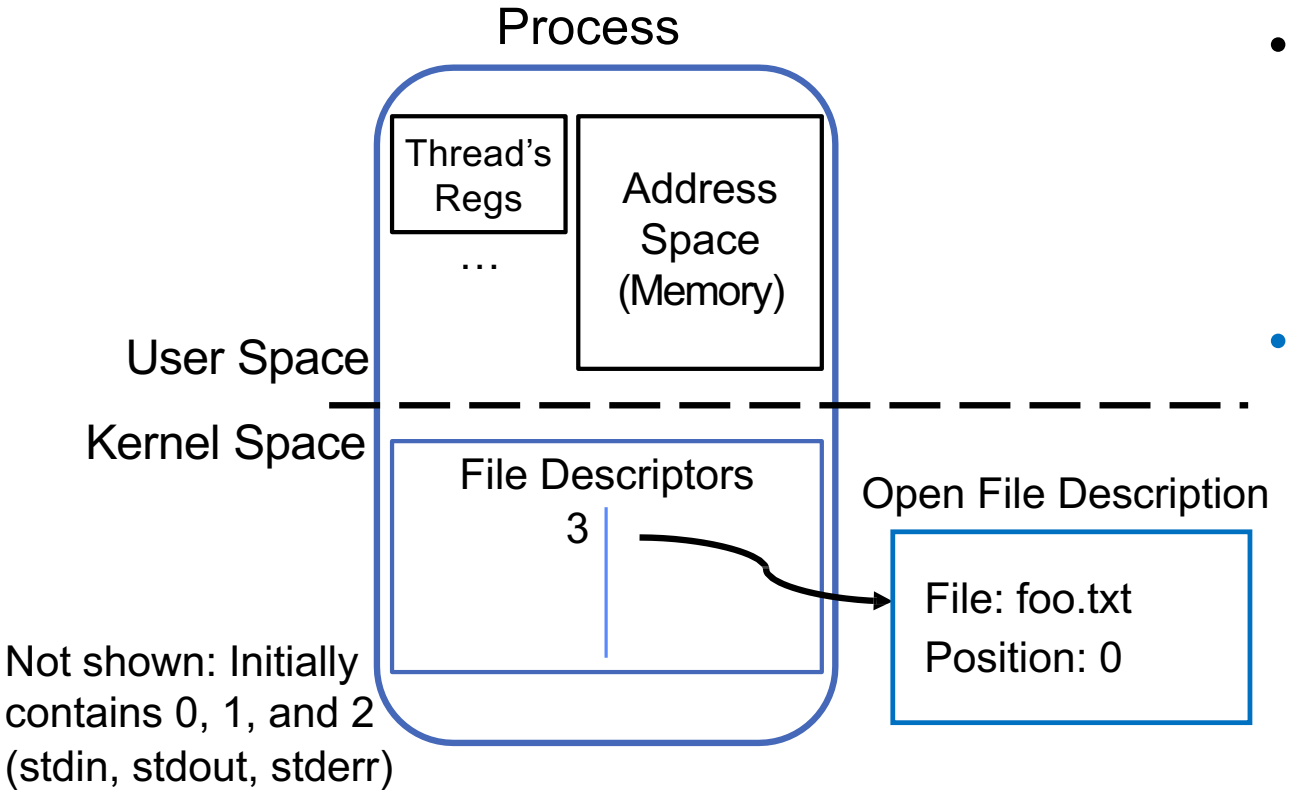
```
1257     */
1258    struct file {
1259            spinlock_t                      f_lock;
1260            fmode_t                         f_mode;
1261            const struct file_operations    *f_op;
1262            struct address_space            *f_mapping;
1263            void                            *private_data;
1264            struct inode                    *f_inode;
1265            unsigned int                    f_flags;
1266            unsigned int                    f_iocb_flags;
1267            const struct cred               *f_cred;
1268            struct fown_struct              *f_owner;
1269            /* --- cacheline 1 boundary (64 bytes) --- */
1270            union {
1271                    const struct path       f_path;
1272                    struct path             __f_path;
1273            };
1274            union {
1275                    /* regular files (with FMODE_ATOMIC_POS) and directories */
1276                    struct mutex            f_pos_lock;
1277                    /* pipes */
1278                    u64                     f_pipe;
1279            };
1280            loff_t                          f_pos;
1281    #ifdef CONFIG_SECURITY
1282            void                            *f_security;
1283    #endif
1284            /* --- cacheline 2 boundary (128 bytes) --- */
1285            errseq_t                        f_wb_err;
1286            errseq_t                        f_sb_err;
```

Process-specific file descriptor table inside kernel



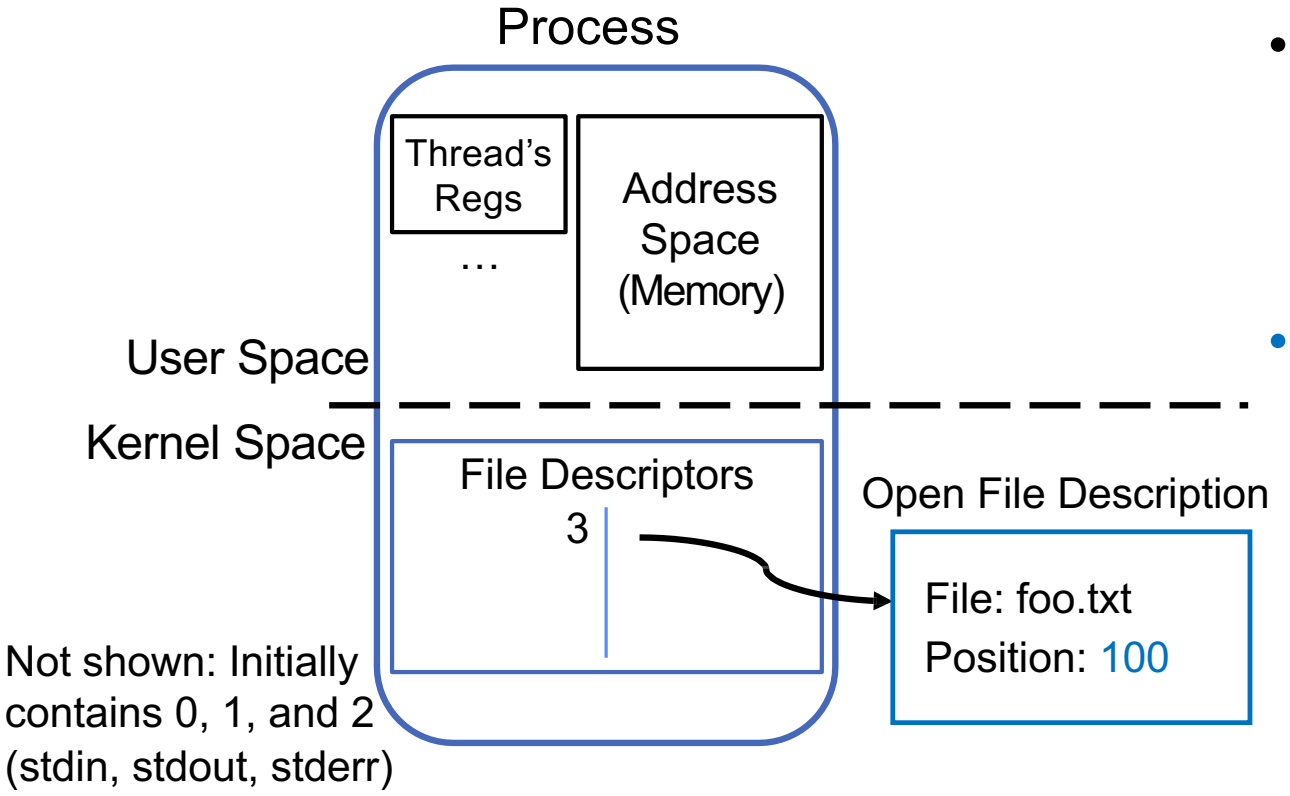
- Suppose that we execute `open("foo.txt")` and that the result is 3

Process-specific file descriptor table inside kernel



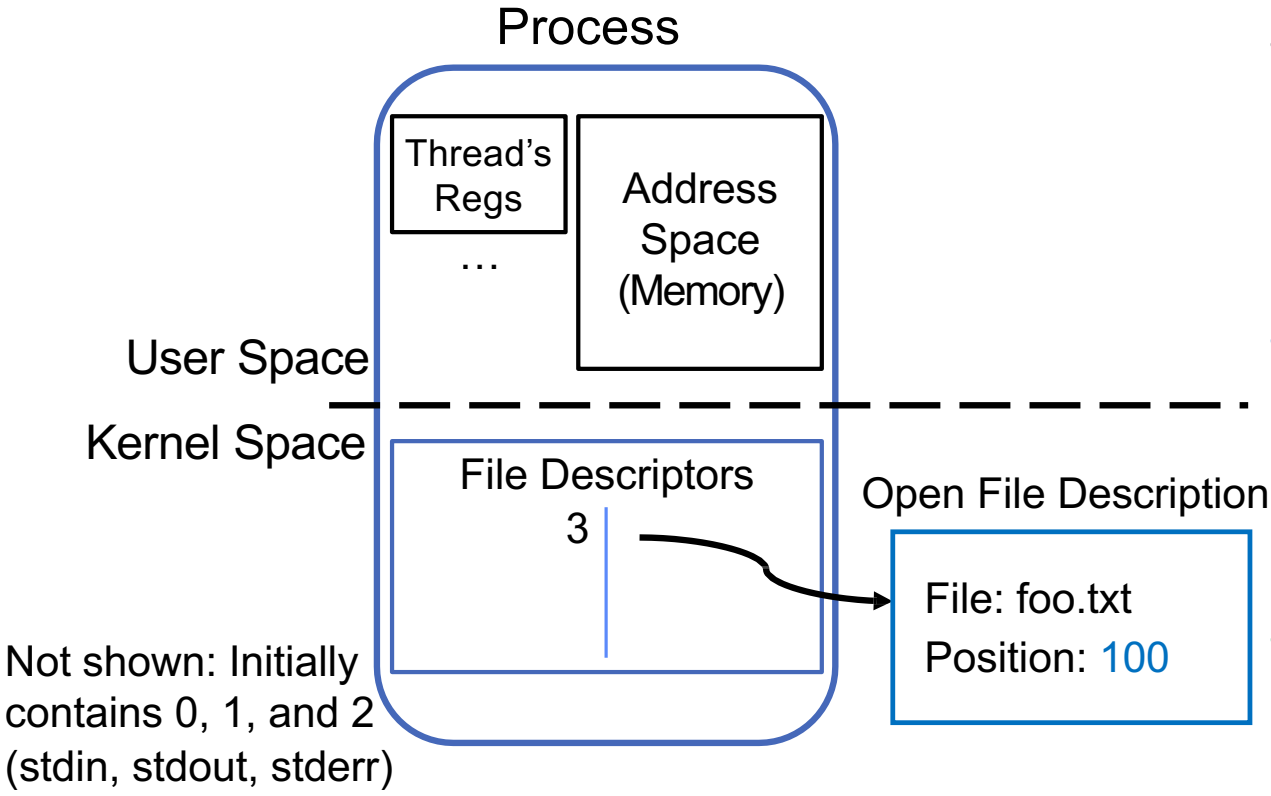
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100

Process-specific file descriptor table inside kernel



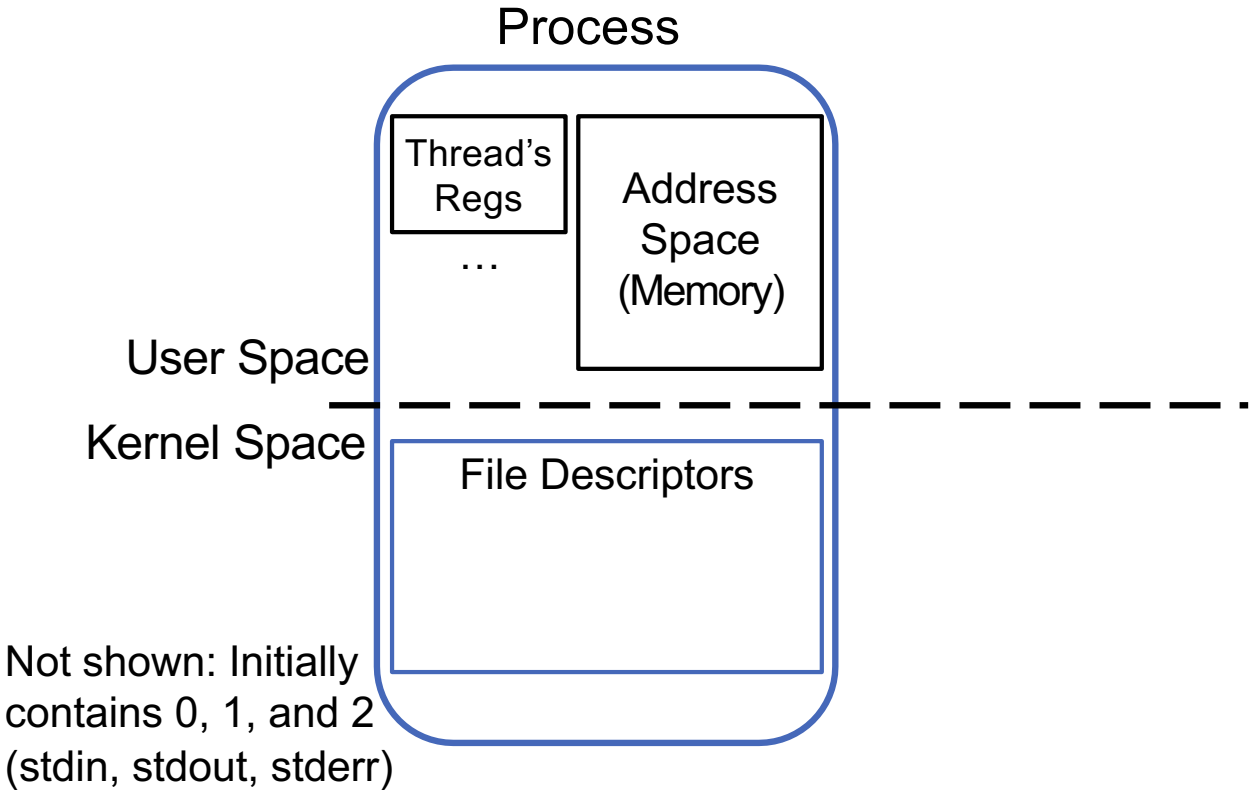
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100

Process-specific file descriptor table inside kernel



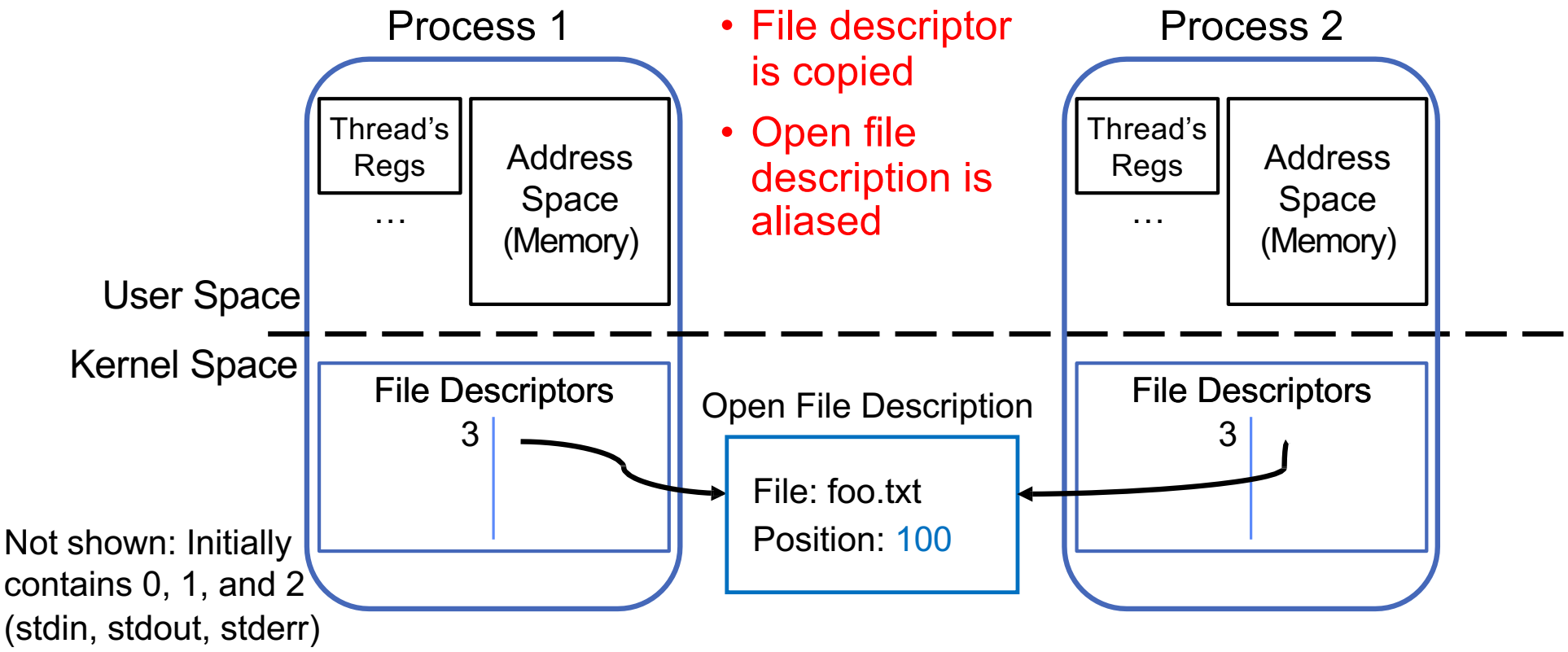
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100
- Finally, suppose that we execute `close(3)`

Process-specific file descriptor table inside kernel

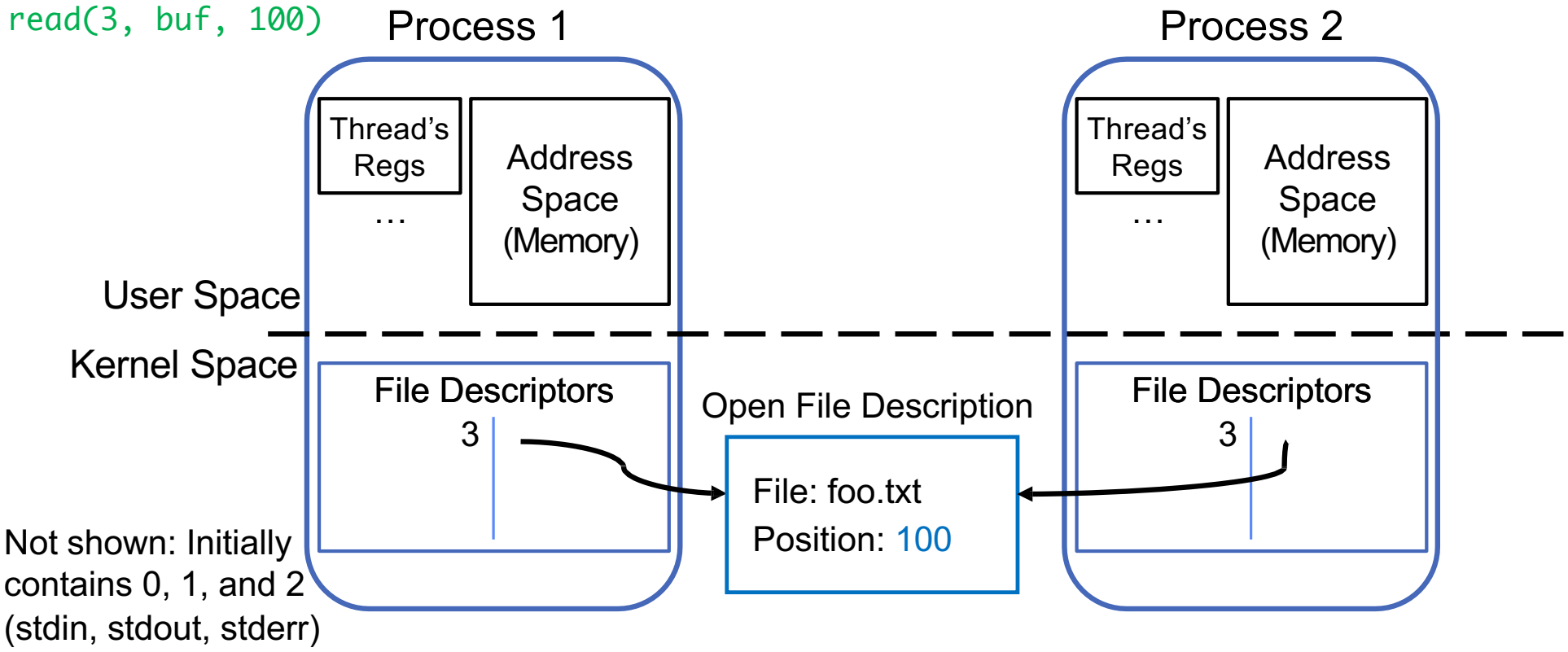


- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100
- Finally, suppose that we execute `close(3)`

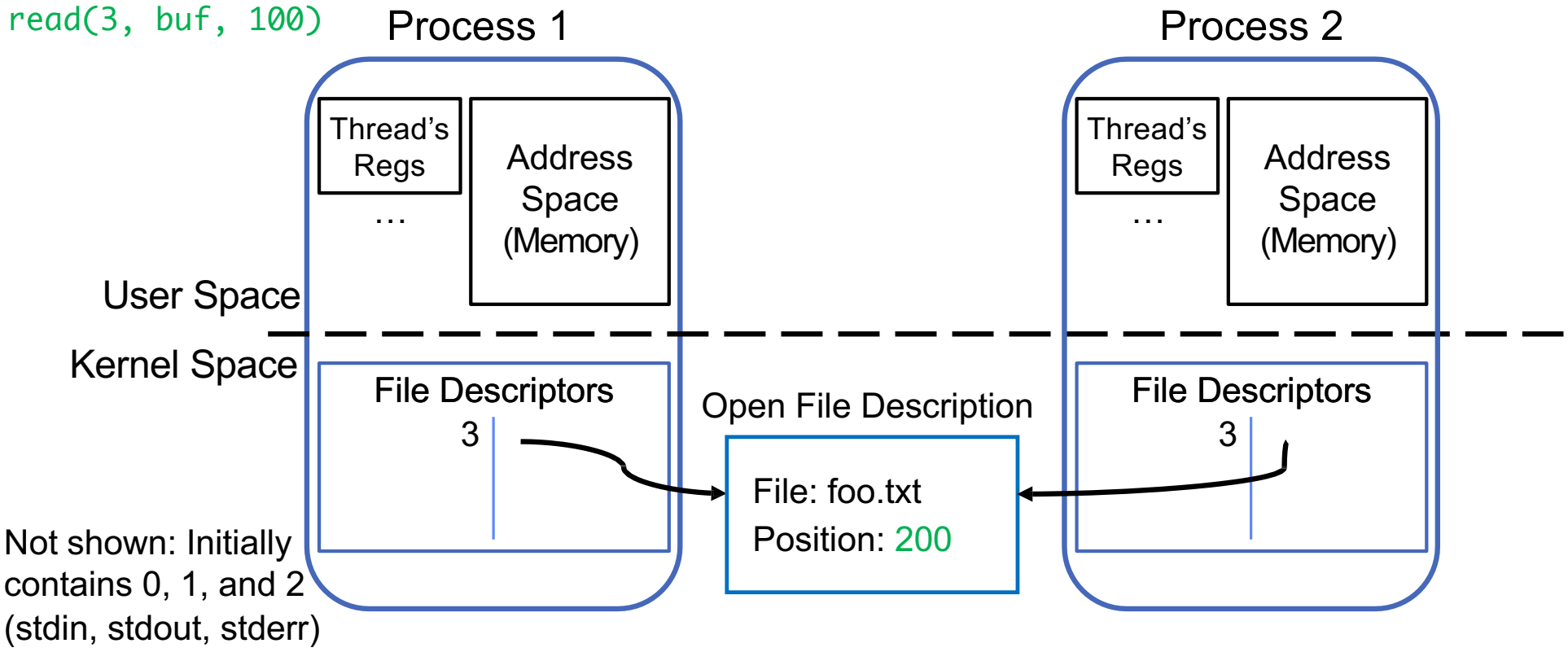
Instead of closing, let's fork()!



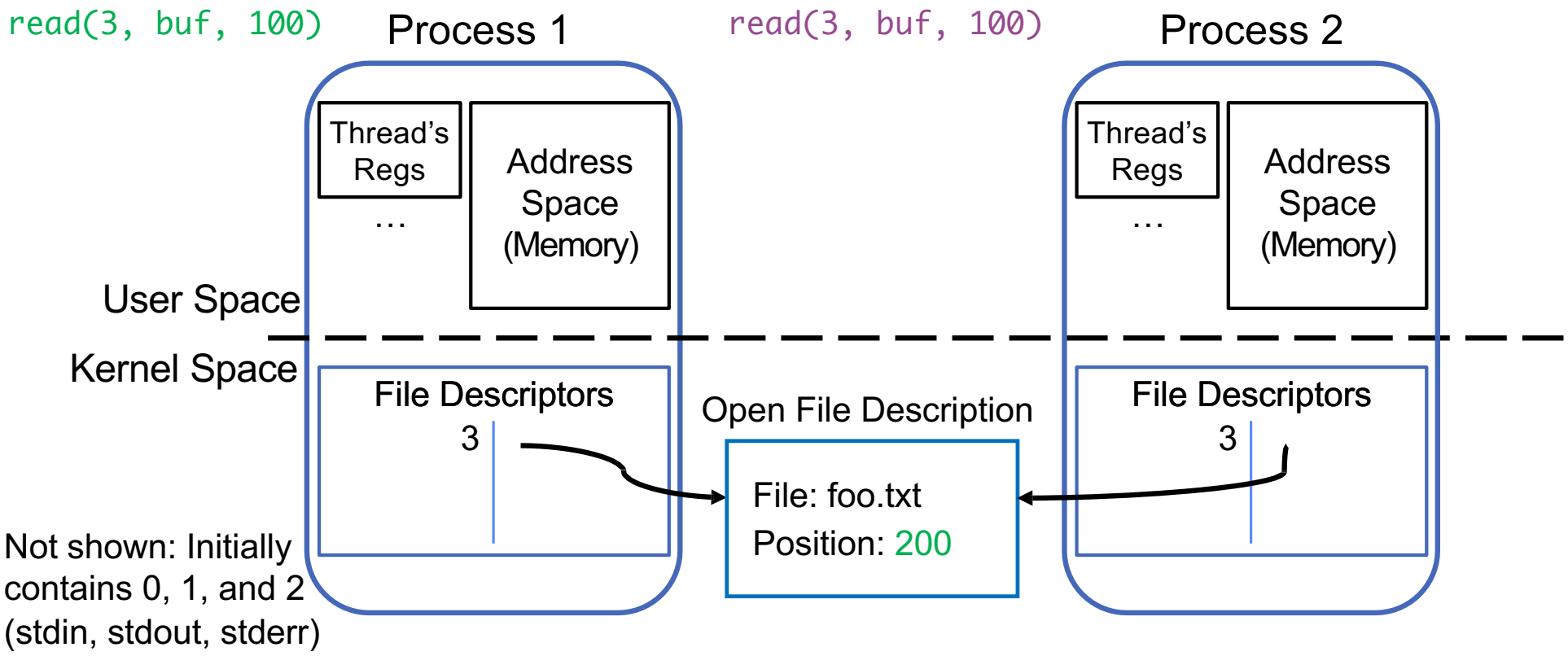
Open file description is *aliased*



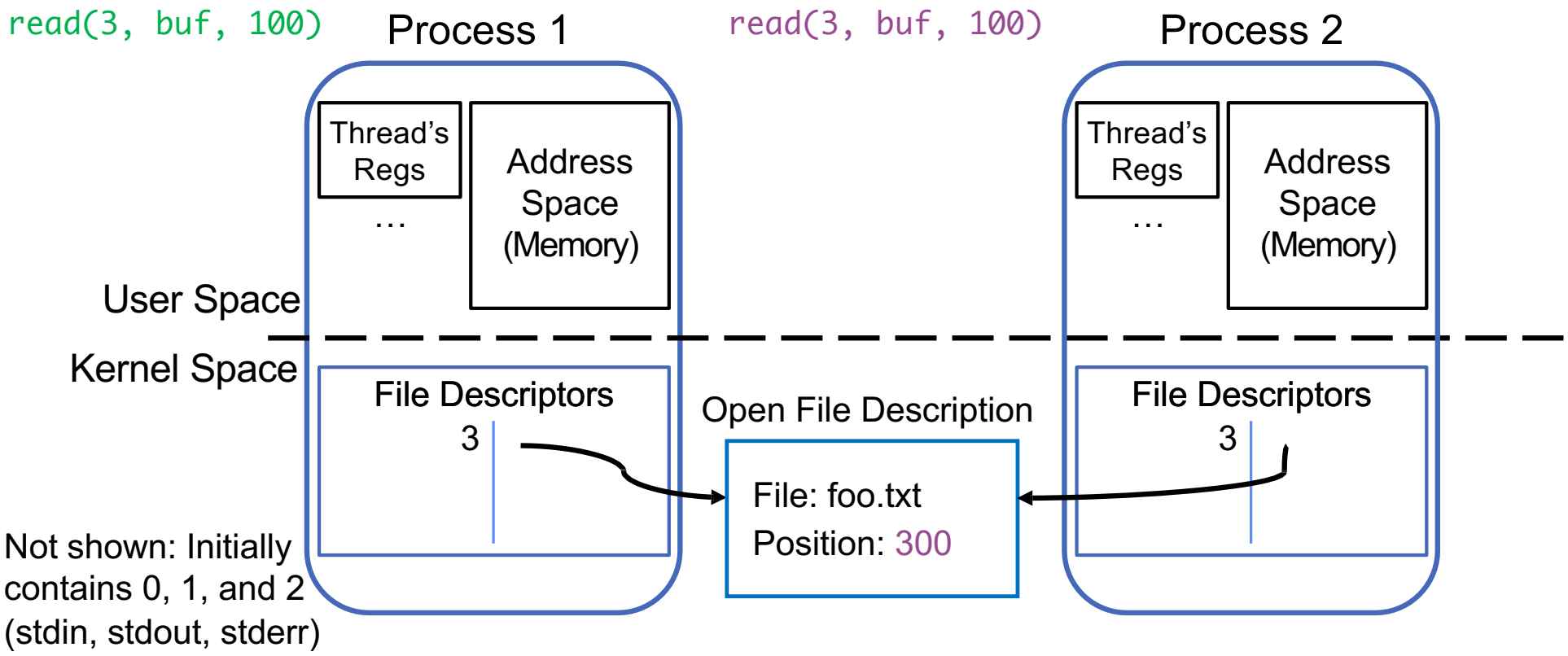
Open file description is *aliased*



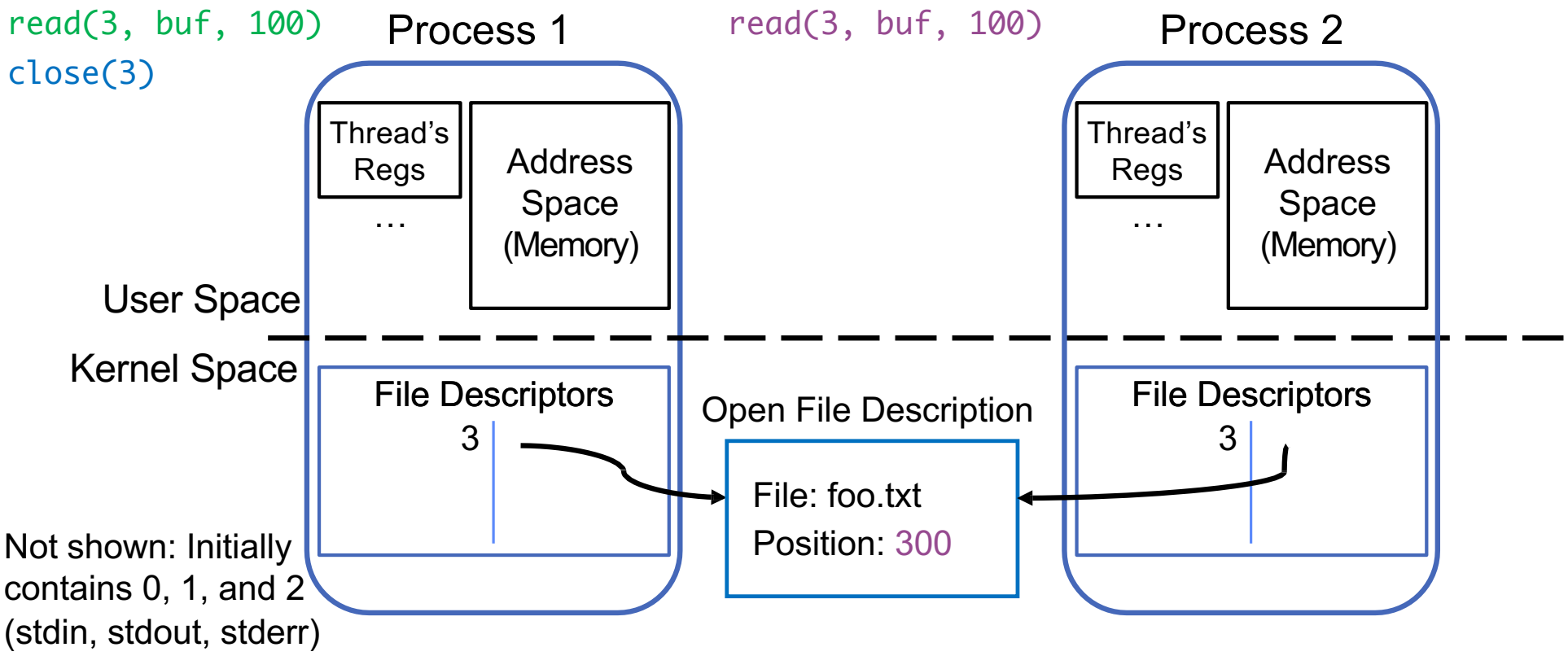
Open file description is *aliased*



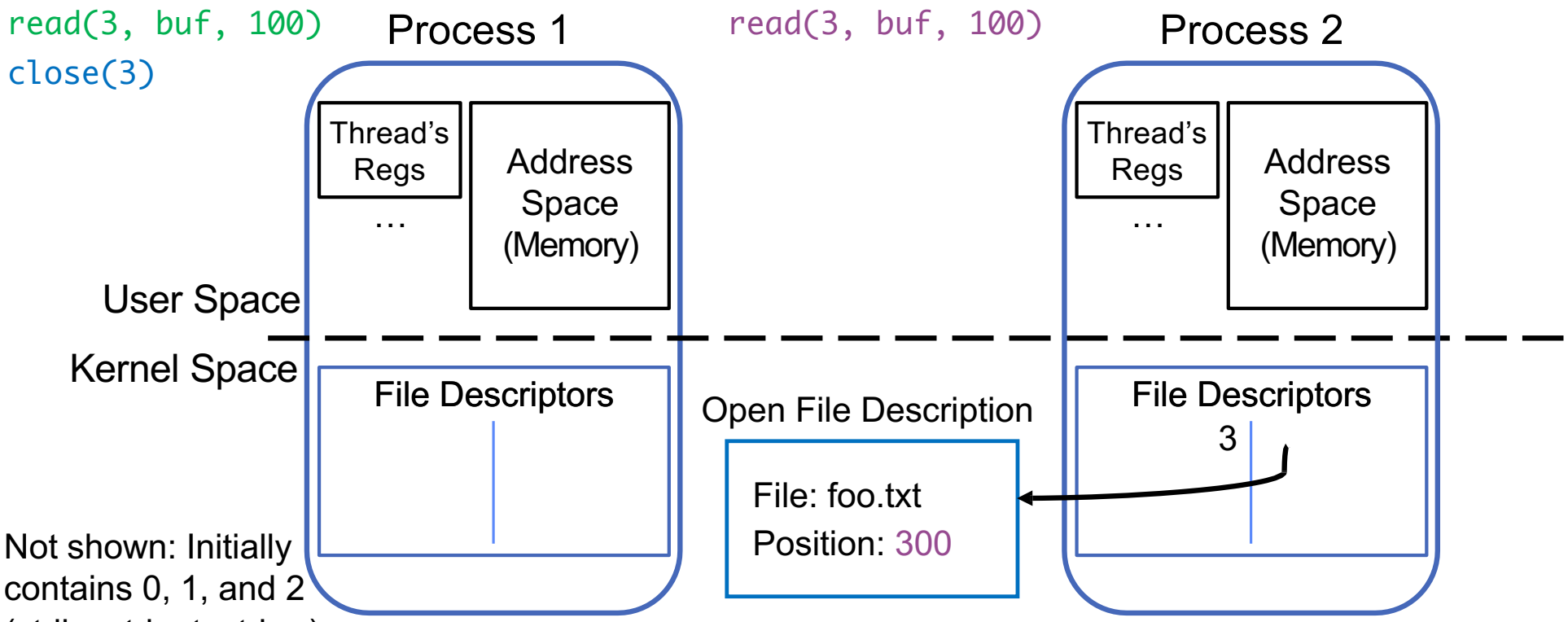
Open file description is *aliased*



File description is *copied*



File description is *copied*



- Open file description remains alive until no file descriptors in any process refer to it

End of recall

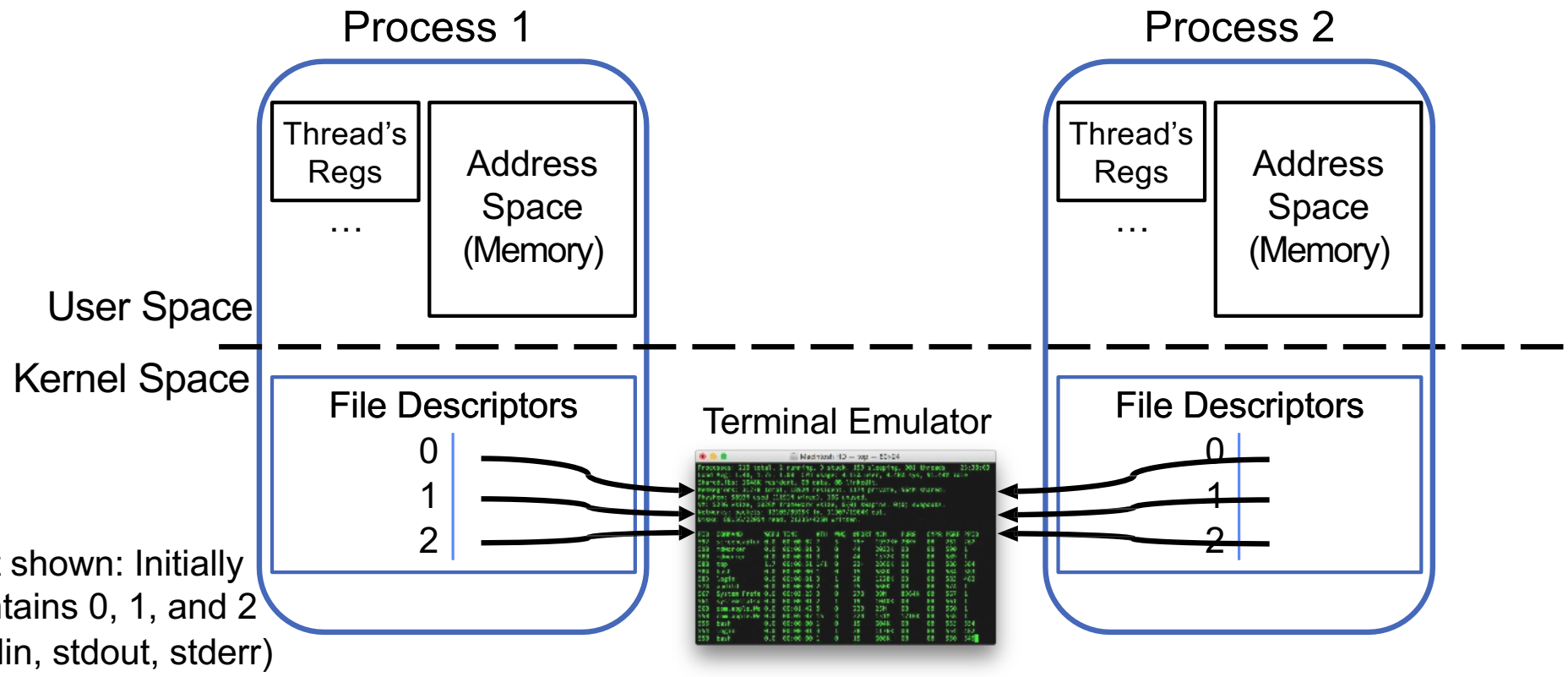
Why is aliasing the open file description a good idea?

- It allows for *shared resources* between processes

Example: shared terminal emulator

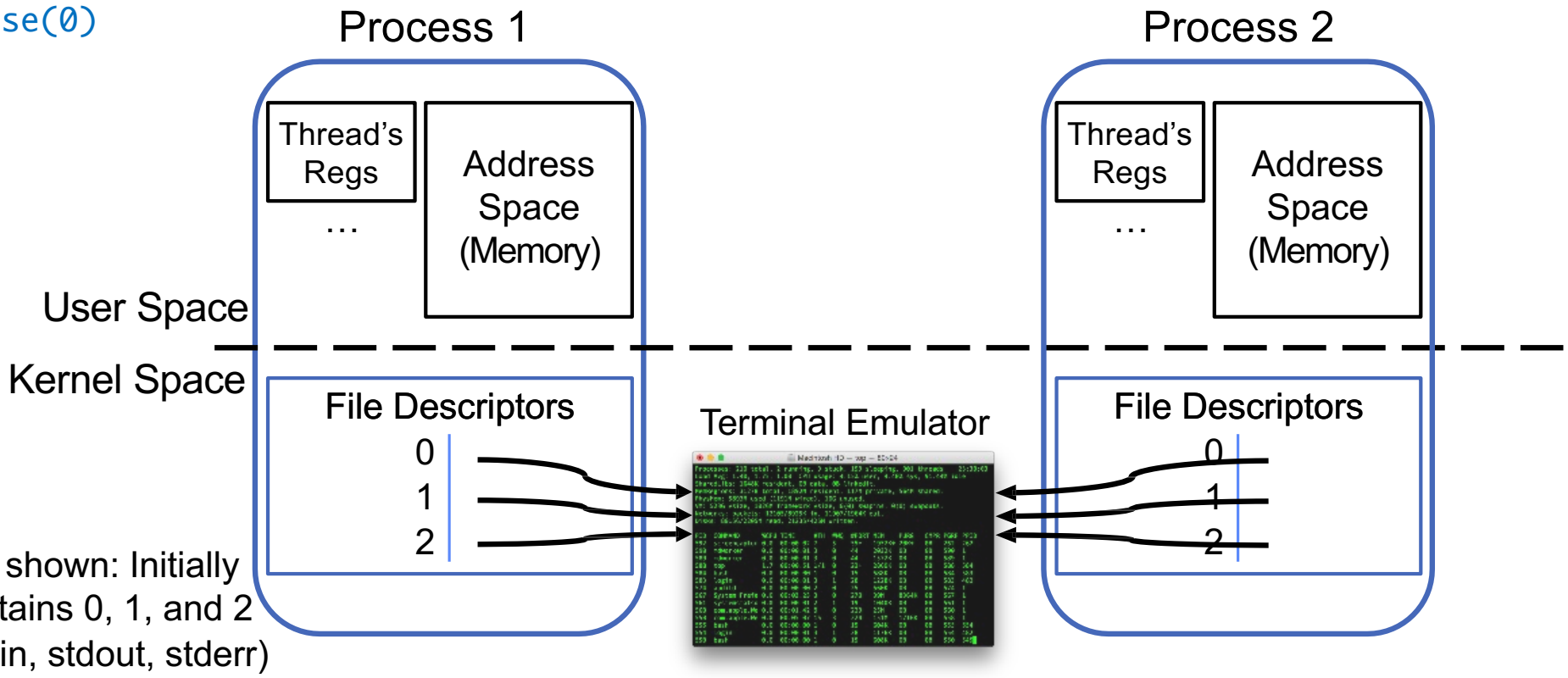
- When you `fork()` a process, the parent's and child's `printf` outputs go to the same terminal

Example: shared terminal emulator



Example: shared terminal emulator

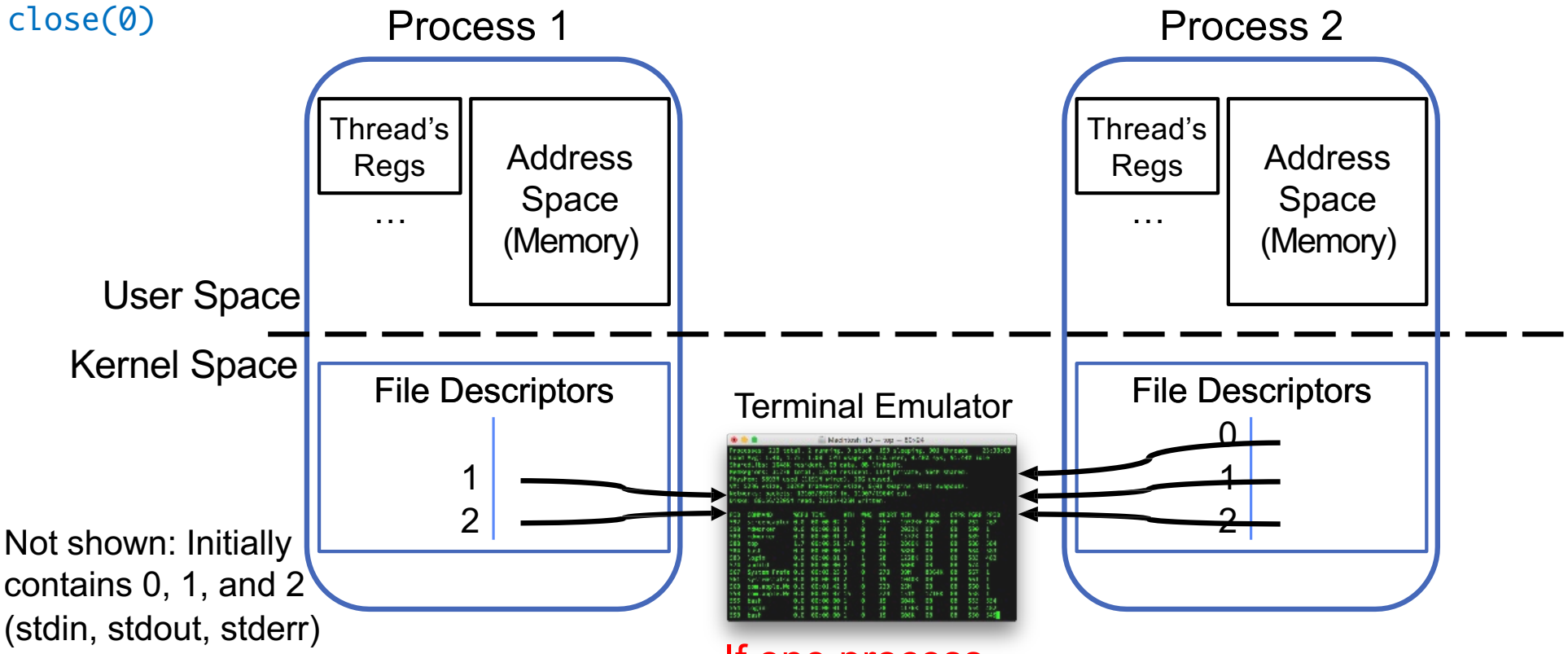
```
close(0)
```



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

Example: shared terminal emulator

```
close(0)
```



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

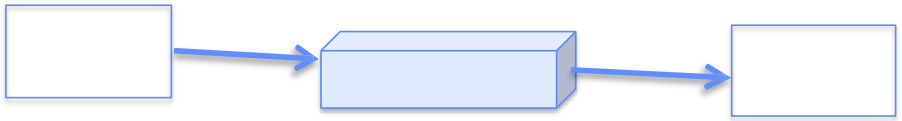
- If one process closes stdin (0), it remains open in other processes

Inter-process communication (IPC)

Communication between processes

- Can we view files as communication channels?

```
write(wfd, wbuf, wlen);
```

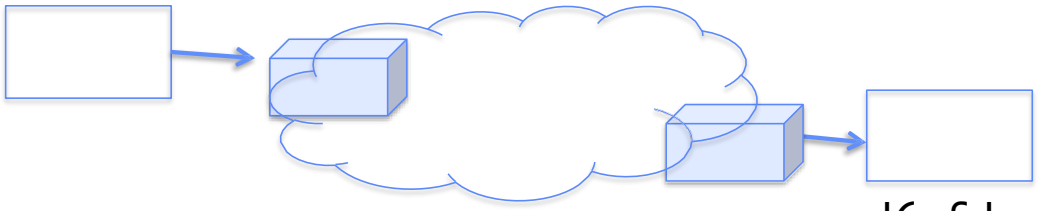


```
n = read(rfd, rbuf, rmax);
```

- Producer and Consumer of a file may be distinct processes
  - May be separated in time (or not)
- However, what if data written once and consumed once?
  - Don't we want something more like a queue?
  - Can still look like file I/O!

Communication across the world looks like file I/O too!

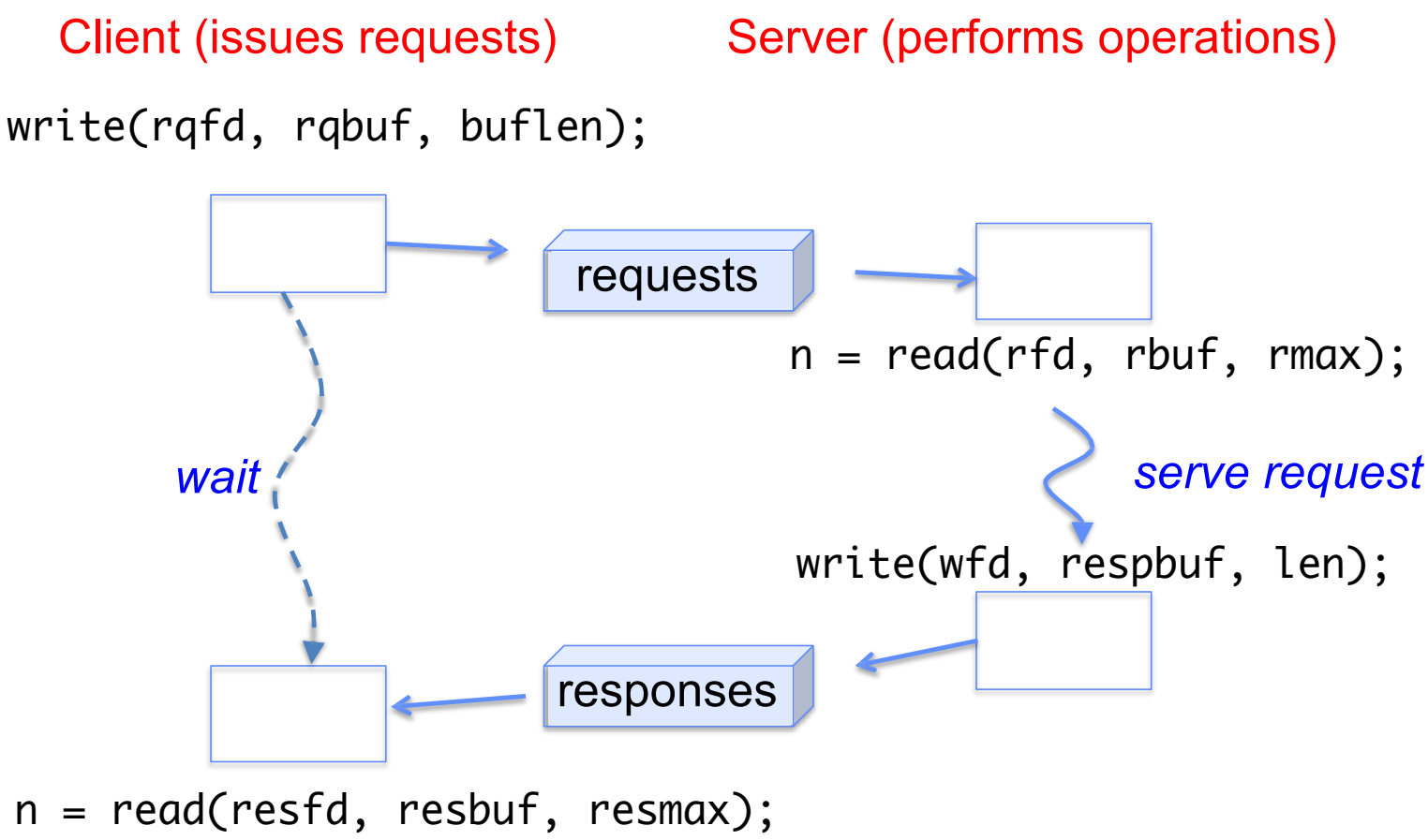
```
write(wfd, wbuf, wlen);
```



```
n = read(rfd, rbuf, rmax);
```

- Connected queues over the Internet
  - But what's the analog of open?
  - What is the namespace?
  - How are they connected in time?

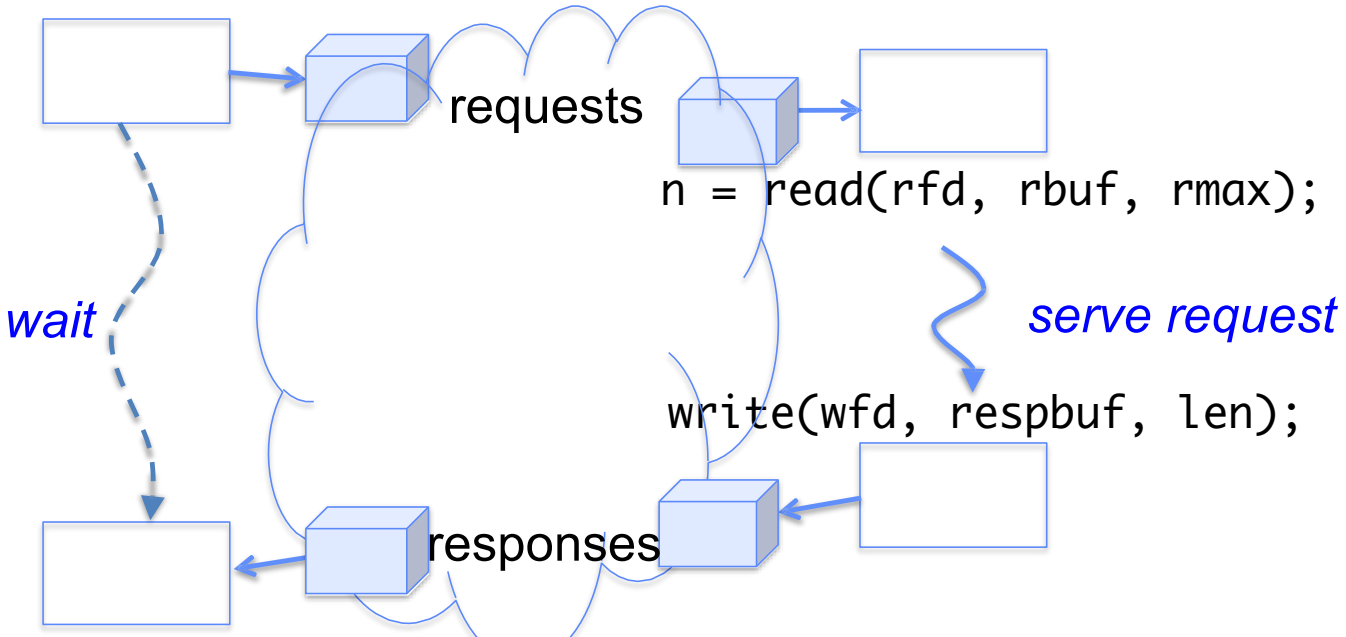
Request response protocol



Request response protocol: across network

Client (issues requests) Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```

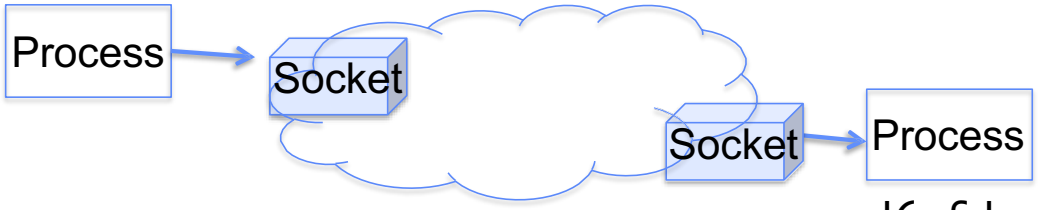



```
n = read(resfd, resbuf, resmax);
```

The socket abstraction: endpoint for communication

- Key idea: communication across the world looks like file I/O

```
write(wfd, wbuf, wlen);
```




```
n = read(rfd, rbuf, rmax);
```

- Sockets: endpoint for communication
  - Queues to temporarily hold results
- Connection: two sockets connected over the network → IPC over network!
  - How to open()?
  - What is the namespace?
  - How are they connected in time?

Sockets: more details

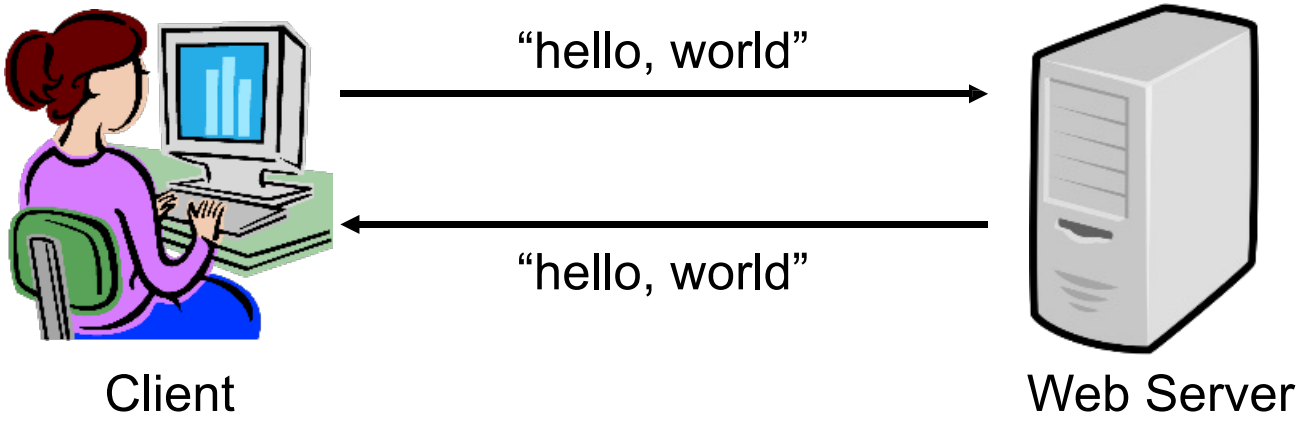
- Socket: an abstraction for one endpoint of a network connection
  - Another mechanism for inter-process communication
  - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
  - Standardized by POSIX
- First introduced in 4.2 BSD (Berkeley Standard Distribution) Unix
- Same abstraction for any kind of network
  - Local (within same machine)
  - The Internet (TCP/IP, UDP/IP)
  - Things "no one" uses anymore (OSI, Appletalk, IPX, …)

Sockets: more details

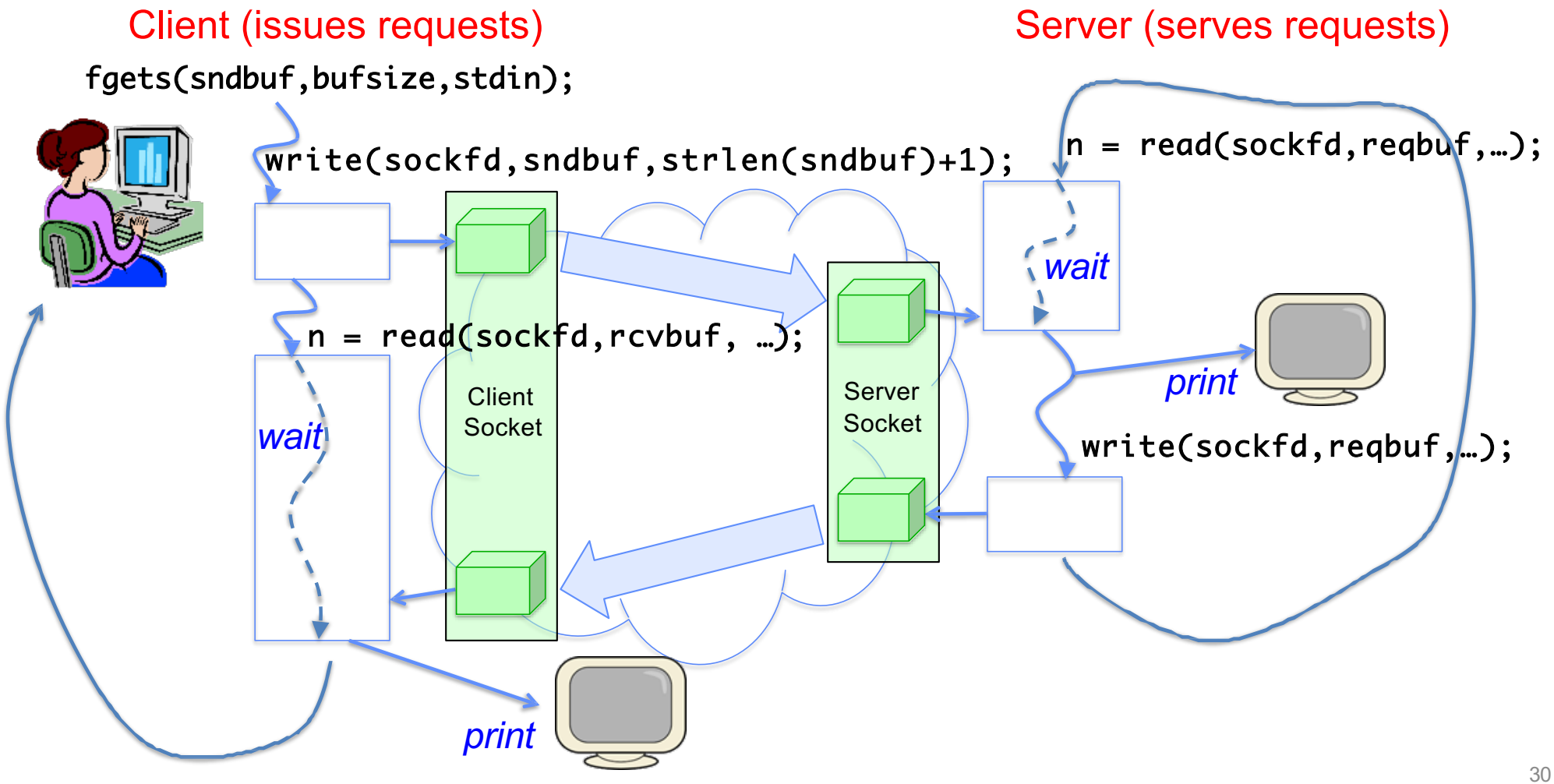
- Looks just like a file with a file descriptor
  - Corresponds to a network connection (two queues)
  - `write` adds to output queue (queue of data destined for other side)
  - `read` removes from it input queue (queue of data destined for this side)
  - Some operations do not work, e.g. `lseek`

- How can we use sockets to support real applications?
  - A bidirectional byte stream isn't useful on its own…
  - May need messaging facility to partition stream into chunks
  - May need RPC facility to translate one environment to another and provide the abstraction of a function call over the network

Simple example: echo server



Simple example: echo server



Echo client-server example

```
void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  while (1) {
    fgets(sndbuf,MAXIN,stdin);                  /* prompt */
    write(sockfd, sndbuf, strlen(sndbuf)+1); /* send (including null terminator) */
    memset(rcvbuf,0,MAXOUT);                    /* clear */
    n=read(sockfd, rcvbuf, MAXOUT);             /* receive */
    write(STDOUT_FILENO, rcvbuf, n);            /* echo */
}
```

```
void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf,0, MAXREQ);
    n = read(consockfd,reqbuf,MAXREQ); /* recv */
    if (n <= 0) return;
    write(STDOUT_FILENO, reqbuf, n);
    write(consockfd, reqbuf, n);        /* echo*/
  }
}
```

What assumptions are we making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.

- In order (sequential stream)
  - Write X then write Y => read gets X then read gets Y

- When ready?
  - File read gets whatever is there at the time
    - Actually, need to loop and read until we receive the terminator (‘\0’)
  - Assumes writing already took place
  - Blocks if nothing has arrived yet

Socket creation

- File systems provide a collection of permanent objects in a structured name space:
  - Processes open, read/write/close them
  - Files exist independently of processes
  - Easy to name what file to open()
- Pipes: one-way communication between processes on same (physical) machine (see later)
  - Single queue
  - Created transiently by a call to pipe()
  - Passed from parent to children (descriptors inherited from parent process)

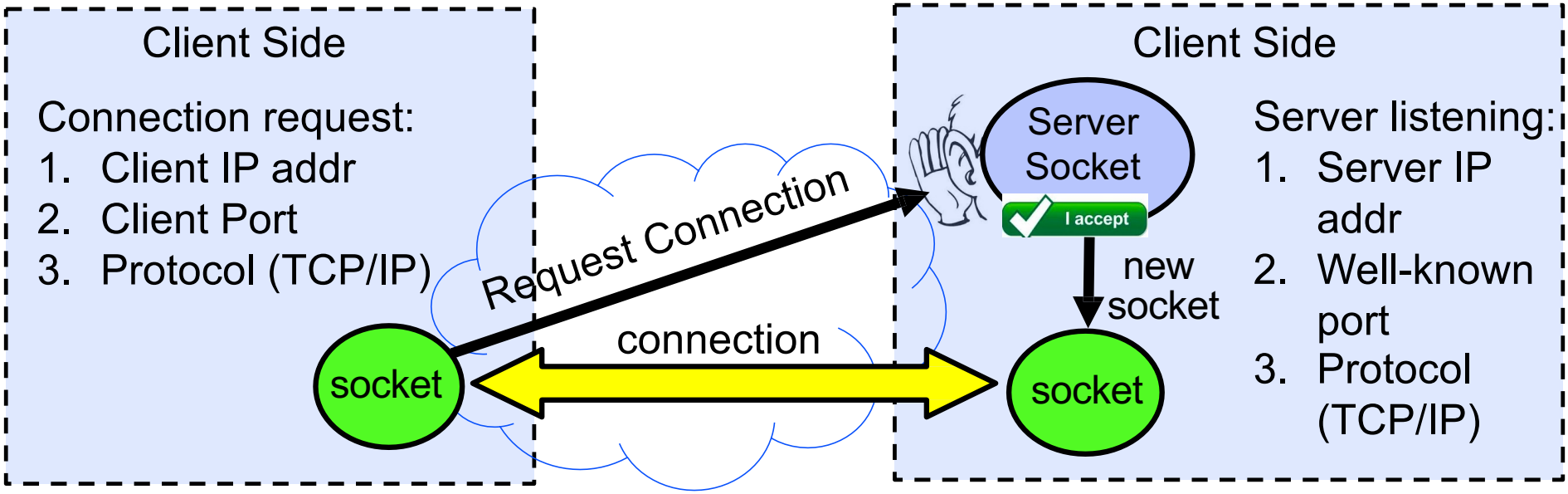
Socket creation

- Sockets: two-way communication between processes on same or different machine
  - Two queues (one in each direction)
  - Processes can be on separate machines: no common ancestor
  - How do we name the objects we are **opening**?
  - How do these completely independent programs know that the other wants to “talk” to them?

Namespaces for communication over IP

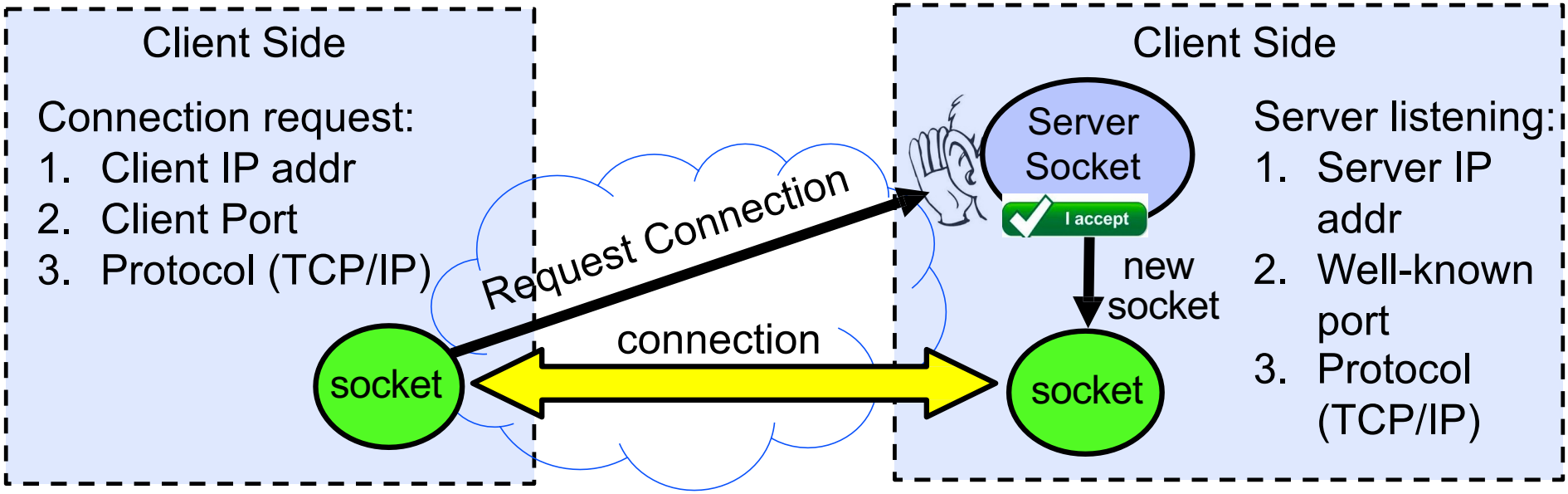
- Hostname
  - yunmingxiao.github.io
- IP address
  - 185.199.111.153 (IPv4, 32-bit Integer)
  - 2606:50c0:8000::153 (IPv6, 128-bit Integer)
- Port Number
  - 0-1023 are “well known” or “system” ports
    - E.g., 80 (http), 443 (secure web)
    - Superuser privileges to bind to one
  - 1024 – 49151 are “registered” ports (registry)
    - Assigned by IANA for specific services
  - 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are “dynamic” or “private”
    - Automatically allocated as “ephemeral ports”

Connection setup over TCP/IP



- Special kind of socket: server socket
  - Has file descriptor
  - Can't read or write
- Two operations:
  - `listen()`: Start allowing clients to connect
  - `accept()`: Create a new socket for a particular client

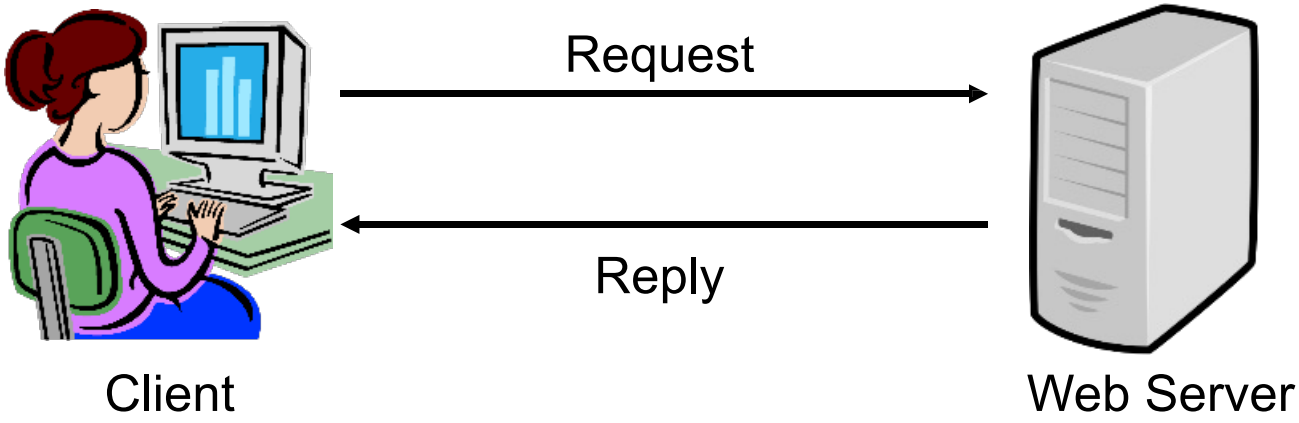
Connection setup over TCP/IP



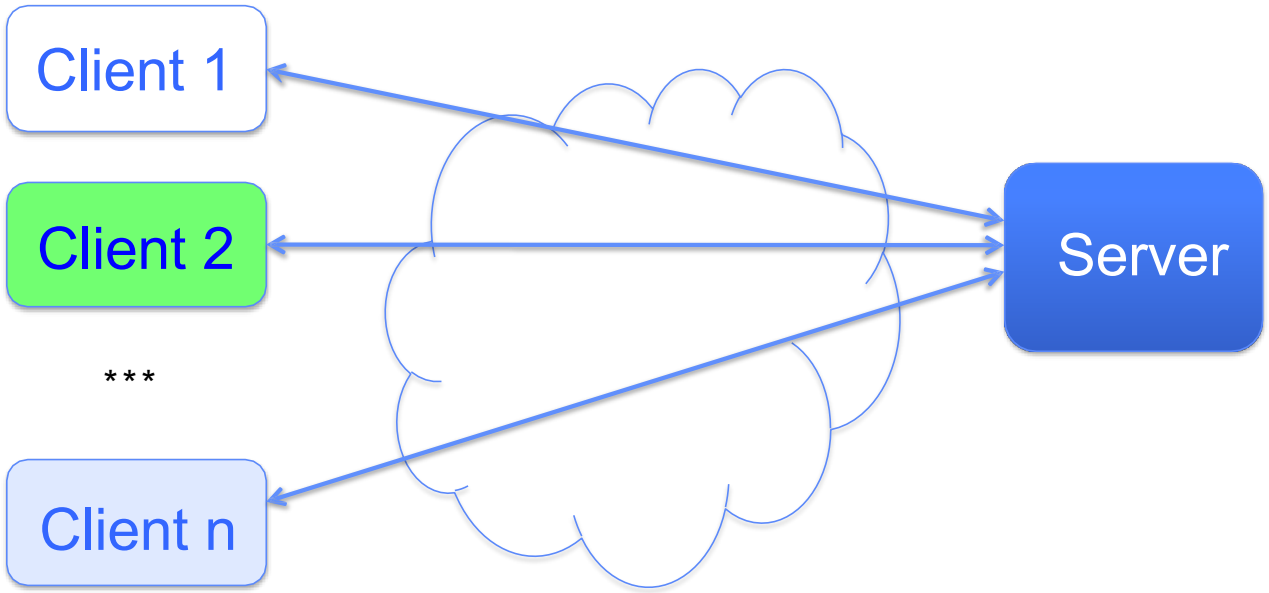
- 5-tuple identifies each connection:
  1. Source IP address
  2. Destination IP address
  3. Source port number
  4. Destination port number
  5. Protocols (always TCP here)

- Often, client port "randomly" assigned
  - Done by OS during client socket setup
- Server port often "well known"
  - 80 (web), 443 (secure web), 25 (sendmail), etc
  - Well-known ports from 0 to 1023

Web server

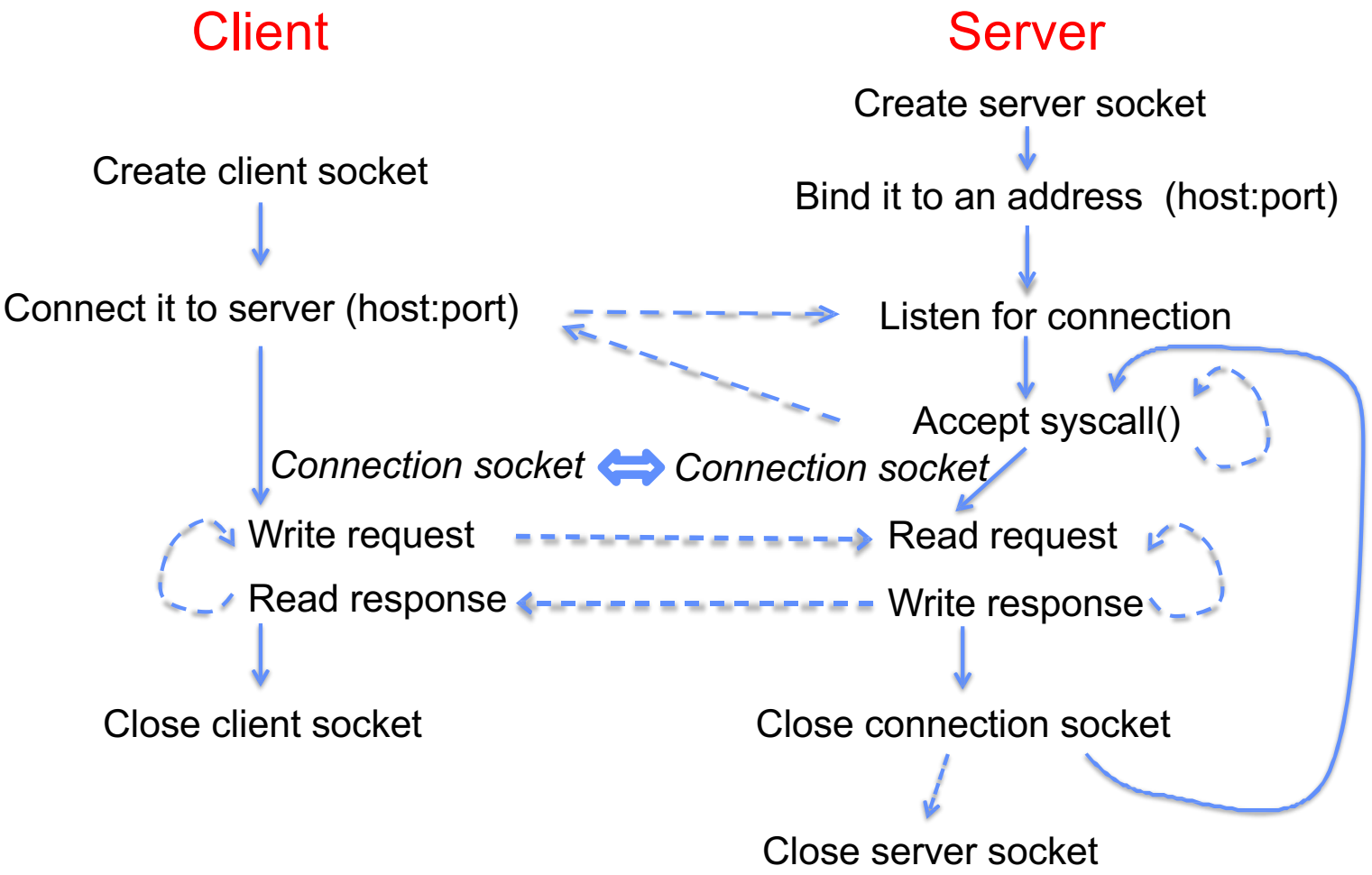


Client-server models



- File servers, web, FTP, Databases, …
- Many clients accessing a common server

Simple web server



Client code

```
char *host_name, *port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                     server->ai_protocol);

// Connect to specified host and port
connect(socket_fd, server->ai_addr, server->ai_addrlen);

// Carry out client-server protocol
run_client(sock_fd);

// Clean up on termination
close(sock_fd);
```

Client-side: getting the server address

```
struct addrinfo *lookup_host(char *host_name, char *port_name) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;          /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;      /* Essentially TCP/IP */

  int rv = getaddrinfo(host_name, port_name, &hints, &server);
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

Server code (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family, server->ai_socktype,
                           server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}
close(server_socket);
```

Server address: itself (wildcard IP), passive

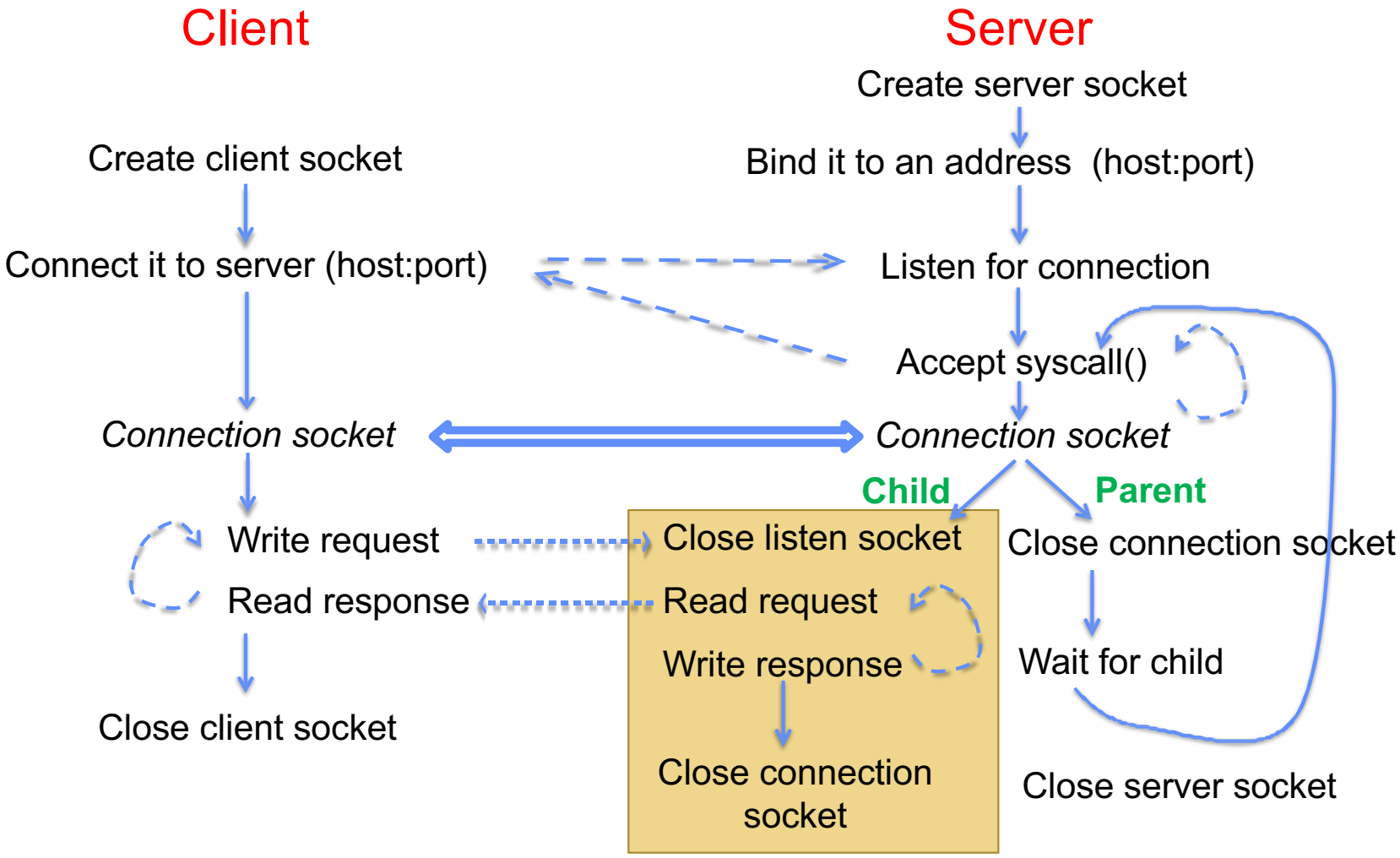
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC; /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM; /* Essentially TCP/IP */
  hints.ai_flags = AI_PASSIVE; /* Set up for server socket */
  int rv = getaddrinfo(NULL, port, &hints, &server); /* No address! */
  if (rv != 0) {
    printf("getaddrinfo failed: %s\n", gai_strerror(rv));
    return NULL;
  }
  return server;
}
```

- Accepts any connections on the specified port

How could the server protect itself?

- Handle each connection in a separate process
  - This will mean that the logic serving each request will be “sandboxed” away from the main server process
- In the following code, keep in mind:
  - `fork()` will duplicate all of the parent’s file descriptors (i.e. pointers to sockets!)
  - We keep control over accepting new connections in the parent
  - New child connection for each remote client

Server with protection (each process has own process)



Server code (v2)

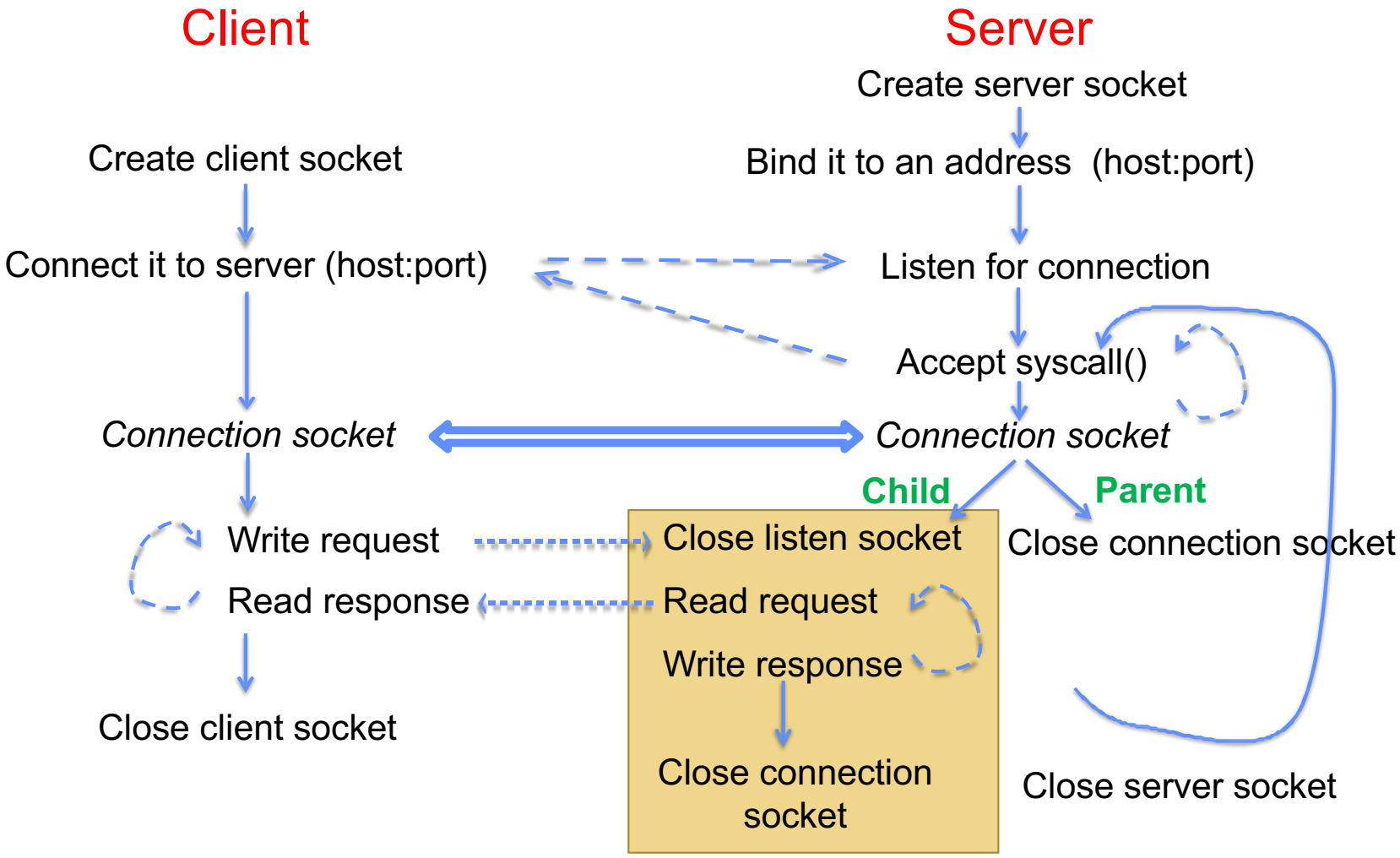
```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    wait(NULL);
  }
}
close(server_socket);
```

How to make a concurrent server

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server *waits* for each connection to terminate before serving the next
    - This is the standard shell pattern

- A concurrent server can handle and service a new connection before the previous client disconnects
  - Simple – just don't wait in parent!
  - Perhaps not so simple – multiple child processes better not have data races with one another through file system/etc!

Server with protection (each process has own process)



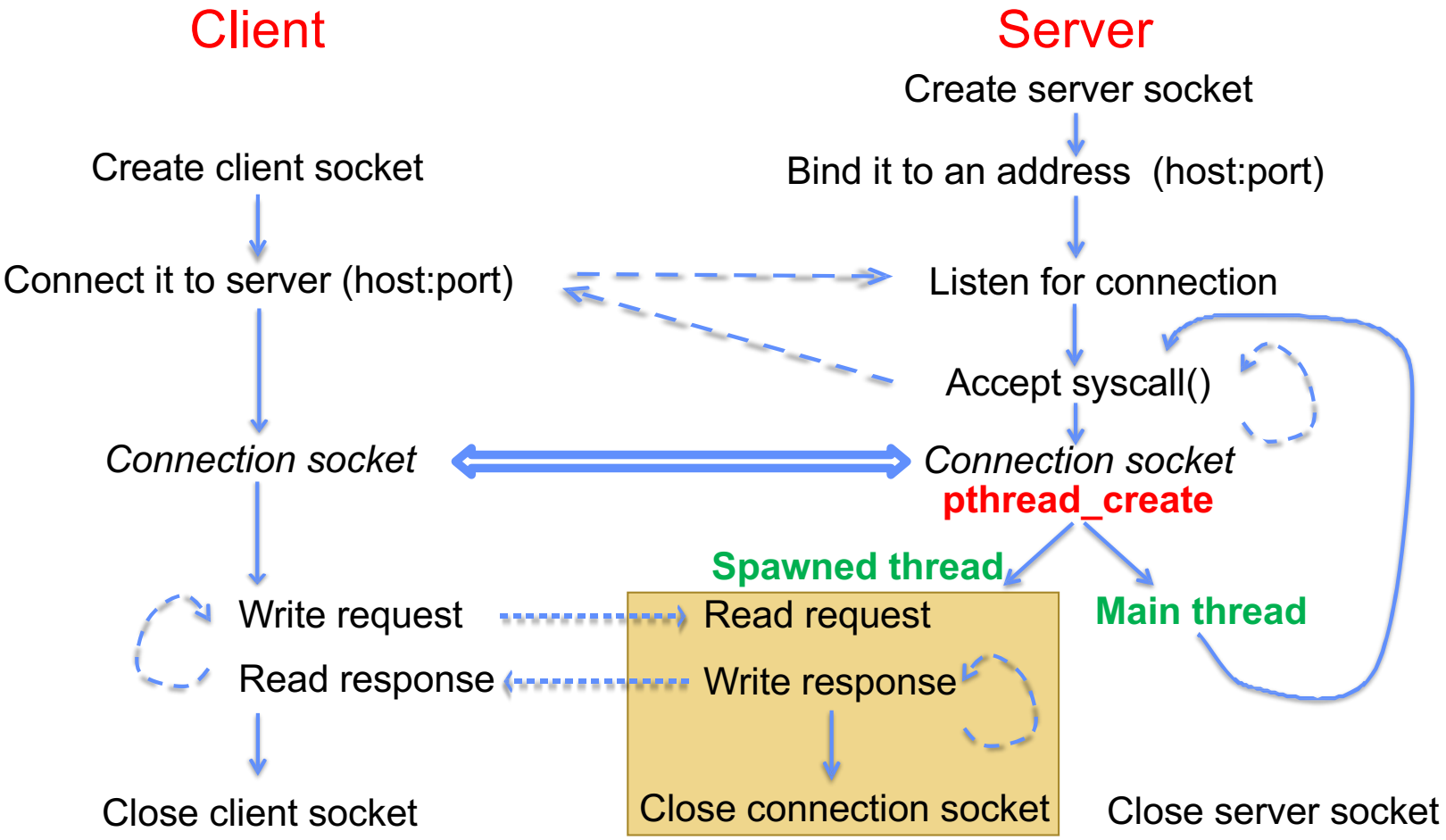
Server code (v3)

```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {
    close(conn_socket);
    // wait(NULL);
  }
}
close(server_socket);
```

Faster concurrent server (without protection)

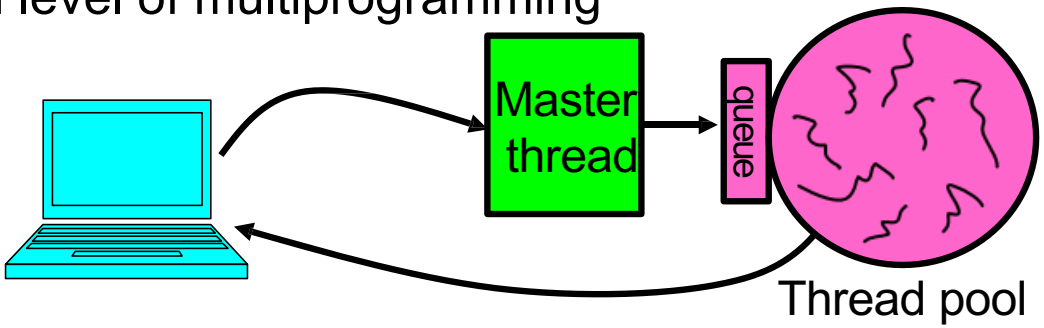
- Spawn a new *thread* to handle each connection
  - Lower overhead spawning process (less to do)
- Main *thread* initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads
- Even more potential for data races (need synchronization?)
  - Through shared memory structures
  - Through file system

Server with concurrency, without protection



Thread pools: more later!

- Problem with previous version: unbounded threads
  - When web-site becomes too popular – throughput sinks
  - Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
  allocThreads(worker, queue);
  while (1) {
    conn = AcceptCon();
    Enqueue(queue, conn);
    wakeup(queue);
  }
}
```

```
worker(queue) {
  while (1) {
    conn = Dequeue(queue);
    if (conn == NULL)
      sleepOn(queue);
    else
       ServeWebPage(conn);
  }
}
```

Now let's talk about pipe

- A different IPC mechanism other than socket

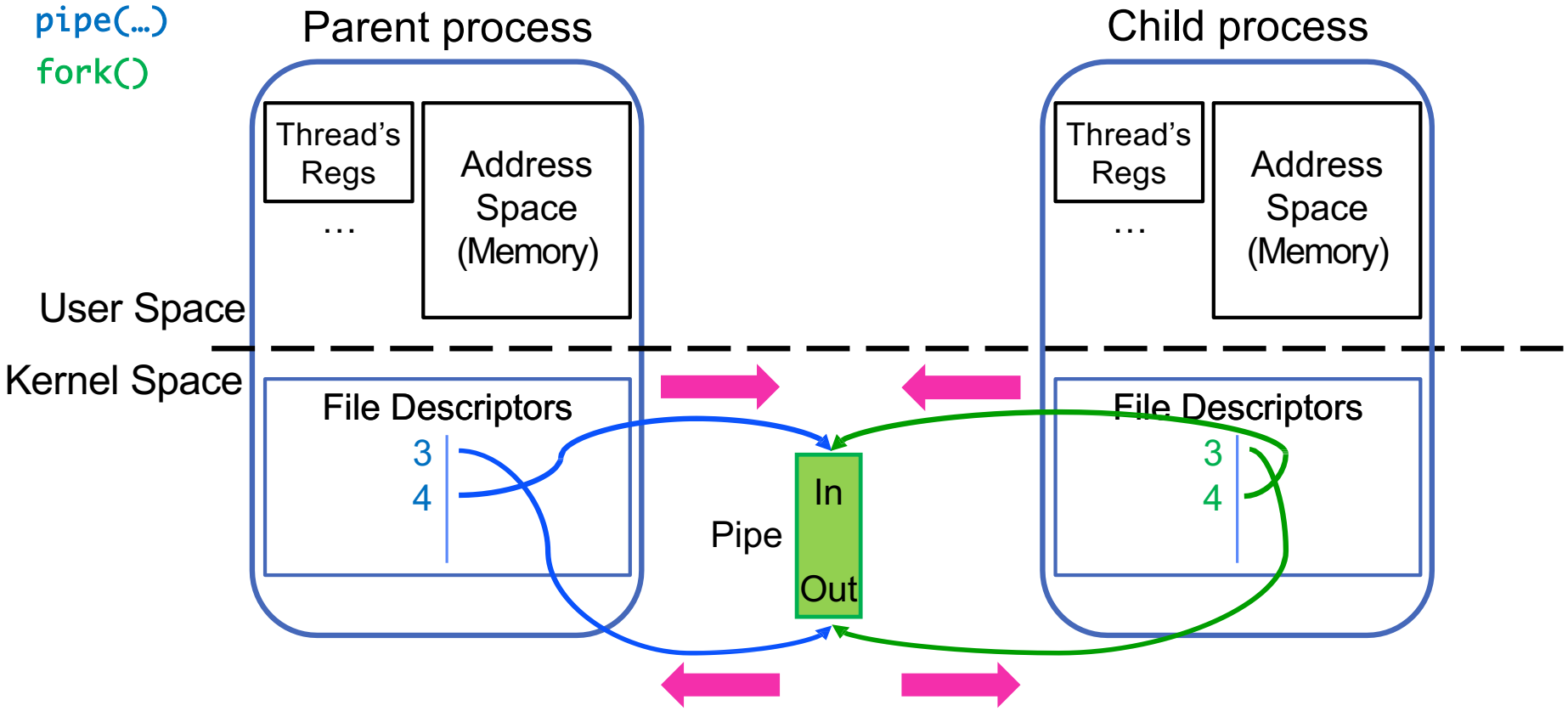
Single-process pipe example (not that interesting yet…)

```
#include <unistd.h>

int main(int argc, char *argv[]) {
   char *msg = "Message in a pipe.\n";
   char buf[BUFSIZE];

   int pipe_fd[2];
   if (pipe(pipe_fd) == -1) {
       fprintf (stderr, "Pipe failed.\n");
       return EXIT_FAILURE;
   }
   ssize_t writelen = write(pipe_fd[1], msg, strlen(msg)+1);
   printf("Sent: %s [%ld, %ld]\n", msg, strlen(msg)+1, writelen);

   ssize_t readlen = read(pipe_fd[0], buf, BUFSIZE);
   printf("Rcvd: %s [%ld]\n", msg, readlen);

   close(pipe_fd[0]);
   close(pipe_fd[1]);
}
```

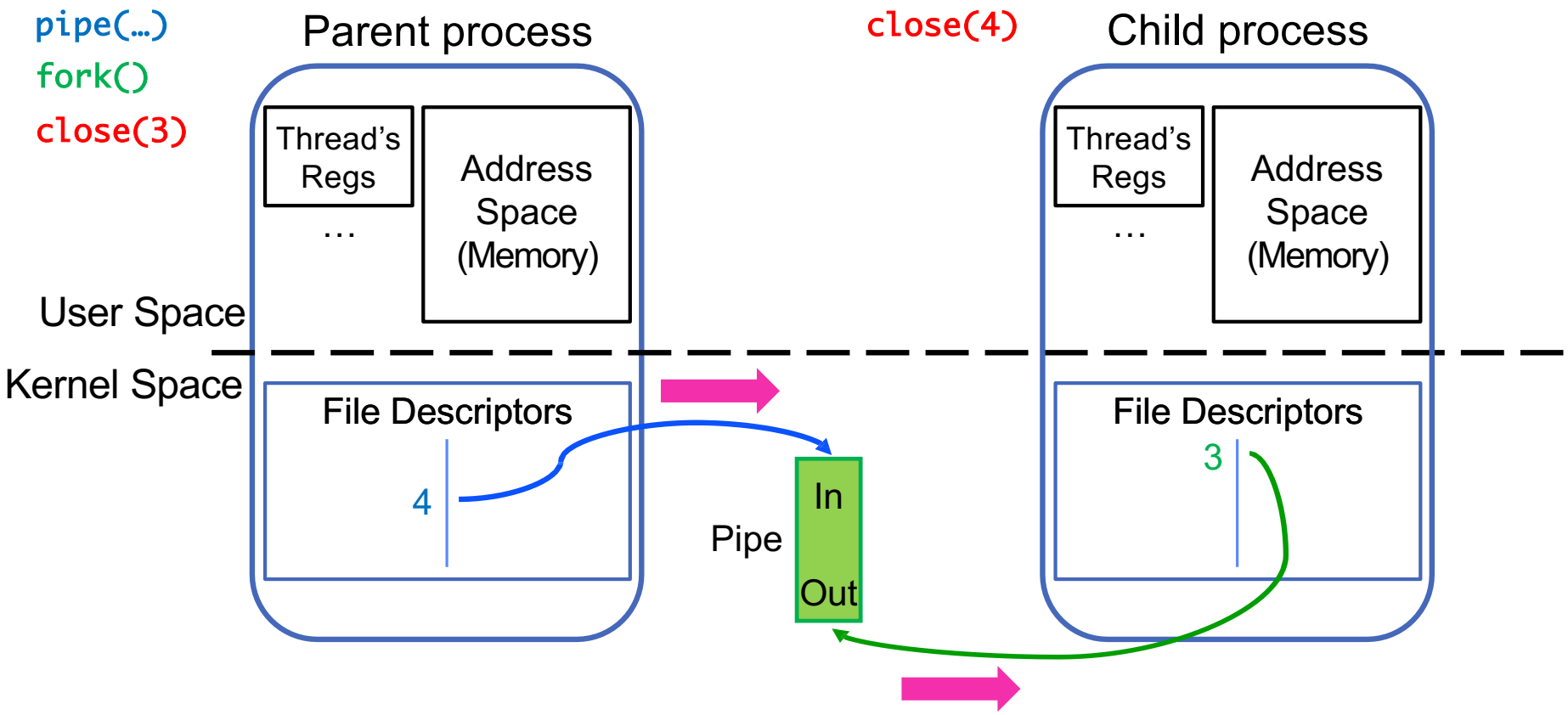
Could be useful for multithreaded processes…

- Write to pipe_fd[1]
- Read from pipe_fd[0]

Example: pipes between processes



Example: channel from parent to child



Inter-process communication (IPC): parent → child

```
// continuing from earlier
pid_t pid = fork();
if (pid < 0) {
  fprintf (stderr, "Fork failed.\n");
  return EXIT_FAILURE;
}
if (pid != 0) {
  close(pipe_fd[0]); // Not using this descriptor!
  ssize_t writelen = write(pipe_fd[1], msg, msglen);
  printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
} else {
  close(pipe_fd[1]); // Not using this descriptor!
  ssize_t readlen = read(pipe_fd[0],buf, BUFSIZE);
  printf("Child Rcvd: %s [%ld]\n", msg, readlen);
}
```

Discussion

- Why pipe over socket?
  - Socket is more general and heavier
    - Connection setup
    - Addressing (IP, port, etc)
    - Protocol machinery (e.g., TCP congestion control)
  - Socket is not fork-native
    - Parent/child must explicitly connect
- Why pipe over local files?
  - Files require a pathname (global namespace)
  - Files are persistent, pipes are ephemeral
  - Files require explicit synchronization
    - File offsets, locking, races

- Essentially, pipe is fork-native

Takeaway

- UNIX provides multiple communication mechanisms

Scope	Mechanism
Same thread	Function calls
Across threads	Shared memory
Parent-child processes	Pipe
Same machine, unrelated processes	UNIX domain socket / files (named shared memory)
Across machines	Socket

Conclusion

- Recall: Everything is a file!
  - `open()`, `read()`, `write()`, and `close()` used for wide variety of I/O:
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local inter-process communication (pipes, sockets)
- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Various textbooks talk about *processes*
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process
- Stack is essential part of computation
  - Every thread has two stacks: user-level (in address space) and kernel
  - The kernel stack + support often called the “kernel thread”

Please help to make this class better!

https://forms.office.com/r/UcqDW4fpdq

Thanks