Abstractions 3: Pipes and Sockets

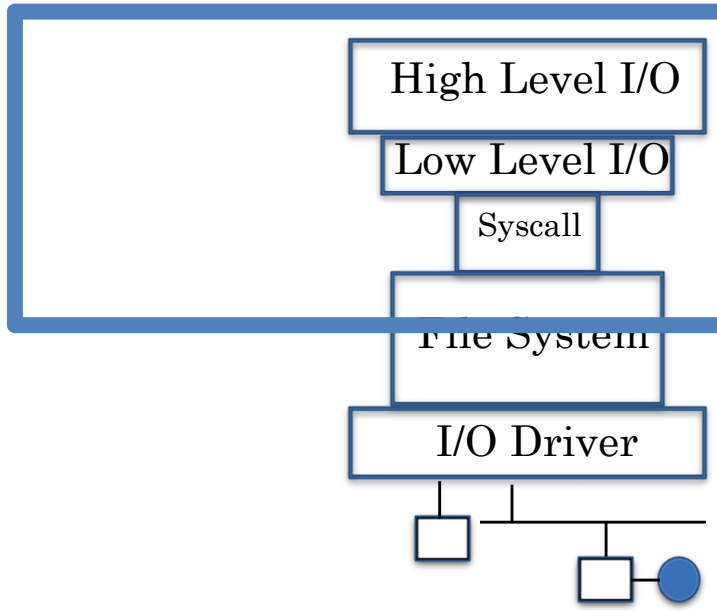
Lecture 5

Hartmut Kaiser

https://teaching.hkaiser.org/spring2026/csc4103/

Recall: I/O and Storage Layers

Application / Service



Streams

File Descriptors

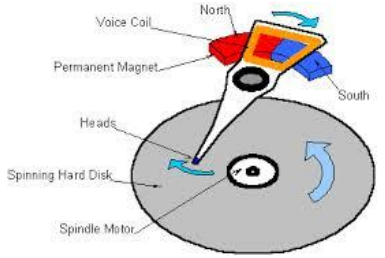
open(), read(), write(), close(), ...

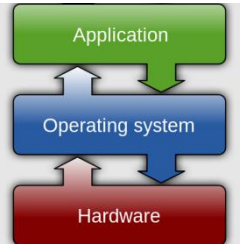
Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

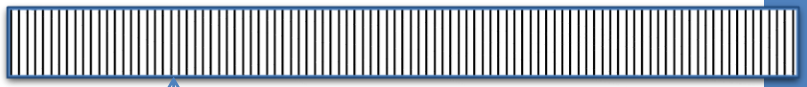
Disks, Flash, Controllers, DMA





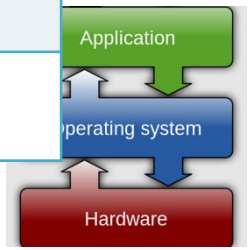
C High-Level File API – Streams

- Operates on "streams" – sequence of bytes, either text or data, with a position



```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append



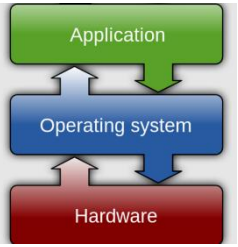
Low-Level File I/O

- Operations on file descriptors
  - Integer that corresponds to an object in the kernel called an open file description
  - Open file description object in the kernel represents an instance of an open file
  - Why not just use a pointer?

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

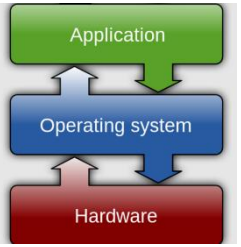
Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User | Group | Other x R | W | X

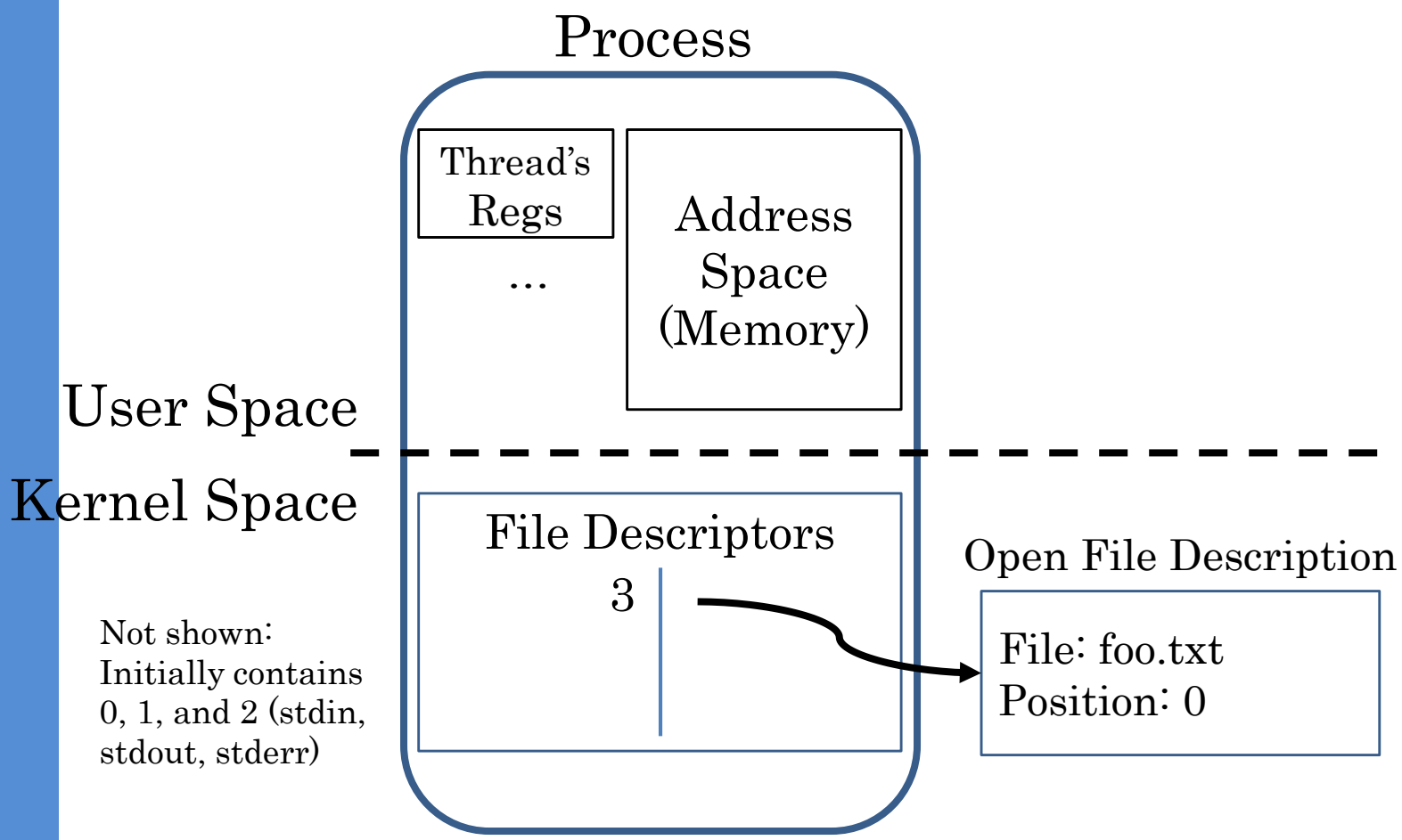


Recall: Key Unix I/O Design Concepts

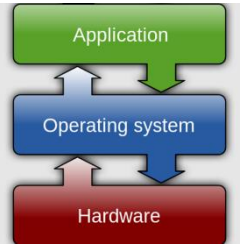
- Uniformity – everything is a file
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - `find | grep | wc ...`
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same, read blocks yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close



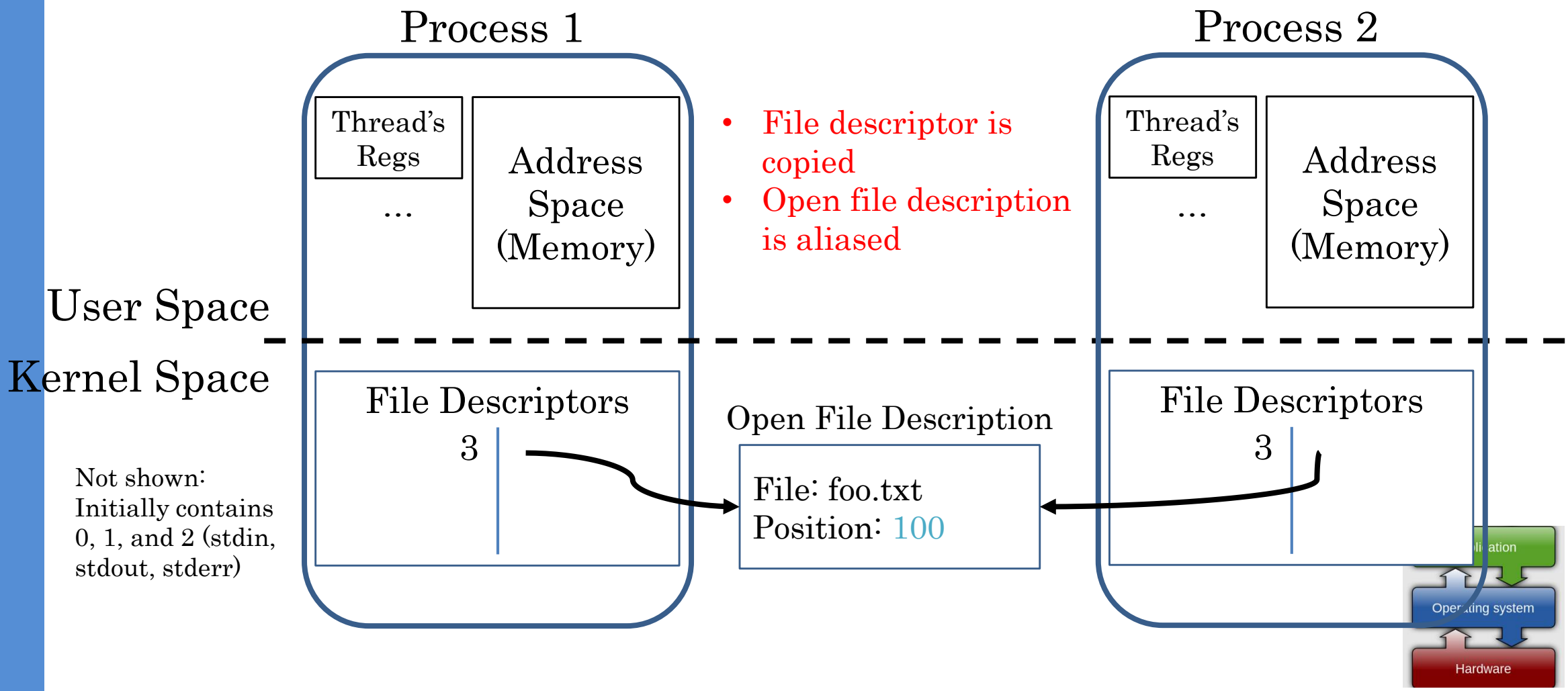
Recall: Abstract Representation of a Process



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

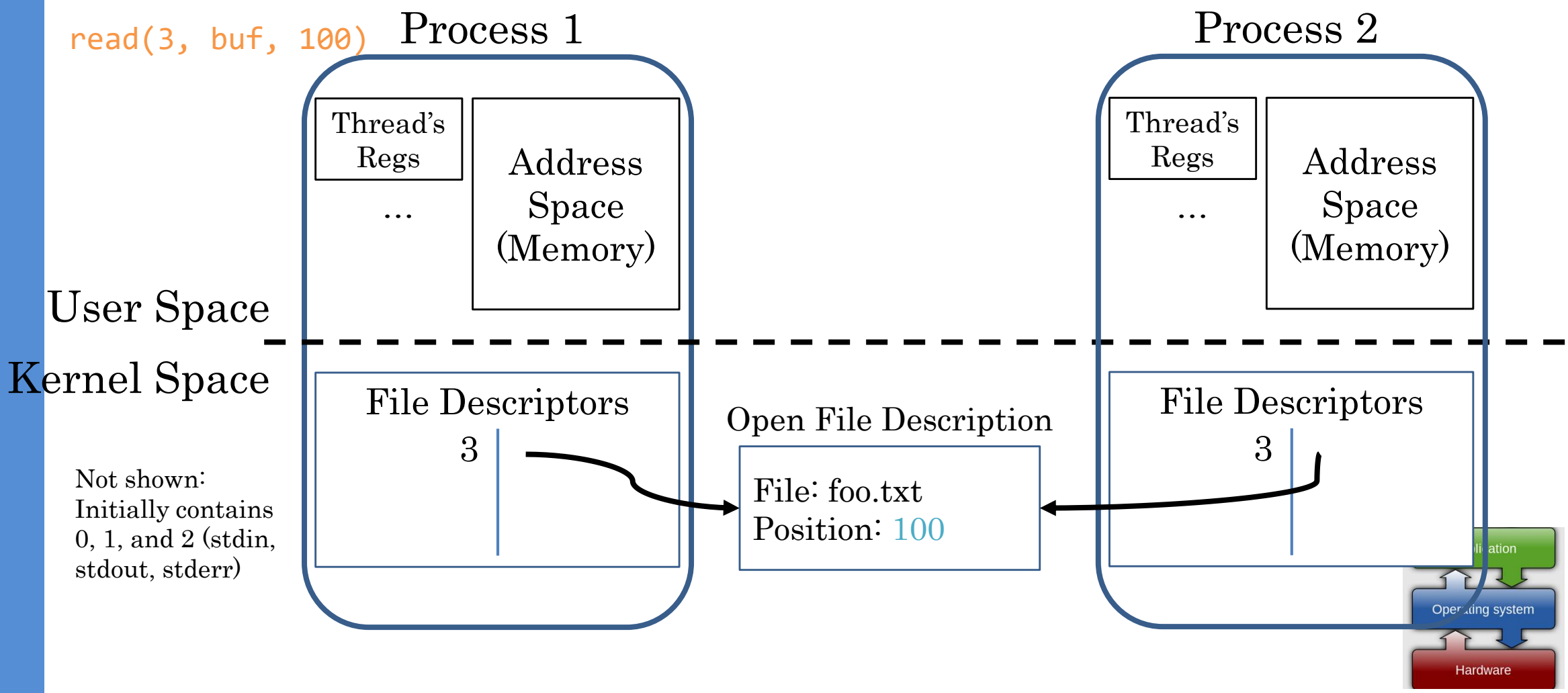


Recall: What happens on fork()?

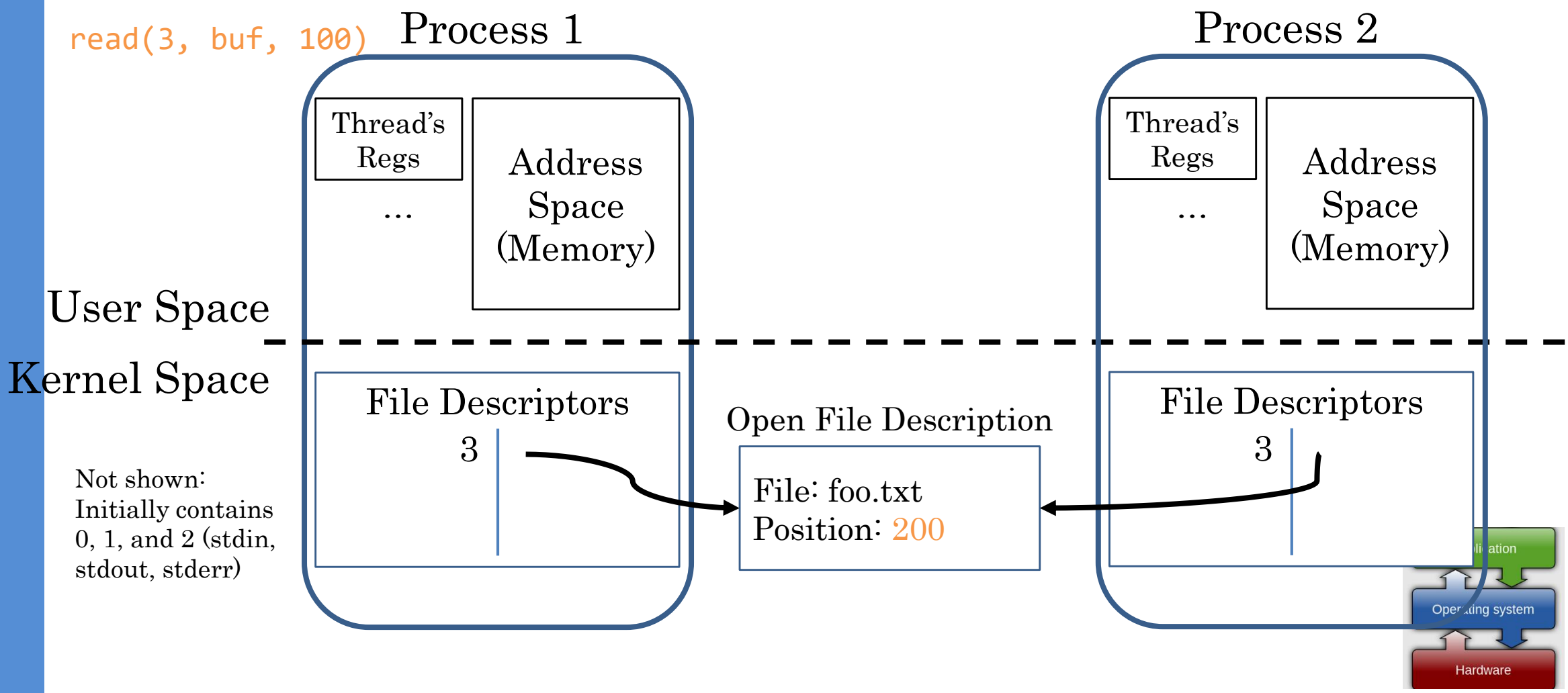


- File descriptor is copied
- Open file description is aliased

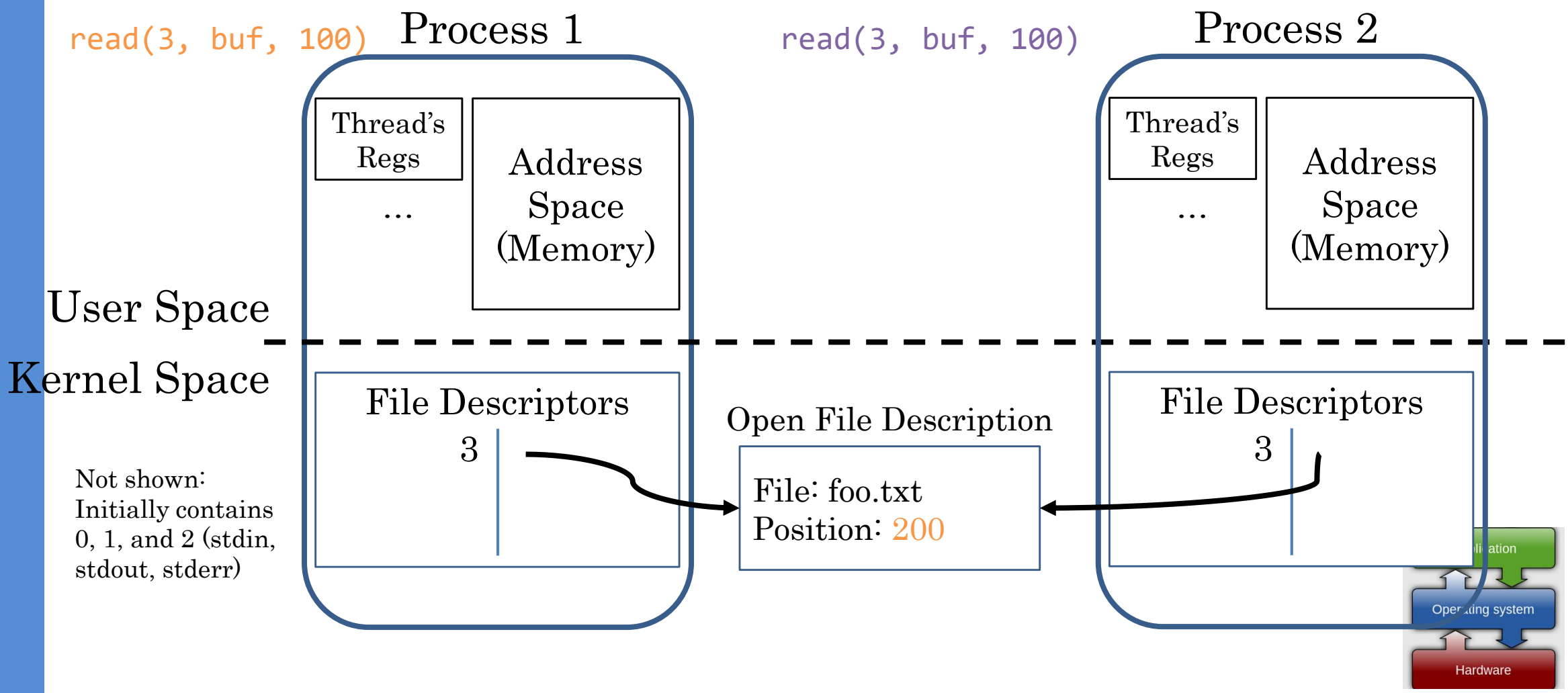
Recall: Open File Description is Aliased



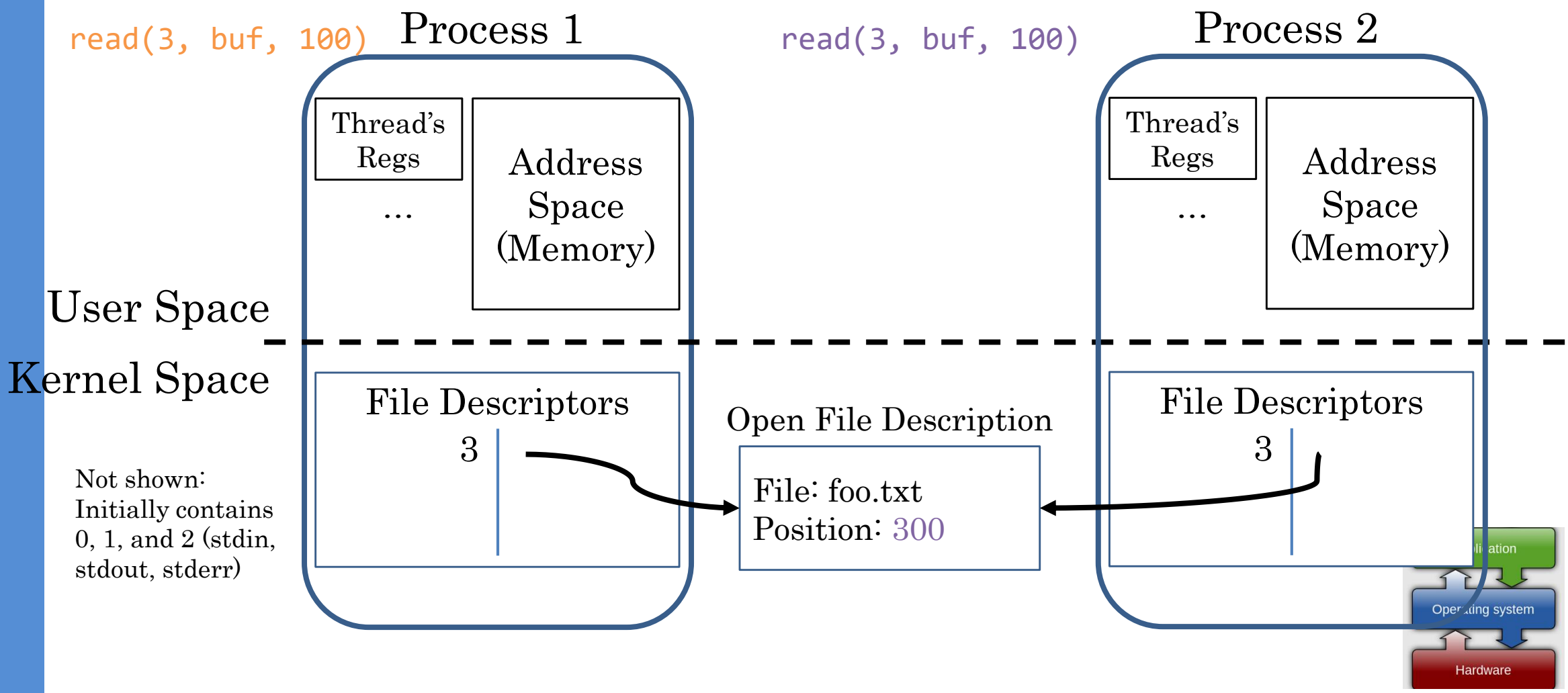
Recall: Open File Description is Aliased



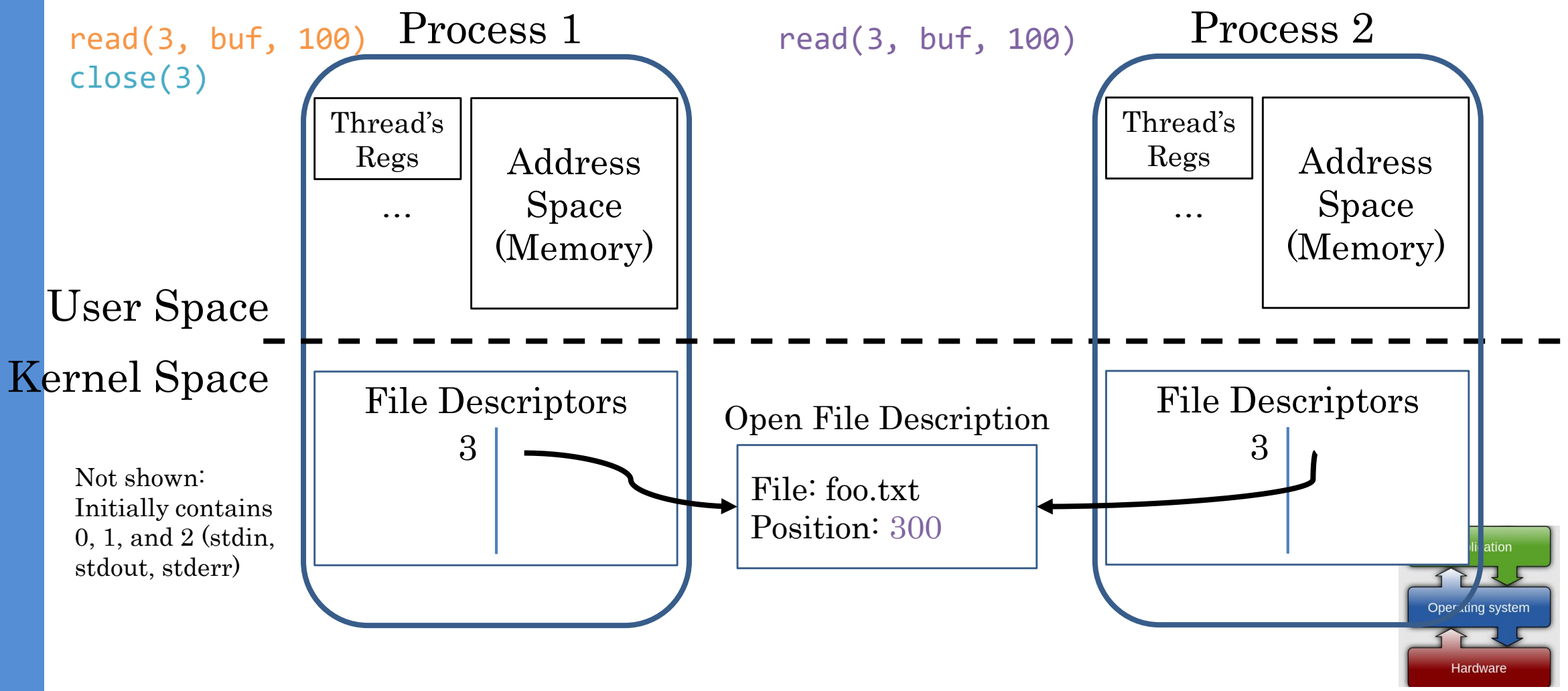
Recall: Open File Description is Aliased



Open File Description is Aliased



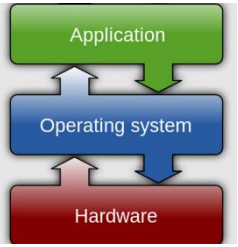
Recall: File Descriptor is Copied



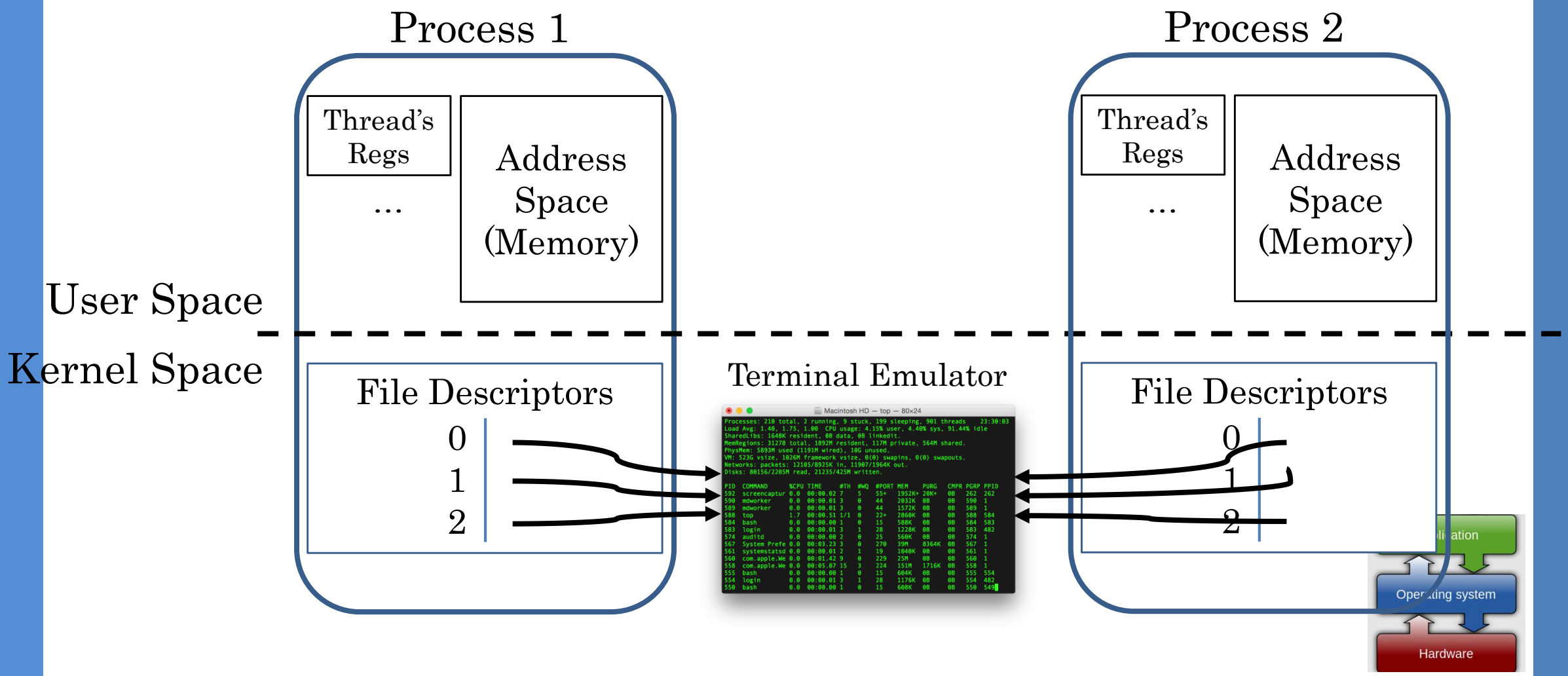
Recall: In POSIX, Everything is a "File"

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)

- Based on the system calls open(), read(), write(), and close()

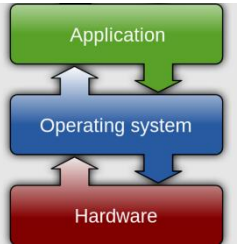


Recall: Shared Terminal Emulator

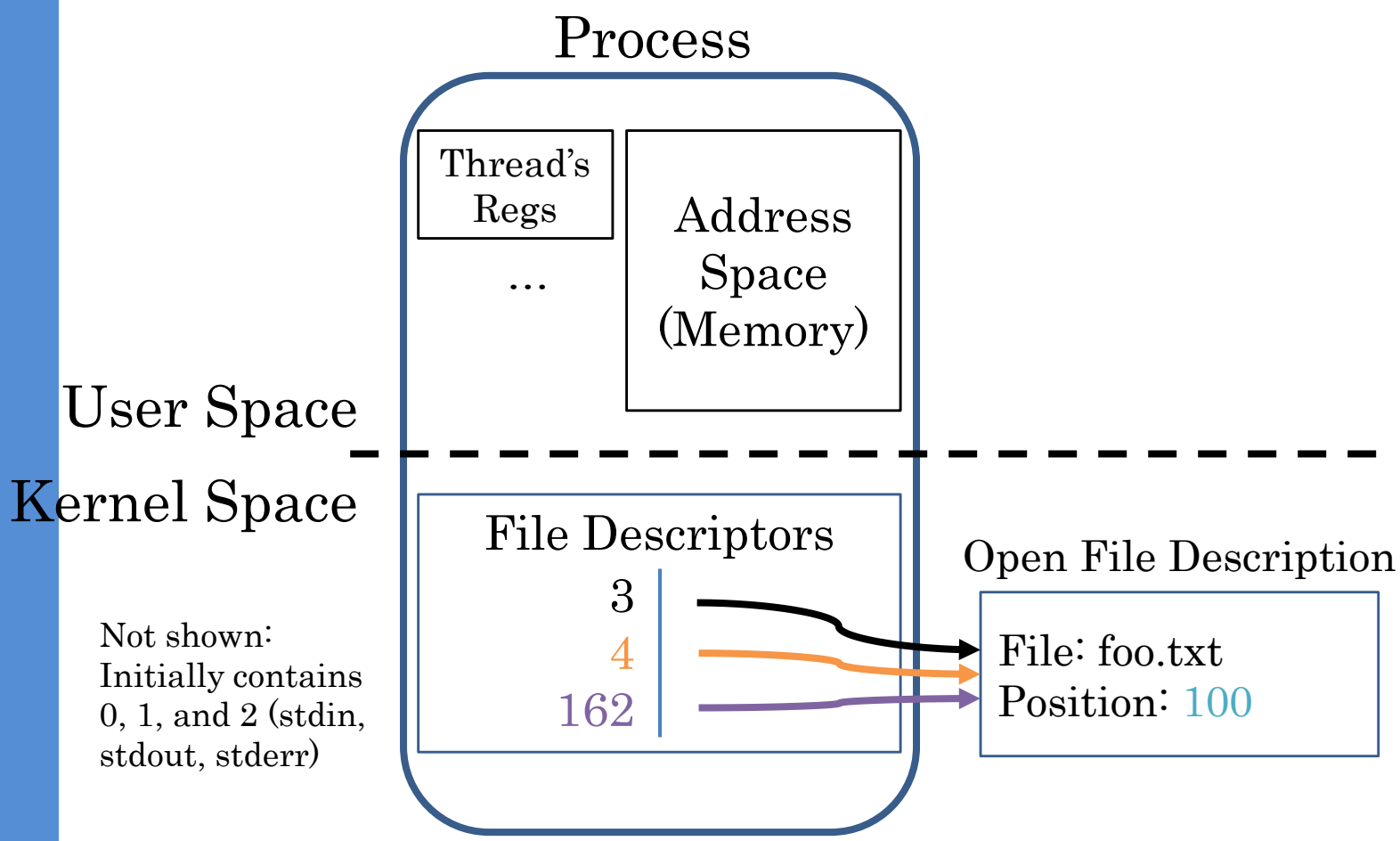


Other Syscalls: dup and dup2

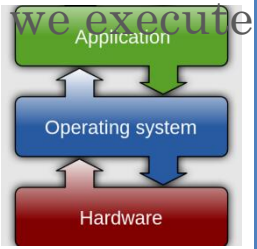
- They allow you to duplicate the file descriptor
- But the open file description remains aliased



Other Syscalls: dup and dup2

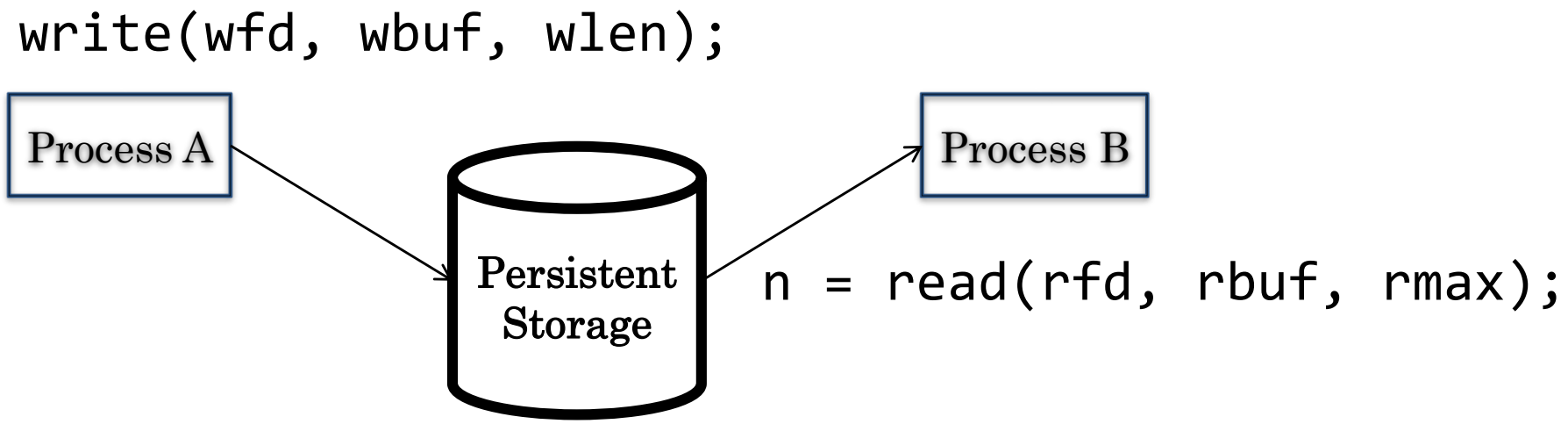


- Suppose that we execute `open("foo.txt")`
- and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)`
- and that the result is 100
- Next, suppose that we execute `dup(3)`
- And that the result is 4
- Finally, suppose that we execute `dup2(3, 162)`

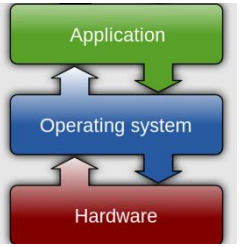


Pipes

Communication Between Processes

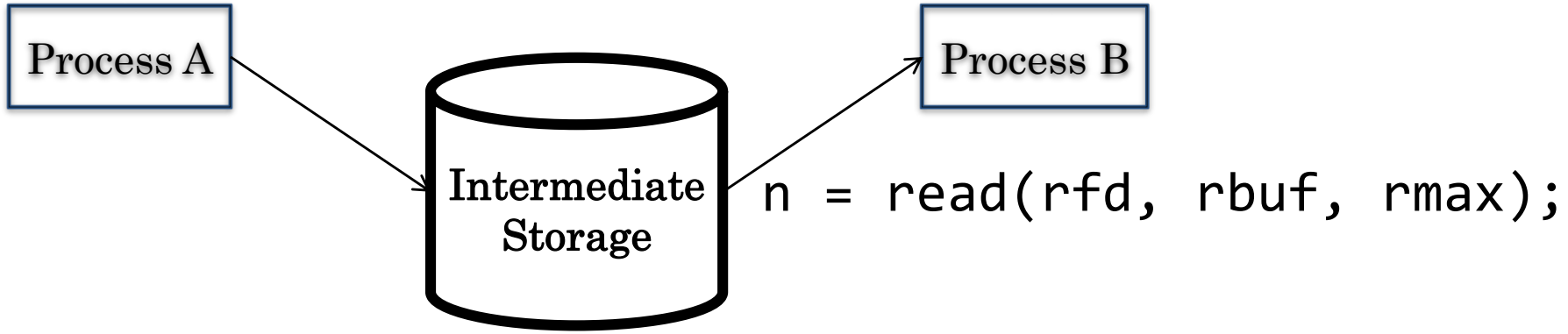


- Producer (writer) and consumer (reader) may be distinct processes
- Potentially separated in time
- Why might it be wasteful to use a file in this way?

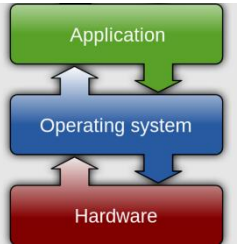


Communication Between Processes

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

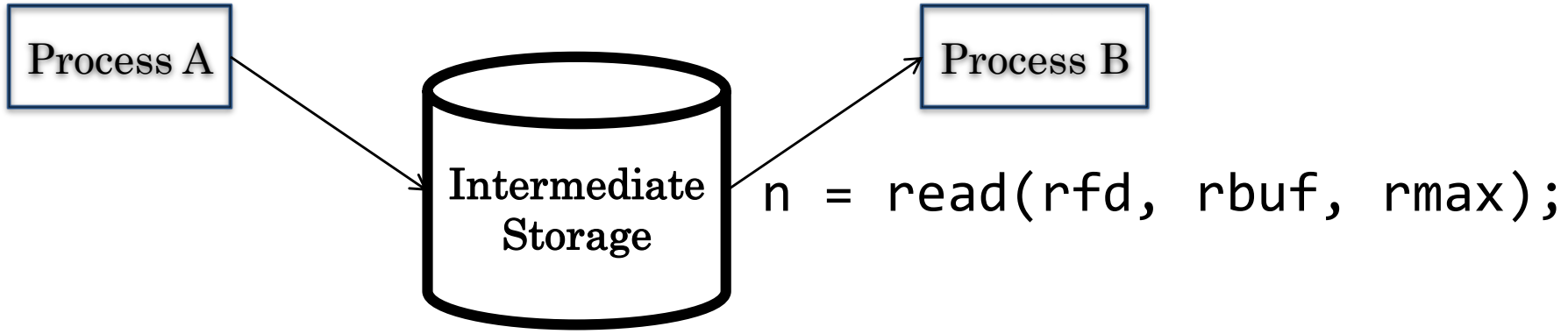


- Data written by A is held in memory until B reads it
- What if A generates data faster than B can process it?

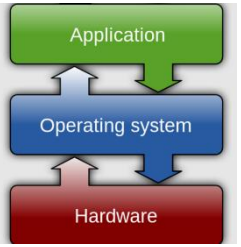


Communication Between Processes

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

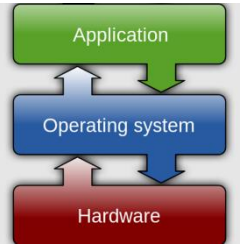


- Data written by A is held in memory until B reads it
- Queue has a fixed capacity
  - Writing to the queue blocks if the queue if full
  - Reading from the queue blocks if the queue is empty
- POSIX provides this abstraction in the form of pipes



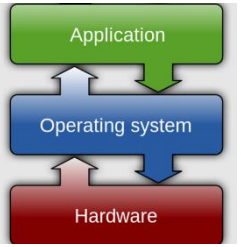
Pipes

- `int pipe(int fileds[2]);`
  - Allocates two new file descriptors in the process
  - Writes to `fileds[1]` read from `fileds[0]`
  - Implemented as a fixed-size queue



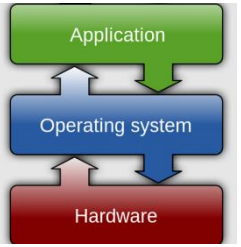
Single-Process Pipe Example

```
#include <unistd.h>
int main(int argc, char *argv[]) {
  char *msg = "Message in a pipe.\n";
  char buf[BUFSIZE] = { '\0' };
  int pipe_fd[2];
  if (pipe(pipe_fd) == -1) {
    fprintf (stderr, "Pipe creation 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", buf, readlen);

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



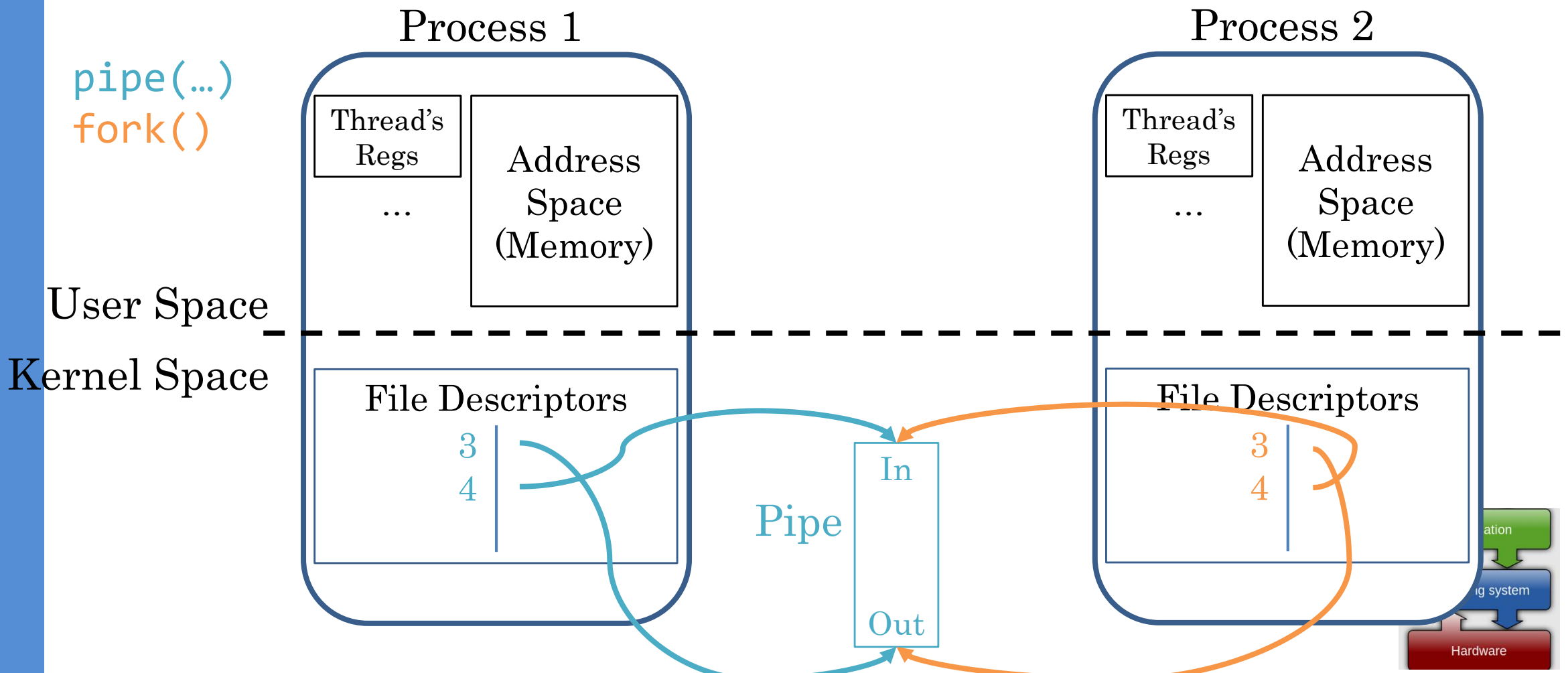
Inter-Process Communication (IPC)

```
pid_t pid = fork();
if (pid < 0) {
  fprintf (stderr, "Fork failed.\n");
  return EXIT_FAILURE;
}
if (pid != 0) {
  ssize_t writelen = write(pipe_fd[1], msg, msglen);
  printf("Parent: %s [%ld, %ld]\n", msg, msglen, writelen);
  close(pipe_fd[0]);
  close(pipe_fd[1]);
} else {
  ssize_t readlen  = read(pipe_fd[0], buf, BUFSIZE);
  printf("Child Rcvd: %s [%ld]\n", msg, readlen);
  close(pipe_fd[0]);
  close(pipe_fd[1]);
}
```

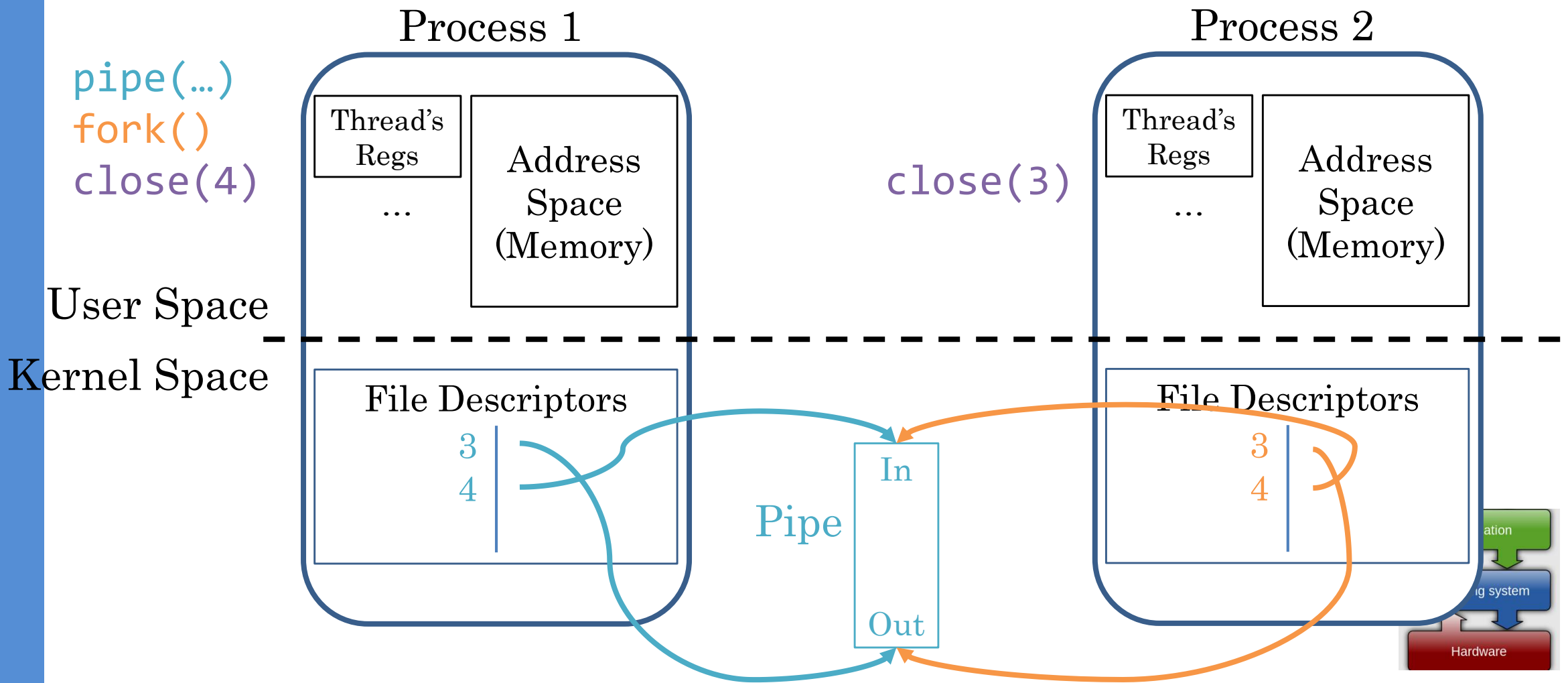


Pipes Between Processes

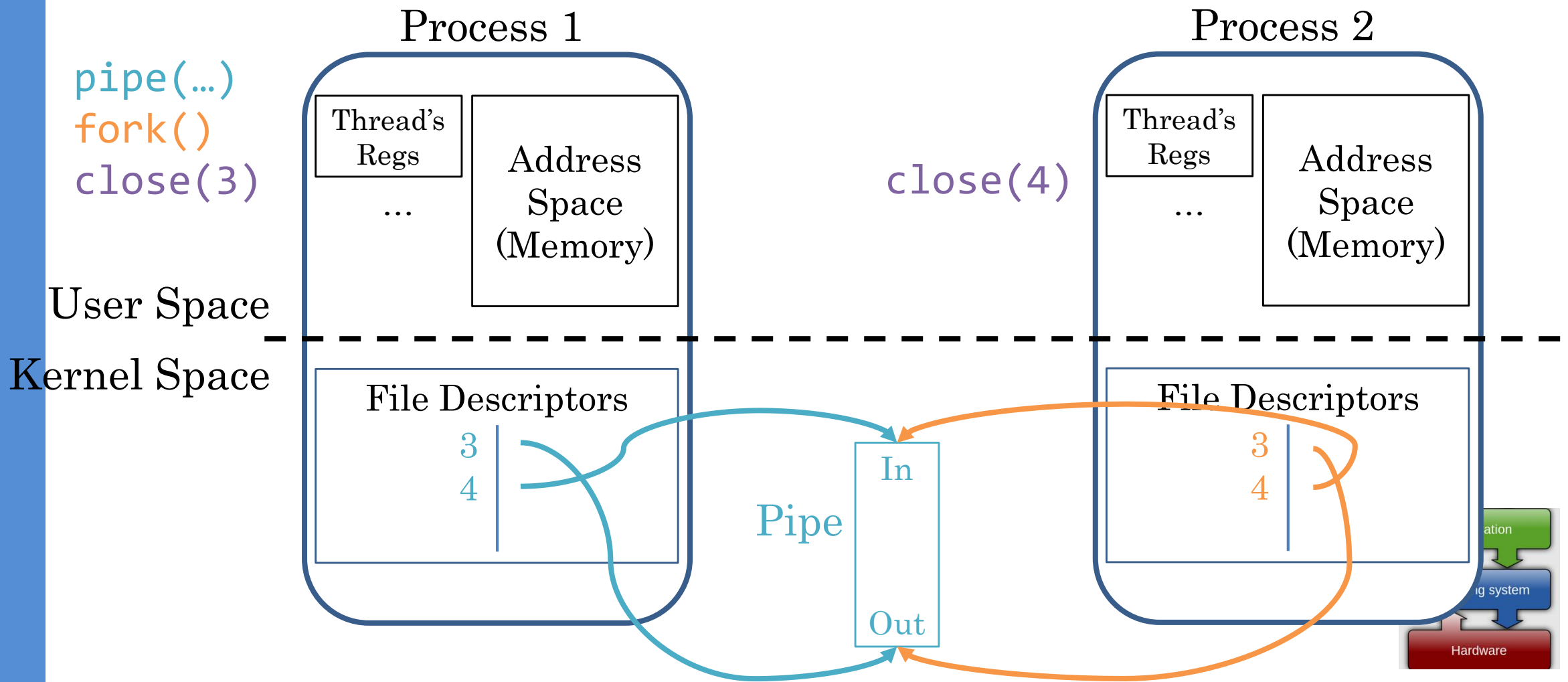
```
pipe(…)
fork()
```



Channel from Child to Parent

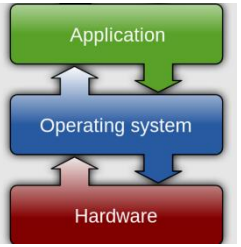


Channel from Parent to Child

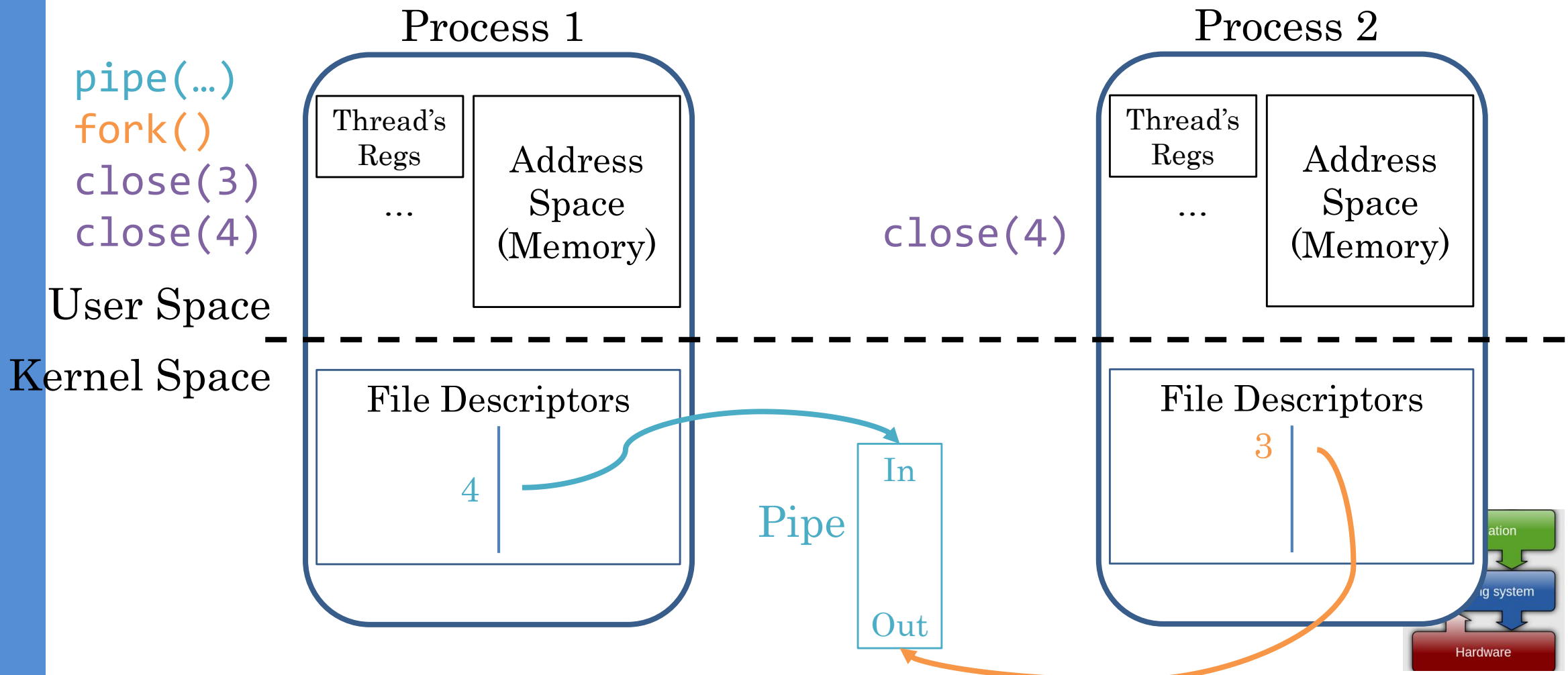


When do we get EOF on a pipe?

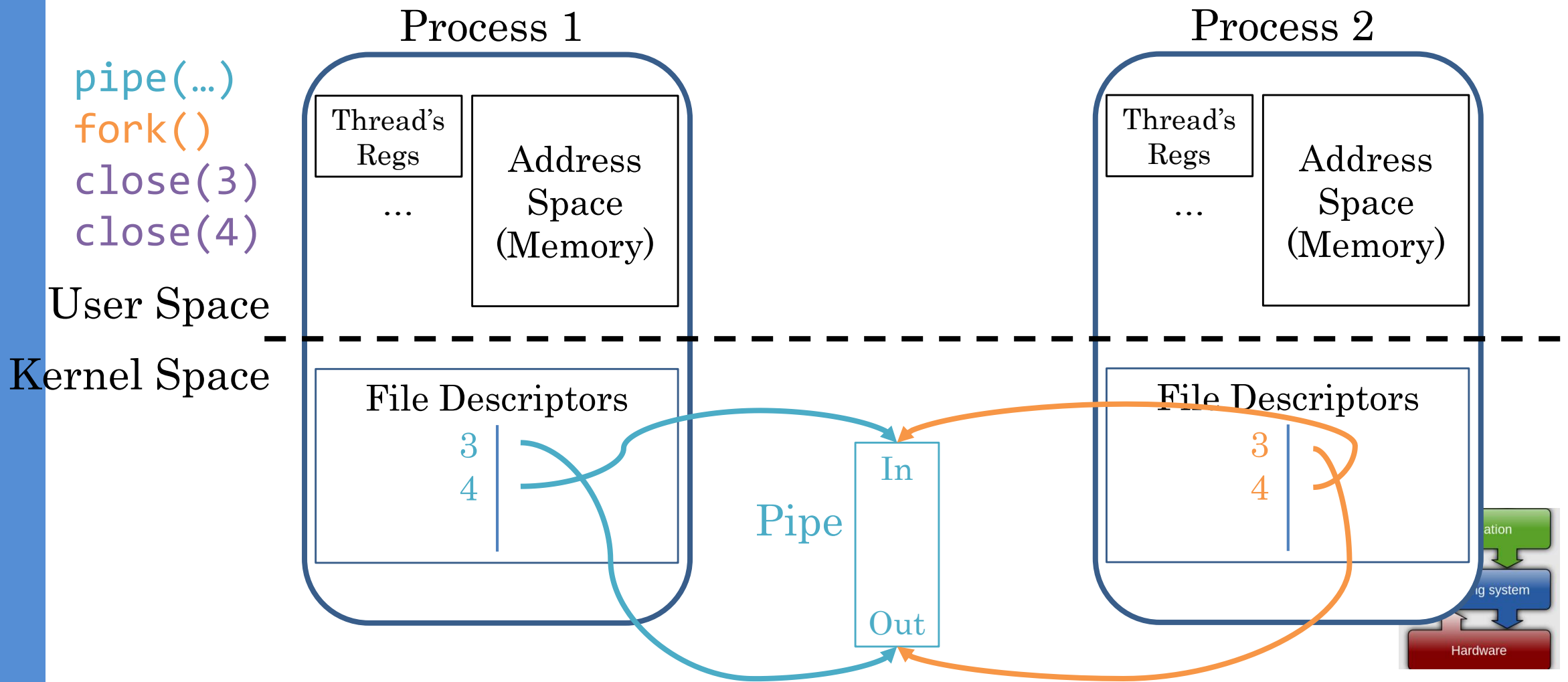
- When there a no more open file descriptors for the “write” end of the pipe



EOF on a Pipe

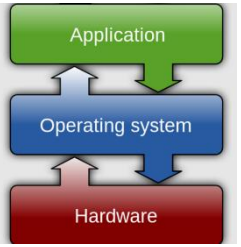

Process 1
pipe(…)
fork()
close(3)
close(4)
Thread's Regs
…
Address Space (Memory)
User Space
Kernel Space
File Descriptors
4
Pipe
In
Out
close(4)
Process 2
Thread's Regs
…
Address Space (Memory)
File Descriptors
3
ation
g system
Hardware

EOF on a Pipe


pipe(…)
fork()
close(3)
close(4)
Process 1
Thread's Regs
…
Address Space (Memory)
Process 2
Thread's Regs
…
Address Space (Memory)
User Space
Kernel Space
File Descriptors
3
4
Pipe
In
Out
File Descriptors
3
4

Announcements

- Assignment 1 due next Monday
  - You should be finishing work on this
- Project 0 was due early this week
  - If you need an extension, please get in contact
- Project 1 will be posted soon

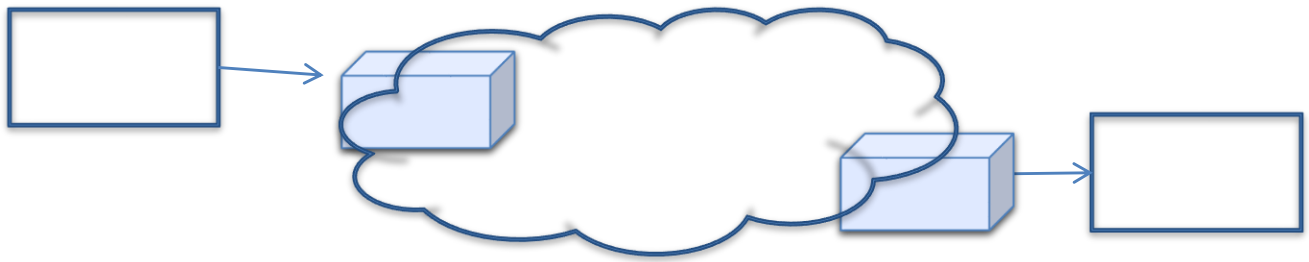


Sockets

Today: The Socket Abstraction

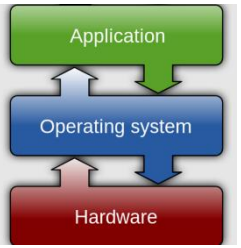
- Key Idea: Communication across the world looks like File I/O

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



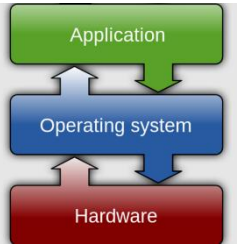
```
n = read(rfd, rbuf, rmax);
```

- Sockets: Connected queues over the Internet
  - How to open()? Filenames?
  - How are the endpoints connected in time?



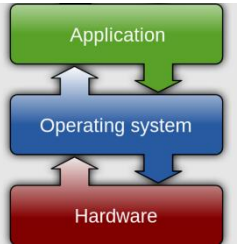
Sockets

- Socket: An abstraction for one endpoint of a network connection
  - Mechanism for inter-process communication
- First introduced in 4.2 BSD Unix
  - Most operating systems (Linux, Mac OS X, Windows) provide this, even if they don't copy rest of UNIX I/O
  - Standardized by POSIX
- 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, …)



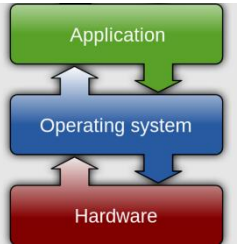
What is a Network Connection?

- In this class we will study so-called "TCP Connections"
- Bidirectional stream of bytes between two processes on possibly different machines
- Abstractly, a connection between two endpoints A and B consists of:
  - A queue (bounded buffer) for data sent from A to B
  - A queue (bounded buffer) for data sent from B to A



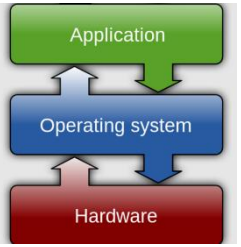
Sockets

- 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…



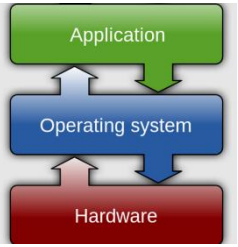
What is a Protocol?

- A protocol is an agreement on how to communicate
- Includes
  - Syntax: how a communication is specified & structured
    - Format and order of messages that are sent and received
  - Semantics: what a communication means
    - Actions taken when transmitting, receiving, or when a timer expires
- Described formally by a state machine
  - Often represented as a message transaction diagram

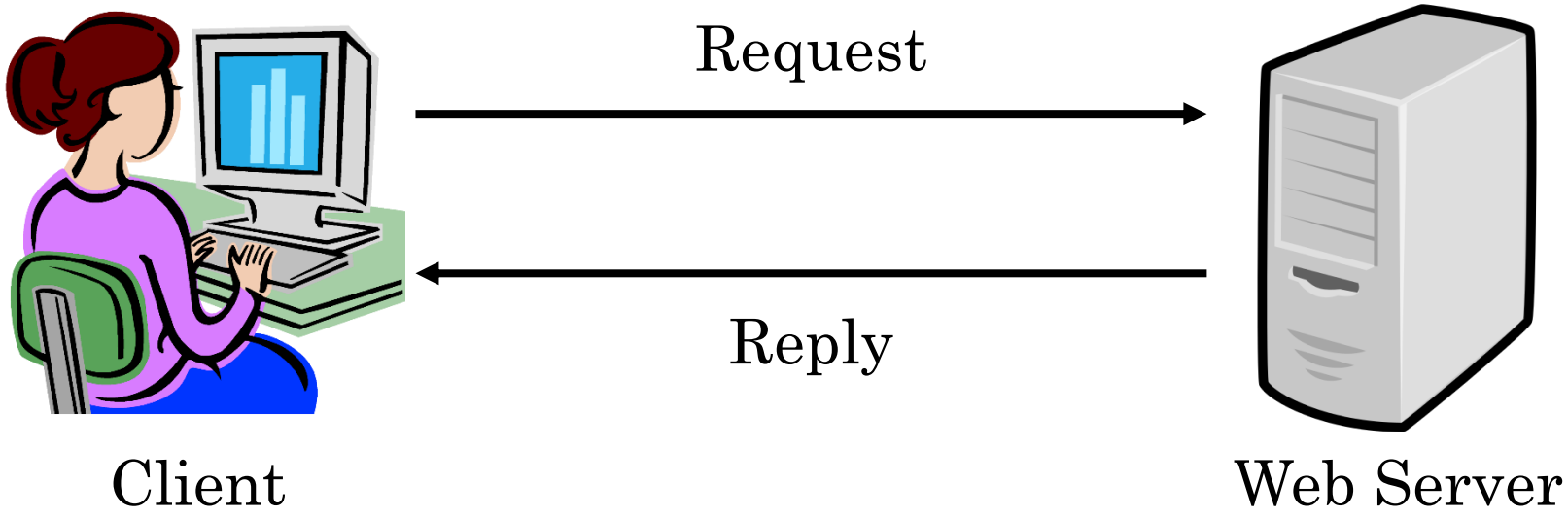


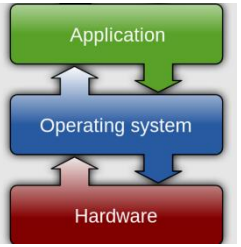
Examples of Protocols in Human Interaction

1. Telephone
2. (Pick up / open up the phone)
3. Listen for a dial tone / see that you have service
4. Dial
5. Should hear ringing …
6. Callee: "Hello?"
7. Caller: "Hi, it's John…."
   Or: "Hi, it's me" (what's that about?)
8. Caller: "Hey, do you think … blah blah blah …" pause
9. Callee: "Yeah, blah blah blah …" pause
10. Caller: Bye
11. Callee: Bye
12. Hang up

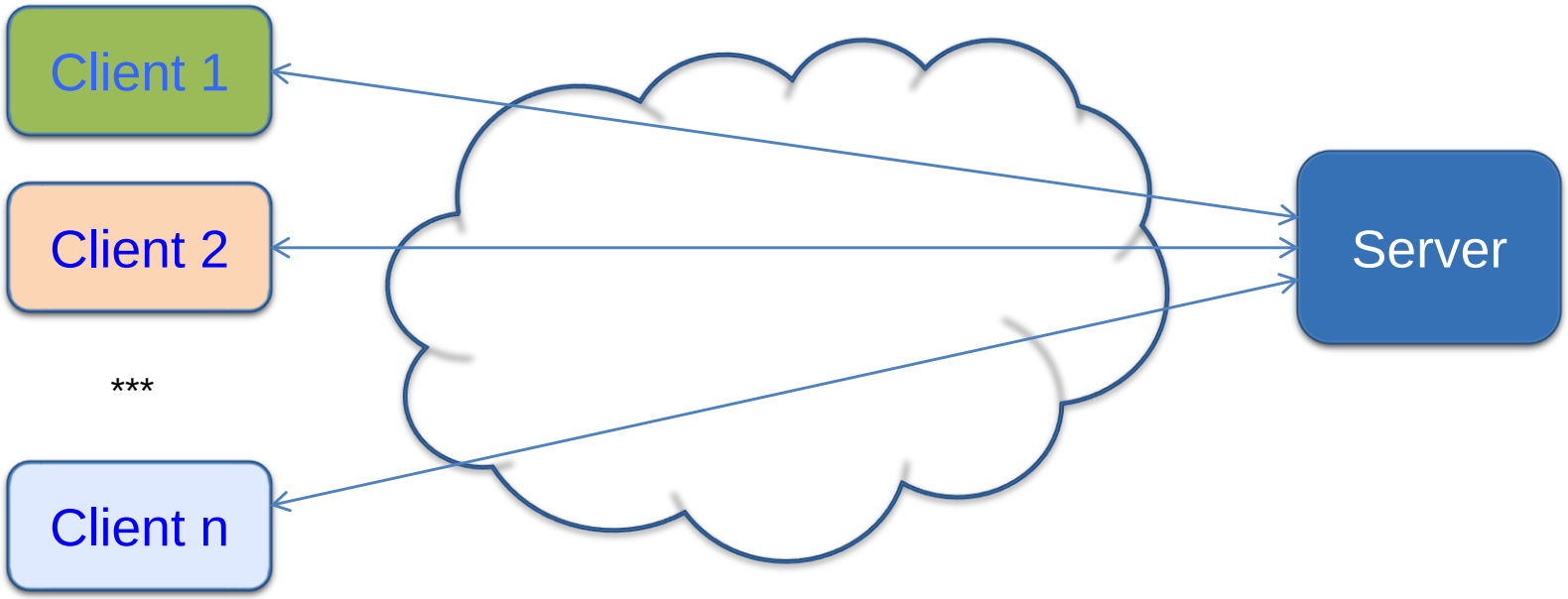


Web Server

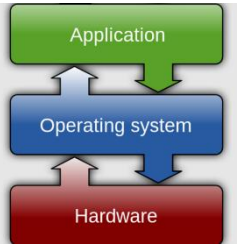

Request
Reply
Client
Web Server


Application
Operating system
Hardware

Client-Server Protocols



- Many clients accessing a common server
- File servers, www, FTP, databases

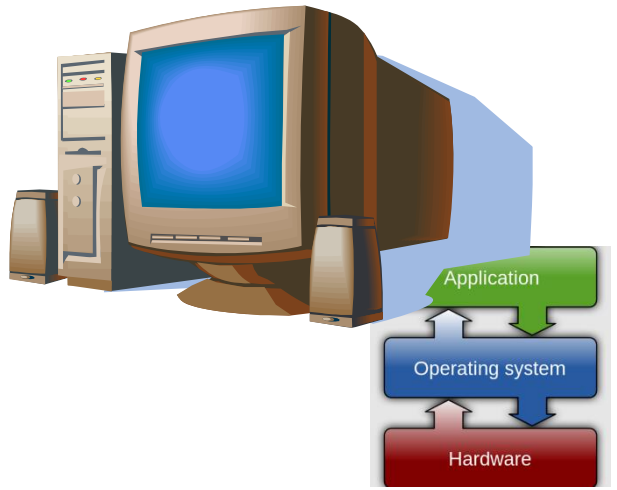


Client-Server Communication

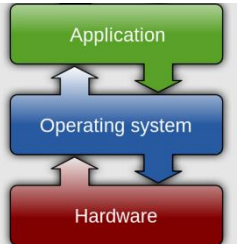
- Client is "sometimes on"
  - Sends the server requests for services when interested
  - E.g., Web browser on laptop/phone
  - Doesn't communicate directly with other clients
  - Needs to know server's address
- Server is "always on"
  - Services requests from many clients
  - E.g., Web server for www.lsu.edu
  - Doesn't initiate contact with clients
  - Needs a fixed, well-known address

```
GET /index.html
```

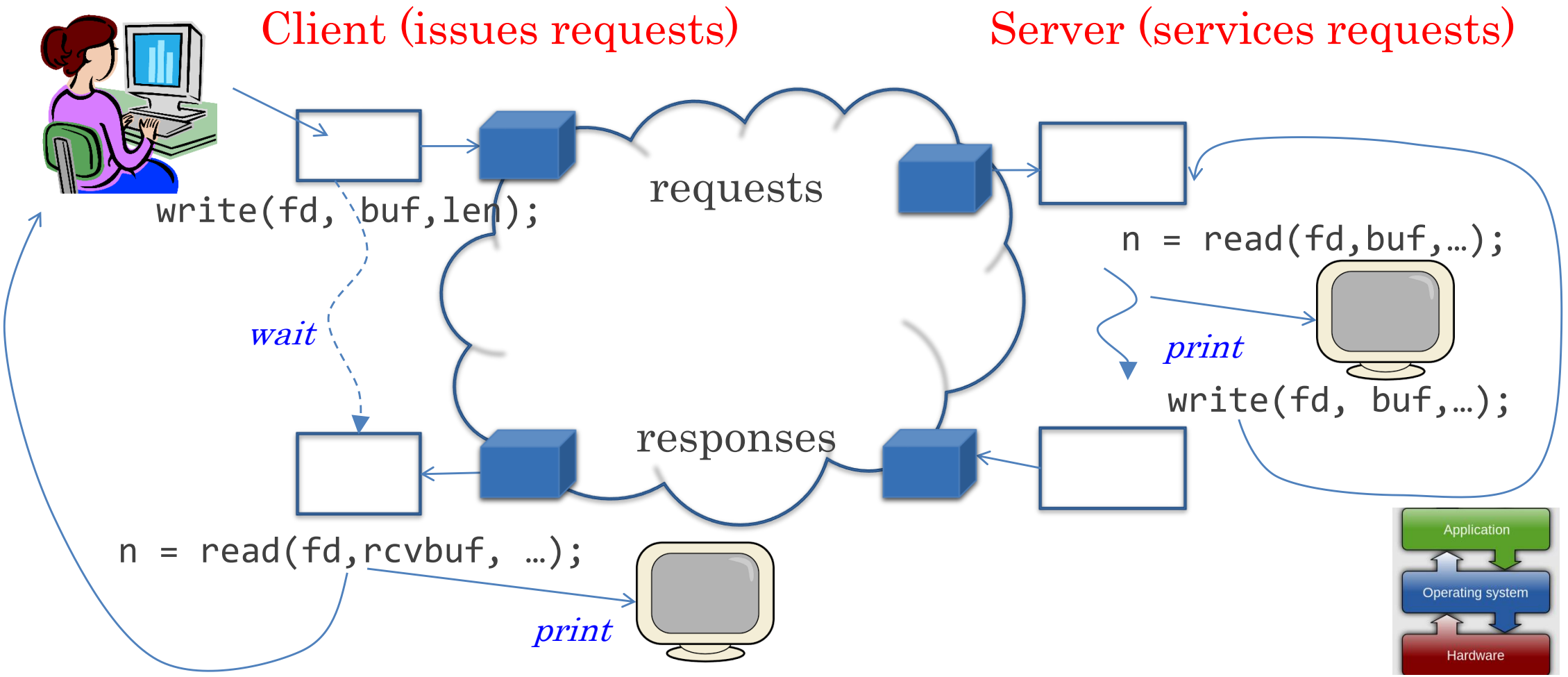
```
"Site under construction"
```



Simple Example: Echo Server



Simple Example: Echo Server



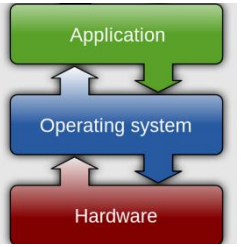
Echo Server (One Request)

client

```
char buf[BUF_SIZE];
fgets(buf, BUF_SIZE, stdin);           // prompt
write(sockfd, buf, strlen(sndbuf));    // send request
memset(buf, 0, BUF_SIZE);              // clear
read(sockfd, buf, BUF_SIZE-1);         // receive response
printf("%s\n", buf);                   // echo
```

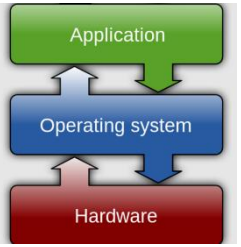
server

```
char buf[BUF_SIZE];
memset(buf, 0, BUF_SIZE);
read(consockfd, reqbuf, MAXREQ-1);          // receive
printf("%s\n", buf);                        // echo
write(consockfd, buf, strlen(reqbuf));      // send response
```



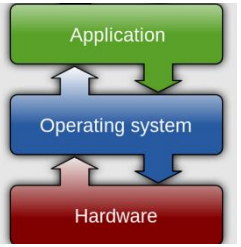
What Assumptions are we Making?

- Reliable
  - Write to a file => Read it back. Nothing is lost.
  - Write to a (TCP) socket => Read from the other side, same.
  - Like pipes
- 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. Assumes writing already took place.
  - Like pipes!



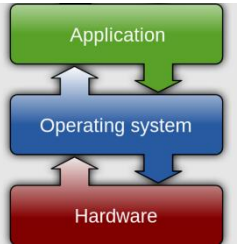
Socket Creation

- Files: permanent objects
  - Files exist independently of processes
  - Easy to name what file to open()
- Pipes: descriptors inherited from parent process
- Sockets are transient, tied to particular processes (the two endpoints!)
  - Processes are 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?

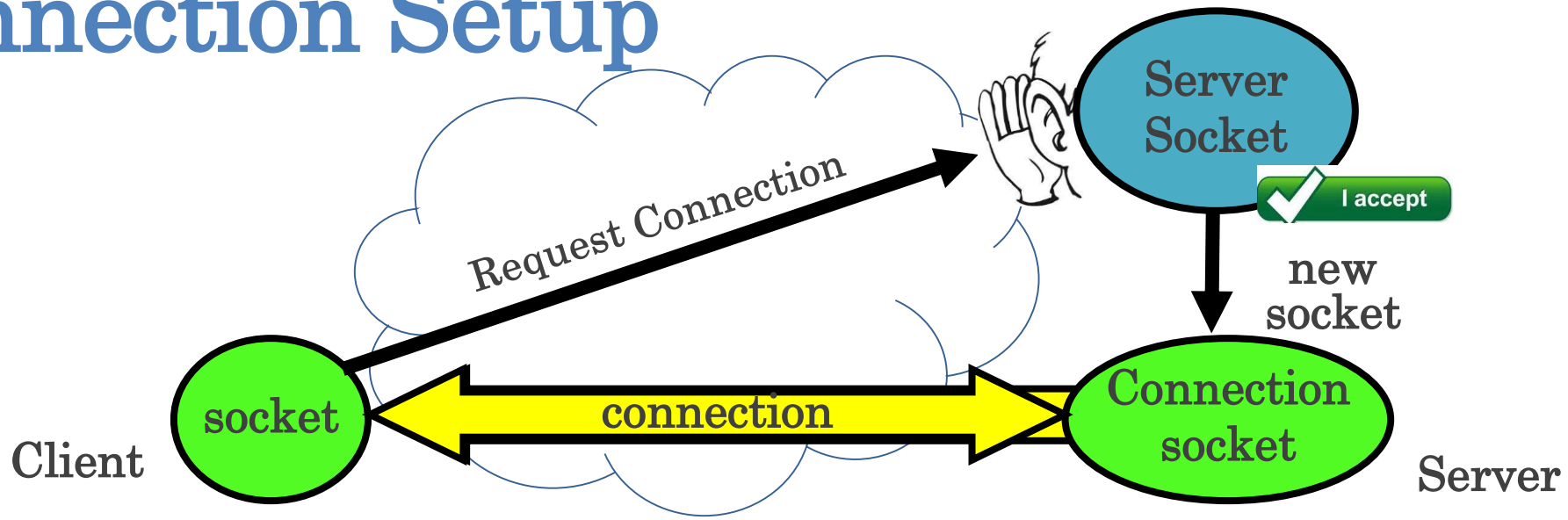


Namespaces for Communication over IP

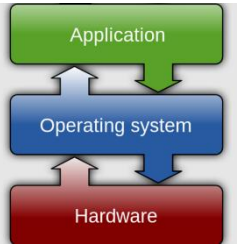
- Hostname
  - www.lsu.edu
- IP address
  - 130.39.6.220 (IPv4, 32-bit Integer)
  - 2600:1702:4930:cb0::1(IPv6, 128-bit Integer)
- Port Number
  - 0 – 1023 are "well known" or "system" ports
    - Superuser privileges to bind to one
  - 1024 – 49151 are "registered" ports (registry)
    - Assigned by IANA for specific services
  - 49152 – 65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are "dynamic" or "private"
    - Automatically allocated as "ephemeral ports"



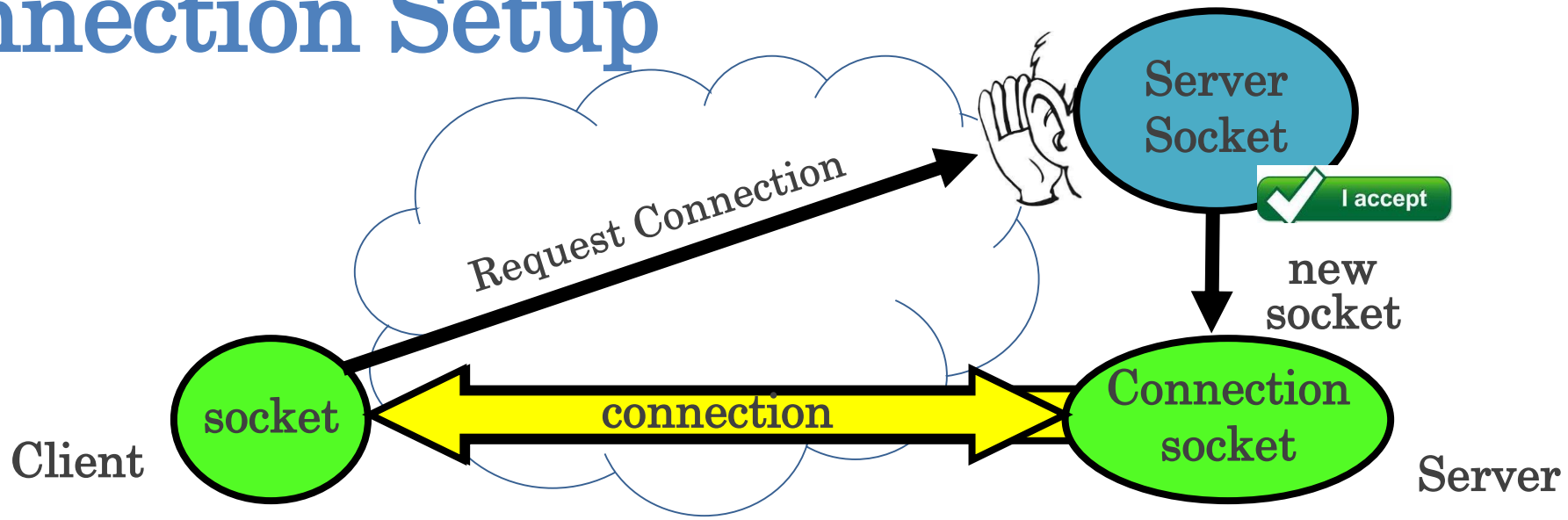
Connection Setup



- 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

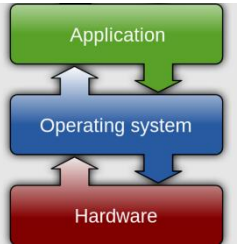


Connection Setup

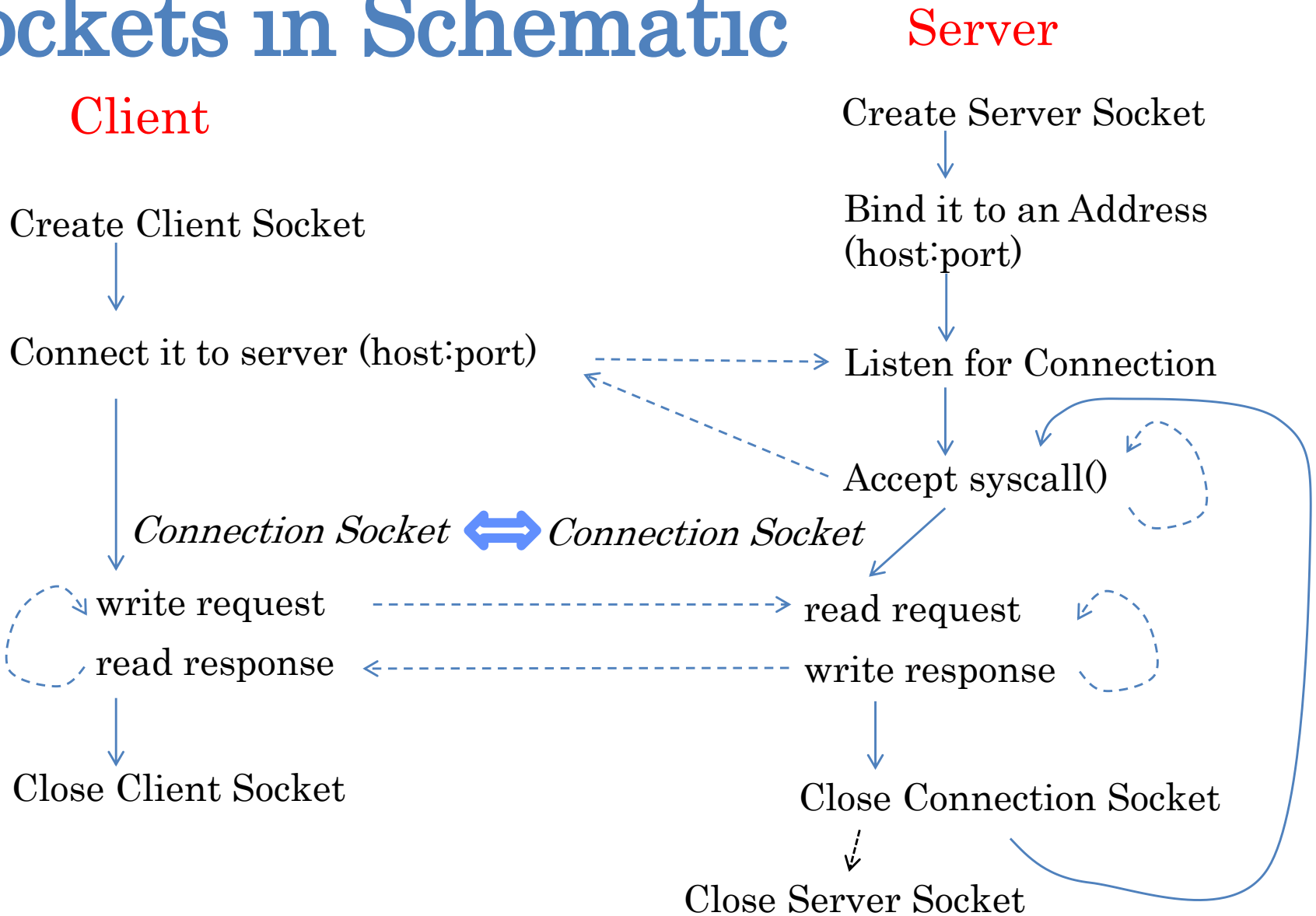


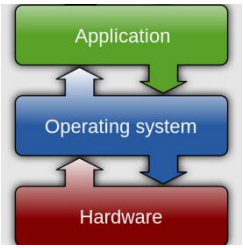
- 5-Tuple identifies each connection:
  - Source IP Address
  - Destination IP Address
  - Source Port Number
  - Destination Port Number
  - Protocol (always TCP here)

- 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…1023



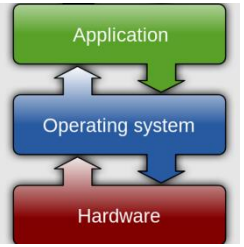
Sockets in Schematic





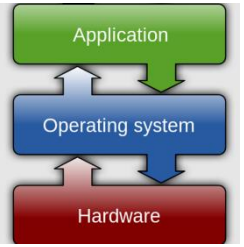
Client Protocol

```
char* host_name = "www.lsu.edu";
char* port = "80";

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

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

// Carry out Client-Server protocol
run_client(sock_fd);

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



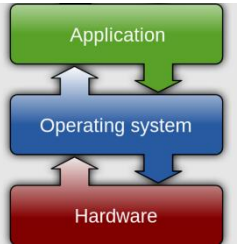
Server Protocol (v1)

```
// Create socket to listen for client connections
char *port = "80";
struct addrinfo *server = setup_address(port);
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);
```

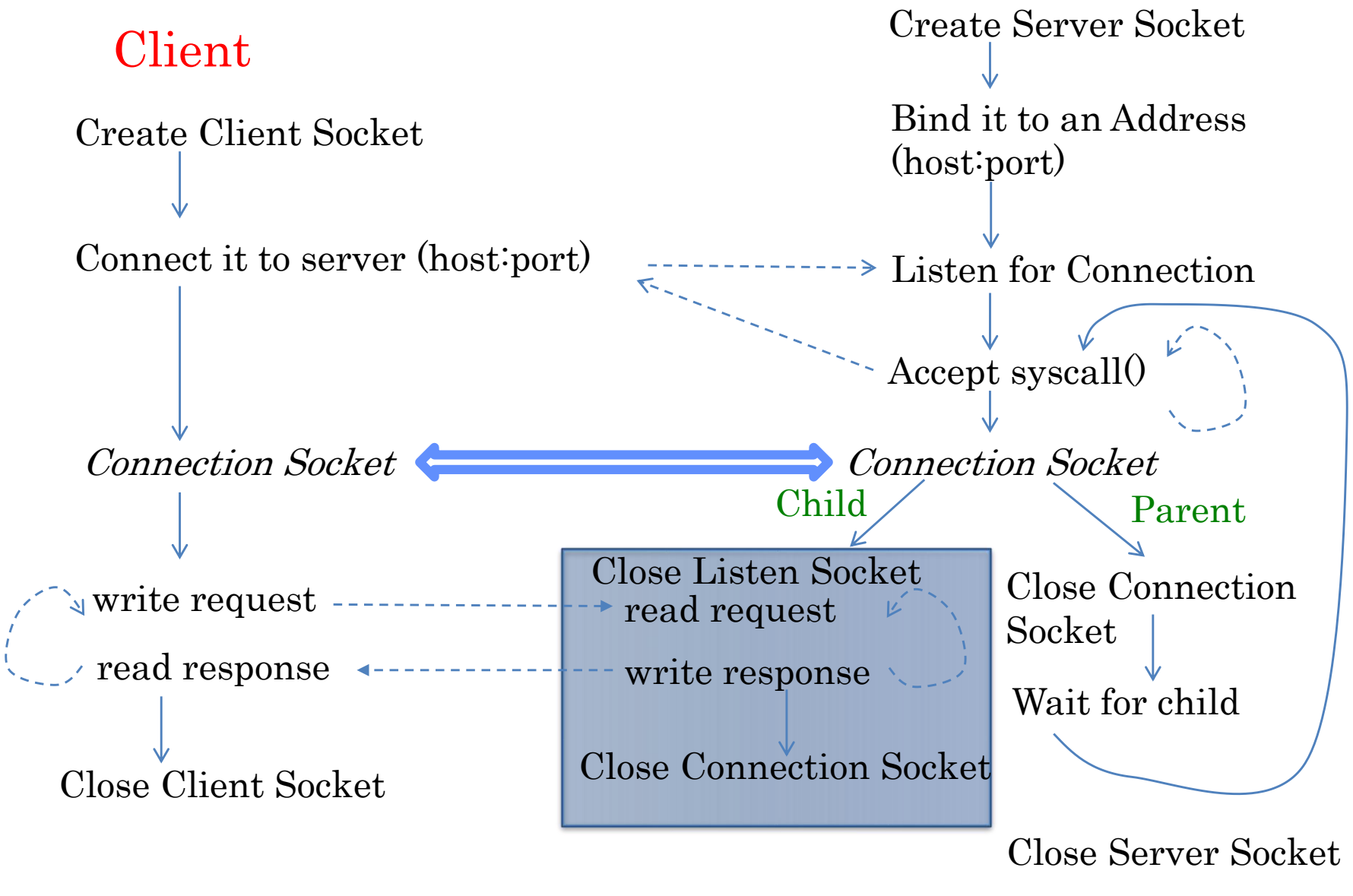


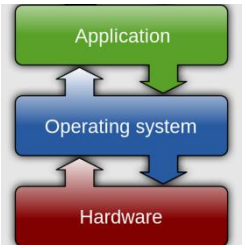
How Does the Server Protect Itself?

- Handle each connection in a separate process



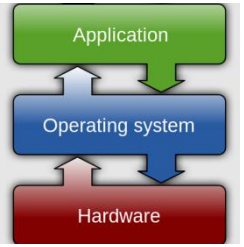
Sockets with Protection





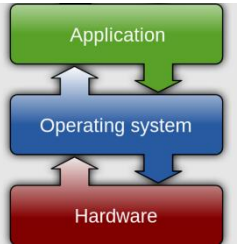
Server Protocol (v2)

```
// Socket setup code elided…
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) {          // child
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {                 // parent
    close(conn_socket);
    wait(NULL);
  }
}
close(server_socket);
```

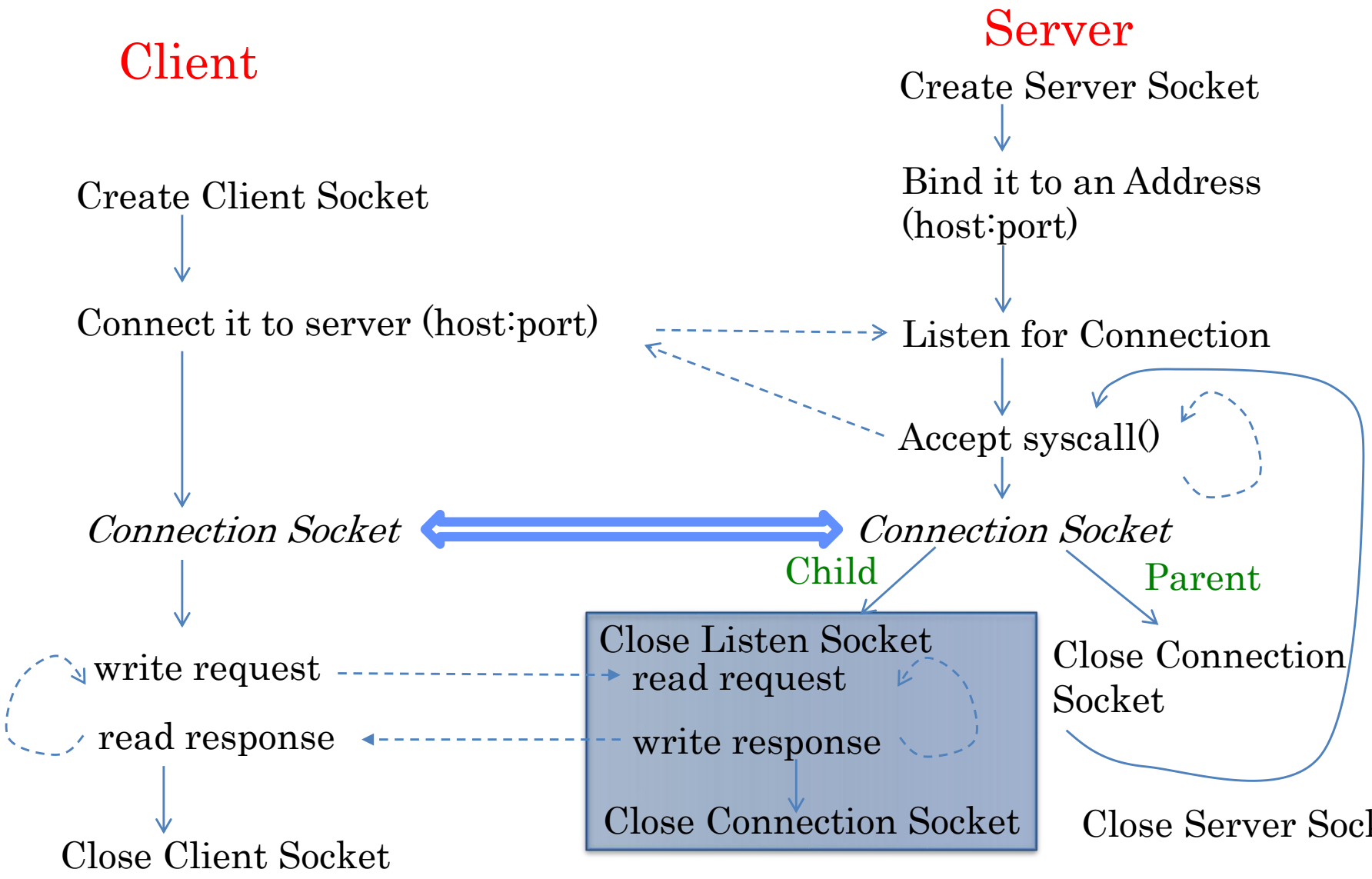


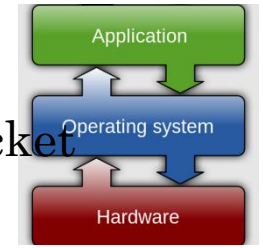
Concurrent Server

- So far, in the server:
  - Listen will queue requests
  - Buffering present elsewhere
  - But server waits for each connection to terminate before servicing the next
- A concurrent server can handle and service a new connection before the previous client disconnects



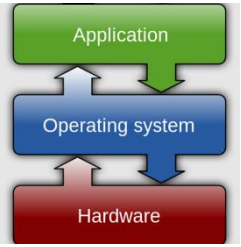
Sockets with Protection and Concurrency





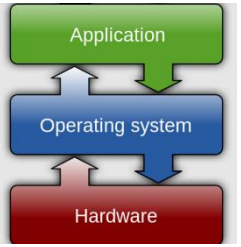
Server Protocol (v3)

```
// Socket setup code elided…
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) {            // child
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else {                   // parent
    close(conn_socket);
    //wait(NULL);
  }
}
close(server_socket);
```

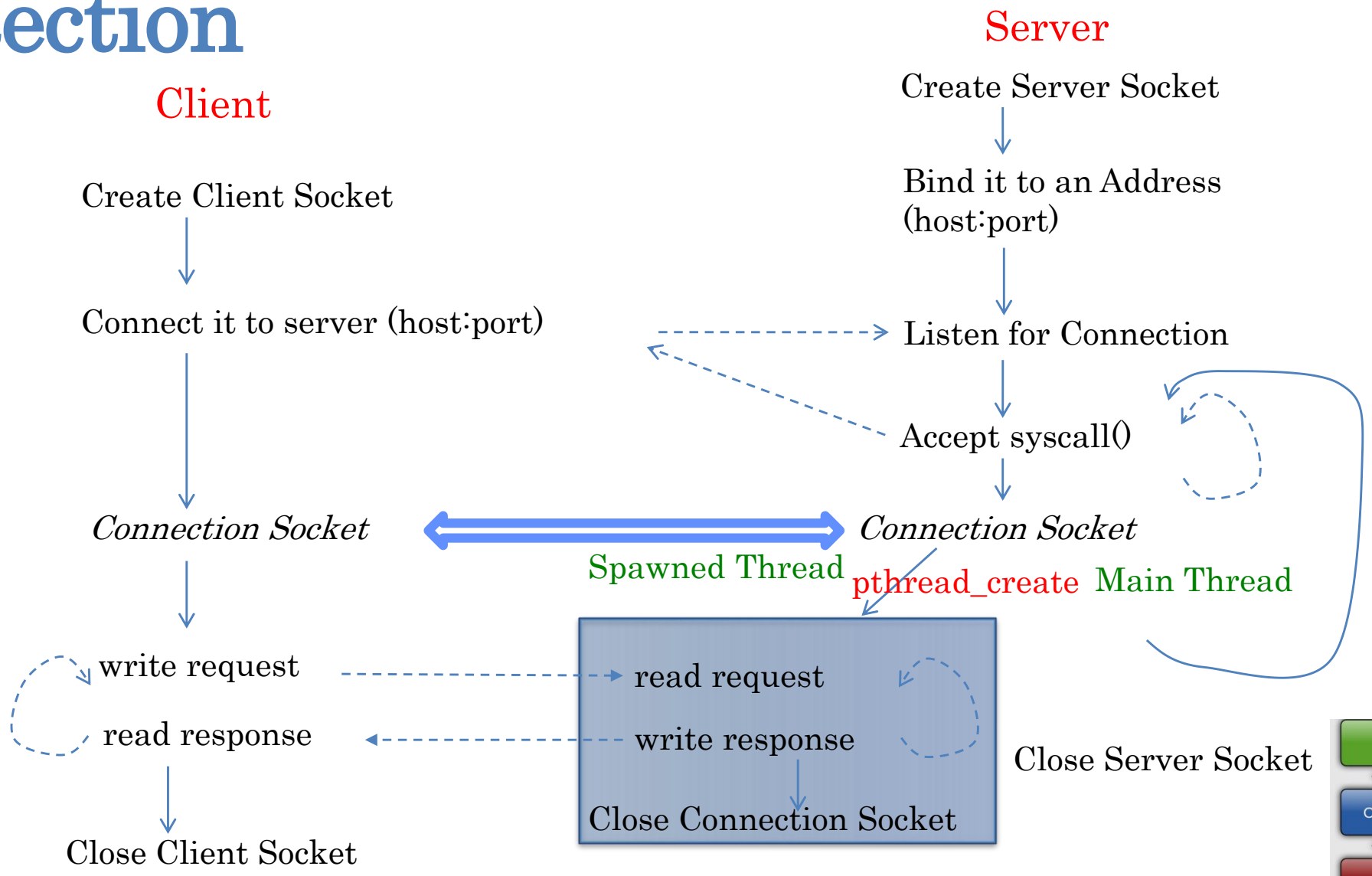


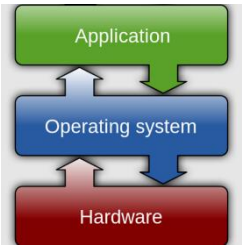
Concurrent Server without Protection

- Spawn a new thread to handle each connection
- 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



Sockets with Concurrency, without Protection

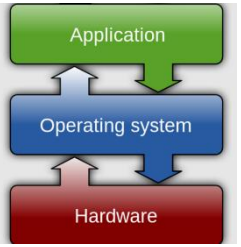




Server Address: Itself

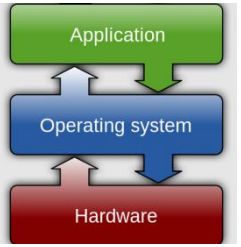
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  getaddrinfo(NULL, port, &hints, &server);
  return server;
}
```

- Accepts any connections on the specified port



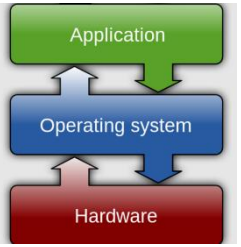
Client: Getting the Server Address

```
struct addrinfo *lookup_host(char *host_name, char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  // hints.ai_flags = AI_PASSIVE;

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



Conclusion

- Pipes are an abstraction of a single queue
  - One end write-only, another end read-only
  - Used for communication between multiple processes on one machine
  - File descriptors obtained via inheritance
- Sockets are an abstraction of two queues, one in each direction
  - Can read or write to either end
  - Used for communication between multiple processes on different machines
  - File descriptors obtained via socket/bind/connect/listen/accept
  - Inheritance of file descriptors on fork() facilitates handling each connection in a separate process
- Both support read/write system calls, just like File I/O



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP