

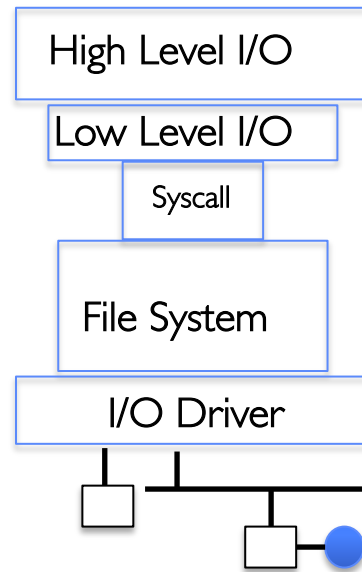
# Operating Systems (Honor Track)

Abstraction 2: Files and IO (cont'd)  
Abstraction 3: IPC, Pipes and Sockets  
A quick, programmer's viewpoint

Xin Jin  
Spring 2022

# Recap: I/O and Storage Layers

## Application / Service



*Streams*

*File Descriptors*

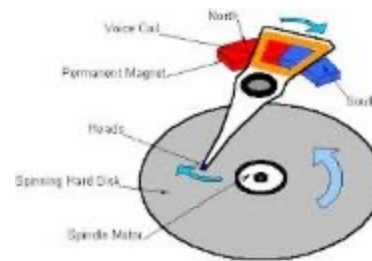
*open(), read(), write(), close(), ...*

*Open File Descriptions*

*Files/Directories/Indexes*

*Commands and Data Transfers*

*Disks, Flash, Controllers, DMA*



# Recap: High-Level vs. Low-Level File API

## High-Level Operation:

```
size_t fread(...) {  
    Do some work like a normal fn...
```

asm code ... syscall # into %eax  
put args into registers %ebx, ...  
*special trap instruction*

Kernel:

get args from regs  
dispatch to system func  
Do the work to read from the file  
Store return value in %eax

get return values from regs  
*Do some more work like a normal fn...*

```
};
```

## Low-Level Operation:

```
ssize_t read(...) {
```

asm code ... syscall # into %eax  
put args into registers %ebx, ...  
*special trap instruction*

Kernel:

get args from regs  
dispatch to system func  
Do the work to read from the file  
Store return value in %eax

get return values from regs

```
};
```

# Recap: High-Level vs. Low-Level File API

## Program 1

```
printf("Beginning of line ");  
sleep(10); // sleep for 10 seconds  
printf("and end of line\n");
```

## Program 2

```
write(STDOUT_FILENO, "Beginning of line ", 18);  
sleep(10);  
write("and end of line \n", 16);
```

- Group discussion
  - What are the behaviors of the two programs? Why?
- Program 1
  - Streams are buffered in user memory
  - Prints out everything at once
- Program 2
  - Operations on file descriptors are visible immediately
  - Outputs "Beginning of line" 10 seconds earlier than "and end of line"

# What's in a FILE?

- What's in the FILE\* returned by fopen?
  - File descriptor (from call to open) <= Need this to interface with the kernel!
  - Buffer (array)
  - Lock (in case multiple threads use the FILE concurrently)
- Of course, there's other stuff in a FILE too...
- ... but this is useful model to have

# FILE Buffering

- When you call `fwrite`, what happens to the data you provided?
  - It gets written to the FILE's buffer
  - If the FILE's buffer is full, then it is *flushed*
    - » Which means it's written to the underlying file descriptor
  - The C standard library *may* flush the FILE more frequently
    - » e.g., if it sees a certain character in the stream
- When you write code, make the weakest possible assumptions about how data is flushed from FILE buffers

# Example

```
char x = 'c';  
FILE* f1 = fopen("file.txt", "w");  
fwrite("b", sizeof(char), 1, f1);  
FILE* f2 = fopen("file.txt", "r");  
fread(&x, sizeof(char), 1, f2);
```

- What is the value of x?
  - The call to fread might see the latest write 'b'
  - Or it might miss it and see end of file (in which case x will remain 'c')

# Example

```
char x = 'c';  
FILE* f1 = fopen("file.txt", "wb");  
fwrite("b", sizeof(char), 1, f1);  
fflush(f1);  
FILE* f2 = fopen("file.txt", "rb");  
fread(&x, sizeof(char), 1, f2);
```

- Now, the call to fread will definitely see the latest write 'b'



# Writing Correct Code with FILE

- Your code should behave correctly regardless of when C Standard Library flushes its buffer
  - Add your own calls to `fflush` so that data is written when you need to
  - Calls to `fclose` flush the buffer before deallocating memory and closing the file descriptor
- With the low-level file API, we don't have this problem
  - After `write` completes, data is visible to any subsequent reads

# Why Buffer in Userspace? Overhead!

- Syscalls are more expensive than function calls
- `read/write` a file byte by byte? Max throughput of **~10MB/second**
- With `fgetc`? Keeps up with your SSD

# Why Buffer in Userspace? Functionality!

- System call operations less capable
  - Simplifies operating system
- Example: No “read until new line” operation in kernel
  - Why? Kernel *agnostic* about formatting!
  - Solution: Make a big read syscall, find first new line in userspace
    - » i.e. use one of the following high-level options:

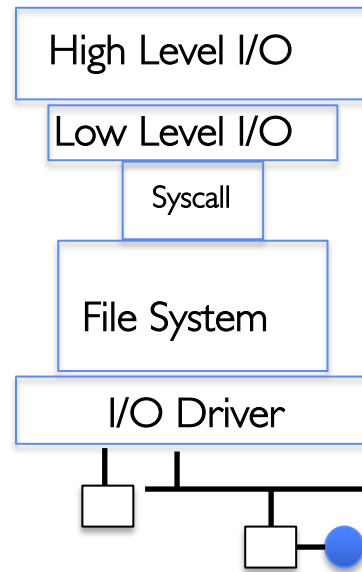
```
char *fgets(char *s, int size, FILE *stream);  
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

# The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- **Process State for File Descriptors**
- Some Pitfalls with OS Abstractions

# I/O and Storage Layers

Application / Service



*Streams*

*File Descriptors*

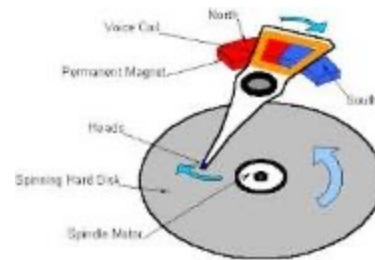
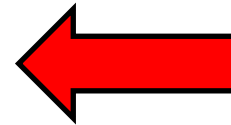
*open(), read(), write(), close(), ...*

*Open File Descriptions*

*Files/Directories/Indexes*

*Commands and Data Transfers*

*Disks, Flash, Controllers, DMA*




# State Maintained by the Kernel

- Recall: On a successful call to `open()`:
  - A file descriptor (int) is returned to the user
  - An open file description is created in the kernel
- For each process, kernel maintains mapping from file descriptor to open file description
  - On future system calls (e.g., `read()`), kernel looks up **open file description** using **file descriptor** and uses it to service the system call:

```
char buffer1[100];  
char buffer2[100];  
int fd = open("foo.txt", O_RDONLY);  
read(fd, buffer1, 100);  
read(fd, buffer2, 100);
```

The kernel remembers that the int it receives (stored in `fd`) corresponds to `foo.txt`



The kernel picks up where it left off in the file



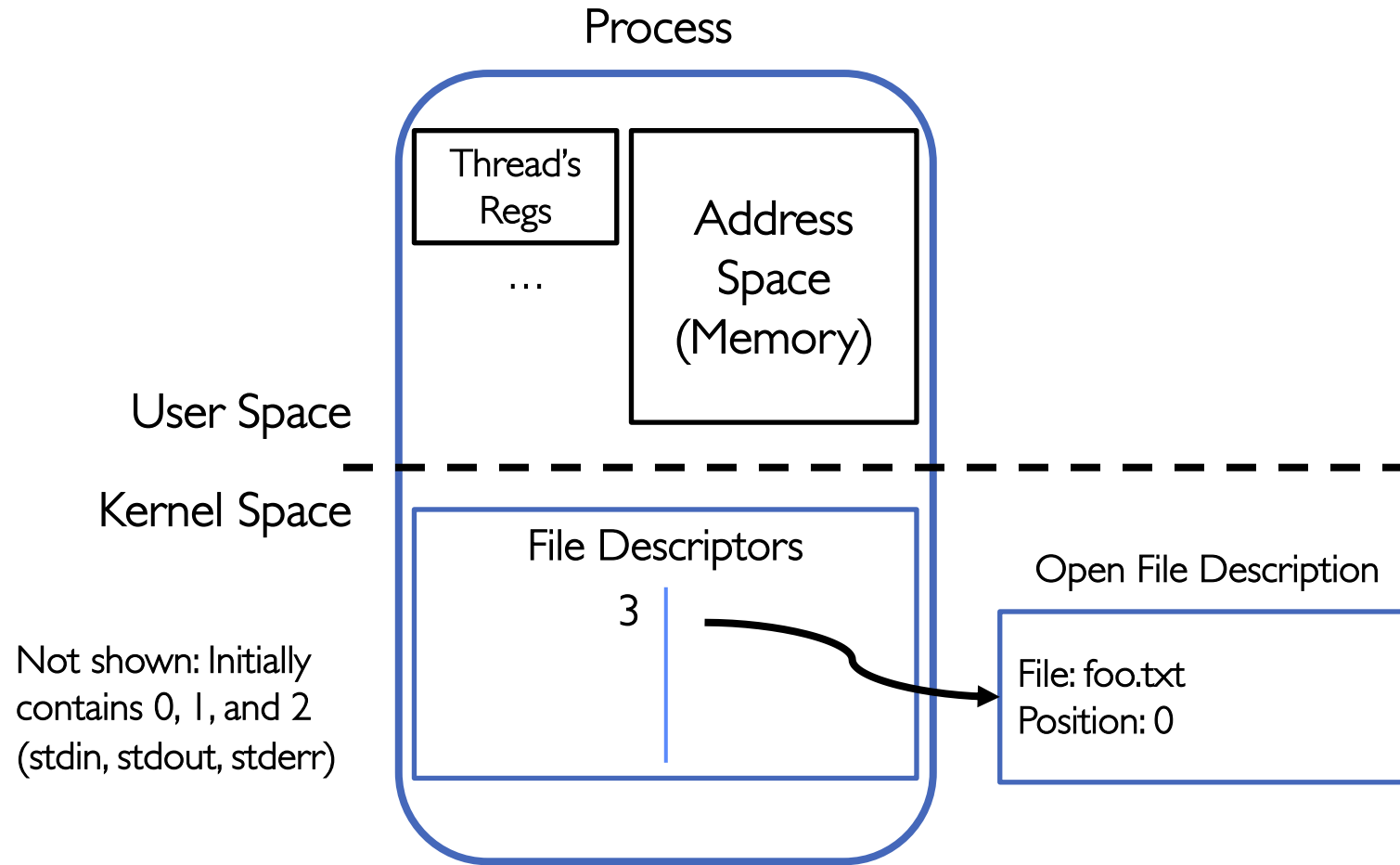
# What's in an Open File Description?

For our purposes, the two most important things are:

- Where to find the file data on disk
- The current position within the file

```
746
747 struct file {
748     union {
749         struct llist_node    fu_llist;
750         struct rcu_head      fu_rcuhead;
751     } f_u;
752     struct path              f_path;
753     #define f_dentry          f_path.dentry
754     struct inode              *f_inode; /* caci
755     const struct file_operations *f_op;
756
757     /*
758      * Protects f_ep_links, f_flags.
759      * Must not be taken from IRQ context.
760      */
761     spinlock_t               f_lock;
762     atomic_long_t             f_count;
763     unsigned int              f_flags;
764     fmode_t                   f_mode;
765     struct mutex              f_pos_lock;
766     loff_t                    f_pos;
767     struct fown_struct         f_owner;
768     const struct cred          *f_cred;
769     struct file_ra_state      f_ra;
770
771     u64                       f_version;
772     #ifdef CONFIG_SECURITY
773     void                       *f_security;
774     #endif
775     /* needed for tty driver, and maybe others */
776     void                       *private_data;
777
778     #ifdef CONFIG_EPOLL
779     /* Used by fs/eventpoll.c to link all the hook:
780     struct list_head          f_ep_links;
781     struct list_head          f_tfile_llink;
782     #endif /* #ifdef CONFIG_EPOLL */
783     struct address_space      *f_mapping;
784 } __attribute__((aligned(4))); /* lest something weird
785
```

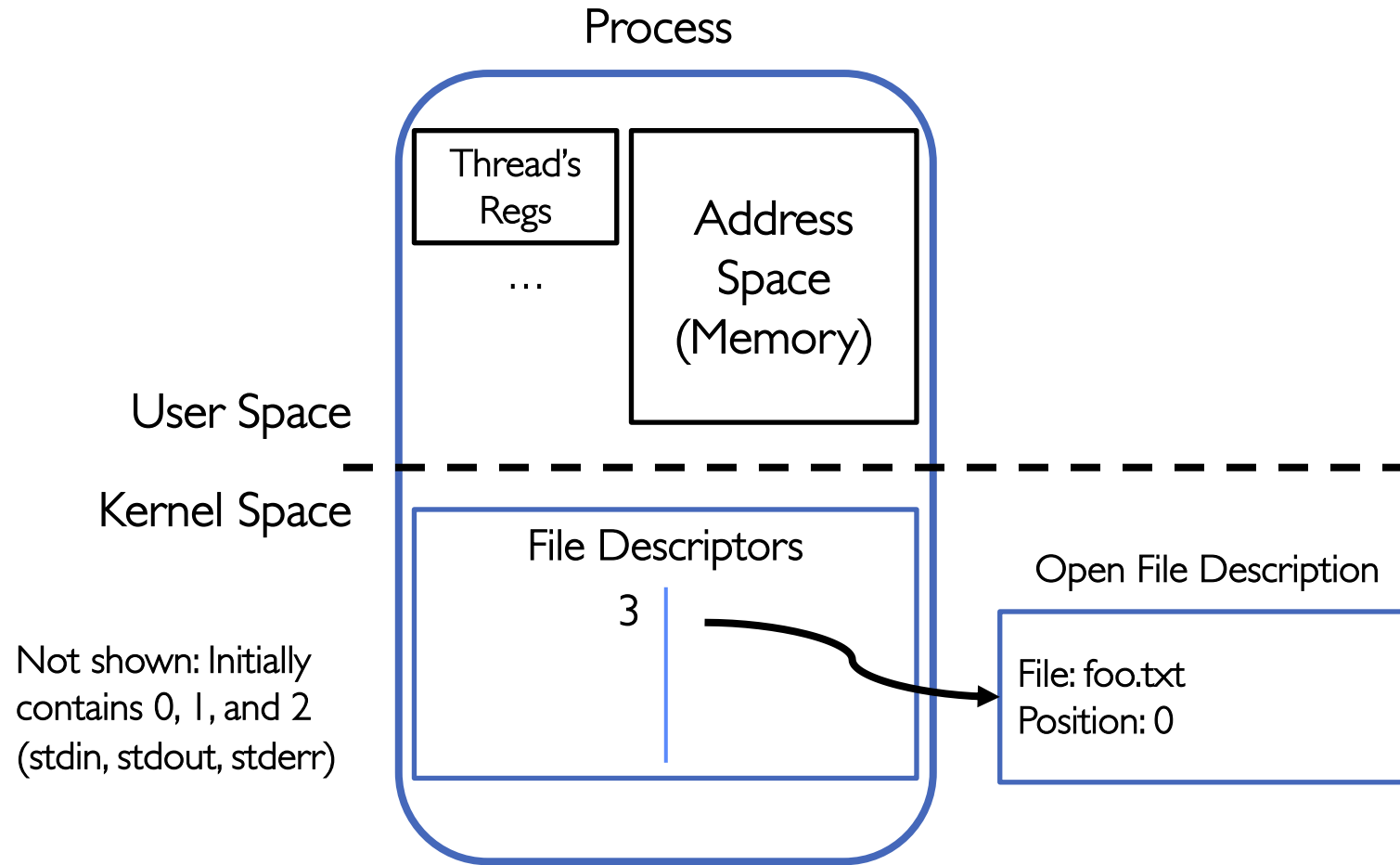
# Abstract Representation of a Process



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



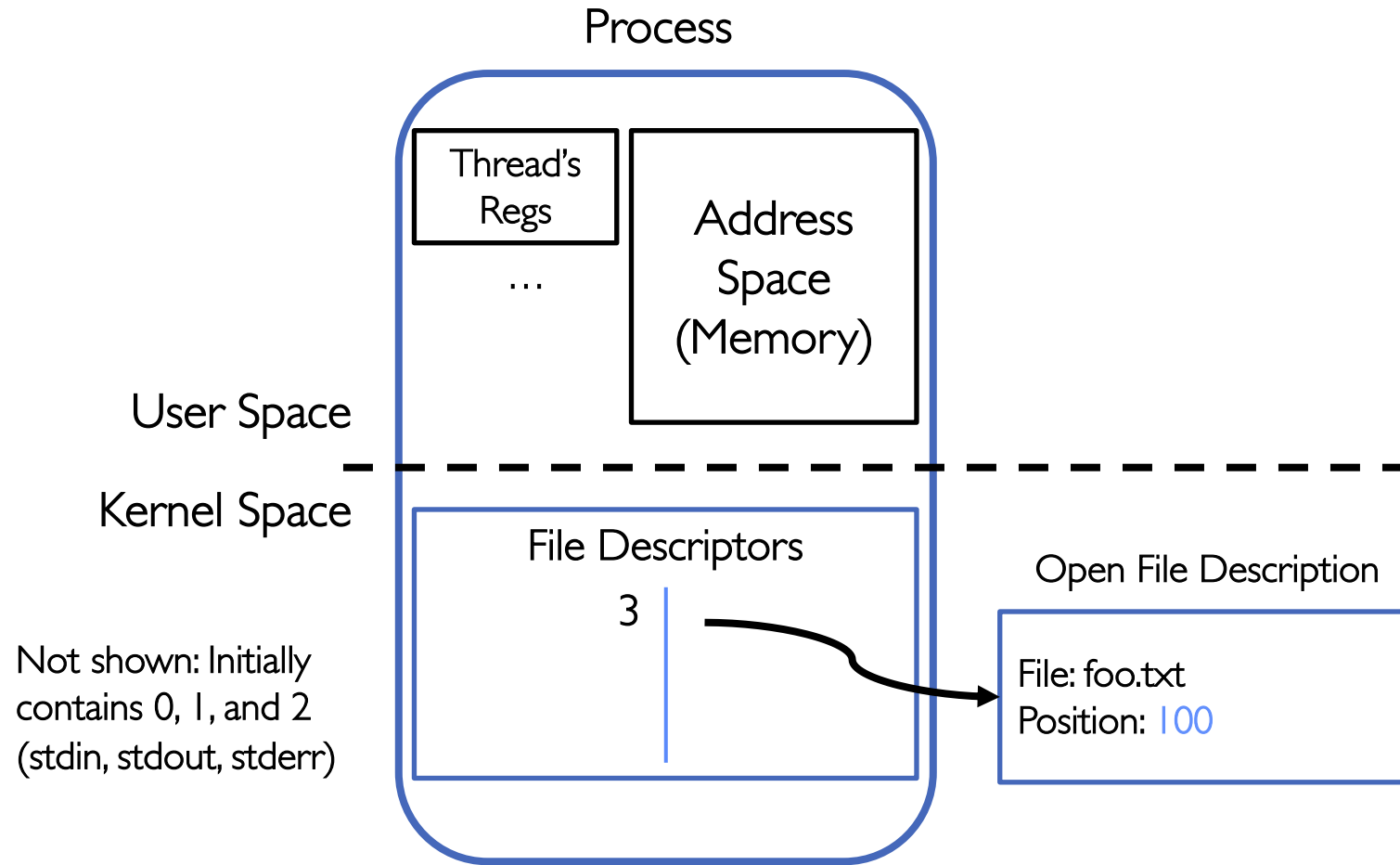
# Abstract Representation of a Process



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

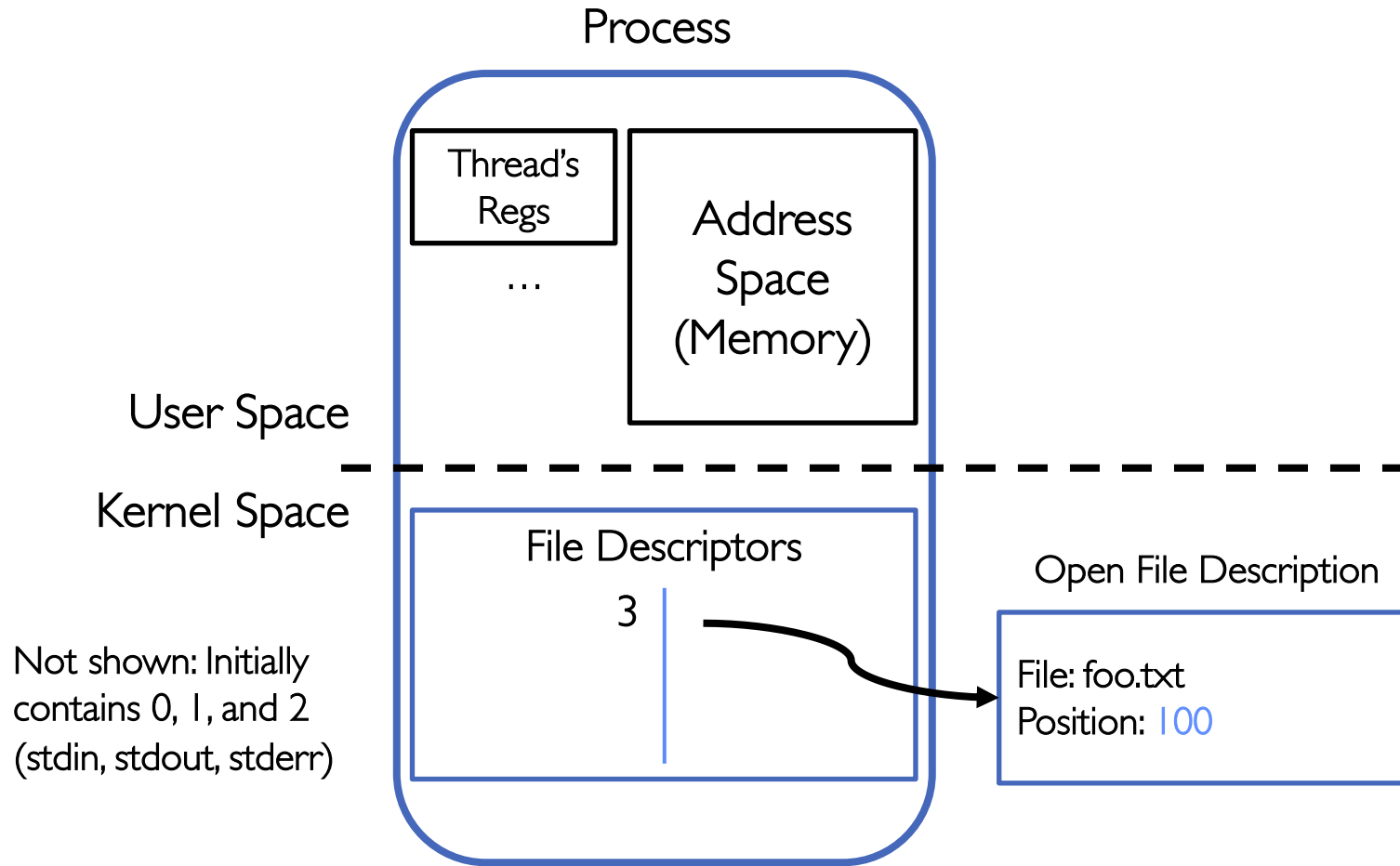
# Abstract Representation of a Process



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

# Abstract Representation of a Process

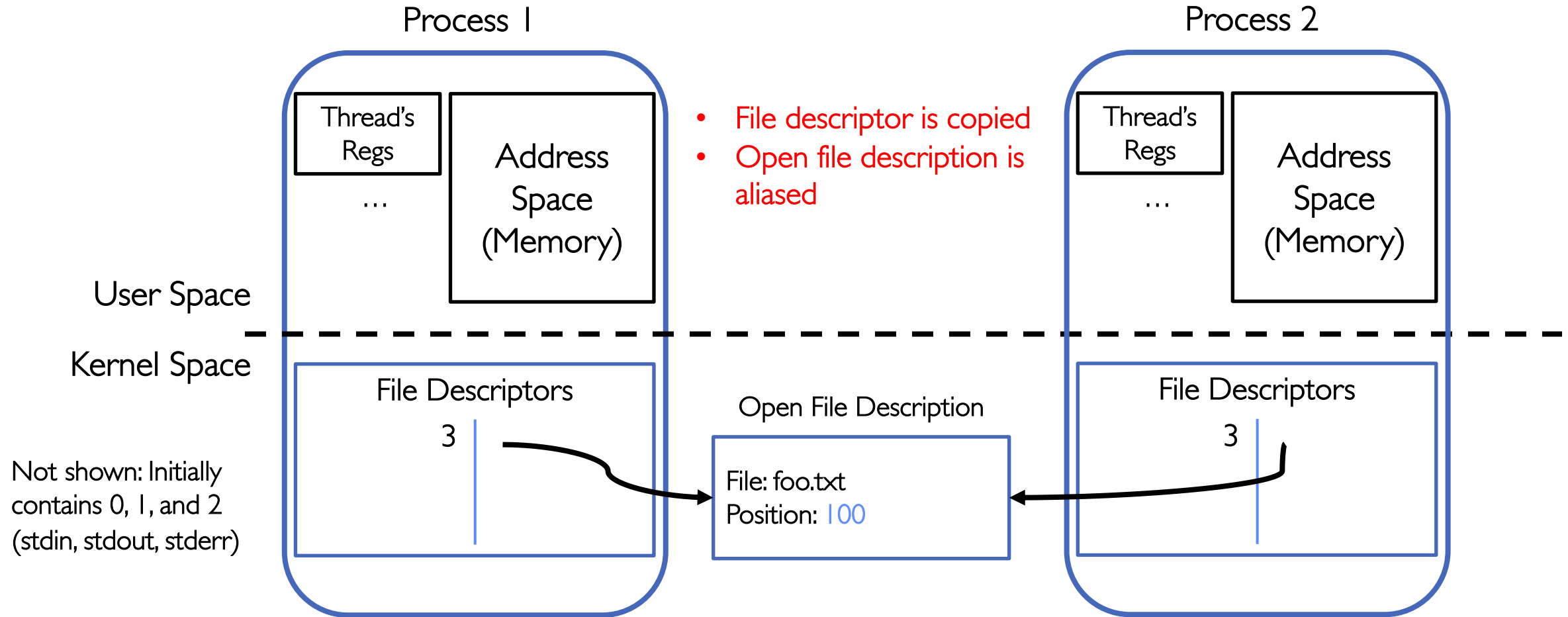


Suppose that we execute  
`open("foo.txt")`  
and that the result is 3

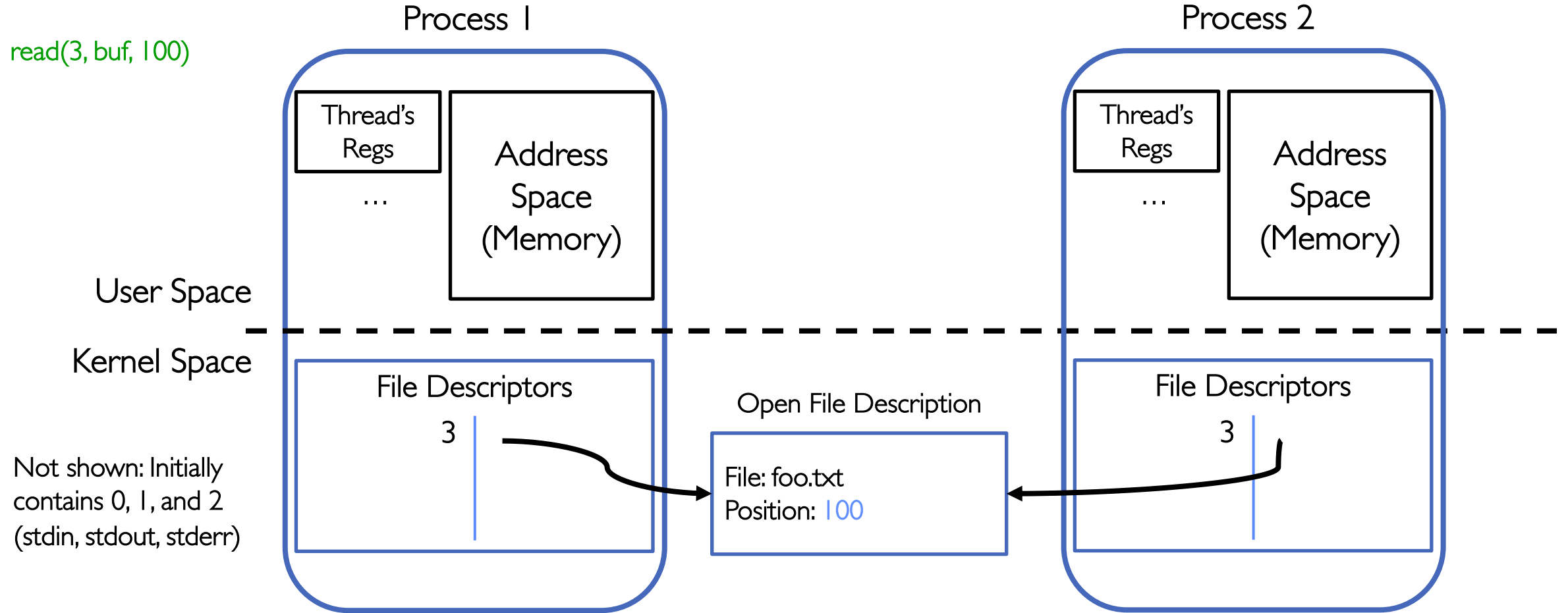
Next, suppose that we execute  
`read(3, buf, 100)`  
and that the result is 100

Finally, suppose that we execute  
`close(3)`

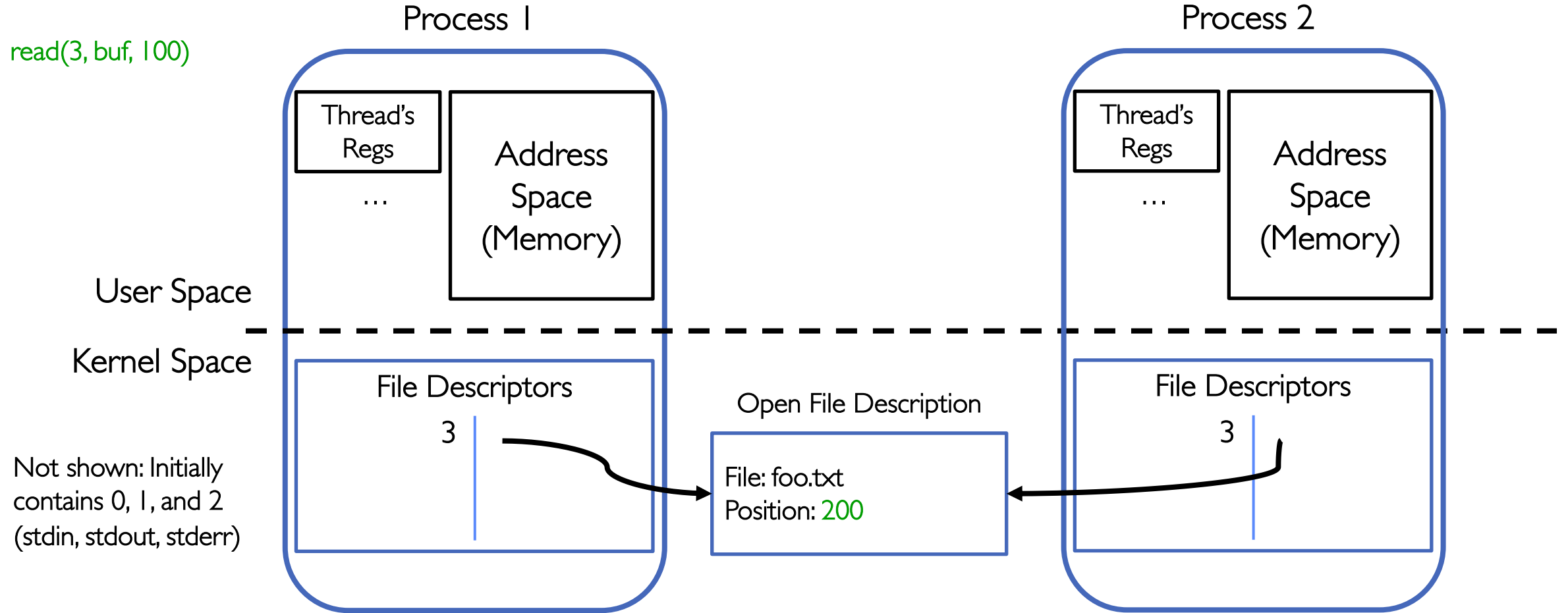
# Instead of Closing, let's fork()!



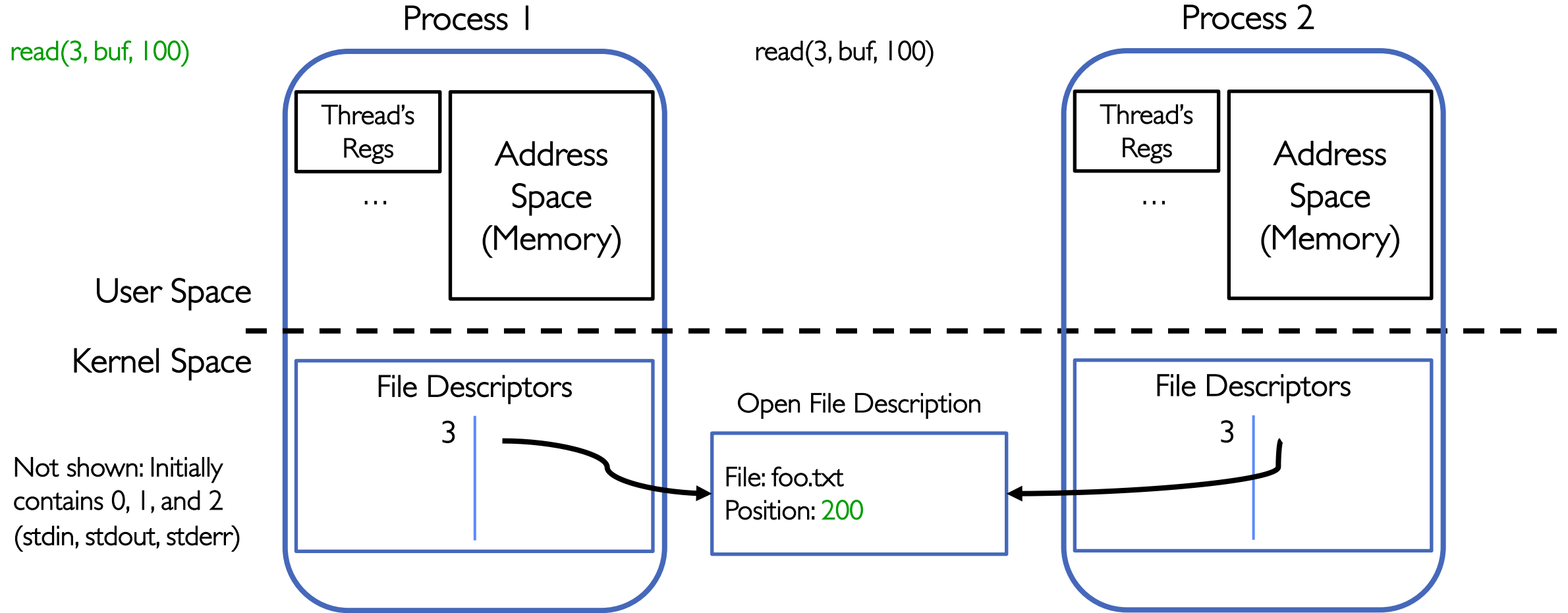
# Open File Description is *Aliased*



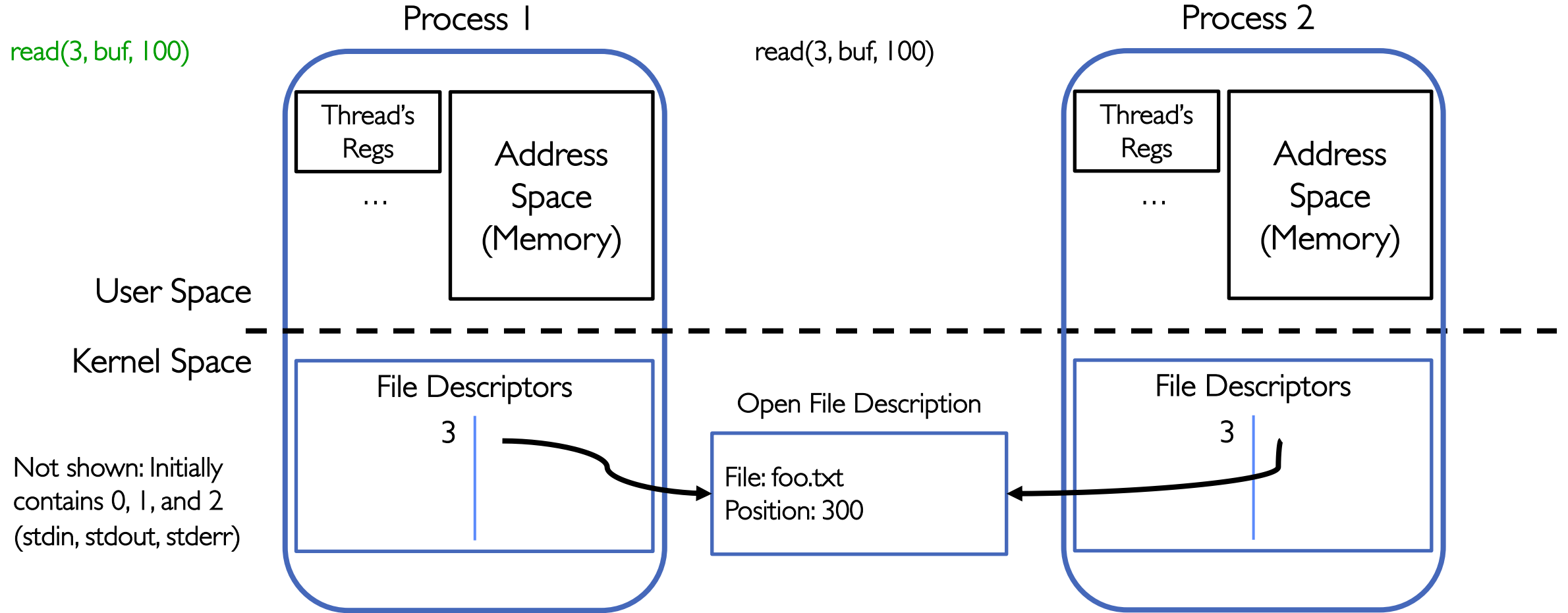
# Open File Description is *Aliased*



# Open File Description is *Aliased*

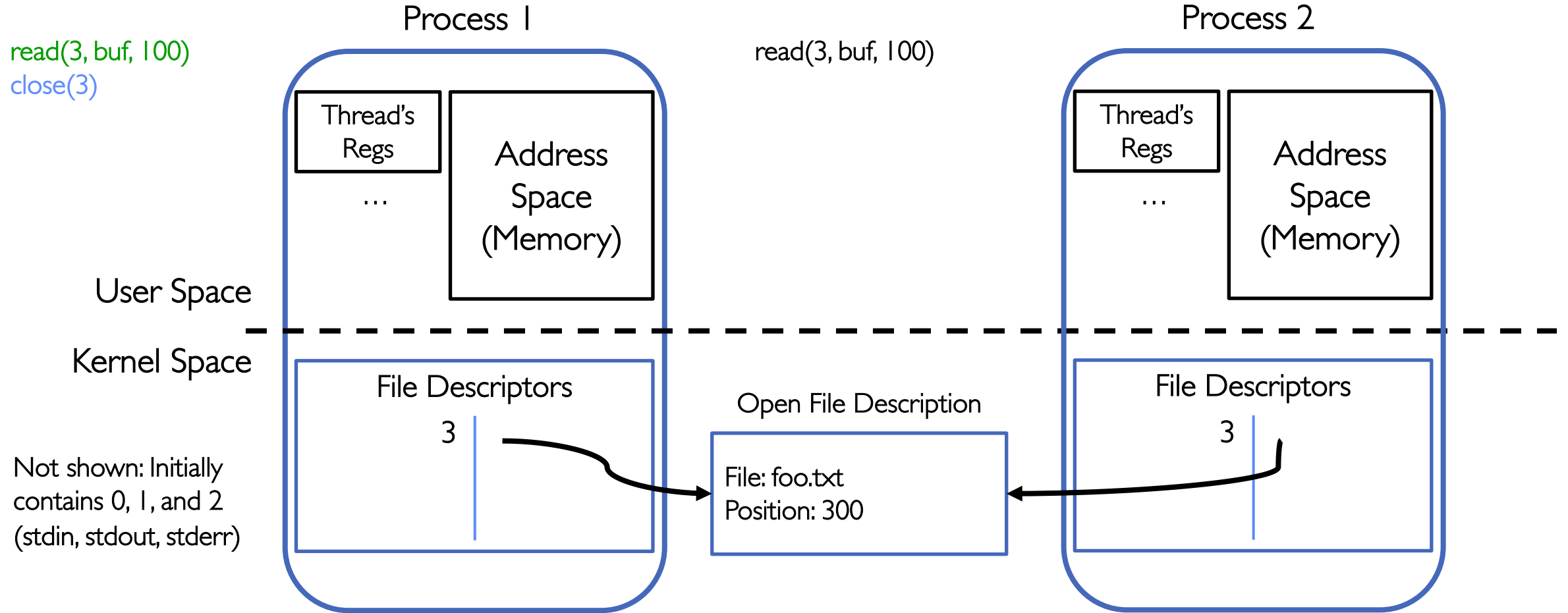


# Open File Description is *Aliased*

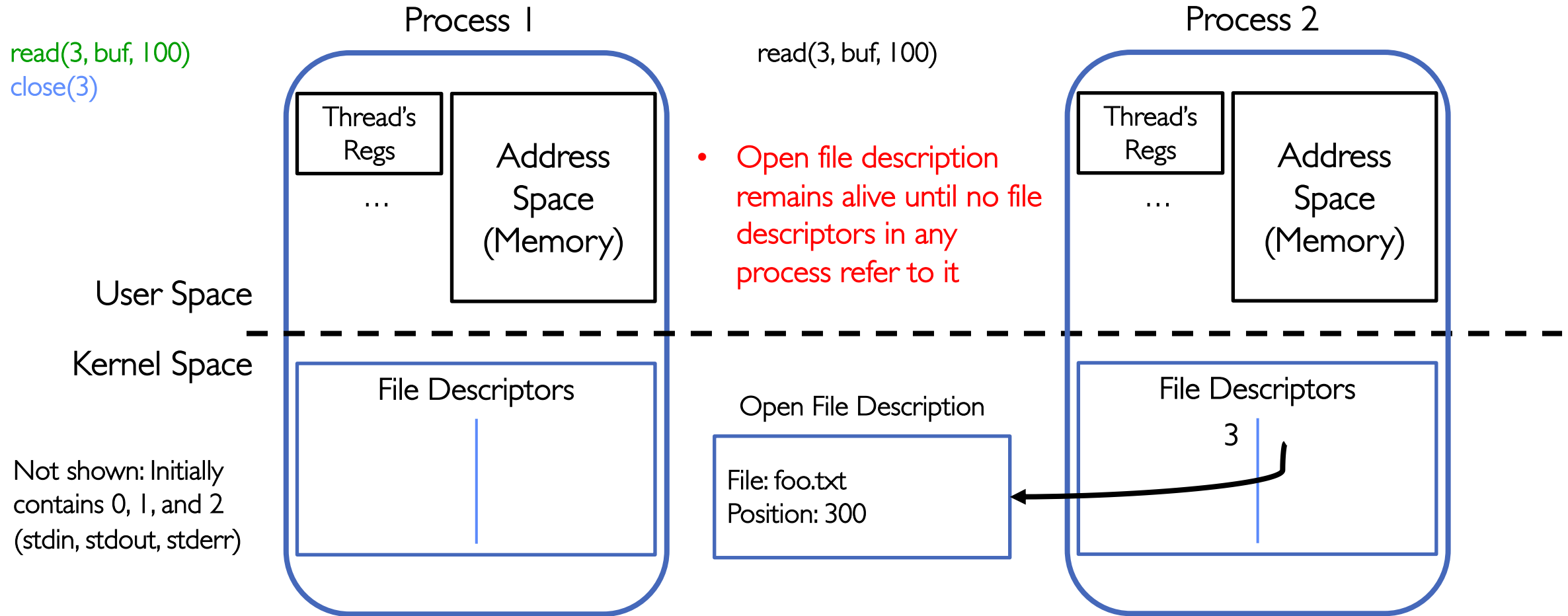




# File Descriptor is *Copied*



# File Descriptor is *Copied*



# Why is Aliasing the Open File Description a Good Idea?

- It allows for *shared resources* between processes

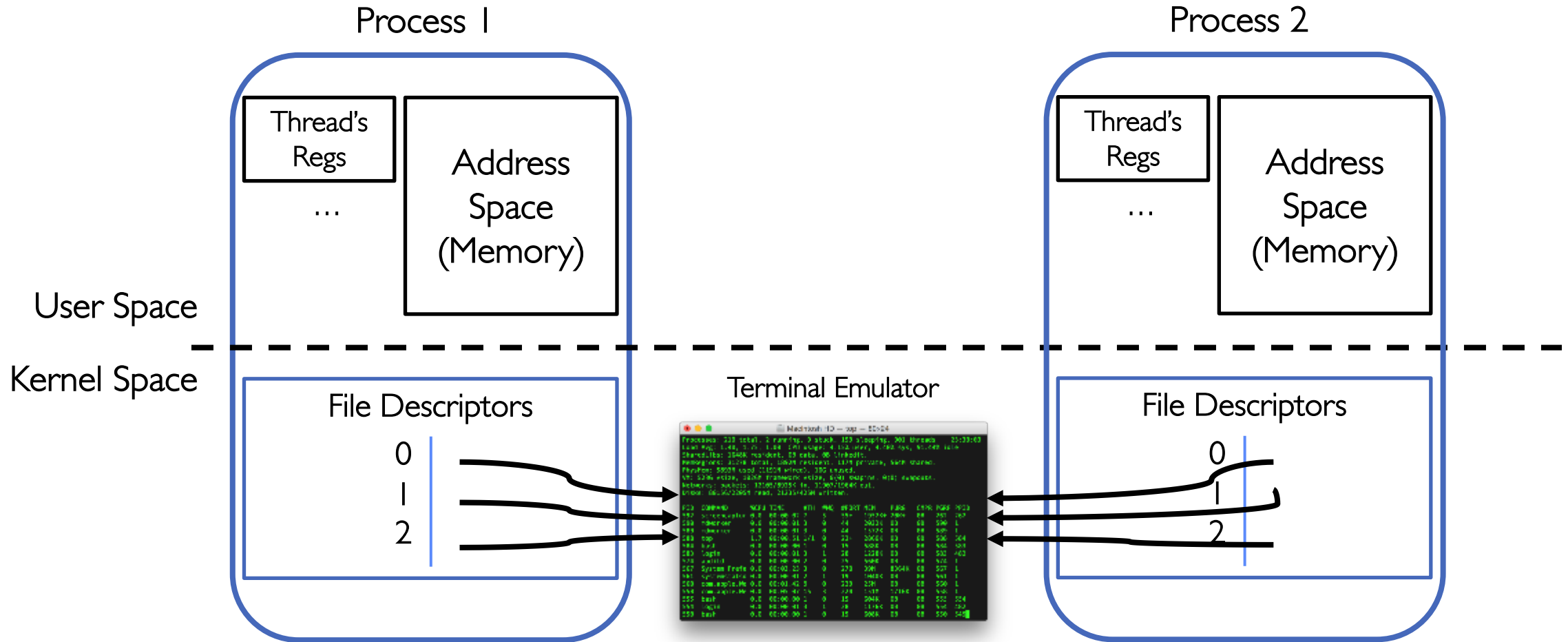
## Recall: In POSIX, Everything is a “File”

- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)
- Based on the system calls **open()**, **read()**, **write()**, and **close()**

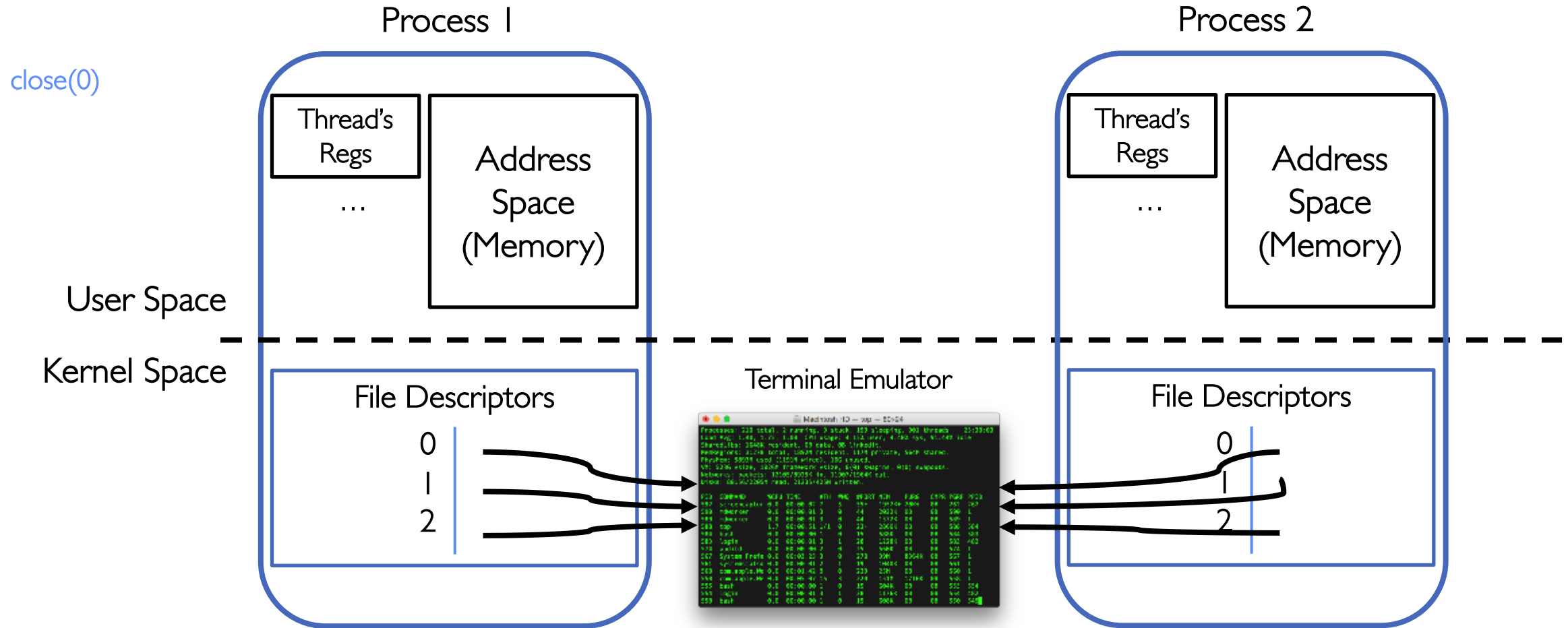
## Example: Shared Terminal Emulator

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

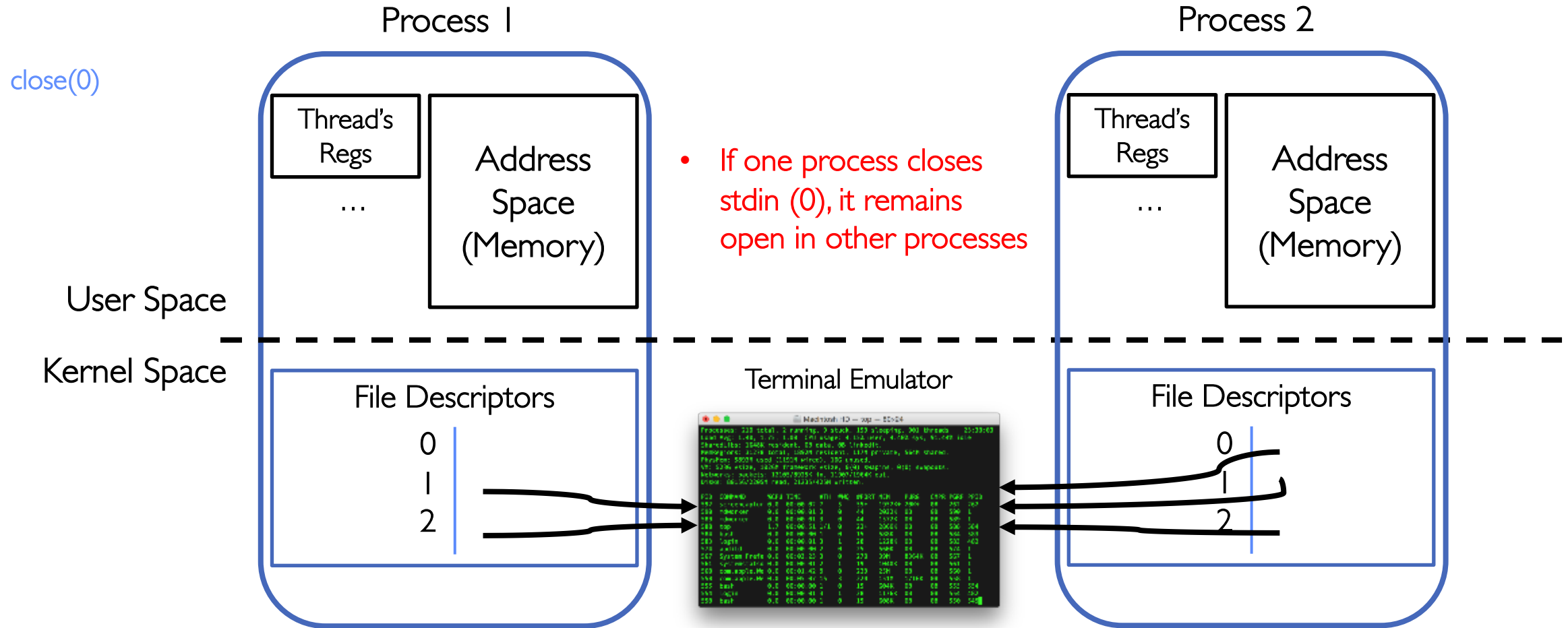
# Example: Shared Terminal Emulator



# Example: Shared Terminal Emulator



# Example: Shared Terminal Emulator





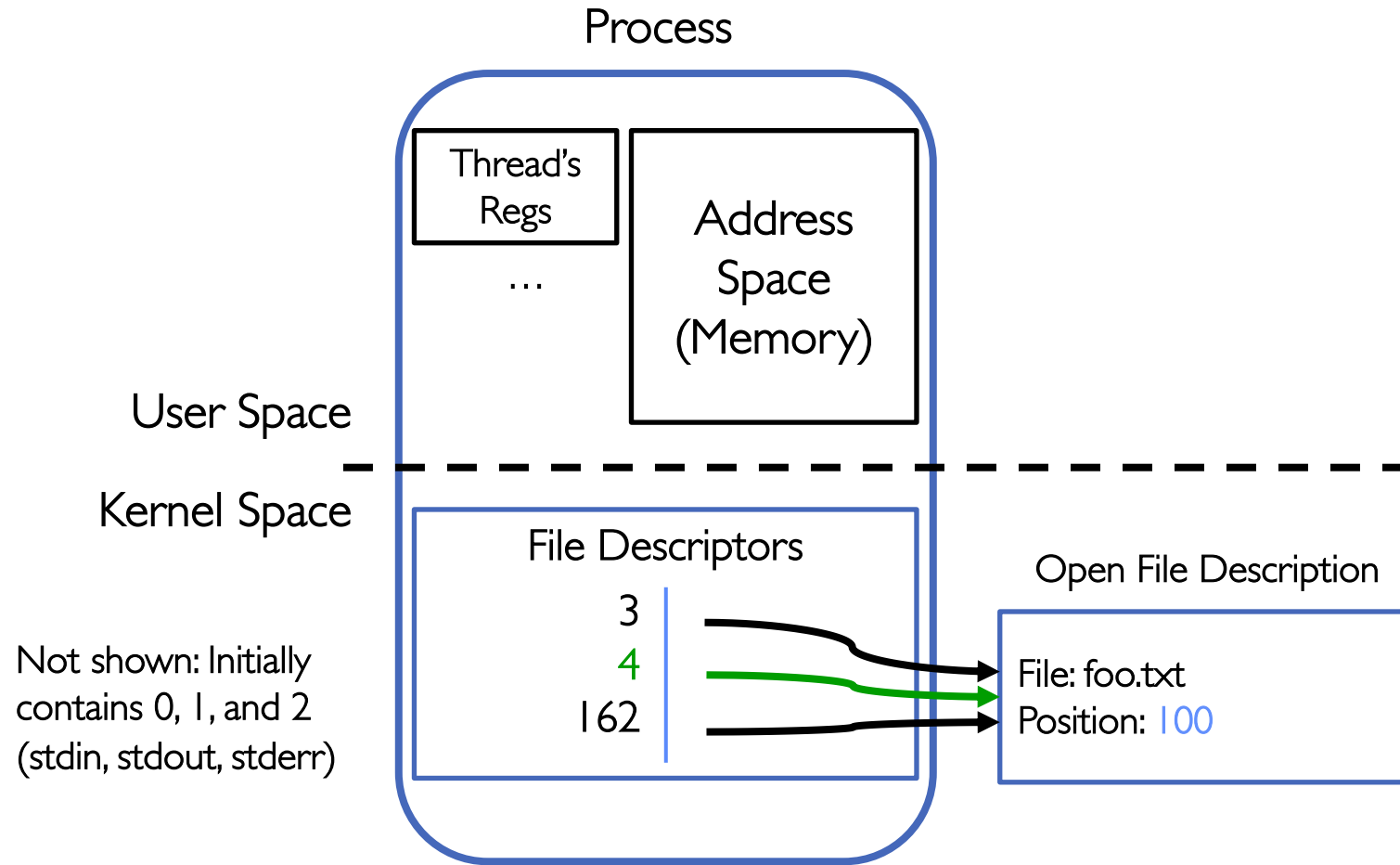
## Other Examples

- Shared network connections after `fork()`
  - Allows handling each connection in a separate process
  - We'll explore this next
- Shared access to pipes
  - Useful for interprocess communication
  - And in writing a shell

## 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)`

# The File Abstraction

- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- *How* and *Why* of High-Level File I/O
- Process State for File Descriptors
- Some Pitfalls with OS Abstractions

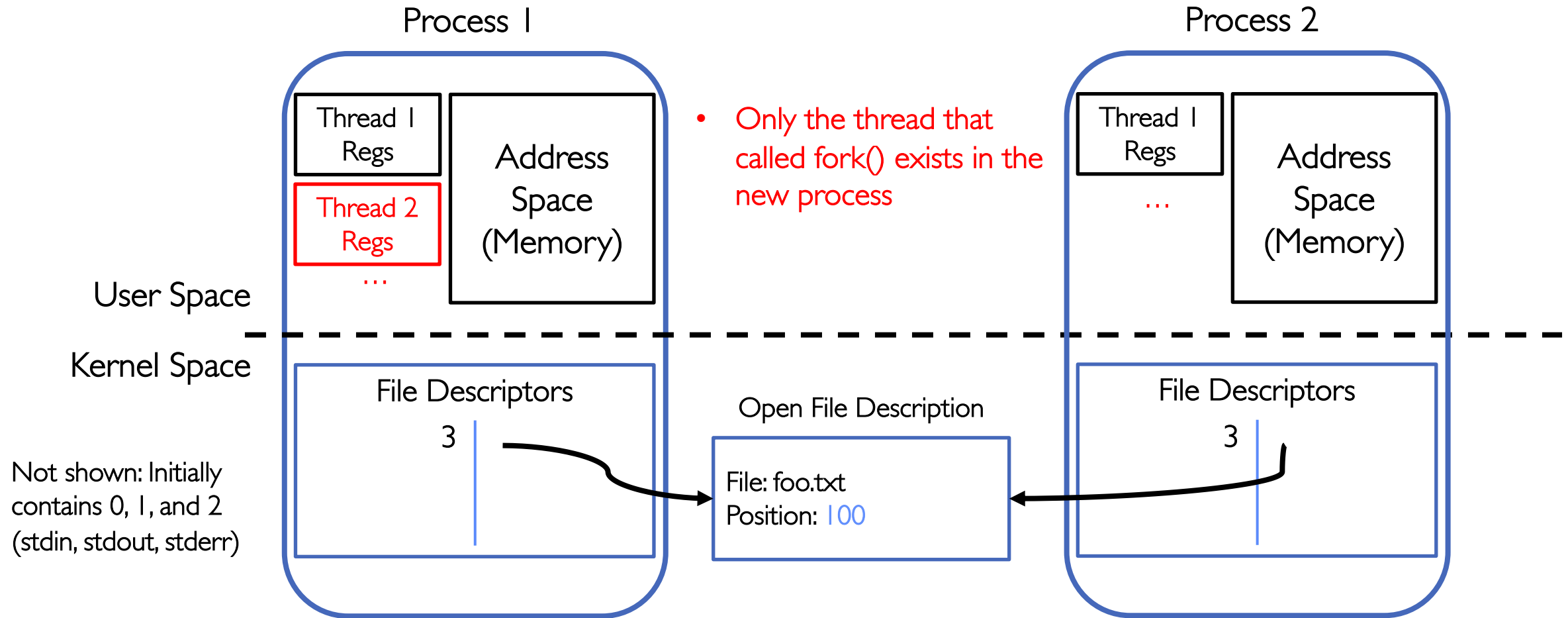
Unless you plan to call `exec()` in the child process

**DON'T FORK() IN A PROCESS THAT ALREADY  
HAS MULTIPLE THREADS**

# fork() in Multithreaded Processes

- The child process always has just a single thread
  - The thread in which fork() was called
- The other threads just vanish

# fork() in a Multithreaded Processes



# Possible Problems with Multithreaded `fork()`

- When you call `fork()` in a multithreaded process, the other threads (the ones that didn't call `fork()`) just vanish
  - What if one of these threads was holding a lock?
  - What if one of these threads was in the middle of modifying a data structure?
  - No cleanup happens!
- It's safe if you call `exec()` in the child
  - Replacing the entire address space



**DON'T CARELESSLY MIX LOW-LEVEL AND HIGH-LEVEL FILE I/O**

# Avoid Mixing FILE\* and File Descriptors

```
char x[10];  
char y[10];  
FILE* f = fopen("foo.txt", "rb");  
int fd = fileno(f);  
fread(x, 10, 1, f); // read 10 bytes from f  
read(fd, y, 10); // assumes that this returns data starting at offset 10
```

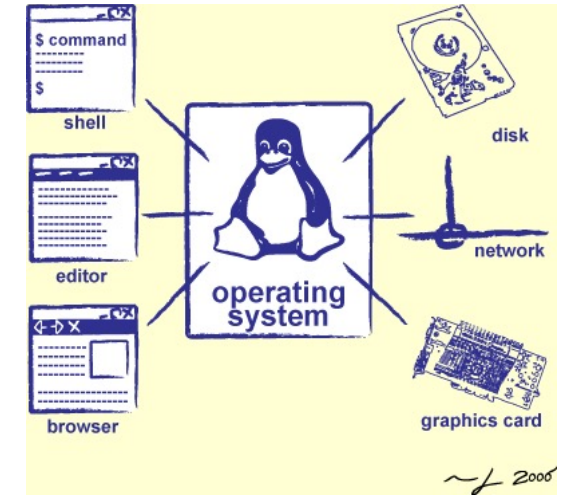
- Which bytes from the file are read into y?
  - A. Bytes 0 to 9
  - B. Bytes 10 to 19
  - C. None of these?
- Answer: C! None of the above.
  - The `fread()` reads a big chunk of file into user-level buffer
  - Might be all of the file!

# Group Discussion

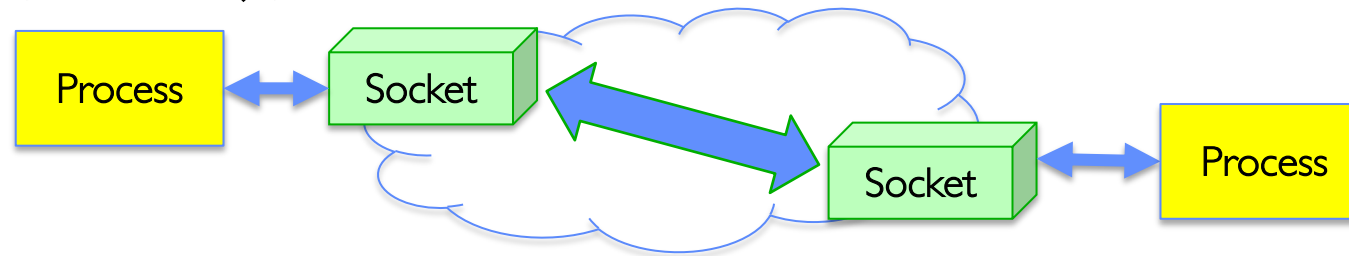
- Topic: High-level vs. low-level File API
  - What are the differences between high-level and low-level file APIs?
  - What are the pros and cons of high-level and low-level file APIs?
  - When to use high-level file API? When to use low-level file API?
- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

# IPC and Sockets

- **Key Idea:** Communication between processes and across the world looks like File I/O
- Introduce Pipes and Sockets
- Introduce TCP/IP Connection setup for Webserver



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

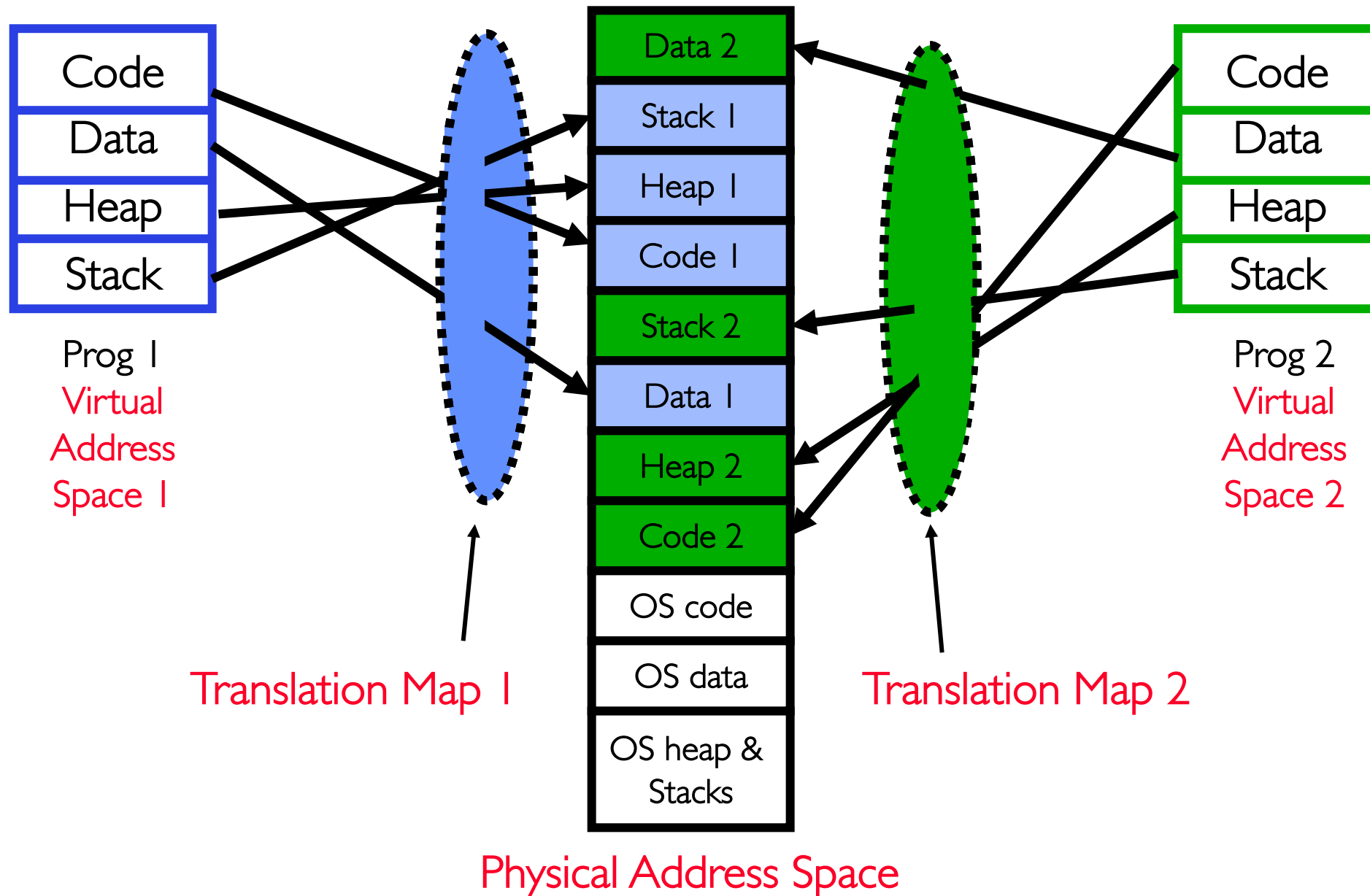


```
n = read(rfd, rbuf, rmax);
```

# Communication Between Processes

- What if processes wish to communicate with one another?
  - Why? Shared Task, Cooperative Venture with Security Implications
- Process Abstraction Designed to Discourage Inter-Process Communication!
  - Prevent one process from interfering with/stealing information from another
- So, must do something special (and agreed upon by both processes)
  - Must “Punch Hole” in security
- This is called “Interprocess Communication” (or IPC)

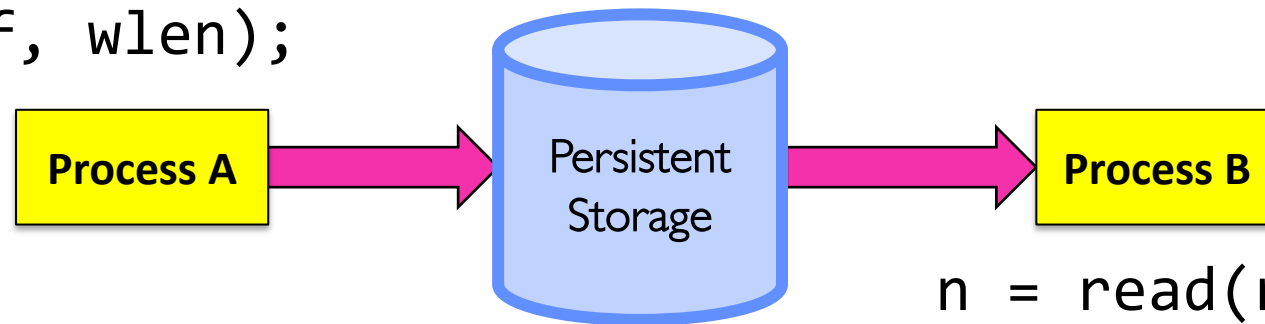
## Recall: Processes Protected from each other



# Communication Between Processes

- Producer (writer) and consumer (reader) may be distinct processes
  - Potentially separated in time
  - How to allow selective communication?
- Simple option: use a file!
  - We have already shown how parents and children share file descriptions:

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



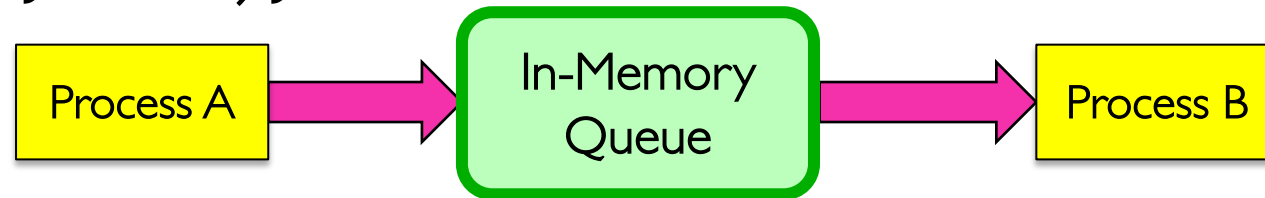
```
n = read(rfd, rbuf, rmax);
```

- Why might this be wasteful?
  - Very expensive if you only want transient communication (non-persistent)

# Communication Between Processes

- Suppose we ask Kernel to help?
  - Consider an in-memory queue
  - Accessed via system calls (for security reasons):

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



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

- Data written by A is held in memory until B reads it
  - Same interface as we use for files!
  - Internally more efficient, since nothing goes to disk
- Some questions:
  - How to set up?
  - What if A generates data faster than B can consume it?
  - What if B consumes data faster than A can produce it?



## One example of this pattern: POSIX/Unix PIPE

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



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

- Memory Buffer is finite:
  - If producer (A) tries to write when buffer full, it *blocks* (Put sleep until space)
  - If consumer (B) tries to read when buffer empty, it *blocks* (Put to sleep until data)

```
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

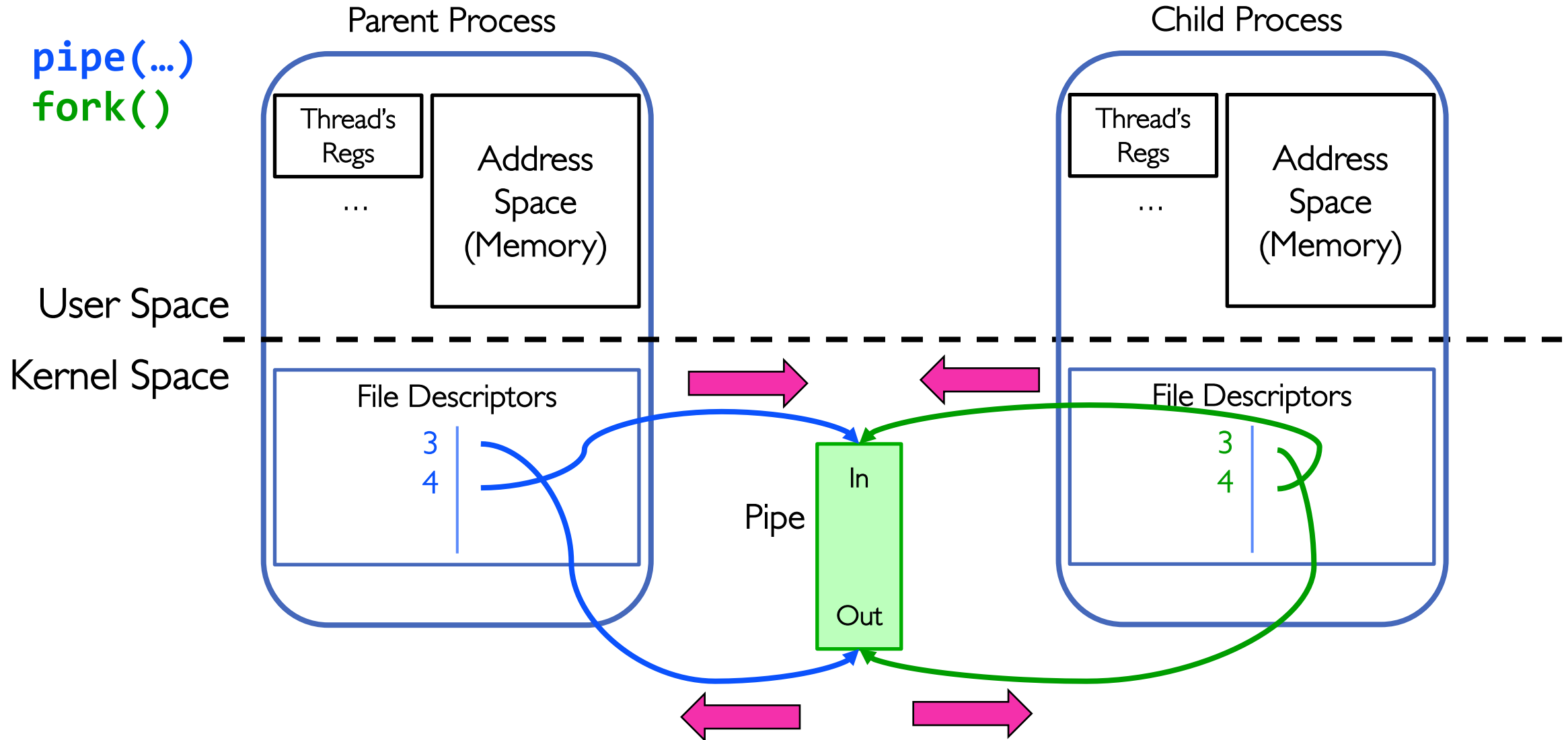
# Single-Process Pipe Example

```
#include <unistd.h>
int main(int argc, char *argv[])
{
    char *msg = "Message in a pipe.\n";
    char buf[BUFSIZE];
    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", buf, readlen);

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

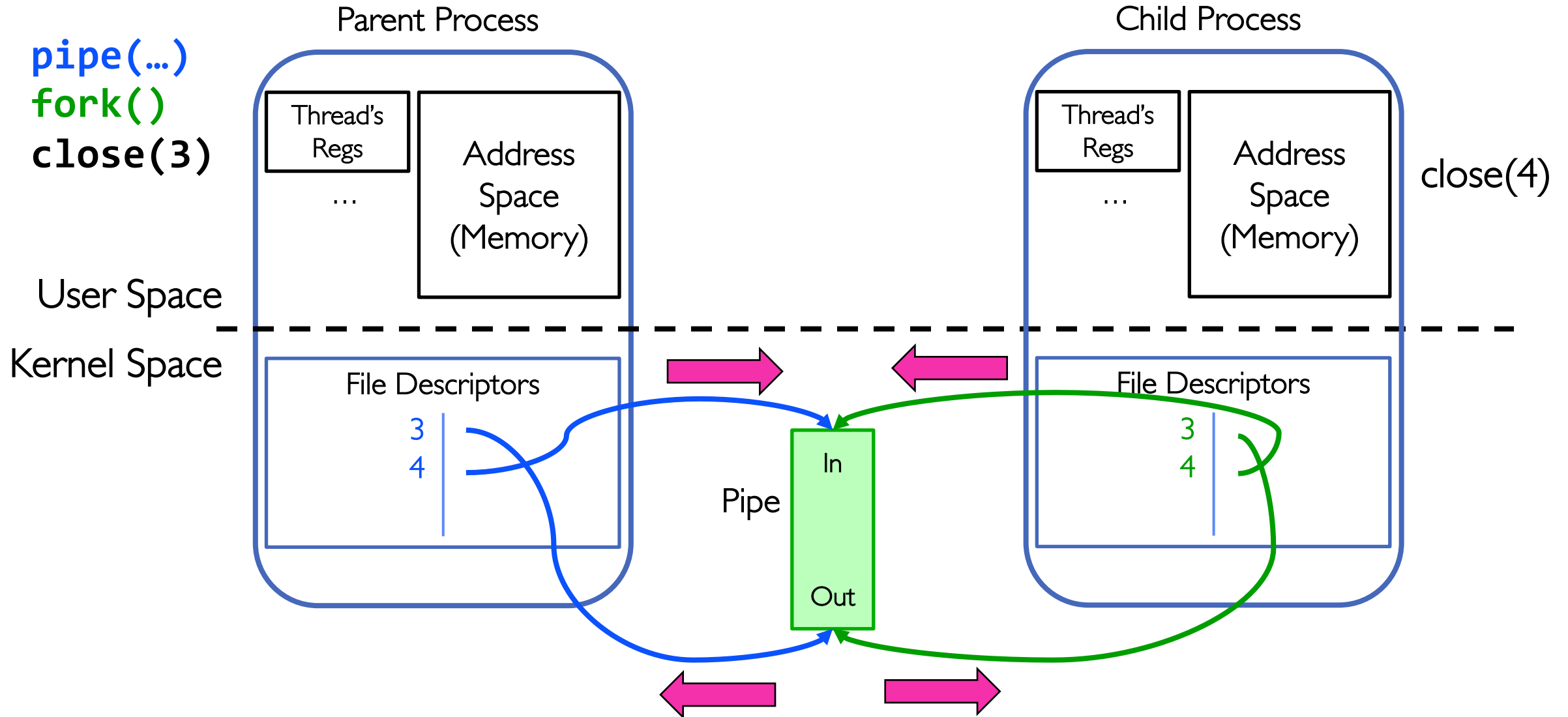
# Pipes *Between* Processes



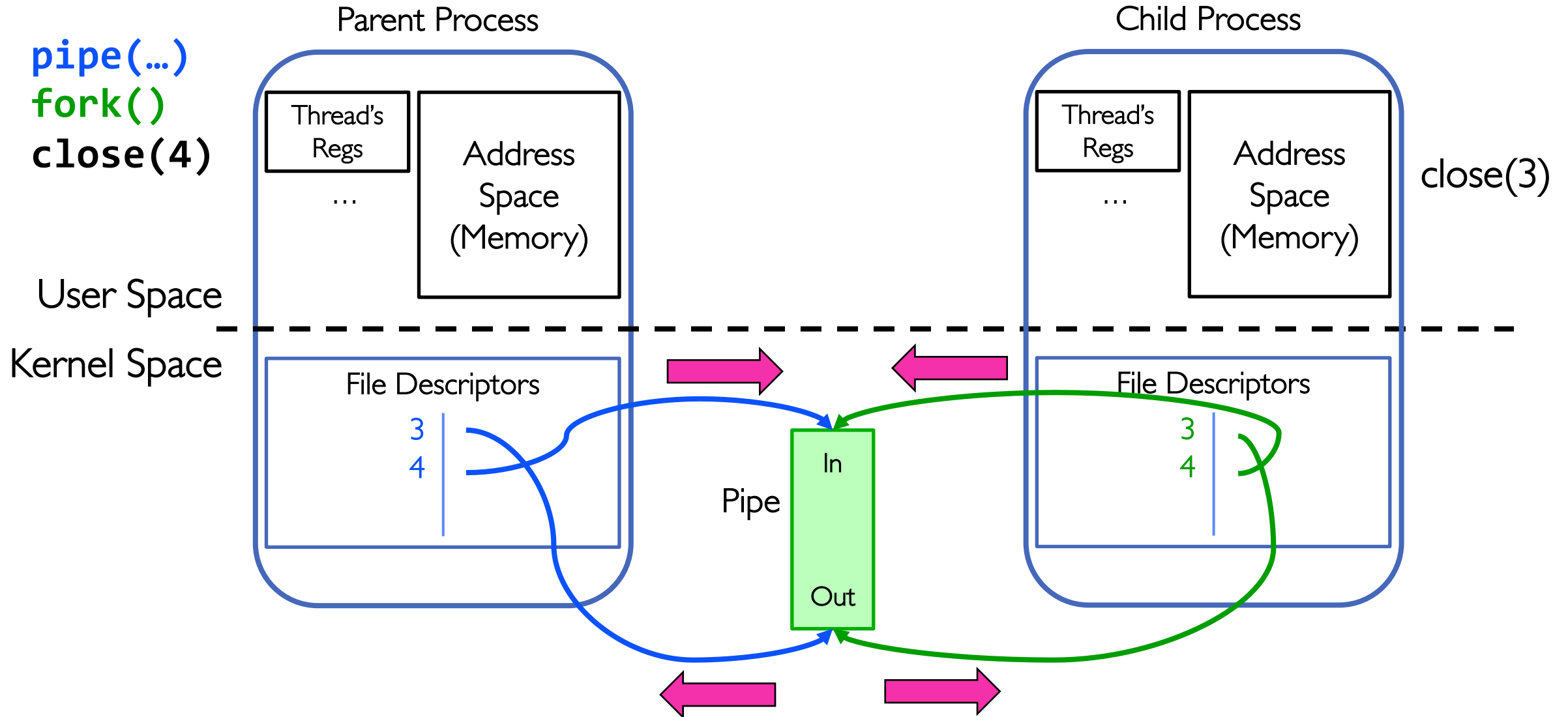
# Inter-Process Communication (IPC): Parent $\Rightarrow$ Child

```
// continuing from earlier
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]);
} else {
    ssize_t readlen = read(pipe_fd[0], buf, BUFSIZE);
    printf("Child Rcvd: %s [%ld]\n", buf, readlen);
    close(pipe_fd[1]);
}
```

# Channel from Parent $\Rightarrow$ Child



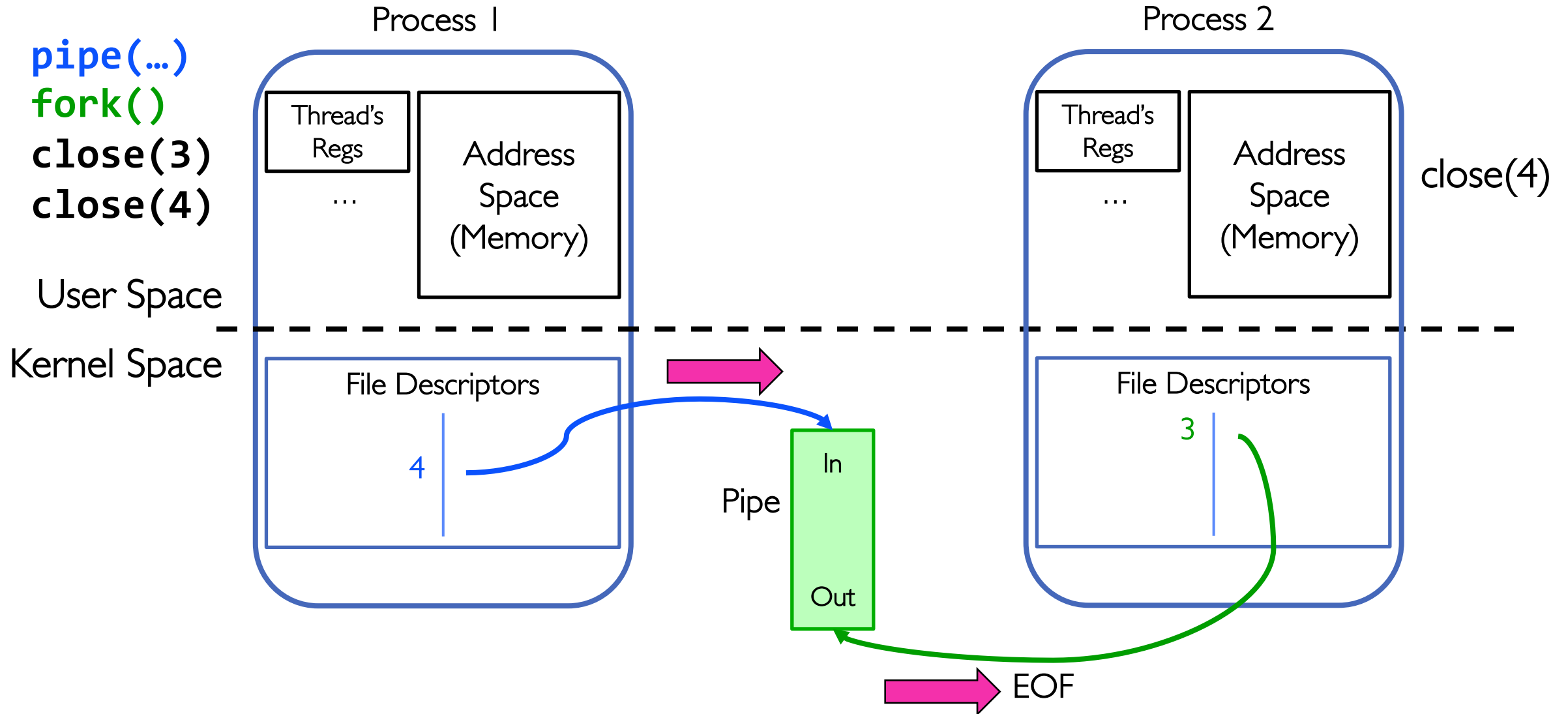
# Instead: Channel from Child $\Rightarrow$ Parent



## When do we get EOF on a pipe?

- After last “write” descriptor is closed, pipe is effectively closed:
  - Reads return only “EOF”
- After last “read” descriptor is closed, writes generate SIGPIPE signals:
  - If process ignores, then the write fails with an “EPIPE” error

# EOF on a Pipe





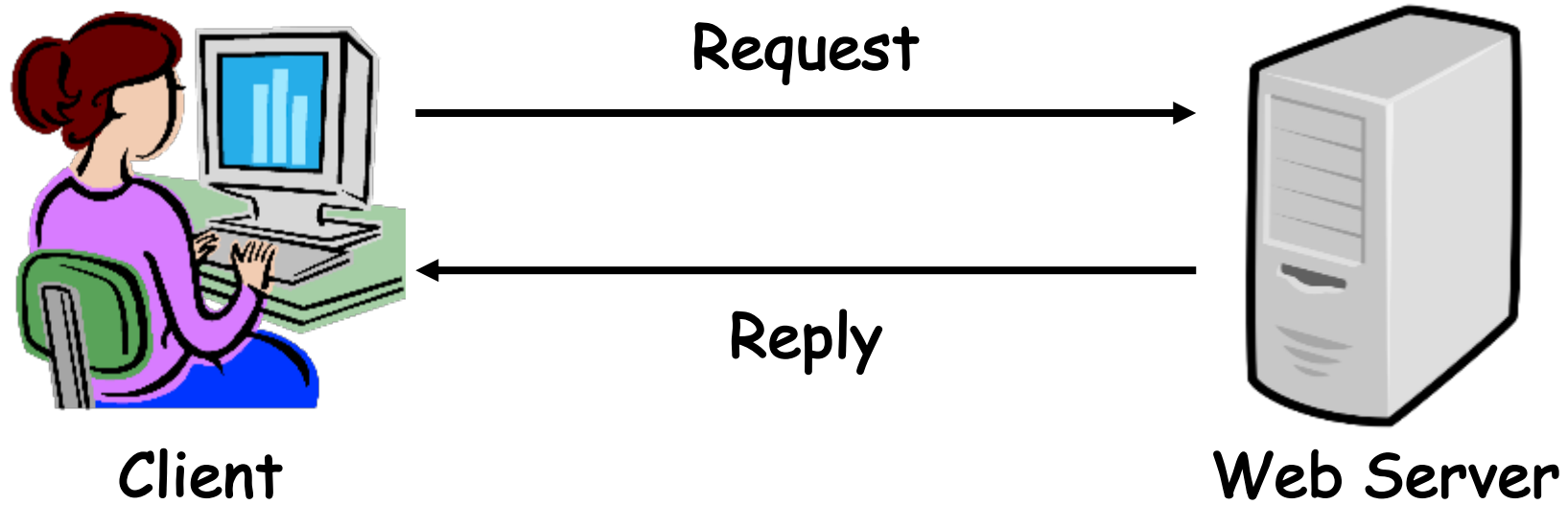
# Once we have communication, we need a *protocol*

- A protocol is an **agreement on how to communicate**
- Includes
  - **Syntax**: how a communication is specified & structured
    - » Format, order messages 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
- In fact, across network may need a way to translate between different representations for numbers, strings, etc.
  - Such translation typically part of a **Remote Procedure Call (RPC)** facility
  - Don't worry about this now, but it is clearly part of the *protocol*

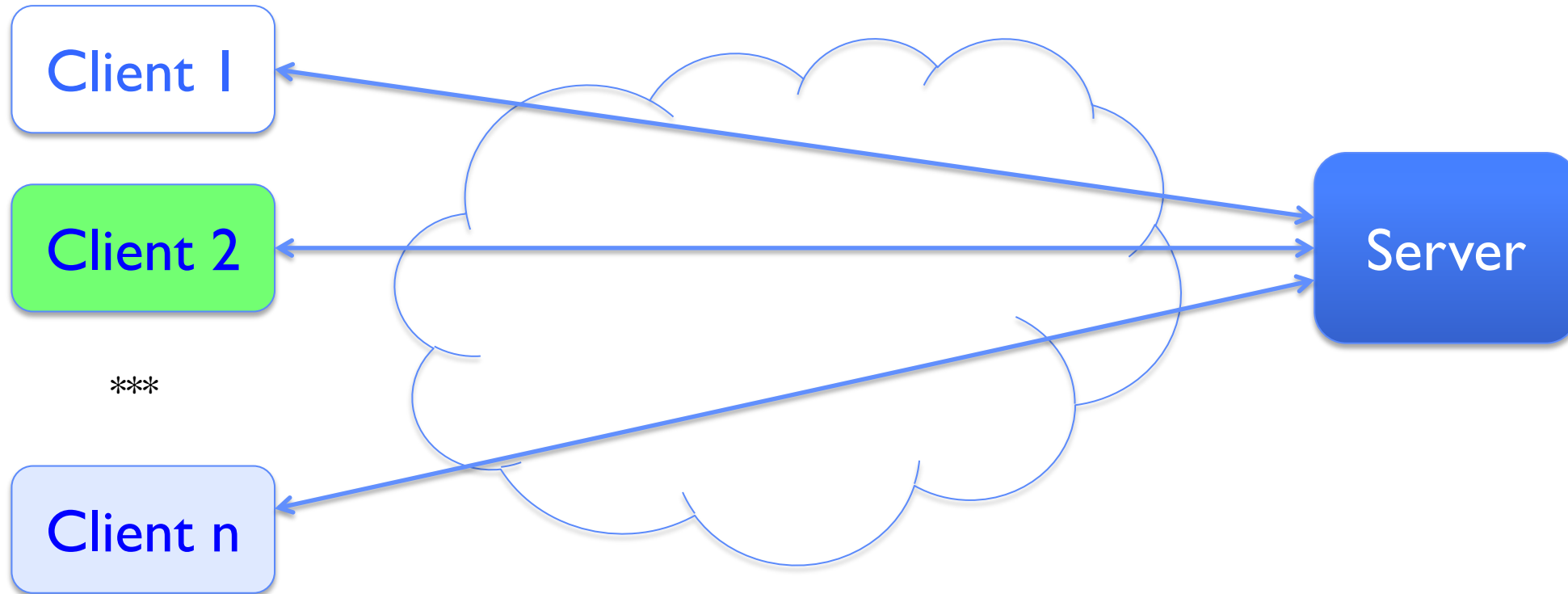
# Examples of Protocols in Human Interaction

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
- 
- ```
graph LR; 5 --> 6; 6 --> 7; 7 --> 8; 8 --> 9; 9 --> 10; 10 --> 11; 11 --> 12
```

# Web Server



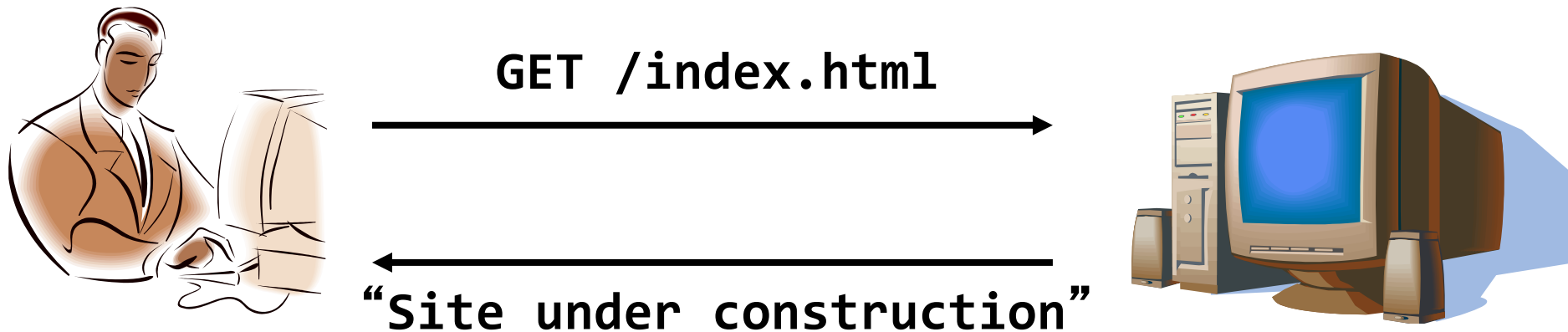
# Client-Server Protocols: Cross-Network IPC



- 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.cnn.com`
  - Doesn’t initiate contact with clients
  - Needs a fixed, well-known address



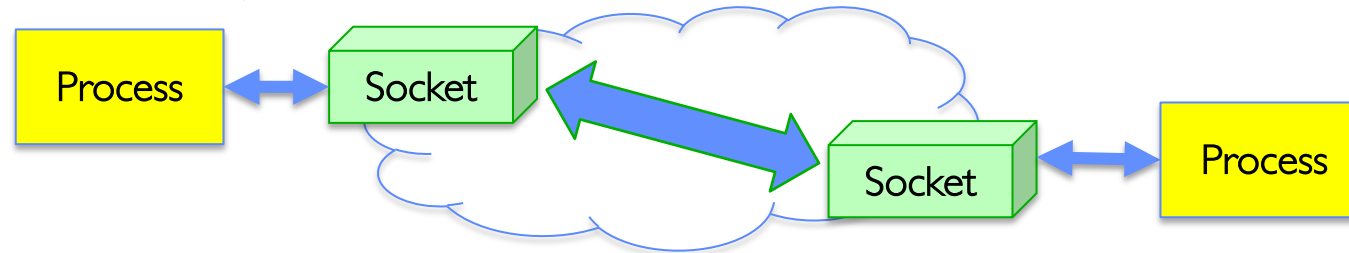
# What is a Network Connection?

- Bidirectional *stream* of bytes between two processes on possibly different machines
  - For now, we are discussing “TCP Connections”
- 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

# 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  $\Rightarrow$  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 Software/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 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

Client (issues requests)

Server (services requests)

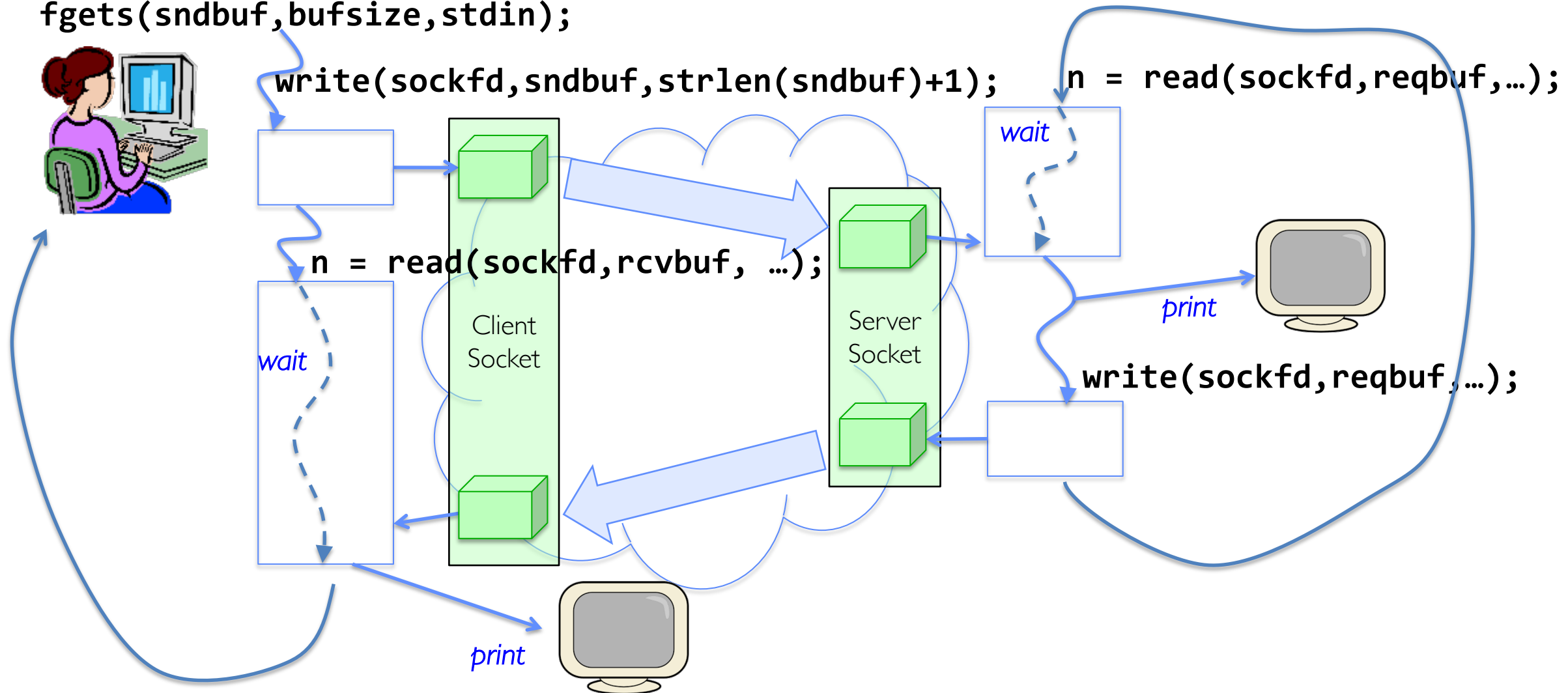
```
fgets(sndbuf, bufsize, stdin);
```

```
write(sockfd, sndbuf, strlen(sndbuf)+1);
```

```
n = read(sockfd, reqbuf, ...);
```

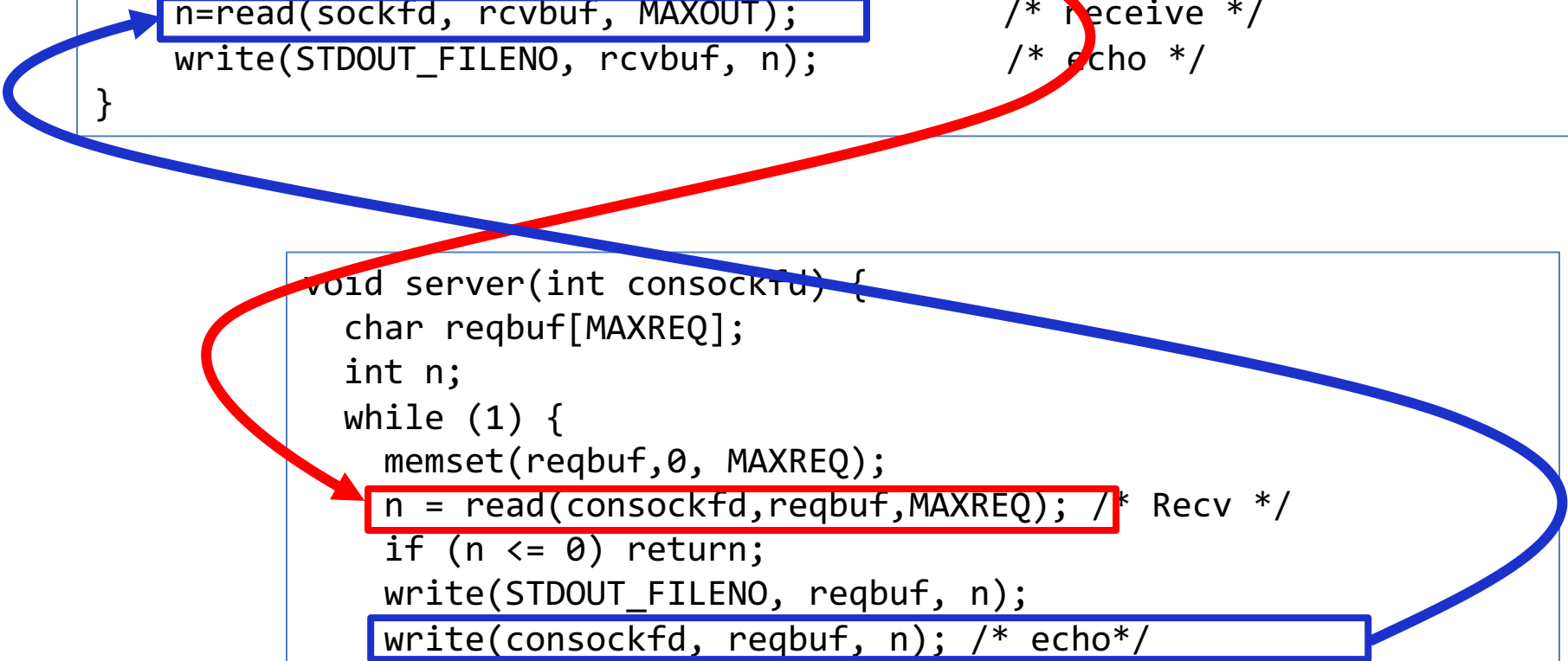
```
n = read(sockfd, rcvbuf, ...);
```

```
write(sockfd, reqbuf, ...);
```



# 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.
  - 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
  - Blocks if nothing has arrived yet
  - Like pipes!

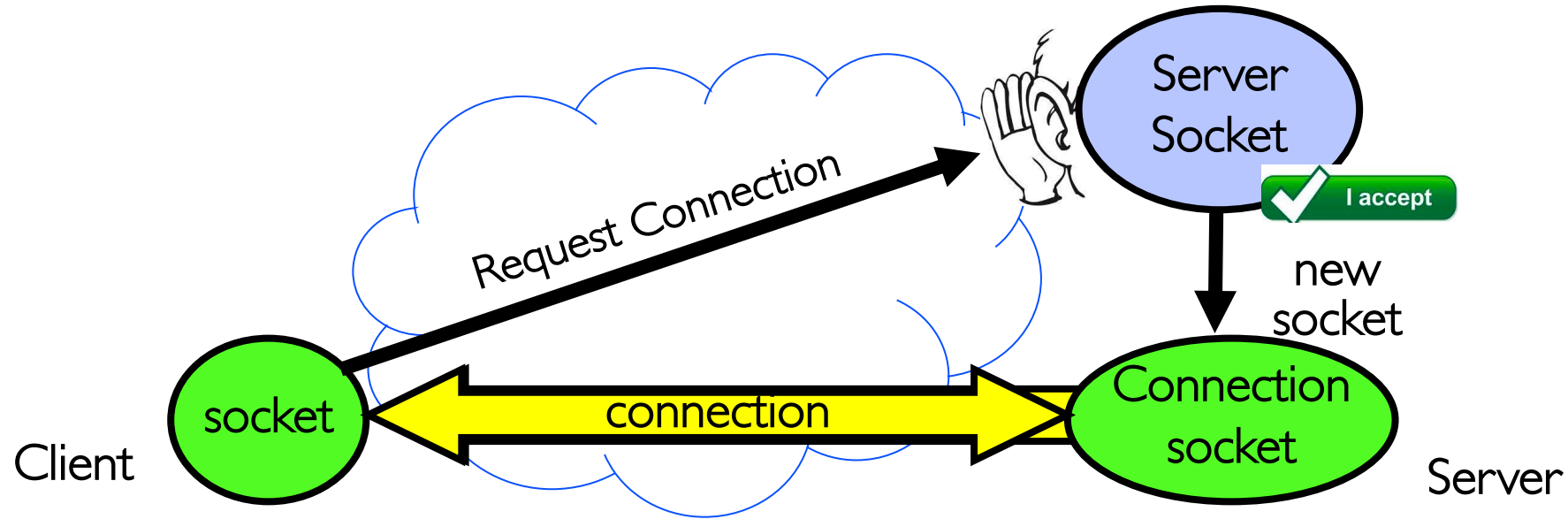
# Socket Creation

- 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
  - Single queue
  - Created transiently by a call to `pipe()`
  - Passed from parent to children (descriptors inherited from parent process)
- 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
  - www.eecs.berkeley.edu
- IP address
  - 128.32.244.172 (IPv4, 32-bit Integer)
  - 2607:f140:0:81:e:f (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”

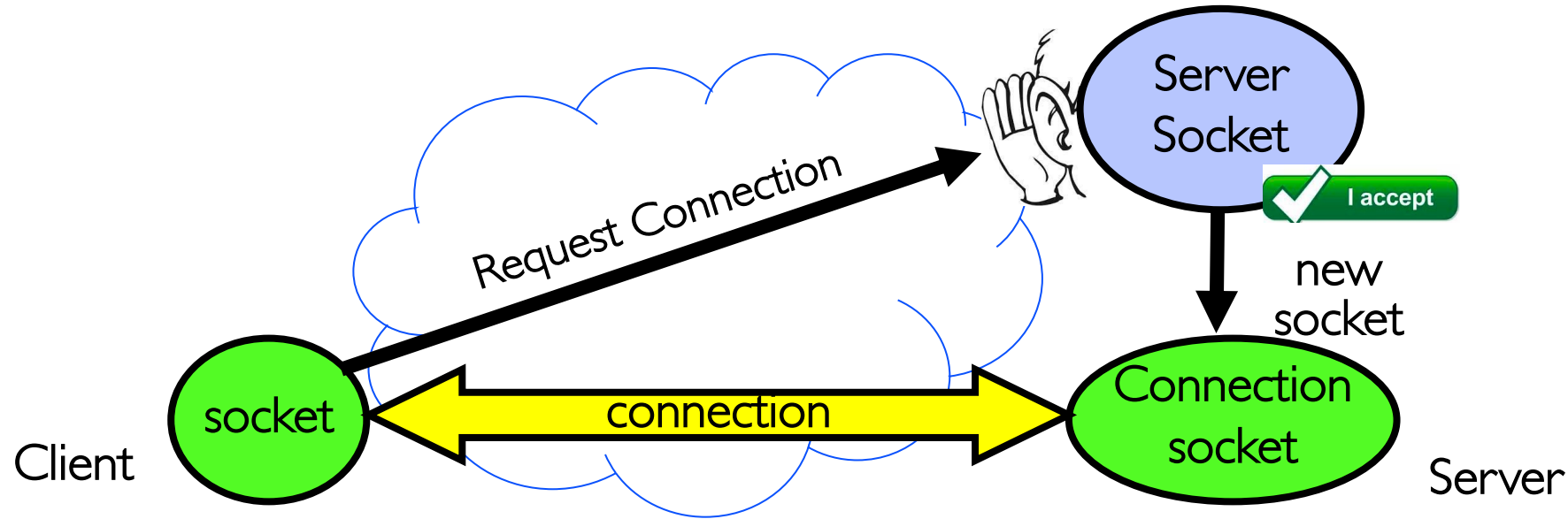
# Connection Setup over TCP/IP



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

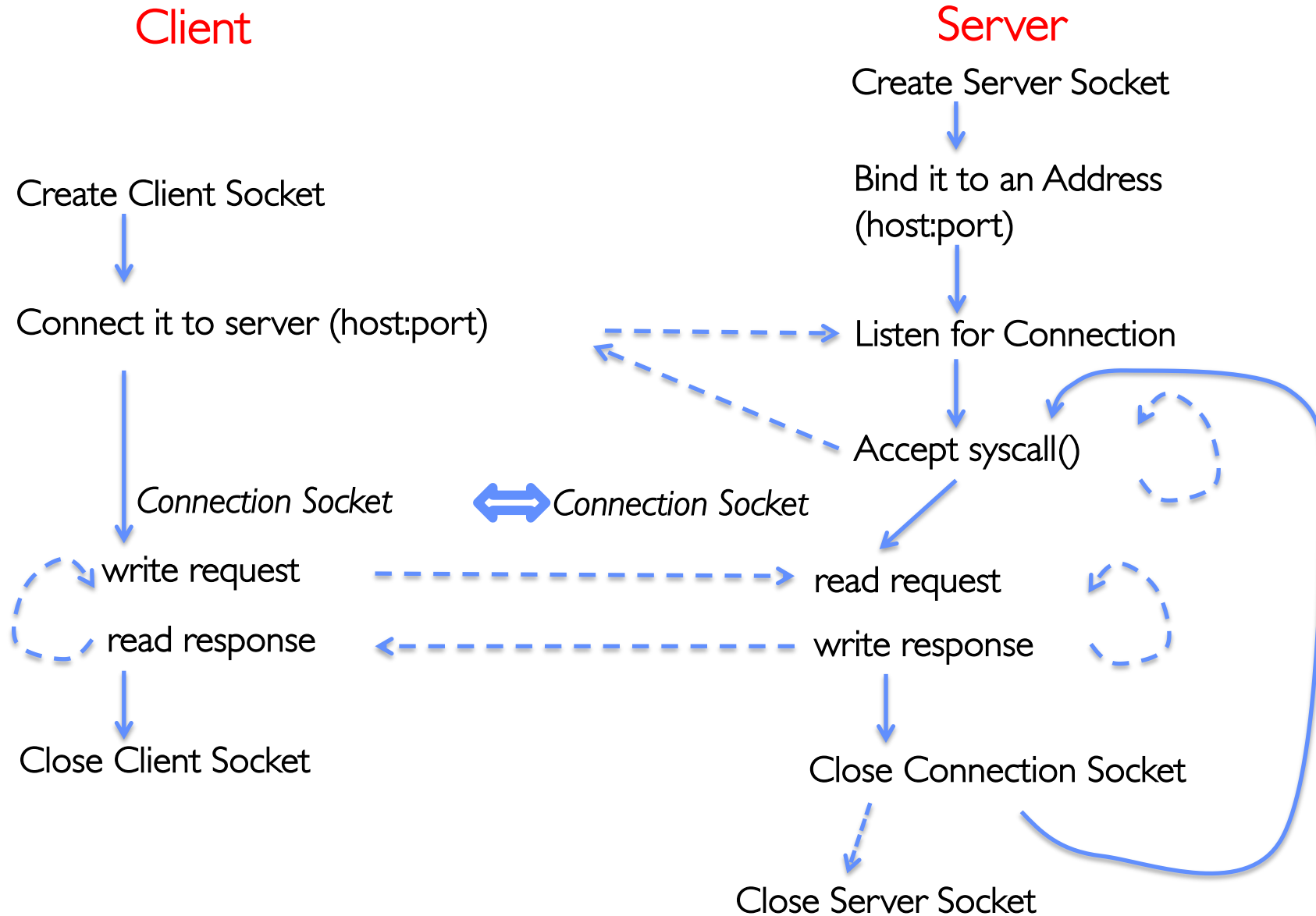


# Connection Setup over TCP/IP



- 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. 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 concept



# Client Protocol

```
char *host_name, *port_name;
```

Address family, e.g.,  
- AF\_INET (IPv4)  
- AF\_INET6 (IPv6)

```
// 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);
```

Protocol type, e.g.,  
- IPPROTO\_TCP  
- 0 (any protocol)

Socket type, e.g.,  
- SOCK\_STREAM  
- SOCK\_DGRAM

```
// 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)

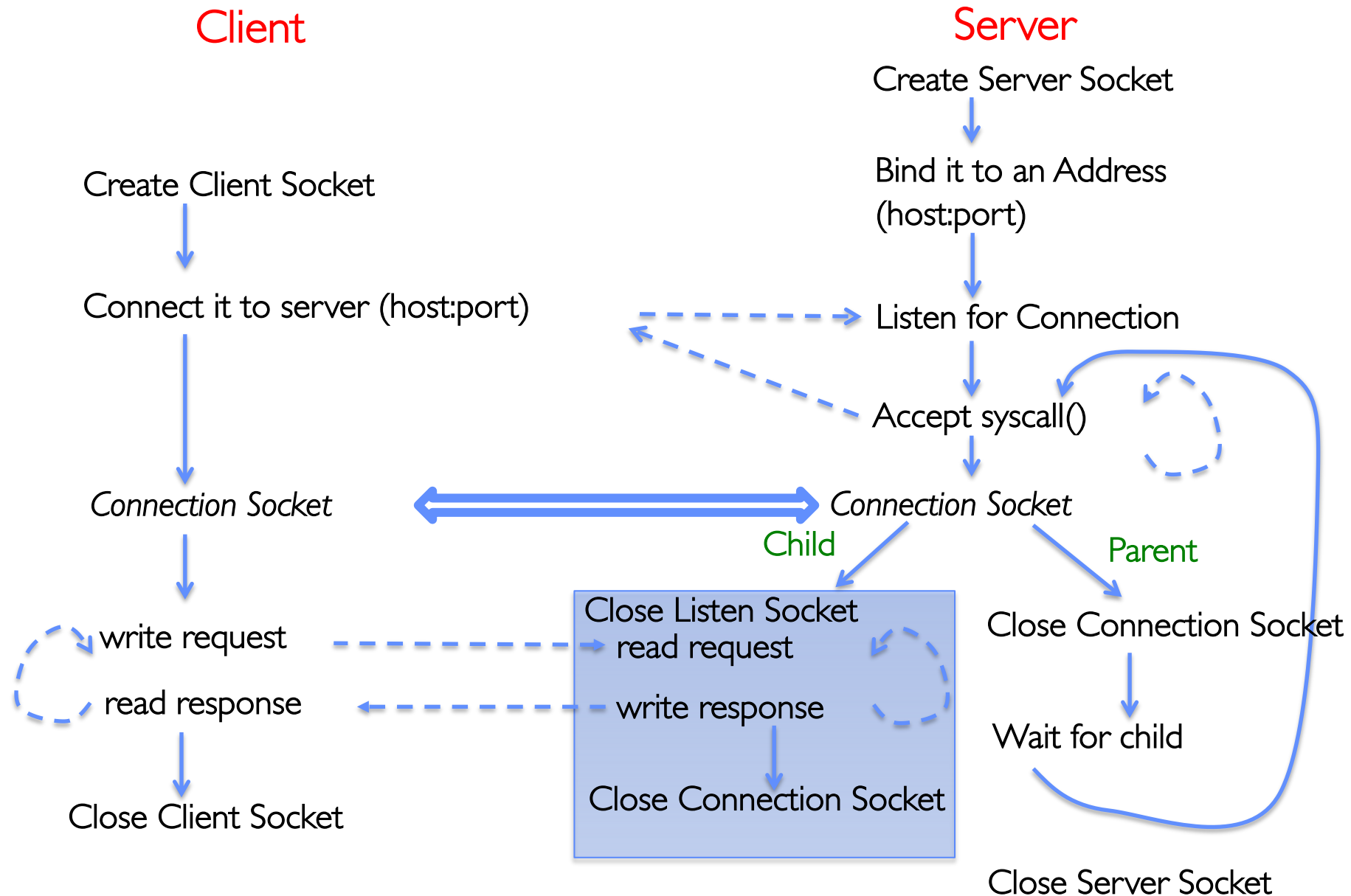
```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family,
                           server->ai_socktype, server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr,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

# How Could the Server Protect Itself?

- Handle each connection in a separate process

# Sockets With Protection (each connection has own process)



## Server Protocol (v2)

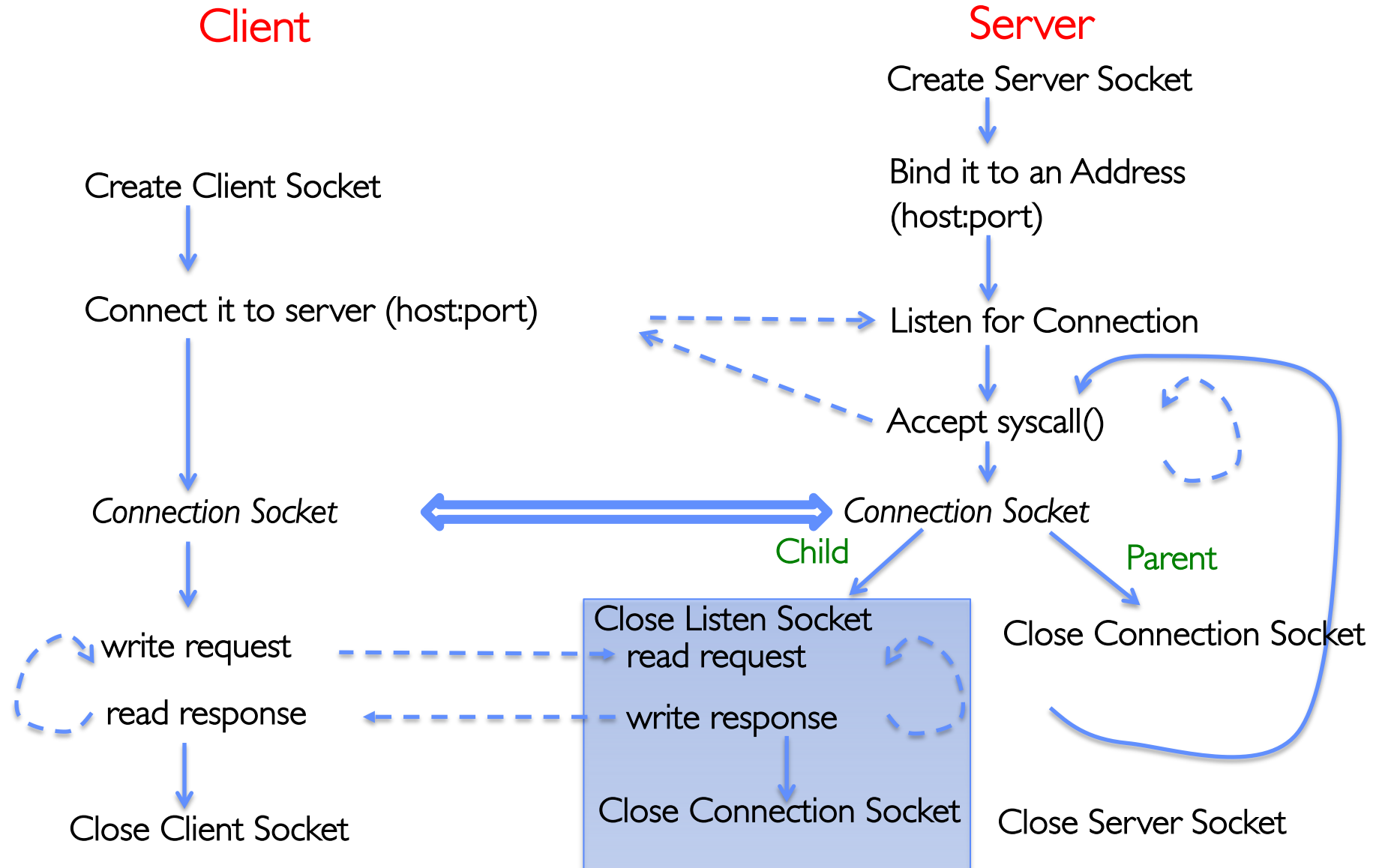
```
// Socket setup code elided...
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    pid_t pid = fork();
    if (pid == 0) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);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



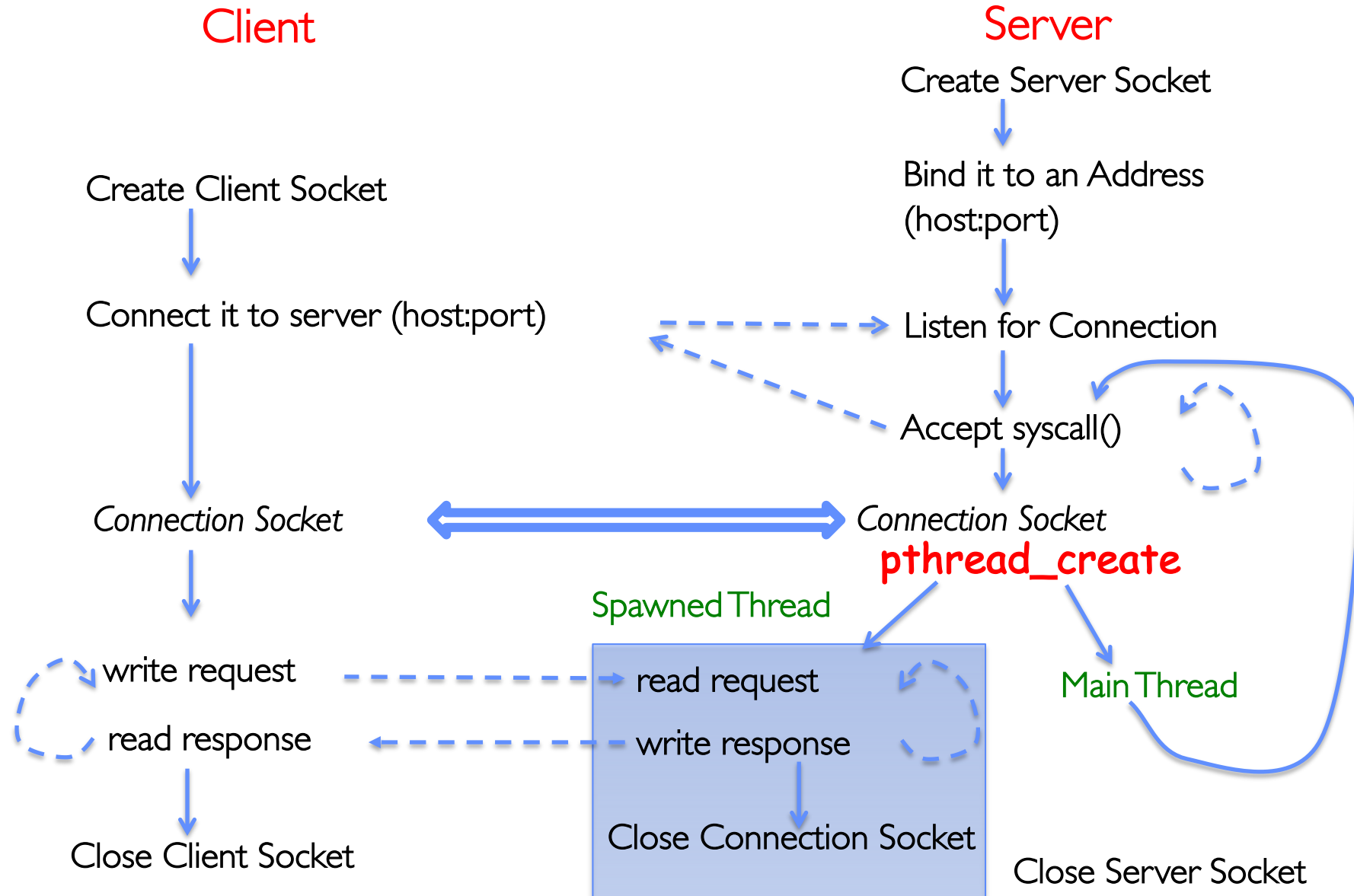
## Server Protocol (v3)

```
// 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) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        //wait(NULL);
    }
}
close(server_socket);
```

# Concurrent Server without Protection

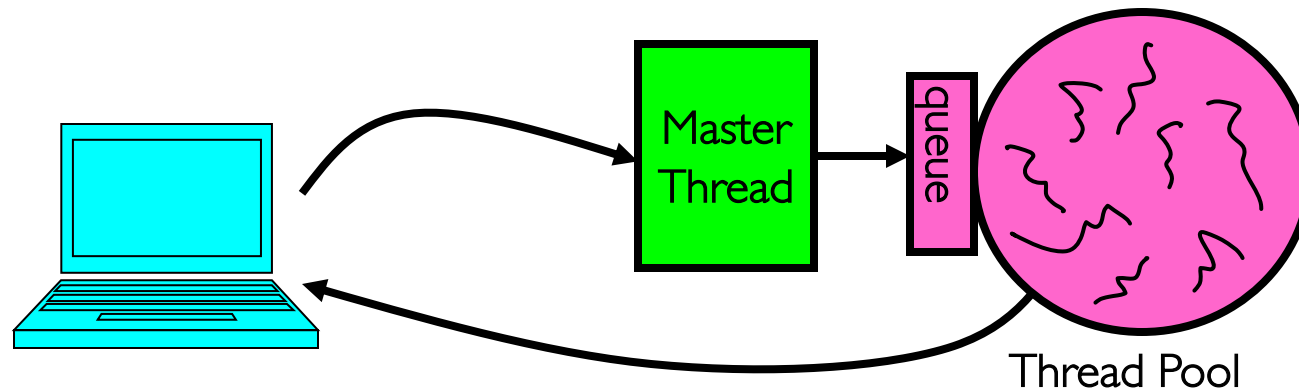
- Spawn a new thread to handle each connection
- Main thread initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient to switch between threads

# Sockets with Concurrency, without Protection



# Thread Pools

- 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(TRUE) {  
        con=AcceptCon();  
        Enqueue(queue, con);  
        wakeUp(queue);  
    }  
}
```

```
worker(queue) {  
    while(TRUE) {  
        con=Dequeue(queue);  
        if (con==null)  
            sleepOn(queue);  
        else  
            ServiceWebPage(con);  
    }  
}
```

# Group Discussion

- Topic: Pipes vs. Sockets
  - What is a pipe? What is a socket?
  - What are similar between pipes and sockets?
  - What are different between pipes and sockets?
- Discuss in groups of two to three students
  - Each group chooses a leader to summarize the discussion
  - In your group discussion, please do not dominate the discussion, and give everyone a chance to speak

# Summary

- Interprocess Communication (IPC)
  - Communication facility between protected environments (i.e. processes)
- 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