# CS162
# Operating Systems and Systems Programming
# Lecture 6

## Abstractions 4: Sockets, I/O, IPC (finished)

September 17th, 2024

Yi Xu

http://cs162.eecs.Berkeley.edu

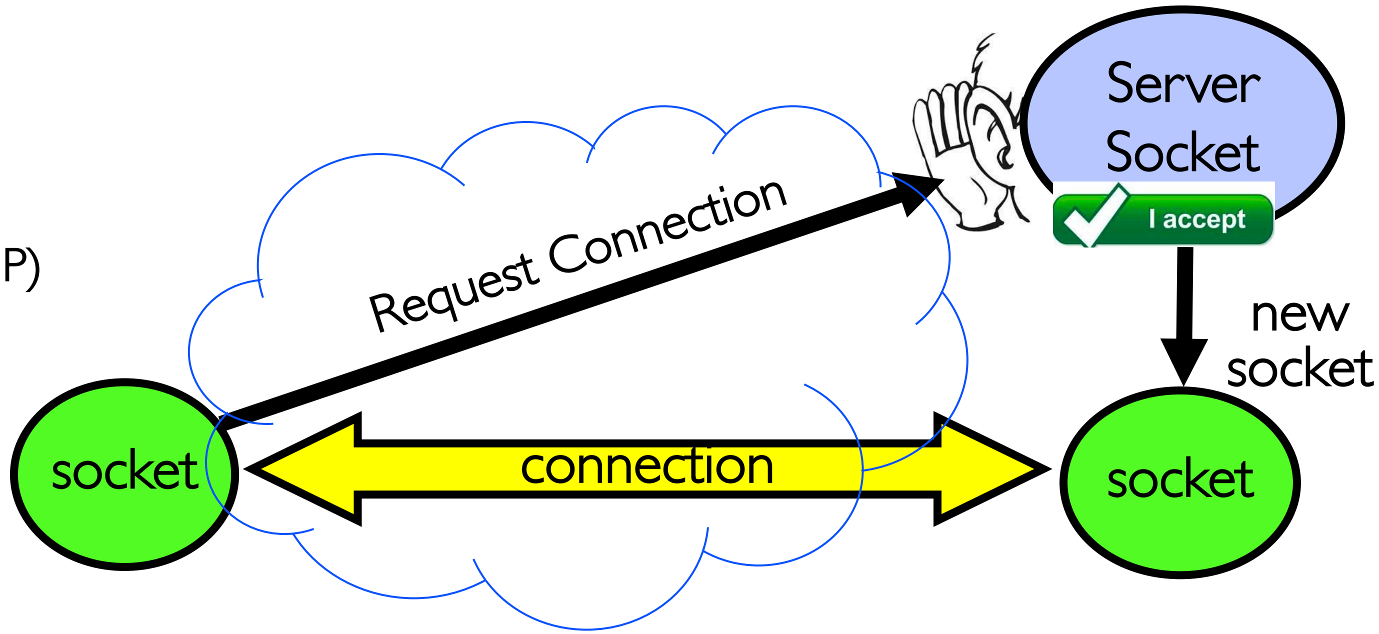
# Recall: Connection Setup over TCP/IP

Client Side

Connection request:
1. Client IP addr
2. Client Port
3. Protocol (TCP/IP)

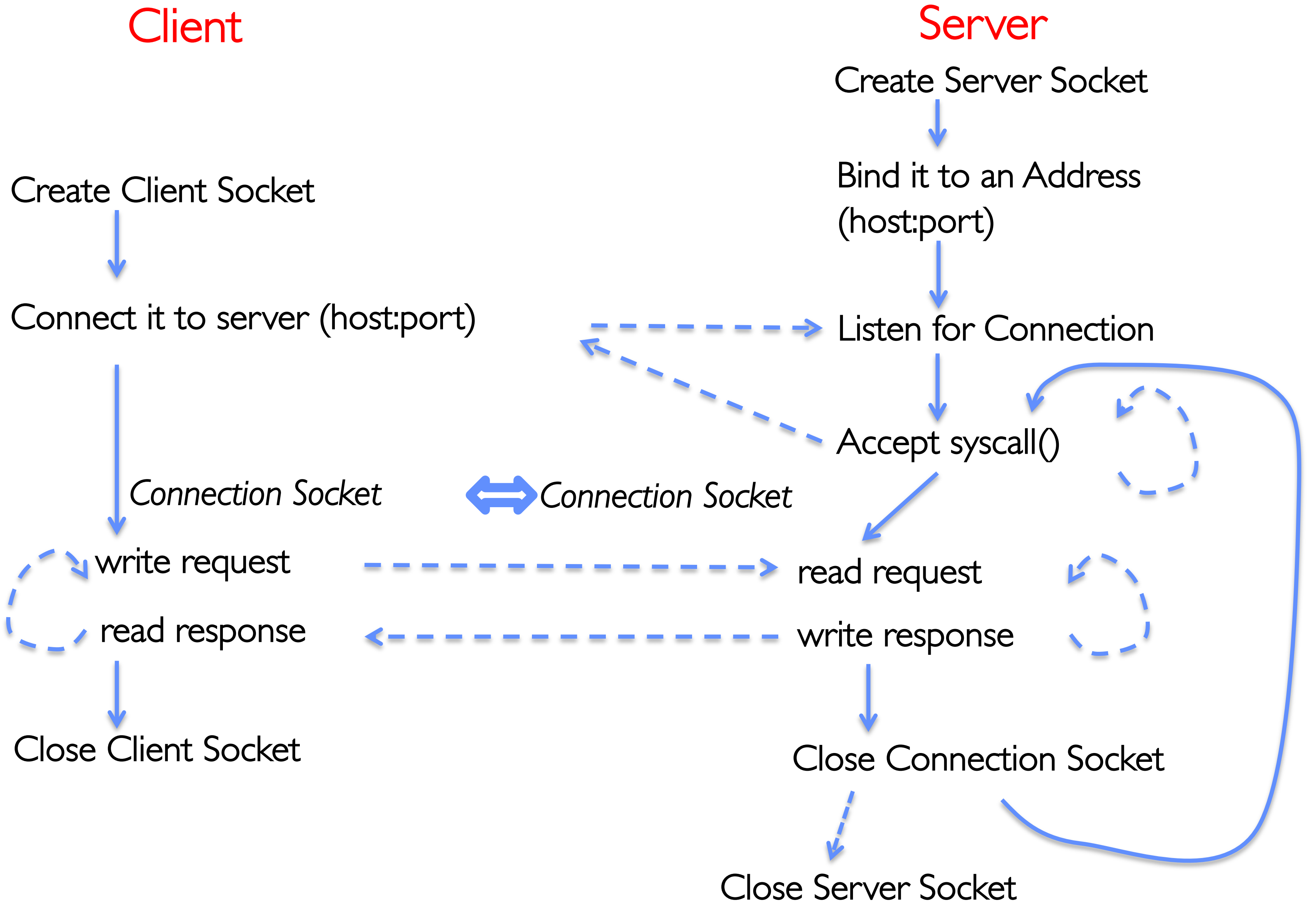
Server Side

Server Listening:
1. Server IP addr
2. well-known port,
3. Protocol (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

# Recall: 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(sock_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) {
  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

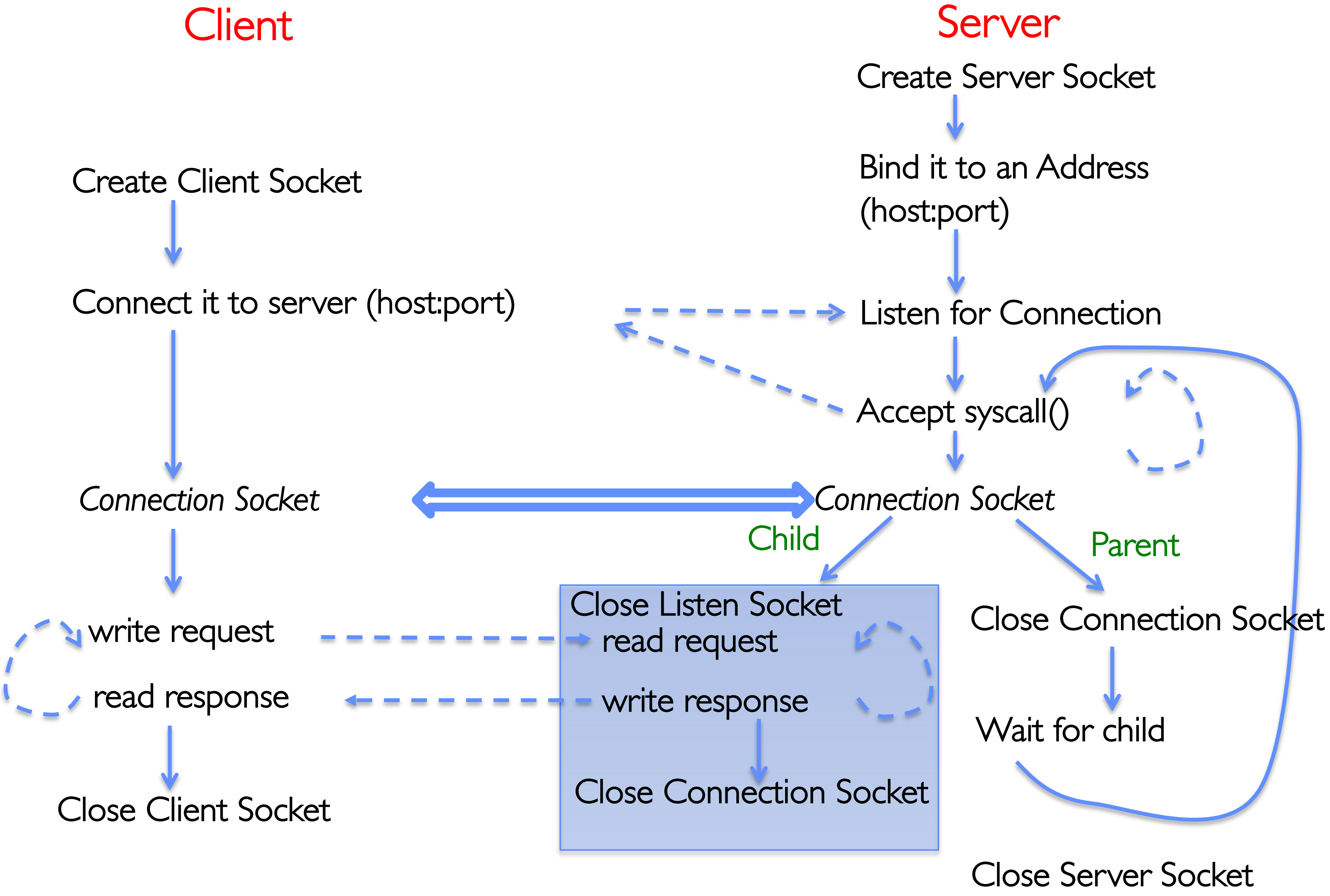
```
struct addrinfo *setup_address(char *port) {
  struct addrinfo *server;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = AF_UNSPEC;           /* Includes AF_INET and AF_INET6 */
  hints.ai_socktype = SOCK_STREAM;       /* Essentially TCP/IP */
  hints.ai_flags = AI_PASSIVE;           /* Set up for server socket */

  int rv = getaddrinfo(NULL, port, &hints, &server);  /* No address! (any local IP) */
  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 connection has own process)



# Server Code (v2)

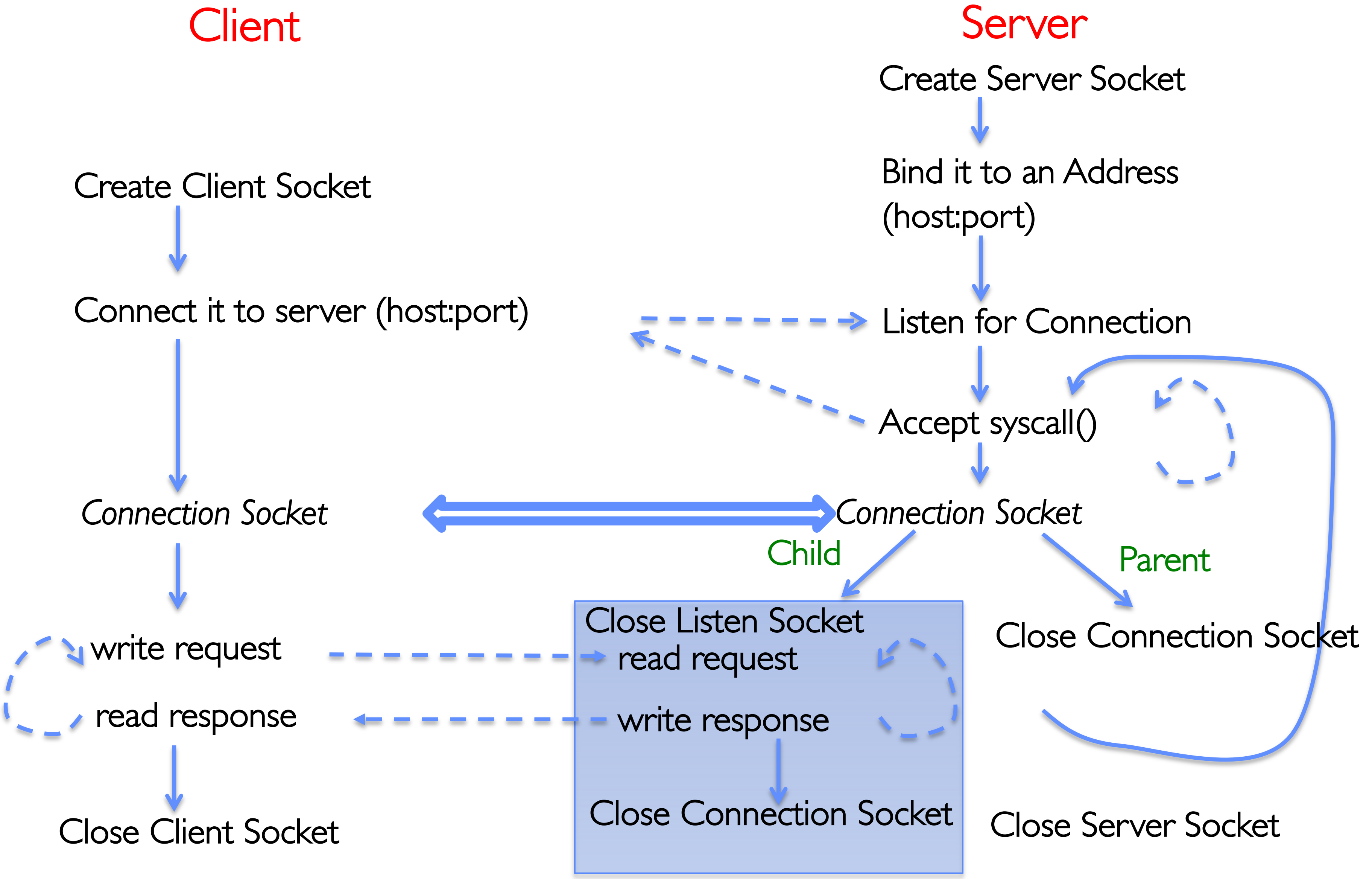
```
// Socket setup code elided…
listen(server_socket, MAX_QUEUE);
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) {
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);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



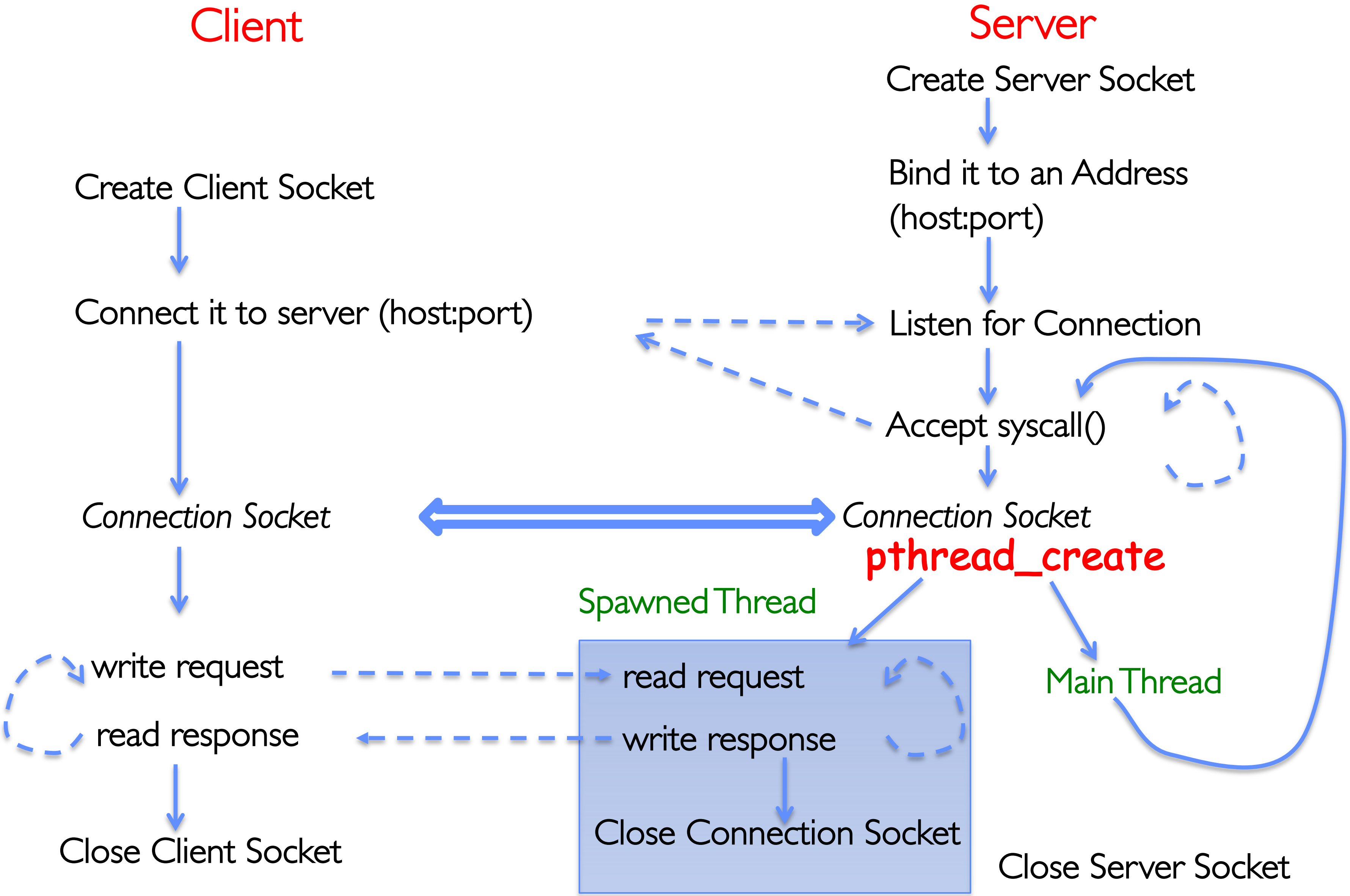
# Server Code (v3)

```
// 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)

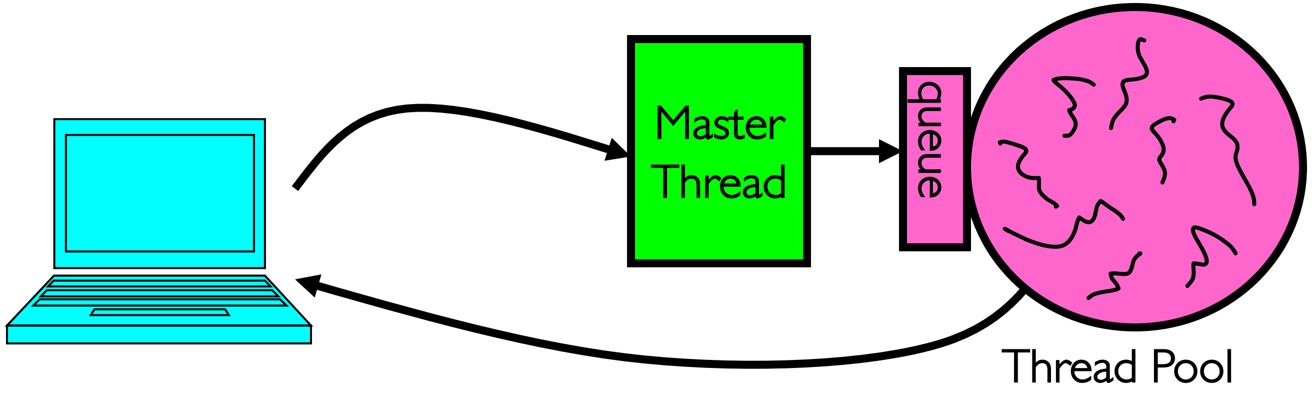
- Spawn a new *thread* to handle each connection
  - Lower overhead spawning process (less to do)
- Main *thread* initiates new client connections without waiting for previously spawned threads
- Why give up the protection of separate processes?
  - More efficient to create new threads
  - More efficient 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

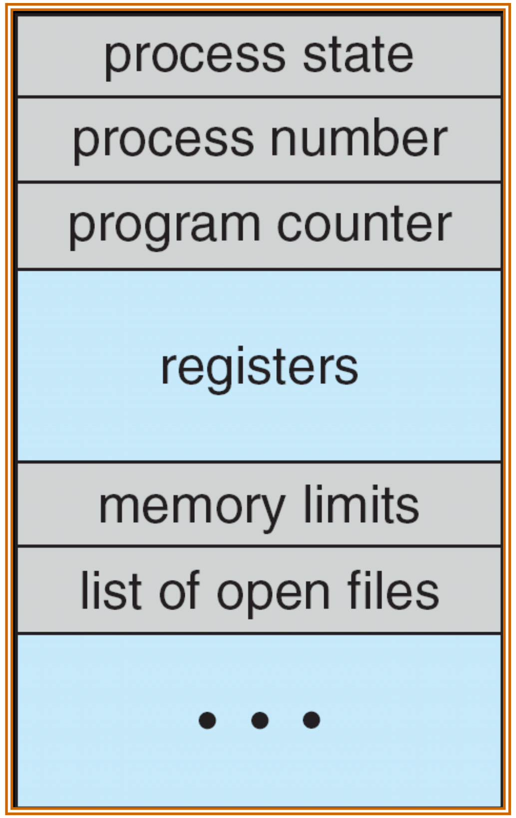



```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

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

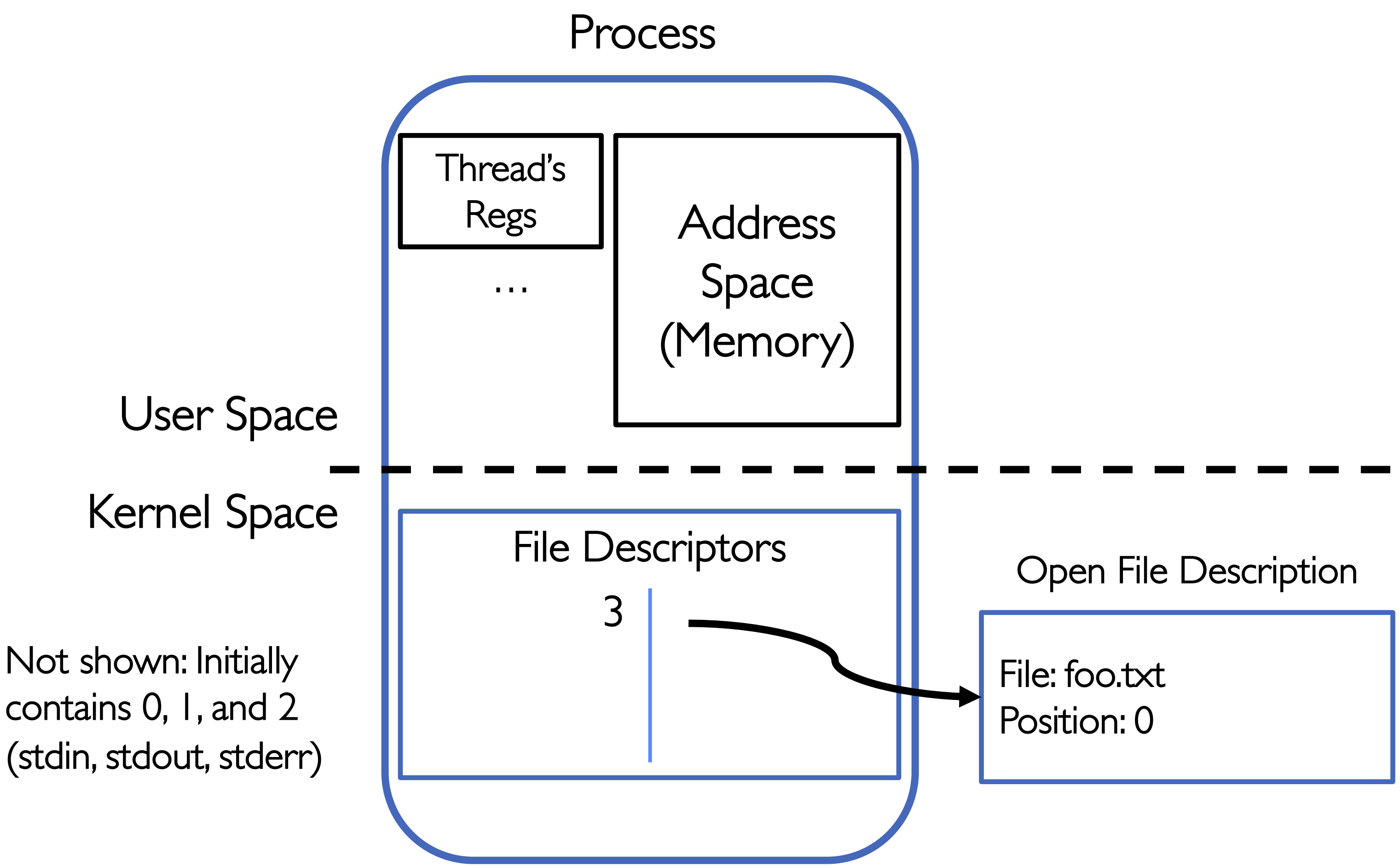
# Recall: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel *Scheduler* maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision




**Process Control Block**

# Process-Specific File Descriptor Table inside Kernel

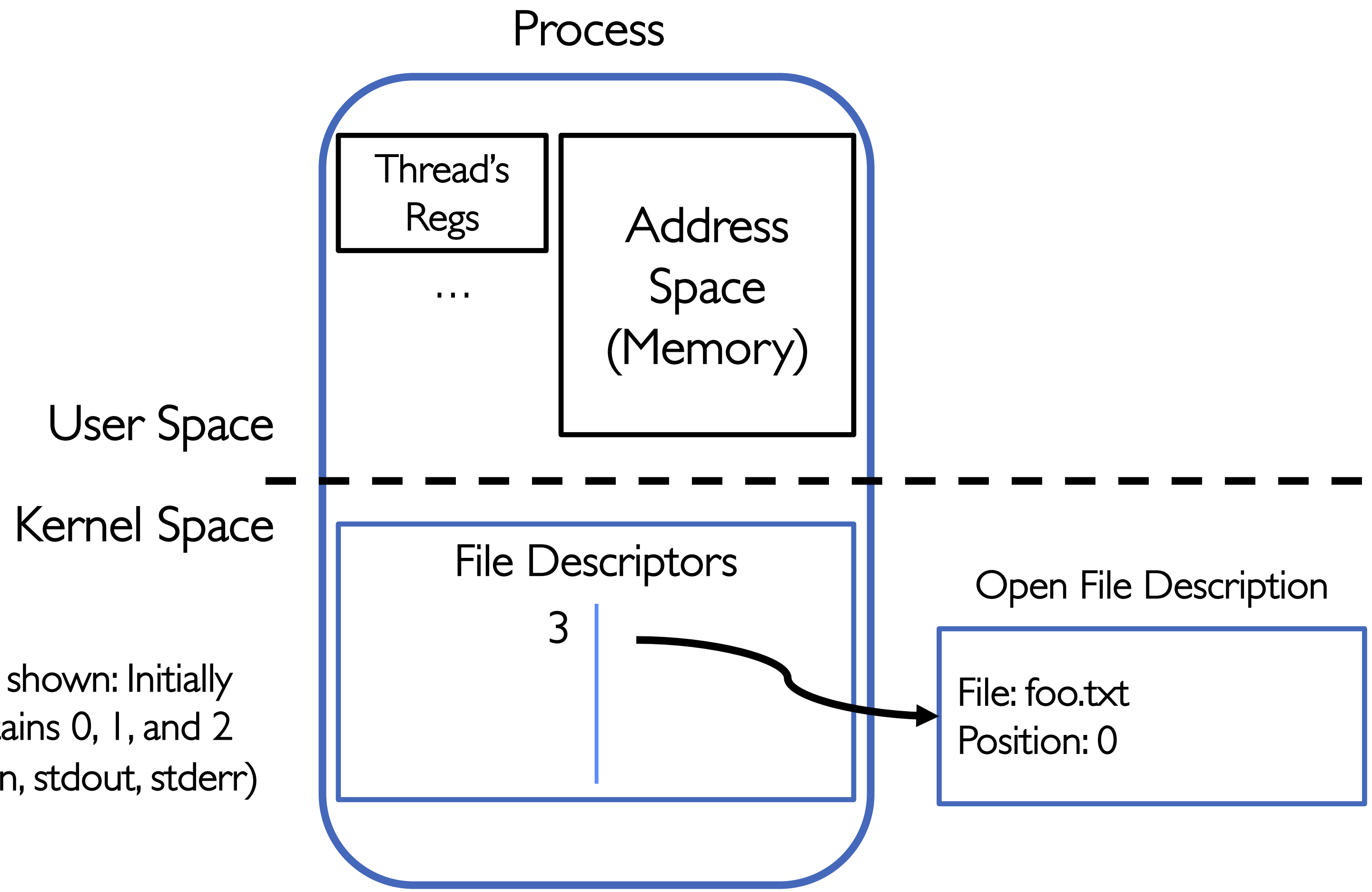



Suppose that we execute

open("foo.txt")

and that the result is 3

# Process-Specific File Descriptor Table inside Kernel

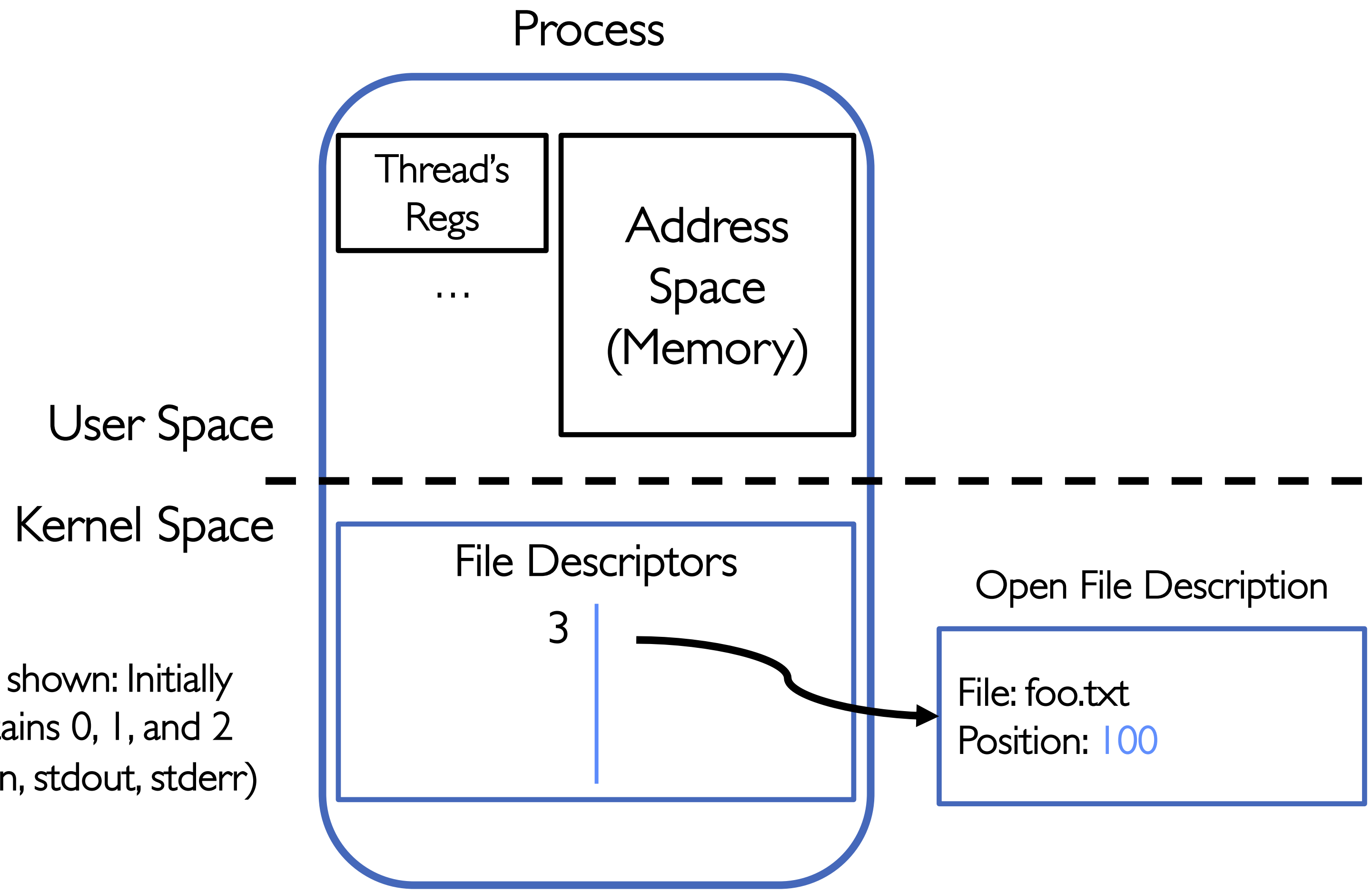



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

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

# Process-Specific File Descriptor Table inside Kernel

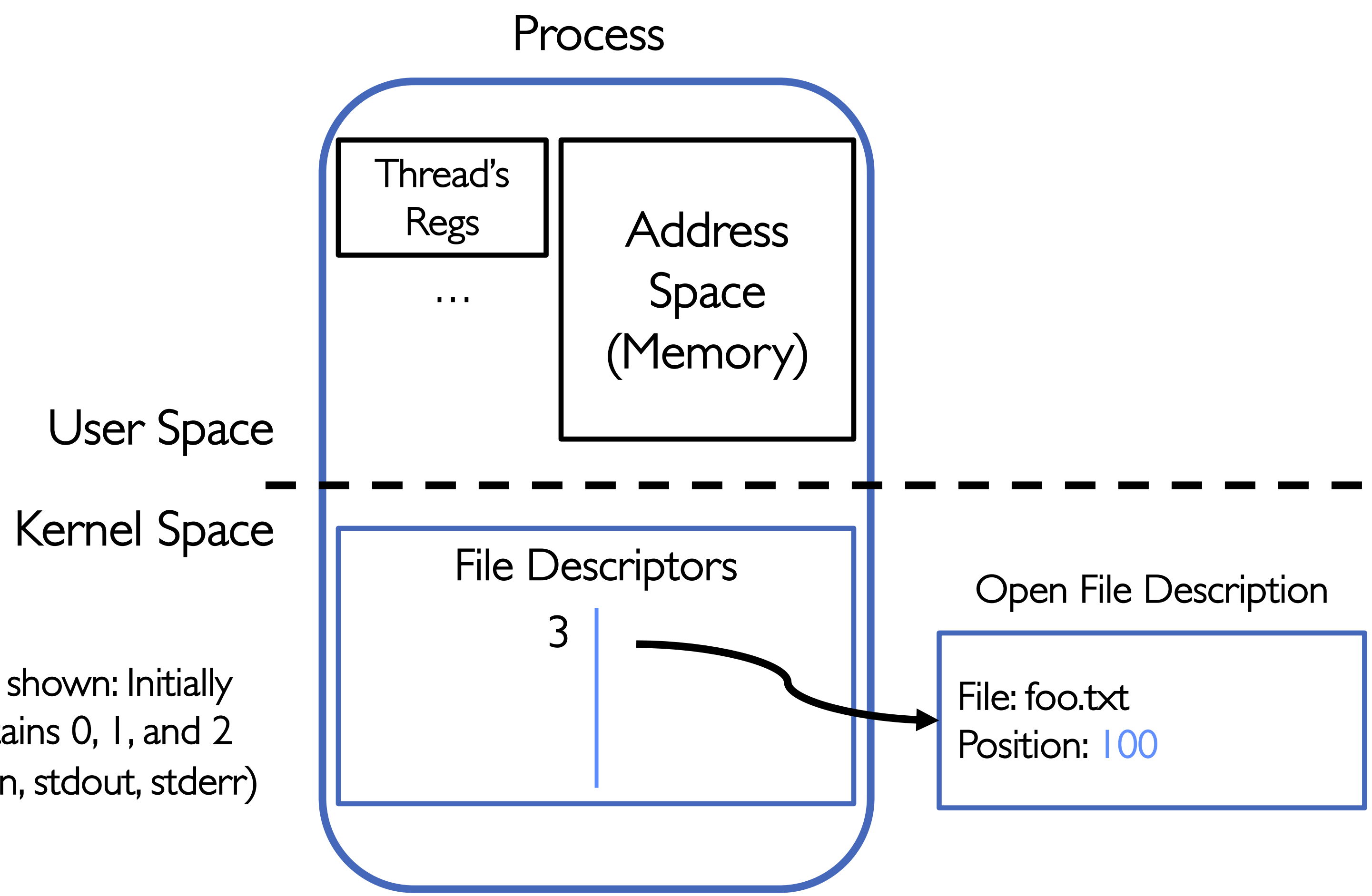



Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

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

# Process-Specific File Descriptor Table inside Kernel



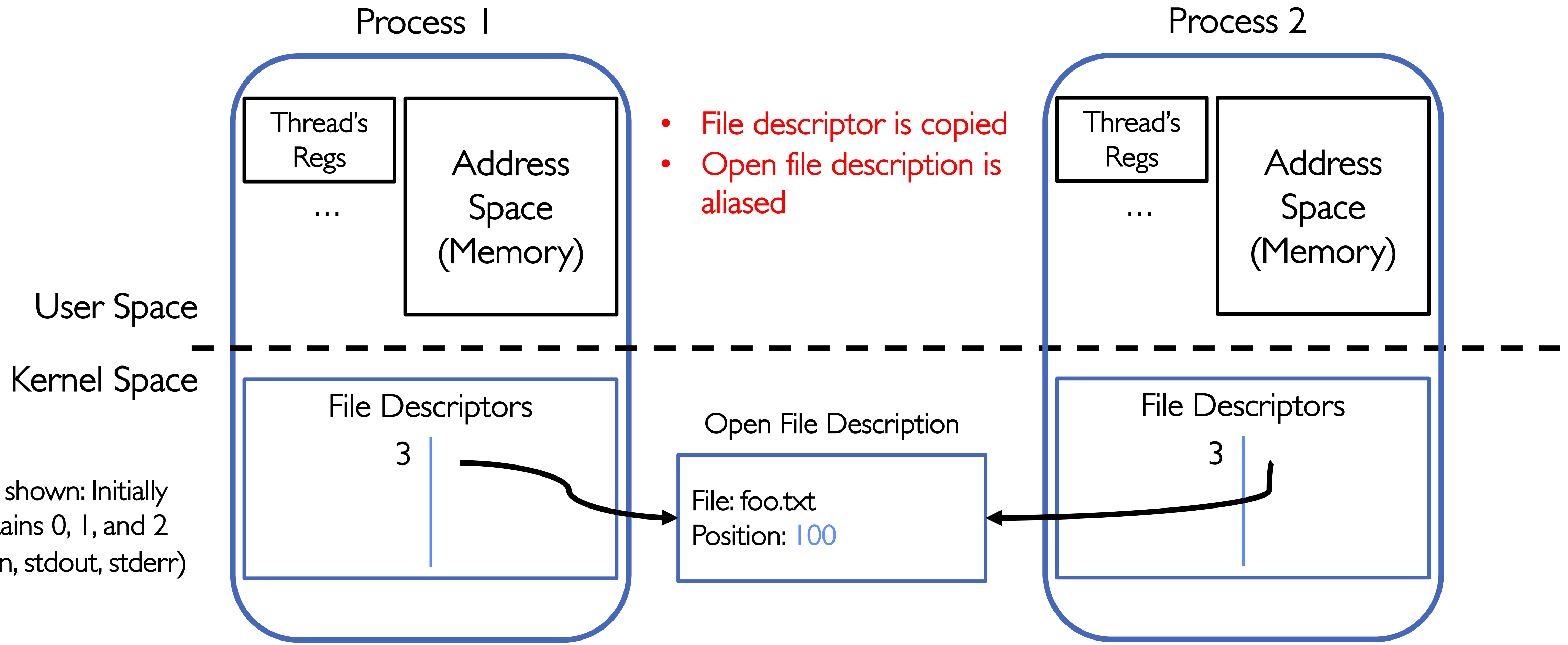

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

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()`!

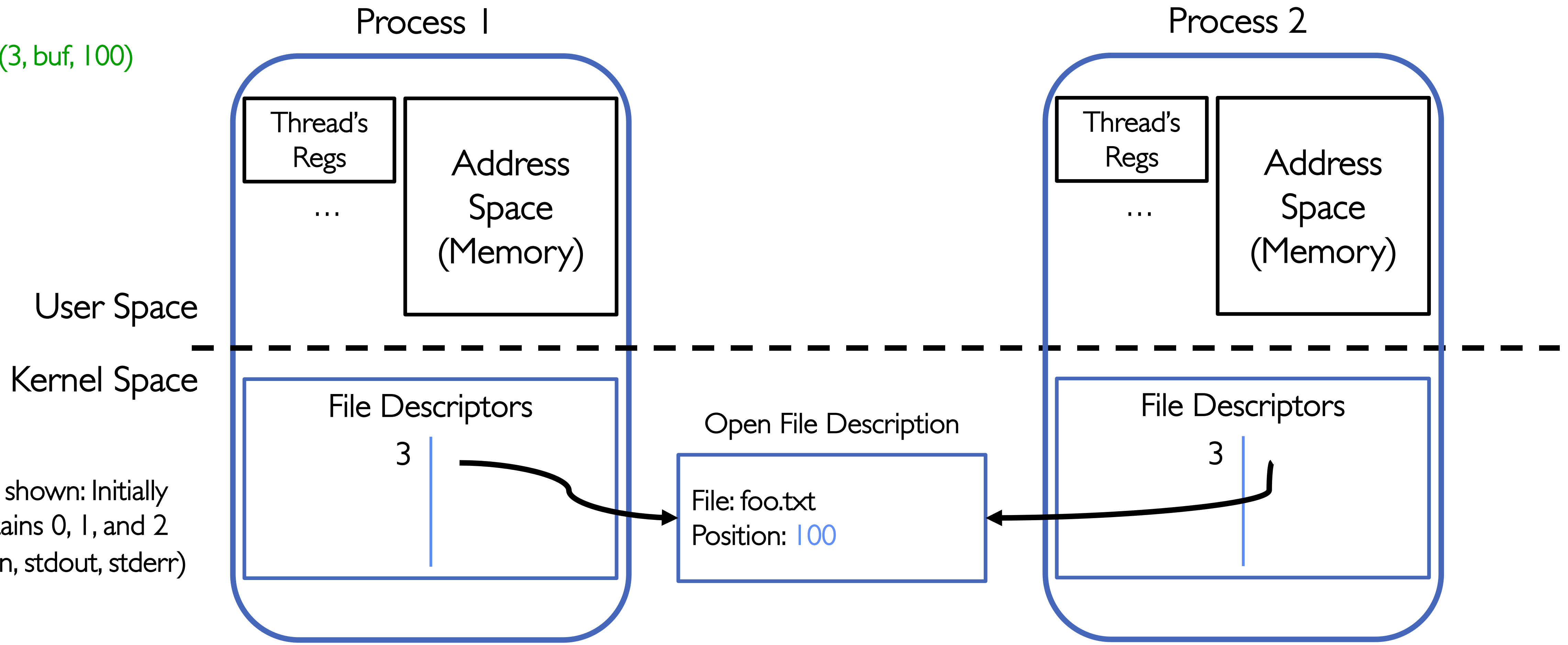



- File descriptor is copied
- Open file description is aliased

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

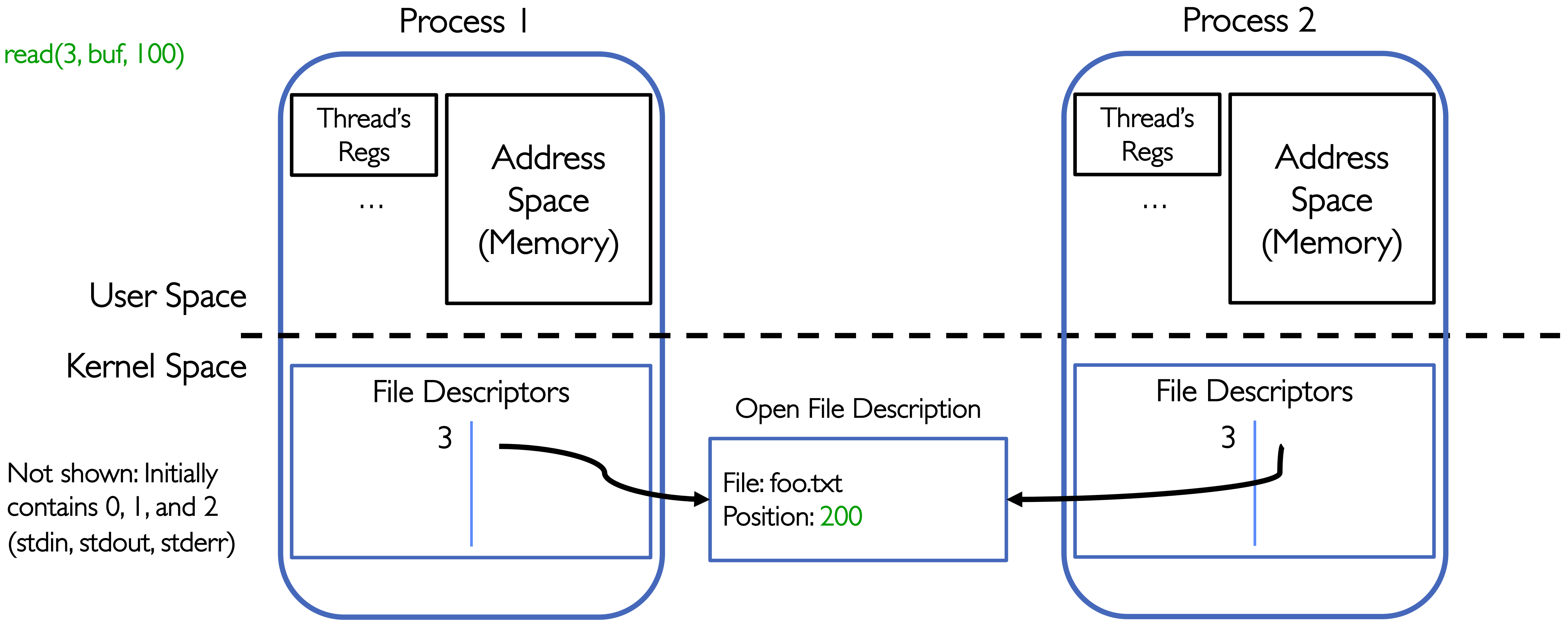
# Open File Description is *Aliased*

read(3, buf, 100)



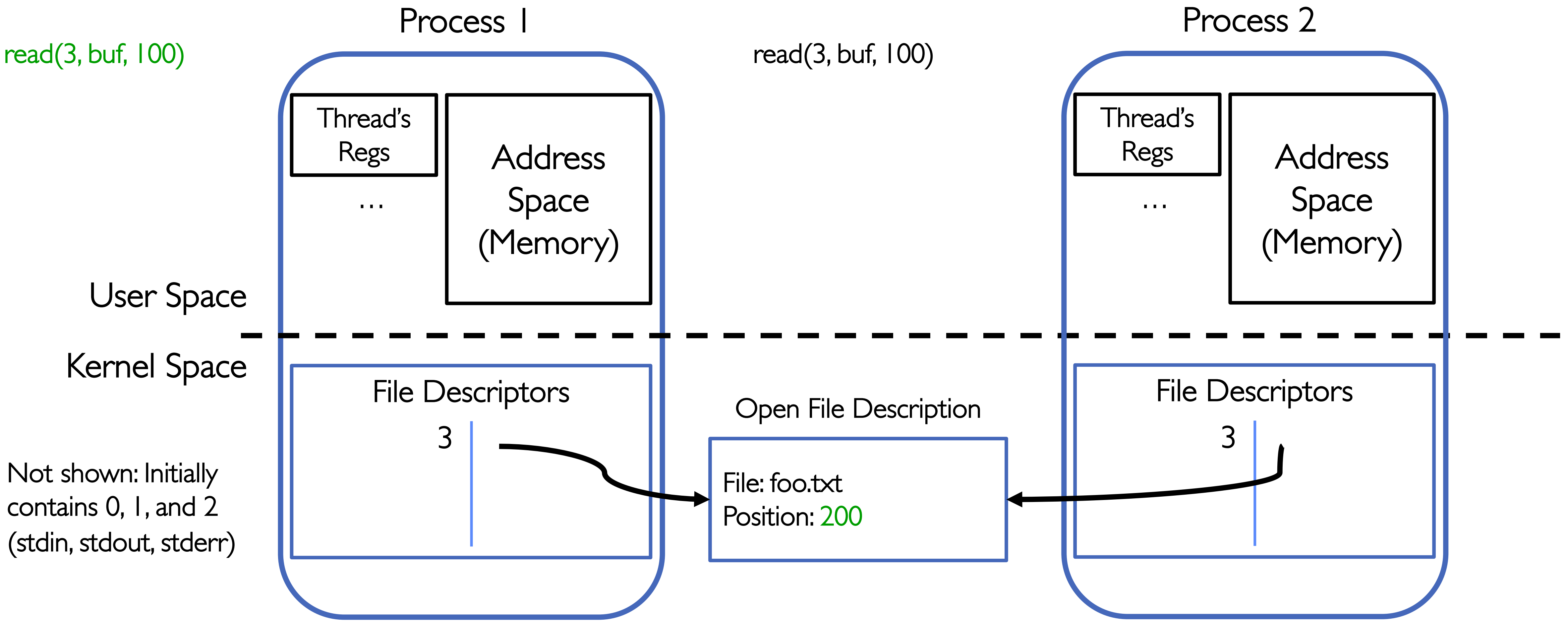

Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

# Open File Description is *Aliased*



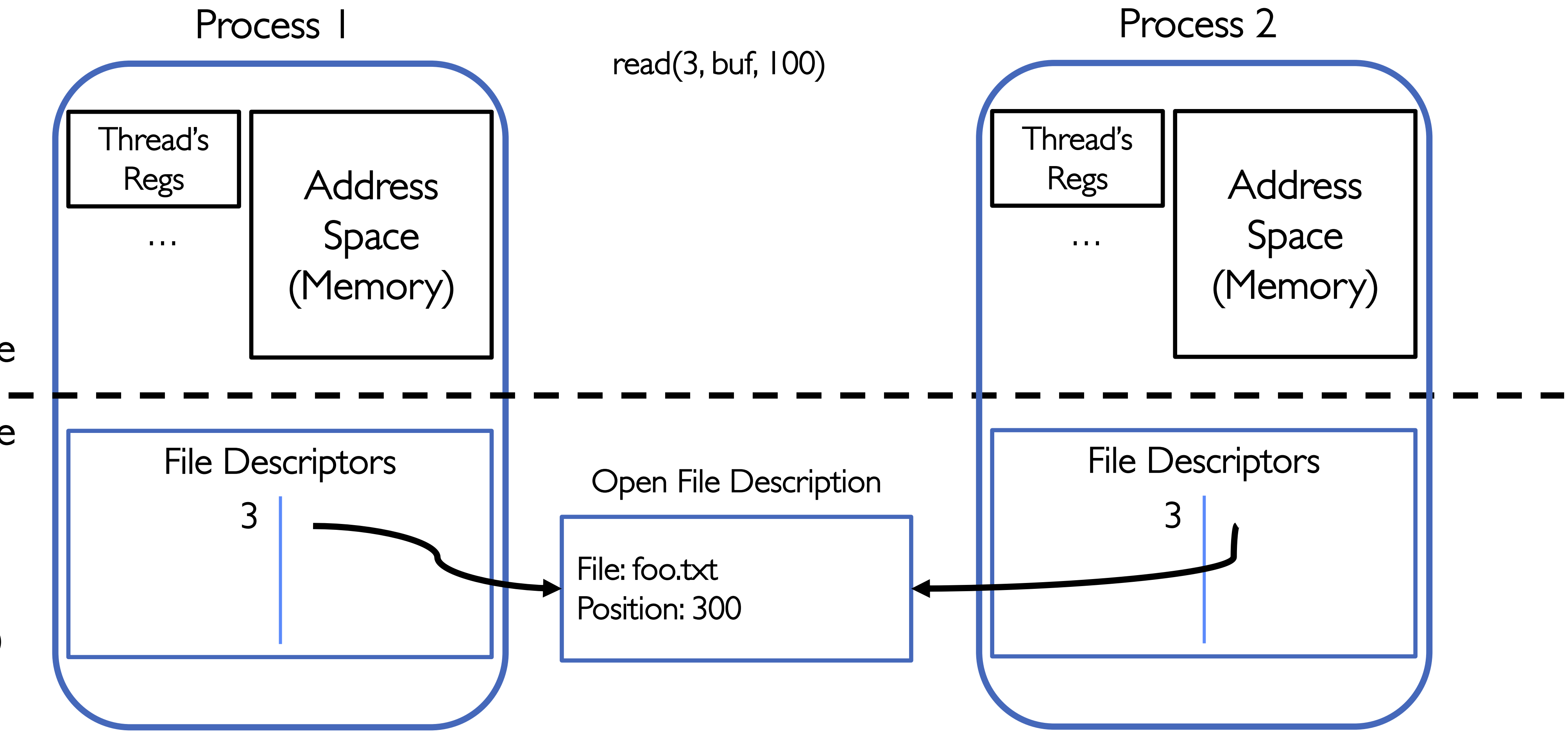
# Open File Description is *Aliased*

read(3, buf, 100)



# Open File Description is *Aliased*

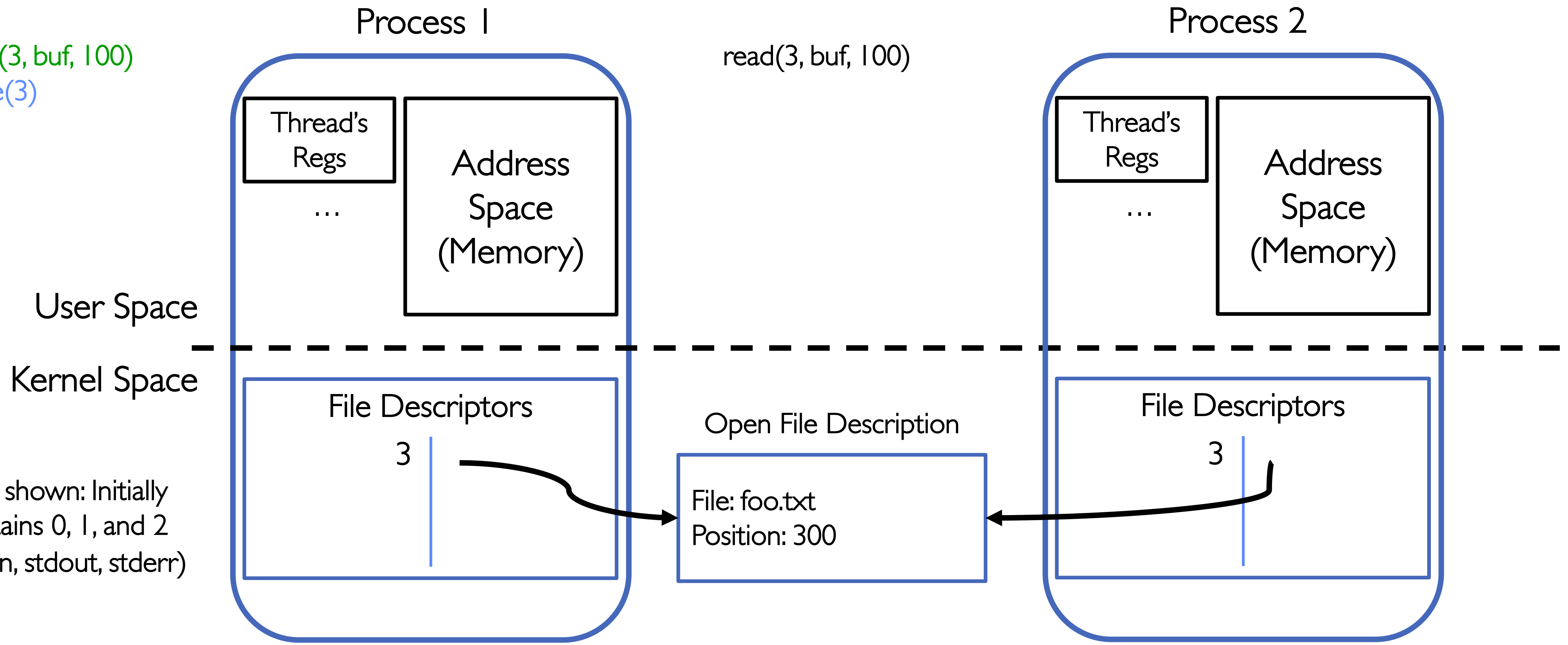
read(3, buf, 100)




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

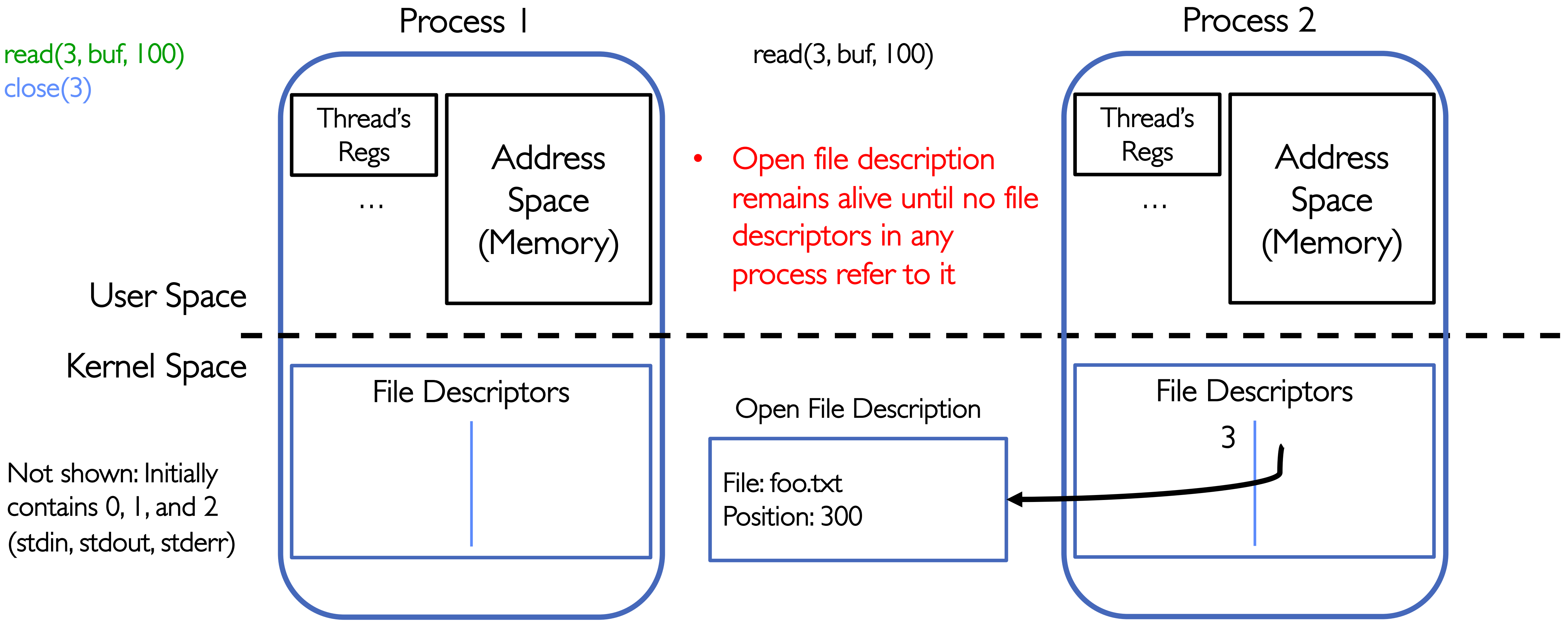
# File Descriptor is *Copied*

read(3, buf, 100)
close(3)




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

# File Descriptor is *Copied*



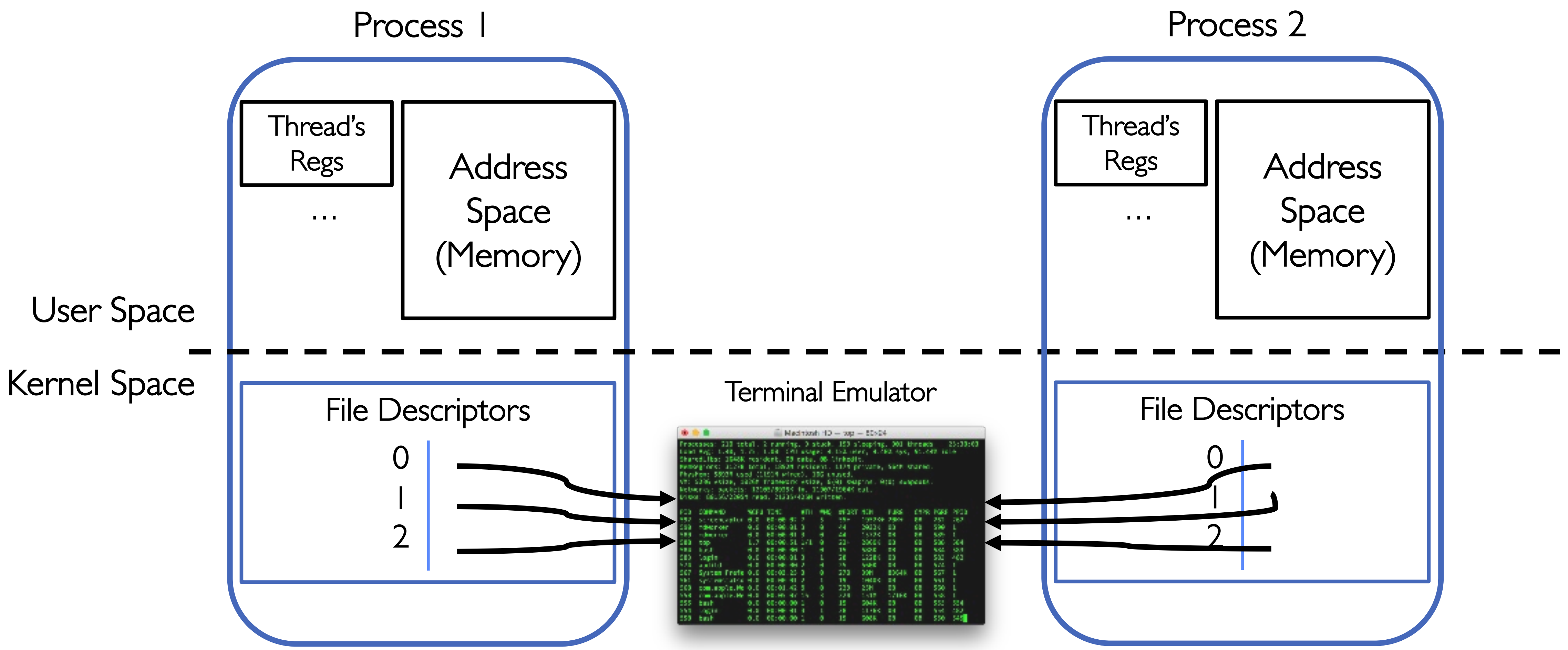
# Why is Aliasing the Open File Description a Good Idea?

- It allows for *shared resources* between processes

# Example: Shared Terminal Emulator

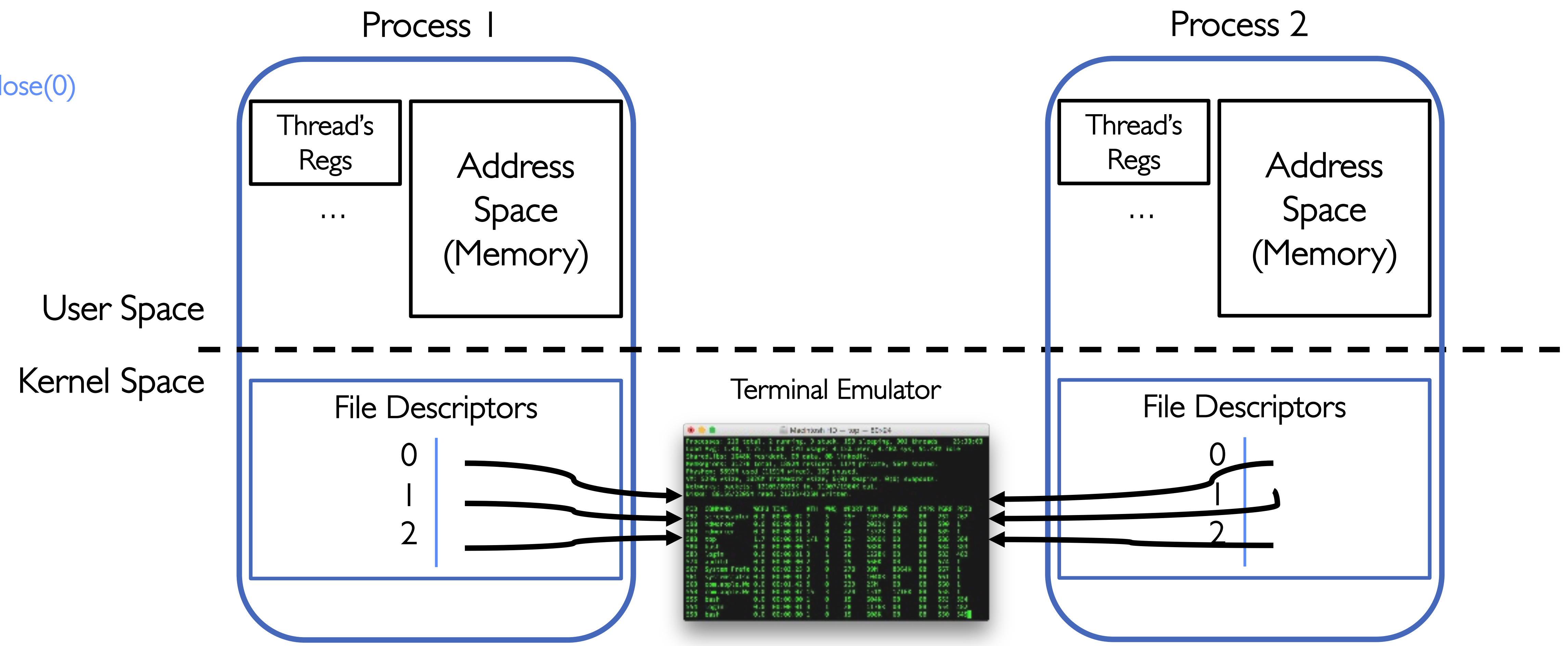
- When you **`fork()`** a process, the parent's and child's **`printf`** outputs go to the same terminal

# Example: Shared Terminal Emulator



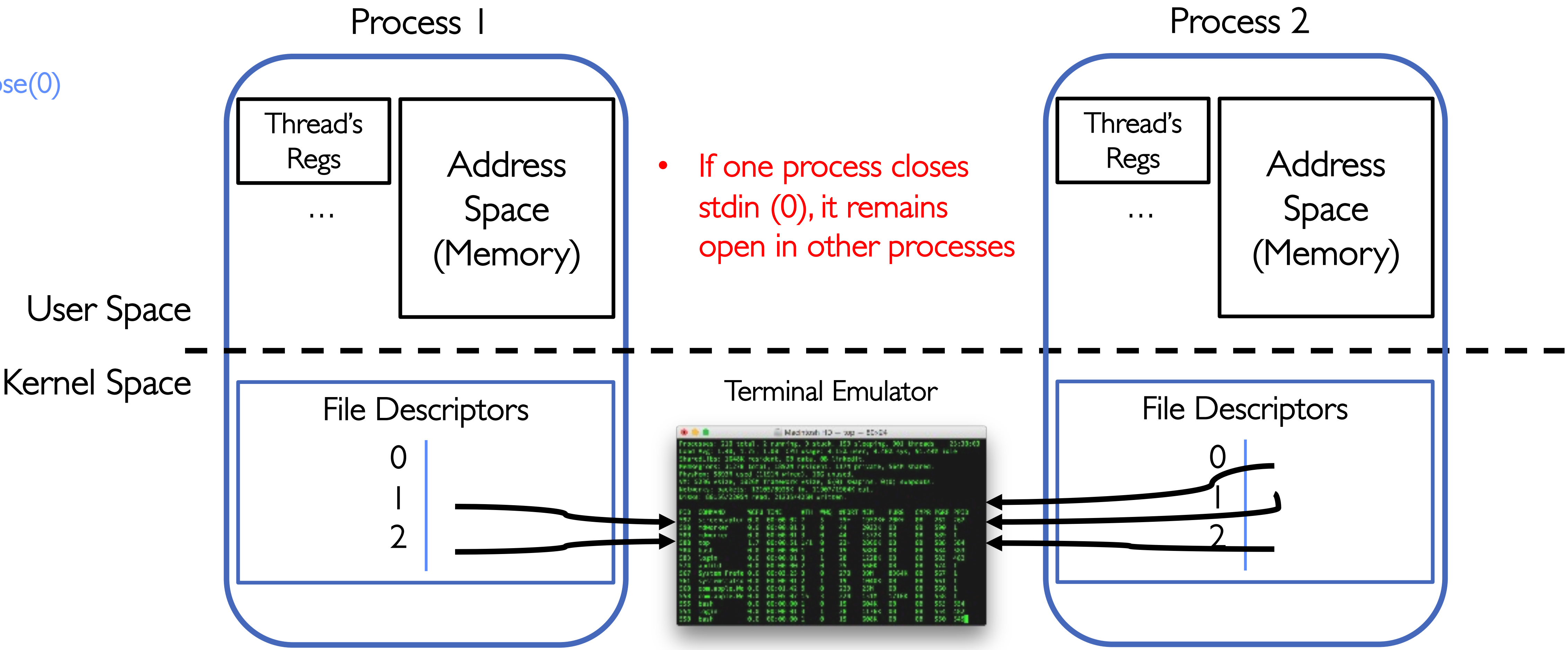
# Example: Shared Terminal Emulator

close(0)



# Example: Shared Terminal Emulator

close(0)



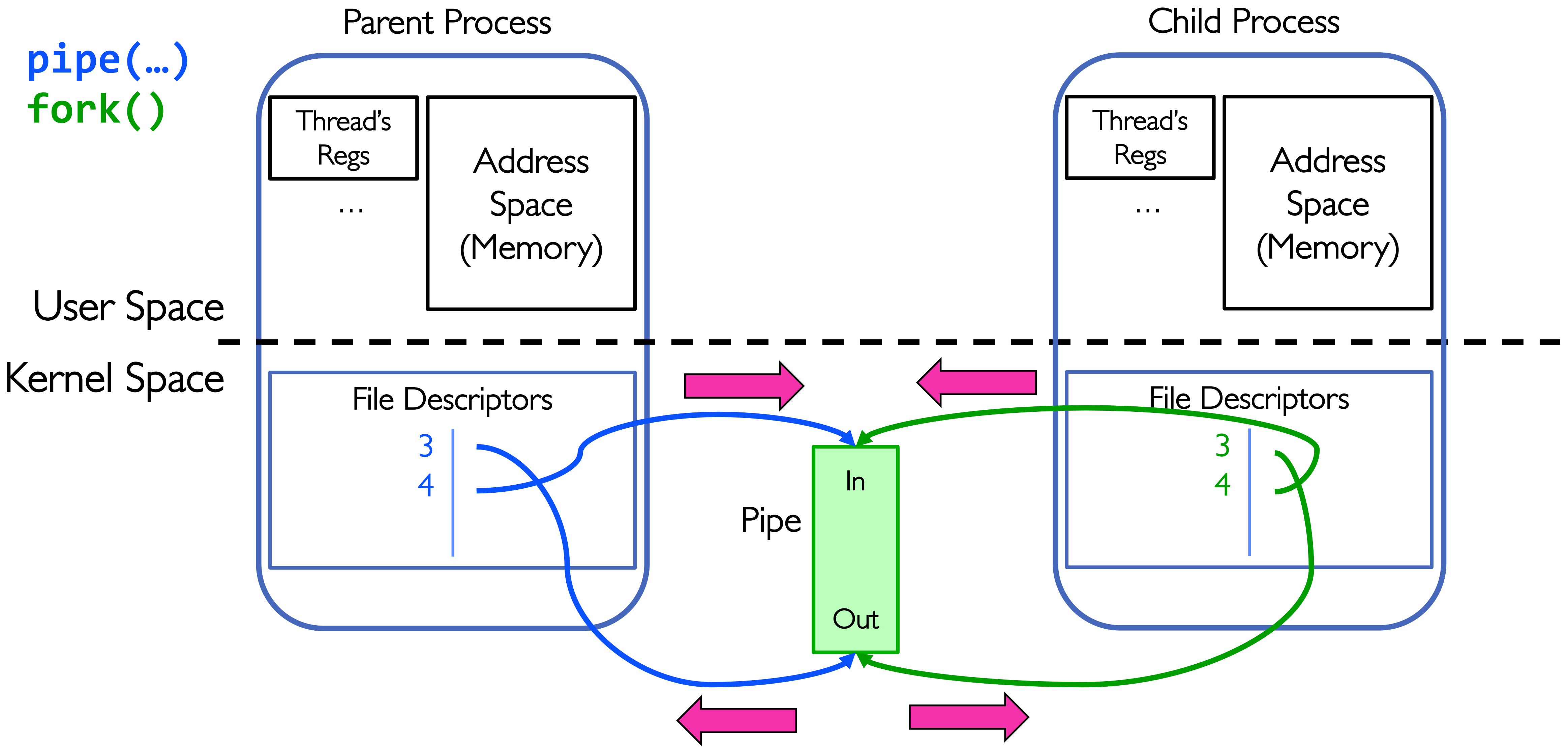

- If one process closes stdin (0), it remains open in other processes

# 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]);
}
```

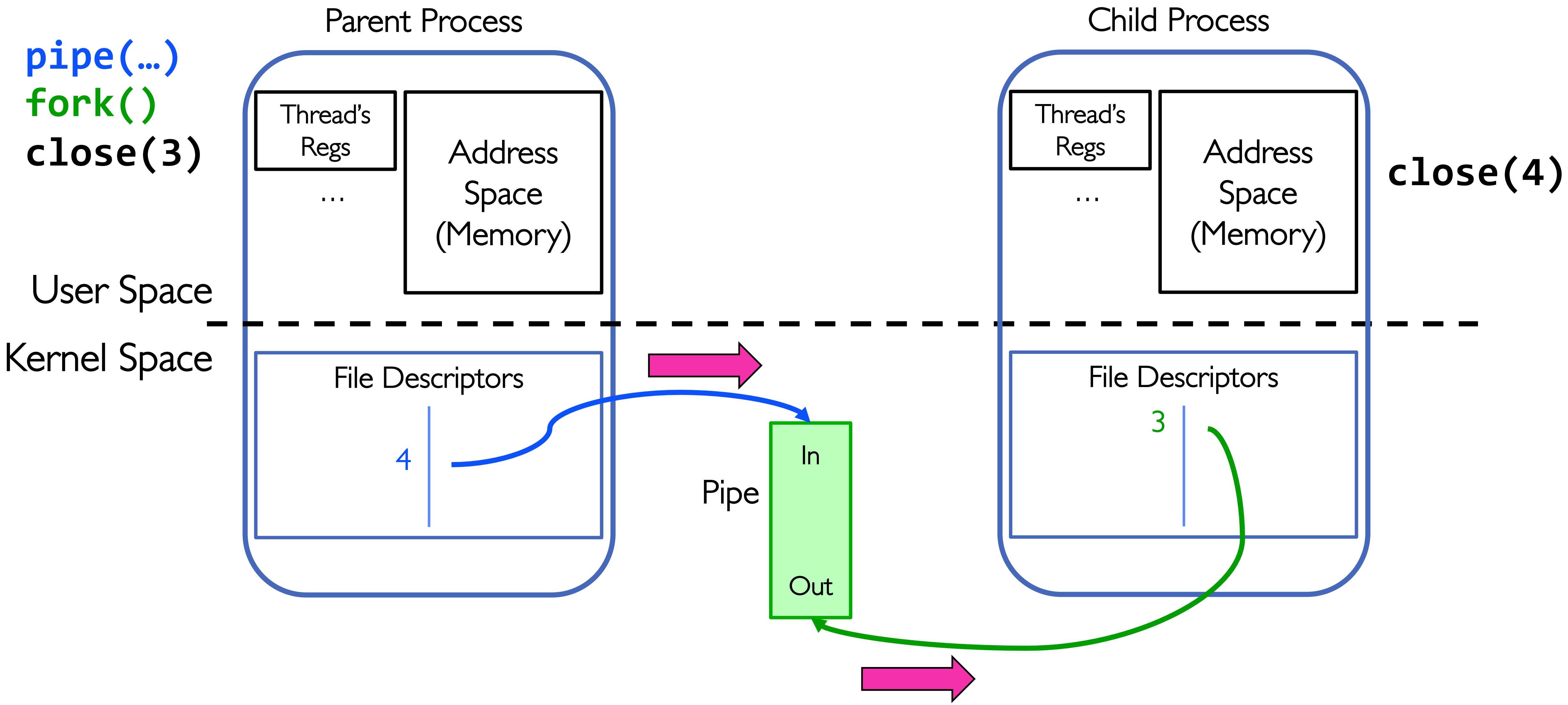
Could be useful for multithreaded processes…

# Example: Pipes *Between* Processes



# Example: Channel from Parent ⇒ Child

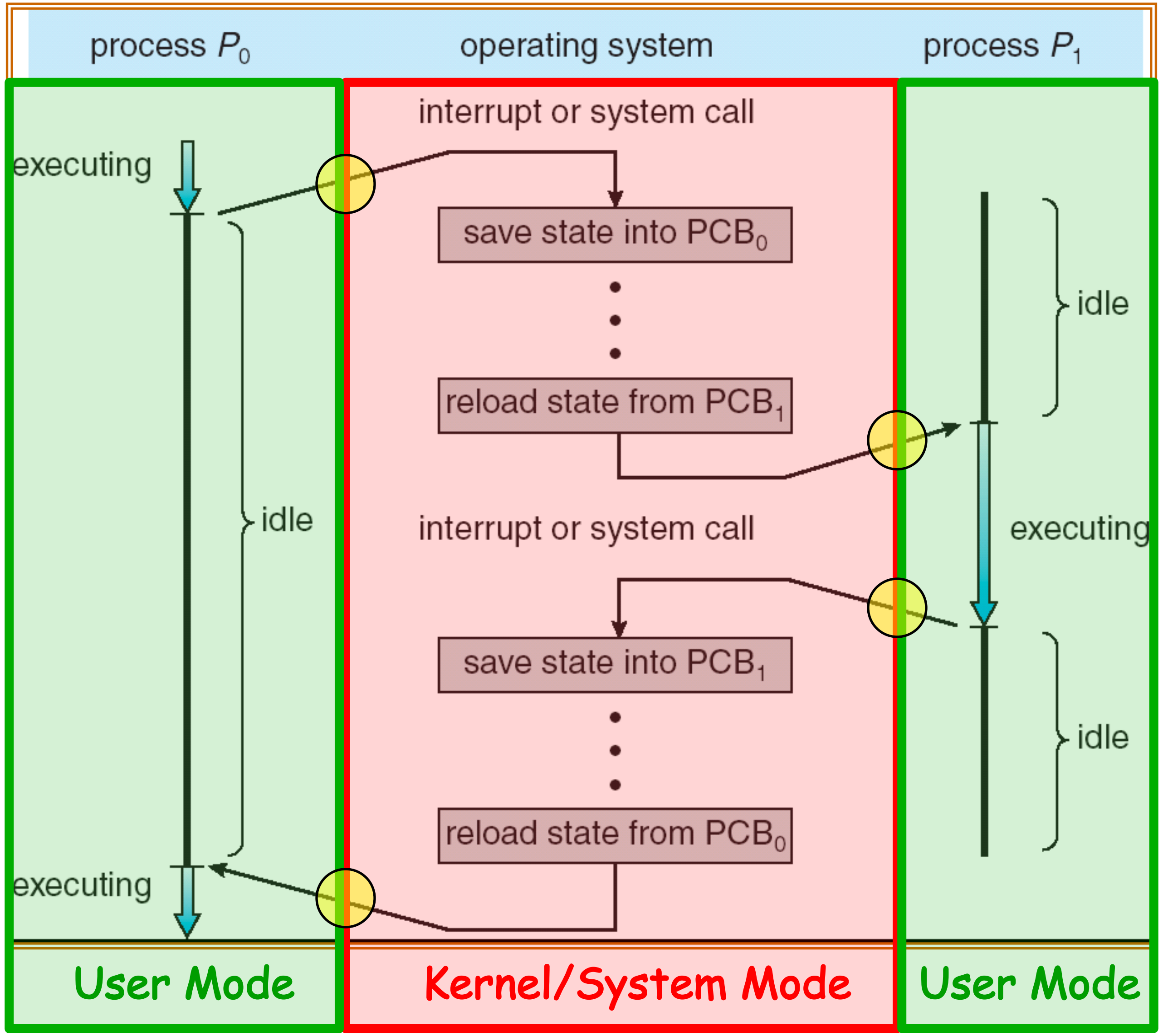
`pipe(…)`
`fork()`
`close(3)`



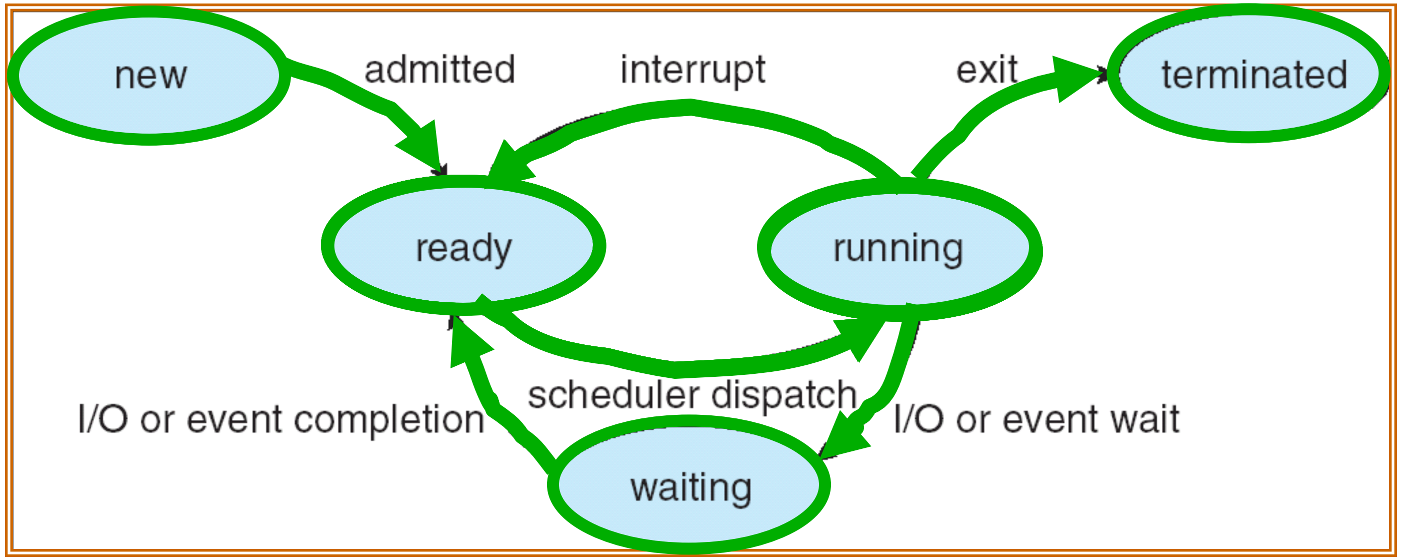
# Inter-Process Communication (IPC): Parent ⇒ Child

```
// 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);
}
```

# Recall: CPU Switch From Process A to Process B

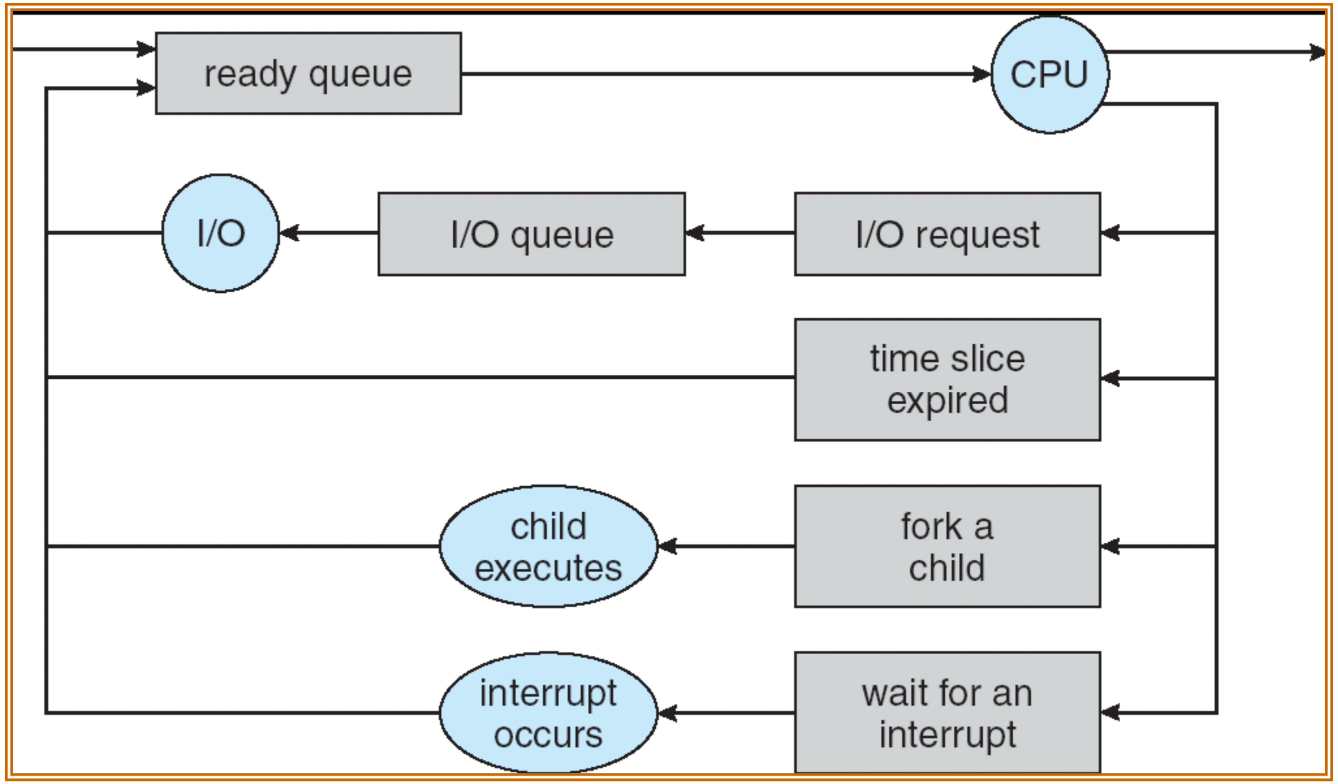


# Lifecycle of a Process




- As a process executes, it changes state:
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution

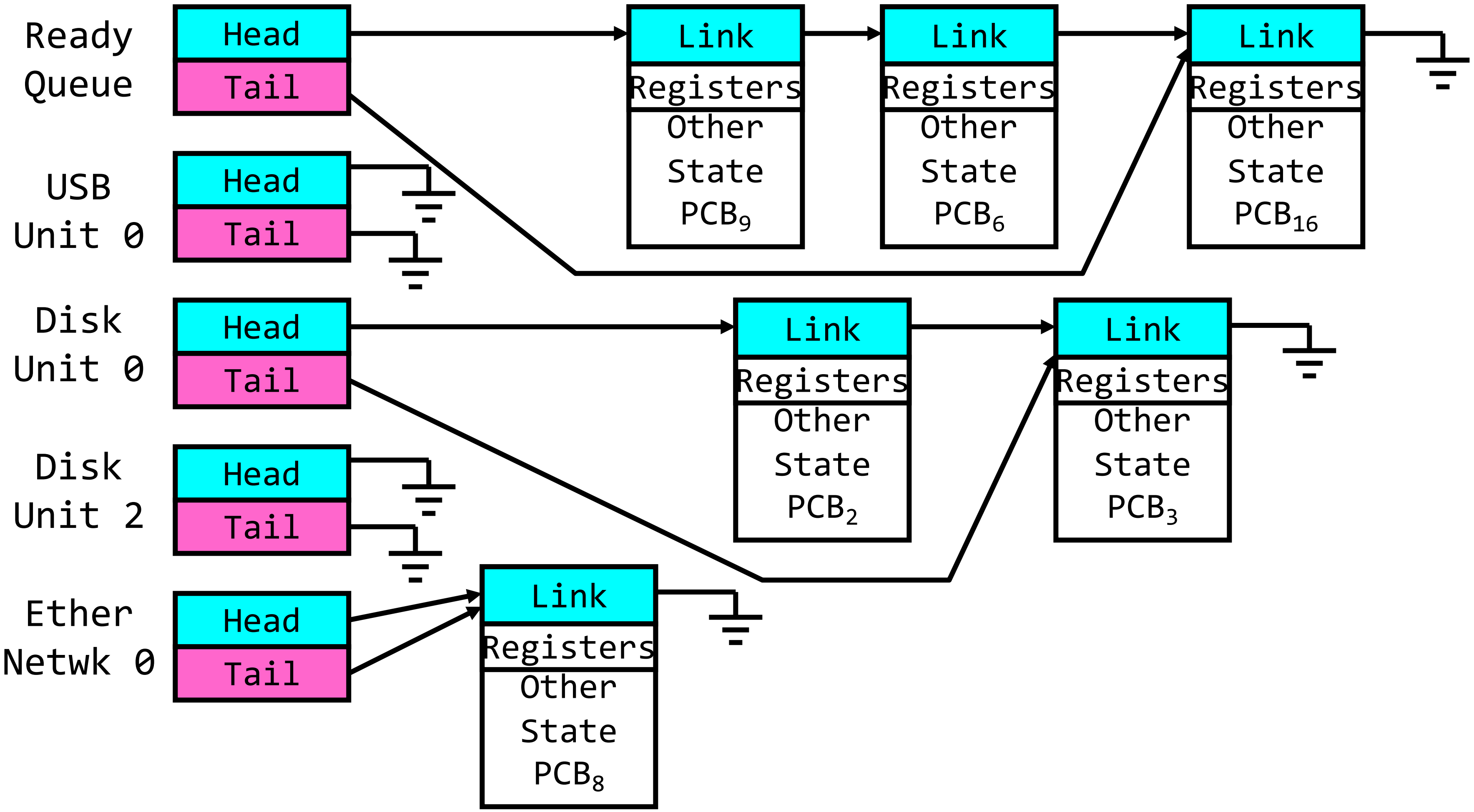
# Process Scheduling




- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

# Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



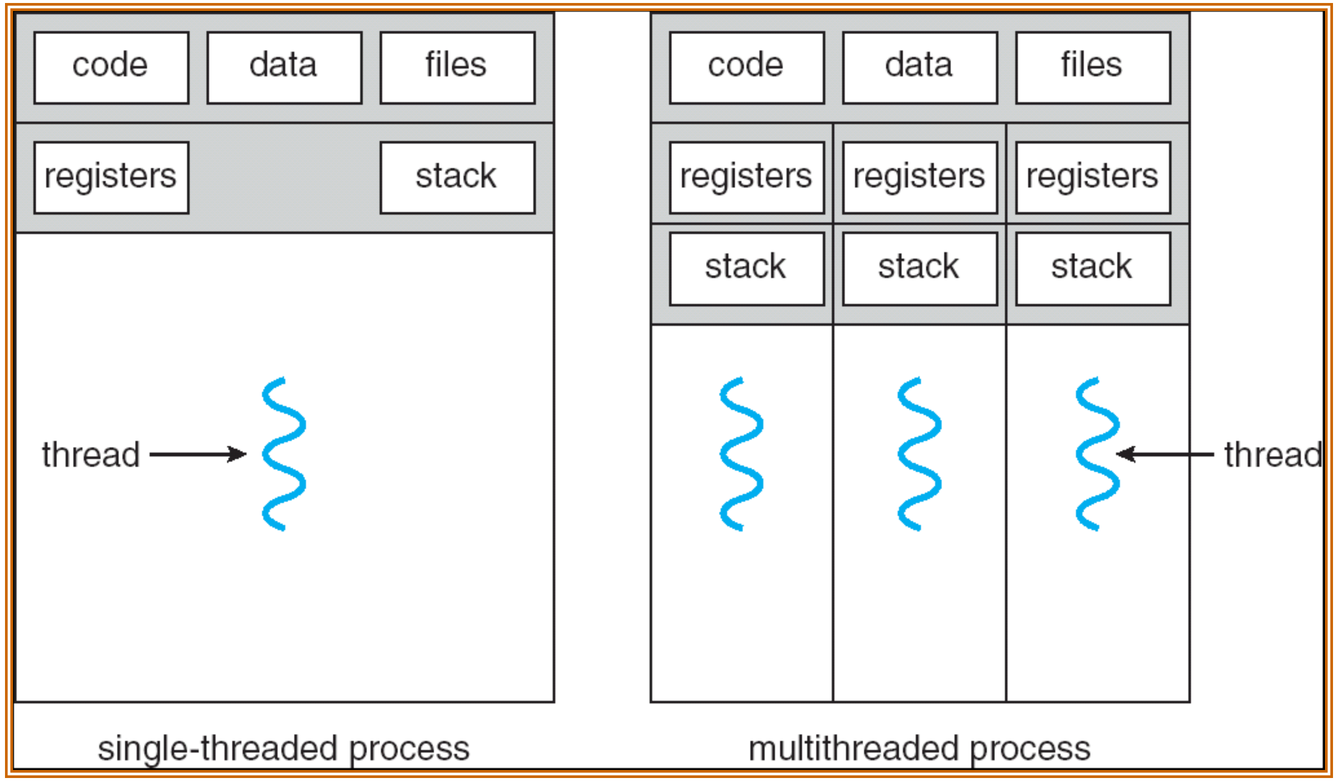
# Recall: Modern Process with Threads

- Thread: *a sequential execution stream within process*
  (Sometimes called a "Lightweight process")
  – Process still contains a single Address Space
  – No protection between threads

- Multithreading: *a single program made up of a number of different concurrent activities*
  – Sometimes called multitasking, as in Ada …

- Why separate the concept of a thread from that of a process?
  – Discuss the "thread" part of a process (concurrency)
  – Separate from the "address space" (protection)
  – Heavyweight Process ≡ Process with one thread

# Recall: Single and Multithreaded Processes

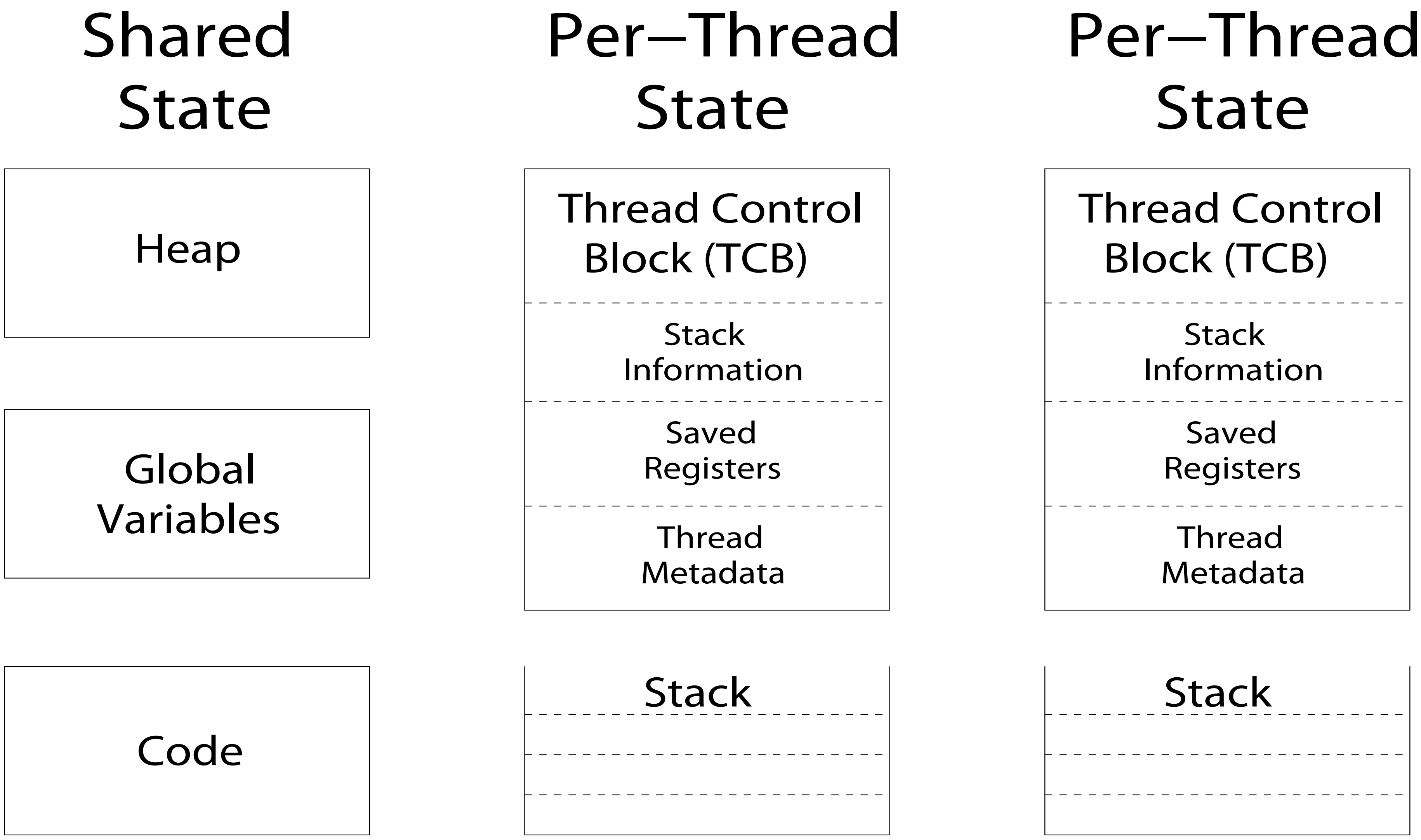



- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

# Recall: Thread State

- State shared by all threads in process/address space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

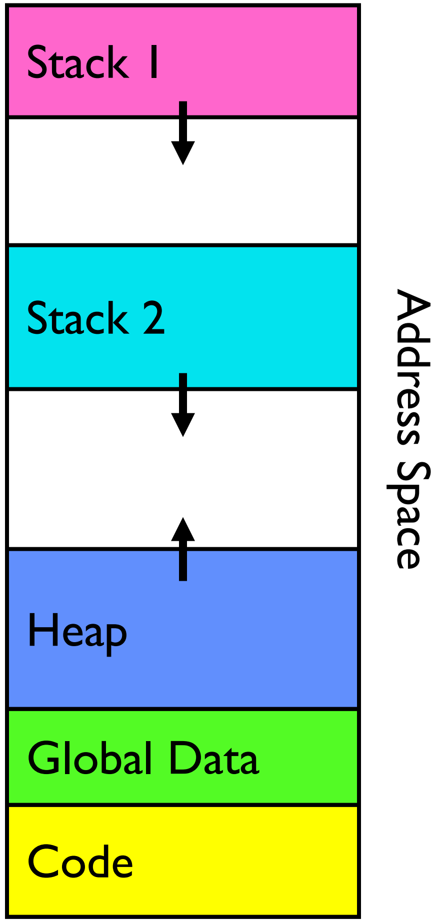
# Shared vs. Per-Thread State



# Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

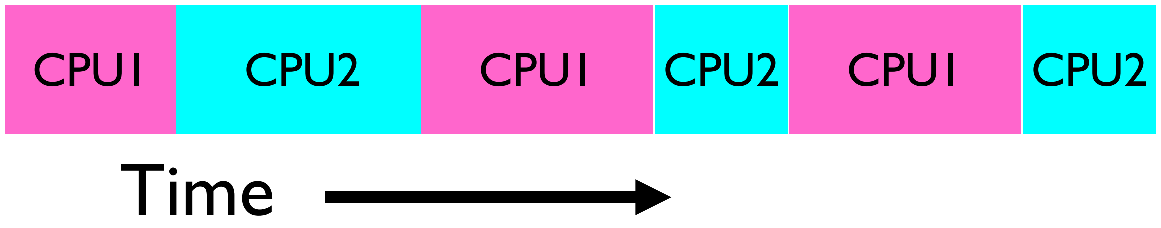


# Recall: Use of Threads

- Version of program with Threads (loose syntax):

```
main() {
    ThreadFork(ComputePI, "pi.txt" ));
    ThreadFork(PrintClassList, "classlist.txt"));
}
```

- What does **`ThreadFork()`** do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



# The Core of Concurrency: the Dispatch Loop

- Conceptually, the scheduling loop of the operating system looks as follows:

```
Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

- This is an *infinite* loop
  – One could argue that this is all that the OS does
- Should we ever exit this loop???
  – When would that be?

# 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 interprocess 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"