Operating Systems (Honor Track)

Abstractions 2: Files and I/O
A quick, programmer's viewpoint

Xin Jin

Spring 2022

Acknowledgments: Ion Stoica, Berkeley CS 162

Today: 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
- Common Pitfalls with OS Abstractions [if time]

Unix/POSIX Idea: 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()`**
- Additional: **`ioctl()`** for custom configuration that doesn't quite fit
- Note that the "Everything is a File" idea was a radical idea when proposed
  - Dennis Ritchie and Ken Thompson described this idea in their seminal paper on UNIX called "The UNIX Time-Sharing System" from 1974

Note: What does POSIX stand for?

- POSIX: Portable Operating System Interface (for uniX?)
  - Interface for application programmers (mostly)
  - Defines the term “Unix,” derived from AT&T Unix
  - Created to bring order to many Unix-derived OSes, so applications are portable
    - » Partially available on non-Unix OSes, like Windows
  - Requires standard system call interface

The File System Abstraction

- File
  - Named collection of data in a file system
  - POSIX File data: sequence of bytes
    - » Could be text, binary, serialized objects, …
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, etc.
- Directory
  - "Folder" containing files & directories
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - /home/ff/pkuos/public_html/sp22/index.html
  - Links and Volumes (later)

Connecting Processes, File Systems, and Users

- **Every process has a** ***current working directory*** **(CWD)**
  - Can be set with system call:
    `int chdir(const char *path); //change CWD`
- Absolute paths ignore CWD
  - /home/ff/pkuos
- Relative paths are relative to CWD
  - index.html, ./index.html
    - » Refers to index.html in current working directory
  - ../index.html
    - » Refers to index.html in parent of current working directory
  - ~/index.html
    - » Refers to index.html in the home directory

I/O and Storage Layers

Application / Service

High Level I/O — *Streams*

Low Level I/O — *File Descriptors*

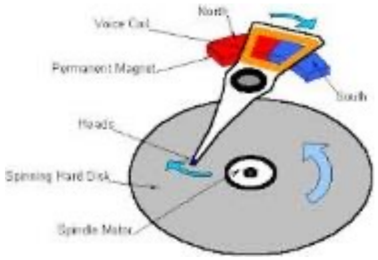
Syscall — *open(), read(), write(), close(), …*

Open File Descriptions

File System — *Files/Directories/Indexes*

I/O Driver — *Commands and Data Transfers*

Disks, Flash, Controllers, DMA



C High-Level File API – Streams

- Operates on "streams" – unformatted sequences of bytes (whether text or binary 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

- Open stream represented by pointer to a FILE data structure
  - Error reported by returning a NULL pointer

C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed.
  - `FILE *stdin` – normal source of input, can be redirected
  - `FILE *stdout` – normal source of output, can too
  - `FILE *stderr` – diagnostics and errors

- STDIN / STDOUT enable composition in Unix

- All can be redirected
  - `cat hello.txt | grep "World!"`
  - **`cat`**'s **`stdout`** goes to **`grep`**'s **`stdin`**

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );                // rtn c or EOF on err
int fputs( const char *s, FILE *fp );        // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

C Streams: Char-by-Char I/O

```
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  int c;

  c = fgetc(input);
  while (c != EOF) {
    fputc(c, output);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

C High-Level File API

```
// character oriented
int fputc( int c, FILE *fp );        // rtn c or EOF on err
int fputs( const char *s, FILE *fp );       // rtn > 0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);
size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

C Streams: Block-by-Block I/O

```
#define BUFFER_SIZE 1024
int main(void) {
  FILE* input = fopen("input.txt", "r");
  FILE* output = fopen("output.txt", "w");
  char buffer[BUFFER_SIZE];
  size_t length;
  length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  while (length > 0) {
    fwrite(buffer, length, sizeof(char), output);
    length = fread(buffer, BUFFER_SIZE, sizeof(char), input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: System Programming

- Systems programmers should always be paranoid!
  - Otherwise you get intermittently buggy code
- We should really be writing things like:

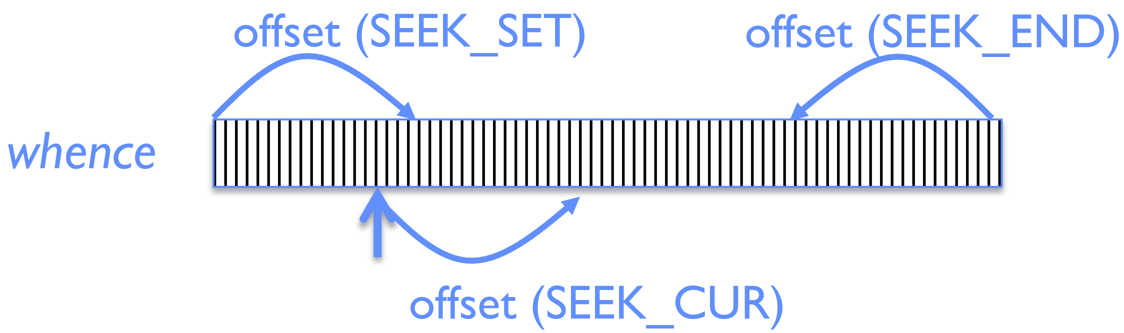
```
FILE* input = fopen(“input.txt”, “r”);
if (input == NULL) {
  // Prints our string and error msg.
  perror(“Failed to open input file”);
}
```

- Be **thorough about checking return values!**
  - Want failures to be systematically caught and dealt with
- I may be a bit loose with error checking for examples in class (to keep short)
  - Do as I say, not as I show in class!

C High-Level File API: Positioning The Pointer

```
int fseek(FILE *stream, long int offset, int whence); // Reposition stream position indicator
long int ftell (FILE *stream) // Get current position in stream
void rewind (FILE *stream) // Set position of stream to the beginning
```

- For `fseek()`, the `offset` is interpreted based on the `whence` argument (constants in `stdio.h`):
  - `SEEK_SET`: Then offset interpreted from beginning (position 0)
  - `SEEK_END`: Then offset interpreted backwards from end of file
  - `SEEK_CUR`: Then offset interpreted from current position



- Overall preserves high-level abstraction of a uniform stream of objects

Today: 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
- Common Pitfalls with OS Abstractions

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

Low-Level File I/O: The RAW system-call interface

```
#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

- Integer return from `open()` is a *file descriptor*
  - *Error indicated by return < 0:* the global `errno` variable set with error (see man pages)
- Operations on *file descriptors*:
  - Open system call created an *open file description* entry in system-wide table of open files
  - *Open file description* object in the kernel represents an instance of an open file
  - Why give user an integer instead of a pointer to the file description in kernel?

C Low-Level (pre-opened) Standard Descriptors

```
#include <unistd.h>
STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

// Get file descriptor inside FILE *
int fileno (FILE *stream)

// Make FILE * from descriptor
FILE * fdopen (int filedes, const char *opentype)
```

Low-Level File API

- Read data from open file using file descriptor:

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
```

  - Reads up to `maxsize` bytes – **might actually read less!**
  - returns bytes read, 0 => EOF, -1 => error

- Write data to open file using file descriptor

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

  - returns number of bytes written

- Reposition file offset within kernel (this is independent of any position held by high-level FILE descriptor for this file!

```
off_t lseek (int filedes, off_t offset, int whence)
```

Example: lowio.c

```
int main() {
  char buf[1000];
  int     fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
  ssize_t rd = read(fd, buf, sizeof(buf));
  int    err = close(fd);
  ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

- How many bytes does this program read?

POSIX I/O: Design Patterns

- Open before use
  - Access control check, setup happens here
- Byte-oriented
  - Least common denominator
  - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close

POSIX I/O: Kernel Buffering

- Reads are buffered inside kernel
  - Part of making everything byte-oriented
  - Process is **blocked** while waiting for device
  - Let other processes run while gathering result
- Writes are buffered inside kernel
  - Complete in background (more later on)
  - Return to user when data is “handed off” to kernel

- This buffering is part of global buffer management and caching for block devices (such as disks)
  - Items typically cached in quanta of disk block sizes
  - We will have many interesting things to say about this buffering when we dive into the kernel

Low-Level I/O: Other Operations

- Operations specific to terminals, devices, networking, …
  - e.g., `ioctl`
- Duplicating descriptors
  - `int dup2(int old, int new);`
  - `int dup(int old);`
- Pipes – channel
  - `int pipe(int pipefd[2]);`
  - `Writes to pipefd[1] can be read from pipefd[0]`
- File Locking
- Memory-Mapping Files
- Asynchronous I/O

Today: 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 [if time]

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

};
```

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);
```

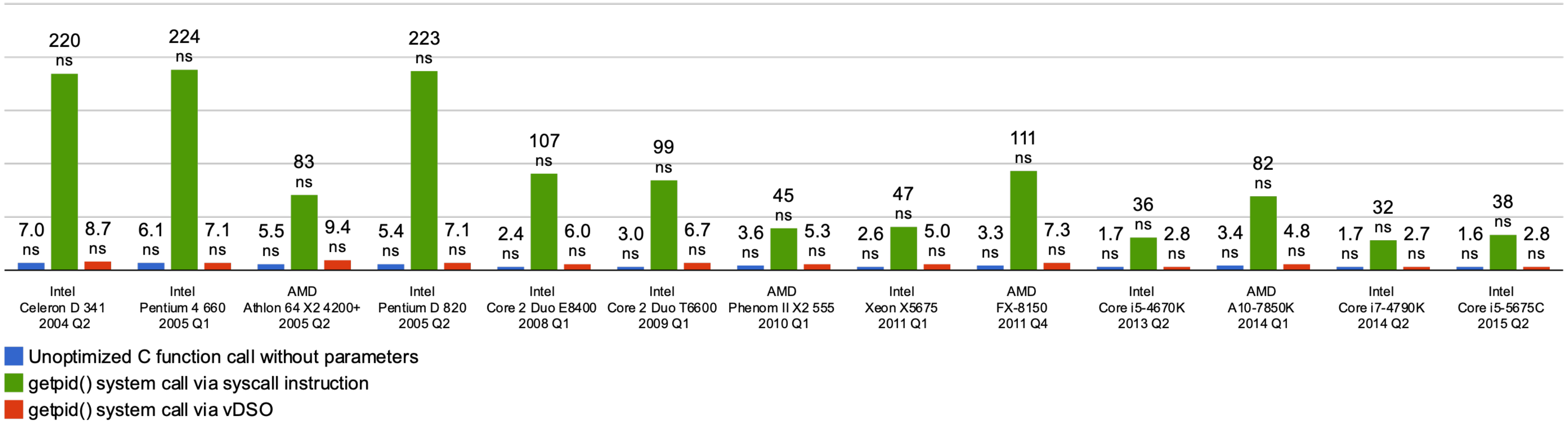
- Now, the call to fread will definitely see the latest write ‘b’

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 `read`s

Why Buffer in Userspace? Overhead!



- Syscalls are 25x more expensive than function calls (~100 ns)
  - This example about special shared-memory interface to the `getpid()` functionality, but point is the same!
- `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);
```

Today: 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

High Level I/O — *Streams*

Low Level I/O — *File Descriptors*

Syscall — *open(), read(), write(), close(), …*

Open File Descriptions

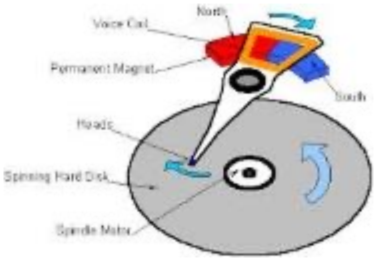
File System — *Files/Directories/Indexes*

I/O Driver — *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);
```

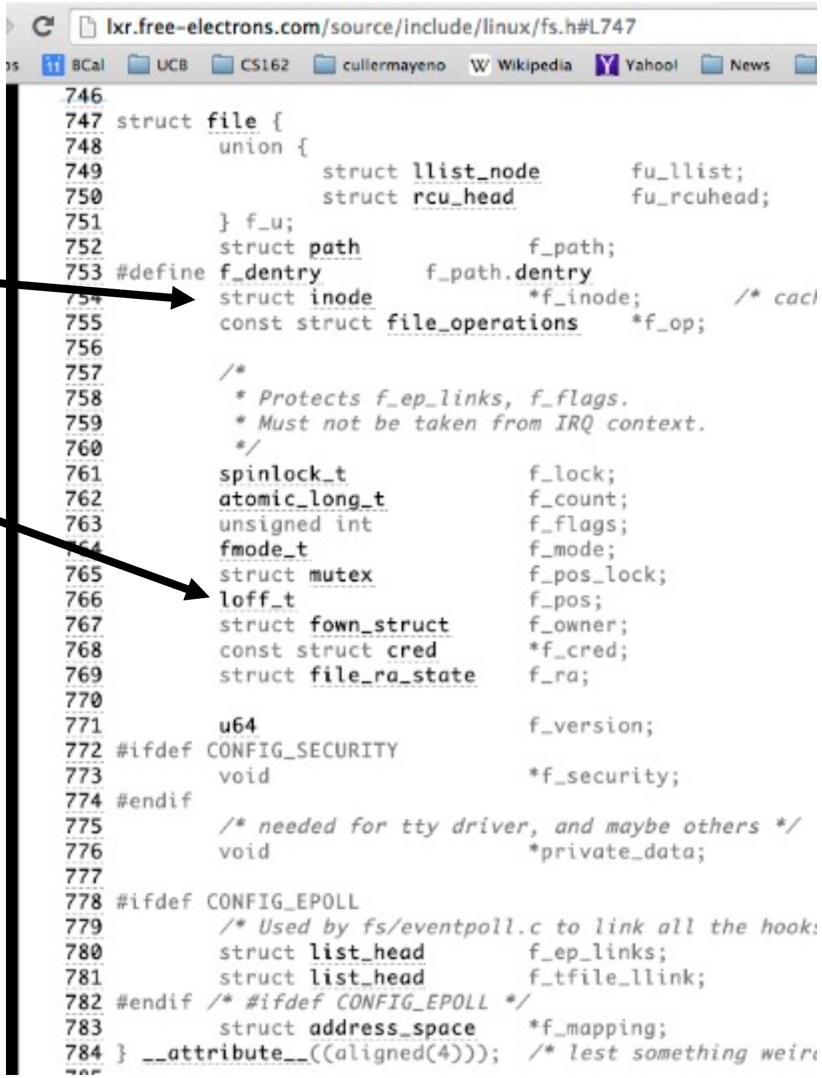
The kernel remembers that the int it receives (stored in fd) corresponds to foo.txt

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



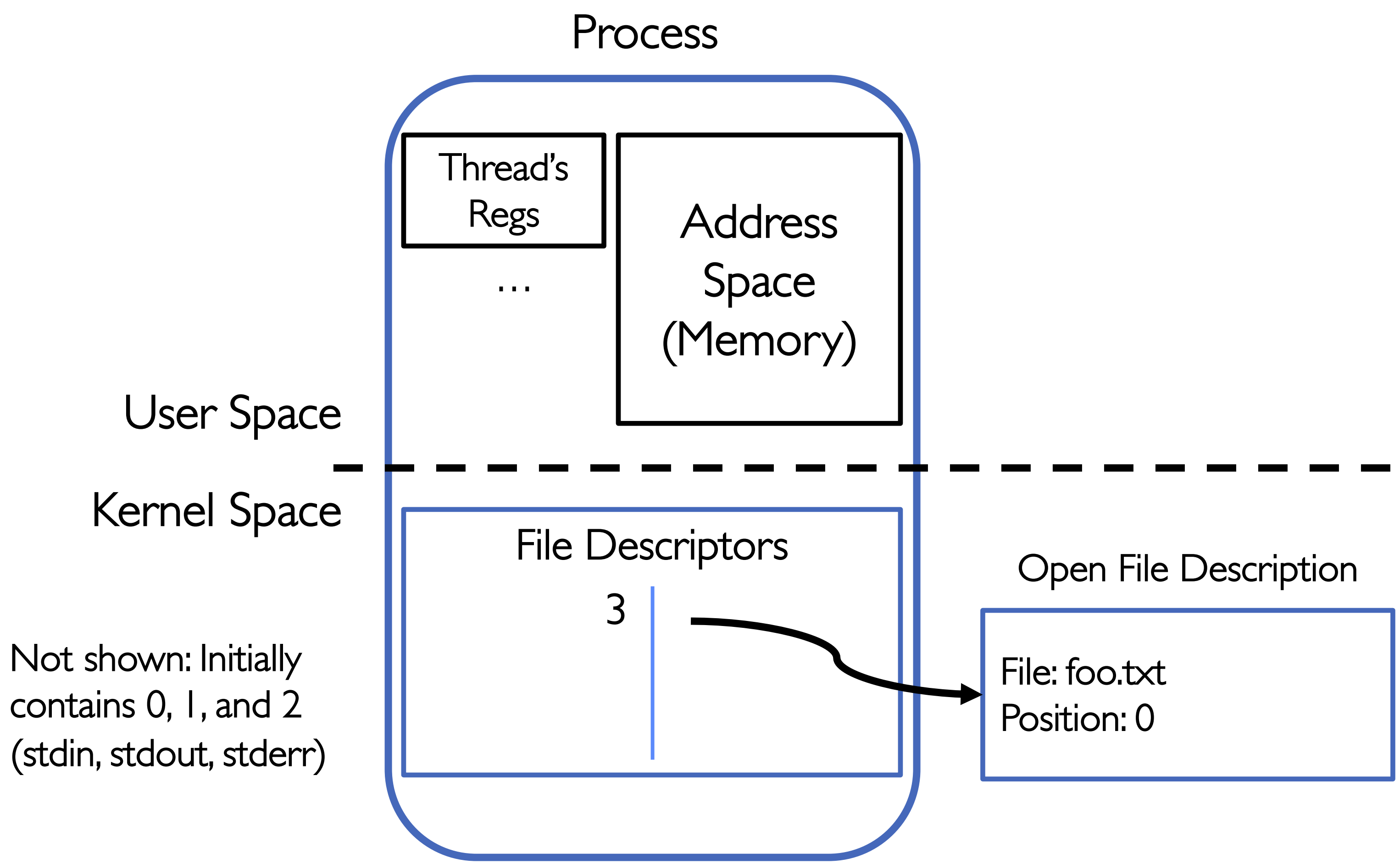

```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
        struct inode            *f_inode;       /* cac
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hook
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weir
```

Abstract Representation of a Process

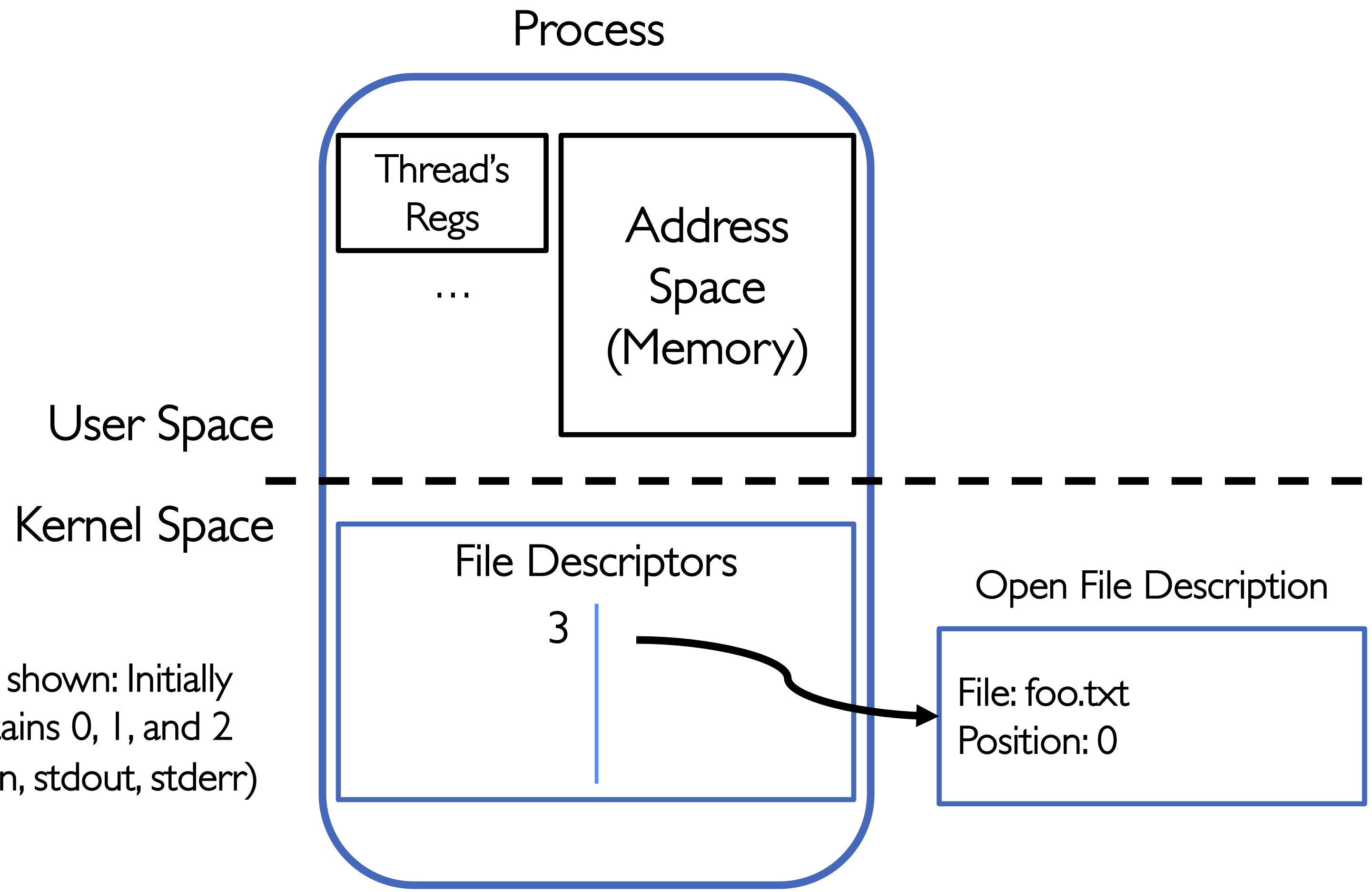


Suppose that we execute

open("foo.txt")

and that the result is 3

Abstract Representation of a Process

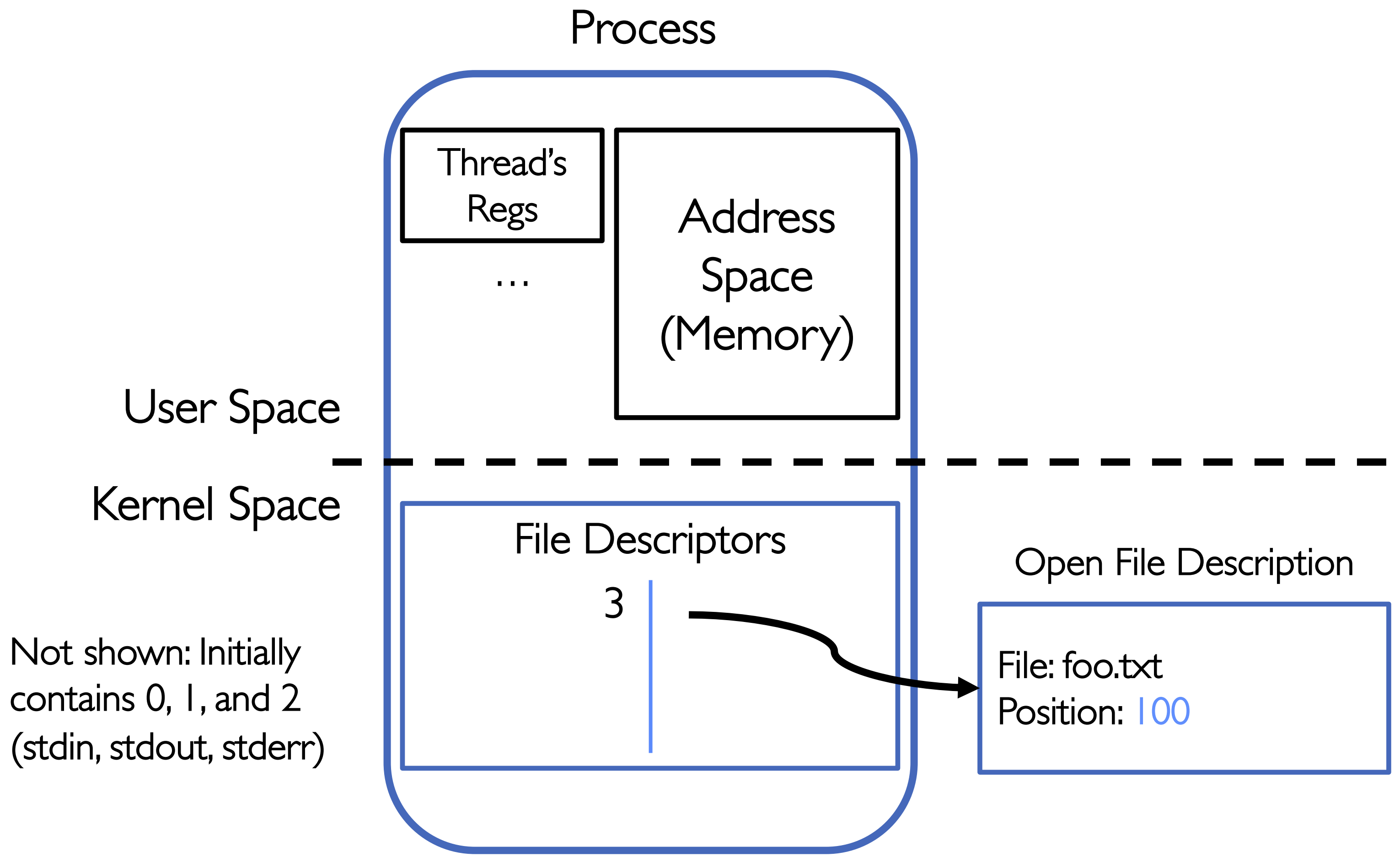


Not shown: Initially contains 0, 1, and 2 (stdin, stdout, stderr)

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

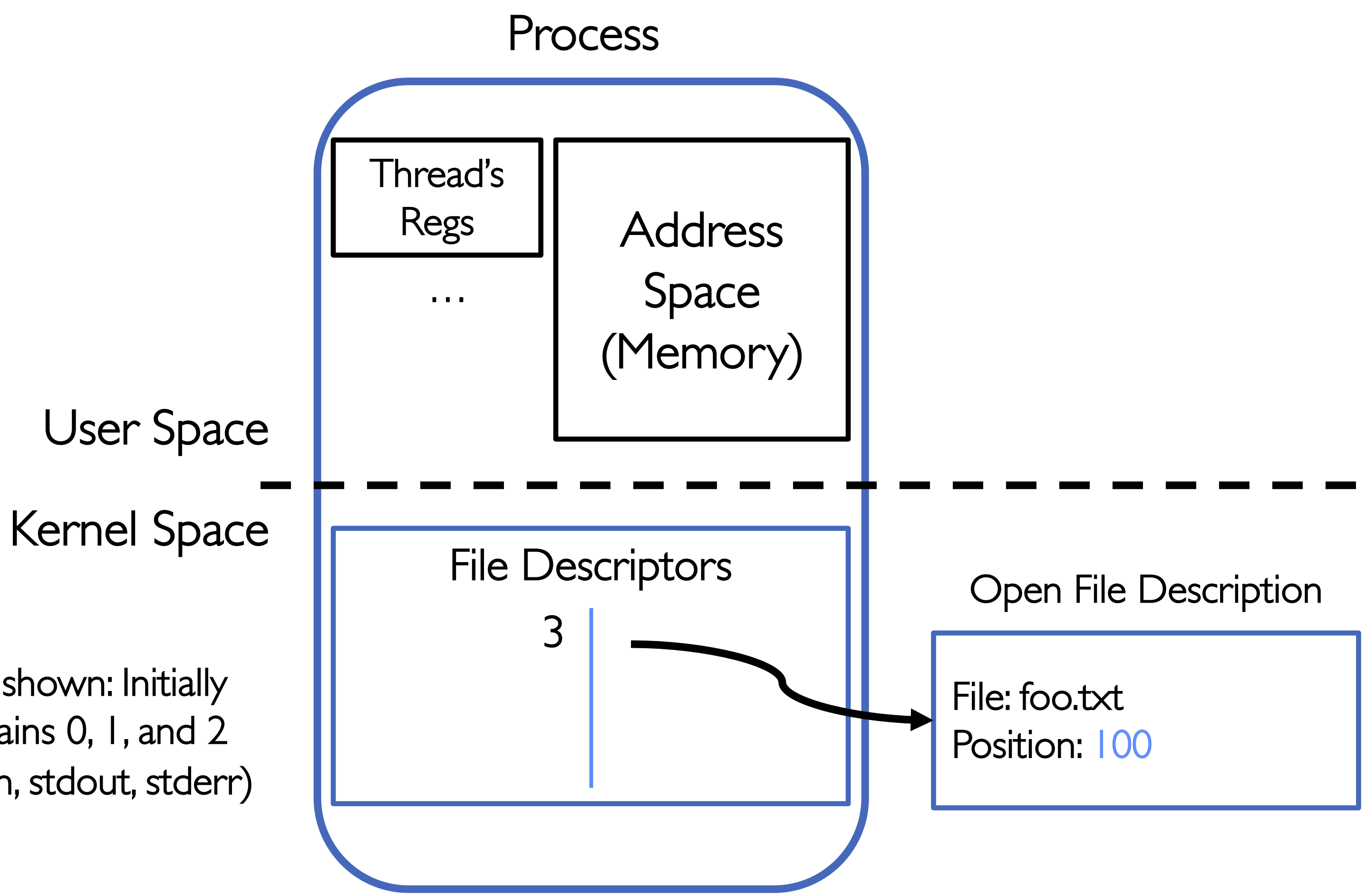
Abstract Representation of a Process



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



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

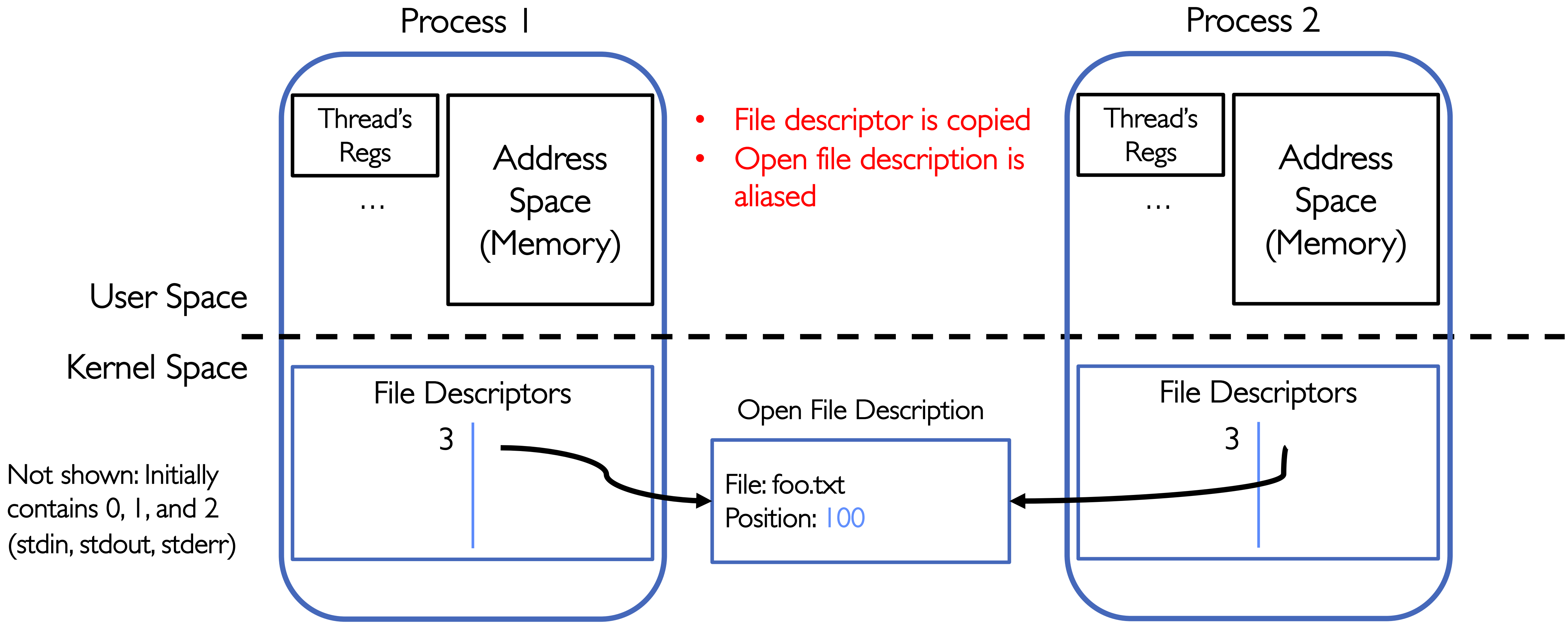
Suppose that we execute
open("foo.txt")
and that the result is 3

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

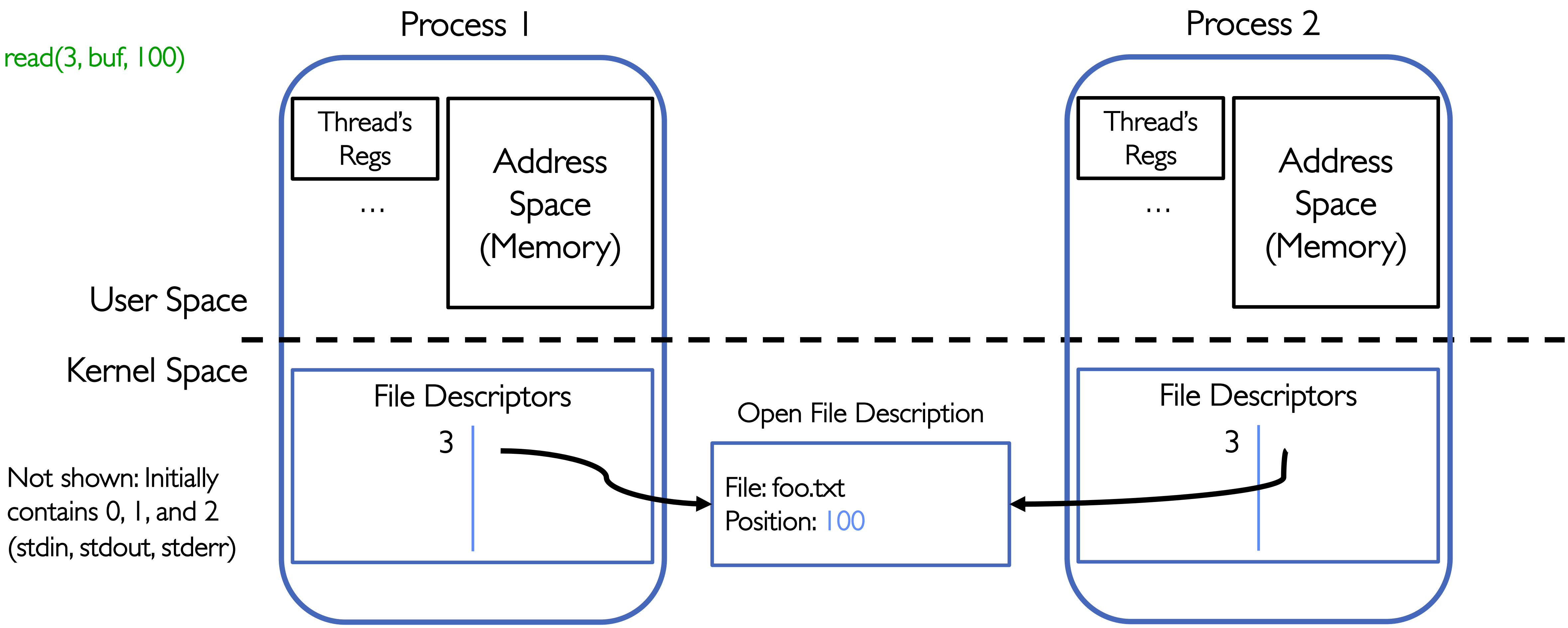
Finally, suppose that we execute
close(3)

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

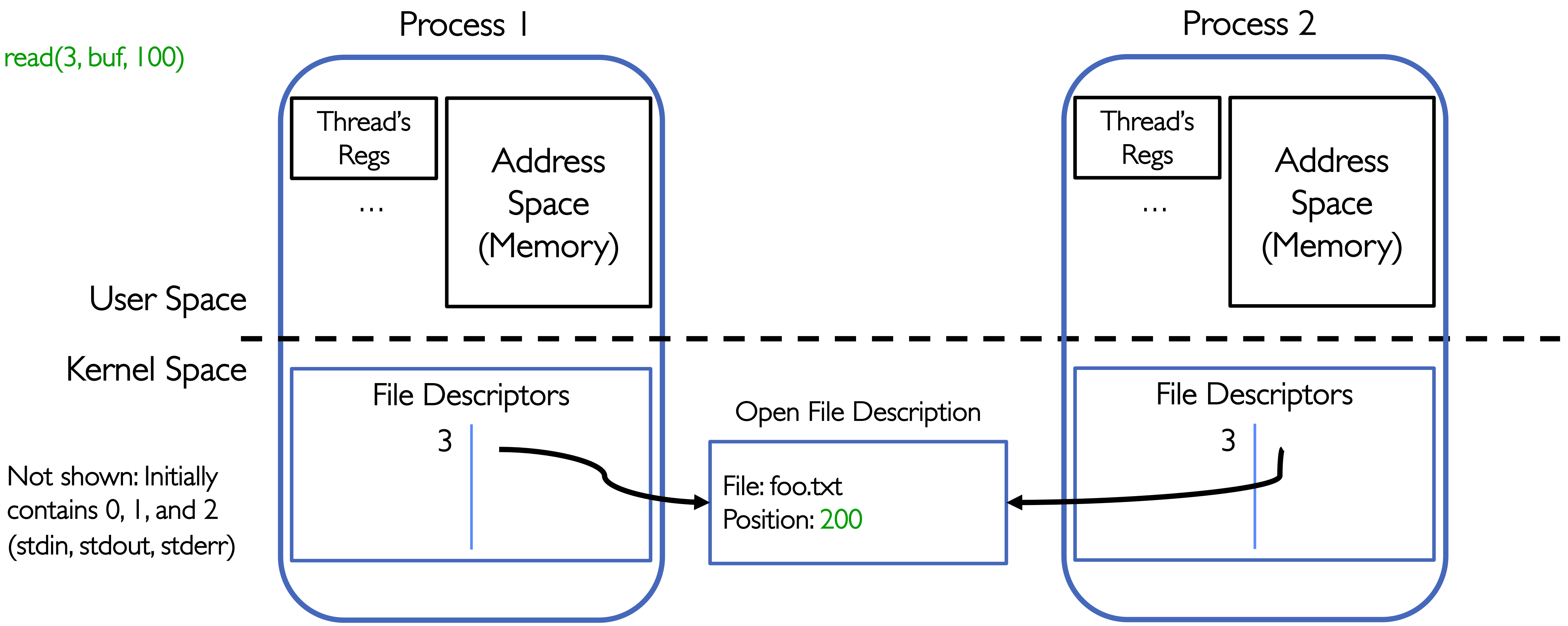
- File descriptor is copied
- Open file description is aliased



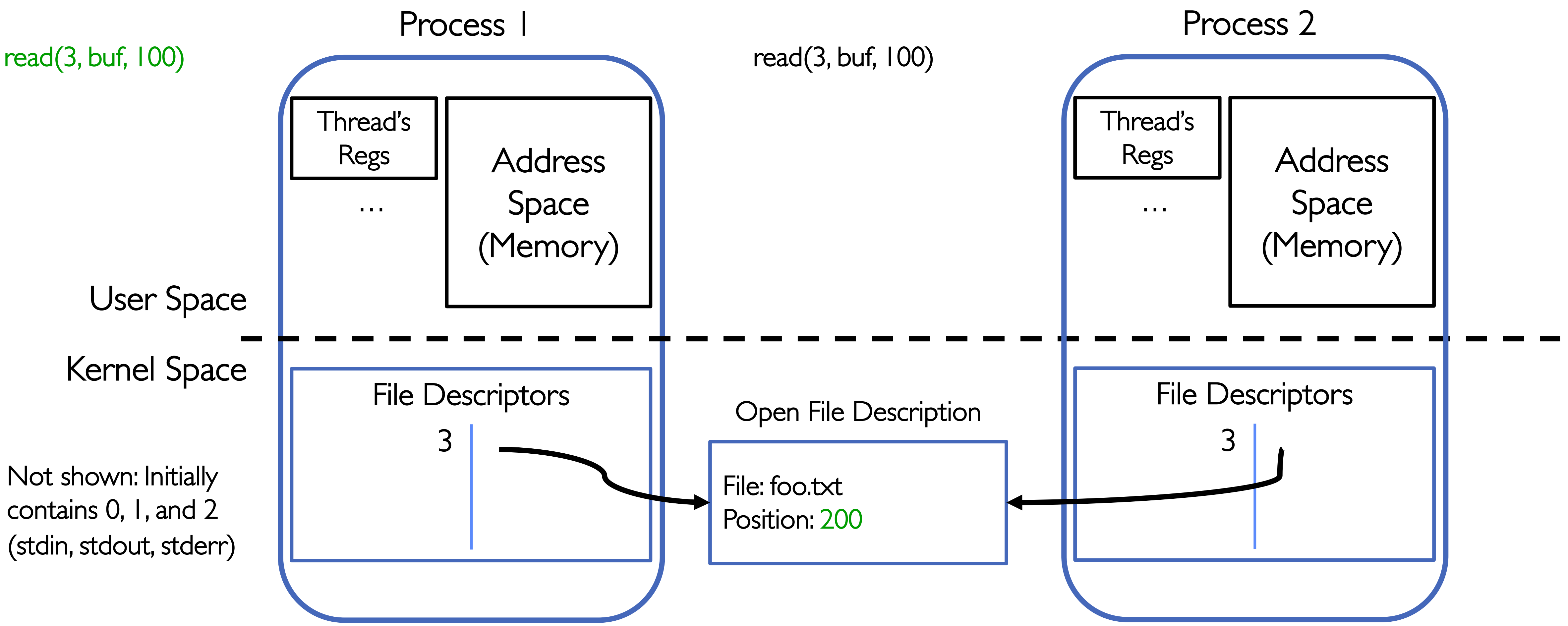
Open File Description is *Aliased*



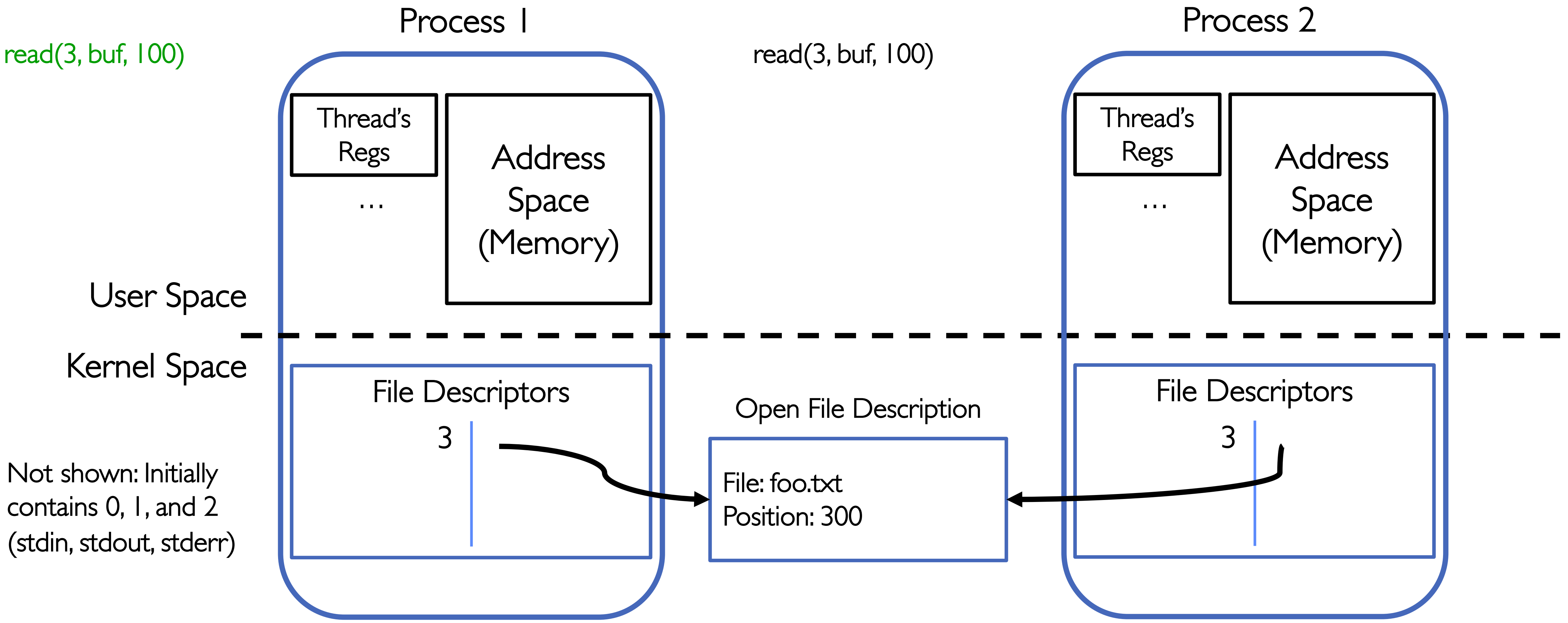
Open File Description is *Aliased*



Open File Description is *Aliased*

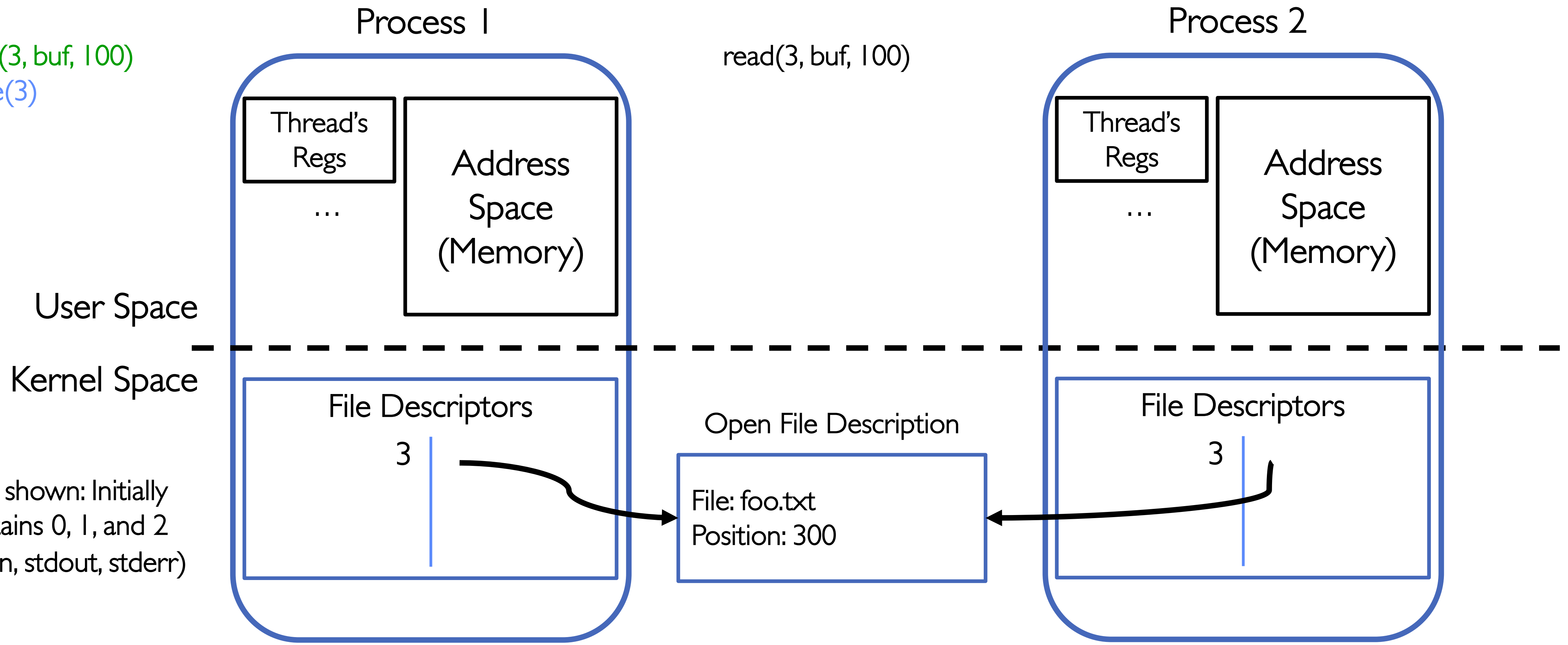


Open File Description is *Aliased*



File Descriptor is *Copied*

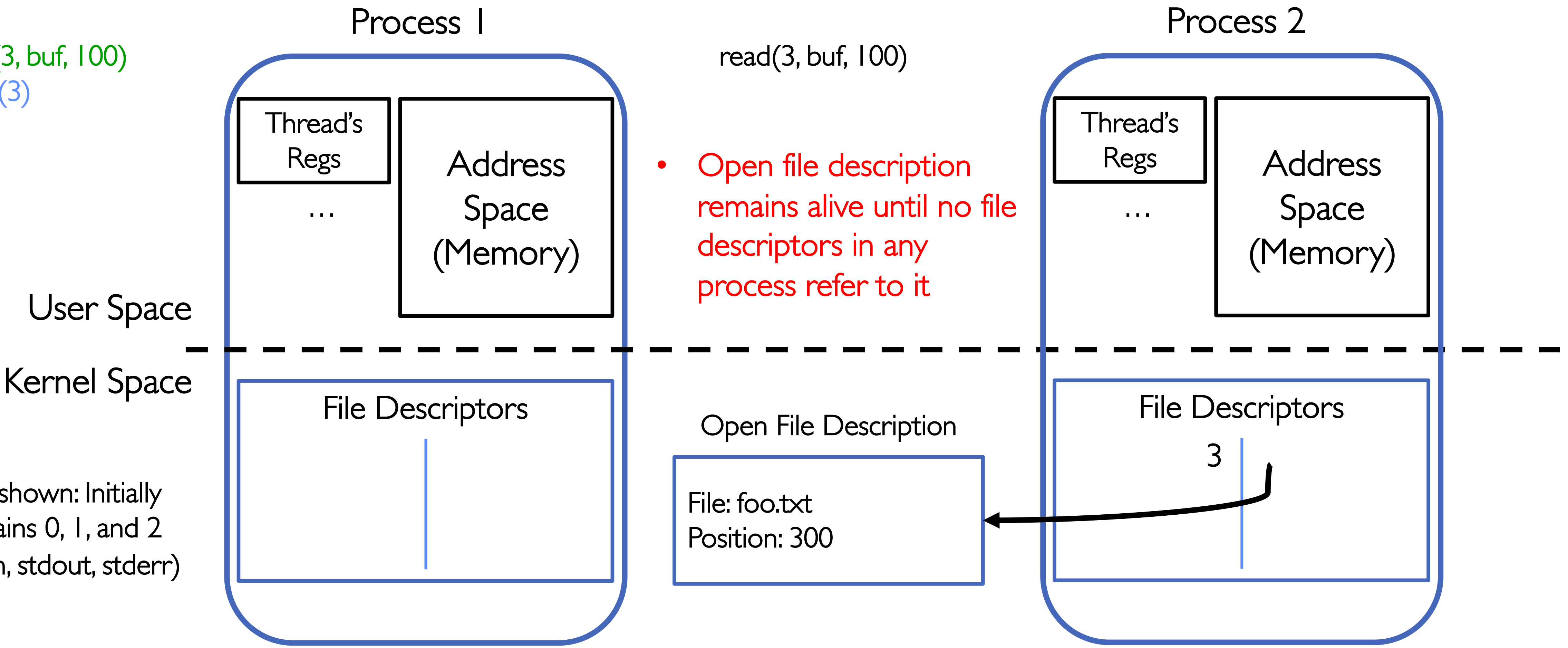
read(3, buf, 100)
close(3)



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

File Descriptor is *Copied*

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



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

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

Why is Aliasing the Open File Description a Good Idea?

- It allows for *shared resources* between processes

Recall: In POSIX, Everything is a "File"

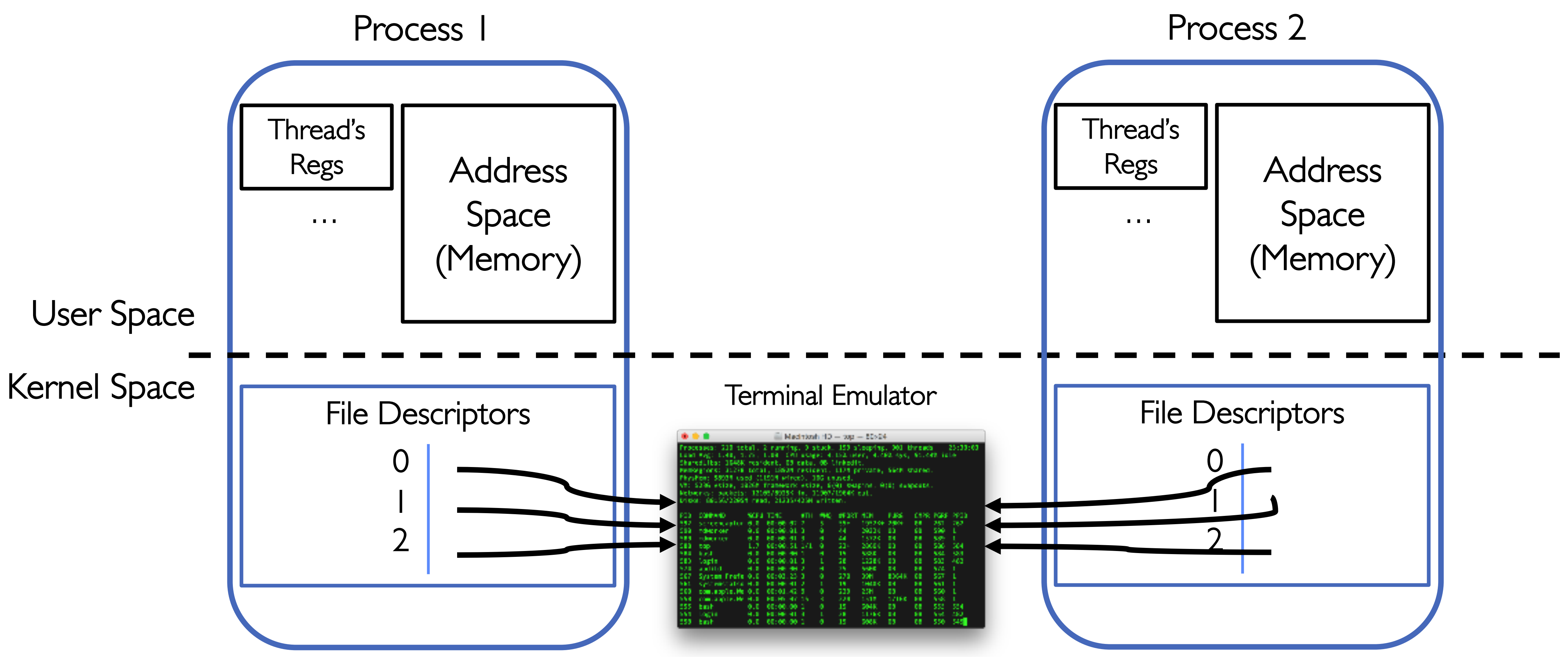
- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Networking (sockets)
  - Local interprocess communication (pipes, sockets)

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

Example: Shared Terminal Emulator

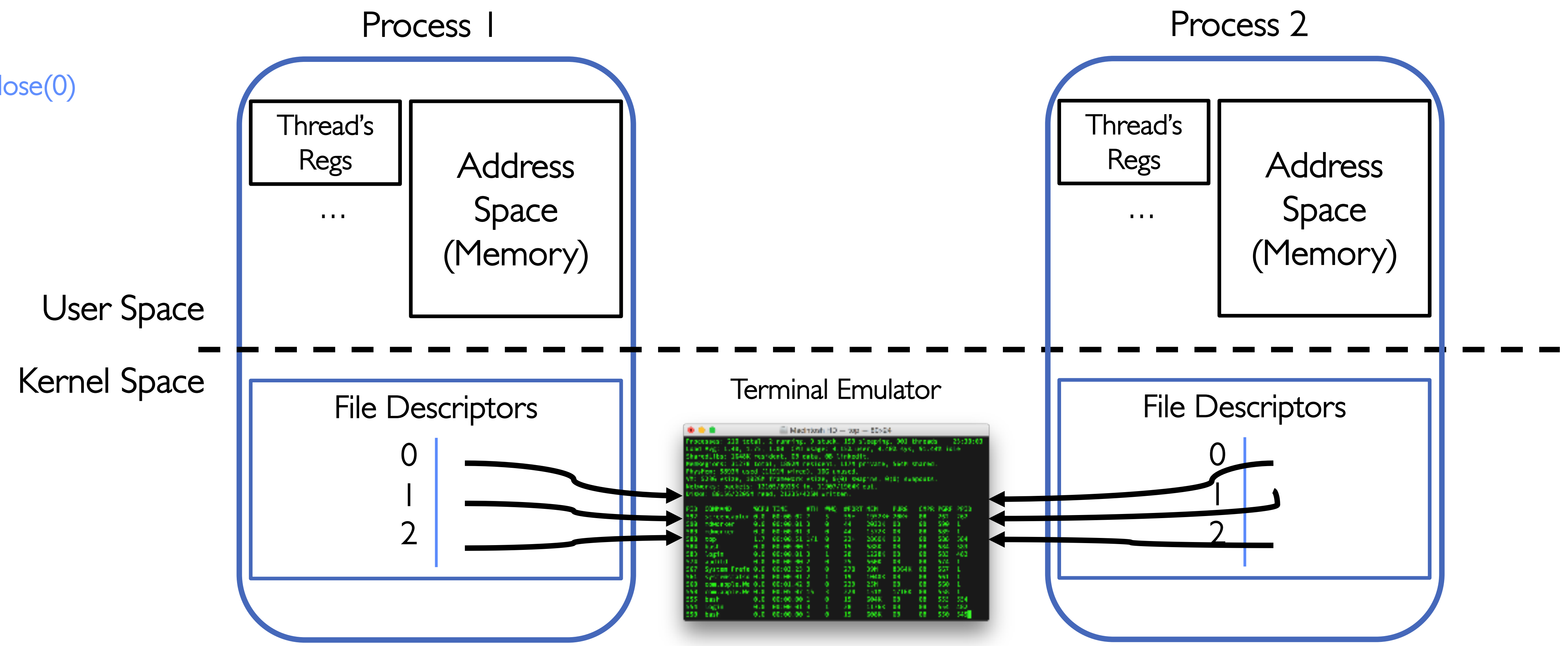
- When you `fork()` a process, the parent's and child's `printf` outputs go to the same terminal

Example: Shared Terminal Emulator



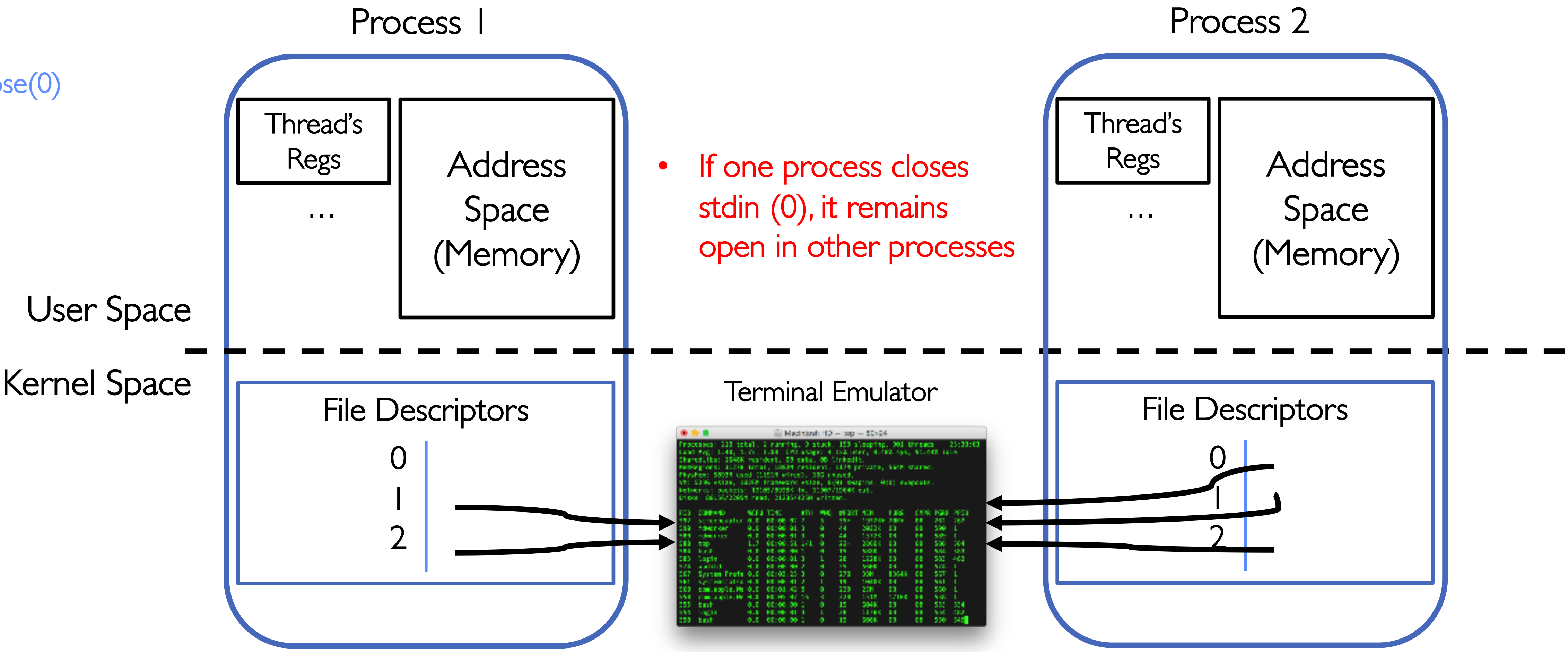
Example: Shared Terminal Emulator

close(0)



Example: Shared Terminal Emulator

close(0)



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

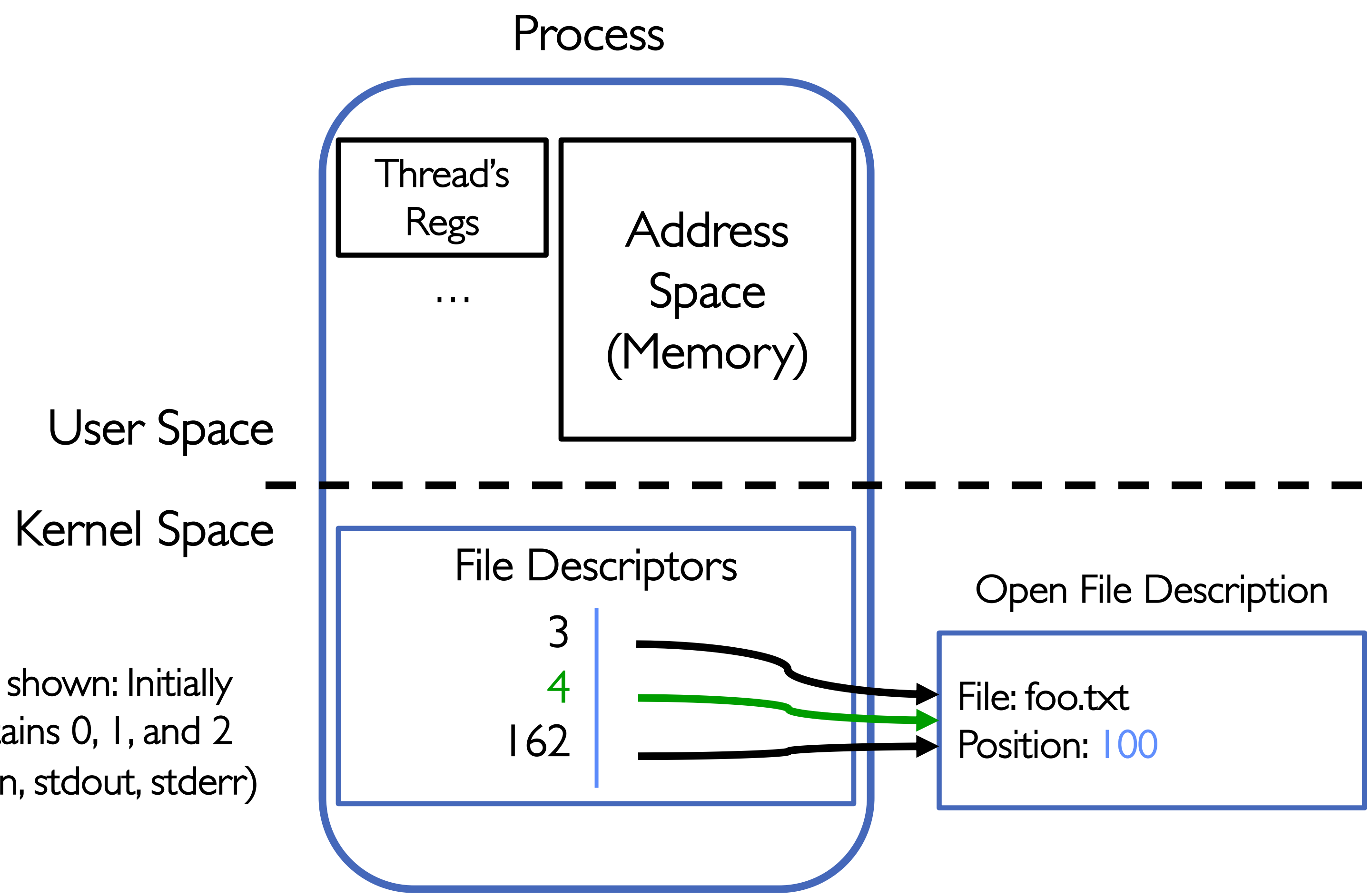
Other Examples

- 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



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

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

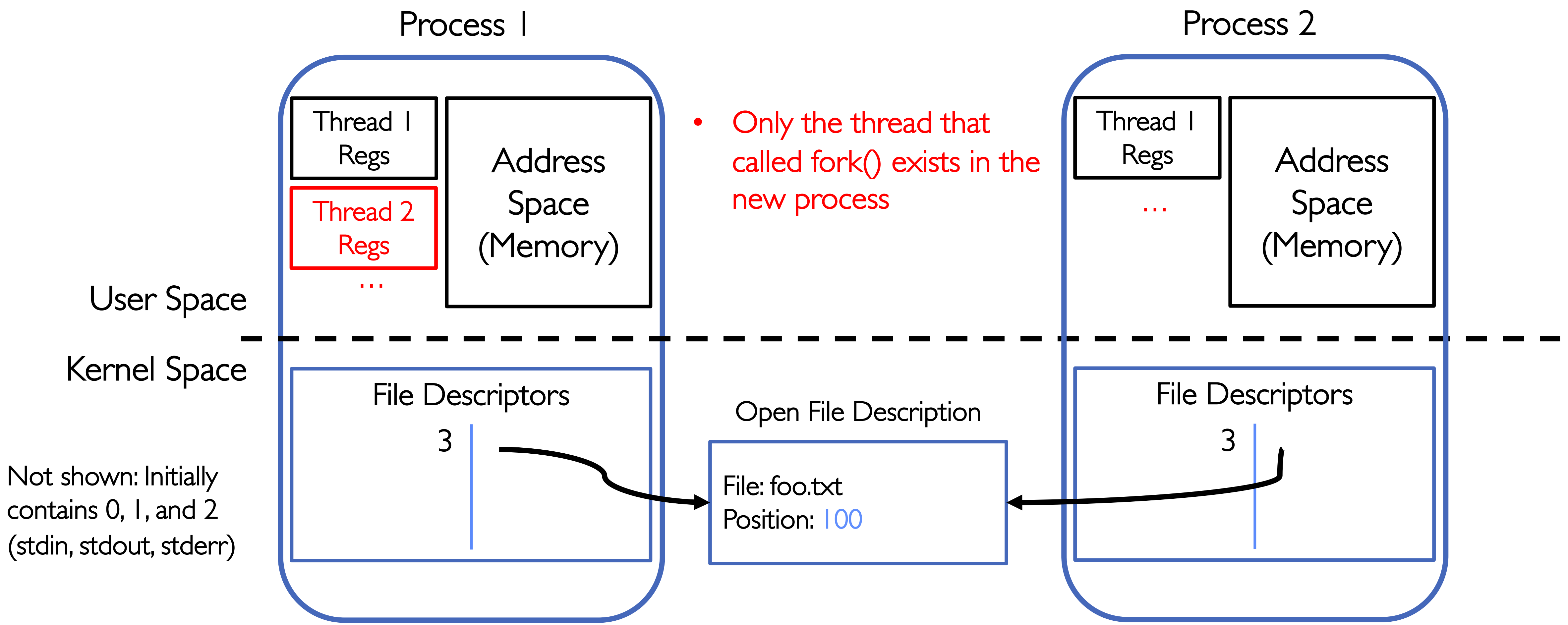
Unless you plan to call `exec()` in the child process

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



- Only the thread that called fork() exists in the new process

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!

Conclusion

- POSIX idea: “everything is a file”
- All sorts of I/O managed by open/read/write/close