CS310 Operating Systems

Lecture 37 : File System System – 5
File Descriptor Manipulation

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley

Reading

- Book: Linux System Programming: talking directly to the kernel and C library, by Robert Love
- Class presentation: University of California, Berkeley, CS162

Previous Classes

C library High level APSs vs Linux Syscalls

- A C library function is a function implemented by the C library implementation
  - C interface the library provides programmers to access kernel related functions
- If a C programmers directly uses file syscalls, he/she needs to read the documentation
- Each syscall has a number of arguments – makes programming complicated
  - A user most of the time may not use all arguments
  - User space programs are expected to find out arguments for each syscall by for example inspecting the documentation
- A C library function is a function implemented by the C library implementation

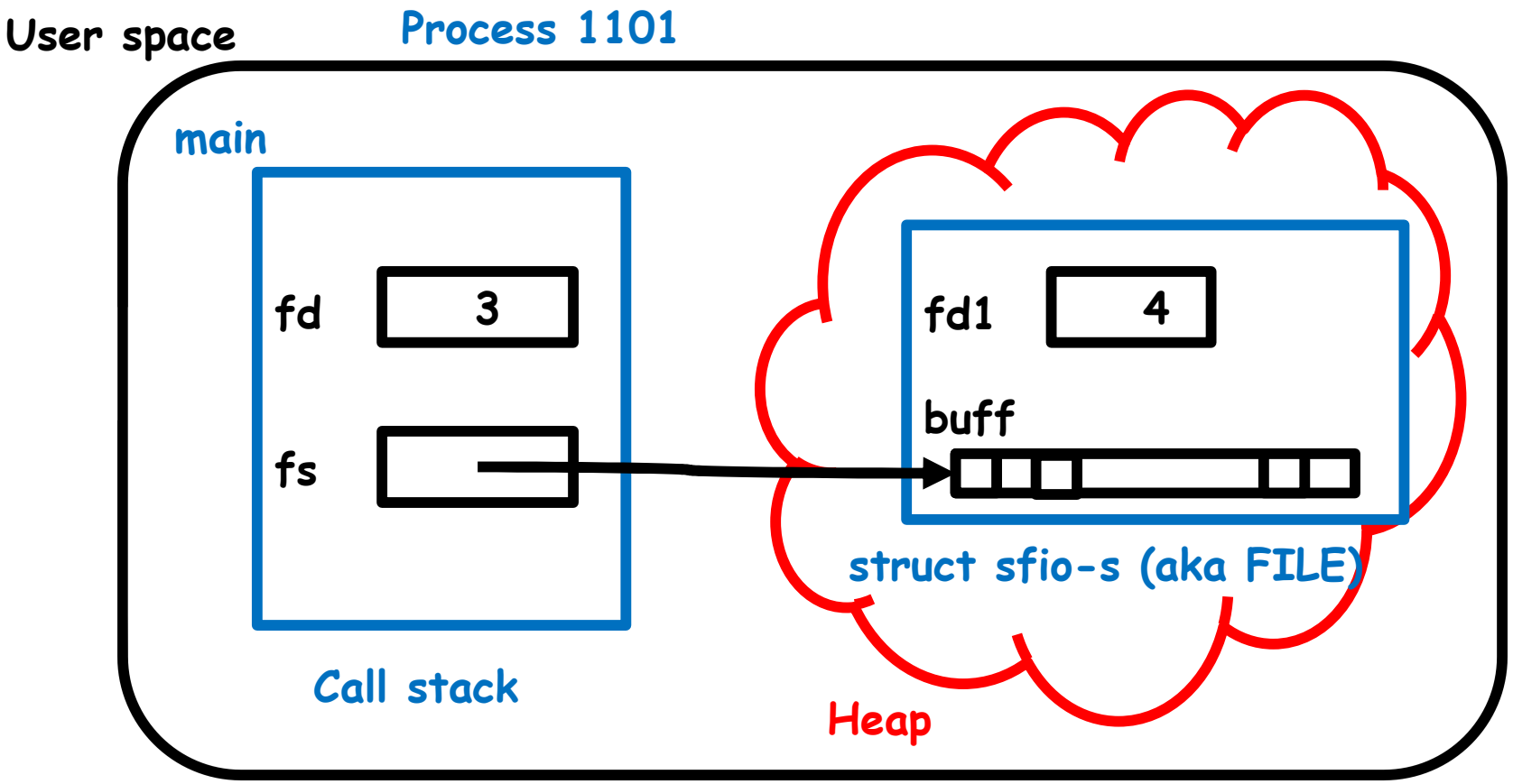
Opening a file

```
#include <stdio.h>
FILE *fopen( const char *path, const char *mode );
```

This function opens the file path with the behavior given by mode and associates a new stream with it.

read() syscall vs fread() API - User space

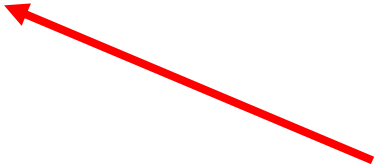
```
Int main(){
    int fd = open("foo.txt", "O_RDONLY");        // for I/O syscall
   FILE *fs = fopen('bar.txt", "w");             // for C lib I/O
```



Kernel Maintains State

```
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 fd corresponds to foo.txt



The kernel picks up where it left off in the file

State Maintained by the Kernel

On a successful call to `open()`:

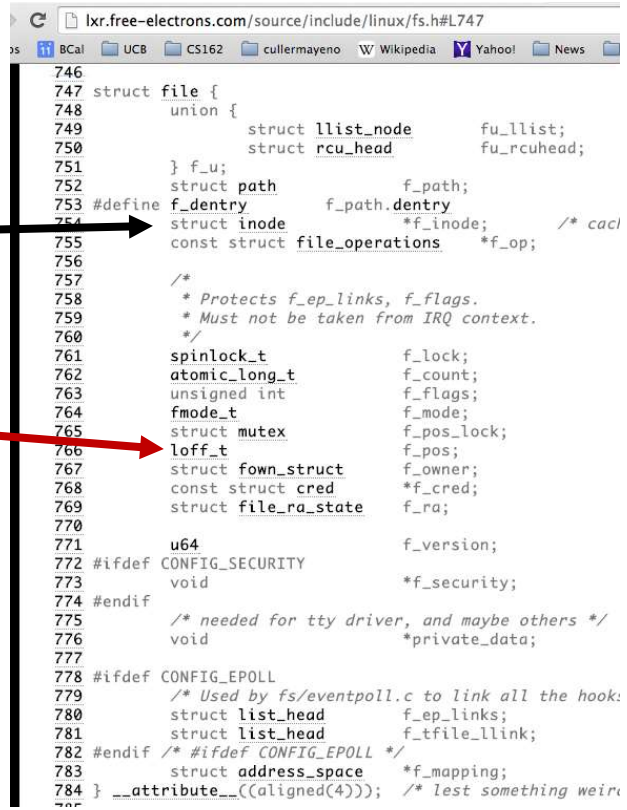
- A *file descriptor* (int) is returned to the user
- An *open file description* is created in the kernel

- For each process, the kernel maintains a mapping from *file descriptor* to *open file description*
- On future system calls (e.g., `read()`), the kernel looks up the open file description corresponding to the provided file descriptor and uses it to service the system call

What's in an Open File Description?

For our purposes, the two most important things are:

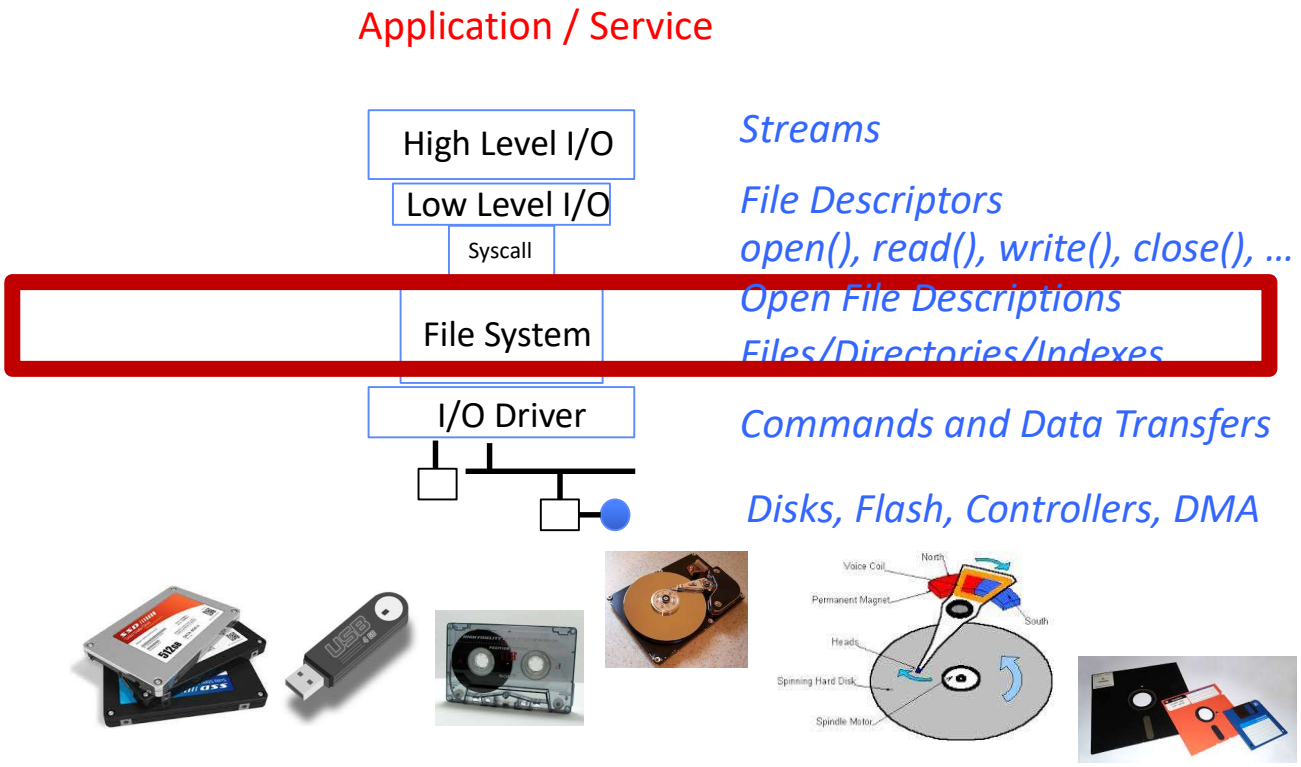
- Where to find the file data on disk
- The current position within the file



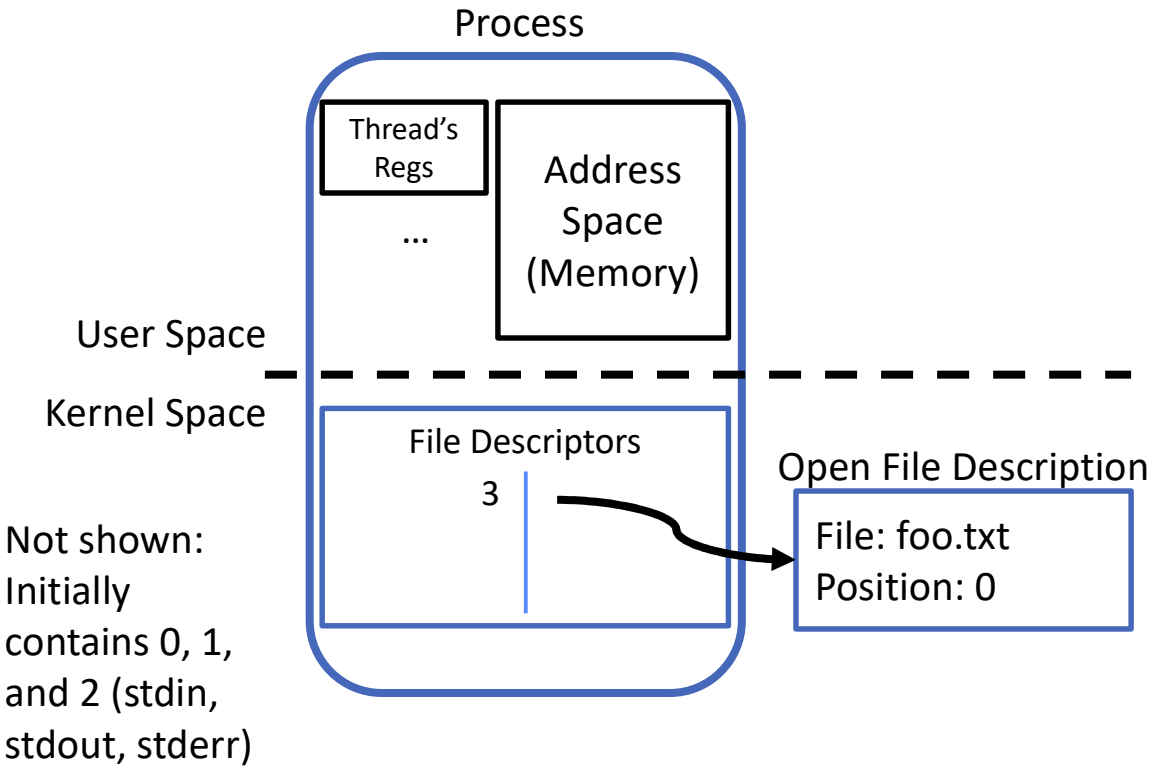
Today, we will study

- Use of file descriptor within a process or across multiple processes

Today, we will study..



Abstract Representation of a Process

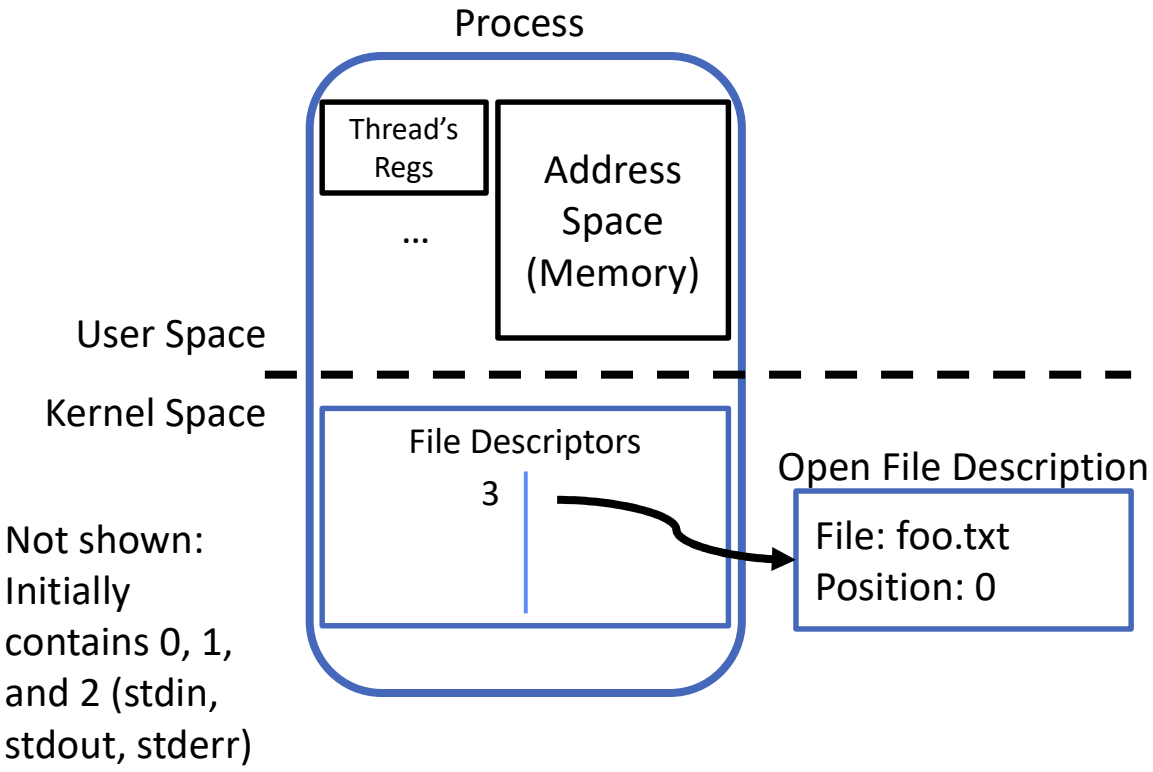


Suppose that we execute

`open("foo.txt")`

and that the result is 3

Abstract Representation of a Process



Suppose that we execute

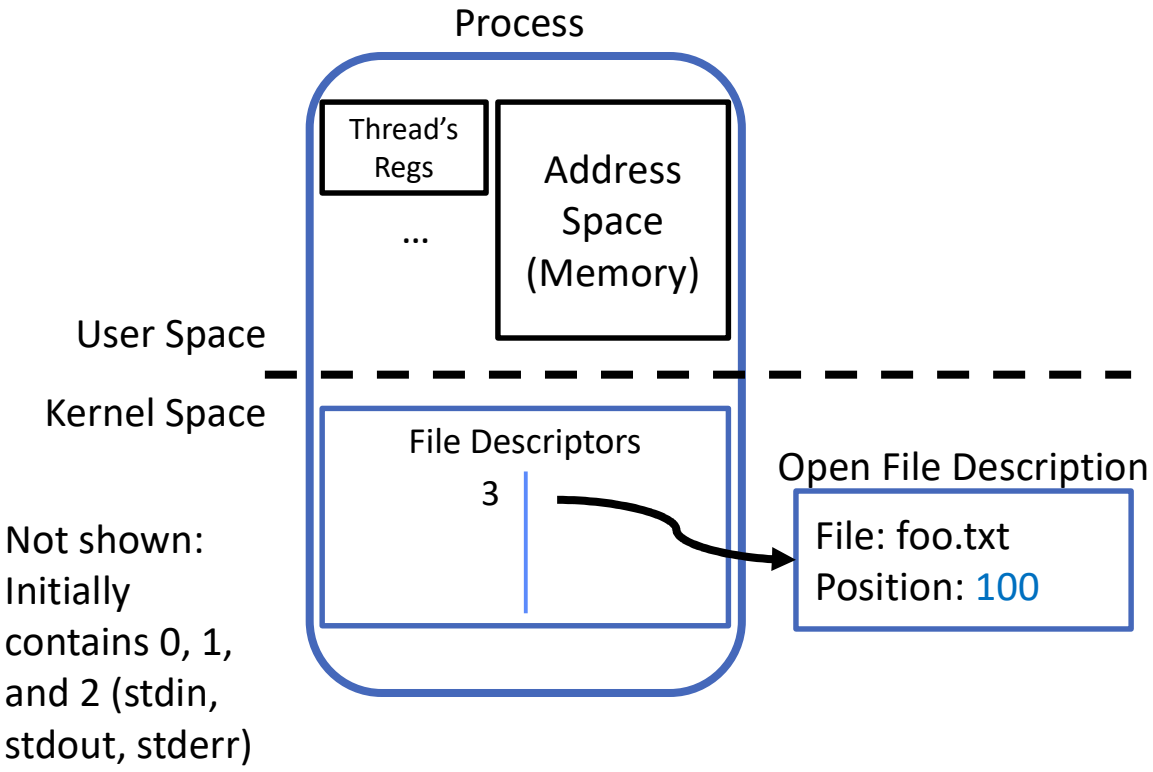
`open("foo.txt")`

and that the result is 3

Next, suppose that we execute

`read(3, buf, 100)`

Abstract Representation of a Process



Suppose that we execute

`open("foo.txt")`

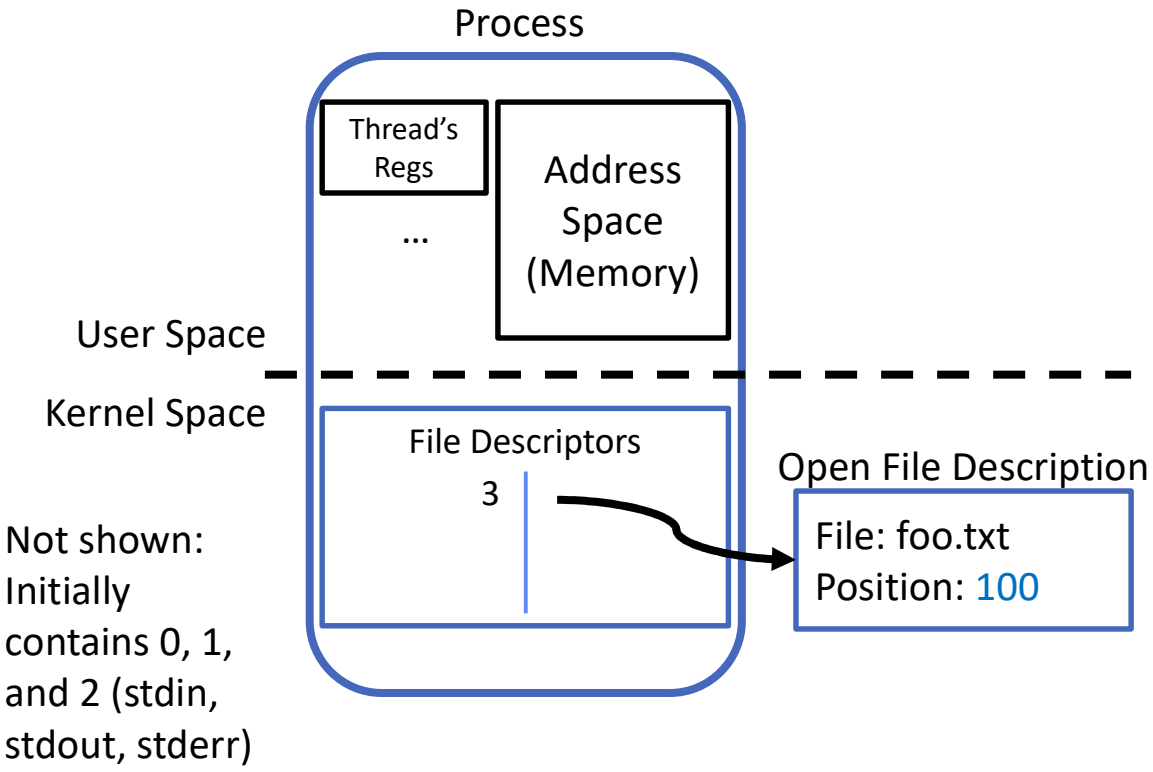
and that the result is 3

Next, suppose that we execute

`read(3, buf, 100)`

The file position is 100

Abstract Representation of a Process



Suppose that we execute

`open("foo.txt")`

and that the result is 3

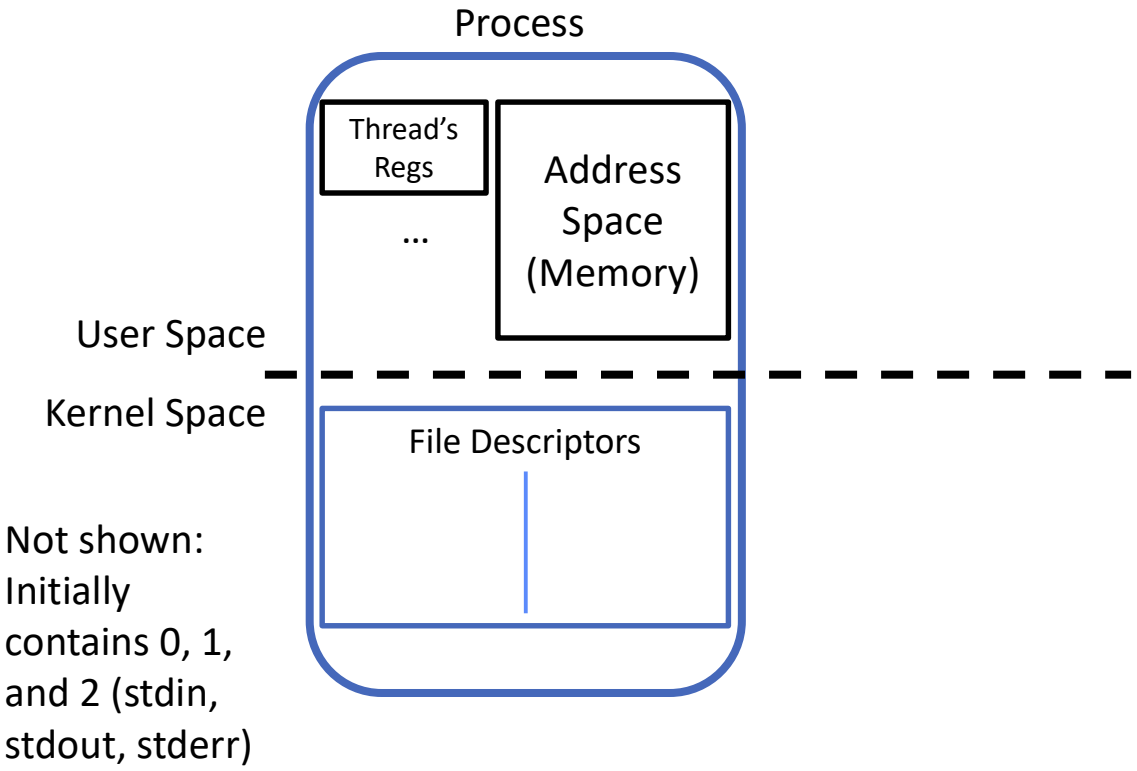
Next, suppose that we execute

`read(3, buf, 100)`

Finally, suppose that we execute

`close(3)`

Abstract Representation of a Process



Suppose that we execute

`open("foo.txt")`

and that the result is 3

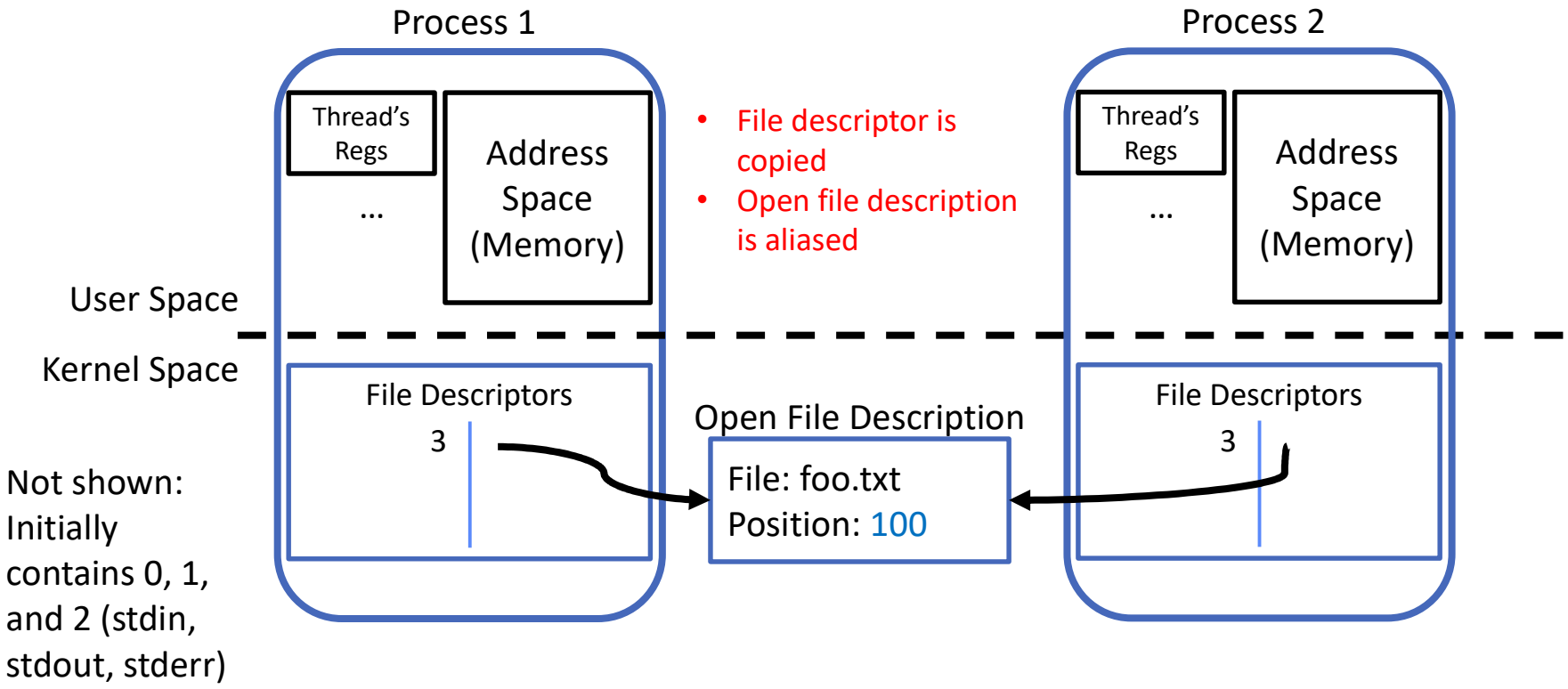
Next, suppose that we execute

`read(3, buf, 100)`

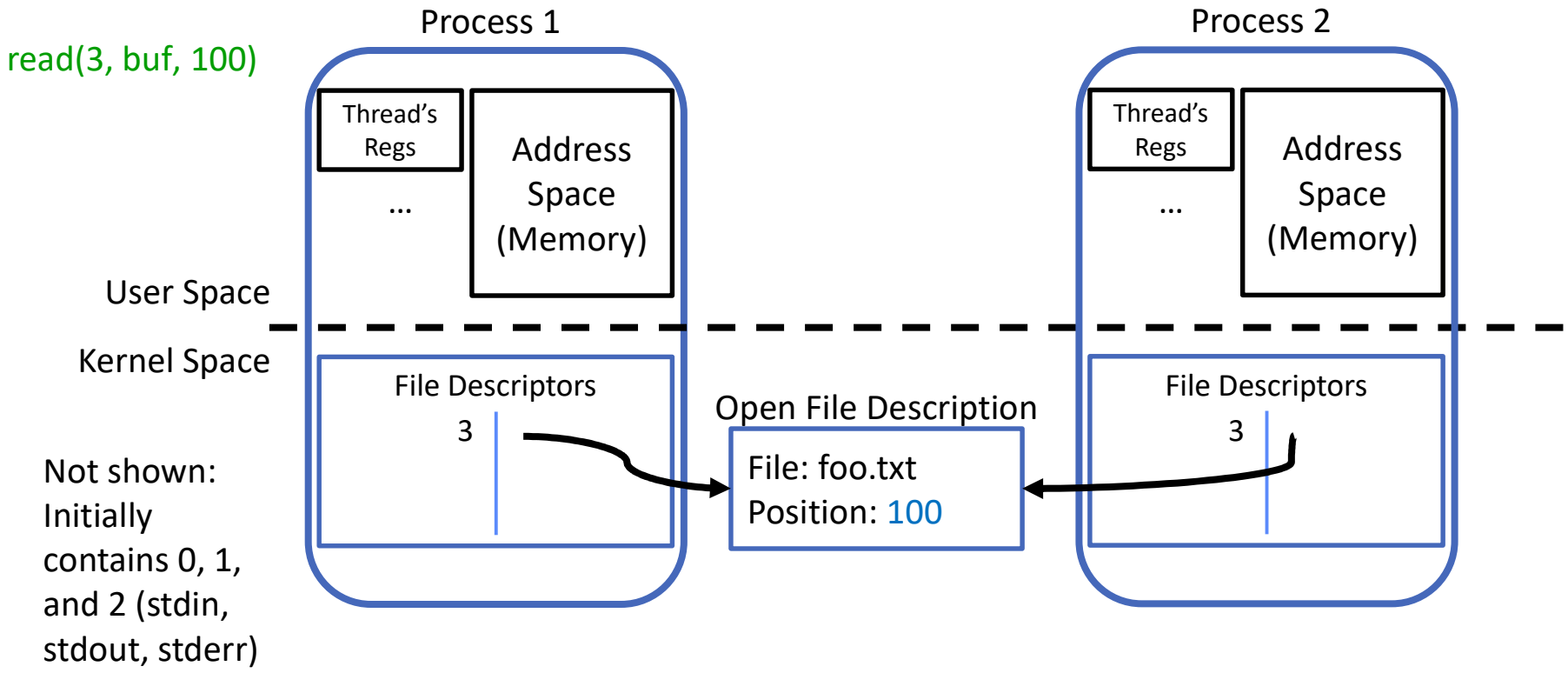
Finally, suppose that we execute

`close(3)`

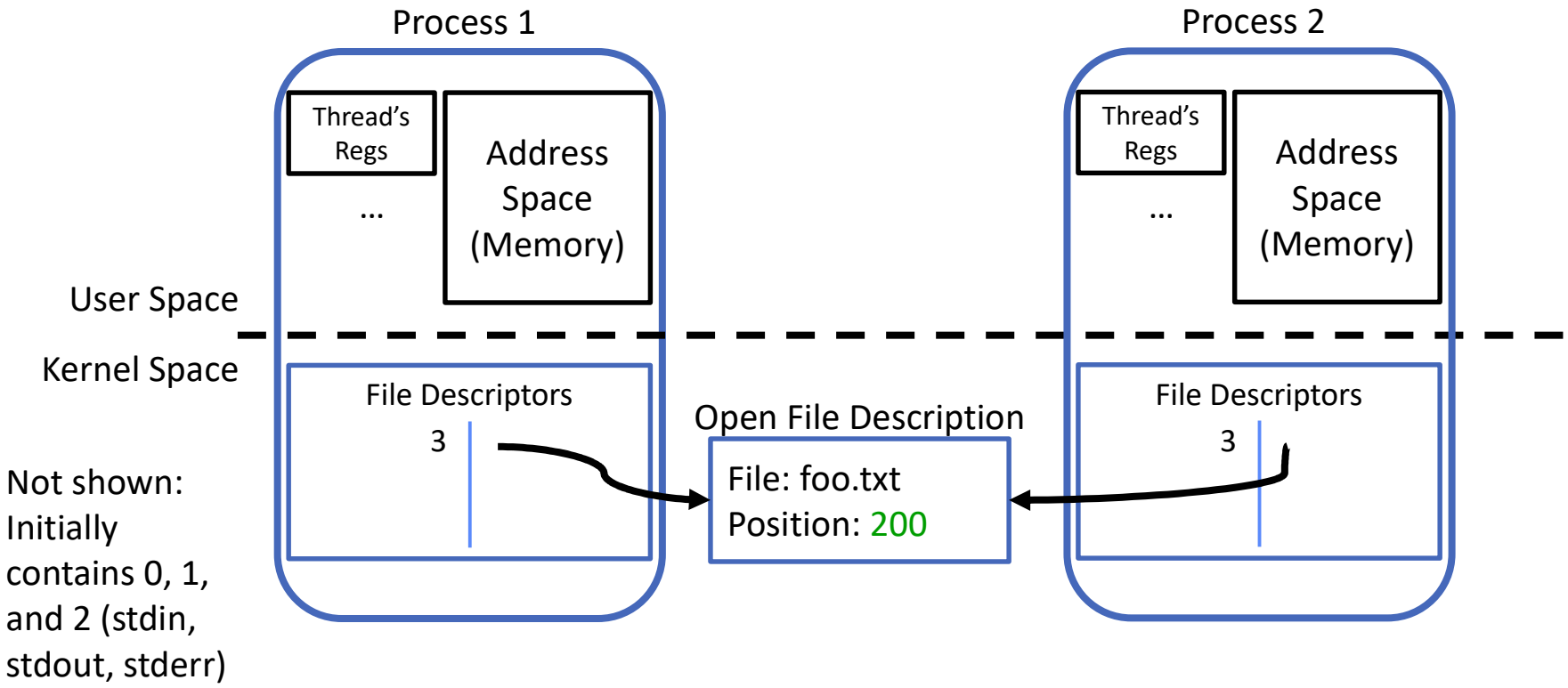
Now, let's fork()!



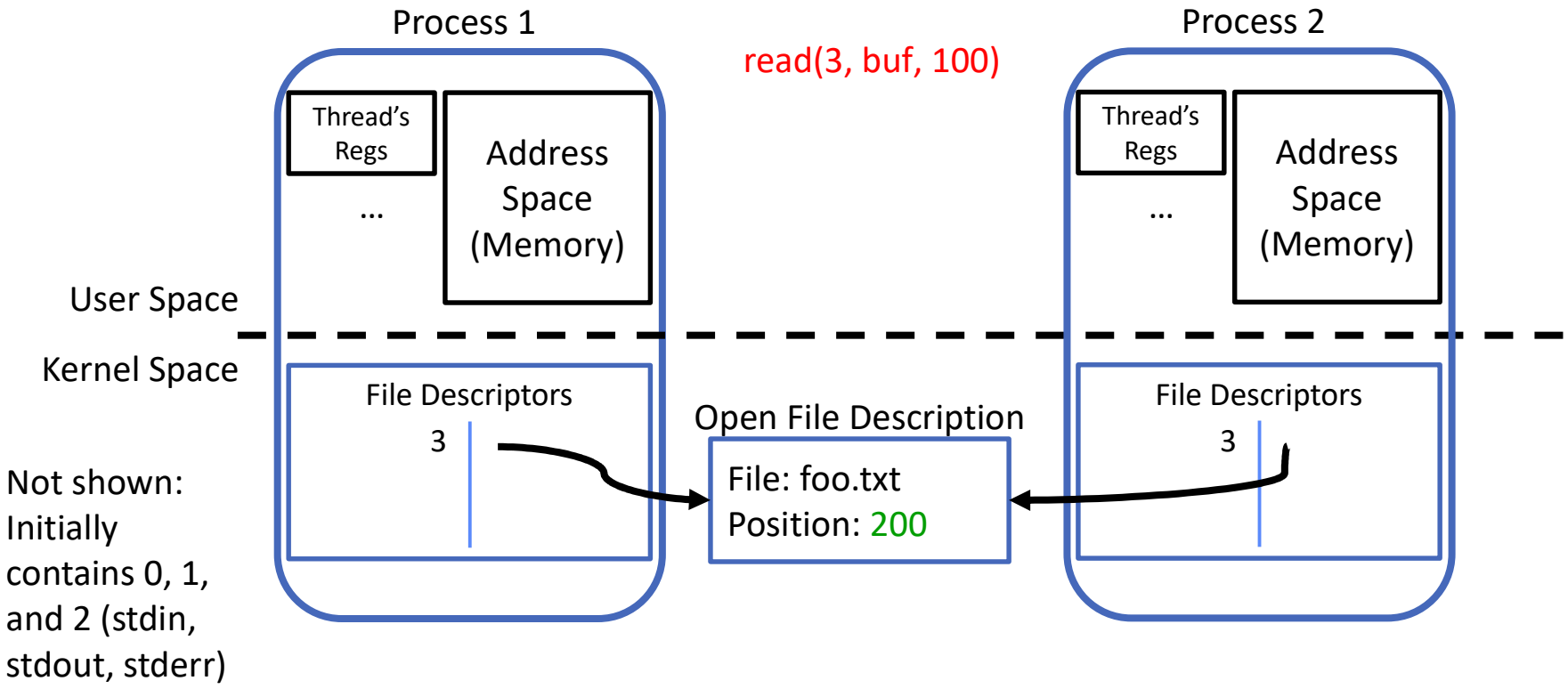
Open File Description is *Aliased*



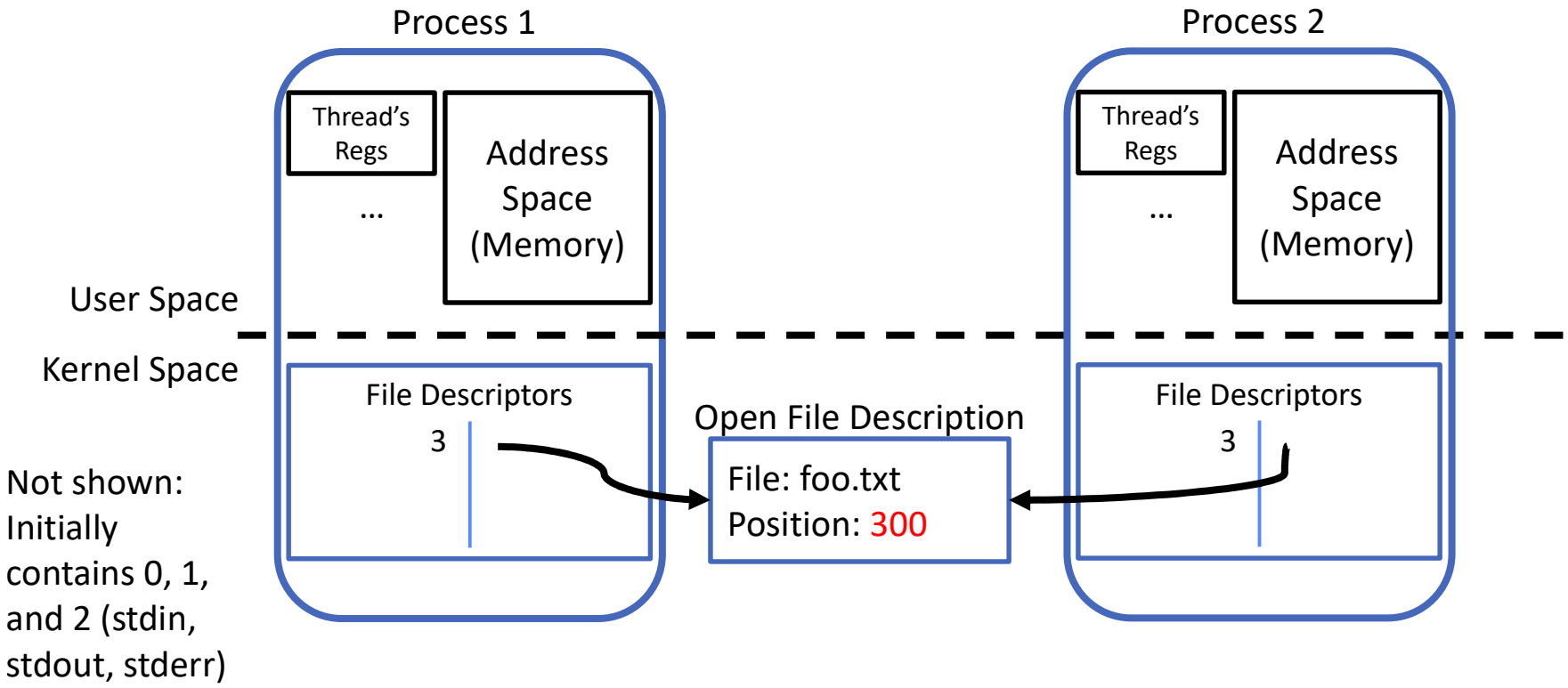
Open File Description is *Aliased*



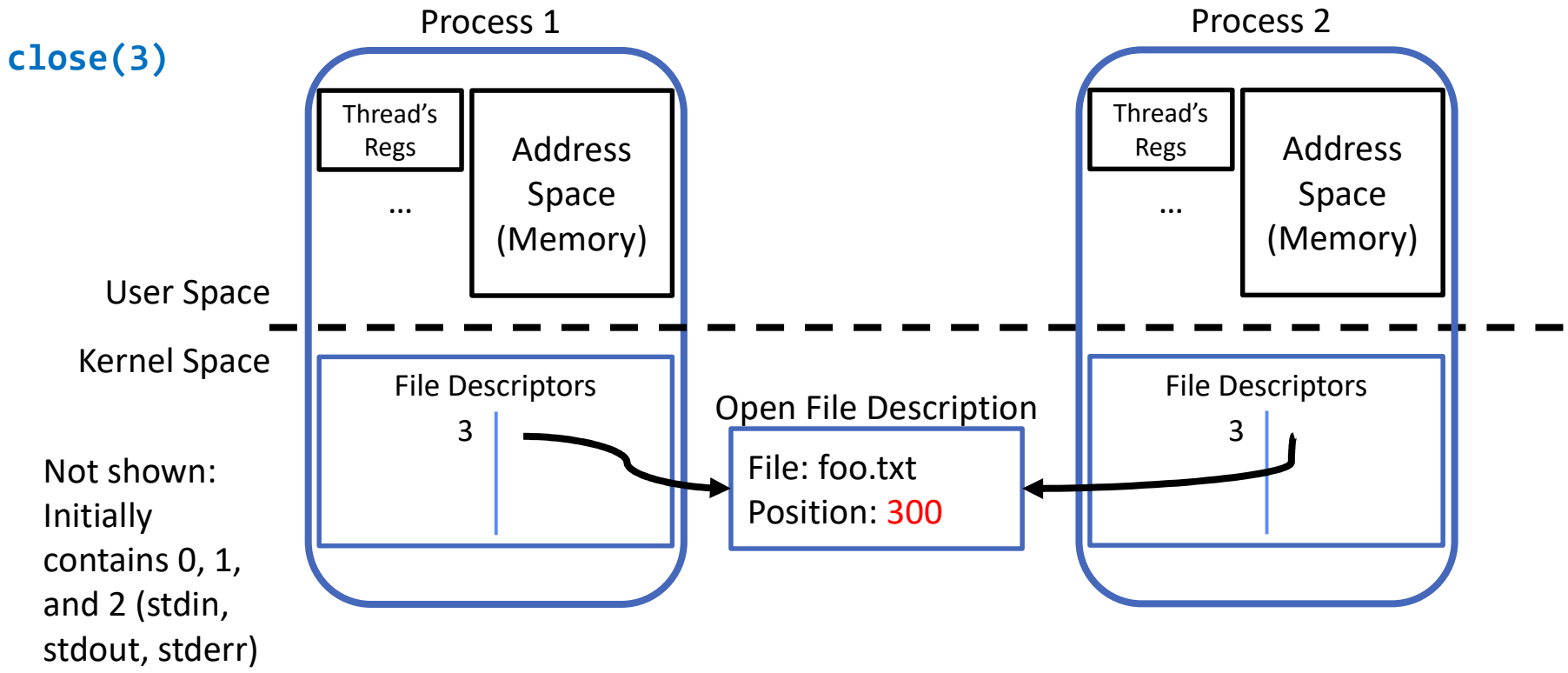
Open File Description is *Aliased*



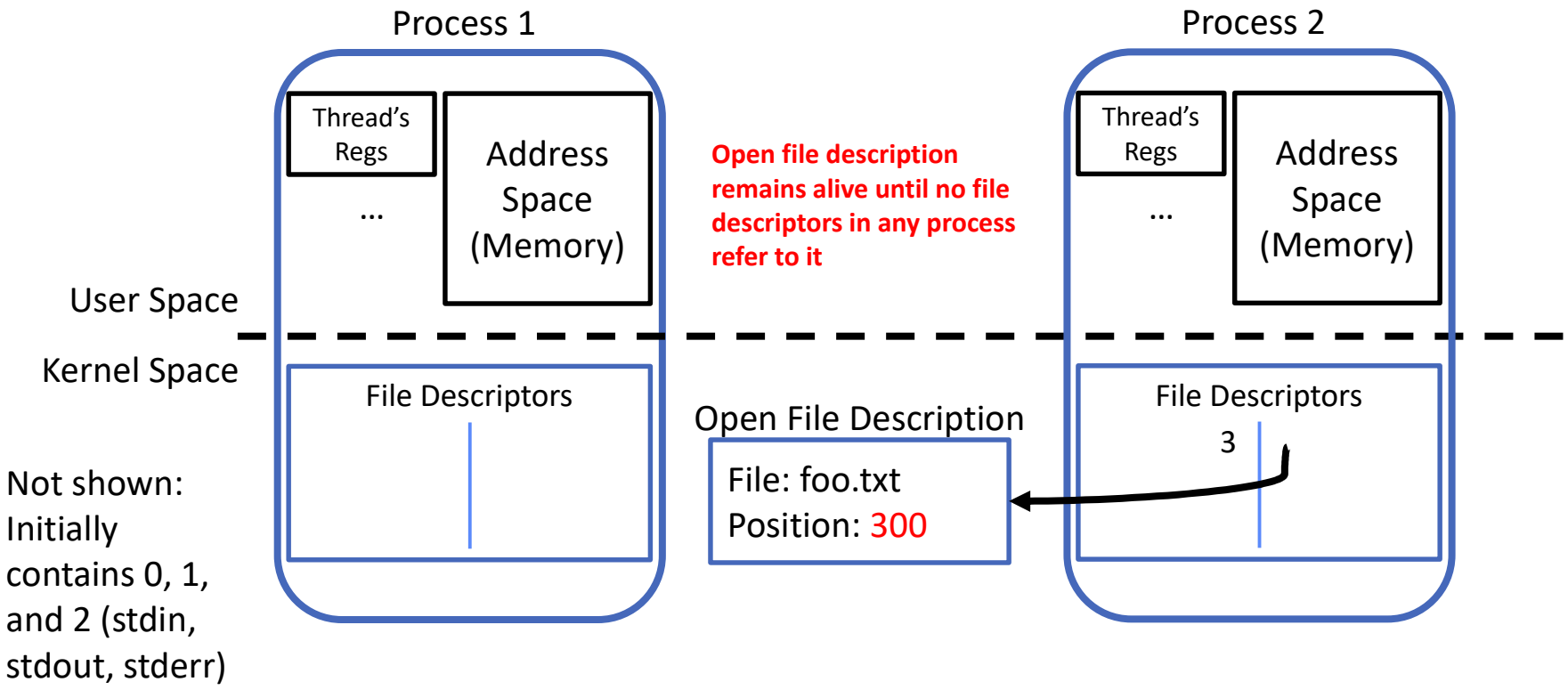
Open File Description is *Aliased*



File Descriptor is *Copied*



File Descriptor is *Copied*



Why is Aliasing the Open File Description a Good Idea?

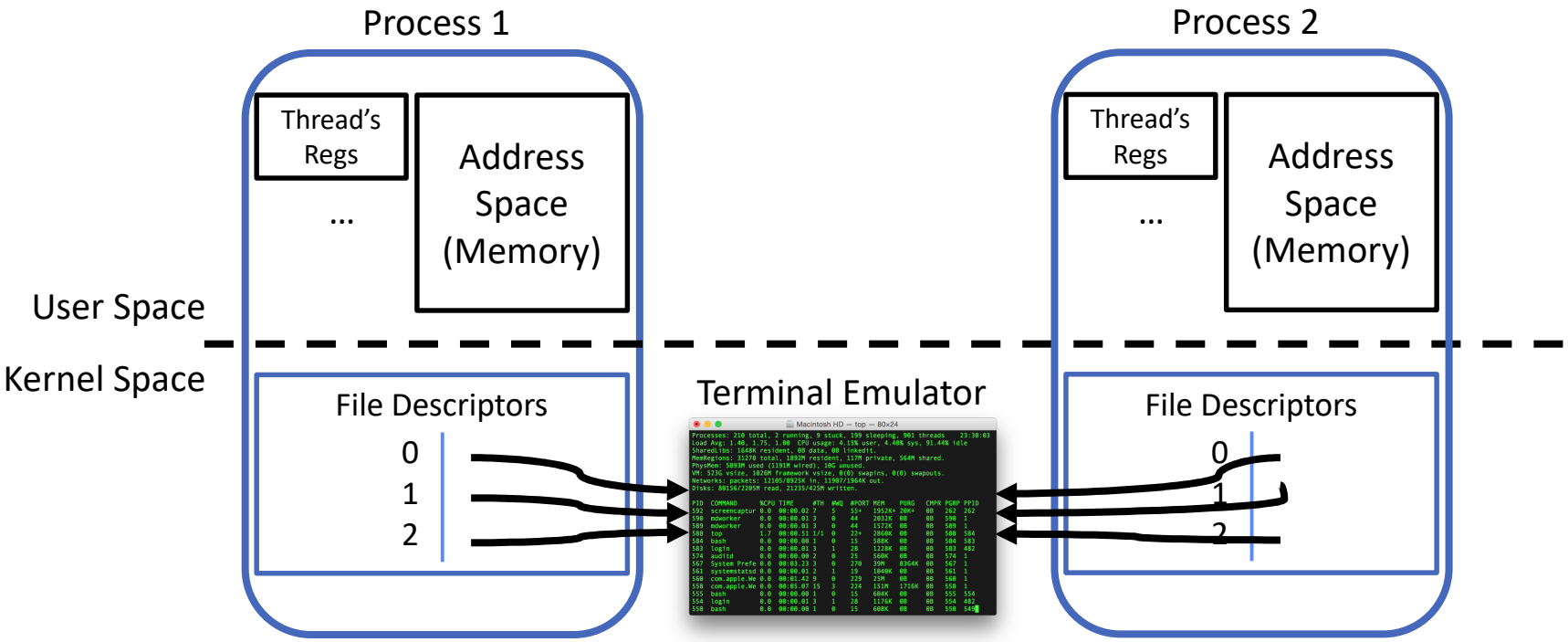
- It allows for *shared resources* between processes

In Linux, Everything is a File

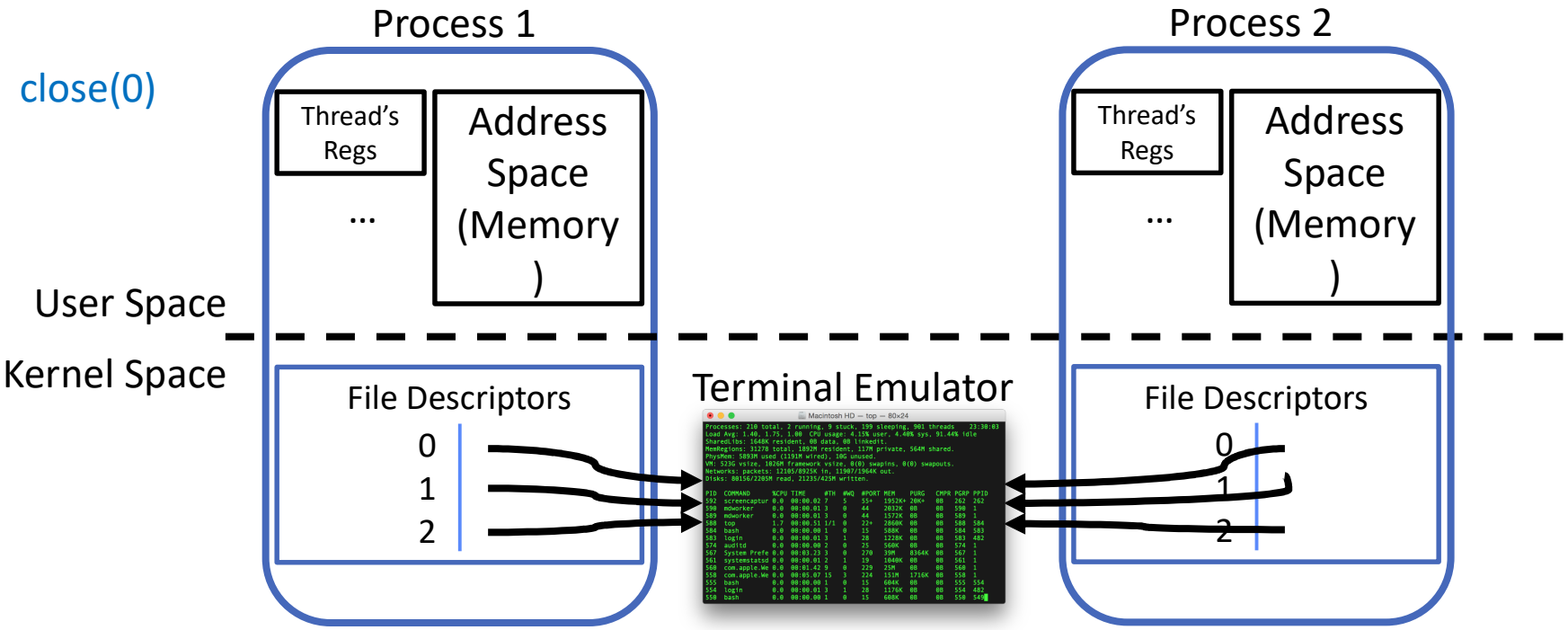
- Identical interface for:
  - Files on disk
  - Devices (terminals, printers, etc.)
  - Regular files on disk
  - Networking (sockets)
  - Local inter-process communication (pipes, sockets)

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

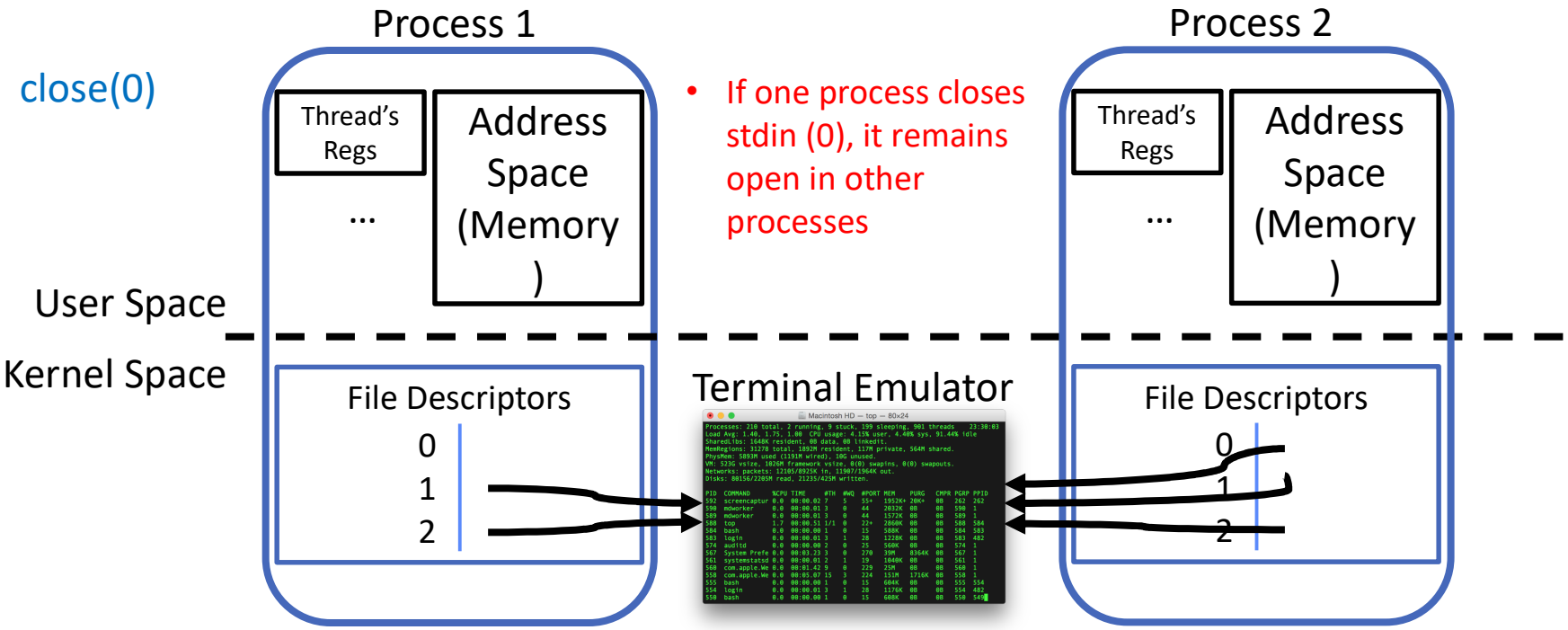
Example: Shared Terminal Emulator



Example: Shared Terminal Emulator



Example: Shared Terminal Emulator

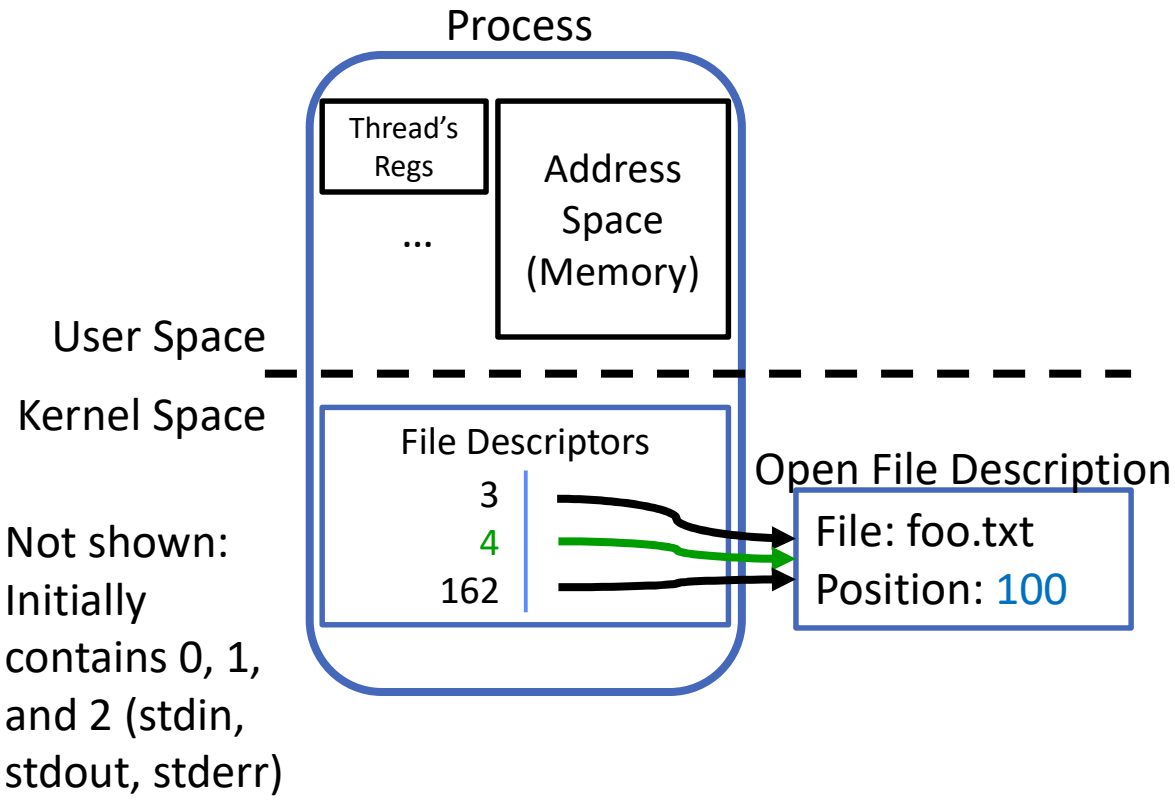


- If one process closes stdin (0), it remains open in other processes

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 position is 100

Next, suppose that we execute
dup(3)
And that the result is 4

Finally, suppose that we execute
dup2(3, 162)

Pitfalls of OS abstraction

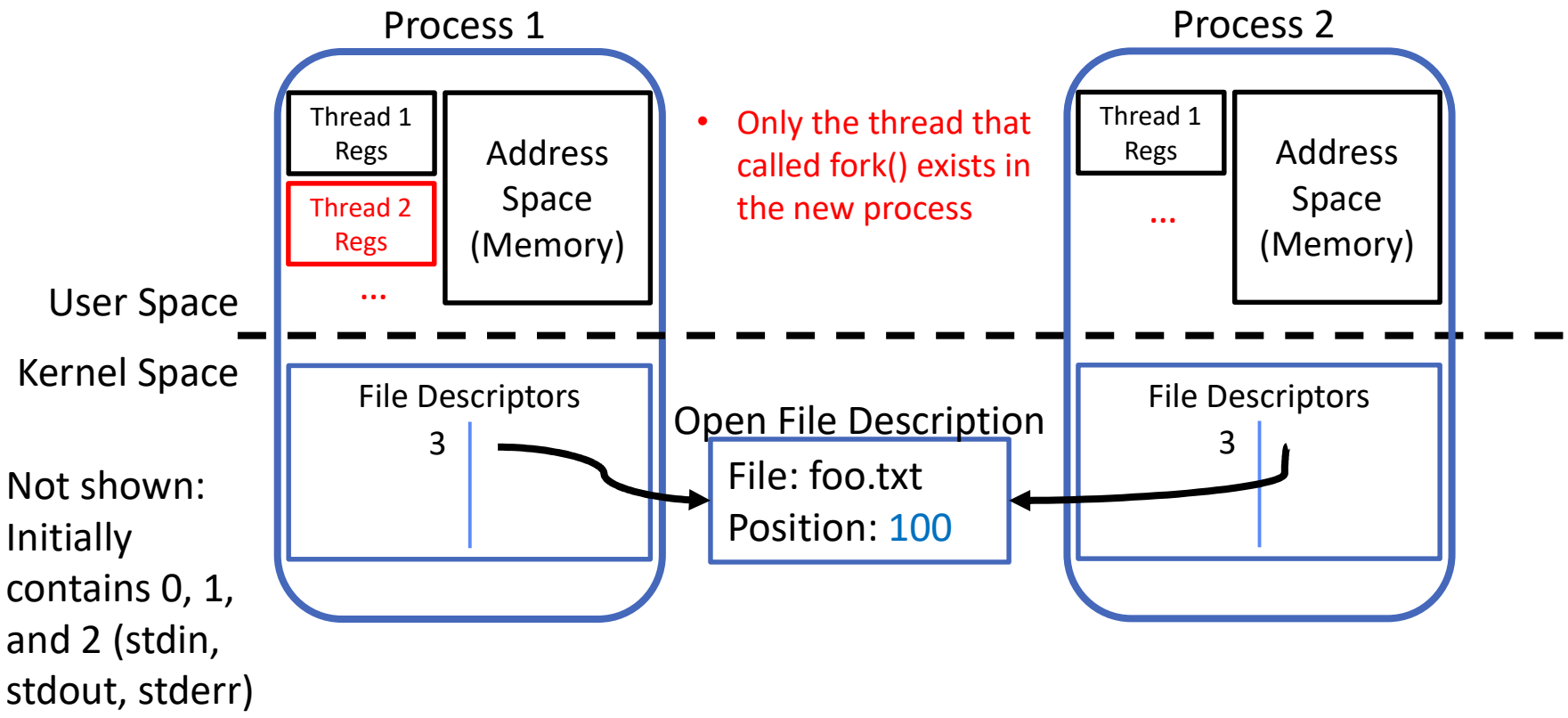
Don’t fork() in a process that already has multiple threads

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

`fork()` in Multithreaded Processes (in POSIX)

- The child process always has just a single thread
  - The thread in which `fork()` was called
- The other threads of the child process just vanish without any notice

fork() in a Multithreaded Processes



pfile2.c

```c
// Creating a sequential file
#include <stdio.h>

int main(void){
   FILE *cfPtr = NULL; // cfPtr = clients.txt file pointer

   // fopen opens the file. Exit the program if unable to create the file
   if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
      puts("File could not be opened");
   }
   else {
      puts("Enter the account, name, and balance.");
      puts("Enter EOF to end input.");
      printf("%s", "? ");

      int account = 0; // account number
      char name[30] = ""; // account name
      double balance = 0.0; // account balance

      scanf("%d%29s%lf", &account, name, &balance);

      // write account, name and balance into file with fprintf
      while (!feof(stdin)) {
         fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
         printf("%s", "? ");
         scanf("%d%29s%lf", &account, name, &balance);
      }

      fclose(cfPtr); // fclose closes file
   }
}
```

```
(base) Ravis-MacBook-Pro-2:cp ravimittal$ ./pf
Enter the account, name, and balance.
Enter EOF to end input.
? 10 ravi 10.0
? 200 ram 50.50
? 300 sam 20.0
```

$ cat clients.txt

```
10 ravi 10.00
200 ram 50.50
300 sam 20.00
```

EOF character
Linux/MAC OS : <Ctrl> d
Windows: <Ctrl> z enter

Lecture Summary

- File descriptors can be manipulated within processes or across processes