香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Operating Systems
Lecture 5

System programming: files

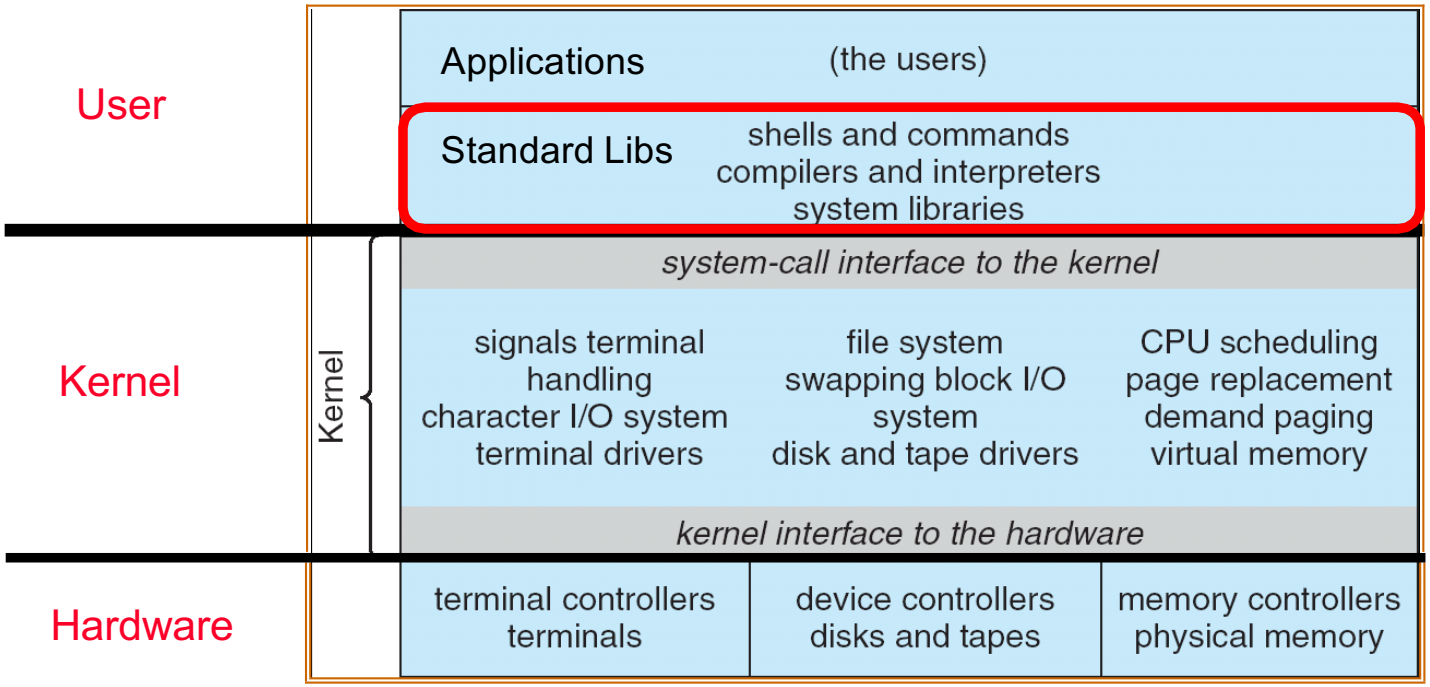
Prof. Yunming XIAO

School of Data Science

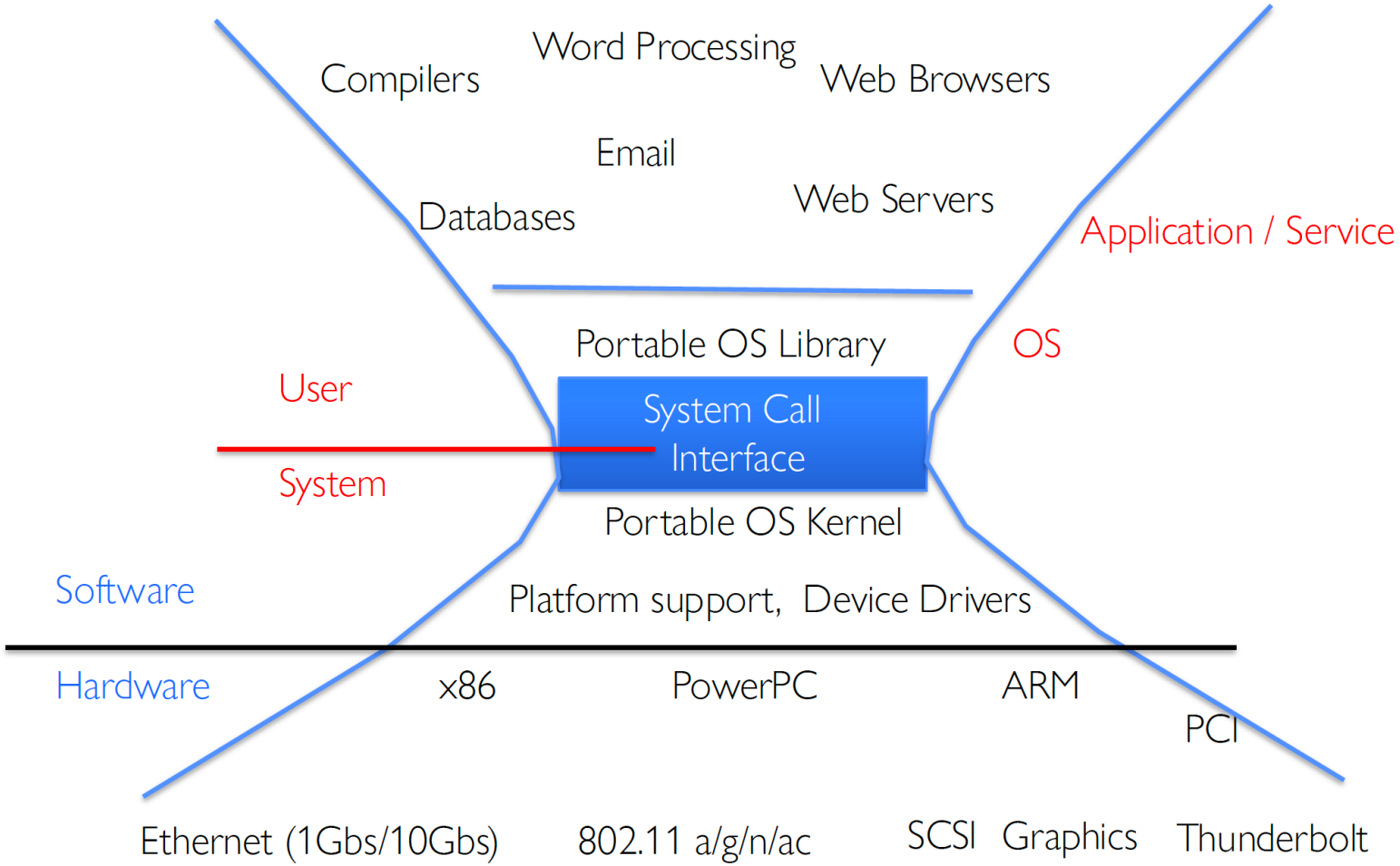
Acknowledgments: Ion Stoica and Matei Zaharia, Berkeley CS 162

Files

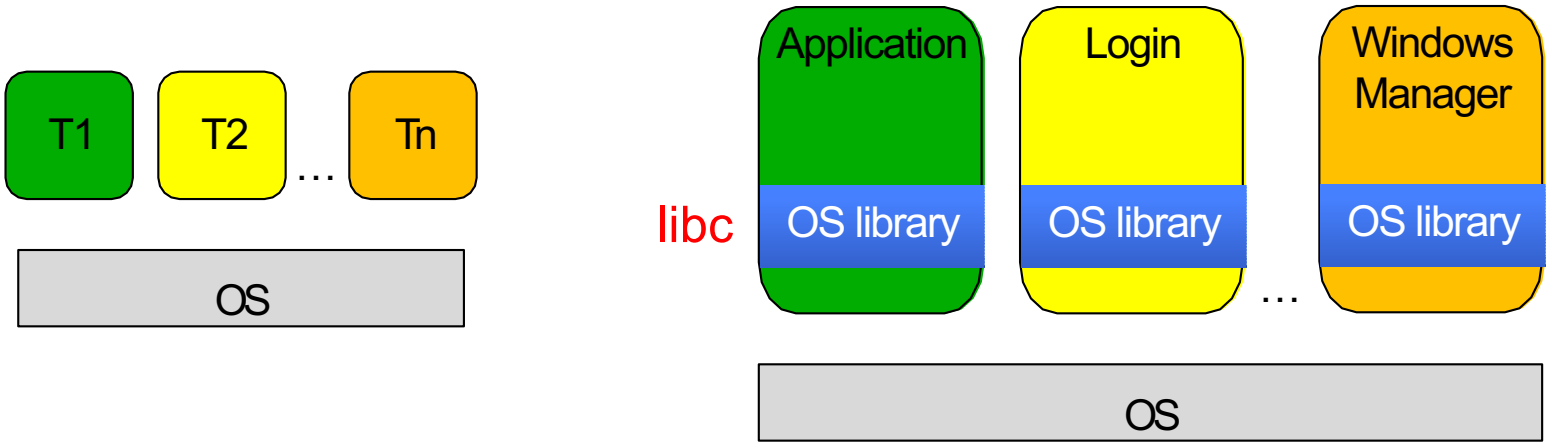
Recall: UNIX system structure



A kind of narrow waist



Recall: OS library (libc) issues syscalls



- OS Library: Code linked into the user-level application that provides a clean or more functional API to the user than just the raw syscalls
  - Most of this code runs at user level, but makes syscalls (which run at kernel level)

UNIX/POSIX idea: everything is a “file”

- Almost 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()`, & `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
  - Available online

Aside: POSIX interfaces

- 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

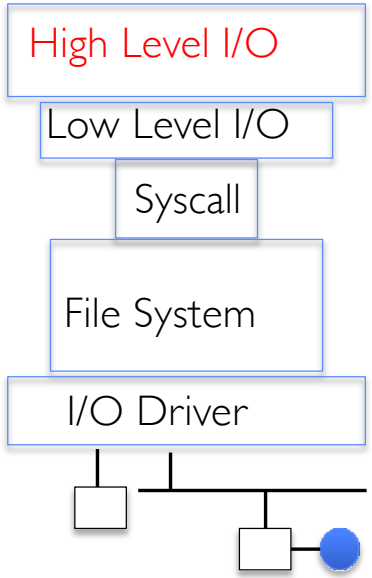
The file system abstraction

- 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, Security info, Access control
- Directory
  - “Folder” containing files & directories
  - Hierarchical naming
    - Path through the directory graph
    - Uniquely identifies a file or directory
  - 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/oski/csc3150
- 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, ~csc3150/index.html
    - Refers to index.html in the home directory

I/O and storage layers



Streams (buffered I/O)

File Descriptors

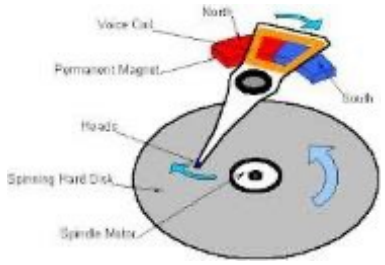
open(), read(), write(), close(), …

Open File Descriptions

Files/Directories/Indexes

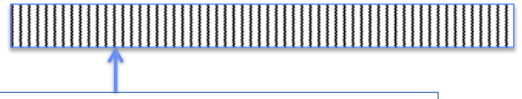
Commands and Data Transfers

Disks, Flash, Controllers, DMA



C high-level file API – streams

- Operates on "streams" – unformatted sequences of bytes (wither 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 stream – 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 );                 // return c or EOF on err
int fputs( const char *s, FILE *fp );         // return > 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(output, c);
    c = fgetc(input);
  }
  fclose(input);
  fclose(output);
}
```

C high-level file API

```
// character oriented
int fputc( int c, FILE *fp );                 // return c or EOF on err
int fputs( const char *s, FILE *fp );         // return > 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, sizeof(char), BUFFER_SIZE, input);
  while (length > 0) {
    fwrite(buffer, sizeof(char), length, output);
    length = fread(buffer, sizeof(char), BUFFER_SIZE, input);
  }
  fclose(input);
  fclose(output);
}
```

Aside: check your errors!

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

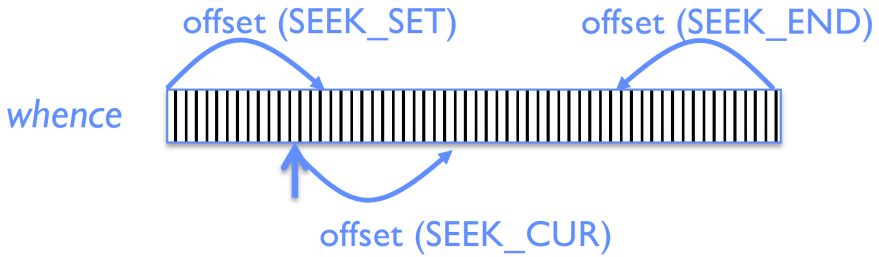
```
FILE* input = fopen("input.txt", "r");
if (input == NULL) {
  // Prints our string and error msg
  perror("Failed to open input file");
}
```

- 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

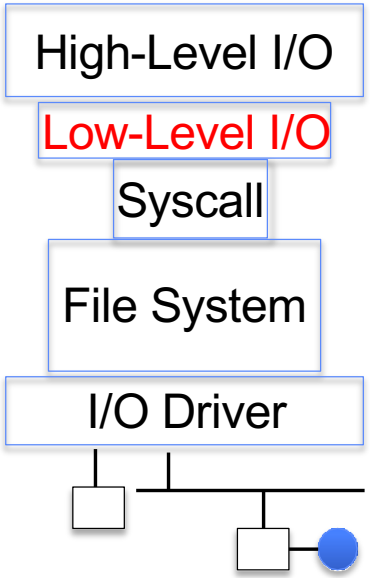
```
int fseek(FILE *stream, long int offset, int whence)
long int ftell(FILE *stream)
void rewind(FILE *stream)
```

- 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

I/O and storage layers



Streams (buffered I/O)

File Descriptors

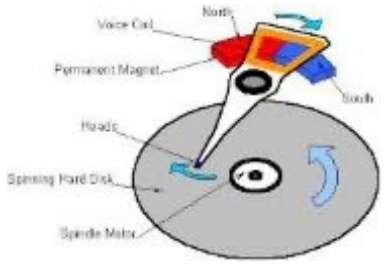
open(), read(), write(), close(), …

Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

Disks, Flash, Controllers, DMA



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 create 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);
  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

Low-level vs. high-level file API

- Low-level direct use of syscall interface:
  open(), read(), write(), close()
- Opening of file returns file descriptor:
  int myfile = open(…);
- File descriptor only meaningful to kernel
  - Index into process (PDB) which holds pointers to kernel-level structure ("file description") describing file.
- Every read() or write() causes syscall no matter how small (could read a single byte)
- Consider loop to get 4 bytes at a time using read():
  - Each iteration enters kernel for 4 bytes.

- High-level buffered access:
  fopen(), fread(), fwrite(), fclose()
- Opening of file returns ptr to FILE:
  FILE *myfile = fopen(…);
- FILE structure is user space contains:
  - a chunk of memory for a buffer
  - the file descriptor for the file (fopen() will call open() automatically)
- Every fread() or fwrite() filters through buffer and may not call read() or write() on every call.
- Consider loop to get 4 bytes at a time using fread():
  - First call to fread() calls read() for block of bytes (say 1024). Puts in buffer and returns first 4 to user.
  - Subsequent fread() grab bytes from buffer

Low-level vs. high-level file API

- Low-level operations

```
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 reg
    return data to caller
}
```

- High-level operations

```
ssize_t fread(…) {
    check buffer for contents
    return data to caller if available
    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 reg
    update buffer with excess data
    return data to caller }
```

Low-level vs. high-level file API

- Low-level operations on file descriptors are visible immediately

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

  - Outputs "Beginning of line" 10 seconds earlier than “and end of line”

- High-level streams are buffered in user memory:

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

  - Prints out everything at once

Group discussion

- Topic: buffer in user space
  - Why buffer in user space? Why not?
  - What are the pros and cons?

- 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

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 kernel

- 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);
```

What's below the surface?

File descriptor number
- an int

File description
- a struct with all the info about the files

Application / Service

High-Level I/O — Streams

Low-Level I/O — Handles

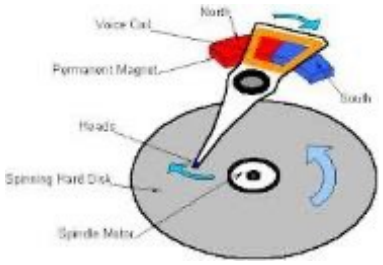
Syscall — open(), read(), write(), close(), …

File System — Descriptions

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?

- Inside kernel!
- For our purpose, the two most important things are:
  - Where to find the file data on disk
  - The current position within the file

Linux:
https://github.com/torvalds/linux/blob/master/include/linux/fs.h

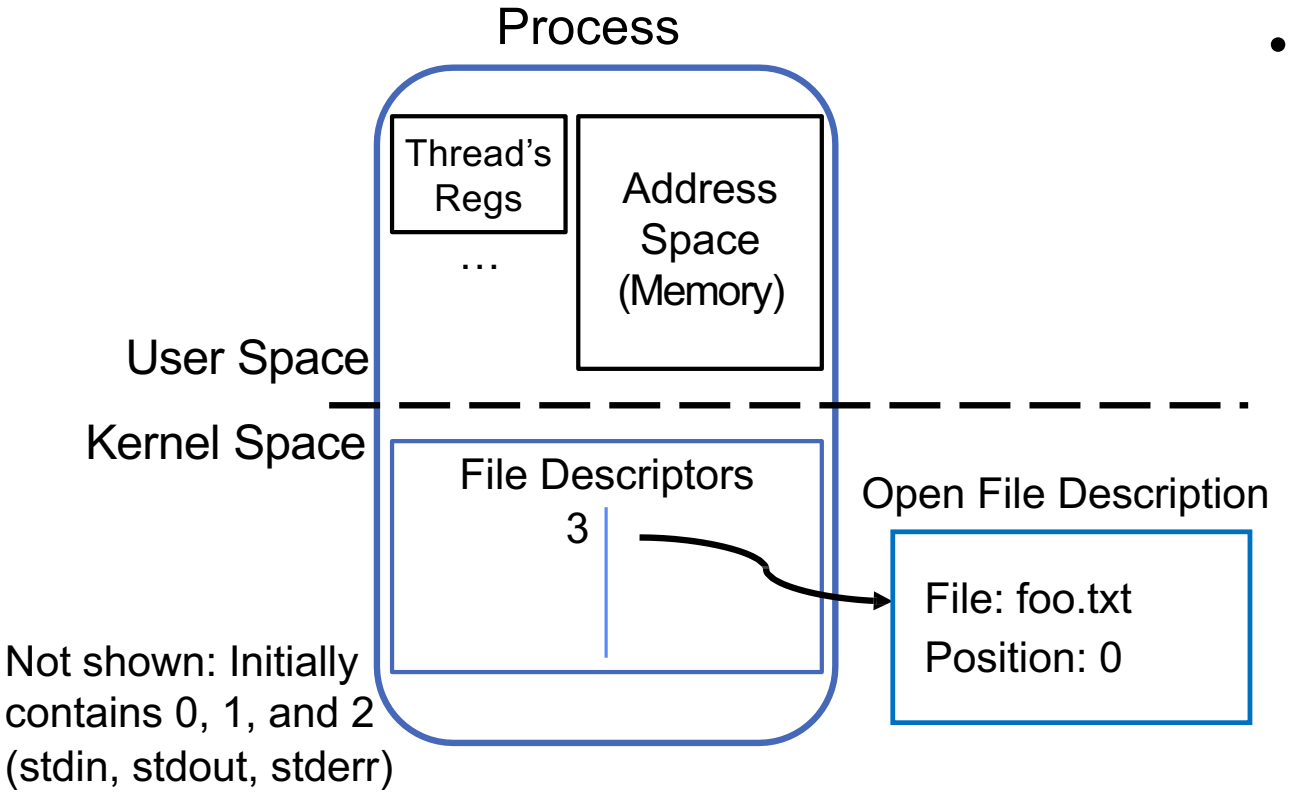
xv6:
https://github.com/mit-pdos/xv6-riscv/blob/riscv/kernel/file.h

linux / include / linux / fs.h

Code | Blame | 3609 lines (3186 loc) · 117 KB

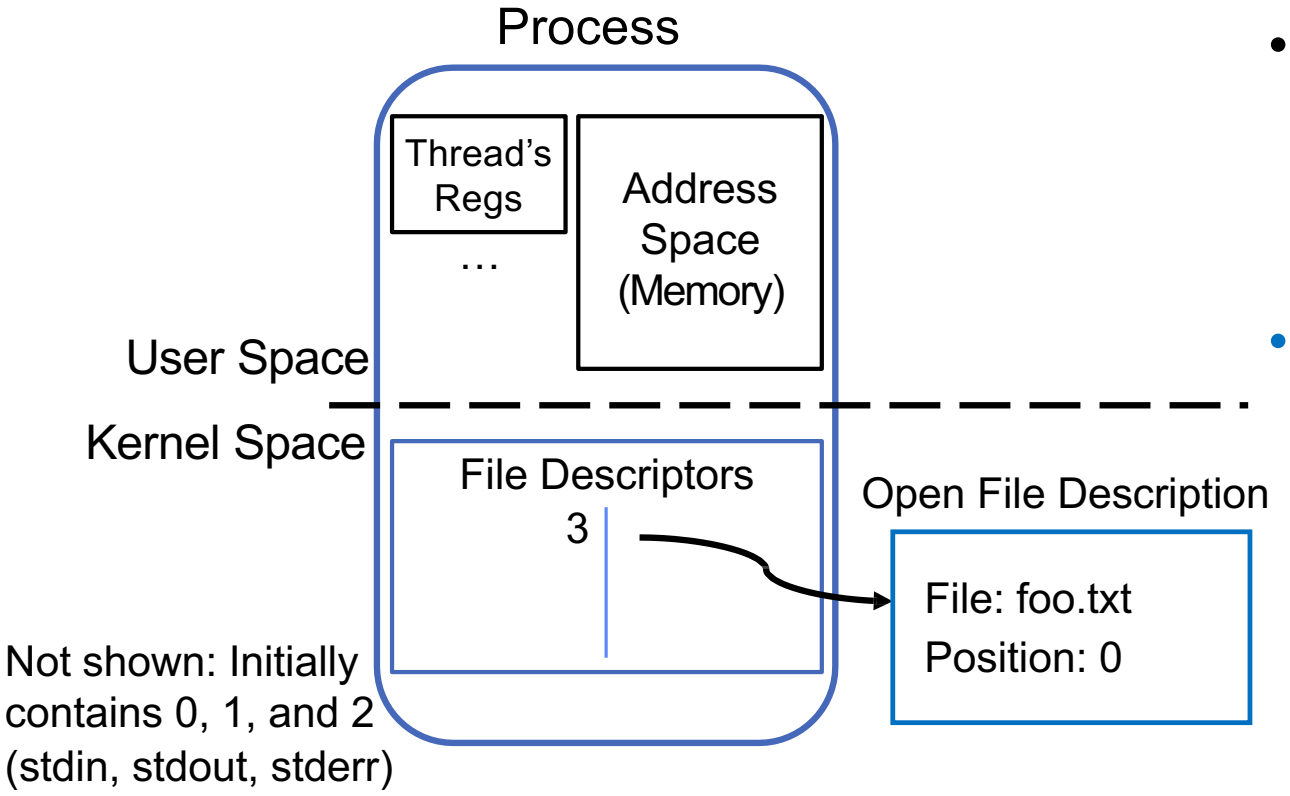
```
1257     */
1258    struct file {
1259            spinlock_t                      f_lock;
1260            fmode_t                         f_mode;
1261            const struct file_operations    *f_op;
1262            struct address_space            *f_mapping;
1263            void                            *private_data;
1264            struct inode                    *f_inode;
1265            unsigned int                    f_flags;
1266            unsigned int                    f_iocb_flags;
1267            const struct cred               *f_cred;
1268            struct fown_struct              *f_owner;
1269            /* --- cacheline 1 boundary (64 bytes) --- */
1270            union {
1271                    const struct path       f_path;
1272                    struct path             __f_path;
1273            };
1274            union {
1275                    /* regular files (with FMODE_ATOMIC_POS) and directories */
1276                    struct mutex            f_pos_lock;
1277                    /* pipes */
1278                    u64                     f_pipe;
1279            };
1280            loff_t                          f_pos;
1281    #ifdef CONFIG_SECURITY
1282            void                            *f_security;
1283    #endif
1284            /* --- cacheline 2 boundary (128 bytes) --- */
1285            errseq_t                        f_wb_err;
1286            errseq_t                        f_sb_err;
```

Process-specific file descriptor table inside kernel



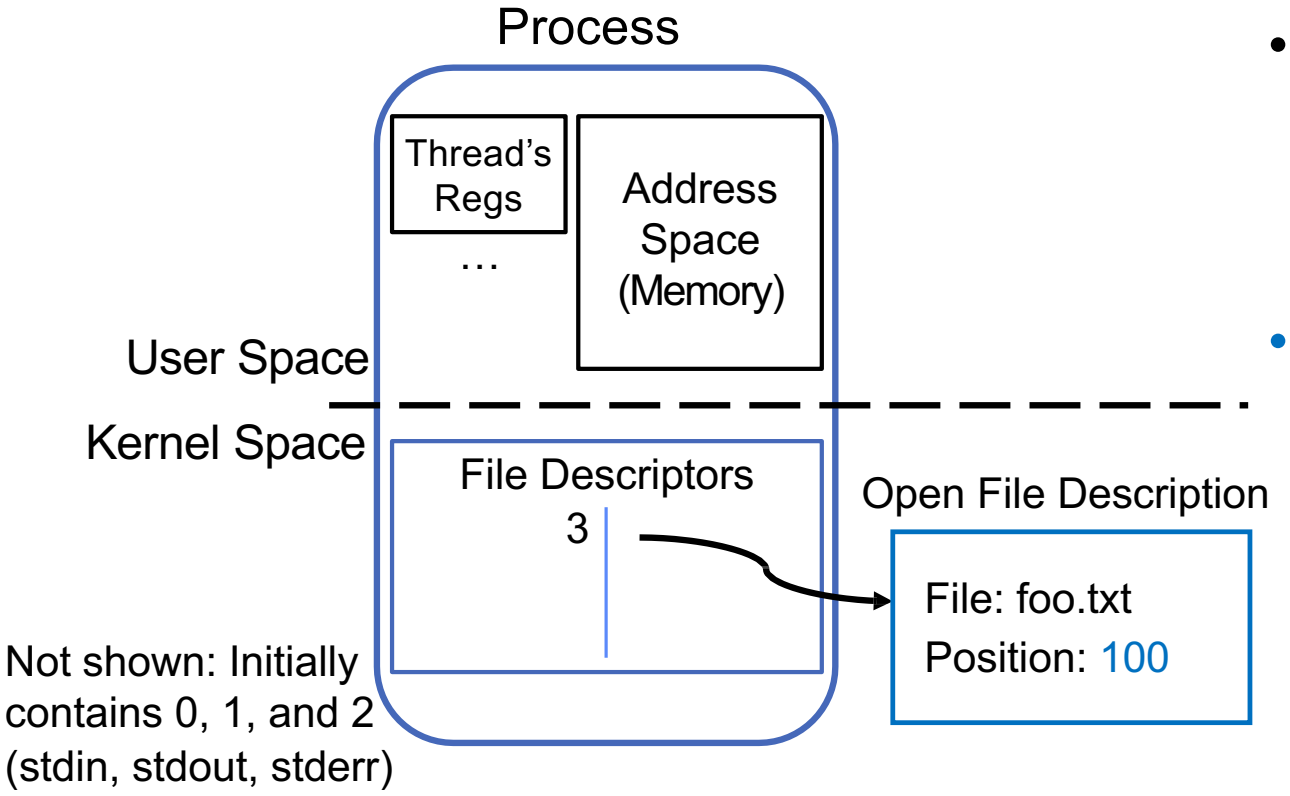
- Suppose that we execute `open("foo.txt")` and that the result is 3

Process-specific file descriptor table inside kernel



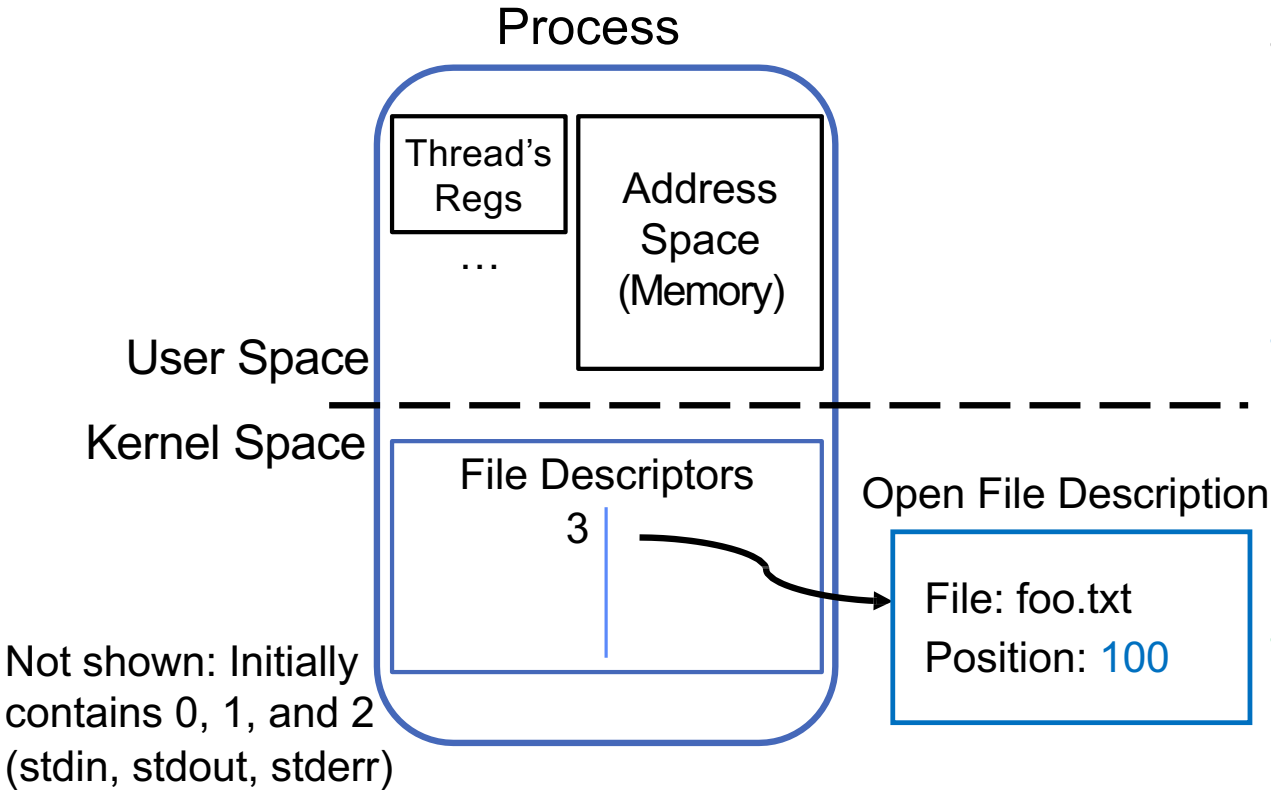
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100

Process-specific file descriptor table inside kernel



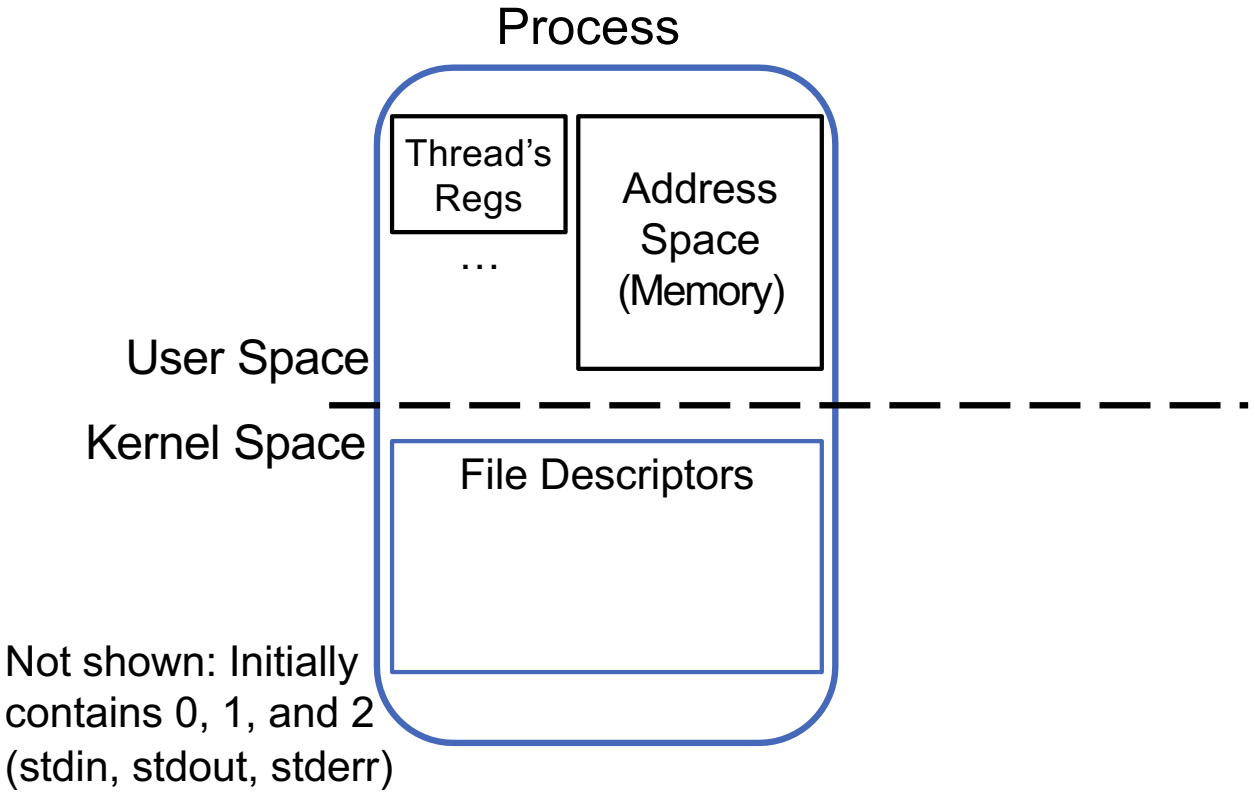
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100

Process-specific file descriptor table inside kernel



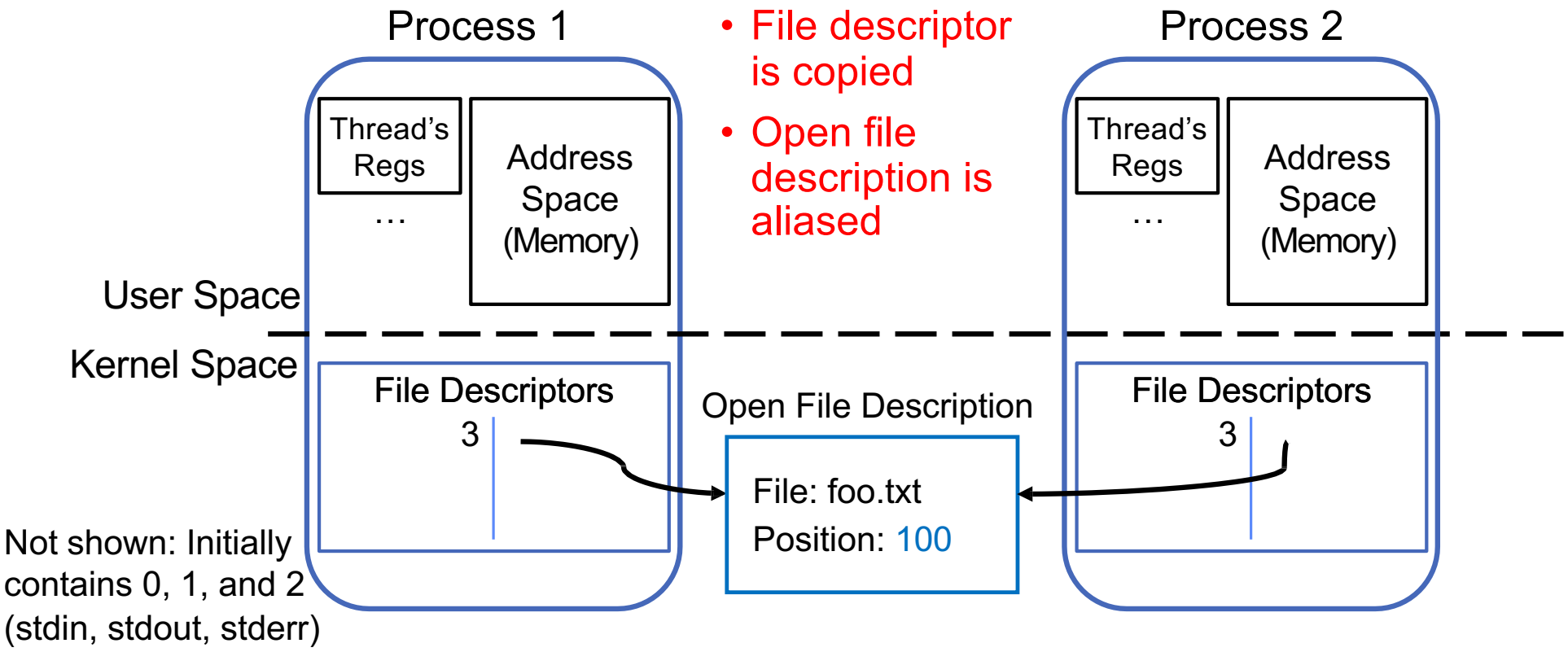
- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100
- Finally, suppose that we execute `close(3)`

Process-specific file descriptor table inside kernel

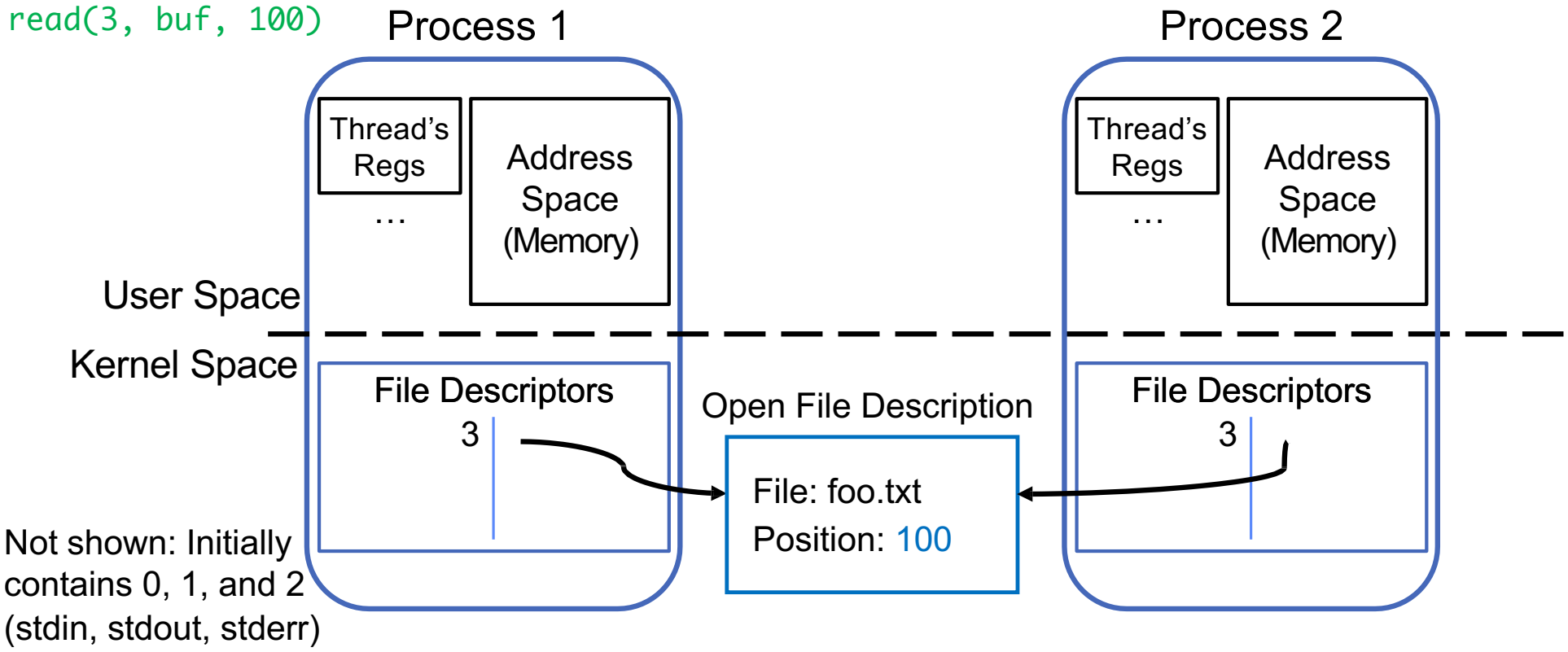


- Suppose that we execute `open("foo.txt")` and that the result is 3
- Next, suppose that we execute `read(3, buf, 100)` and the result is 100
- Finally, suppose that we execute `close(3)`

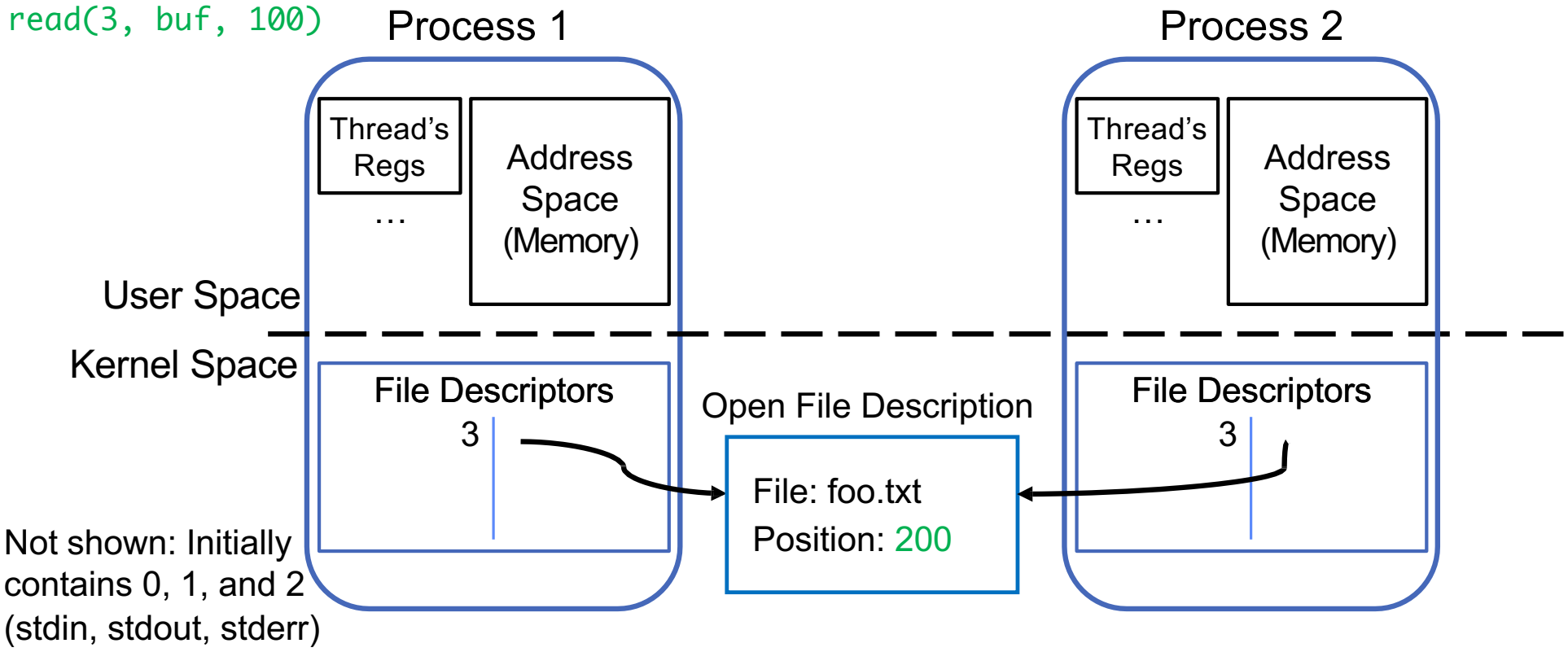
Instead of closing, let's fork()!



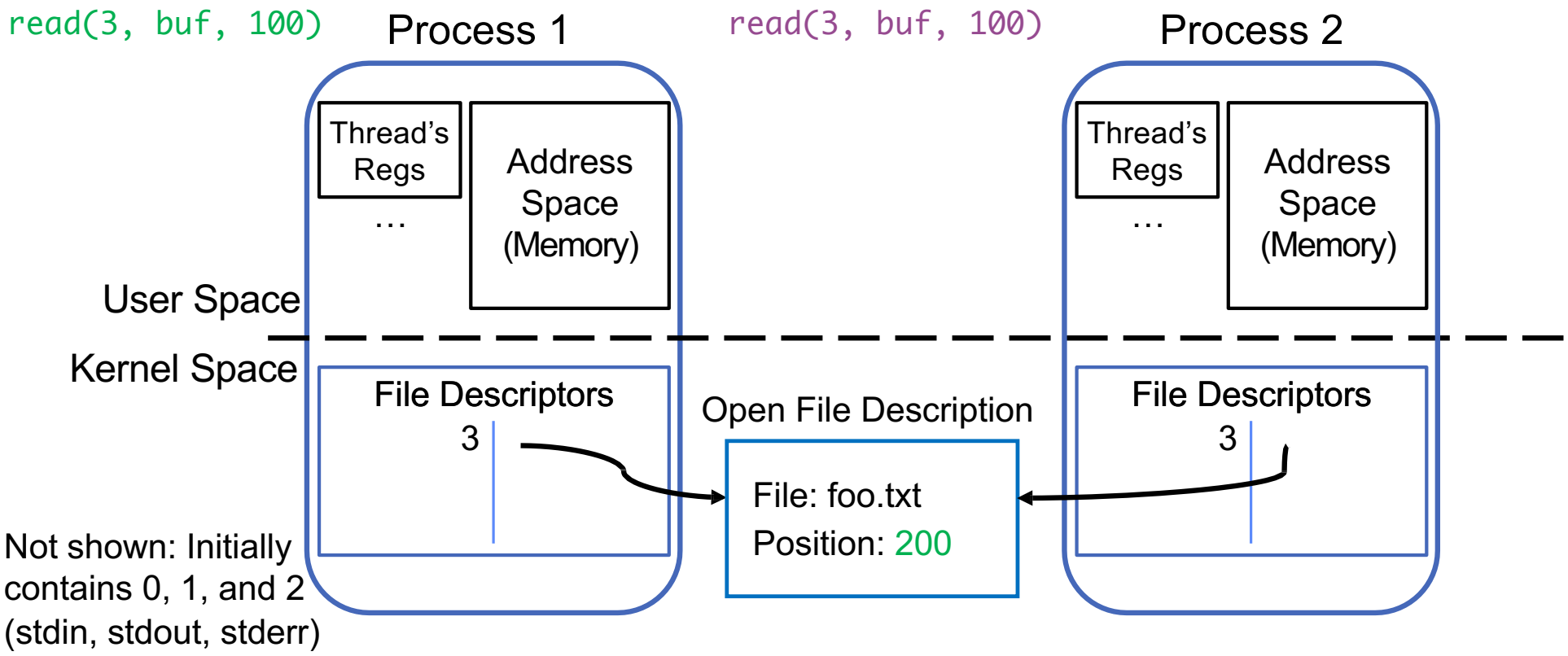
Open file description is *aliased*



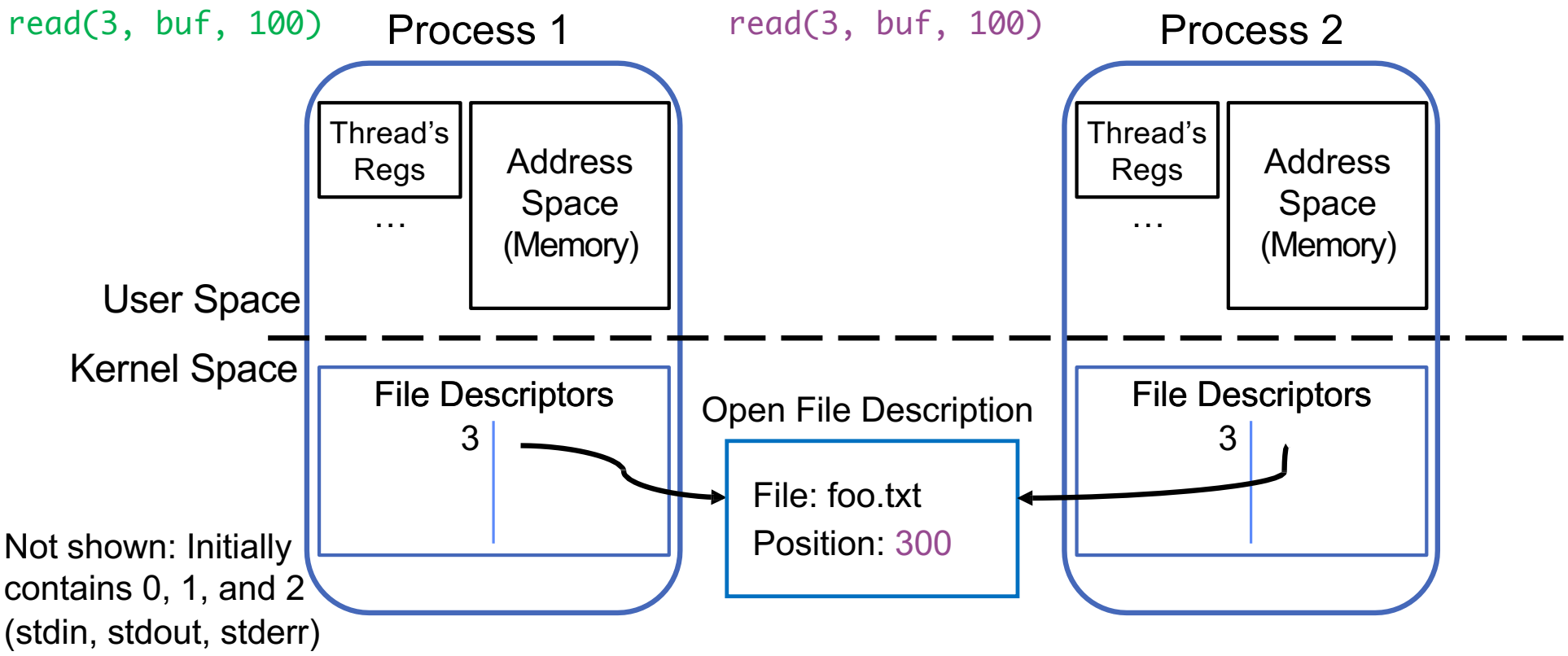
Open file description is *aliased*



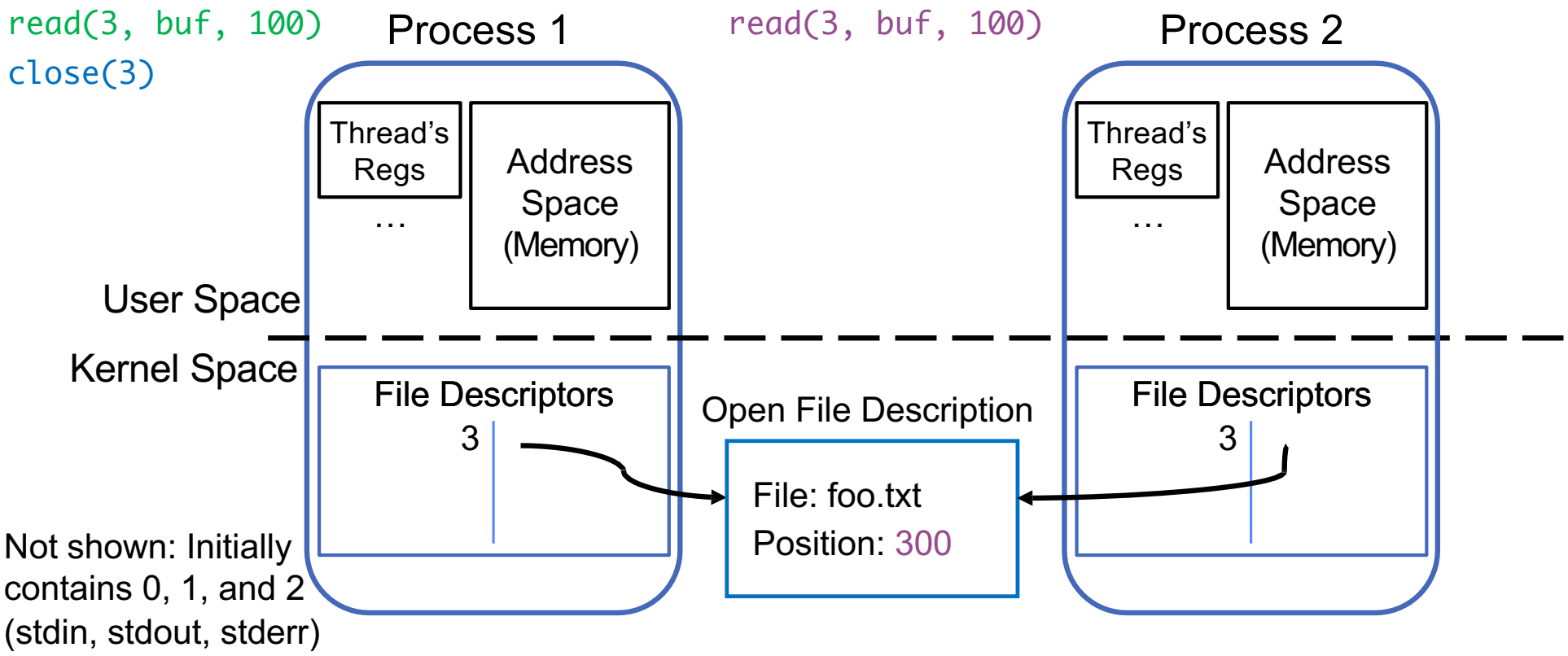
Open file description is *aliased*



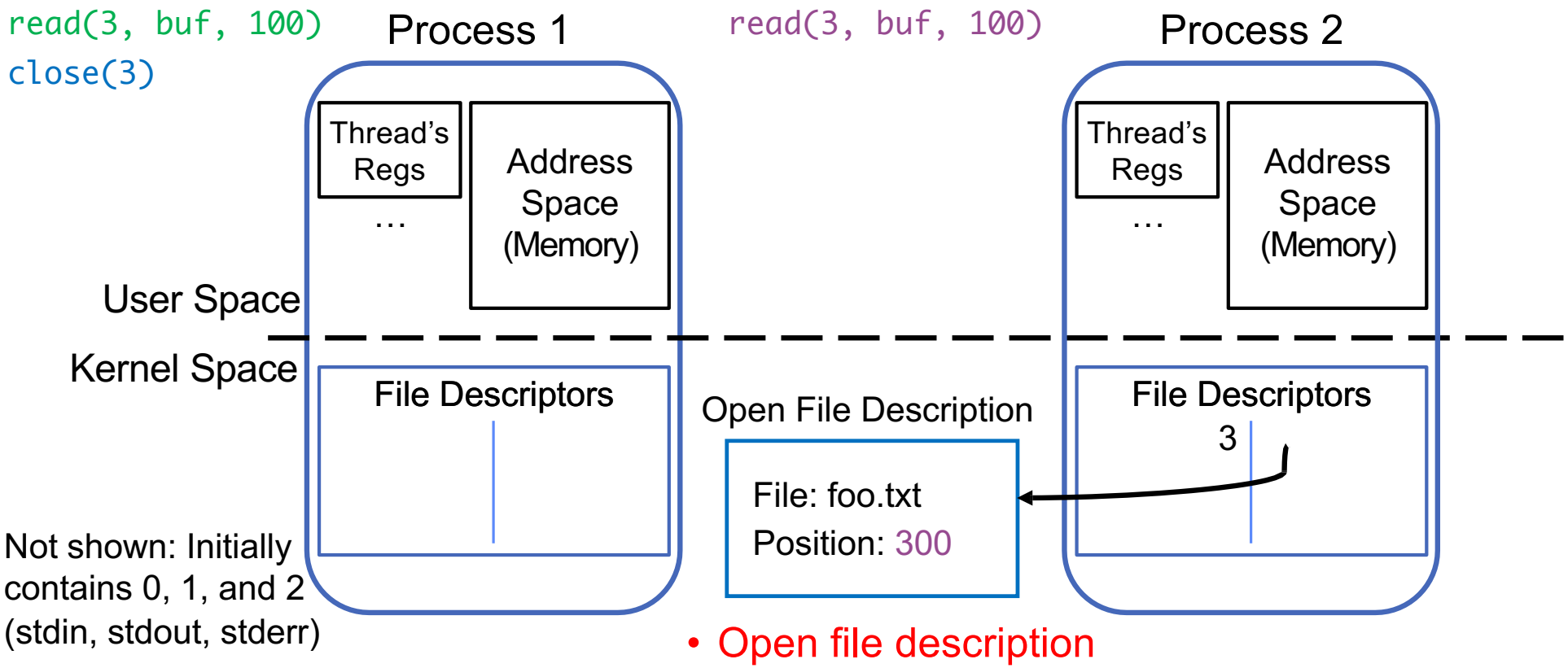
Open file description is *aliased*



File description is *copied*



File description is *copied*



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

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 inter-process communication (pipes, sockets)
- High-level I/O abstracts
  - Operates on “streams” – unformatted sequences of bytes, with a position
  - Provides user-space buffering to enable efficiency and usability
- Low-level I/O abstracts
  - File descriptor as an integer, linking to an Open File Descriptor inside kernel
  - Directly iterating with kernel

Thanks