

Abstractions 2: Files

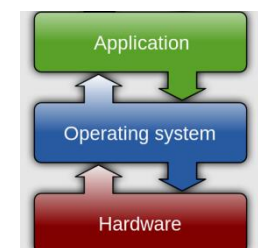
Lecture 4

Hartmut Kaiser

<https://teaching.hkaiser.org/spring2026/csc4103/>

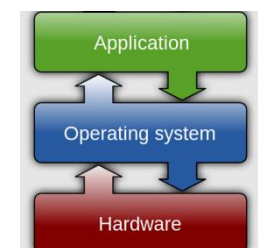
Recall: Threads

- Independently schedulable execution sequence that runs concurrently with other threads
 - It can block waiting for something while others progress
 - It can work in parallel with others
- Has local state (its stack, registers) and shares static data and heap with other threads in the same process
- In the absence of synchronization operations, arbitrary interleaving of threads may occur



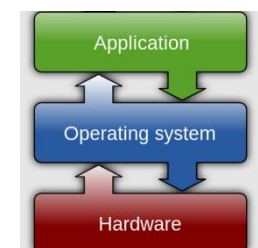
Recall: Synchronization

- **Mutual Exclusion**: Ensuring only one thread does a particular thing at a time (one thread excludes the others)
- **Critical Section**: Code exactly one thread can execute at once
 - Result of mutual exclusion
- **Lock**: An object only one thread can hold at a time
 - Provides mutual exclusion
 - Offers two atomic operations:
 - `Lock.Acquire()` – wait until lock is free; then grab
 - `Lock.Release()` – Unlock, wake up waiters
- Need other tools for “cooperation”
 - e.g., semaphores



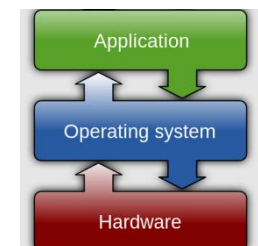
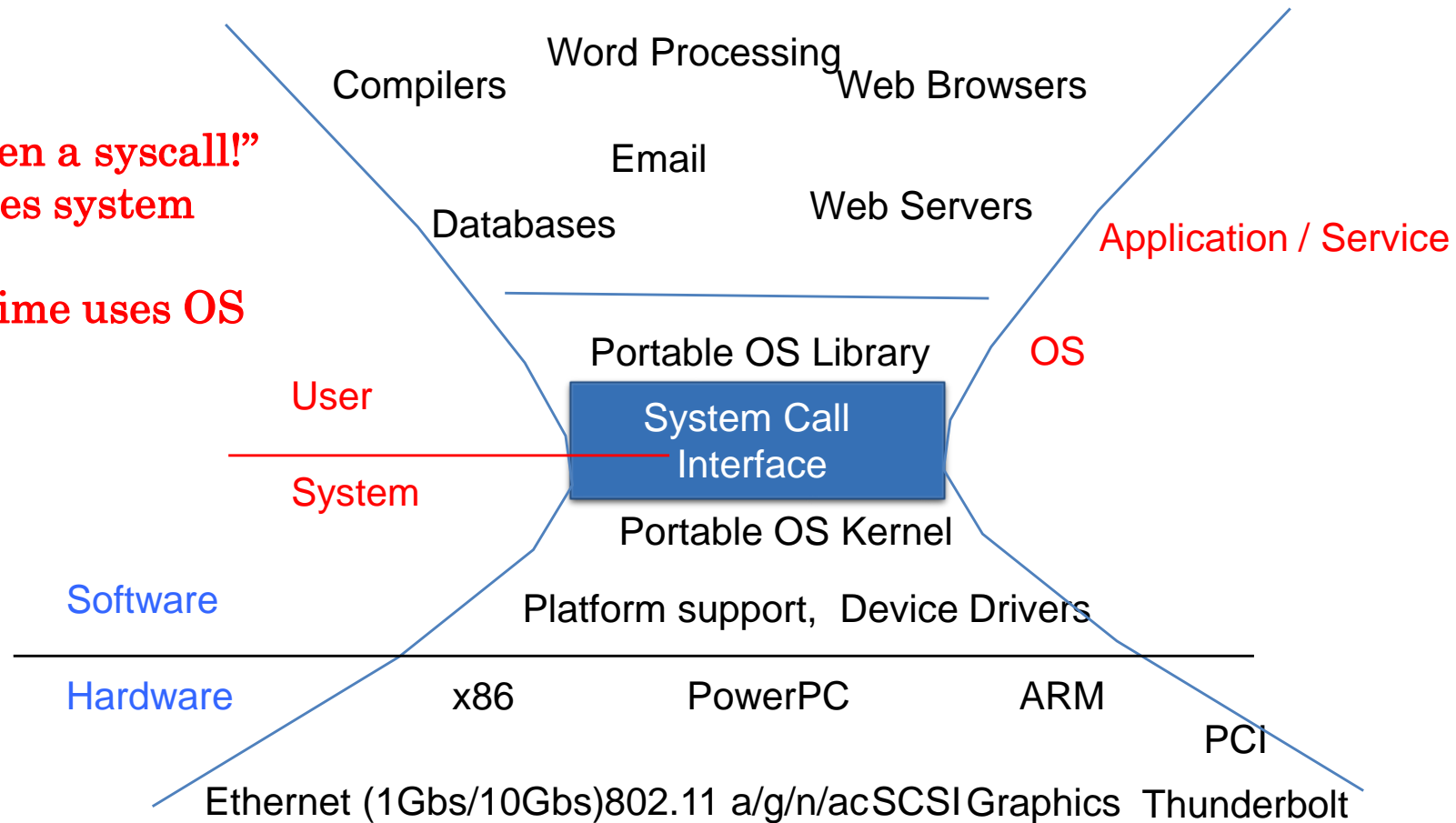
Recall: Processes

- Definition: execution environment with restricted rights
 - One or more threads executing in a single address space
 - Owns file descriptors, network connections
- Instance of a running program
 - When you run an executable, it runs in its own process
 - Application: one or more processes working together
- Protected from each other; OS protected from them
- In modern OSes, anything that runs outside of the kernel runs in a process

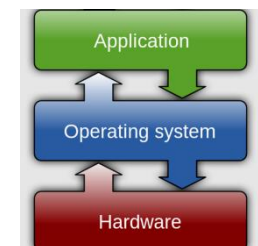
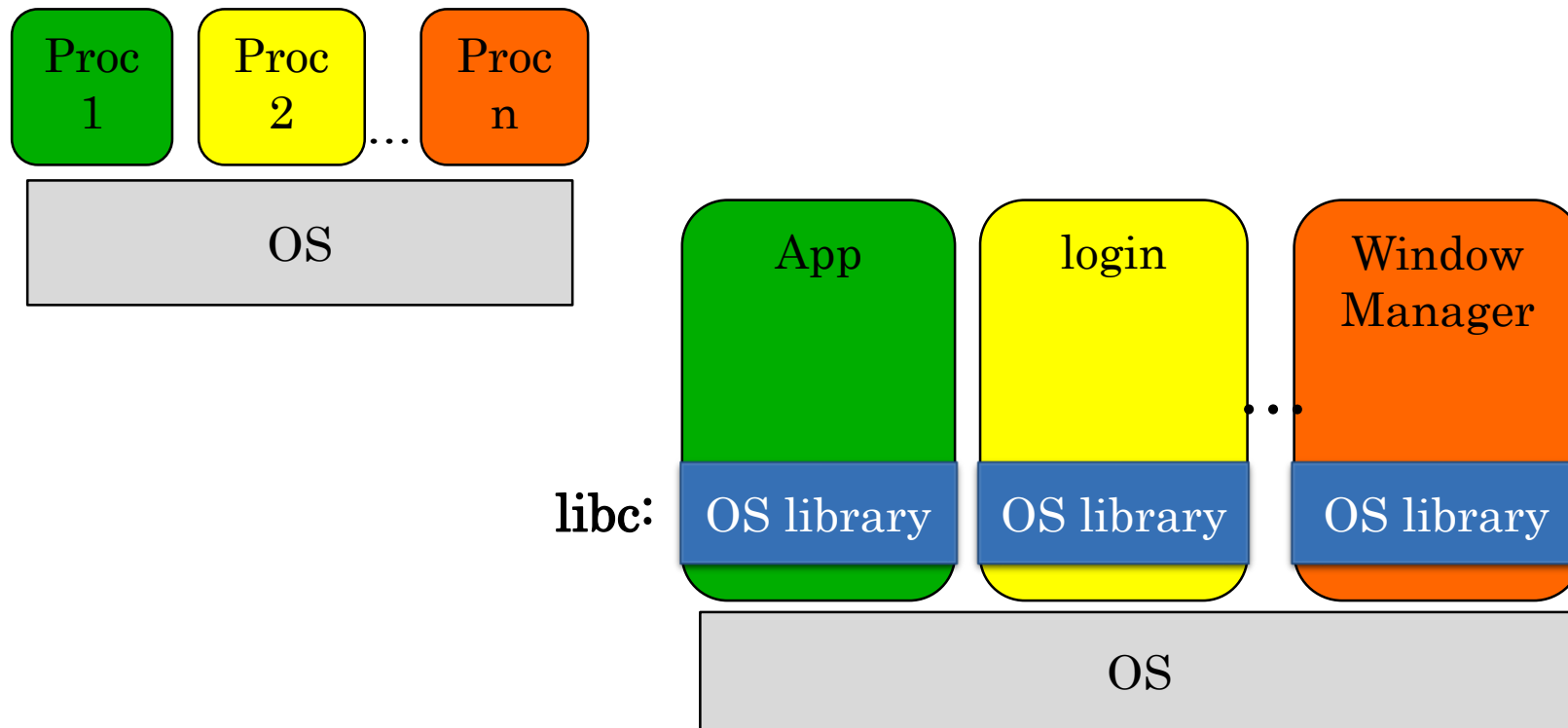


Recall: System Calls (“Syscalls”)

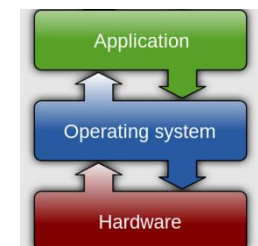
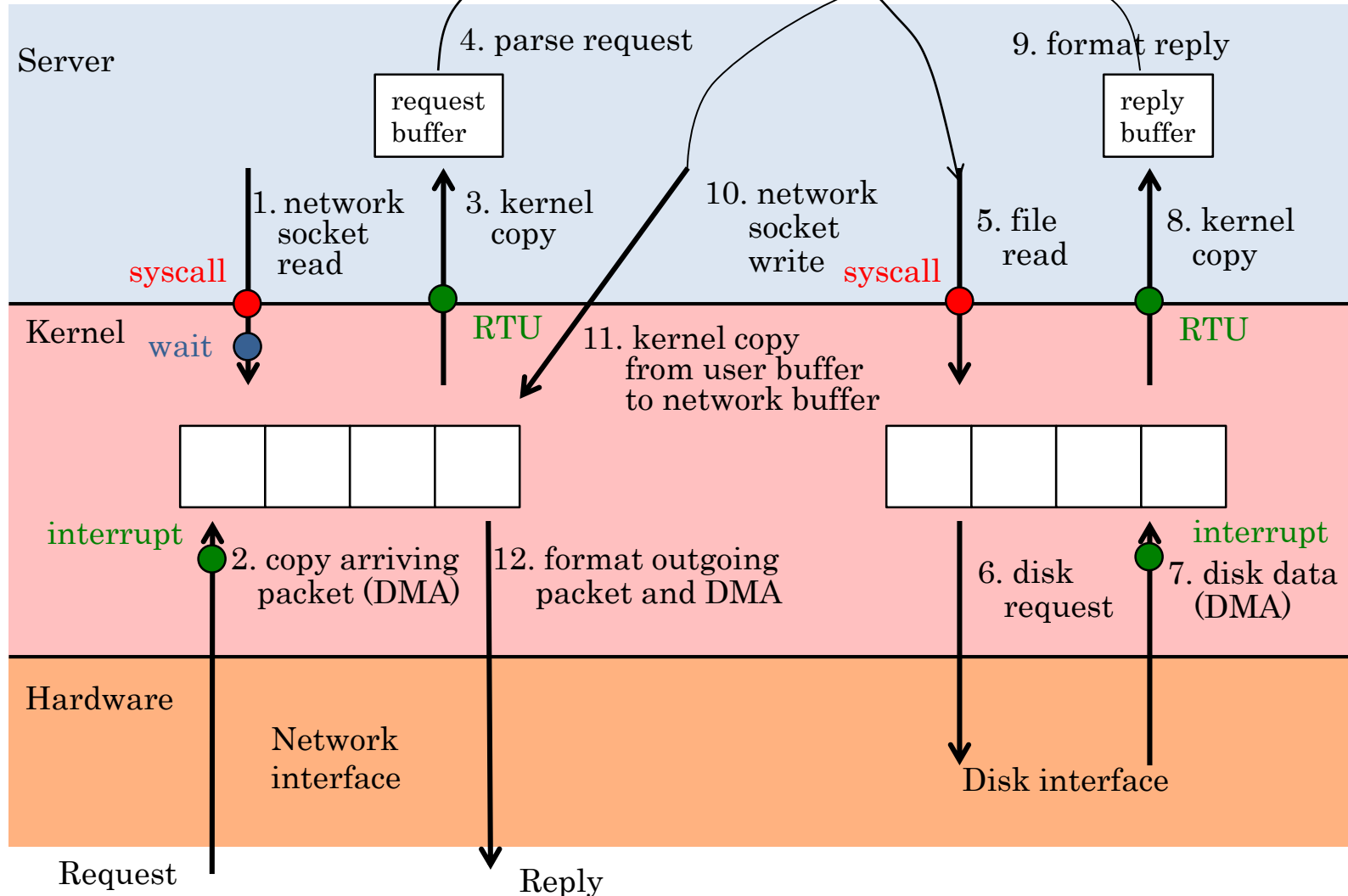
- “But, I’ve never seen a syscall!”
- OS library issues system call
 - Language runtime uses OS library...



Recall: OS Library Issues Syscalls

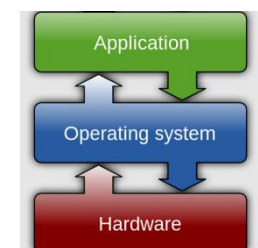


Putting it all Together: Web Server



What does pthread stand for?

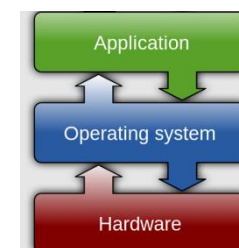
- pthread library: POSIX thread library
- POSIX: Portable Operating System Interface (X?)
 - Interface for application programmers (mostly)
 - Defines the term “Unix,” derived from AT&T Unix
 - Created to bring order to many Unix-derived OSes, so applications are portable
 - Requires standard system call interface



Files

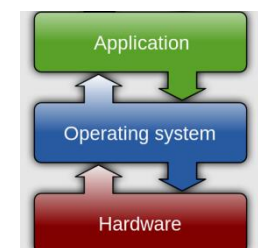
Unix/POSIX Idea: Everything is a “File”

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



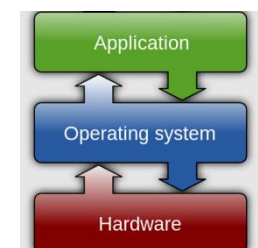
The File System Abstraction

- 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 (in addition to its name)
 - Size, Modification Time, Owner, Security info, Access control
- Directory
 - “Folder” containing files & directories
 - Hierarchical (graphical) naming
 - Path through the directory graph
 - Uniquely identifies a file or directory
 - `/home/ff/csc4103/public_html/fa14/index.html`
- Links and Volumes (later)



Connecting Processes, File Systems, and Users

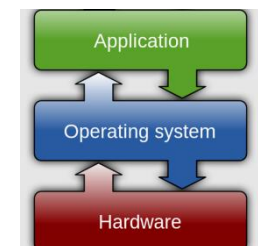
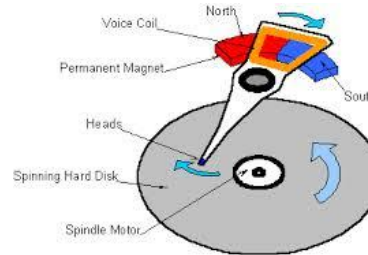
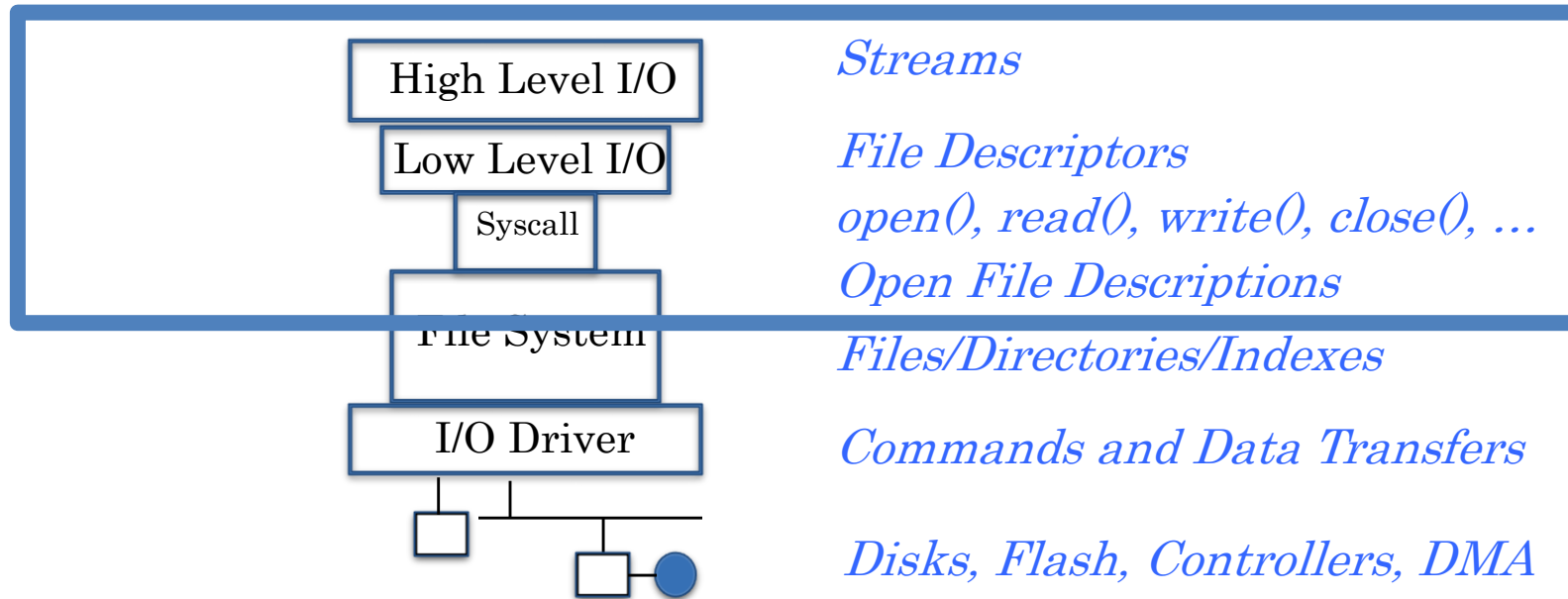
- Every process has a current working directory
 - Stored in the process control block (PCB)
- Absolute paths
 - /home/csc4103
- Relative paths
 - index.html, ./index.html
 - Refers to index.html in current working directory
 - ../index.html
 - Refers to index.html in parent of current working directory
 - ~/index.html, ~csc4103/index.html
 - Refers to index.html in the home directory



I/O and Storage Layers

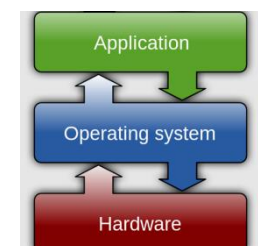
Application / Service

Focus of today's lecture



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



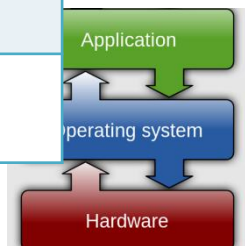
C High-Level File API – Streams

- Operates on “streams” – sequence of bytes, either text or data, with a position



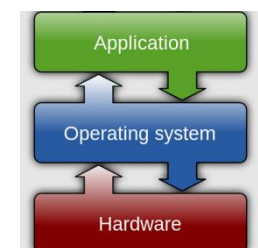
```
#include <stdio.h>
FILE* fopen(char const* filename, char const* mode);
int fclose(FILE* fp);
```

Mode Text	Binary	Descriptions
"r"	"rb"	Open existing file for reading; fails if file doesn't exist
"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; fails if file doesn't exist
"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



C API Standard Streams – `stdio.h`

- Three predefined streams are opened implicitly when the program is executed (by C standard library)
 - `FILE*` `stdin` – normal source of input (keyboard), can be redirected
 - `FILE*` `stdout` – normal source of output (terminal), can be redirected 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`



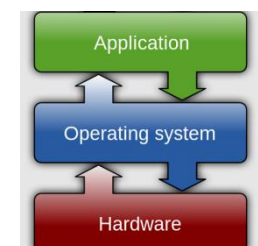
C High-Level File API

```
// character oriented
int fputc(int c, FILE* fp);           // rtn c or EOF on err
int fputs(char const* 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* stream, char const* format, ...);
int fscanf(FILE* stream, char const* format, ... );
```

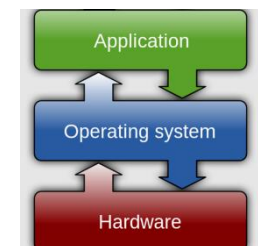


C Streams: Char-by-Char I/O

```
int main(void)
{
    FILE* input = fopen("input.txt", "r");    // returns NULL on error
    FILE* output = fopen("output.txt", "w"); // don't forget to check - always!
    int c;

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

    fclose(input);
    fclose(output);
}
```



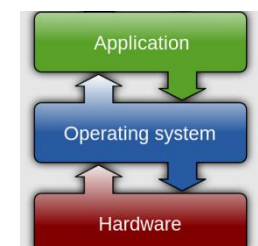
C High-Level File API

```
// character oriented
int fputc(int c, FILE* fp);           // returns c or EOF on err
int fputs(const char* s, FILE* fp);   // returns > 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* stream, const char* format, ...);
int fscanf(FILE* stream, const char* format, ... );
```



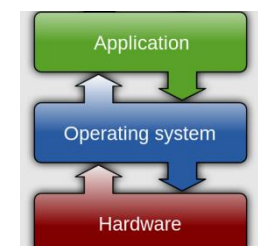
C Streams: Block-by-Block I/O

```
#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);
}
```

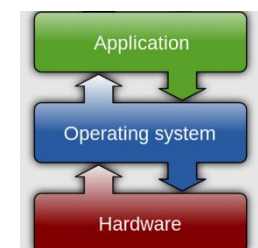


Aside: System Programming

- Systems programmers are paranoid
- We should really be writing things like:

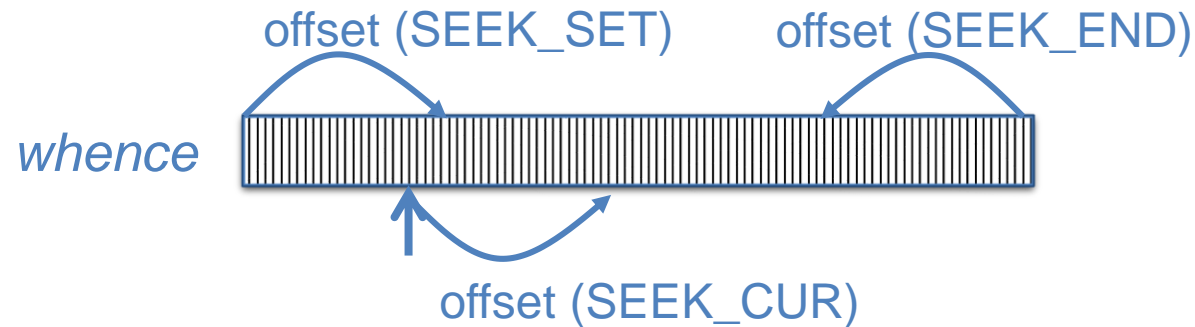
```
FILE* input = fopen("input.txt", "r");
if (input == NULL) {
    // Prints our string and error msg.
    perror("Failed to open input file");
    exit(-1);
}
```

- Check for return value from `fclose()` as well
- Be thorough about checking return values
 - Want failures to be systematically caught and dealt with

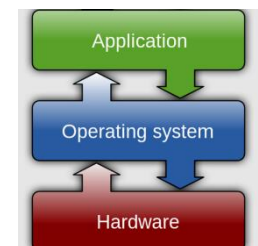


C High-Level File API: Positioning

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

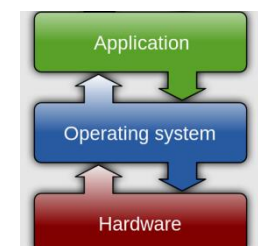


- Preserves high level abstraction of a uniform stream of objects



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 [if time]



Low-Level File I/O

- Operations on file descriptors
 - Integer that corresponds to an object in the kernel called an open file description
 - Open file description object in the kernel represents an instance of an open file
 - Why not just use a pointer?

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

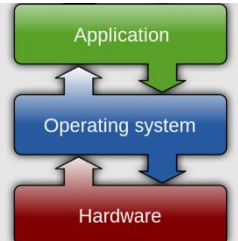
int open (const char* filename, int flags [, mode_t mode])
int creat (const char* filename, mode_t mode)
int close (int filedes)
```

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
- Needed only for file creation



C Low-Level Standard Descriptors

```
#include <unistd.h>
```

```
STDIN_FILENO - macro has value 0
```

```
STDOUT_FILENO - macro has value 1
```

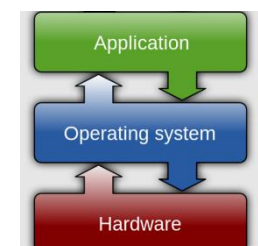
```
STDERR_FILENO - macro has value 2
```

```
int fileno (FILE* stream);
```

```
// don't close using close(), use fclose()!
```

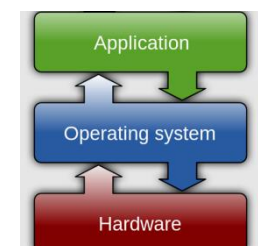
```
FILE* fdopen (int fileno, const char* opentype);
```

```
// don't close using close(), use fclose()!
```



Low-Level File API

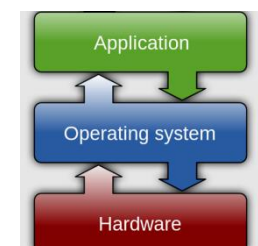
- `ssize_t read (int filedesc, void* buffer, size_t maxsize)`
 - Reads up to `maxsize` bytes – might actually read less!
 - Returns number of bytes read, 0 => EOF, -1 => error
- `ssize_t write (int filedesc, const void* buffer, size_t size)`
 - Returns number of bytes written
- `off_t lseek (int filedesc, off_t offset, int whence)`
 - Moves current position



Example: lowio.c

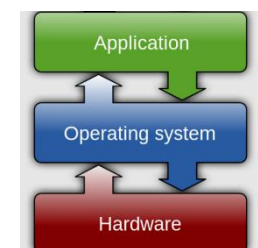
```
int main() {
    char buf[1000];
    int    fd = open("lowio.c", O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR);
    ssize_t rd = read(fd, buf, sizeof(buf));
    int    err = close(fd);
    ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

- How many bytes does this program read?



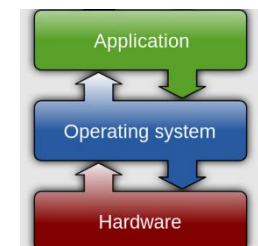
POSIX I/O: Design Patterns

- 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



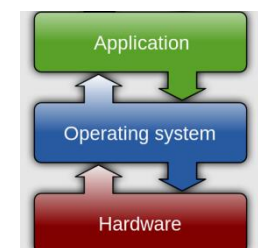
POSIX I/O: Kernel Buffering

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



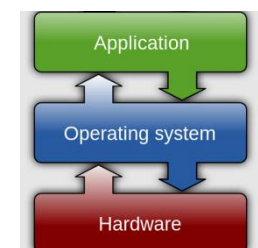
Key Unix I/O Design Concepts

- 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 look the same, reading blocks yields processor to other task
- Kernel buffered writes
 - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close



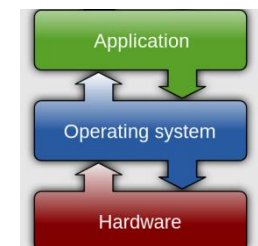
Low-Level I/O: Other Operations

- 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



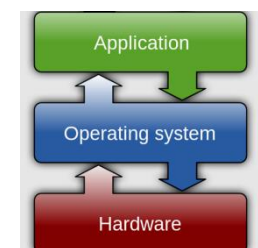
Announcements

- Project 0 deadline is next Monday
- Assignment 1 out, deadline February 23
 - You should be working on this!
- Midterm is now scheduled for Tuesday, March 17, 1200PFT, 10.30am



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(...) {
```

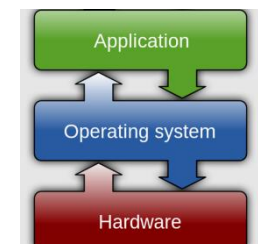
```
asm code ... syscall # into %eax  
put args into registers %ebx, ...  
special trap instruction
```

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

- Streams are buffered in user memory:

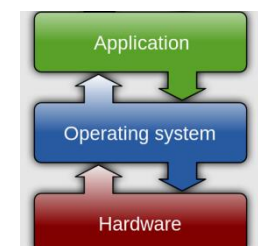
```
printf("Beginning of line ");  
sleep(10); // sleep for 10 seconds  
printf("and end of line\n");
```

- Prints out everything at once

- Operations on file descriptors are visible immediately

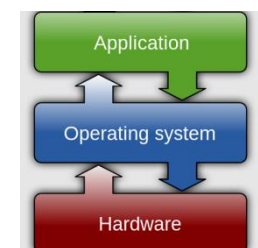
```
write(STDOUT_FILENO, "Beginning of line ", 18);  
sleep(10);  
write(STDOUT_FILENO, "and end of line \n", 16);
```

- Outputs "Beginning of line" 10 seconds earlier



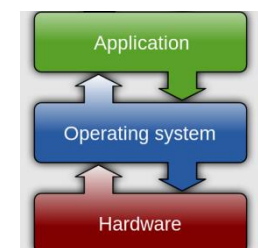
What's in a FILE*?

- FILE instance lives in user space, fopen returns pointer to it
- What's in the FILE* returned by fopen?
 - File descriptor (from call to open)
 - 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 (in user space)
 - 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 (e.g., `'\n'`)
- When you write code, make the weakest possible assumptions about how data is flushed from FILE buffers



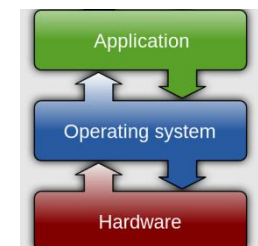
Example

- What will x be after the following code execution?

```
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);
```

- The call to fread might see the latest write 'b'
- Or it might miss it, seeing the end of file (in which case x will remain 'c')



Example

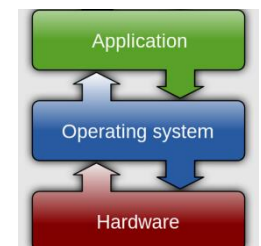
- What will x be after the following code execution?

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

```
fflush(f1);
```

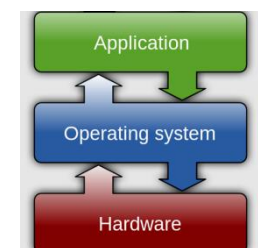
```
FILE* f2 = fopen("file.txt", "r");  
fread(&x, sizeof(char), 1, f2);
```

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



Writing Correct Code with FILE

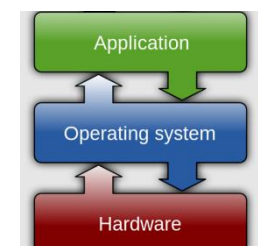
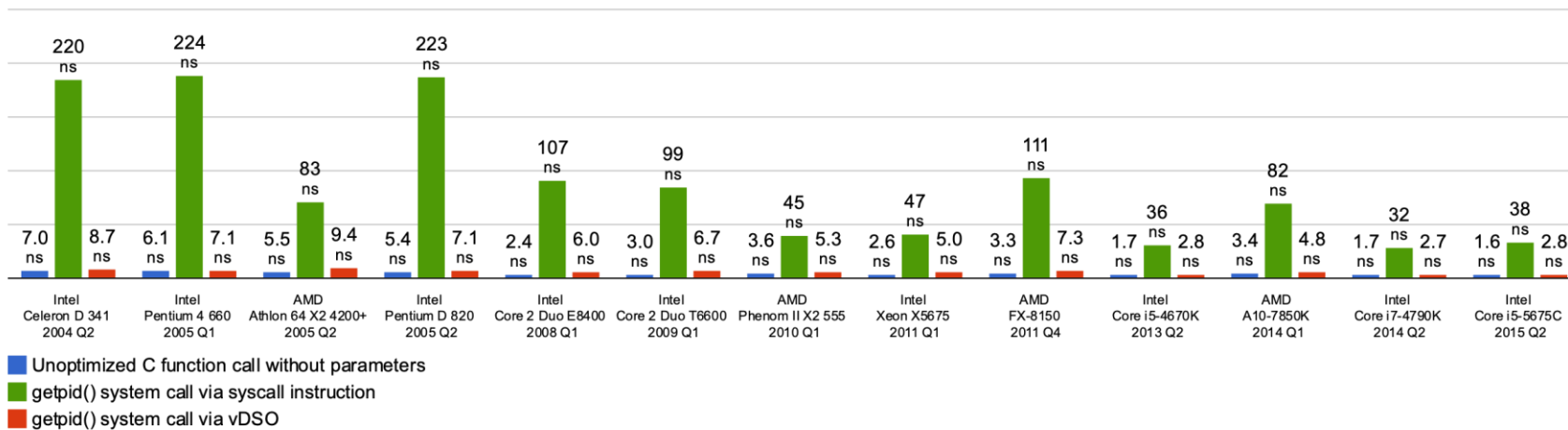
- Your code should behave correctly regardless of when/if C Standard Library flushes its buffer
 - Add your own calls to `fflush` so that data is written when you need to
 - Calls to `fclose` flush the buffer before deallocating memory and closing the file descriptor
- With the low-level file API, we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



Why Buffer in Userspace?

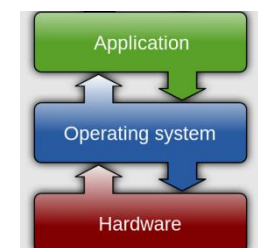
Overhead!

- Syscalls are 25x more expensive than function calls (~100 ns)
- read/write a file byte by byte? Max throughput of ~10MB/second
- With fgetc? Keeps up with your SSD (much faster)



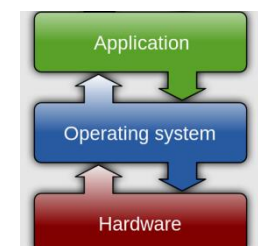
Why Buffer in Userspace? Functionality!

- System call implements operations less capable
 - Simplifies operating system
 - Makes system call composable
- Example: No “read until new line” operation
 - Solution: Make a big read syscall, find first new line in userspace



Today: The File Abstraction

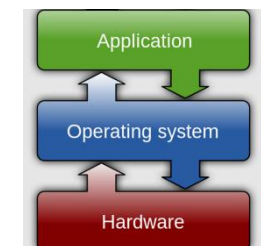
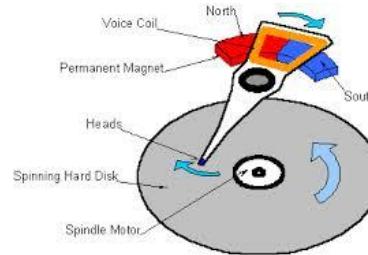
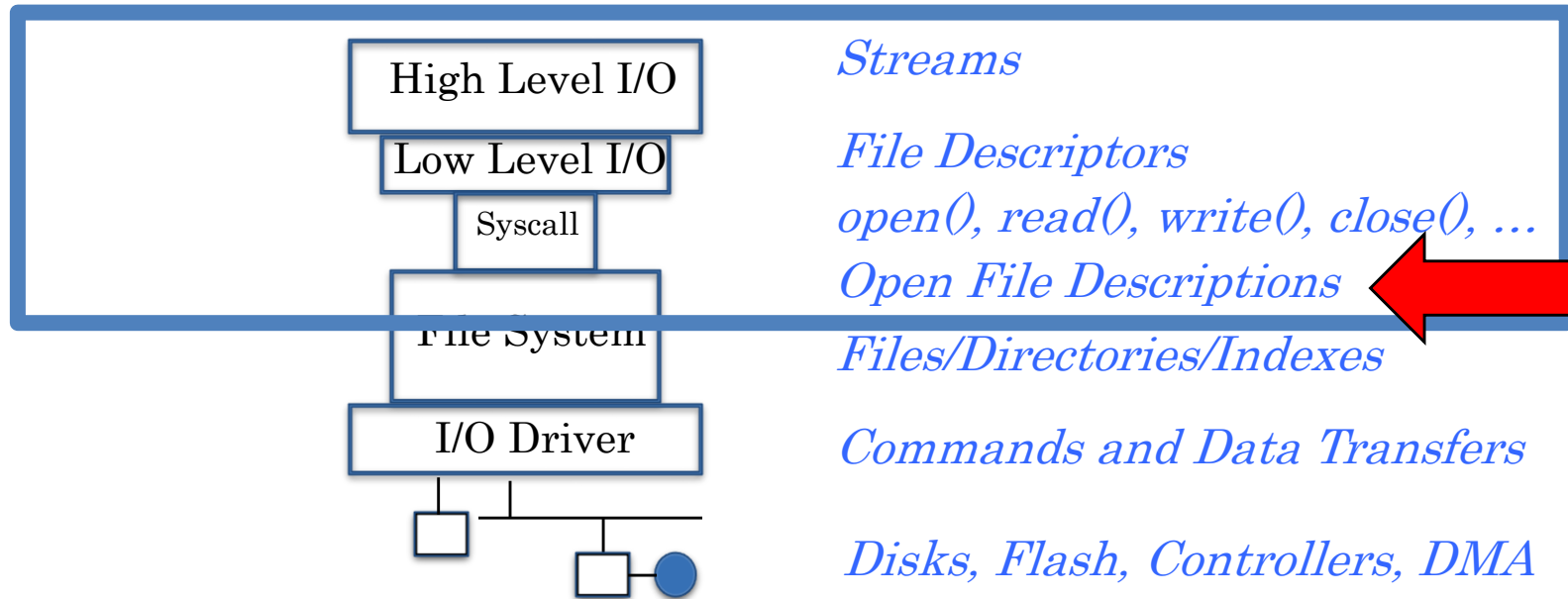
- High-Level File I/O: Streams
- Low-Level File I/O: File Descriptors
- How and Why of High-Level File I/O
- Process State for File Descriptors
- Some Pitfalls with OS Abstractions [if time]



I/O and Storage Layers

Application / Service

Focus of today's lecture

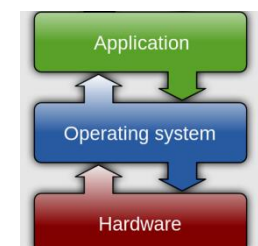


Kernel Maintains State

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

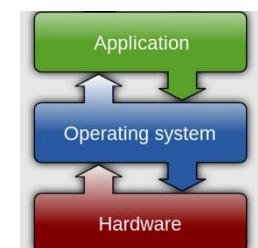
The kernel remembers that the `int` it receives (stored in `fd`) corresponds to `foo.txt`

The kernel picks up where it left off in the file



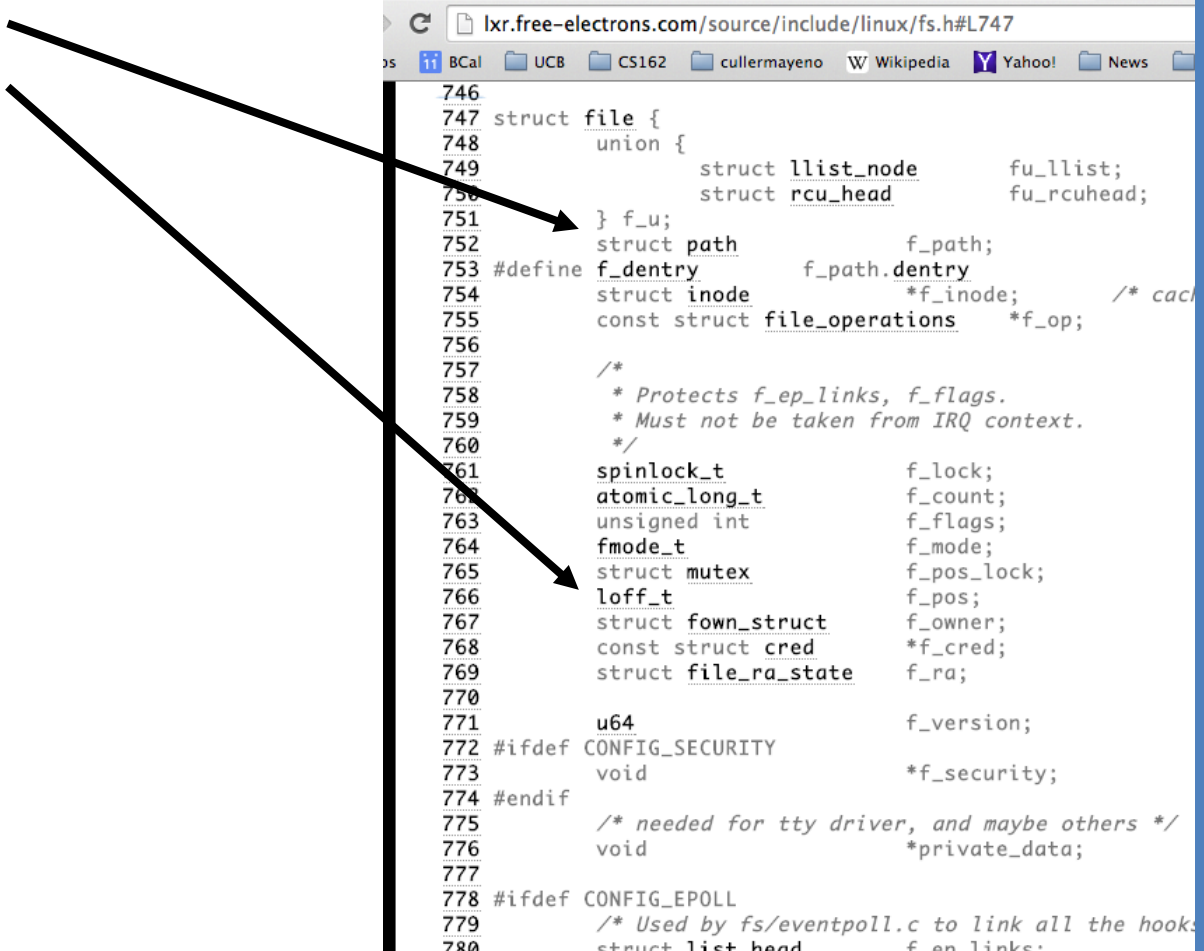
State Maintained by the Kernel

- 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 a file descriptor to an 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
- A call to `close()` removes the file descriptor mapping and deallocates the file description (if no other processes refer to it)



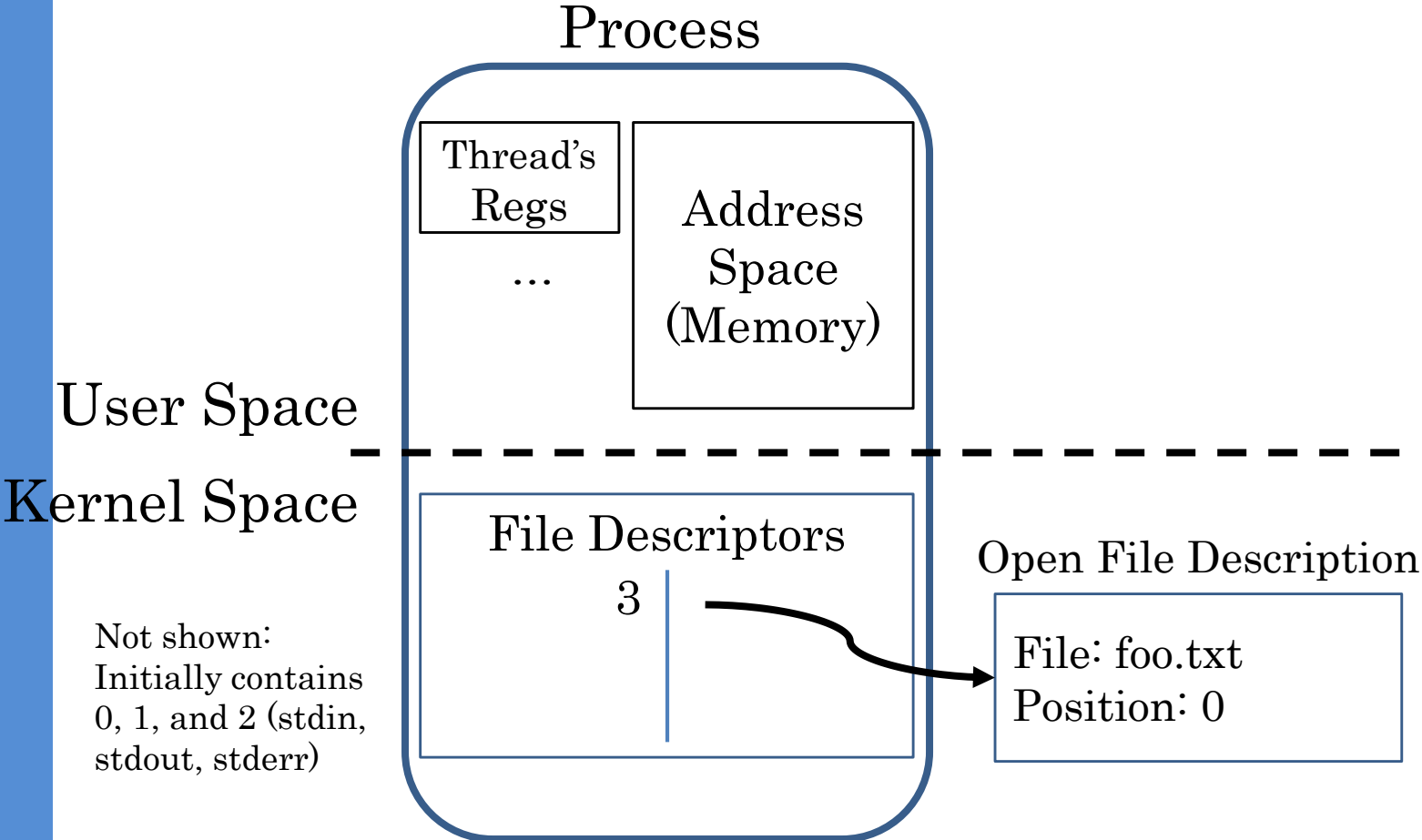
What's in an Open File Description?

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

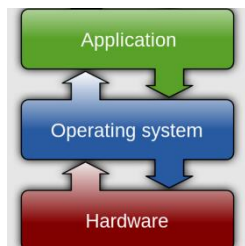


```
746
747 struct file {
748     union {
749         struct llist_node    fu_llist;
750         struct rcu_head      fu_rcuhead;
751     } f_u;
752     struct path              f_path;
753 #define f_dentry            f_path.dentry
754     struct inode             *f_inode; /* cache */
755     const 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 hooks
780      * struct list head      f_ep_links;
```

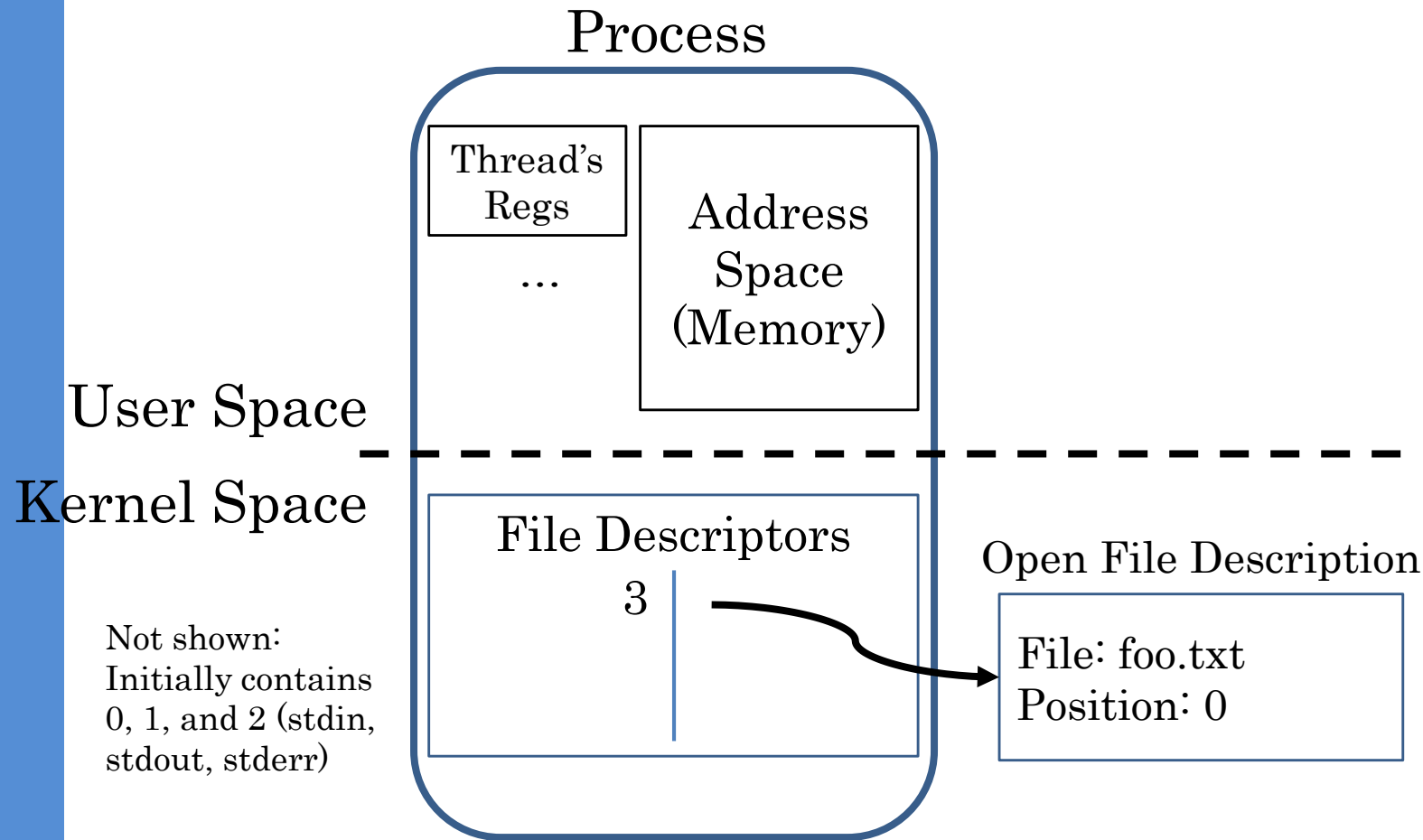
Abstract Representation of a Process



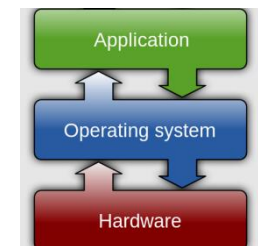
- Suppose that we execute `open("foo.txt", ...)`
- and that the result is 3



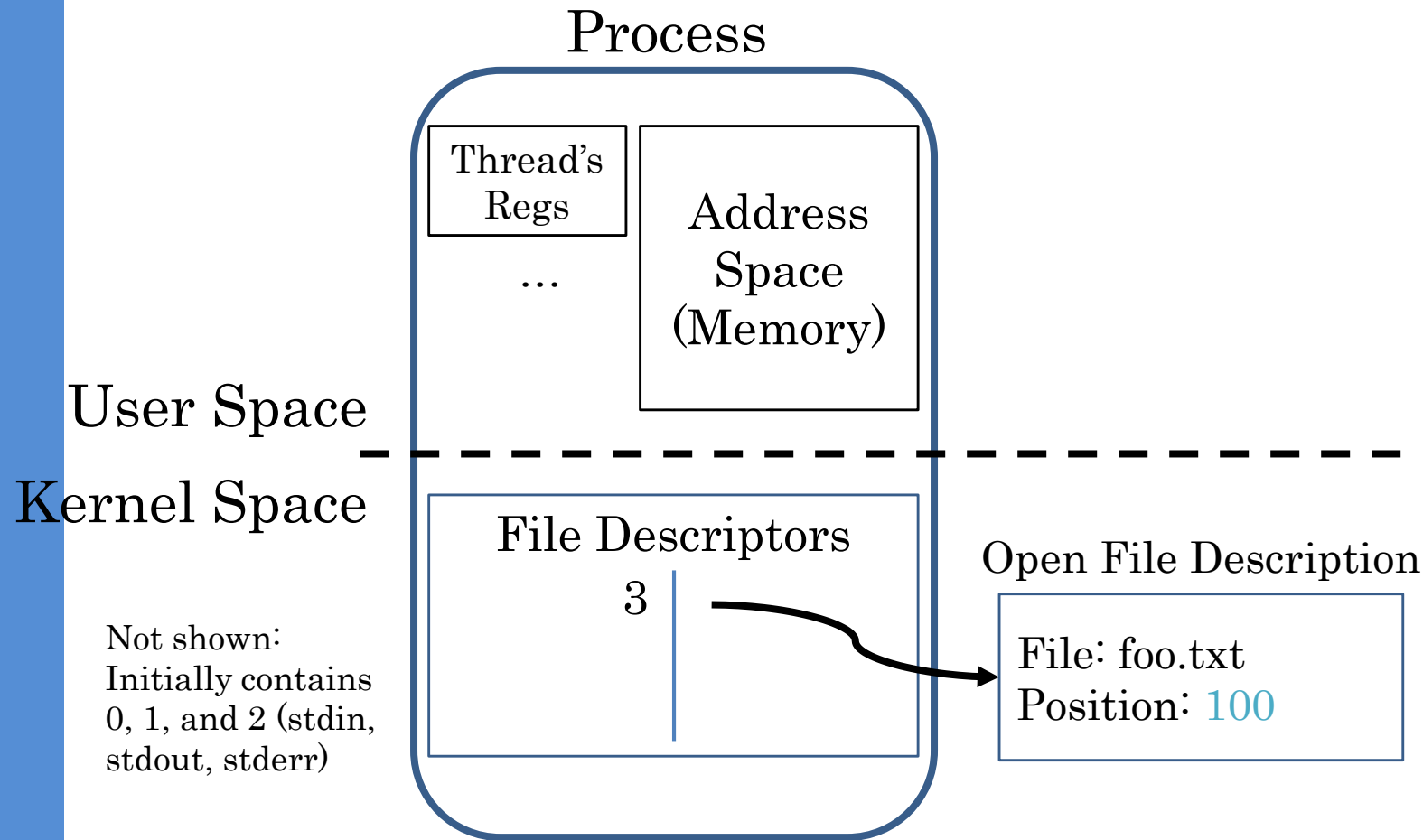
Abstract Representation of a Process



- 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

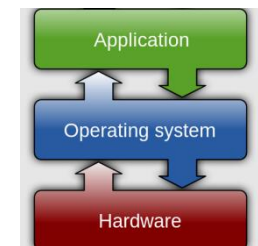


Abstract Representation of a Process

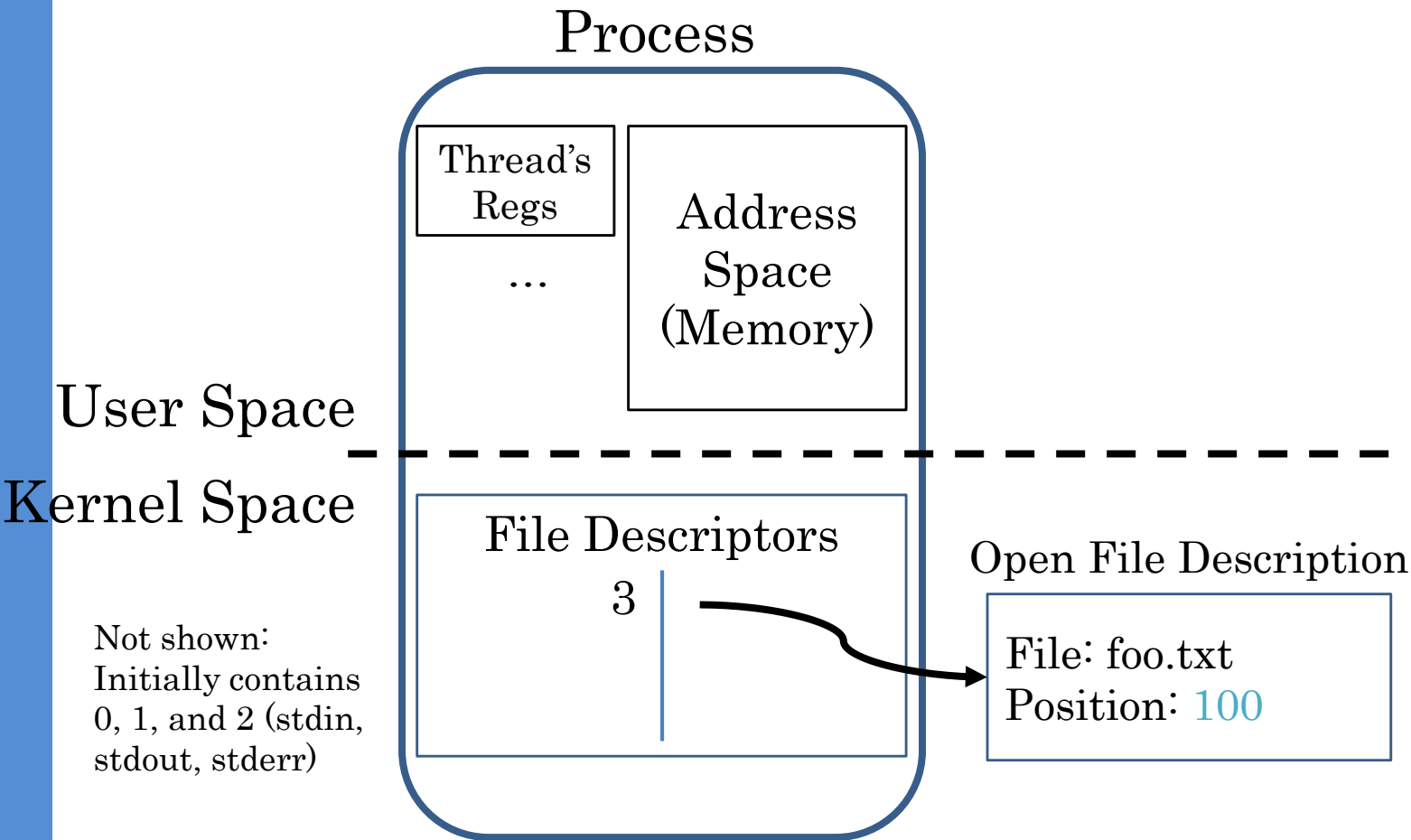


Not shown:
Initially contains
0, 1, and 2 (stdin,
stdout, stderr)

- 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

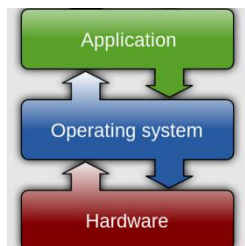


Abstract Representation of a Process

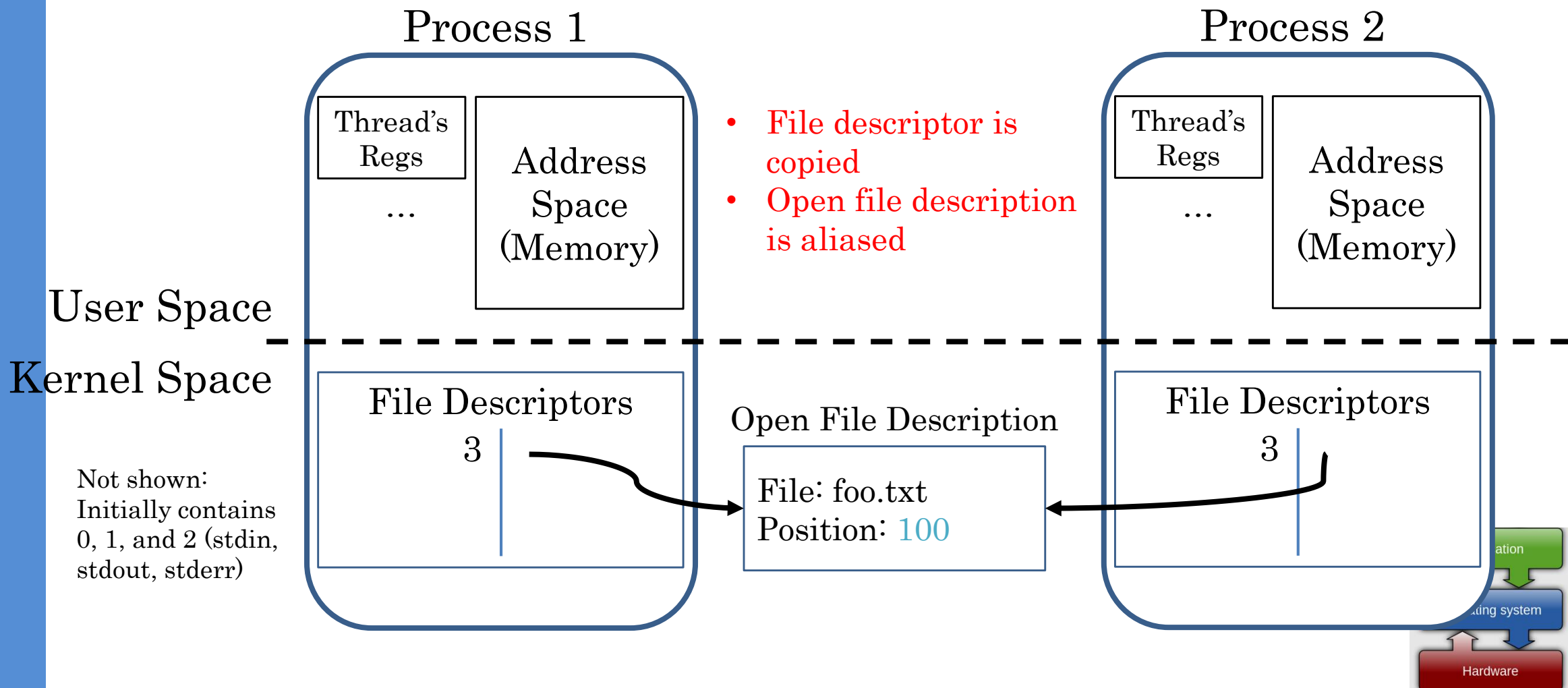


Not shown:
Initially contains
0, 1, and 2 (stdin,
stdout, stderr)

- 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)`



Now, let's fork()!



Open File Description is Aliased

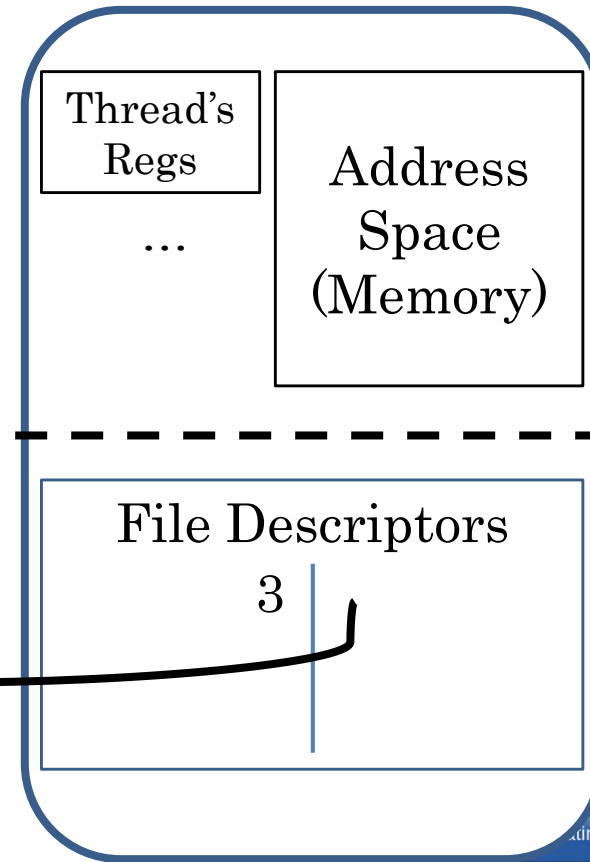
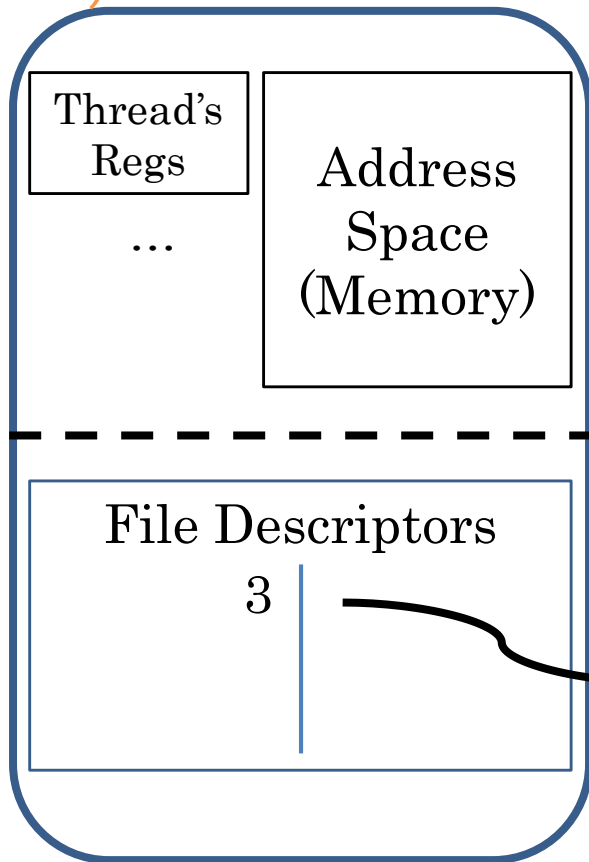
`read(3, buf, 100)`

Process 1

Process 2

User Space

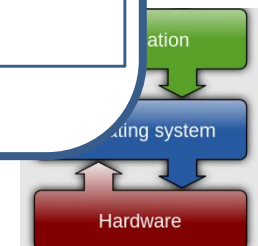
Kernel Space



Open File Description



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



Open File Description is Aliased

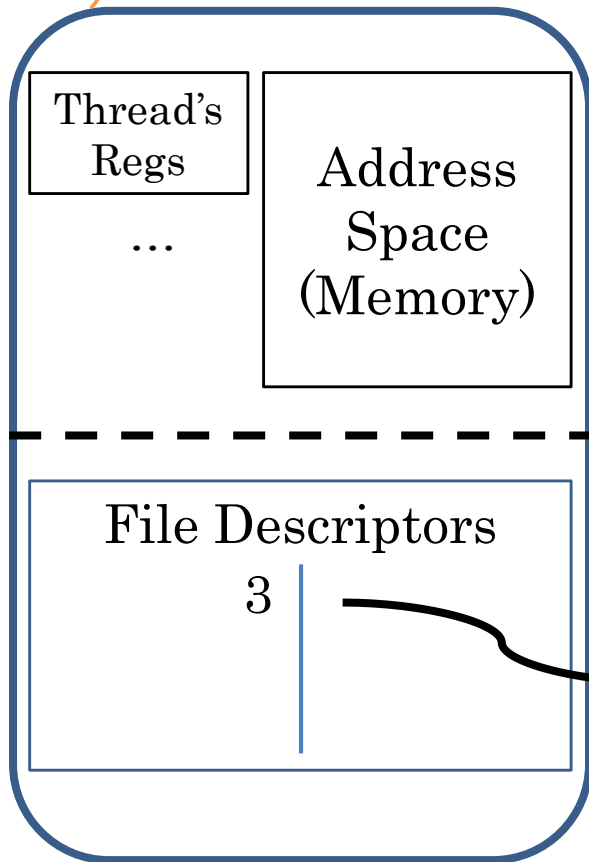
`read(3, buf, 100)`

Process 1

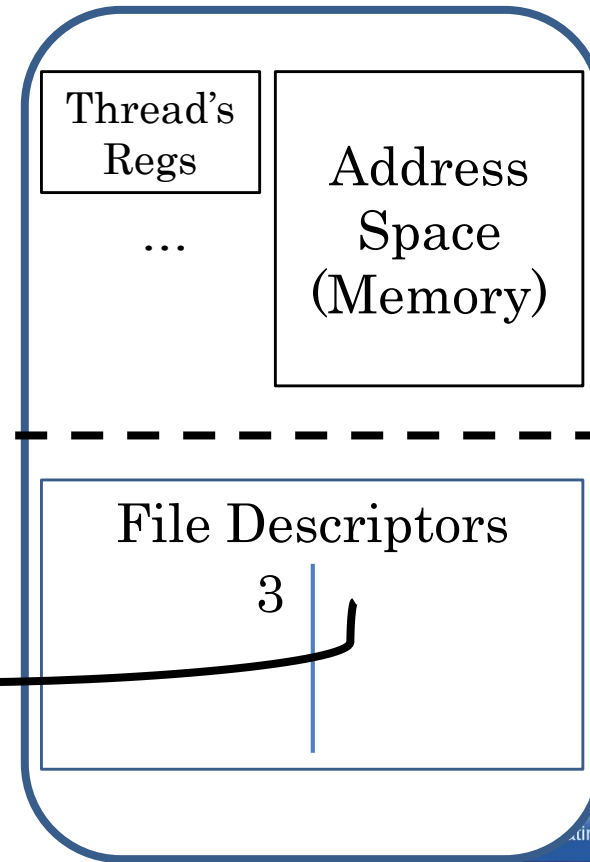
Process 2

User Space

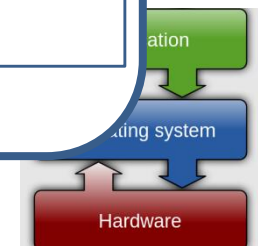
Kernel Space



Open File Description



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



Open File Description is Aliased

`read(3, buf, 100)`

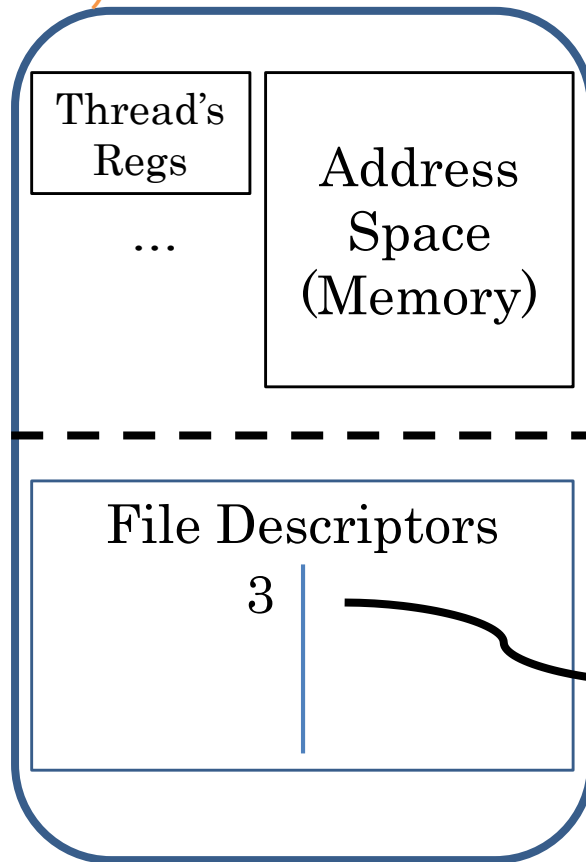
Process 1

`read(3, buf, 100)`

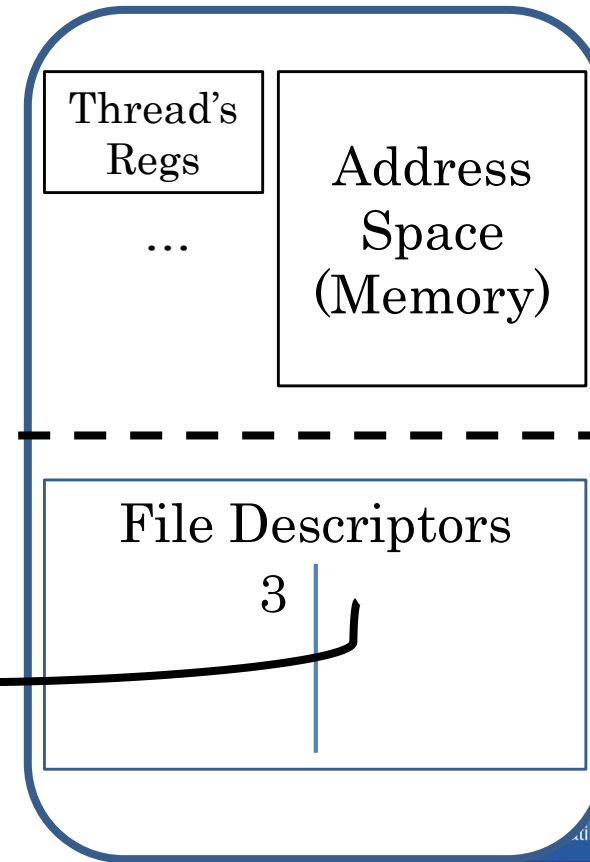
Process 2

User Space

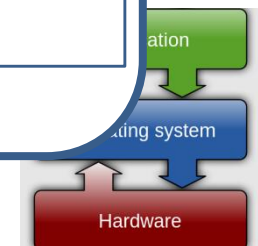
Kernel Space



Open File Description



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



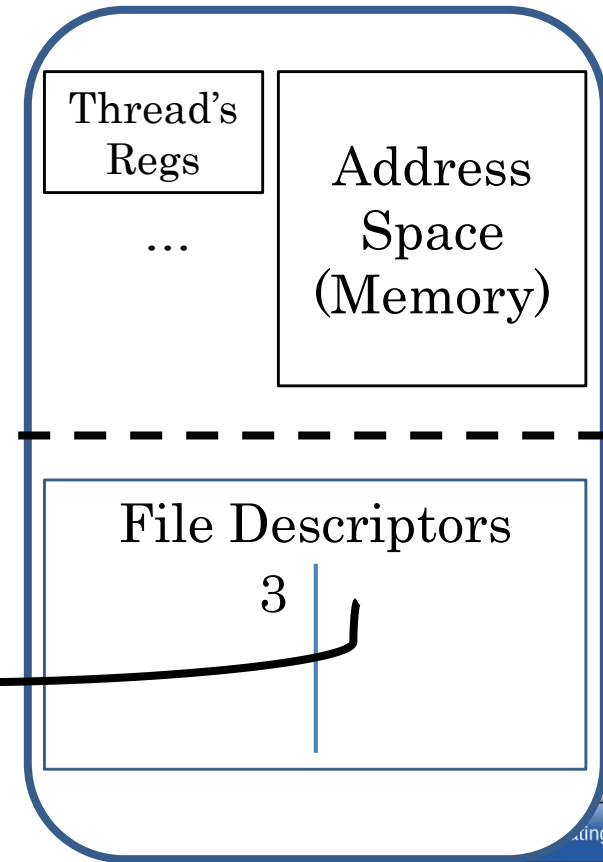
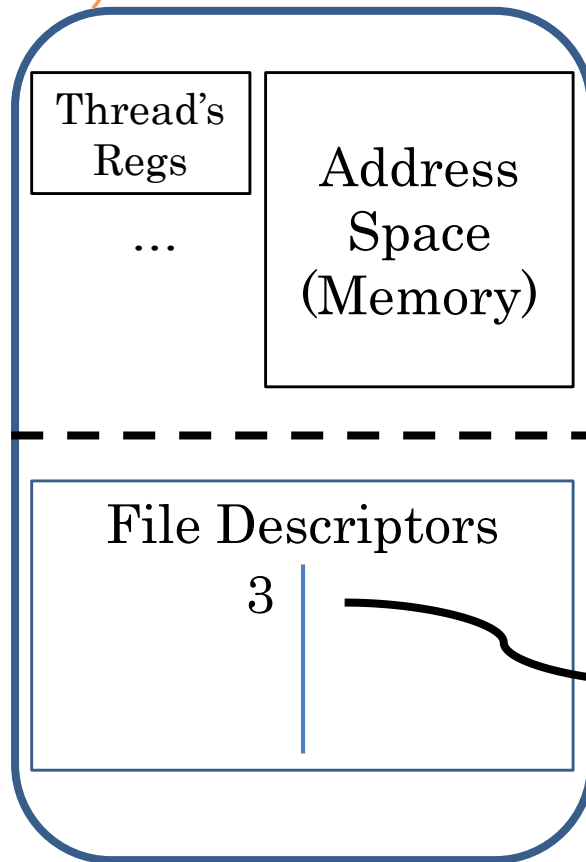
Open File Description is Aliased

`read(3, buf, 100)`

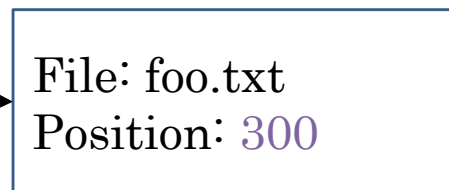
Process 1

`read(3, buf, 100)`

Process 2



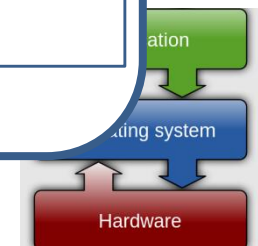
Open File Description



User Space

Kernel Space

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



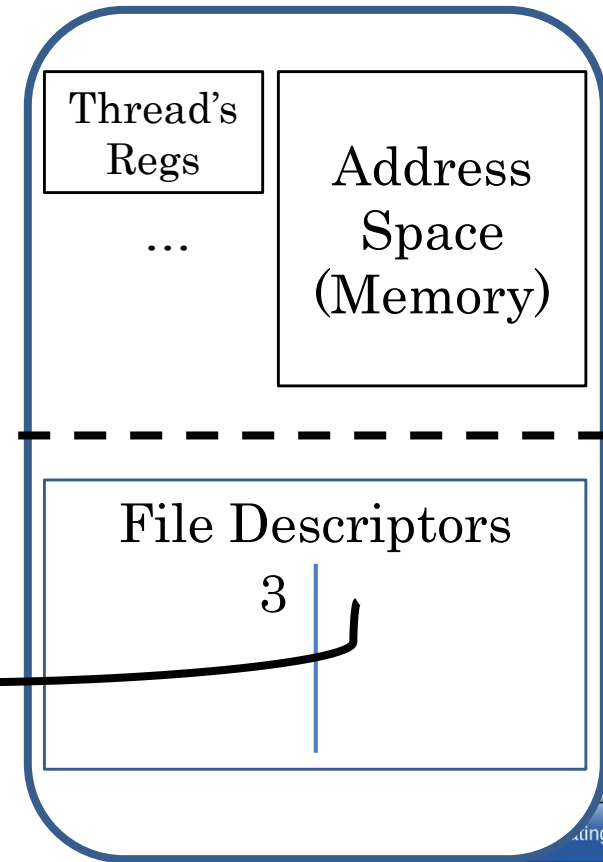
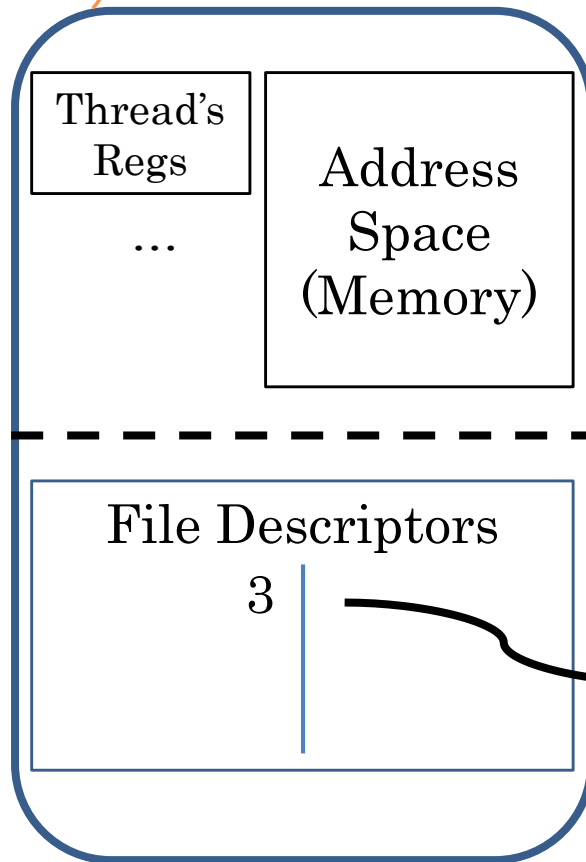
File Descriptor is Copied

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

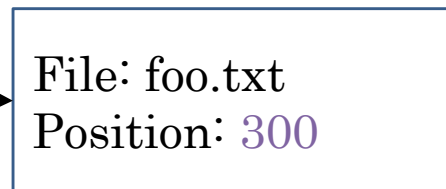
Process 1

`read(3, buf, 100)`

Process 2



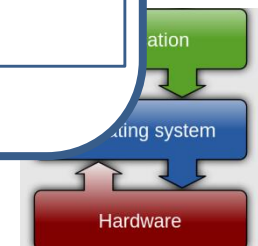
Open File Description



User Space

Kernel Space

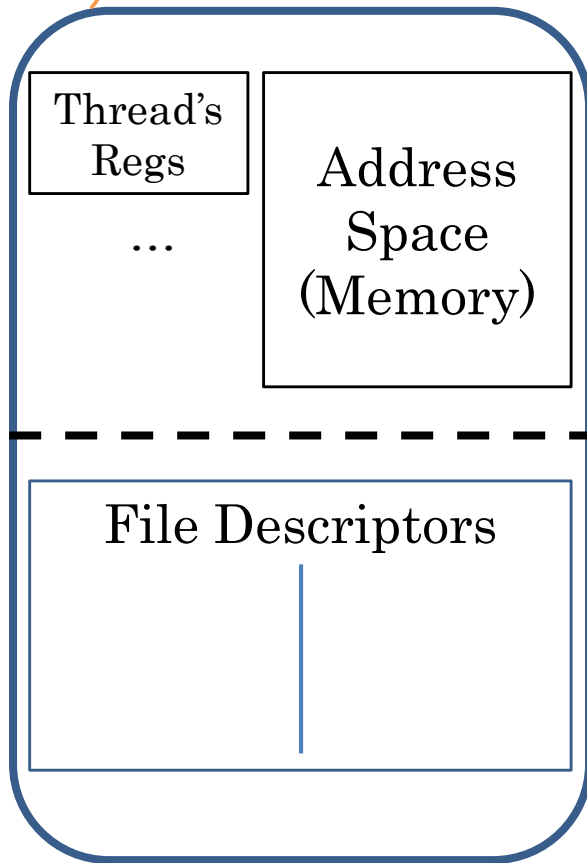
Not shown:
Initially contains
0, 1, and 2 (stdin,
stdout, stderr)



File Descriptor is Copied

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

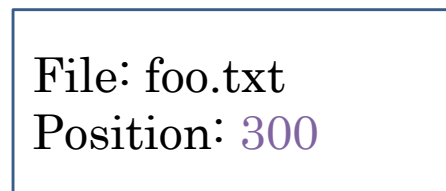
Process 1



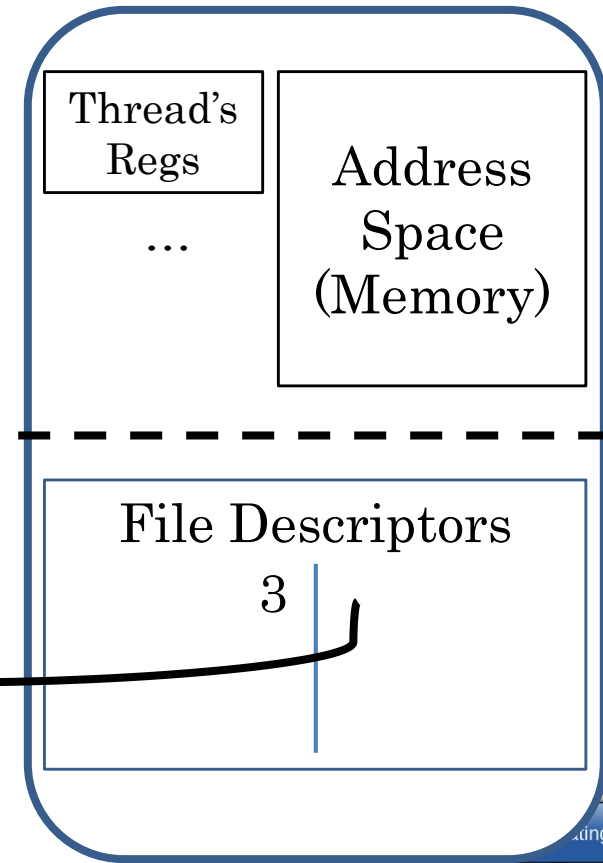
`read(3, buf, 100)`

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

Open File Description



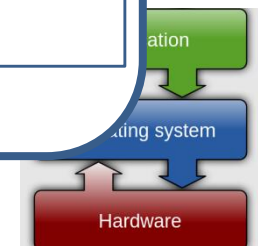
Process 2



User Space

Kernel Space

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

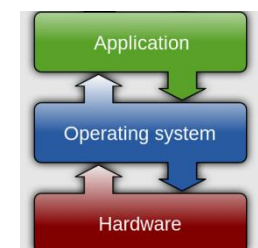


Why is Aliasing the Open File Description a Good Idea?

It allows for shared resources between processes

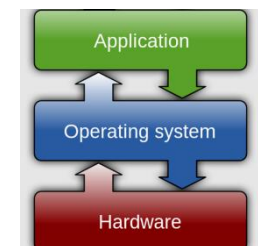
Recall: In POSIX, Everything is a “File”

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

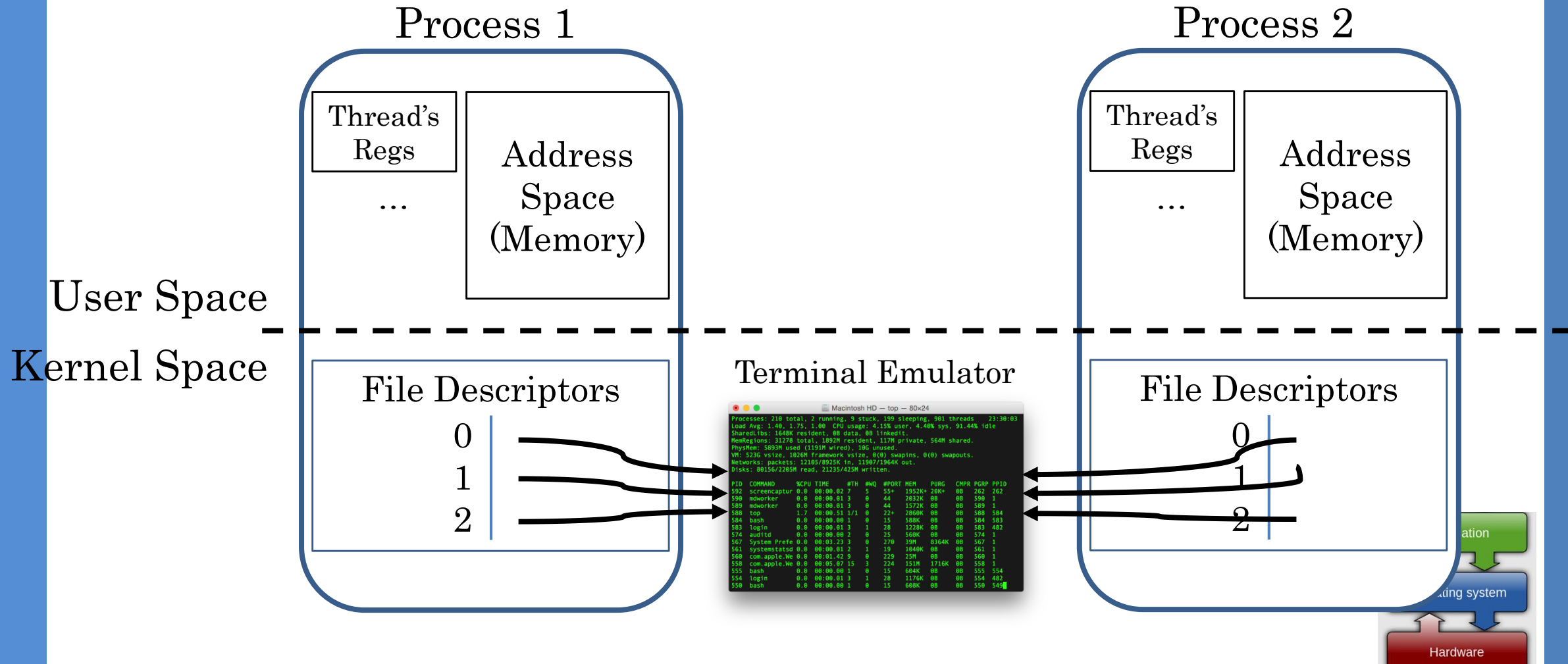


Example: Shared Terminal Emulator

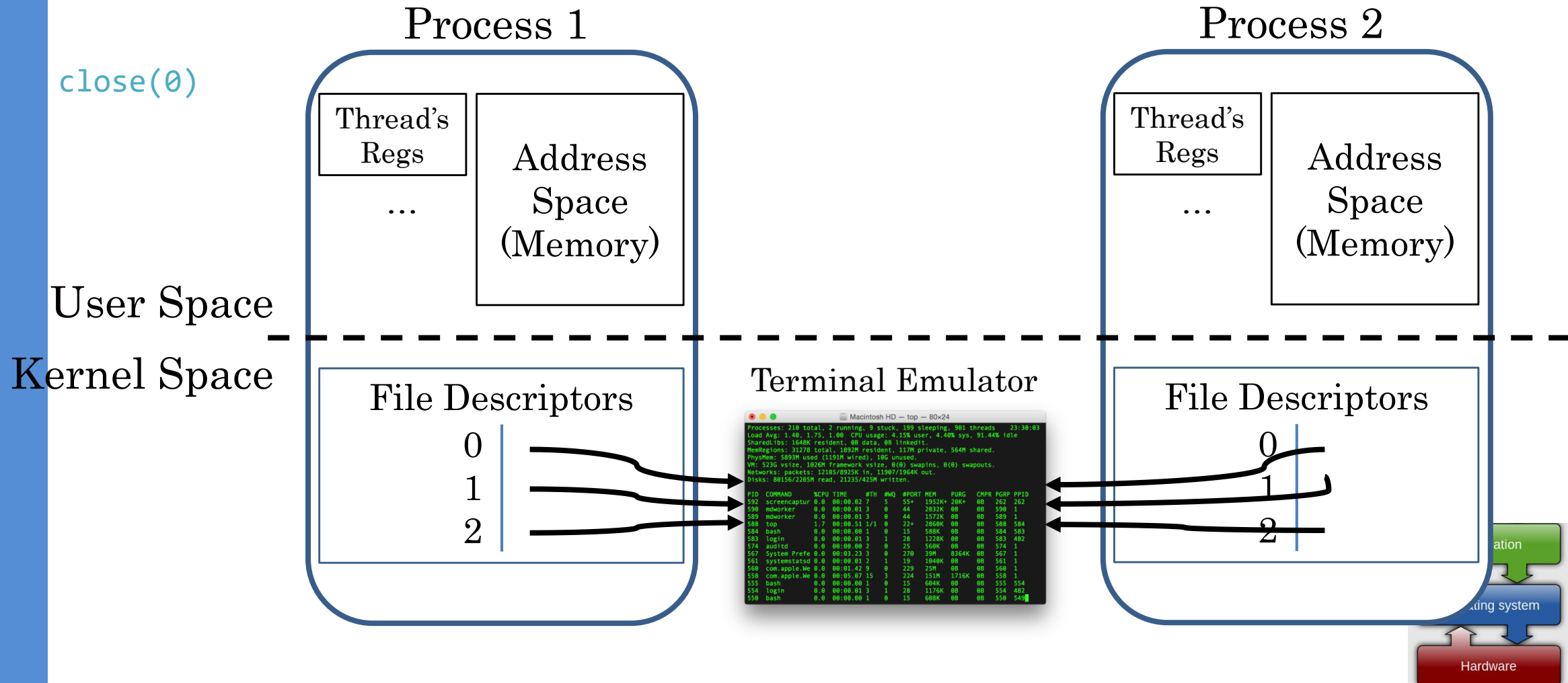
- When you `fork()` a process, the parent's and child's `printf` outputs go to the same terminal



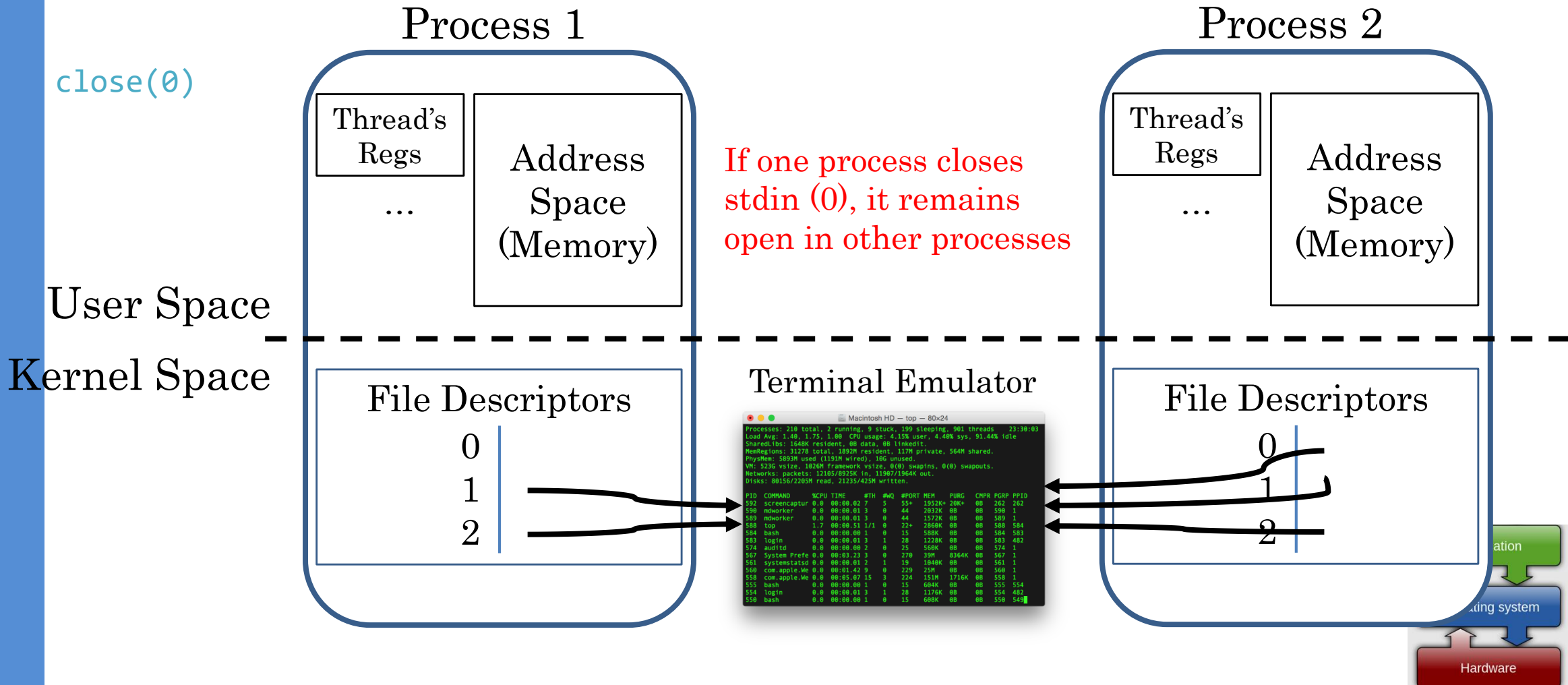
Example: Shared Terminal Emulator



Example: Shared Terminal Emulator

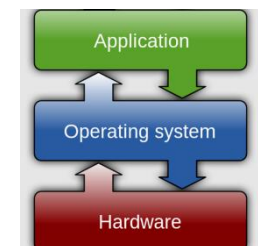


Example: Shared Terminal Emulator



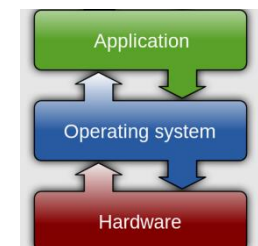
Other Examples

- Shared network connections after `fork()`
 - Allows handling each connection in a separate process
 - We'll explore this next time
- Shared access to pipes
 - Useful for interprocess communication
 - And in writing a shell (Assignment 2)

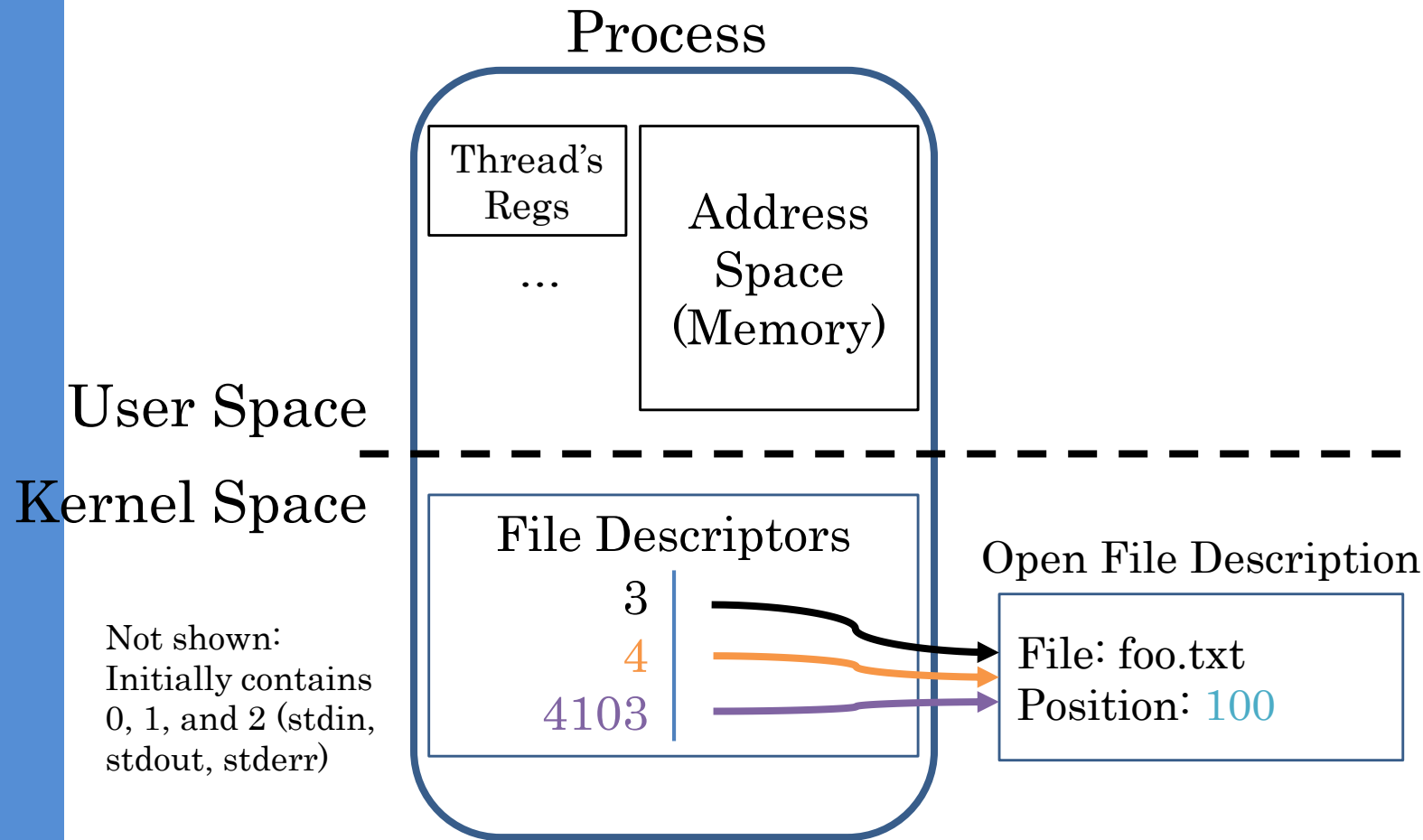


Other Syscalls: dup and dup2

- They allow you to duplicate the file descriptor
- But the open file description remains aliased

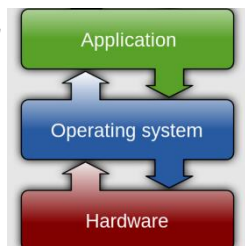


Other Syscalls: dup and dup2



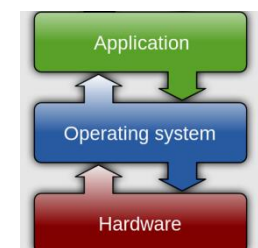
Not shown:
Initially contains
0, 1, and 2 (stdin,
stdout, stderr)

- 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, 4103)`



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]

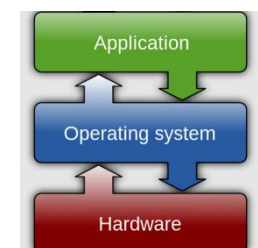


Don't `fork()` in a process that already has multiple threads

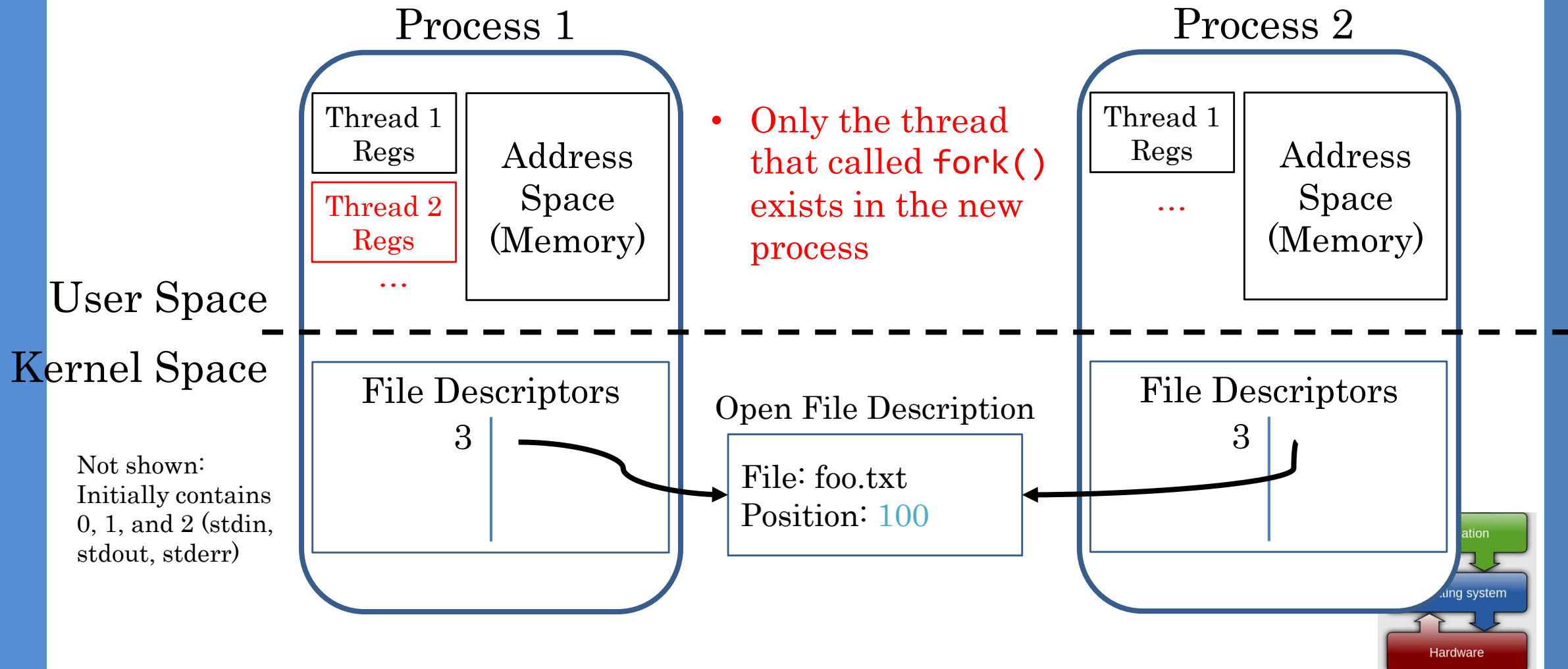
Unless you plan to call `exec()` in the child process

fork() in Multithreaded Processes

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

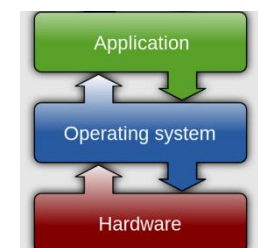


fork() in a Multithreaded Processes



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



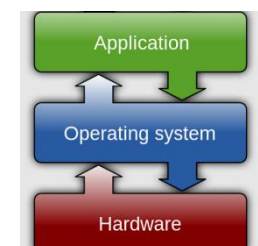
Don't carelessly mix
low-level and high-
level file I/O

Avoid Mixing FILE* and File Descriptors

- What is the value has y after executing the following code?

```
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 10
```

- Bytes 0 to 9?
- Bytes 10 to 19?
- None of these?



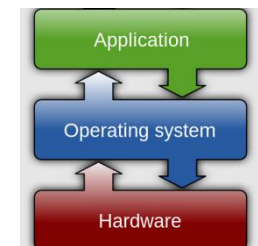
Be careful with `fork()` and `FILE*`

Be Careful Using `fork()` with `FILE*`

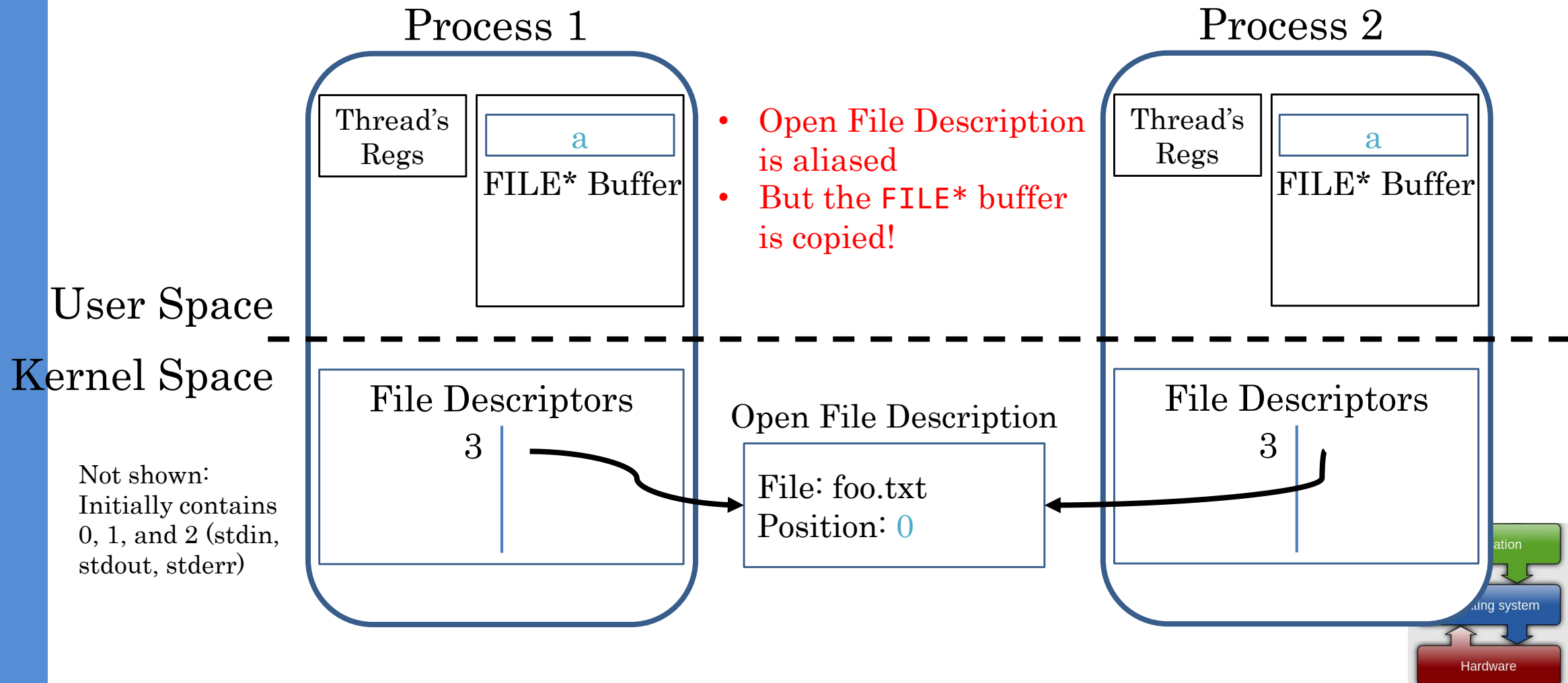
```
FILE* f = fopen("foo.txt", "w");  
fwrite("a", 1, 1, f);  
fork();  
fclose(f);
```

← Depends on whether this `fwrite` call flushes...

- After all processes exit, what is in `foo.txt`?
 - Could be either `a` or `aa`
- Usually `aa` based on what I've observed in Linux...



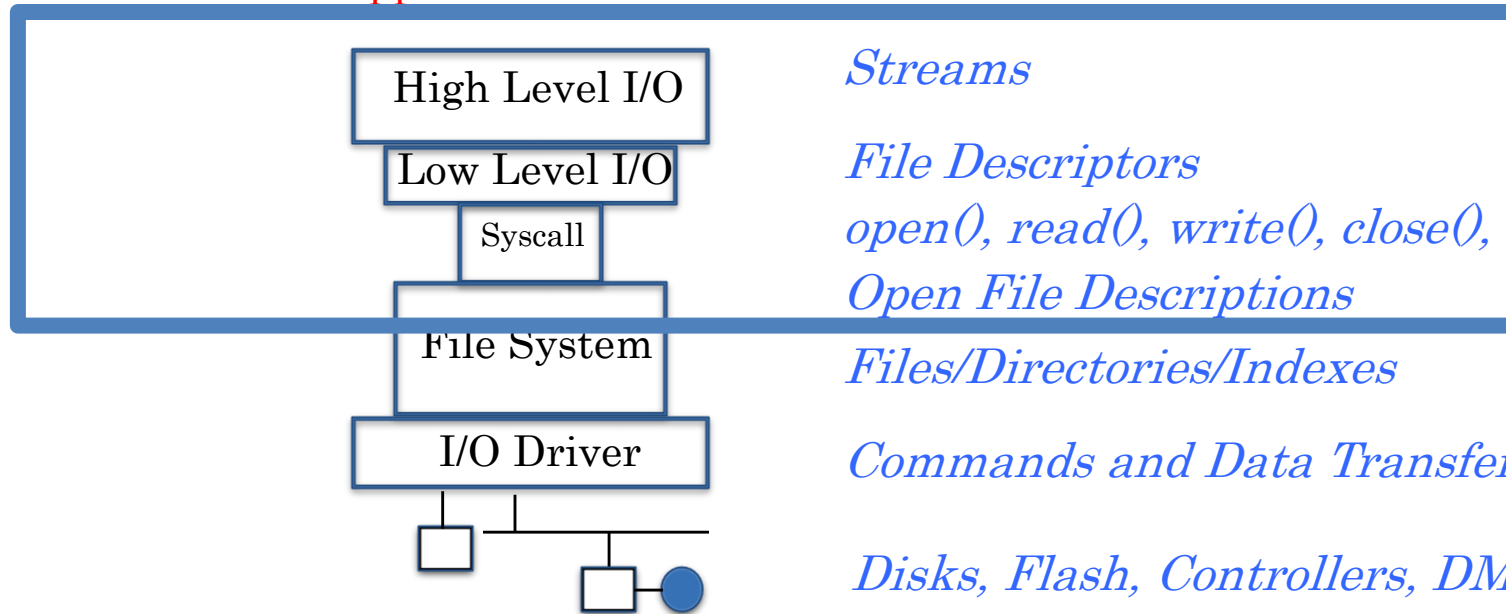
Be Careful Using `fork()` with `FILE*`



Conclusion

Application / Service

Focus of today's lecture



Streams

File Descriptors

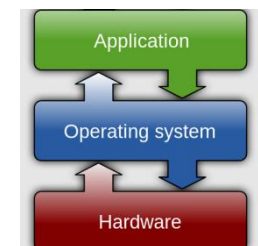
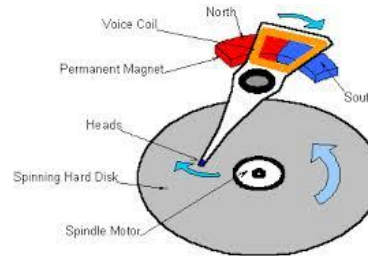
open(), read(), write(), close(), ..

Open File Descriptions

Files/Directories/Indexes

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Conclusion

- POSIX idea: “everything is a file”
- All sorts of I/O managed by open/read/write/close
- We added two new elements to the PCB:
 - Mapping from file descriptor to open file description
 - Current working directory

