CS 1217

Lecture 5: fork(), pipe(), exec()

Logistics

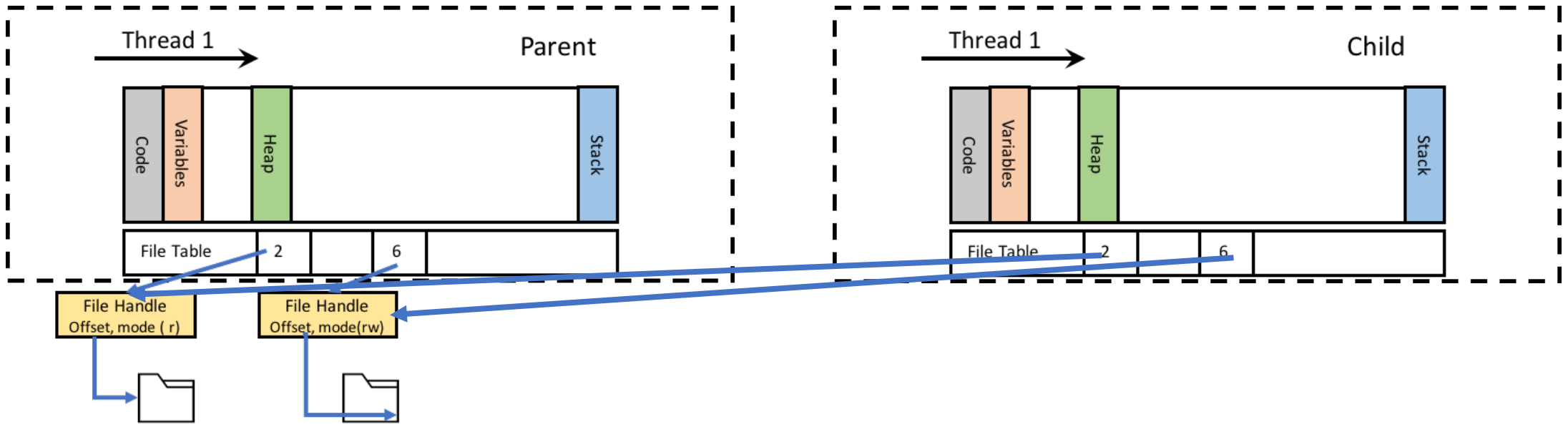
- Assignment 2 goes out today

- Soham will explain Github Classroom after lecture; stay back for a few minutes

Recap

- Process creation
- `fork()`
- Semantics of `fork()`
  - `fork()` copies the caller process.
  - `fork()` copies the address space.
  - `fork()` copies the process file table.

Process Creation : `fork()`



Process creation : fork()

```
int returnCode = fork();

if (returnCode == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());

} else {
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n",
           __________ , (int) getpid());
}
```

- How many times do you think fork() returns?
- What are the return values?

`fork()` Issues

- For multi-threaded programs, `fork()` copies the context of the thread calling `fork()`

- What if the child process wanted to create more threads?

What happens here?

```
while (1) {
        fork();
}
```

`fork()` Issues

- Can be expensive. Why?
  - Needs to copy a lot of state, esp. in memory, could be very large
  - Might not be worthwhile if all the child process wants to do is something **new**.

- How can you optimize?
  - **Optimize existing semantics**: through copy-on-write, a clever memory-management optimization we will discuss later in the class.
  - **Change the semantics**: `vfork()`: fails if the child does anything other than immediately load a new executable.

Optimizing `fork()`

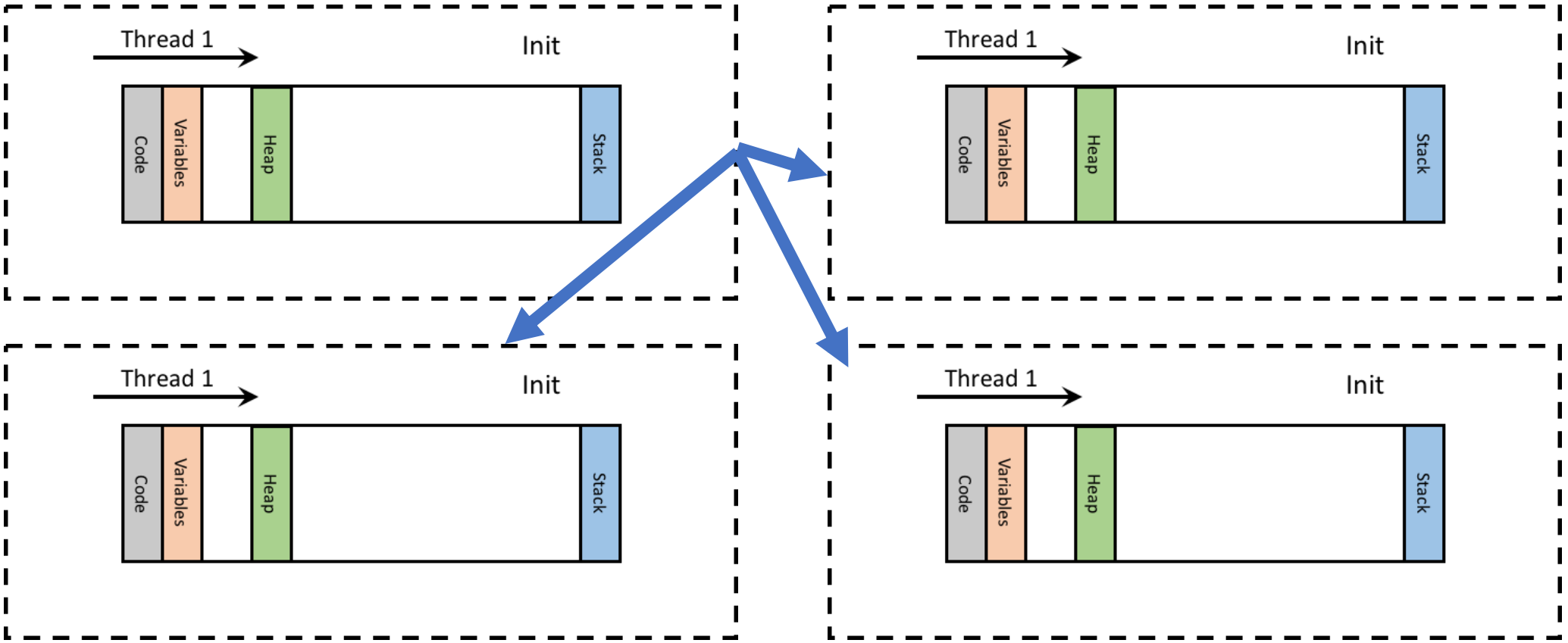
- `man clone`
  - Remember, man pages are your friend

- `clone()`, similar to `fork()`, but
  - Allows for more “stuff” to be shared between the parent and child, optimizing process creation by allowing for copy-on-write semantics for the child
    - What could this stuff be?
  - Can allow the child to start execution from a function, using a function pointer, rather than where `fork()` left off

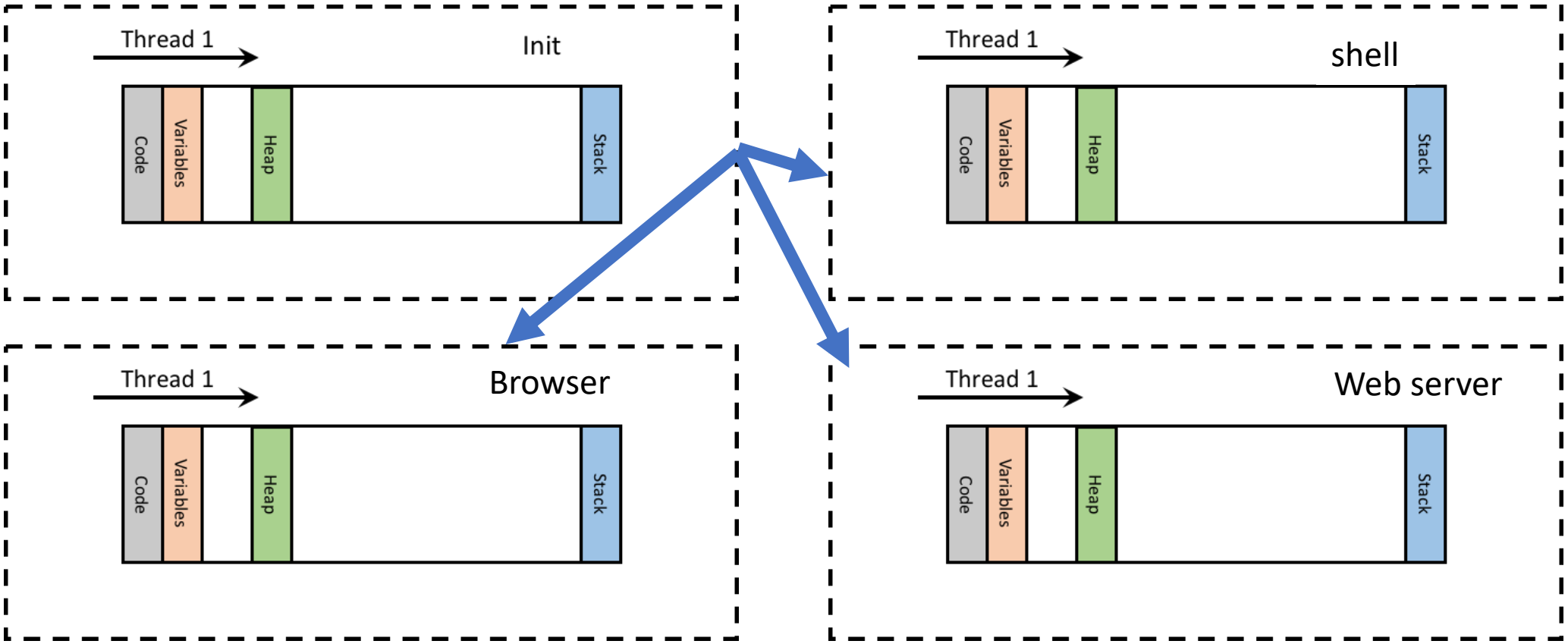
Process Birth, different karmic path: `exec()`

- `fork()` replicates the program's process, what if you wanted the child to run a different program?

The need for `exec()`



The need for `exec()`



`exec()` Semantics

- Find and read the executable from the disk
  - Interpret the executable
- Replace the abstractions of the newly created process with those of the new program, or what the new process wants to be
- Things that need to be changed
  - The address space
  - CPU state, where to start executing instructions from

Interpreting “executable” files

- ELF: Executable and Linkable Format: Defines the format in which information is stored for all executable files

```
cs304@cs304-devel:~/test$ readelf -l cpu_test | more

Elf file type is EXEC (Executable file)
Entry point 0x400a30
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000b56b6 0x00000000000b56b6  R E    0x200000
  LOAD           0x00000000000b6120 0x00000000006b6120 0x00000000006b6120
                 0x00000000000051b8 0x00000000000068e0  RW     0x200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      0x4
  TLS            0x00000000000b6120 0x00000000006b6120 0x00000000006b6120
                 0x0000000000000020 0x0000000000000060  R      0x8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
```

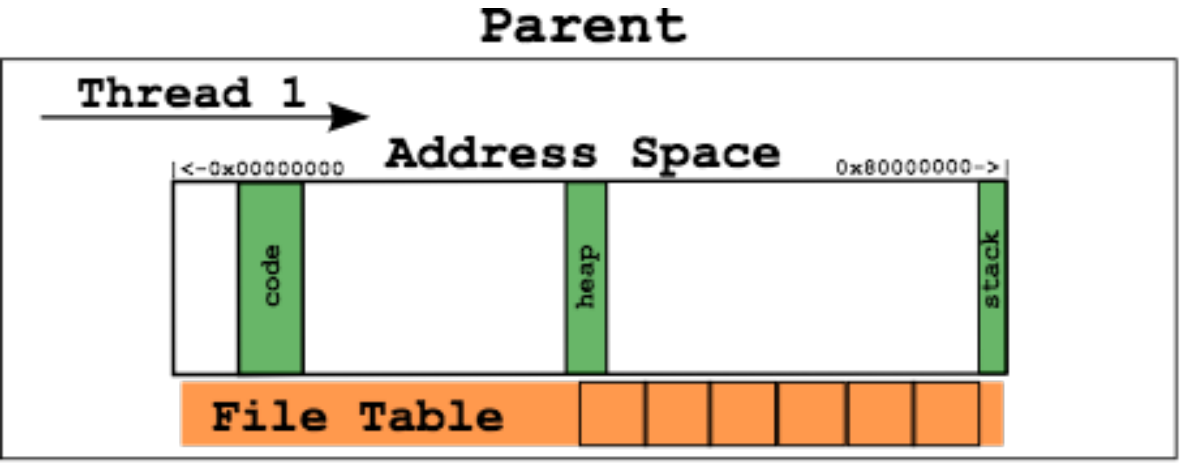
pipe()

```
cs304@cs304-devel:~$ ps aux | grep cs304
```

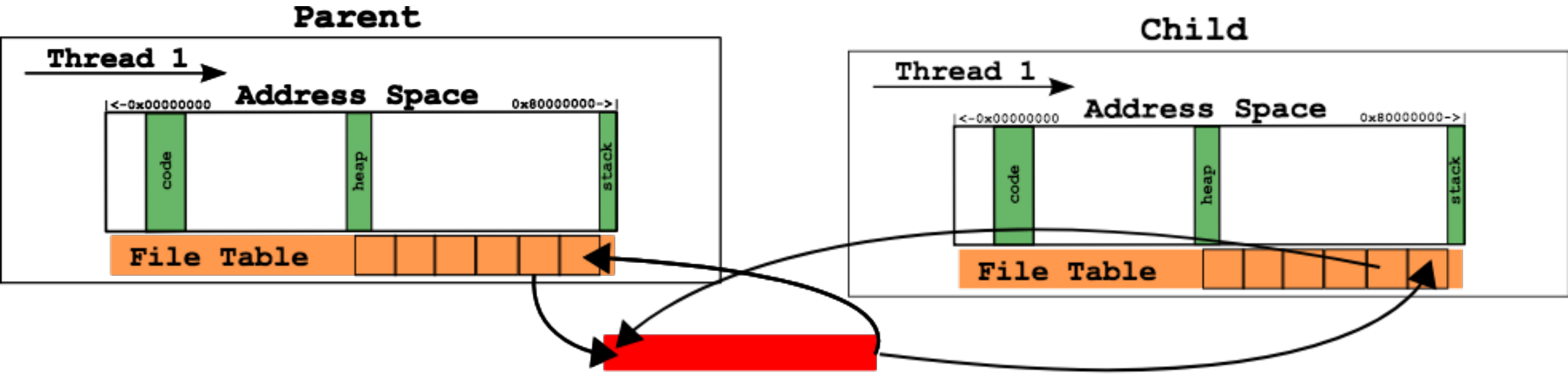
- There are two process created, which communicate using a pipe
- The pipe is implemented using the `pipe()` syscall
- `pipe()` creates an “anonymous pipe object” and returns a two **file descriptors**: one for the read-only end, and the other for the write-only end.
- Data written to the write-only end is *immediately* available at the read-only end

- Where are the pipe contents located?

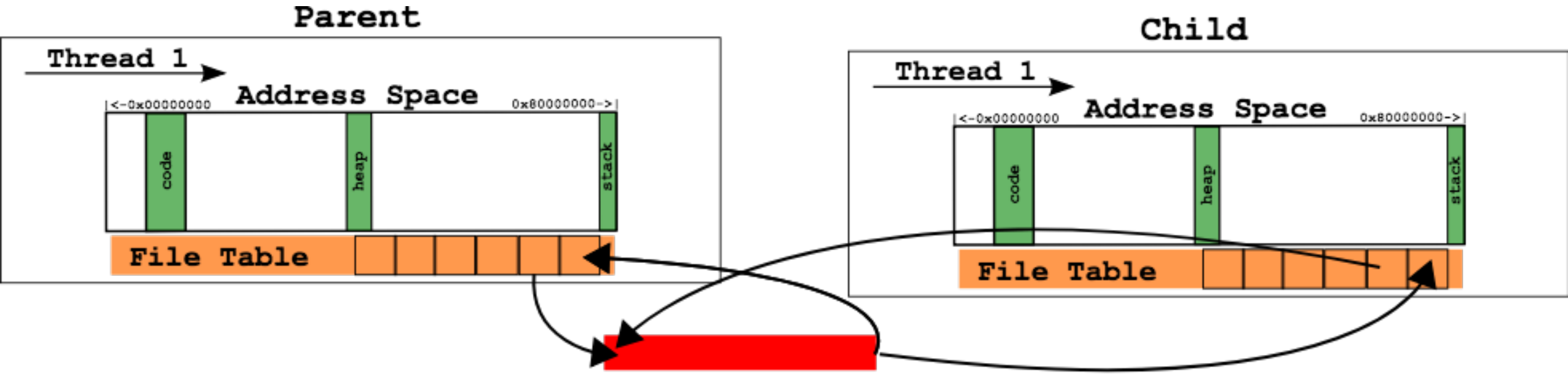
IPC using `fork()` and `pipe()`



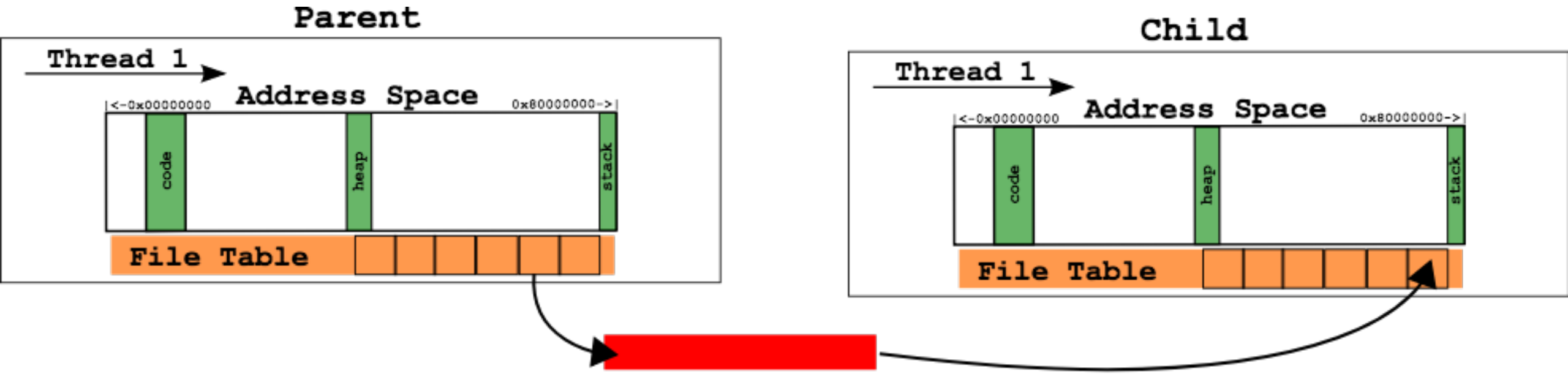
IPC using `fork()` and `pipe()`



IPC using `fork()` and `pipe()`



IPC using `fork()` and `pipe()`



`fork()` and `pipe()` in action

```
//pipeEnds[0] gets the read end;
//pipeEnds[1] gets the write end.
int pipeEnds[2];
pipe(pipeEnds);
int returnCode = fork();

if (returnCode == 0) {
    //Why do this?
    close(pipeEnds[1]);
    //Read some data from the pipe.
    char data[14];
    read(pipeEnds[0], data, 14);
    } else {
    //Why do this?
    close(pipeEnds[0]);

    //Write some data to the pipe.
    write(pipeEnds[1], "Hello, child!\n", 14);
    }
```

Passing arguments to new process

- The parameters for the new process are passed to the process calling `exec()`
  - These parameters are saved by the OS kernel
  - When the new process is created successfully, the parameters are placed at the “correct” regions of the new process’ address space
  - The new process “discovers” them during the regular course of execution

- Pop Quiz: How many times does `exec()` return?
- Who does it return to?
- What are the values that it returns?