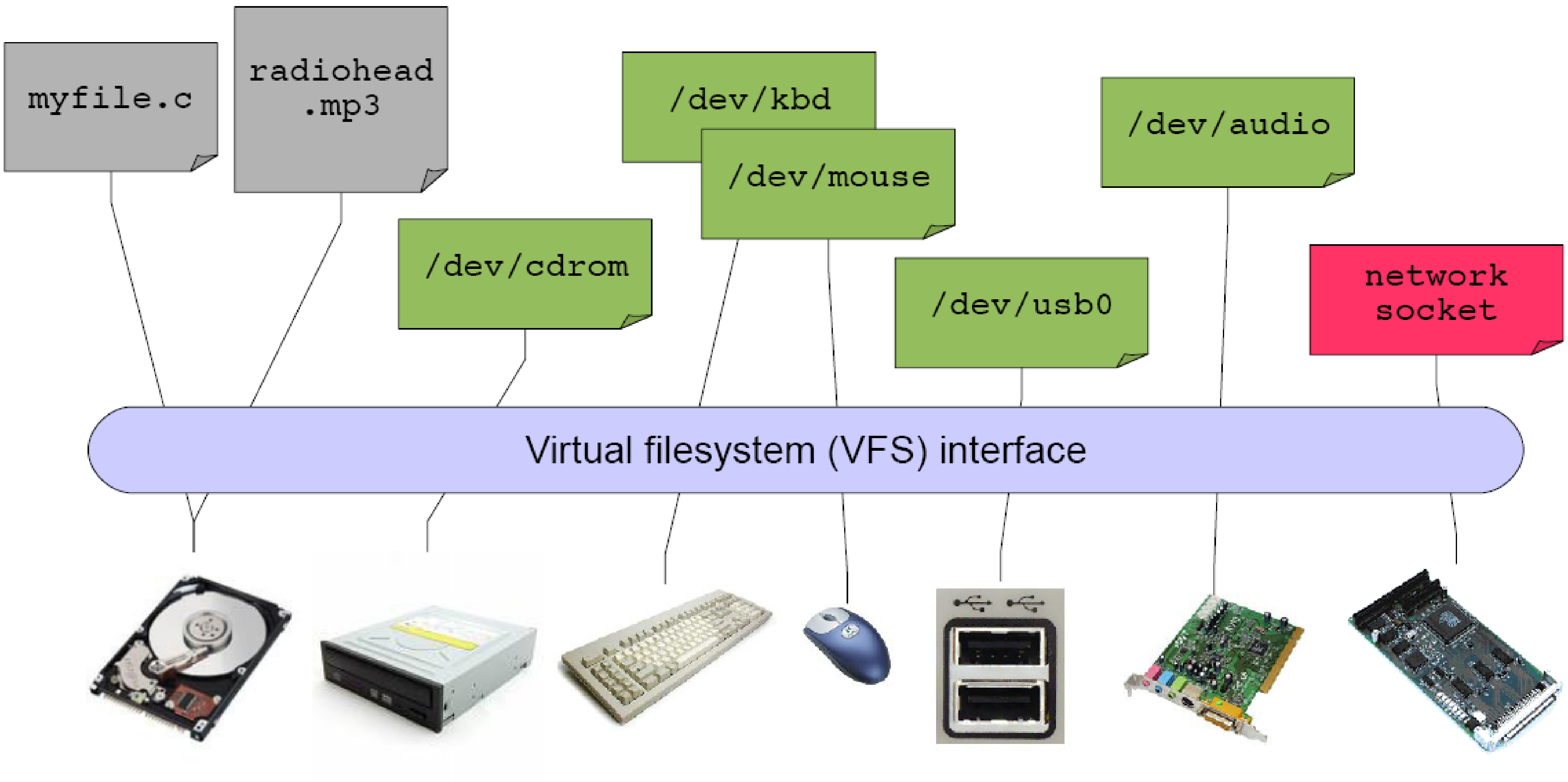
File Abstraction

Most of the following slides are adapted from slides of Randy Bryant of Carnegie Mellon Univ.

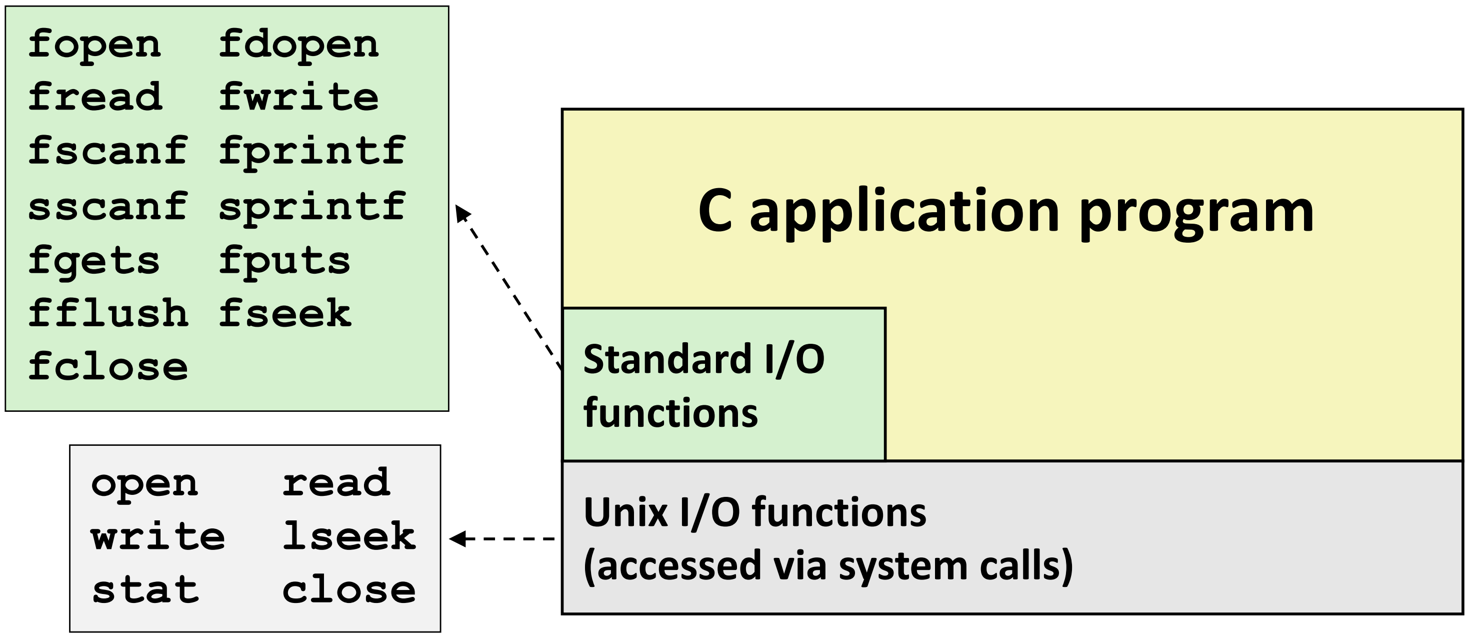
UNIX File Abstraction

- **In UNIX, the file is the basic abstraction used for I/O**
  - Used to access disks, CDs, DVDs, USB and serial devices, network



Unix I/O and C Standard I/O

- **C Standard**
  - Most useful for reading/writing files in applications
  - Provides buffering between program and actual files
- **Unix I/O**
  - Lower level
  - Required for system and network programming



Unix I/O Overview

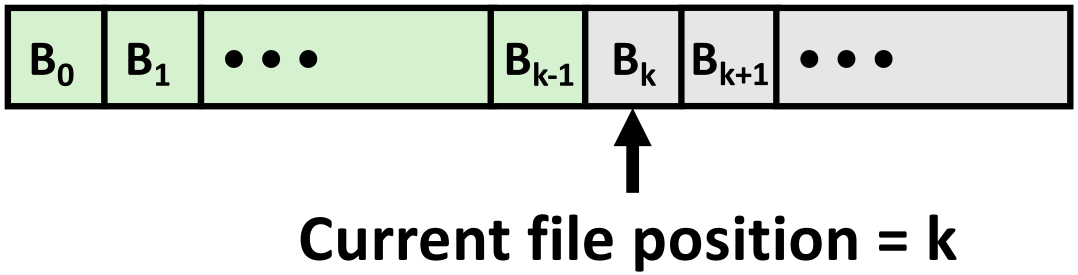
- **A Linux *file* is a sequence of *m* bytes:**
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$

- **Cool fact: All I/O devices are represented as files:**
  - **`/dev/sda2`** (**`/usr`** disk partition)
  - **`/dev/tty2`** (terminal)

- **Even the kernel is represented as a file:**
  - **`/boot/vmlinuz-3.13.0-55-generic`** (kernel image)
  - **`/proc`** (kernel data structures)

Unix I/O Overview

- **Elegant mapping of files to devices allows kernel to export simple interface called *Unix I/O:***
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the ***current file position*** (seek)
    - indicates next offset into file to read or write
    - **`lseek()`**



Opening Files

- **Opening a file informs the kernel that you are getting ready to access that file**

```
int fd;   /* file descriptor */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
   perror("open");
   exit(1);
}
```

- **Returns a small identifying integer *file descriptor***
  - **`fd == -1`** indicates that an error occurred

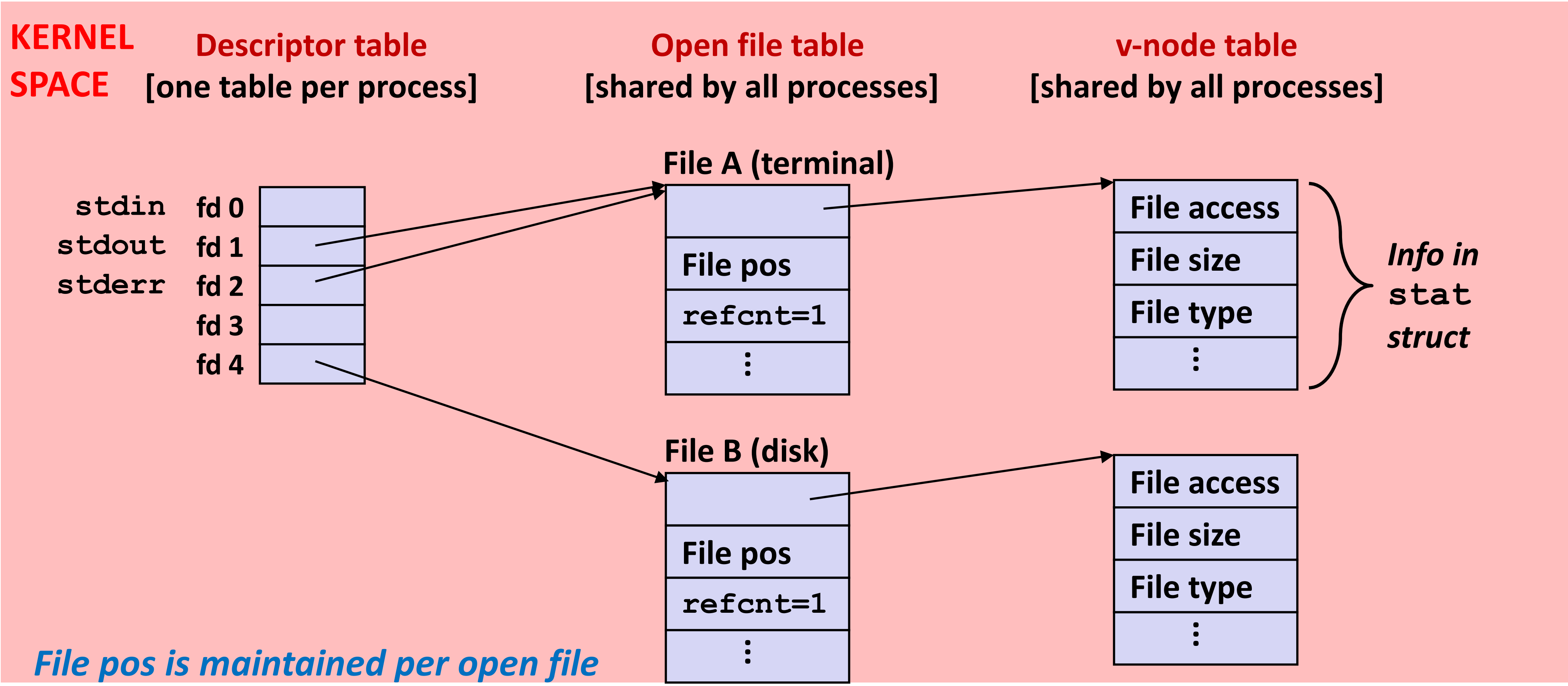
stdin, stdout, stderr

- **In UNIX, every process has three "special" files already open:**
  - standard input (stdin) – filehandle 0
  - standard output (stdout) – filehandle 1
  - standard error (stderr) – filehandle 2
- **By default, stdin and stdout are connected to the terminal device of the process.**
  - Originally, terminals were physically connected to the computer by a serial line
  - These days, we use "virtual terminals" using ssh

VT100 terminal

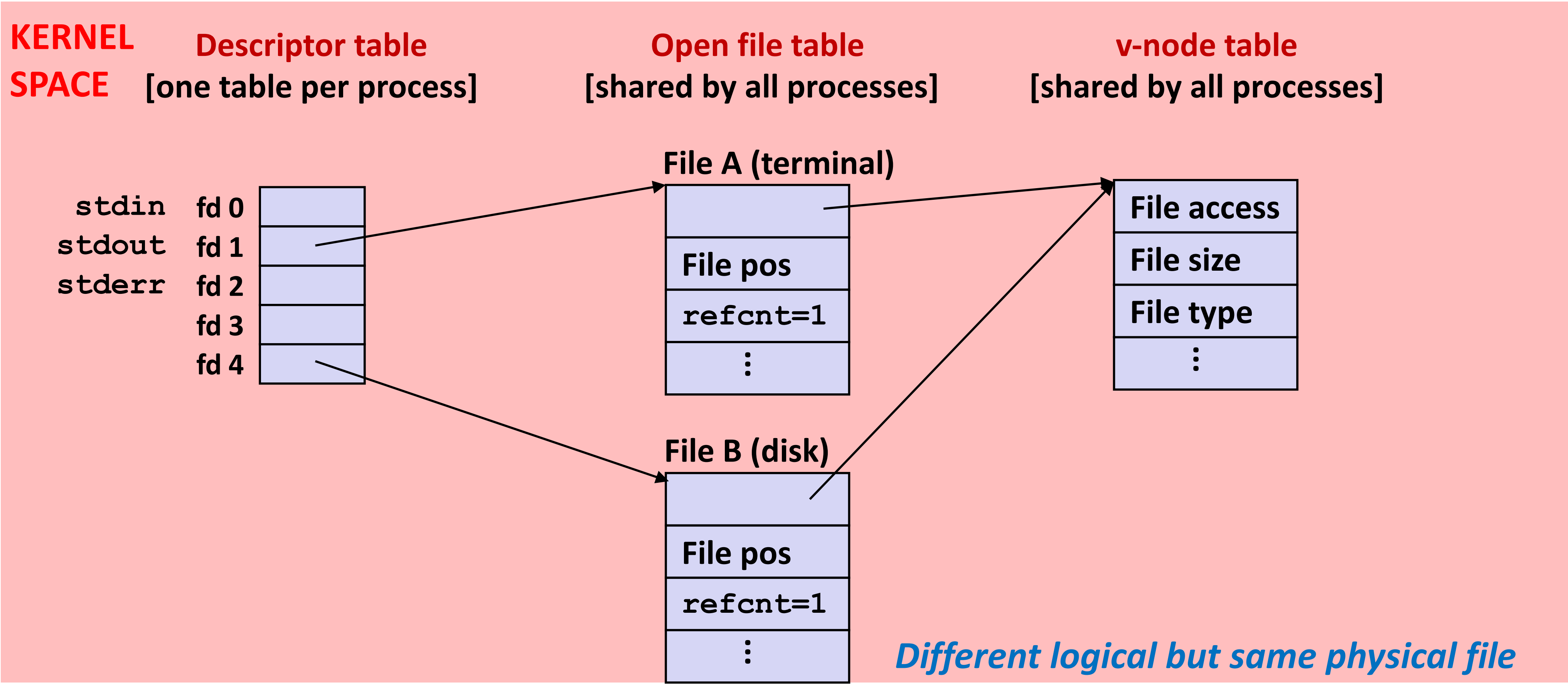
How the Unix Kernel Represents Open Files

- **Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) and 2 (stderr) points to terminal, and descriptor 4 points to file opened on the disk.**



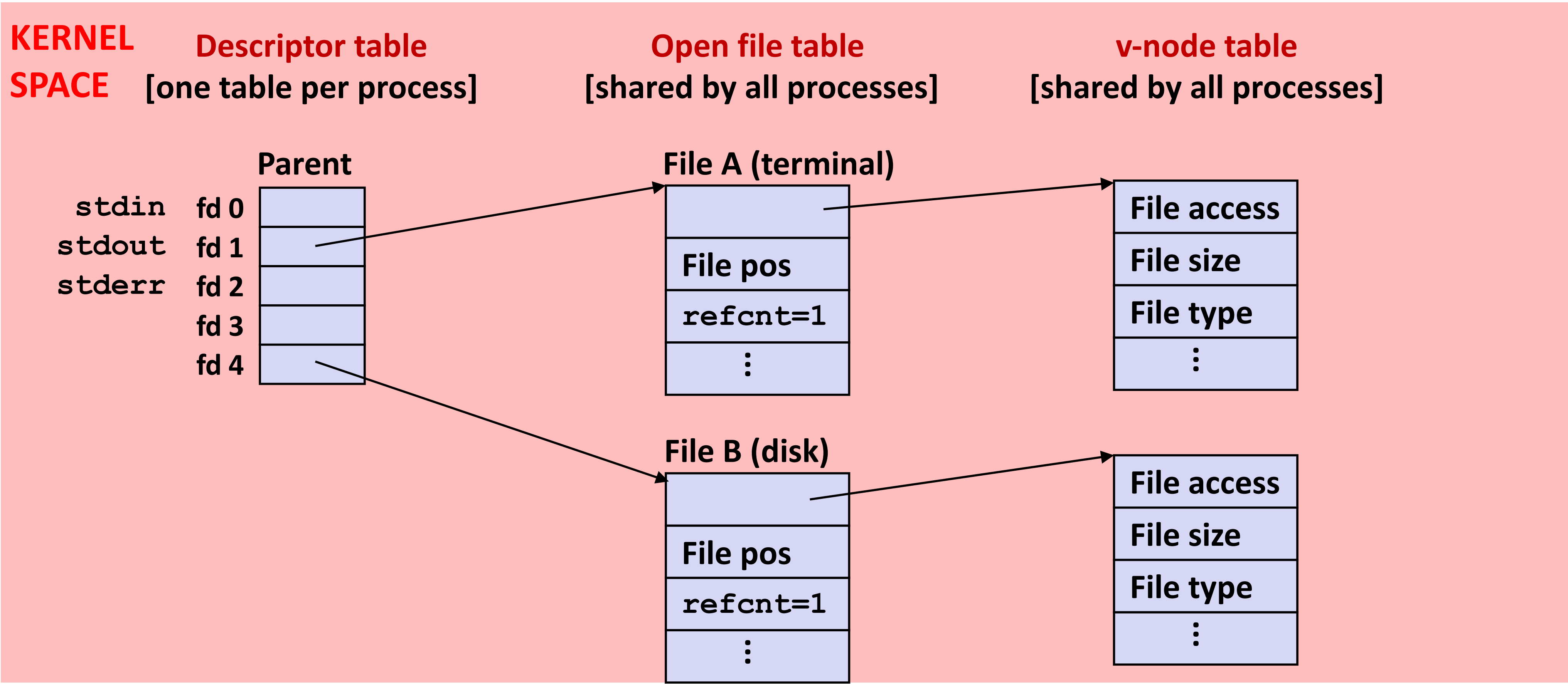
File Sharing

- **Two distinct descriptors sharing the same disk file through two distinct open file table entries**
  - E.g., Calling **`open`** twice with the same **`filename`** argument



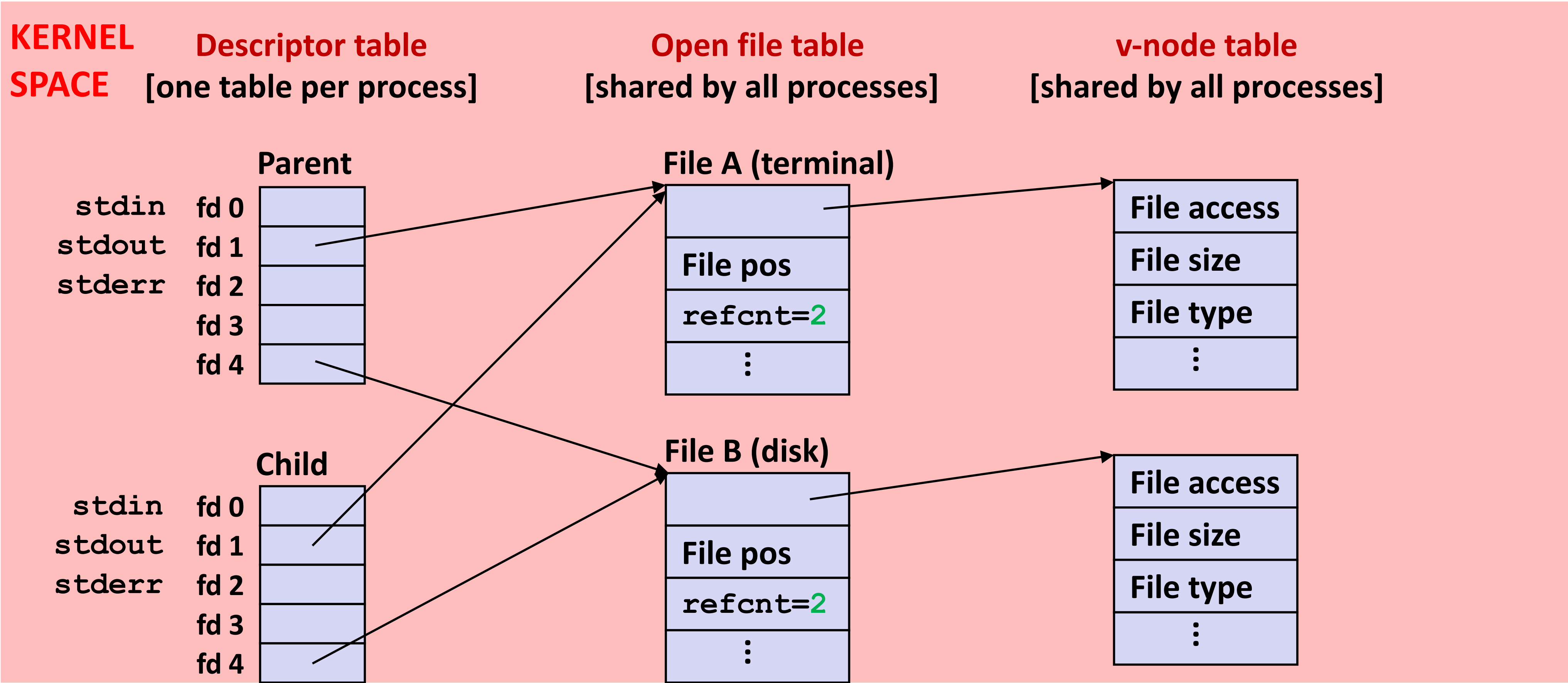
How Processes Share Files: `fork()`

- **A child process inherits its parent's open files**
  - Note: situation unchanged by **`exec()`** functions
- ***Before* fork() call:**



How Processes Share Files: `fork()`

- **A child process inherits its parent's open files**
- ***After* fork():**
  - Child's table same as parents, and +1 to each `refcnt`



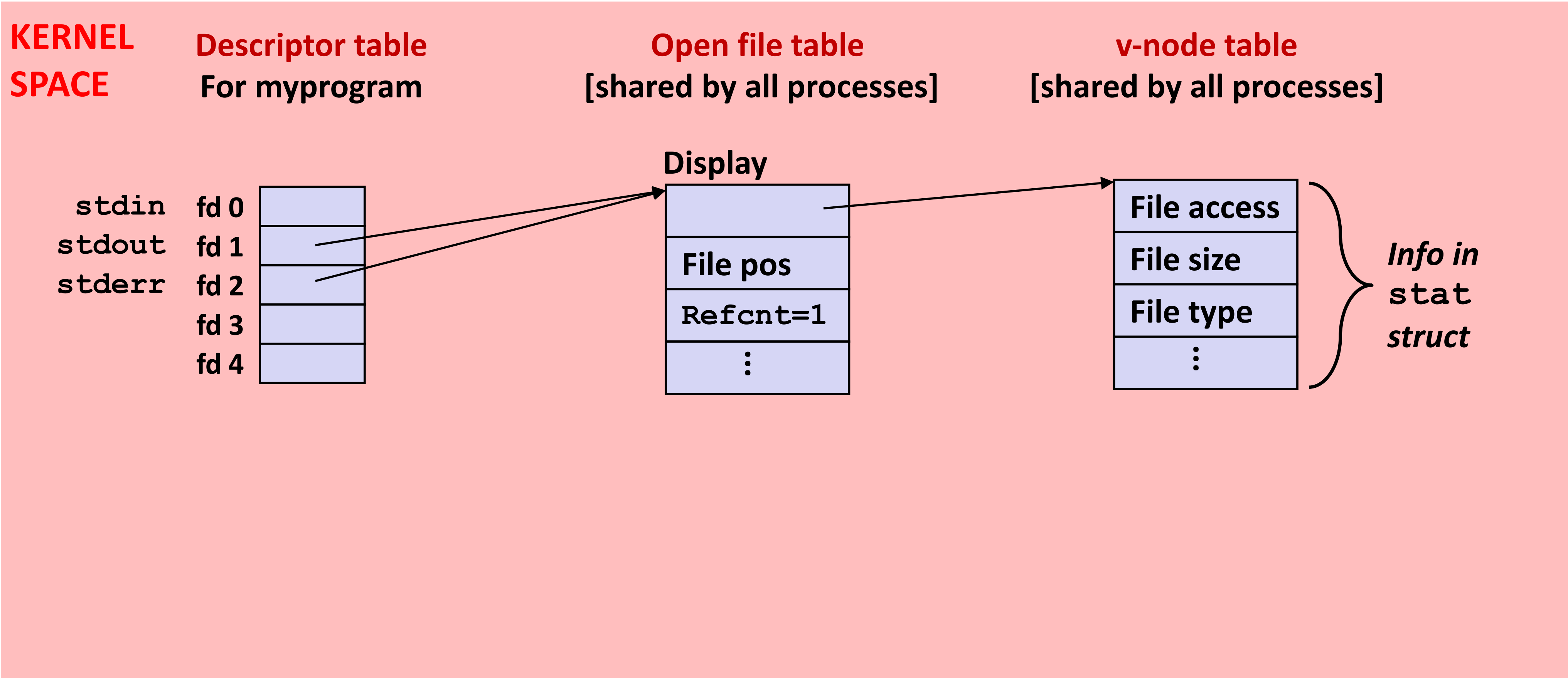
File is shared between processes

Shell redirection

- **The shell allows `stdin`, `stdout`, and `stderr` to be redirected (say, to or from a file).**
  - **`> ./myprogram > somefile.txt`**
    - Connects stdout of "myprogram" to somefile.txt
  - **`> ./myprogram < input.txt > somefile.txt`**
    - Connects stdin to input.txt and stdout to somefile.txt
  - **`> ./myprogram 2> errors.txt`**
    - Connects stderr to errors.txt
- **In this case, the shell simply opens the file, making sure the file handle is 0, 1, or 2, as appropriate.**
  - Problem: **`open()`** decides what the file handle number is.
  - How do we coerce the filehandle to be 0, 1, or 2?

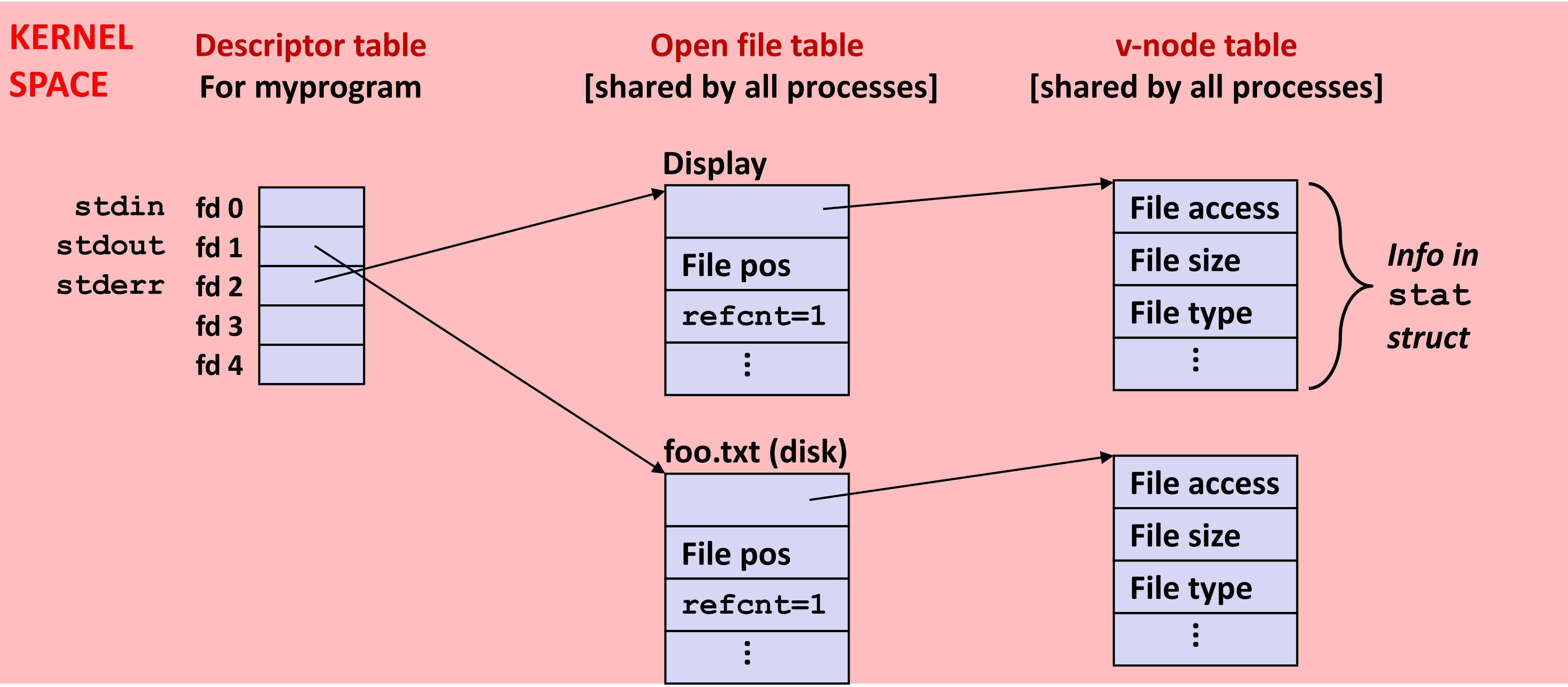
Initially

- **`stdout` prints to the display of the terminal as default.**



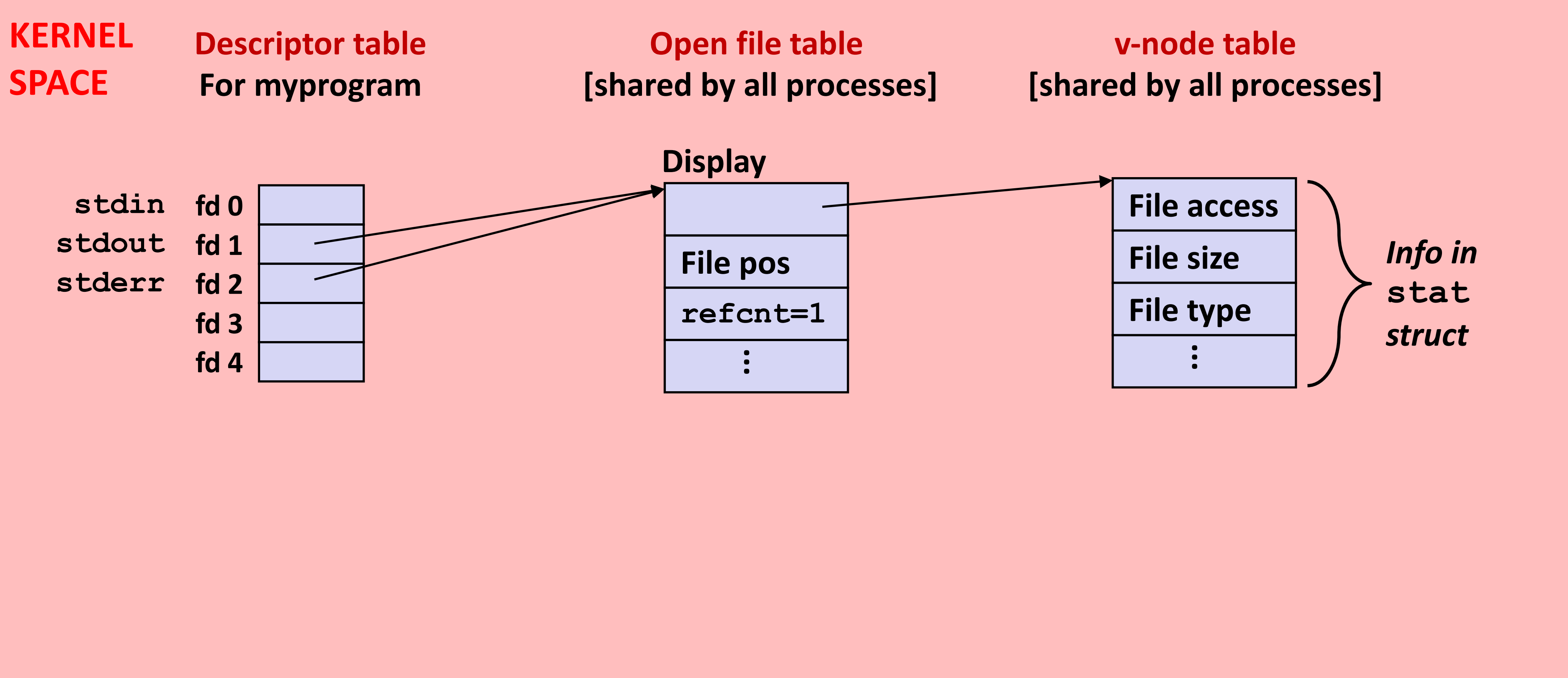
All we need to do is to point `stdout` to a file

- Question: But the Descriptor table is kernel space, and we cannot modify it directly.
- Need to use system calls!



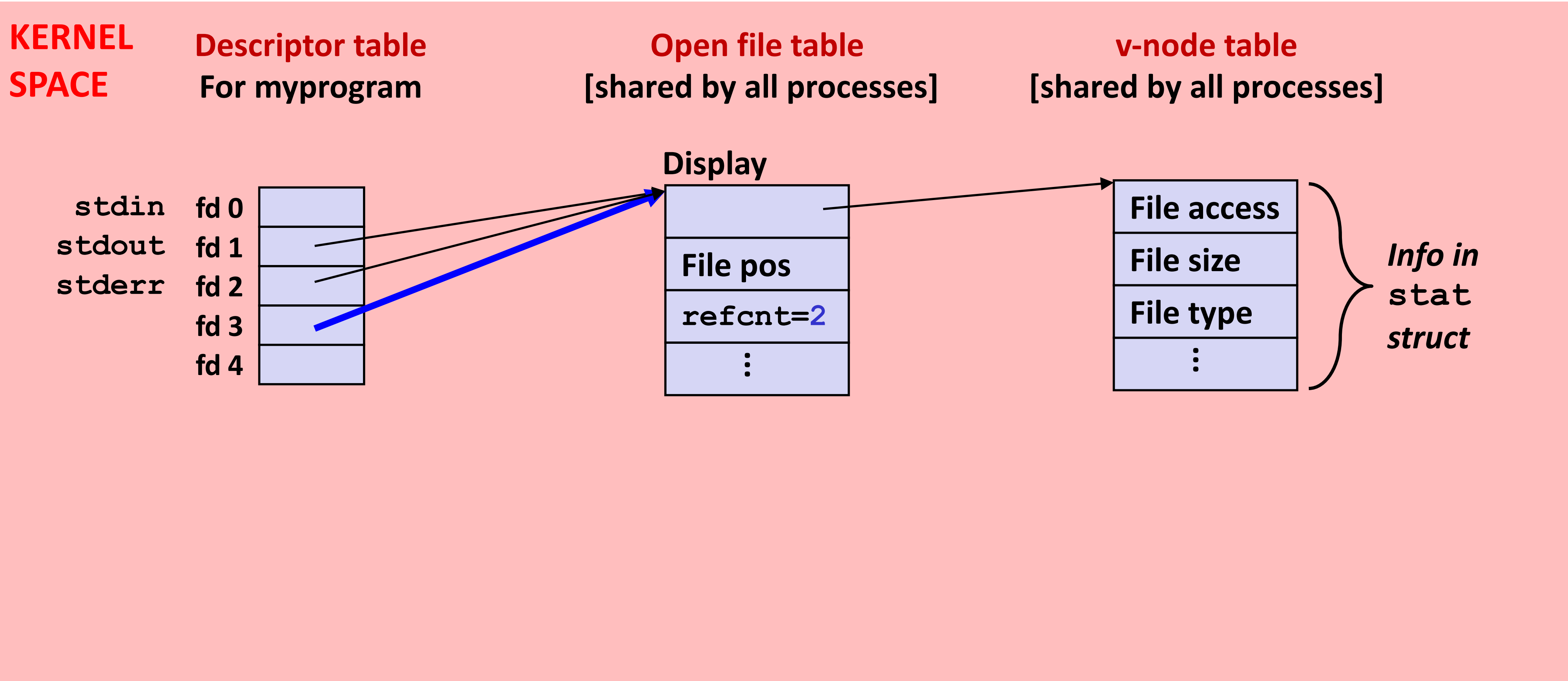
dup() : before

```
#include <unistd.h>
int dup(int filedes);
//dup() returns lowest available file descriptor, now
referring to whatever filedes refers to
newfd = dup(1); // newfd will be 3.
```



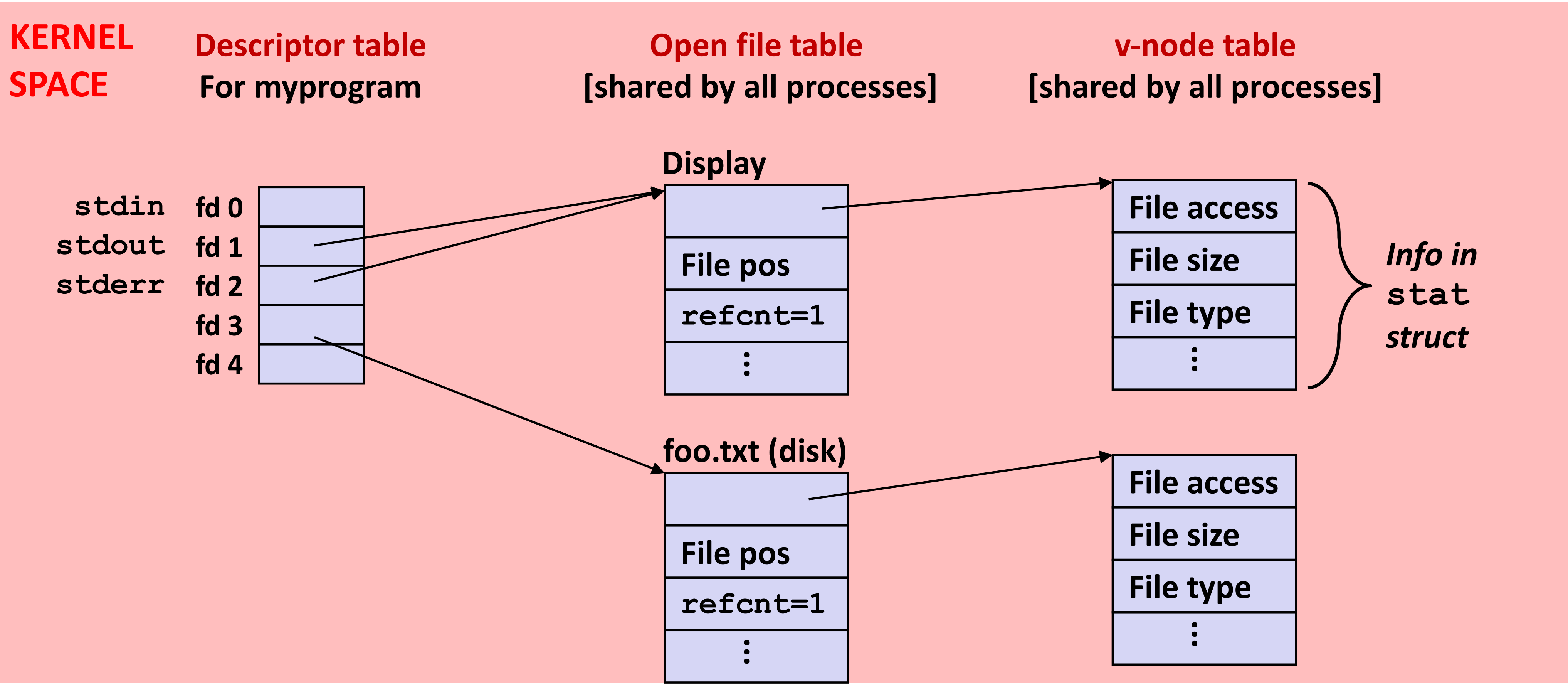
dup() : after

```
#include <unistd.h>
int dup(int filedes);
//dup() returns lowest available file descriptor, now
referring to whatever filedes refers to
newfd = dup(1); // newfd will be 3.
```



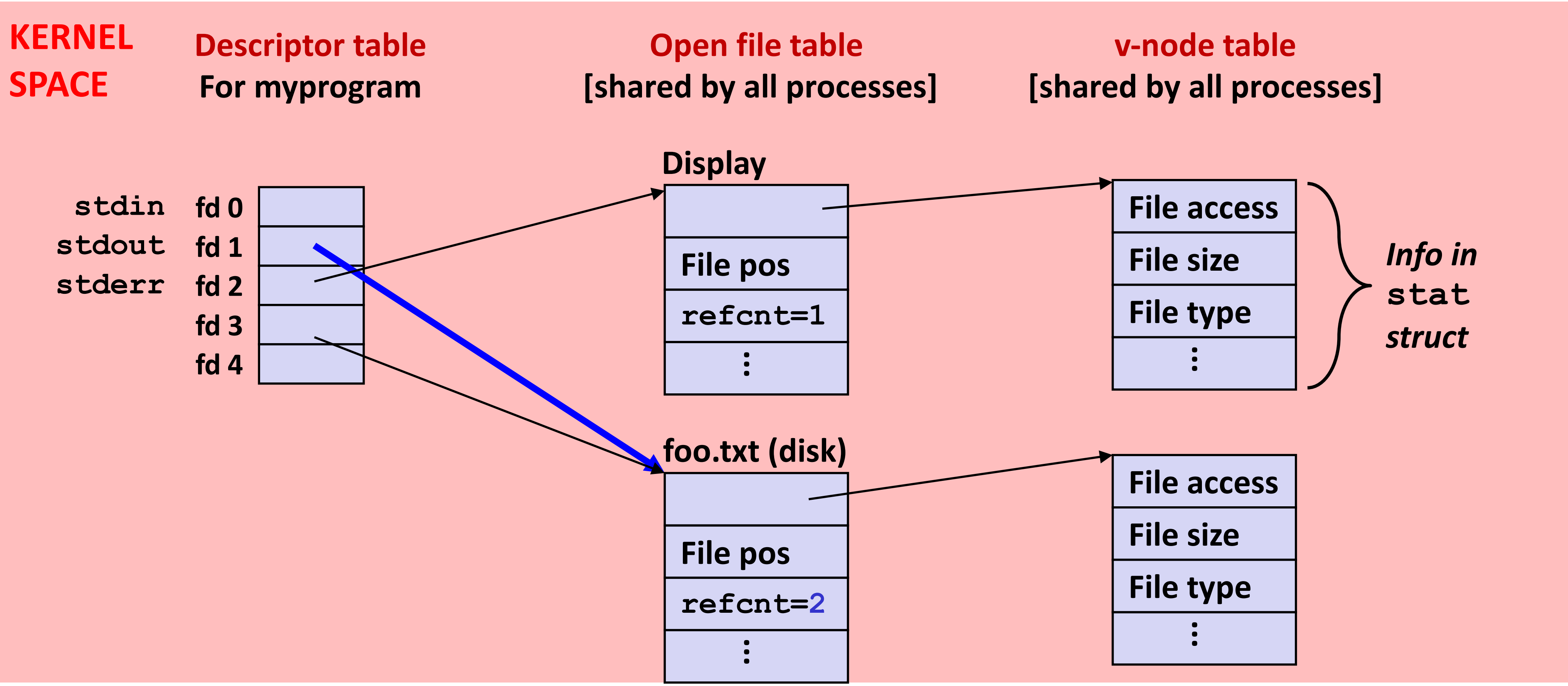
dup2() : before

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
//Copies descriptor table entry oldfd  to entry newfd
int foofd = open("foo.txt", O_WRONLY); //foofd becomes 3.
if (dup2(foofd, stdout)>0) printf("printing to foo.txt\n");
```



dup2() : after

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
//Copies descriptor table entry oldfd  to entry newfd
int foofd = open("foo.txt", O_WRONLY); //foofd becomes 3.
if (dup2(foofd, stdout)>0) printf("printing to foo.txt\n");
```



dup() and dup2() pseudocode

- **dup(fd) returns lowest available file descriptor, now referring to whatever oldfd refers to refers to.**

```
//Descriptor table
void *DT[maxFd];

int dup(int oldfd){
   //get the lowest available
   //file descriptor
   newfd = lowestFd(DT);
   DT(newfd)=DT(oldfd);
   return(newfd);
}
```

- **dup2(oldfd,newfd) copies descriptor table entry oldfd to entry newfd.**

```
//Descriptor table
void *DT[maxFd];

int dup2(int oldfd, int newfd){
    DP[newfd]=DP[oldfd];
    return(newfd);
}
```

- If **oldfd** is not a valid file descriptor, then the call fails, and **newfd** is not closed.
- If **oldfd** is a valid file descriptor, and **newfd** has the same value as **oldfd**, then **dup2()** does nothing, and returns **newfd**.

I/O and Redirection Example

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = open(fname, O_RDONLY, 0);
    fd2 = open(fname, O_RDONLY, 0);
    fd3 = open(fname, O_RDONLY, 0);
    dup2(fd2, fd3);
    read(fd1, &c1, 1);
    read(fd2, &c2, 1);
    read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

ffiles1.c

- **What would this program print for file containing "abcde"?**

I/O and Redirection Example

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1);
    Read(fd2, &c2, 1);
    Read(fd3, &c3, 1);
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
```

ffiles1.c

`c1 = a, c2 = a, c3 = b`

`dup2(oldfd, newfd)`

- What would this program print for file containing "abcde"?

Master Class: Process Control and I/O

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}
```

ffiles2.c

- What would this program print for file containing “abcde”?

Master Class: Process Control and I/O

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1);
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}
```

ffiles2.c

```
Child: c1 = a, c2 = b
Parent: c1 = a, c2 = c
```

```
Parent: c1 = a, c2 = b
Child: c1 = a, c2 = c
```

Bonus: Which way does it go?

- **What would this program print for file containing "abcde"?**

For Further Information

- **The Unix bible:**
  - W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 2nd Edition, Addison Wesley, 2005
    - Updated from Stevens' 1993 book

- **Stevens is arguably the best technical writer ever.**
  - Produced authoritative works in:
    - Unix programming
    - TCP/IP (the protocol that makes the Internet work)
    - Unix network programming
    - Unix IPC programming

https://github.com/shihyu/Linux_Programming/tree/master/books

Bonus material

- The following slides are provided as extra and is not part of the course coverage.
- Enjoy!

System Call Error Handling

- **On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.**
- **Hard and fast rule:**
  - You must check the return status of every system-level function
  - Only exception is the handful of functions that return **`void`**
- **Example:**

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(-1);
}
```

Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(-1);
}
```

```
 if ((pid = fork()) < 0)
    unix_error("fork error");
```

Note: csapp.c exits with 0.

Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

```
pid = Fork();
```

- NOT what you generally want to do in a real application

Standard I/O Streams

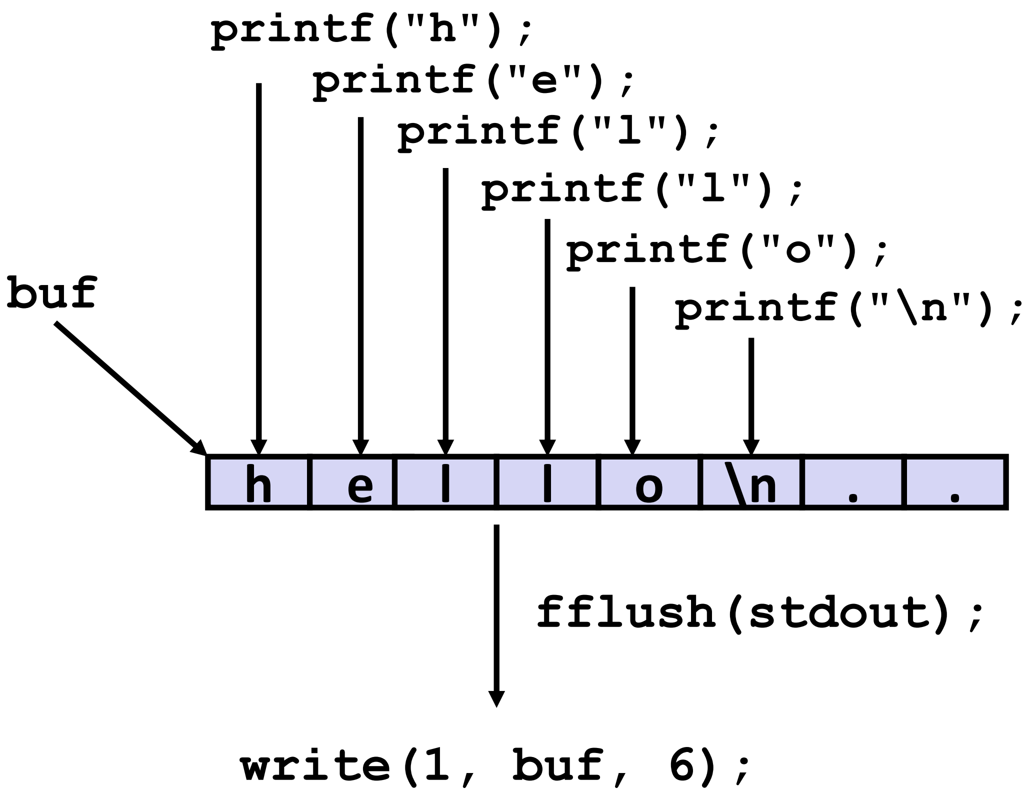
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory.
  -
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - **`stdin`** (standard input)
  - **`stdout`** (standard output)
  - **`stderr`** (standard error)

```
#include <stdio.h>
extern FILE *stdin;  /* standard input  (descriptor 0) */
extern FILE *stdout; /* standard output (descriptor 1) */
extern FILE *stderr; /* standard error  (descriptor 2) */

int main() {
    fprintf(stdout, "Hello, world\n");
}
```

Buffering in Standard I/O

- Standard I/O functions use buffered I/O



- Buffer flushed to output `fd` on "`\n`" or `fflush()` call

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Unix `strace` program:

```
#include <stdio.h>

int main()
{
    printf("h");
    printf("e");
    printf("l");
    printf("l");
    printf("o");
    printf("\n");
    fflush(stdout);
    exit(0);
}
```

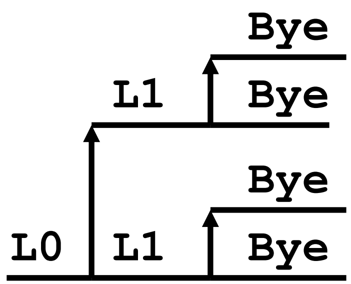
```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6...)                    = 6
...
_exit(0)                                      = ?
```

`strace`: a debugging tool in Linux.
When you start a program using **`strace`**, it prints a list of system calls made by the program.

Fork Example #2 (Earlier Lecture)

- **Key Points**
  - Both parent and child can continue forking

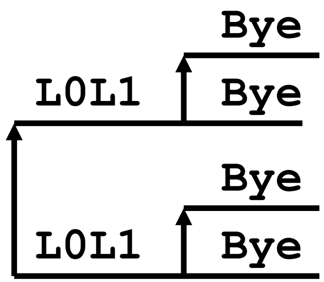
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #2 (modified)

- **Removed the "\n" from the first printf**
  - As a result, "L0" gets printed twice

```
void fork2a()
{
    printf("L0");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Repeated Slide: Reading Files

- **Reading a file copies bytes from the current file position to memory, and then updates file position**

```
char buf[512];
int fd;         /* file descriptor */
int nbytes;     /* number of bytes read */

/* Open file fd ...  */
/* Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
   perror("read");
   exit(1);
}
```

- **Returns number of bytes read from file `fd` into `buf`**
  - Return type `ssize_t` is signed integer
  - `nbytes < 0` indicates that an error occurred
  - ***short counts*** (`nbytes < sizeof(buf)`) are possible and are not errors!

Dealing with Short Counts

- **Short counts can occur in these situations:**
  - Encountering (end-of-file) EOF on reads
  - Reading text lines from a terminal
  - Reading and writing network sockets or Unix pipes

- **Short counts never occur in these situations:**
  - Reading from disk files (except for EOF)
  - Writing to disk files