Reference:

Advanced Programming in the UNIX Environment, Third Edition, W. Richard Stevens and Stephen A. Rago, Addison-Wesley Professional Computing Series, 2013. Chapter 3

Computer Systems: A Programmer's Perspective, Randal E. Bryant and David R. O'Hallaron, Prentice Hall, 3rd edition, 2016, Chapter 10

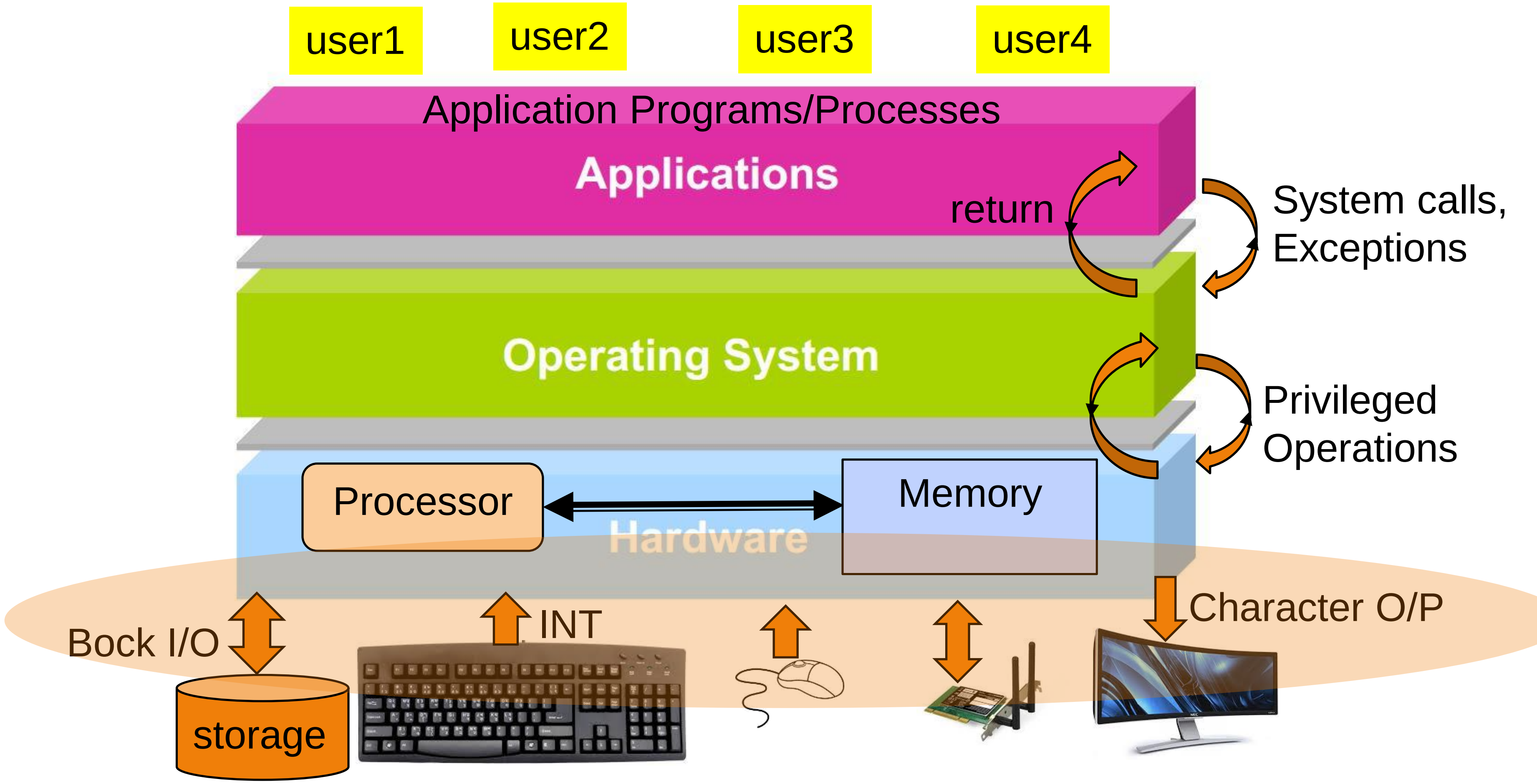
Acknowledgment: A number of examples in a box with this color are taken from CSAPP book cited above

WEEK 5: UNIX IO

Aakash Tyagi
CSCE 313 Spring 2017

Unix IO

- How do we communicate with IO Devices?



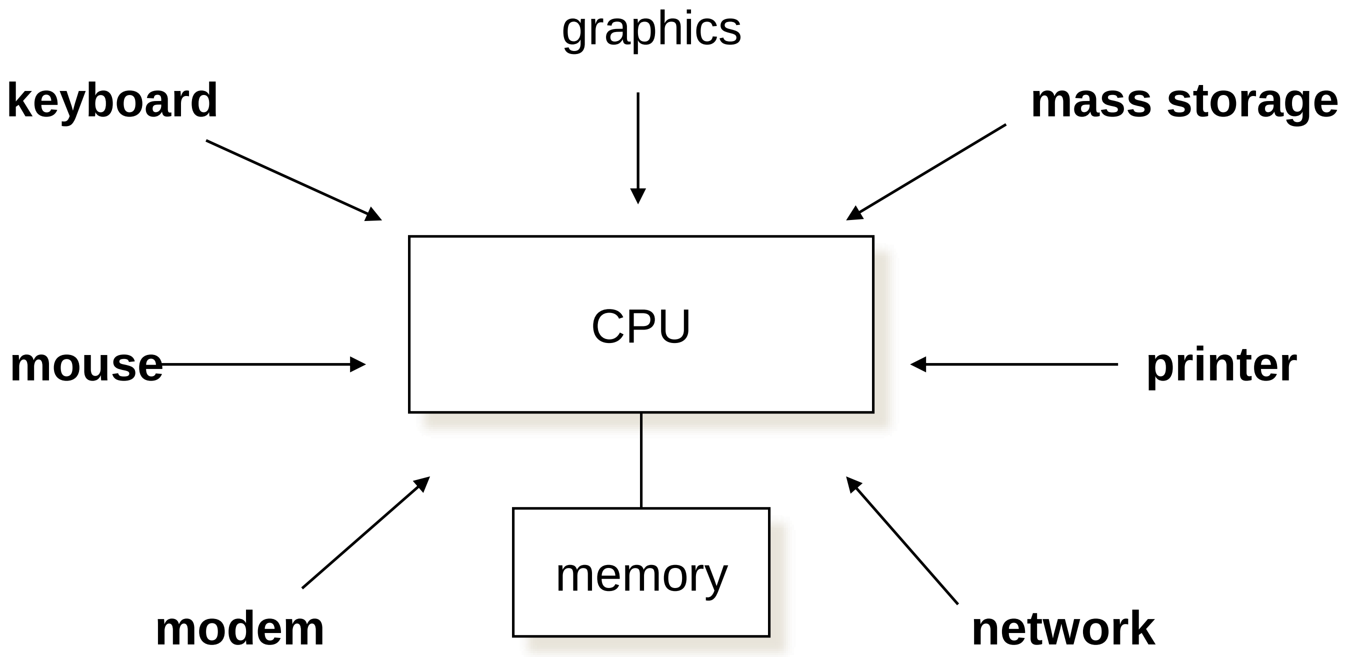
Motivation

- I/O is the process of copying data between main memory and external devices (disk drives, terminals, networks)
- Language run-time systems provide higher-level facilities for performing I/O (e.g. printf, scanf)
  - On Unix systems, these higher-level I/O functions are implemented using **System-Level Unix I/O functions** provided by the Kernel

Unix Files

- A Unix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$
- All I/O devices (networks, disks, terminals) are represented as files:
  - **`/dev/sda2`** (**`/usr`** disk partition; a is the order and #2 is the partition)
  - **`/dev/tty2`** (terminal)
- Even the kernel is represented as a file:
  - **`/dev/kmem`** (kernel memory image)
  - **`/proc`** (kernel data structures)
- All I/O is performed by reading and writing the appropriate files

Files are not always “Files”: I/O Devices



Unix File Types

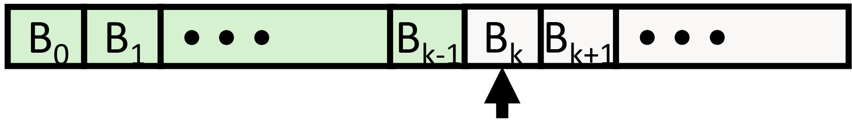
- **Regular file**
  - File containing user/app data (binary, text, whatever)
- **Directory file**
  - A file that contains the names and locations of other files
- **Character special and block special files**
  - Terminals (character special) and disks (block special)
- **FIFO** (named pipe)
  - A file type used for inter-process communication
- **Socket**
  - A file type used for network communication between processes

Unix I/O

- Key Features
  - Elegant mapping of files to devices allows kernel a simple interface called Unix I/O
  - Important idea: All input and output is handled in a consistent and uniform way

Unix I/O

- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - **`open()`** and **`close()`**
  - Reading and writing a file
    - **`read()`** and **`write()`**
  - Changing the *current file position* (seek)
    - Kernel maintains a file position (initially 0) for each open file
    - indicates next (byte) offset into file to read or write
    - **`lseek()`**



Current file position = k

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
- Kernel keeps track of all information about the open file. The application only keeps track of the descriptor

Opening Files

- Each process created by a Unix shell begins life with three open files associated with a terminal:
  - 0: standard input
  - 1: standard output
  - 2: standard error

Closing Files

- Closing a file informs the kernel that you are finished accessing that file

```
int fd;      /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
   perror("close");
   exit(1);
}
```

- The kernel responds by freeing the data structures associated with the file and restoring the descriptor to the available pool of available descriptors
- When a process terminates for any reason, the kernel closes all open files and frees their memory resources

Example

- What is the output of the following program?

```
int main ()
{
        int fd1, fd2;
        fd1 = open("foo.txt", O_RDONLY, 0);
        close(fd1);
        fd2 = open("baz.txt", O_RDONLY, 0);
        printf("fd2=%d\n", fd2);
        exit(0);
}
```

- Unix processes begin life with open descriptors assigned to stdin (fd=0), stdout (fd=1), and stderr (fd=2). The open function always returns the lowest unopened descriptor so the output will be “fd2=3”

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);
}
```

Reading Files

```
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 (eg. EOF, reading text from terminal etc.)

Writing Files

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

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

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

Writing Files

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

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

Returns number of bytes written from `buf` to file `fd`

- **`nbytes < 0`** indicates that an error occurred
- As with reads, short counts are possible and are not errors!

Simple Unix I/O example

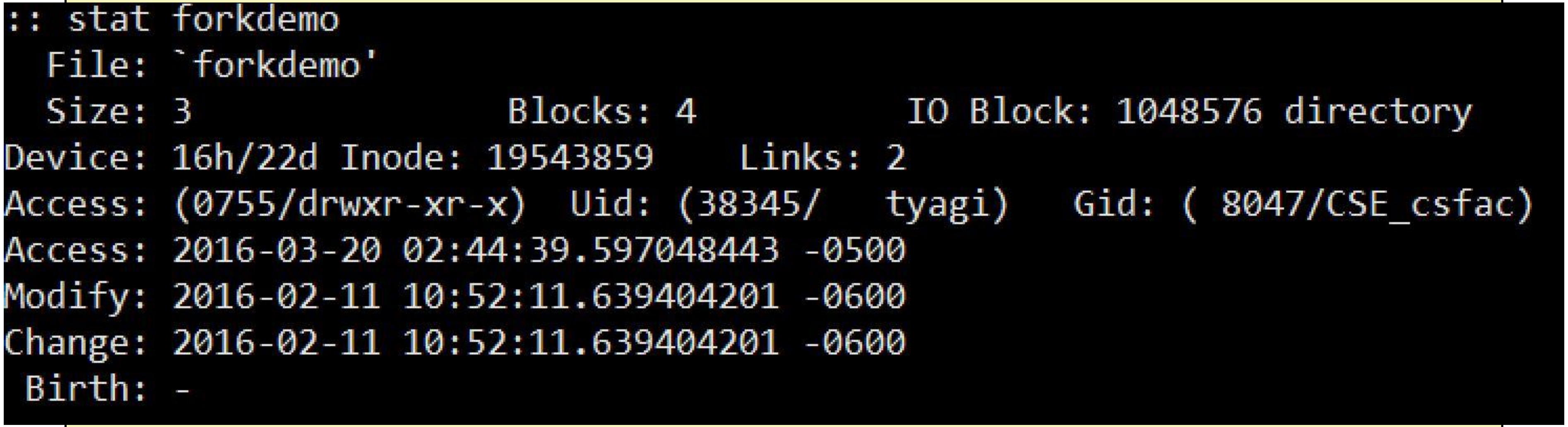
- Copying standard in to standard out, one byte at a time

```
int main(void)
{
    char c;

    while(read(STDIN_FILENO, &c, 1) != 0)
      write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

File Metadata

- *Metadata* is data about data, in this case file data
- Per-file metadata maintained by kernel
  - accessed by users with the **`stat`** function

```
/* Metadata returned by the stat function */
struct stat {
```



```
:: stat forkdemo
  File: `forkdemo'
  Size: 3               Blocks: 4          IO Block: 1048576 directory
Device: 16h/22d Inode: 19543859    Links: 2
Access: (0755/drwxr-xr-x)  Uid: (38345/   tyagi)   Gid: ( 8047/CSE_csfac)
Access: 2016-03-20 02:44:39.597048443 -0500
Modify: 2016-02-11 10:52:11.639404201 -0600
Change: 2016-02-11 10:52:11.639404201 -0600
 Birth: -
```

```
    time_t          st_atime;     /* time of last access */
    time_t          st_mtime;     /* time of last modification */
    time_t          st_ctime;     /* time of last change */
};
```

Accessing Directories

- Reading Directory Entries
  - **`dirent`** structure contains information about a directory entry
  - DIR structure contains information about directory while stepping through its entries

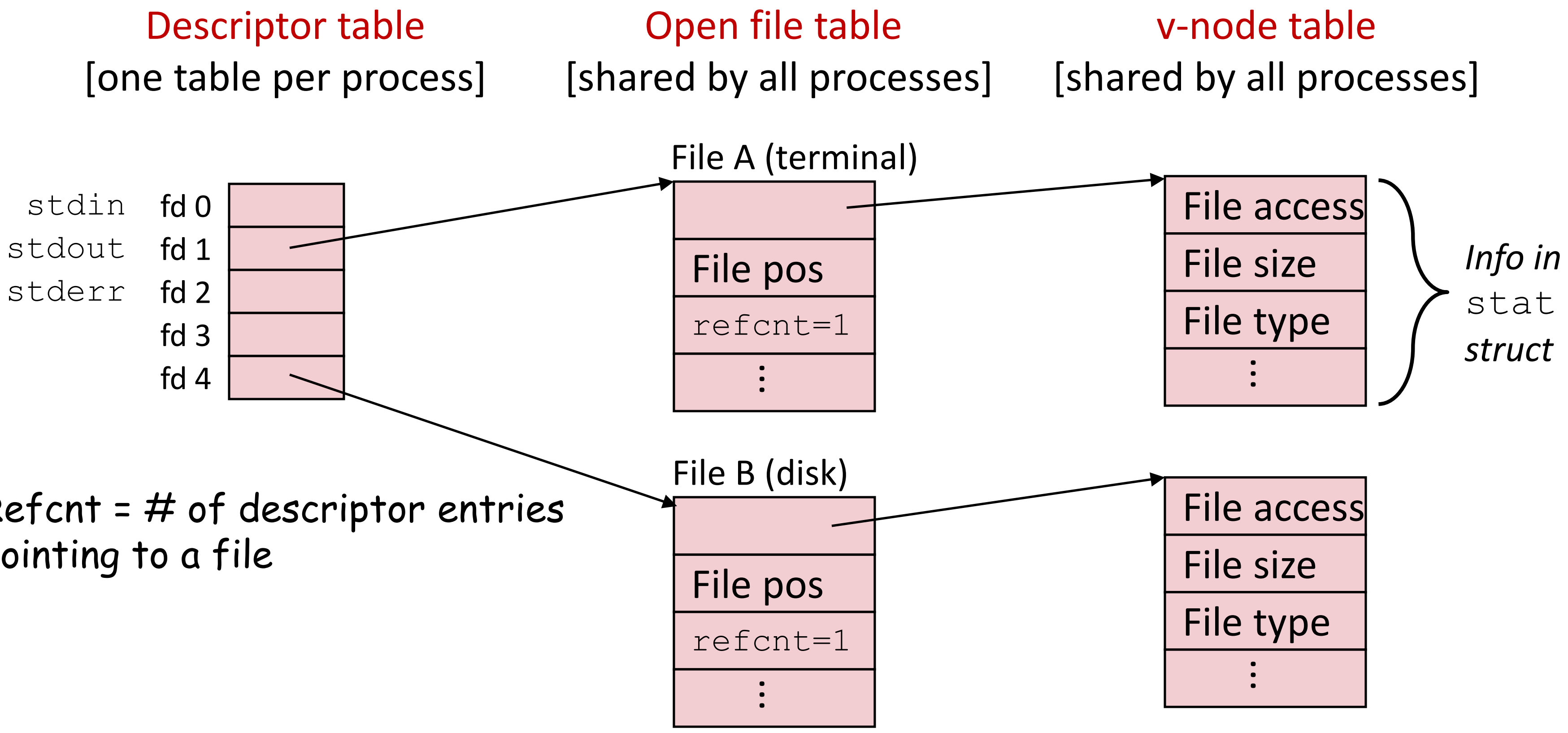
```
#include <sys/types.h>
#include <dirent.h>
{
  DIR *directory;
  struct dirent *de;
  ...
  if (!(directory = opendir(dir_name)))
      error("Failed to open directory");
  ...
  while (0 != (de = readdir(directory))) {
      printf("Found file: %s\n", de->d_name);
  }
  ...
  closedir(directory);
}
```

File Representation to User

- The kernel represents open files using three related data structures
  - **Descriptor Table**
    - Each process has its own separate descriptor table whose entries are indexed by the process's open file descriptors. Each open descriptor entry points to an entry in the file table.
  - **v-node Table**
    - Each entry (per file) contains information in the stat structure
  - **File Table**
    - Shared by all processes. Each entry consists of the current file position, reference count of # of descriptor entries that point to it, and a ptr to the entry in the v-node table, along with status flags

How the Unix Kernel Represents Open Files

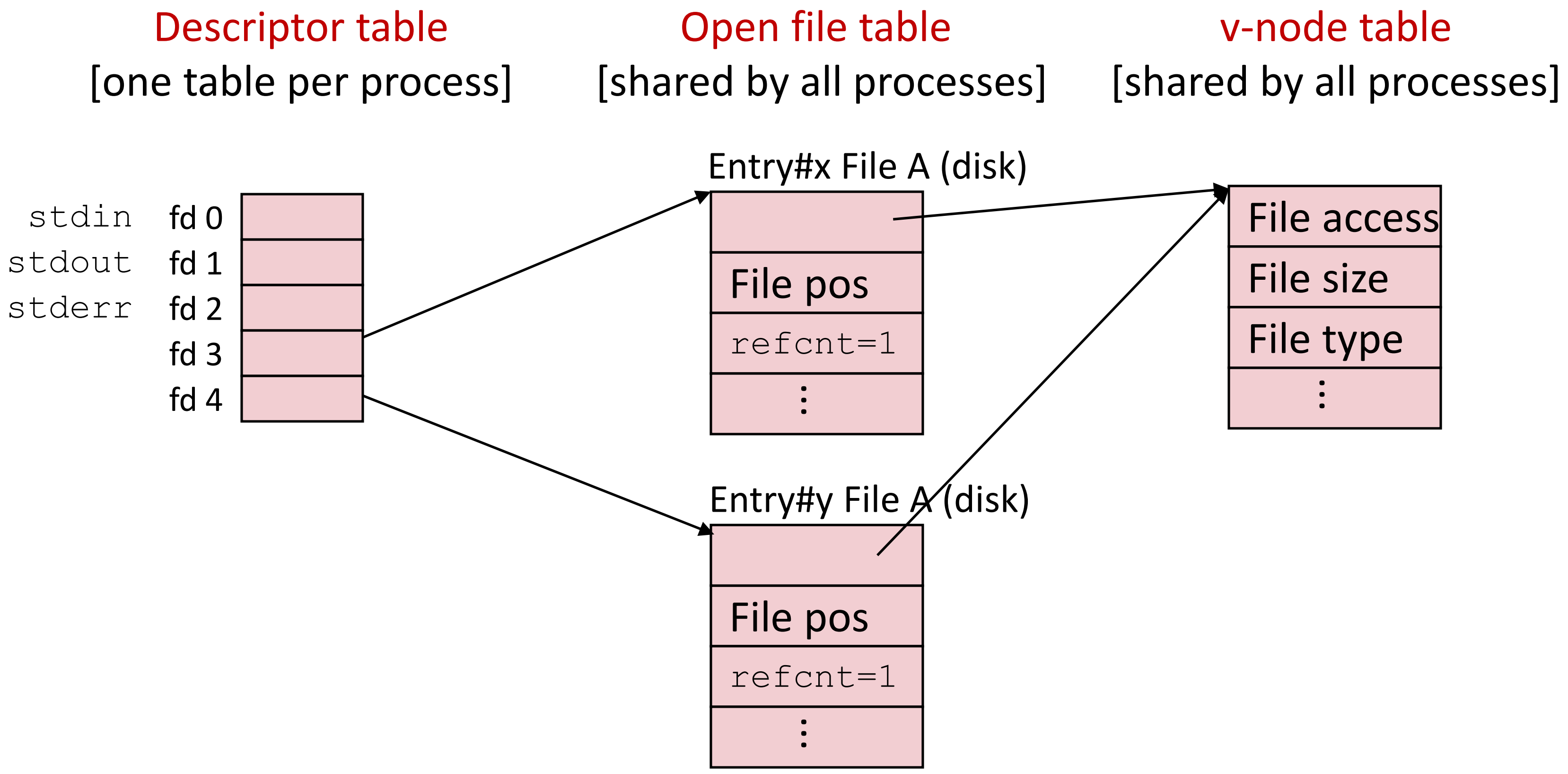
- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



Refcnt = # of descriptor entries pointing to a file

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



Example

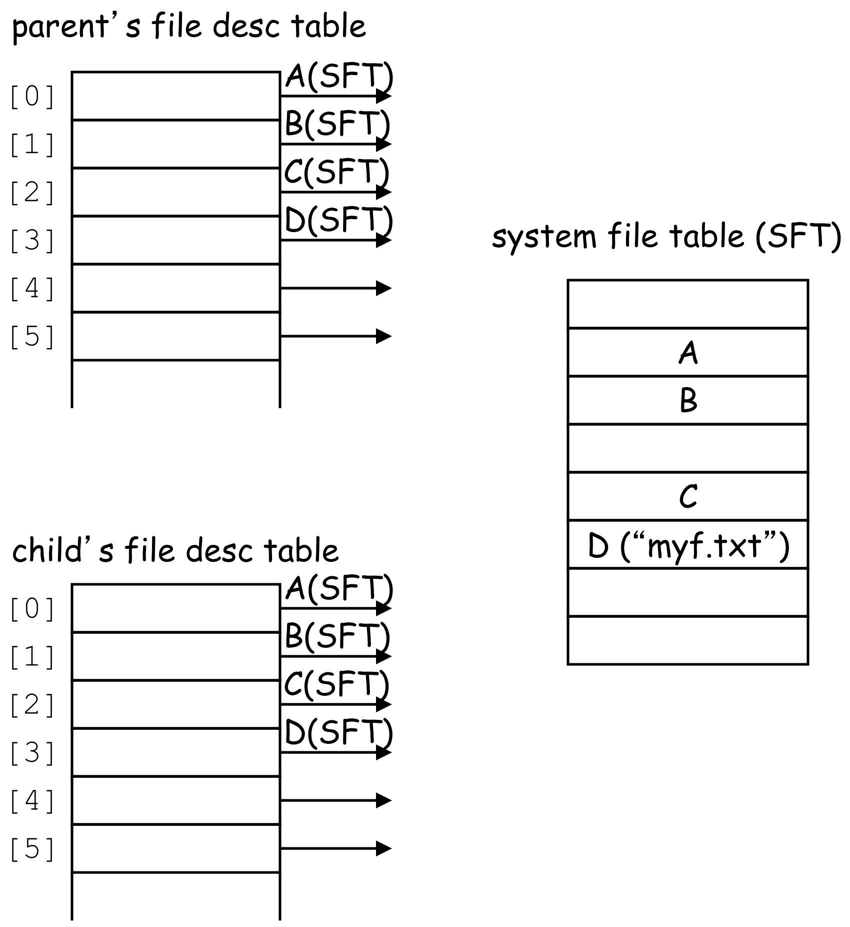
- Suppose the disk file foobar.txt consists of the six ASCII characters “foobar”. Then what is the output of the following program:

```
int main ()
{
        int fd1, fd2;
        char c;
        fd1 = open(“foobar.txt”, O_RDONLY, 0);
        fd2 = open(“foobar.txt”, O_RDONLY, 0);
        read(fd1, &c, 1);
        read(fd2, &c, 1);
        printf(“c=%c\n”, c);
        exit(0);
}
```

- *Answer: The descriptors fd1 and fd2 each have their own open file table entry so each descriptor has its own file position for foobar.txt. Thus the read from fd2 reads the first byte of foobar.txt and the output is “c=f” and NOT “c=o”*

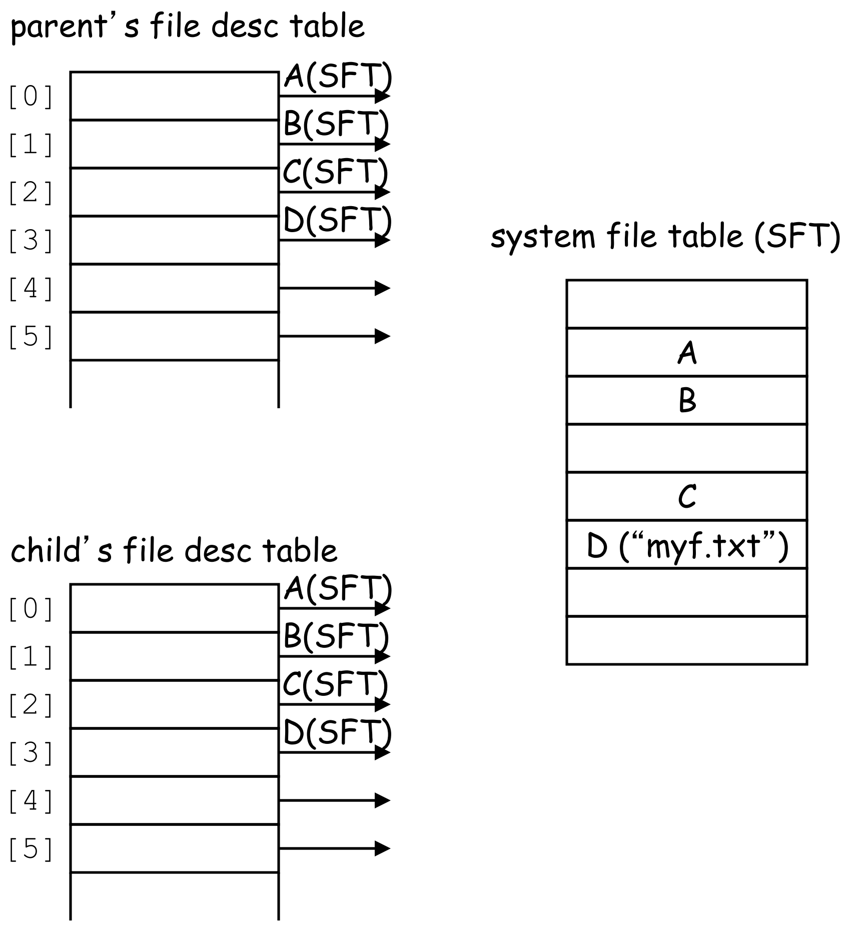
File Descriptors and `fork()`

- With `fork()`, child inherits content of parent's address space, including most of parent's state:
  - scheduling parameters
  - file descriptor table
  - signal state
  - environment
  - etc.



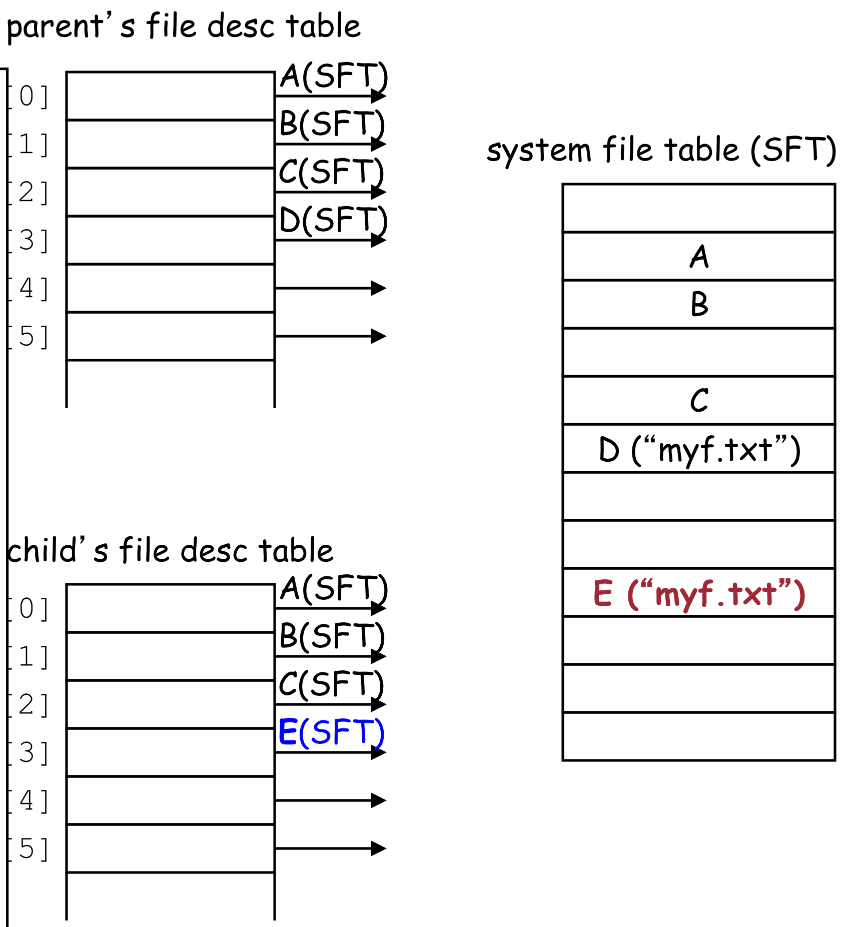
File Descriptors and `fork()` (II)

```
int main(void) {
  char c = '!';
  int myfd;

  myfd = open('myf.txt', O_RDONLY);

  fork();

  read(myfd, &c, 1);

  printf('Process %ld got %c\n',
             (long)getpid(), c);

  return 0;
}
```



File Descriptors and `fork()` (III)

```
int main(void) {
   char c = '!';
   int myfd;

   fork();

   myfd = open('myf.txt', O_RDONLY);

   read(myfd, &c, 1);

   printf('Process %ld got %c\n',
                (long)getpid(), c);

   return 0;
}
```



Example

- Suppose the disk file foobar.txt consists of the six ASCII characters “foobar”. Then what is the output of the following program:

```
int main ()
{
        int fd;
        char c;
        fd = open(“foobar.txt”, O_RDONLY, 0);
        if (fork() == 0) {
                read(fd, &c, 1);
                exit(0);
        }
        wait(NULL)
        read(fd, &c, 1);
        printf(“c=%c\n”, c);
        exit(0);
}
```

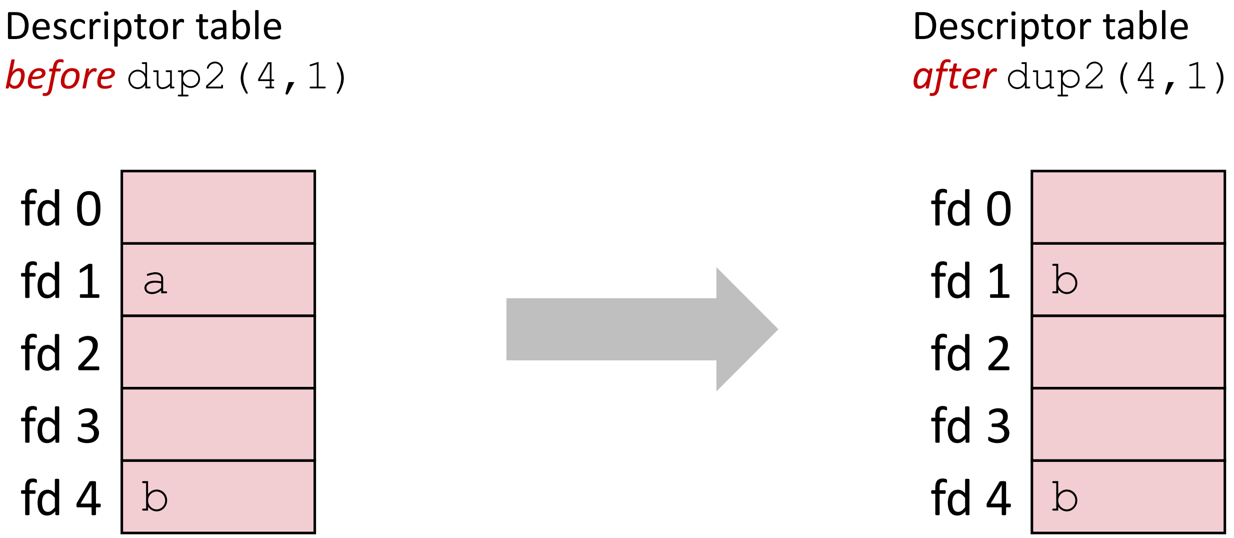
- *Answer: The child inherits the parent’s descriptor table and all processes share the same file table. Thus the descriptor fd in both the parent and child points to the same open file table entry. When the child reads the first byte of the file, the file position increments by 1. Thus the parent reads the second byte and output is “c=o”*

I/O Redirection

- Question: How does a shell implement I/O redirection?

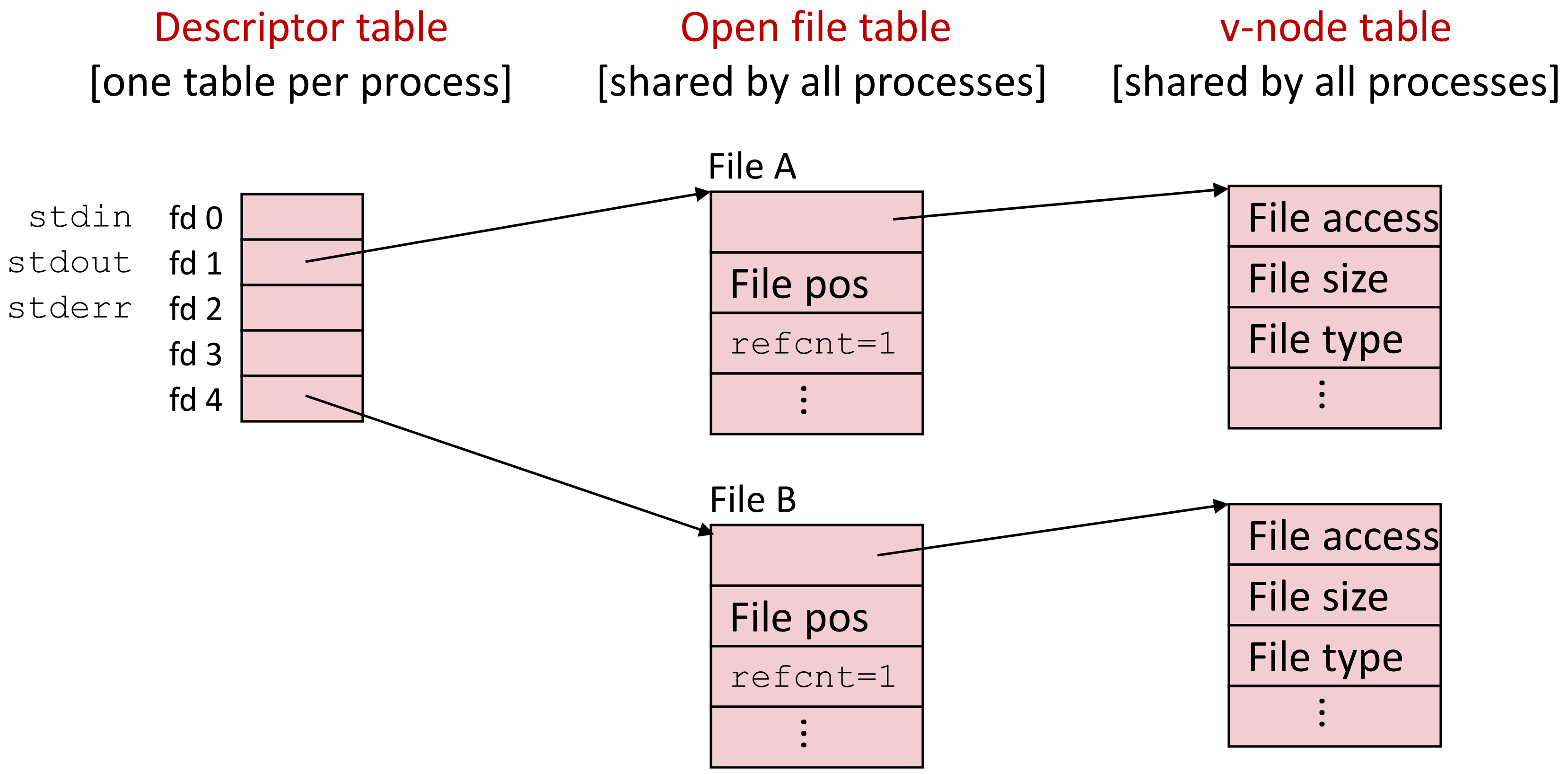
```
unix> ls > foo.txt
```

- Answer: By calling the `dup2(oldfd, newfd)` function
  - Copies (per-process) descriptor table entry **`oldfd`** to entry **`newfd`**



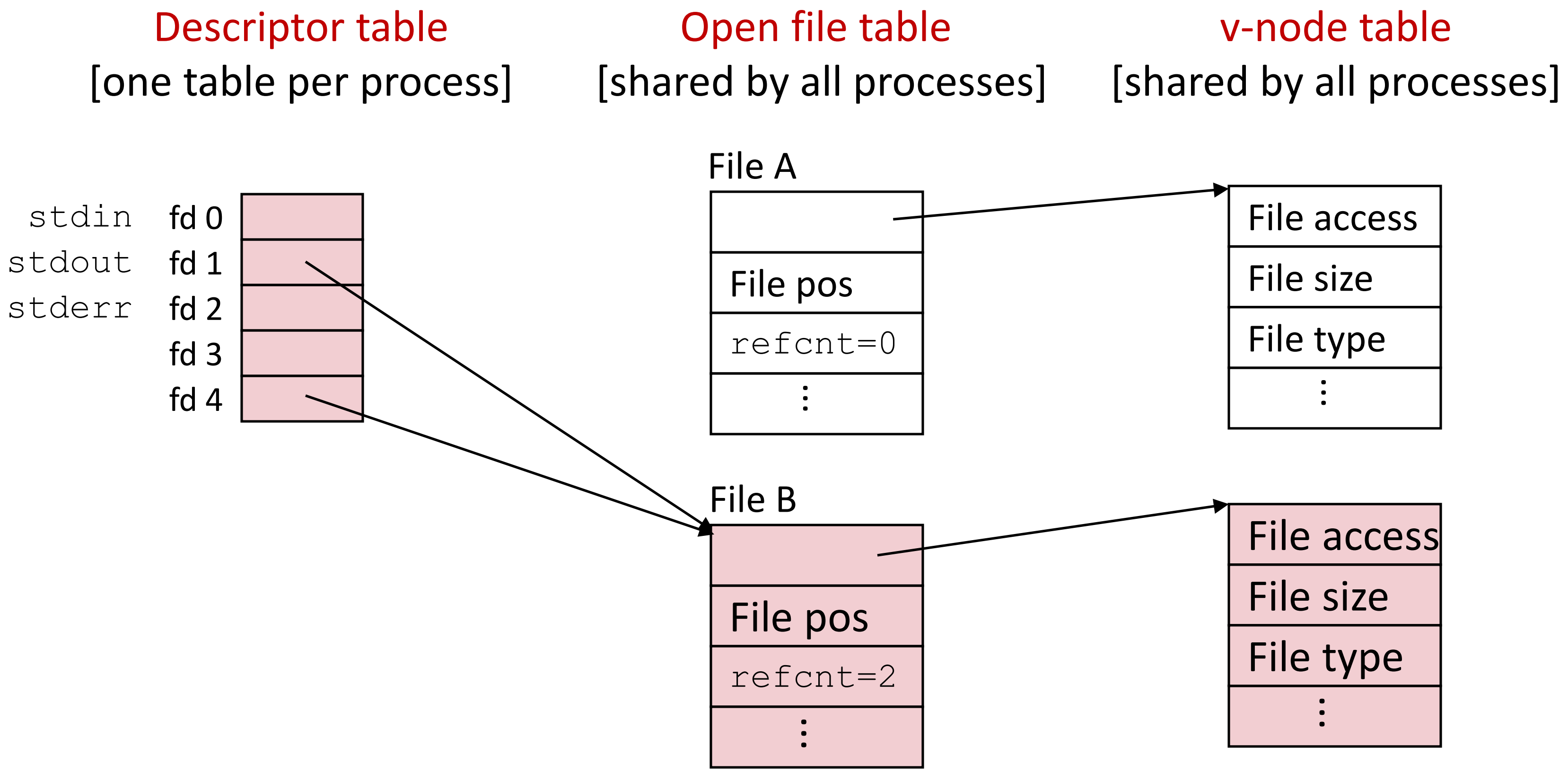
I/O Redirection Example

- Step #1: open file to which stdout should be redirected
  - Happens in child executing shell code, before **`exec`**



I/O Redirection Example (cont.)

- Step #2: call `dup2(4,1)`
  - cause fd=1 (stdout) to refer to disk file pointed at by fd=4



Example

- Assuming that the disk file foobar.txt consists of six ASCII characters “foobar” what is the output?

```
int main()
{
    int fd1, fd2;
    char c;

    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    Read(fd2, &c, 1);
    Dup2(fd2, fd1);
    Read(fd1, &c, 1);
    printf("c = %c\n", c);
    return 0;
}
```

- Because we are redirecting fd1 to fd2, the output is c=o

Practice: Fun with File Descriptors (1)

```
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;
}
```

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

Practice: Fun with File Descriptors (2)

```
#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;
}
```

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

Practice: Fun with File Descriptors (3)

```
#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1);  /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}
```

- What would be the contents of the resulting file?

Standard I/O Functions

- The C standard library (`libc.so`) contains a collection of higher-level *standard I/O* functions
  - Documented in Appendix B of K&R.

- Examples of standard I/O functions:
  - Opening and closing files (**`fopen`** and **`fclose`**)
  - Reading and writing bytes (**`fread`** and **`fwrite`**)
  - Reading and writing text lines (**`fgets`** and **`fputs`**)
  - Formatted reading and writing (**`fscanf`** and **`fprintf`**)

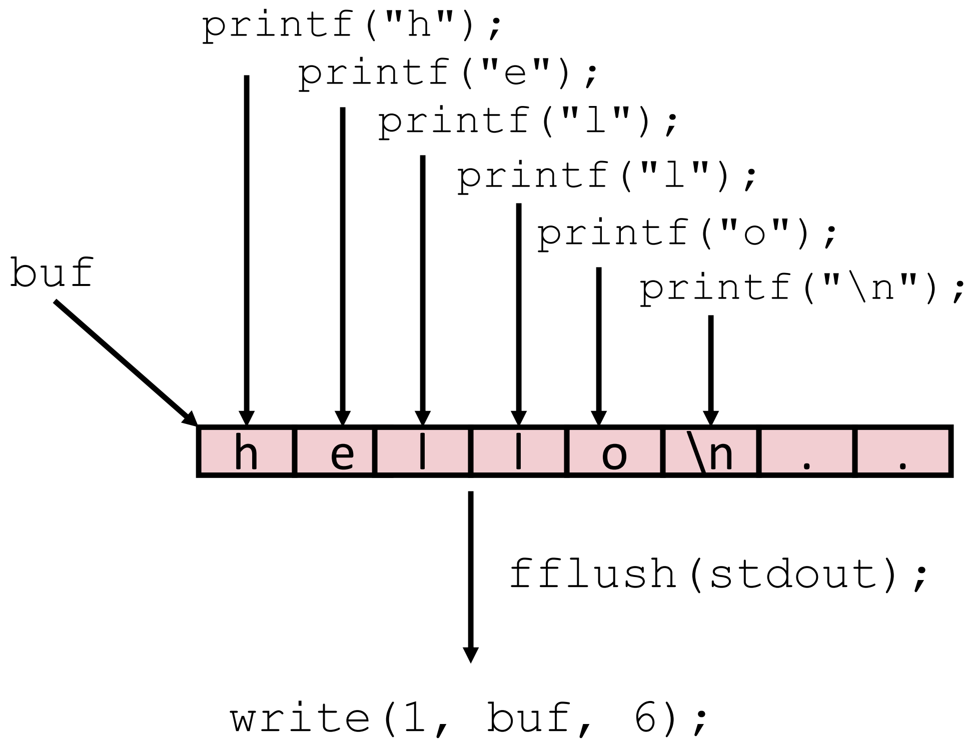
Standard I/O Streams

- 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

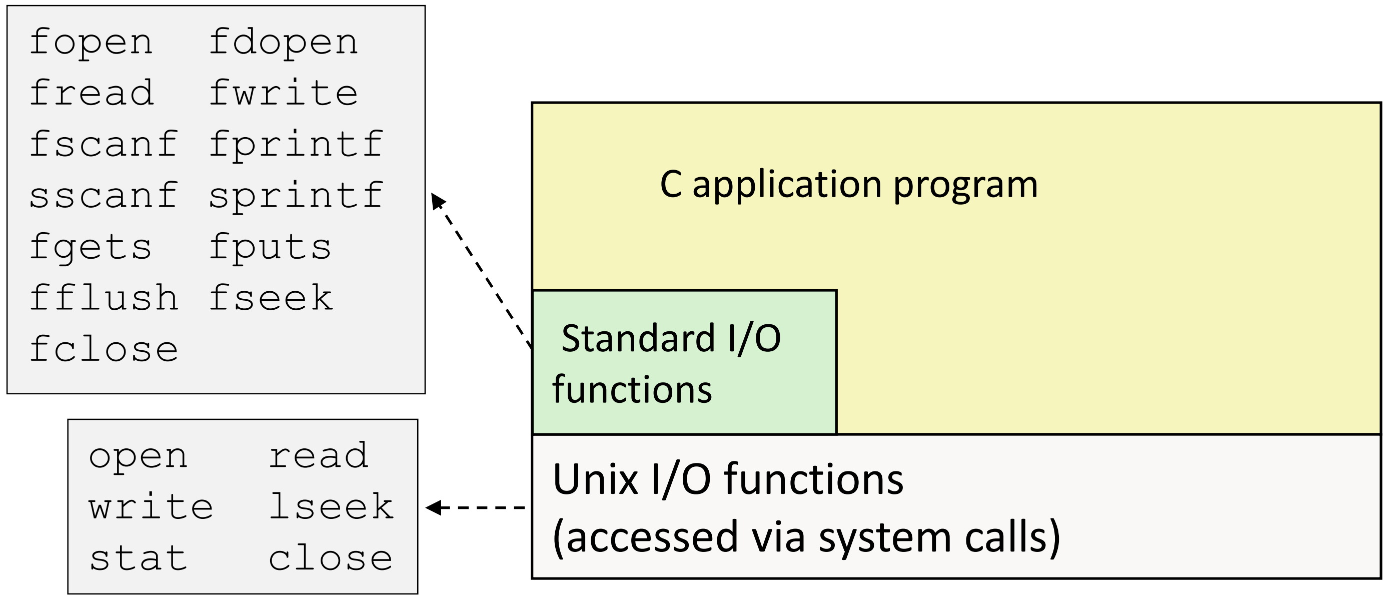
- Standard I/O functions use buffered I/O



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

Unix I/O vs. Standard I/O

- Standard I/O is implemented using low-level Unix I/O



- Which ones should you use in your programs?

Pros and Cons of Unix I/O

- Pros
  - Unix I/O is the most general and lowest overhead form of I/O
    - All other I/O packages are implemented using Unix I/O functions
  - Unix I/O provides functions for accessing file metadata

- Cons
  - Dealing with short counts is tricky and error prone
  - Efficient reading of text lines requires some form of buffering, also tricky and error prone
  - Both of these issues are addressed by standard I/O packages

Pros and Cons of Standard I/O

- Pros:
  - Buffering increases efficiency by decreasing the number of **`read`** and **`write`** system calls
  - Short counts are handled automatically
- Cons:
  - Provides no function for accessing file metadata
  - Standard I/O is not appropriate for input and output on network sockets
    - There are poorly documented restrictions on streams that interact badly with restrictions on sockets (CS:APP2e, Sec 10.9)

Choosing I/O Functions

- General rule: use the highest-level I/O functions you can
  - Many C programmers are able to do all of their work using the standard I/O functions

- When to use standard I/O
  - When working with disk or terminal files
- When to use raw Unix I/O
  - Inside signal handlers, because Unix I/O is async-signal-safe
  - In rare cases when you need absolute highest performance

For Further Information

- The Unix bible:
  - W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 3nd Edition, Addison Wesley, 2013
- *Computer Systems: A Programmer's Perspective*, Randal E. Bryant and David R. O'Hallaron,
  - Prentice Hall, 3rd edition, 2016, Chapter 10