Reference:

Text 1 Chapter 10

FILE I/O

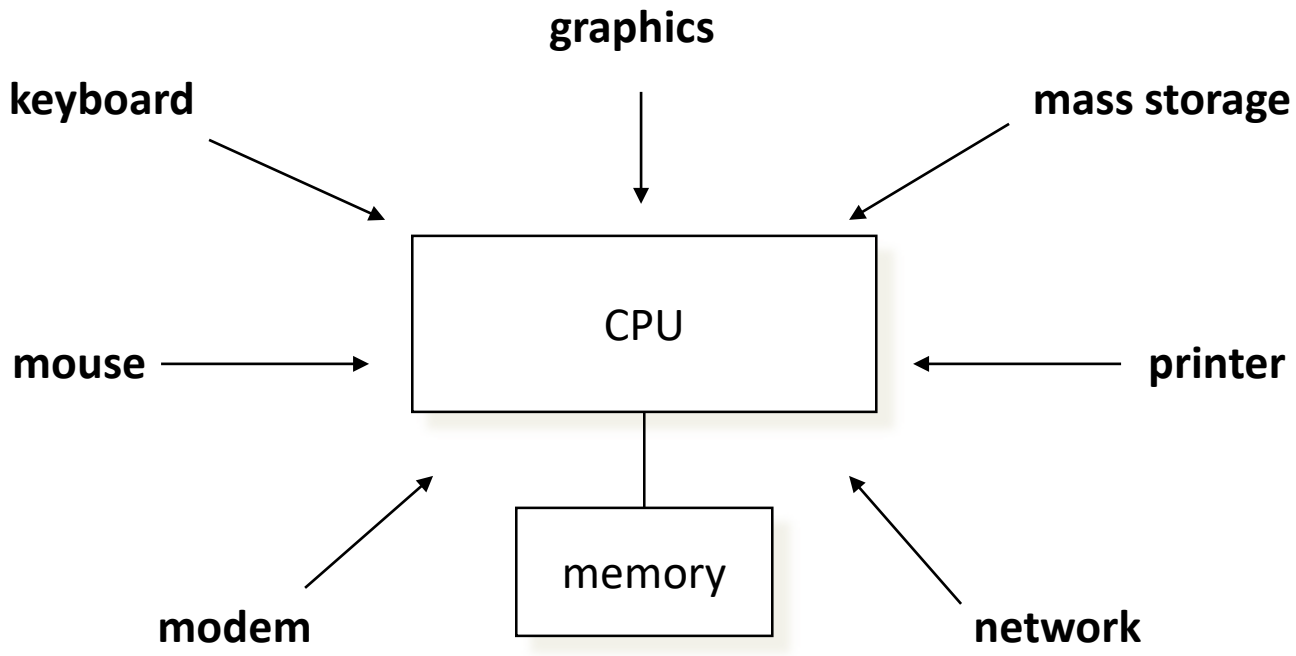
Tanzir Ahmed

CSCE 313 Spring 2021

What is I/O

- 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/cout`, `scanf/cin`)
  - *On Unix systems, these higher-level I/O functions eventually call* ***System-Level Unix I/O functions*** *provided by the Kernel (e.g.,* ***`printf()`*** *internally calls* ***`write()`****)*
  - *The same applies to Windows and other OSs*
    - **`printf()`** calls native windows function **`WriteFile()`**

Files are not always Disk Files: I/O Devices

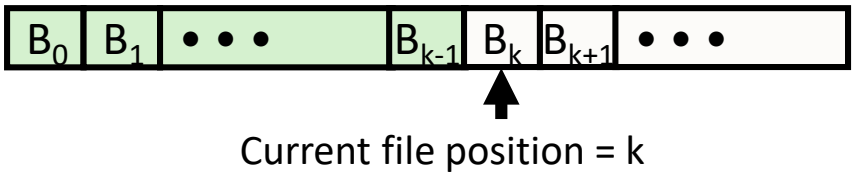


Unix File Types

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

Unix I/O

- Key Features
  - *Simple and consistent interface for I/O methods*
- Basic Unix I/O operations (system calls):
  - *Opening and closing files*
    - **`open()`** and **`close()`**
  - *Reading and writing a file*
    - **`read()`** and **`write()`**
  - *Current file position*
    - Kernel maintains a file position (initially 0) for each open file (aka **file pointer**)
    - Indicates next (byte) offset into file to read or write
    - Reading and writing automatically **advance** the file pointer
    - **`lseek()`** can be used to change file position **manually** at programmer's wish
      - *Instead of the default which is tied to read and write operations*



Opening Files

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

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

- Returns an identifying integer *file descriptor*
  - ***fd == -1*** *indicates that an error occurred*
- Kernel keeps track of all information about the open file. E.g., current file position, permissions etc.
- Each process created by a Unix shell begins with three open files associated with the terminal:
  - *0: standard input*
  - *1: standard output*
  - *2: standard error*
- On success, open returns the smallest available fd value
  - *E.g., in the above* ***fd == 3*** *(0-2 are occupied)*

Closing Files

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

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

- The kernel responds by
  - *freeing some data structures*
  - *restoring the descriptor to the pool of available descriptors*
- When a process terminates for any reason, the kernel performs **close** on all open files

Example – Descriptor Recycling

- What is the output of the following program?

```
int main (){
    int fd1 = open("foo.txt",O_RDONLY);
    close(fd1);
    int fd2 = open("baz.txt", O_RDONLY);
    cout << "fd2 = " << fd2 << endl;
}
```

- 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"

Creating a File

- So far, we have only opened files for reading
  - *The file must have existed beforehand*
- To create a file, we need to call **`open()`** with write **flag** and **mode** specifying appropriate permissions

```
int open(const char *pathname, int flags, mode_t mode);
```

- Consult linux man pages for more details. However, here are some common use cases
- We will commonly use one of these 3 flags: **`O_RDONLY`**, **`O_WRONLY`**, or **`O_RDWR`** for read, write or both, respectively
  - *There are other sub-flags that can be bitwise OR-ed with these*
- When opening for read, no need to use the last argument **`mode`**, because you are not creating a file
- However, when creating one, you must set the permissions of a file, which is next

Creating a File (2)

```
int open(const char *pathname, int flags, mode_t mode);
```

- The following two things specifies permissions of a file:

Types of Access	Types of Users
• Read (R)	• User/Owner (U)
• Write (W)	• Group (G)
• Execute (X)	• Others (O)

- For each user, there is a 3-digit octet **RWX** that determines what level of permissions he has.
- For instance, if the User/Owner has R, W, and X permission, he gets 7 (=111 in binary), if the group has only R and W permissions, everybody in that group gets 6 (=110 in binary), and no permission for others would give 0. Together, the permission for the file should be: **760**

Creating a File (3)

```
int open(const char *pathname, int flags, mode_t mode);
```

- To create a file only for yourself:

```
int main (){
      int fd = open ("test.txt",
      O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU);
      if (fd<0){
            perror ("Cannot create file
      ");exit(0);
      }
      cout << "File created successfully" << endl;
}
```

```
prompt> ls -l
total 36
-rwxrwxr-x 1 osboxes osboxes 17424 Sep  7 15:23 a.out
-rw-rw-r-- 1 osboxes osboxes   256 Sep  2 14:21 exec1.cpp
-rw-rw-r-- 1 osboxes osboxes   313 Sep  7 15:23 fileio1.cpp
-rw-rw-r-- 1 osboxes osboxes   288 Sep  1 23:08 fork1.cpp
-rw-rw-r-- 1 osboxes osboxes   496 Sep  2 14:01 fork2.cpp
-rwx------ 1 osboxes osboxes     0 Sep  7 15:23 test.txt
```

- However, keep in mind that you cannot just directly put `700` for permission because `700` in decimal is not the same as 700 in octet. You can write `0700` though, because that represents octal number

This is incorrect

```
int main (){
      int fd = open ("test.txt",
      O_WRONLY|O_CREAT|O_TRUNC,700);
      if (fd<0){
            perror ("Cannot create file
      ");exit(0);
      }
      cout << "File created successfully" << endl;
}
```

```
prompt> ls -l
total 36
-rwxrwxr-x 1 osboxes osboxes 17424 Sep  7 15:22 a.out
-rw-rw-r-- 1 osboxes osboxes   256 Sep  2 14:21 exec1.cpp
-rw-rw-r-- 1 osboxes osboxes   309 Sep  7 15:22 fileio1.cpp
-rw-rw-r-- 1 osboxes osboxes   288 Sep  1 23:08 fork1.cpp
-rw-rw-r-- 1 osboxes osboxes   496 Sep  2 14:01 fork2.cpp
--w-rwxr-T 1 osboxes osboxes     0 Sep  7 15:22 test.txt
```

Reading Files

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

```
char buf[512];
int fd = open (filename, O_RDONLY);
int nbytes;
/* 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 when the file does not have `nbytes` of data starting from the file pointer, reading text from terminal where user gave `< nbytes` etc.)*

Reading Files - Example

- Output of the following program reading a file containing “foobar”

```
char buf [2];
int fd = open (filename, O_RDONLY);
int nbytes = read(fd, buf, sizeof(buf)); // buf = ?
nbytes = read(fd, buf, sizeof(buf)); // buf = ?
```

Answer: “fo” and “ob”
Why not “fo” and “foob”??

Writing Files

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

```
char buf[512];
int nbytes;   /* number of bytes written*/
int fd = open (filename, O_WRONLY|O_CREAT, 0700);
/* 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!*

Writing Files - Example

- What would be the state of the file after executing the following program?

```
char buf[] = {'a', 'b'}; // what if char* buf = “ab”
int fd;        /* file descriptor */
/* Open the file fd ... */
/* Then write up to 2 bytes from buf to file fd */
write(fd, buf, sizeof(buf)); //file = ?
write(fd, buf, sizeof(buf)); //file = ?
```

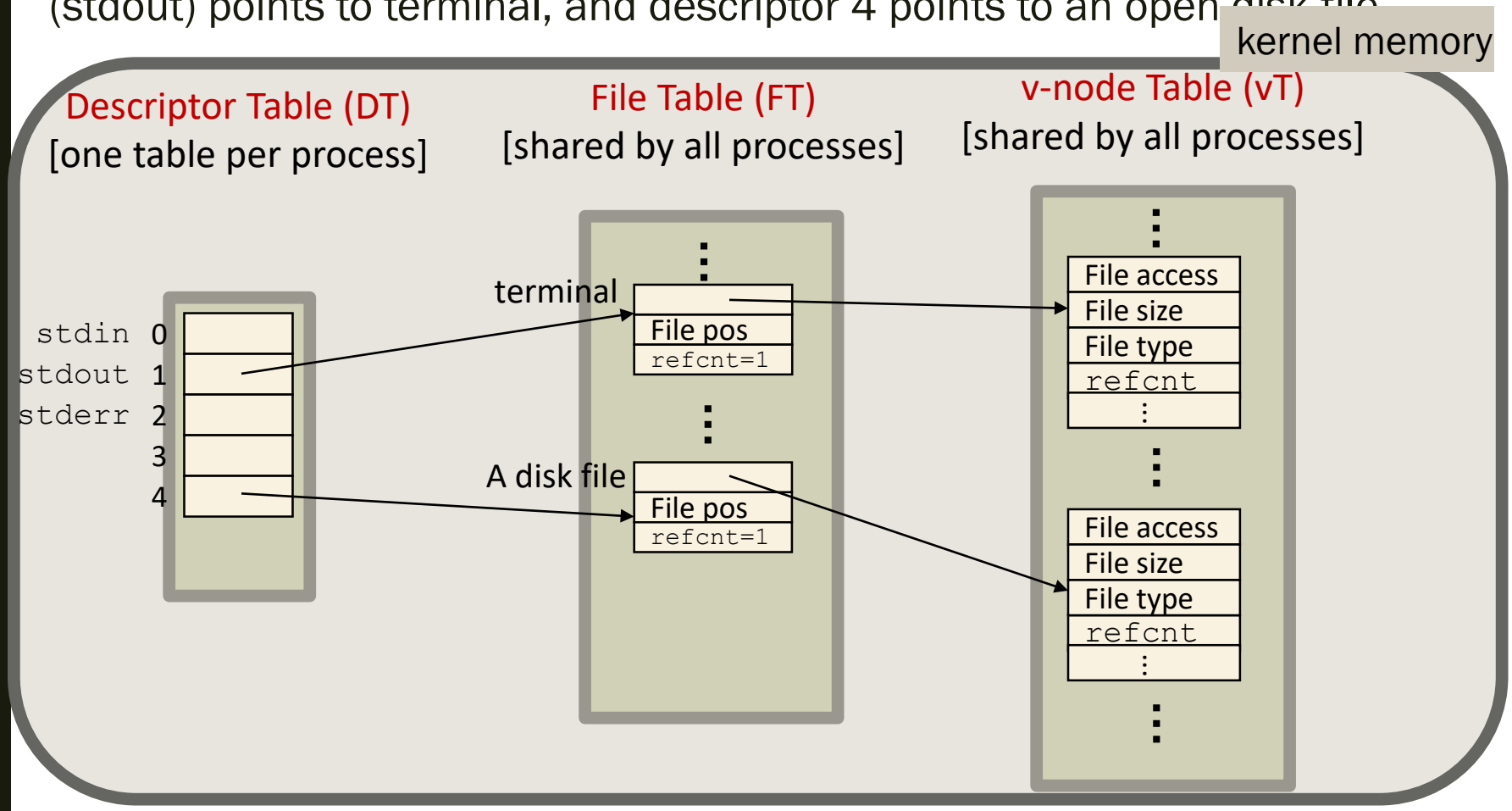
abab

File Representation – 3 Different Tables

- **Descriptor Table**
  - *Each process has its own separate descriptor table (only accessible directly by the kernel)*
  - *Entries are indexed by the process's open file descriptors*
  - *Each entry points to a **File Table** entry*
- **File Table**
  - *Shared by all processes.*
  - *Each entry consists of*
    - File Cursor
    - Reference count: # of descriptor entries from all processes that point to it (why do we need this??)
    - ptr to the v-node table entry, along with status flags
- **v-node Table**
  - *Each entry (1 per file) contains information about the file meta data (e.g., access/modification date, permissions)*

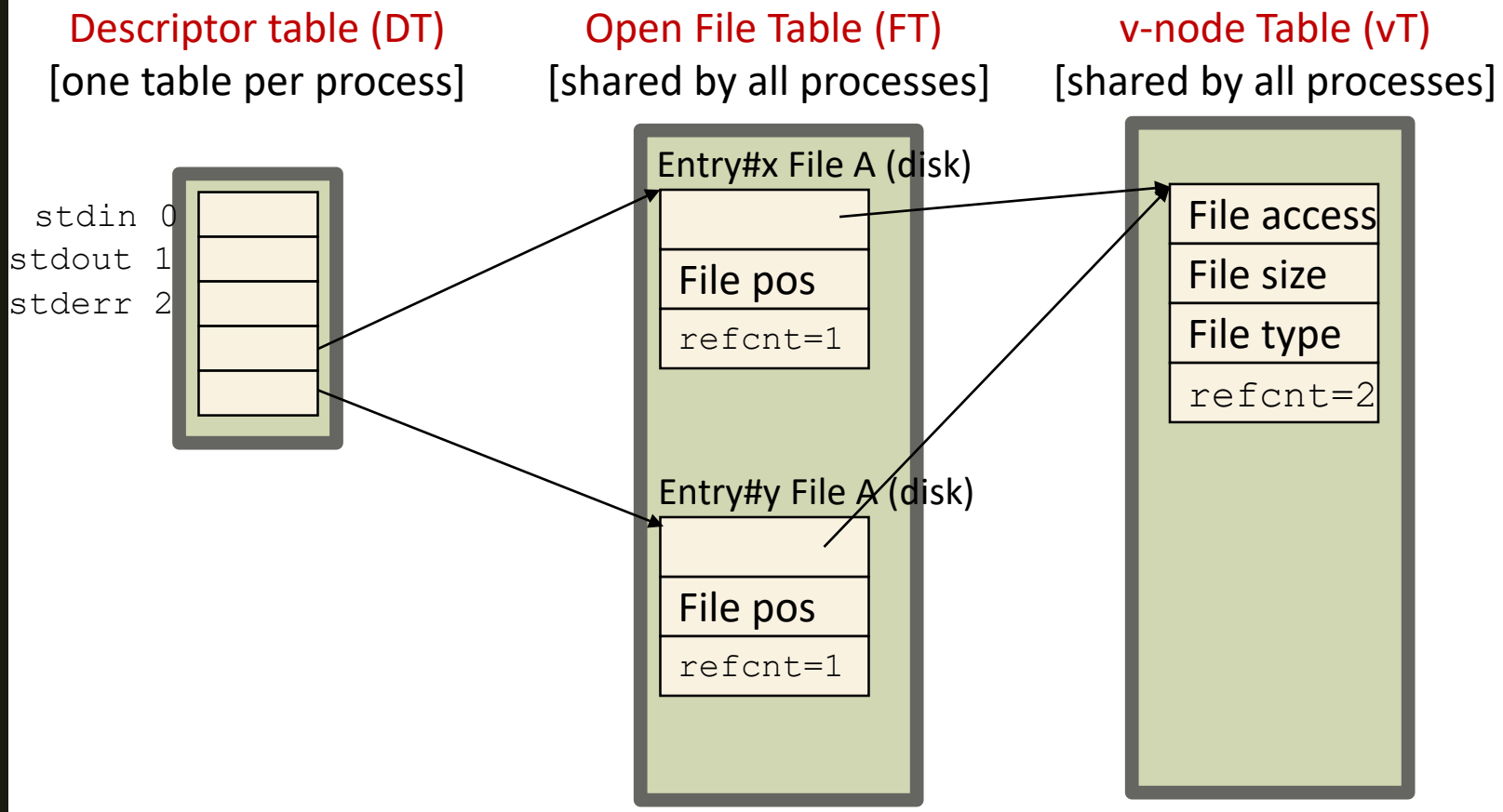
How the Unix Kernel Represents Open Files

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



File Sharing

- Two distinct DTs sharing the same disk file through two distinct open FT entries
  - *E.g., calling* ***`open`*** *twice with the same* ***`filename`*** *argument*
  - *Every* ***`open`*** *creates a new DT and a new FT entry, also possibly a vT entry*



Example

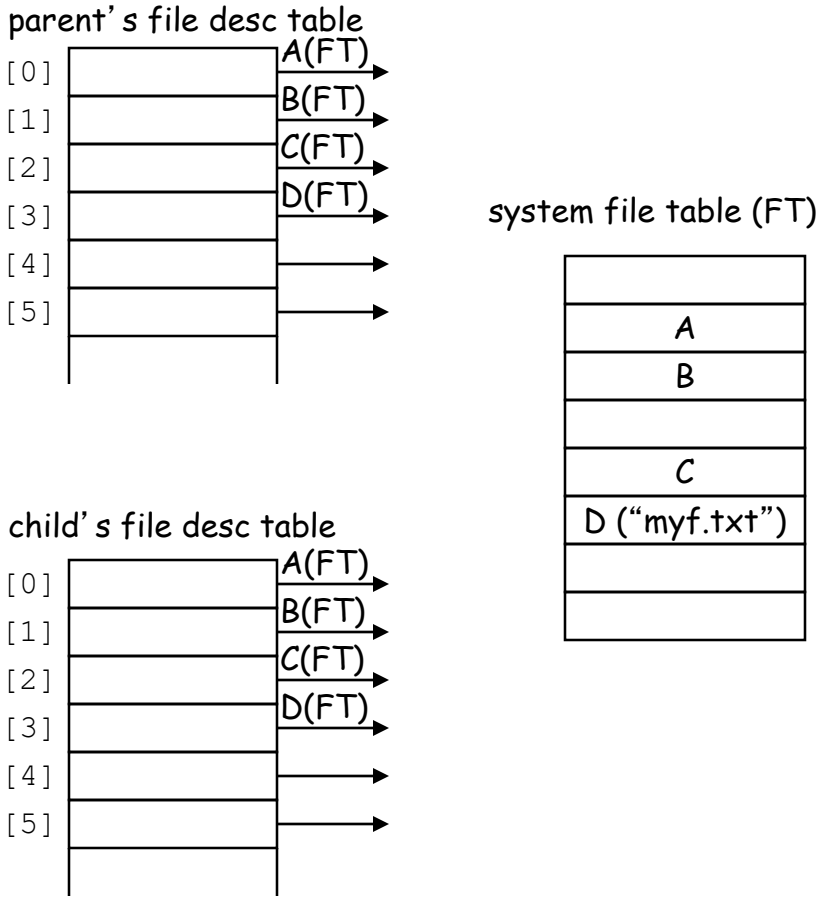
- Suppose the disk file foobar.txt consists of the six ASCII characters “foobar”. Then what is the output of the following program:

```
int main (){
    char c;
    int fd1 = open("foobar.txt", O_RDONLY, 0);
    int fd2 = open("foobar.txt", O_RDONLY, 0);
    read(fd1, &c, 1);
    cout << c << endl;
    read(fd2, &c, 1);
    cout << c << endl;
}
```

- *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 fd1 reads the first byte of foobar.txt and the output is “**f**”. Same is true for fd2 and the output is again “**f**”*

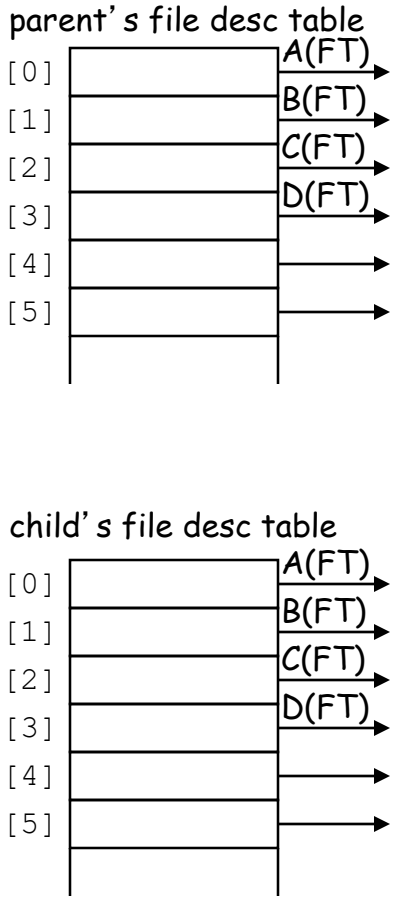
File Descriptors and `fork()`

- With `fork()`, child inherits content of parent's address space, including file descriptor table



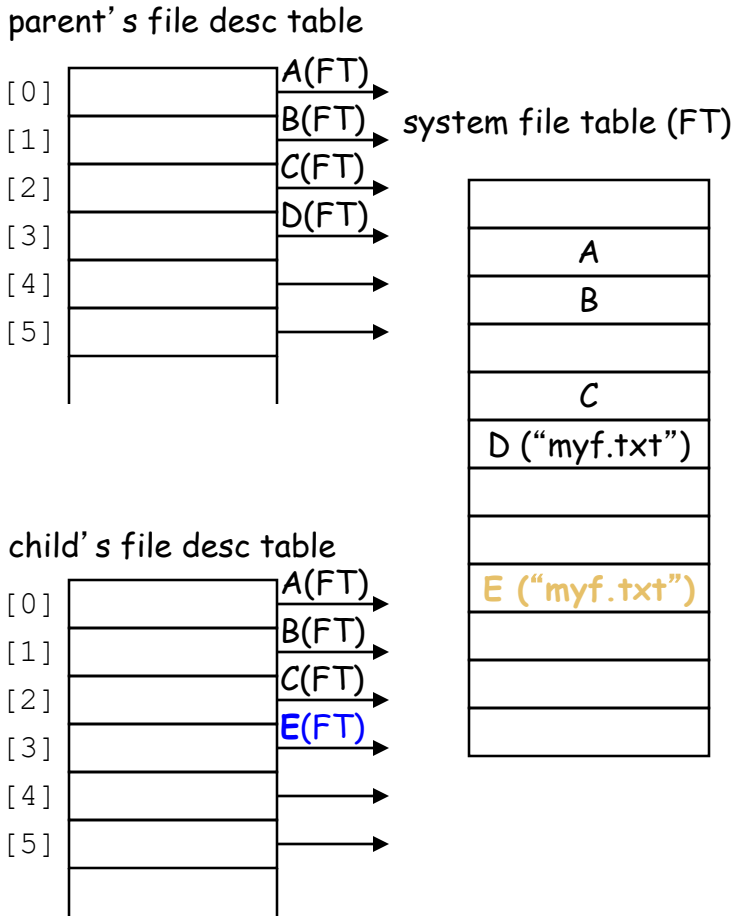
File Descriptors and `fork()`

```
int main(void) {
  char c;
  int myfd=open("myf.txt",O_RDONLY);
  fork();
  read(myfd, &c, 1);
  printf("Got %c\n", ), c);
}
```



File Descriptors and `fork()` (III)

```
int main(void) {
  char c;
  fork();
  int myfd=open("myf.txt",O_RDONLY);
  read(myfd, &c, 1);
  printf("Got %c\n", c);
}
```



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 (){
    char c;
    int fd = open("foobar.txt", O_RDONLY);
    if (fork() == 0) {
       read(fd, &c, 1);
       return 0;
    }else{
        wait(0);
        read(fd, &c, 1);
        cout << "c=" << c << endl;
    }
}
```

- *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"*

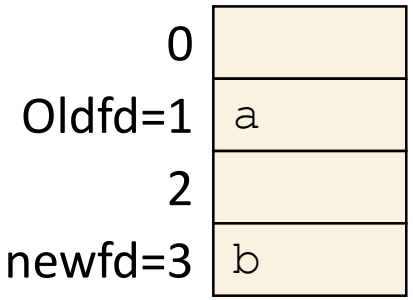
I/O Redirection

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

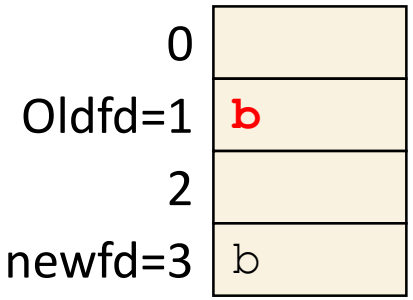
```
unix> ls > foo.txt
```

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

Descriptor table
before `dup2(3,1)`

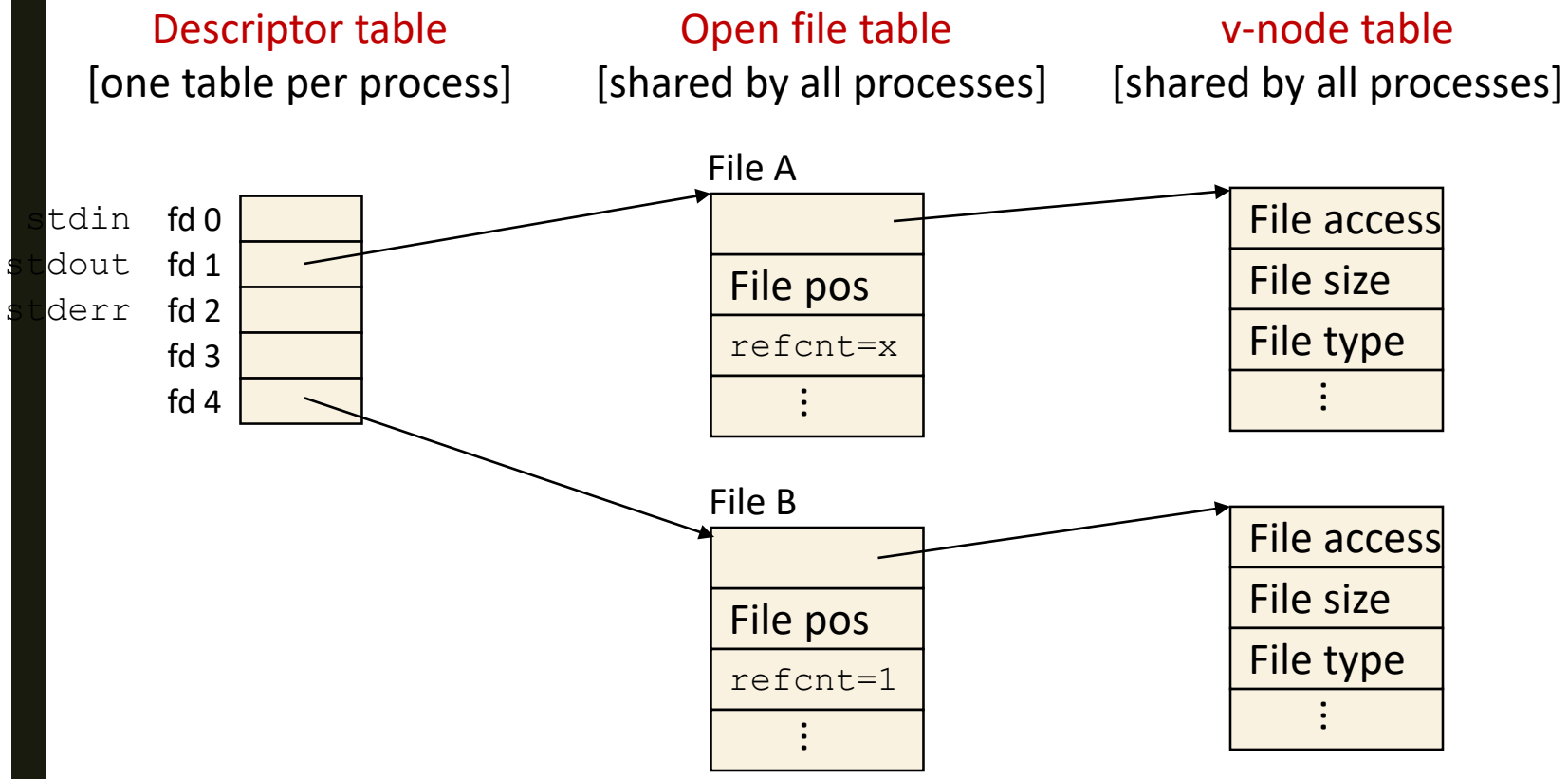


Descriptor table
after `dup2(3,1)`



I/O Redirection Example

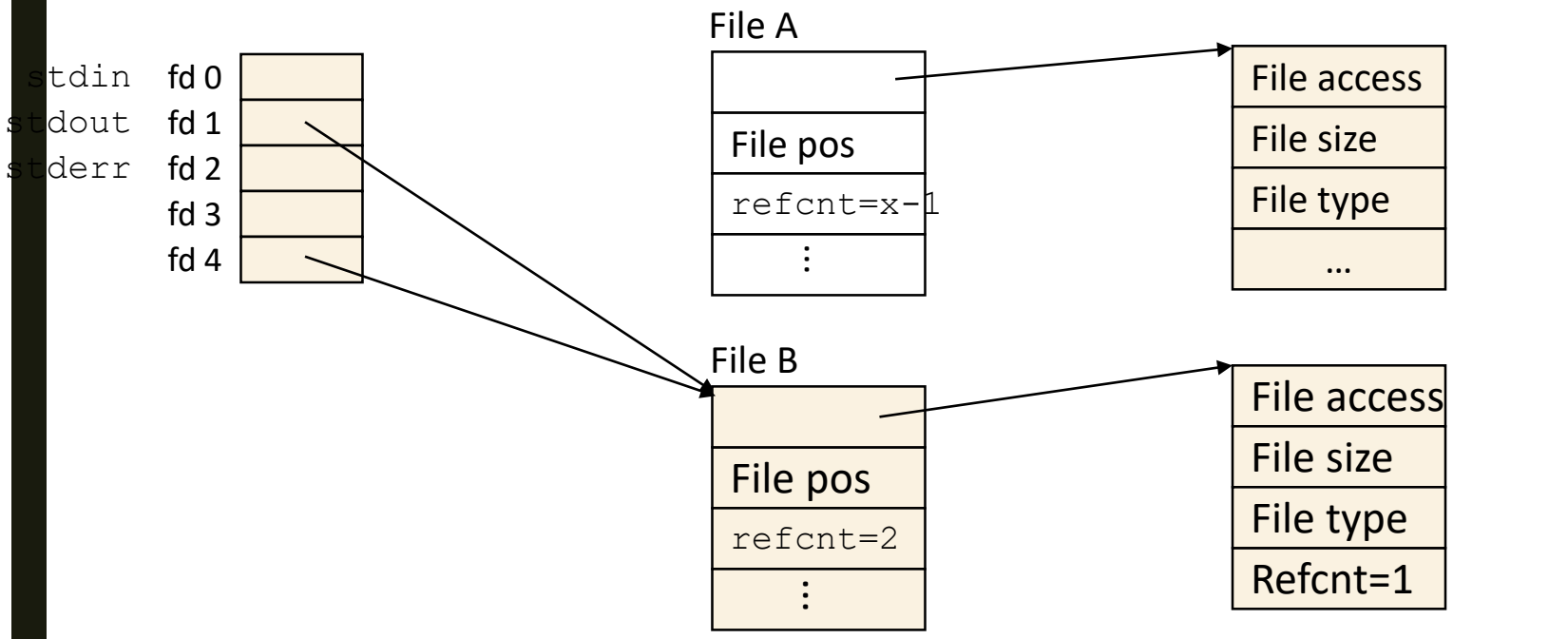
Step #1: open a that will be the target of redirection



I/O Redirection Example (cont.)

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

Descriptor table [one table per process] | Open file table [shared by all processes] | v-node table [shared by all processes]



Example

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

```
int main(){
    int fd1 = open("foobar.txt", O_RDONLY);
    int fd2 = open("foobar.txt", O_RDONLY);
    char c;
    read(fd2, &c, 1);   // read a char fd2
    dup2(fd2, fd1);     // duplicate
    read(fd1, &c, 1);   // read a char fd1
    cout << "c = " << c << endl;
}
```

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

Practice: Fun with File Descriptors (1)

```
int main(){
    char c1, c2, c3;
    char *fname = "file.txt";
    fd1 = open(fname, O_RDONLY);
    fd2 = open(fname, O_RDONLY);
    fd3 = open(fname, O_RDONLY);
    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);
}
```

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

Practice: File Descriptors (2)

```c
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);
    fd2 = open(fname, O_APPEND|O_WRONLY, 0);
    write(fd2, "jklmn", 5);
    fd3 = dup(fd1);  /* Allocates descriptor */
    write(fd3, "wxyz", 4);
    write(fd2, "ef", 2);
    // playing with file position
    lseek (fd1, 0, SEEK_SET);
    write (fd1, "ab", 2);
    return 0;
}
```

Memory Mapped File

- When working with files, it may be inconvenient to process structured information
  - *This could be much easier using simple pointer arithmetic if data was in memory*
- Luckily, we have a way to do that. The system call for that is `mmap`, whose signature is as follows:

```
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

  - ***addr*** *is a hint, which can be set to* ***NULL***
  - ***length*** *is the length of the segment*
  - ***prot*** *must match with how you opened the file initially*
  - ***flags*** *tell whether you want a convenient a local copy (MAP_PRIVATE) or your changes reflected to the disk (MAP_SHARED) and possibly other processes*
  - ***fd*** *is the descriptor and* ***offset*** *counted from the beginning*

mmap Example

```
int main (){
        int fd = open ("test.txt", O_RDWR);
        off_t len = lseek (fd, 0, SEEK_END); // go to end to get file size
        lseek (fd, 0, SEEK_SET); // seek back to the beginning

        char *buf = (char *) mmap (0, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        //copying bytes 20-29 off the file to the test buffer
        char test [11];
        memcpy (test, buf+20, 10);
        test [10] = 0; // putting a NULL so it can be printed w/o crashing
        cout << "Bytes 20-29 in file: " << test << endl;

        // writing to file
        memcpy (buf, "Hola Mundo", strlen ("Hola Mundo"));

        // unmapping
        munmap (buf, len);
        close (fd);
}
```

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

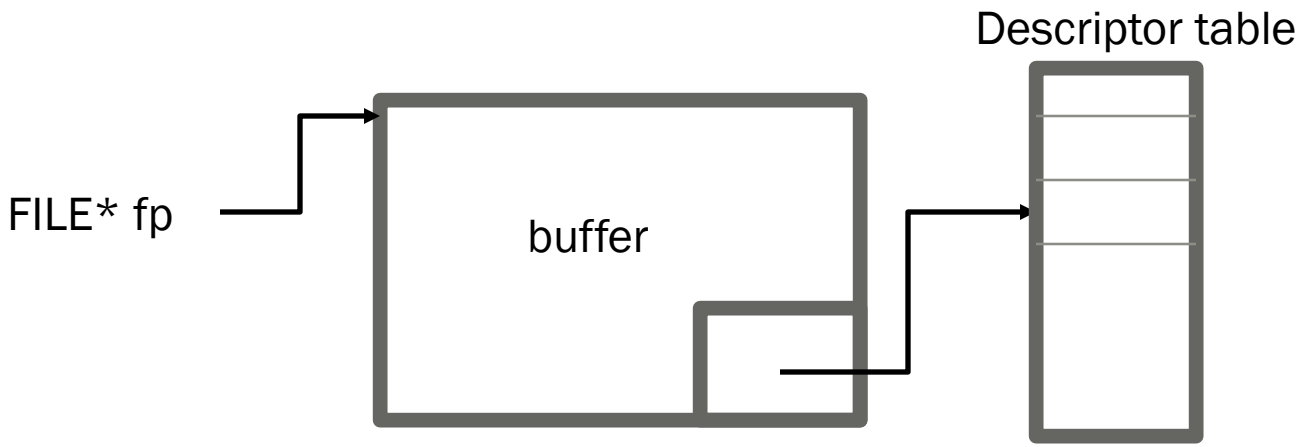
- Standard I/O models open files as *streams*
  - *Abstraction for a file descriptor and a buffer in memory.*
- C programs begin life with three open streams (defined in `stdio.h`)
  - ***stdin*** *(standard input)*
  - ***stdout*** *(standard output)*
  - ***stderr*** *(standard error)*

```
#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");
}
```

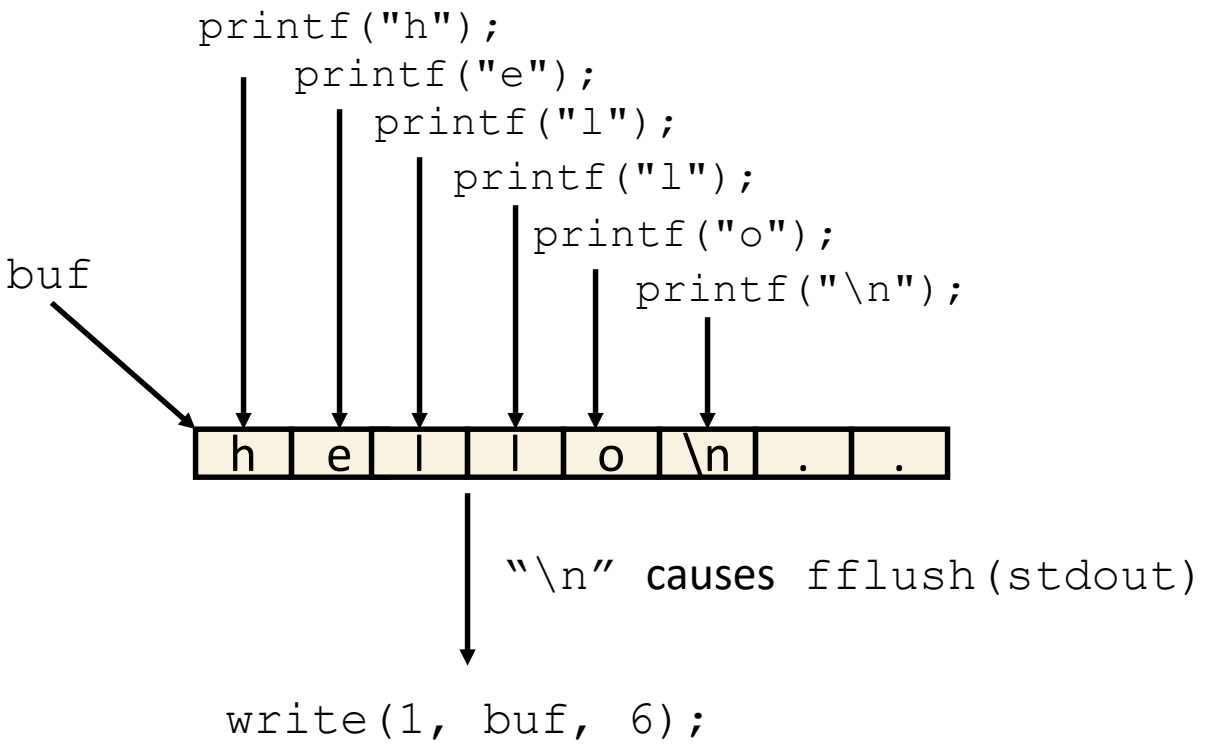
Standard I/O – An Extra Layer

- Motivations
  - *Portability: Code is compiled to Windows-native or Linux-native code, not tied to a particular architecture*
  - *Buffering: For efficiency, considering device type*
    - **`fprintf`** to **`stdout`** is flushed on '\n' character
    - **`fprintf`** to disk files is flushed after reaching file buffer size



Buffering in Standard I/O

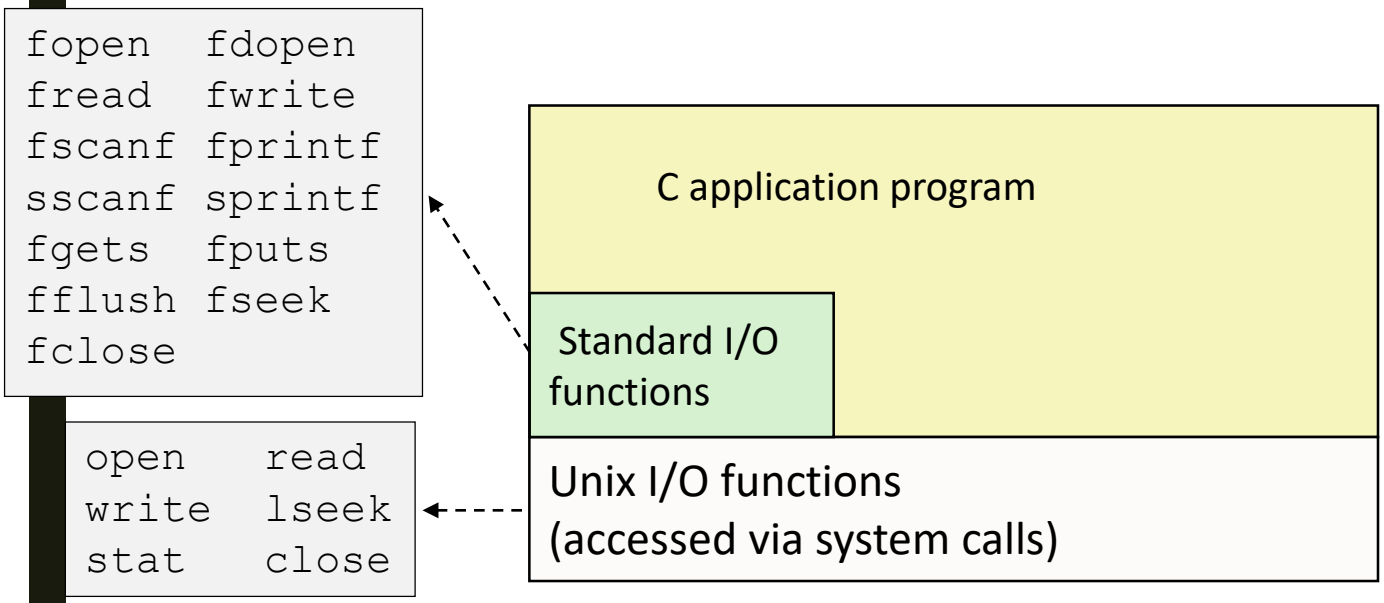
- Standard I/O functions use buffered I/O



- 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
  - *It is the most general and lowest overhead form of I/O*
    - All other I/O packages are implemented using Unix I/O functions on a Unix system
  - *It provides functions for accessing file metadata*
- Cons
  - *Efficient data access requires some form of buffering, which is:*
    - Device dependent (e.g., a whole track of a disk can be read at once)
    - tricky and error prone
  - *Both of these issues are addressed by standard I/O packages*

Pros and Cons of Standard I/O

- Pros:
  - *Portable code – same code works on Windows and Unix*
  - *Buffering increases efficiency by decreasing the number of* ***`read`*** *and* ***`write`*** *system calls*
  - *Less burden on the programmer*
- 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*
    - You do your own buffering depending on the application nature and knowledge about the underlying hardware
    - On a 24 hard-drive (i.e., 48 TB cap) RAID system, we needed to exploit windows-native I/O to get nearly 3 GB/s read/write speed
    - Standard I/O would only give us < 1GBps read/write speed

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*