Lecture 7: File I/O in Linux

Lecturer: Prof. Zichen Xu

Notice

- The Labors day is coming right at the corner
  - I’m gonna miss you guys for about two weeks
- The sad story is we have another class scheduled on Saturday
  - ☹ ☹ ☹
- The good news is that we do not have to get up that early
  - Our class is rescheduled to Tuesday 1st, and 2nd class in the afternoon
  - The lab session keeps on

Outline

- Unix I/O
- Robust reading and writing
- Reading file metadata
- Sharing files
- I/O redirection
- Standard I/O

*nix Files

- A *nix *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots, B_k, \ldots, B_{m-1}$
- All I/O devices are represented as files:
  - `/dev/sda2` (`/usr` disk partition)
  - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
  - `/dev/kmem` (kernel memory image)
  - `/proc` (kernel data structures)

*nix File Types

- Regular file
  - Binary or text file.
  - Unix does not know the difference!
- 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 interprocess comunication
- Socket
  - A file type used for network communication between processes

*nix I/O

- The elegant mapping of files to devices allows kernel to export simple interface called *nix I/O.
- Key idea: All input and output is handled in a consistent and uniform way.
- Basic Unix I/O operations (system calls):
  - Opening and closing files
    - `open()` and `close()`
  - Changing the *current file position* (seek)
    - `lseek` (not discussed)
  - Reading and writing a file
    - `read()` and `write()`

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
- 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.

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

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

- Closing an already closed file is a recipe for disaster in threaded programs (more on this later)
- Moral: Always check return codes, even for seemingly benign functions such as `close()`

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`
  - `nbytes < 0` indicates that an error occurred.
  - *short counts* (`nbytes < sizeof(buf)`) are possible and are not errors!

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 read */

/* 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!
- Transfers up to 512 bytes from address `buf` to file `fd`

*nix I/O Example

- Copying standard input to standard output one byte at a time.

```
#include "csapp.h"

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.
- Maintained by kernel, accessed by users with the `stat` and `fstat` functions.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device)
*/
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
```

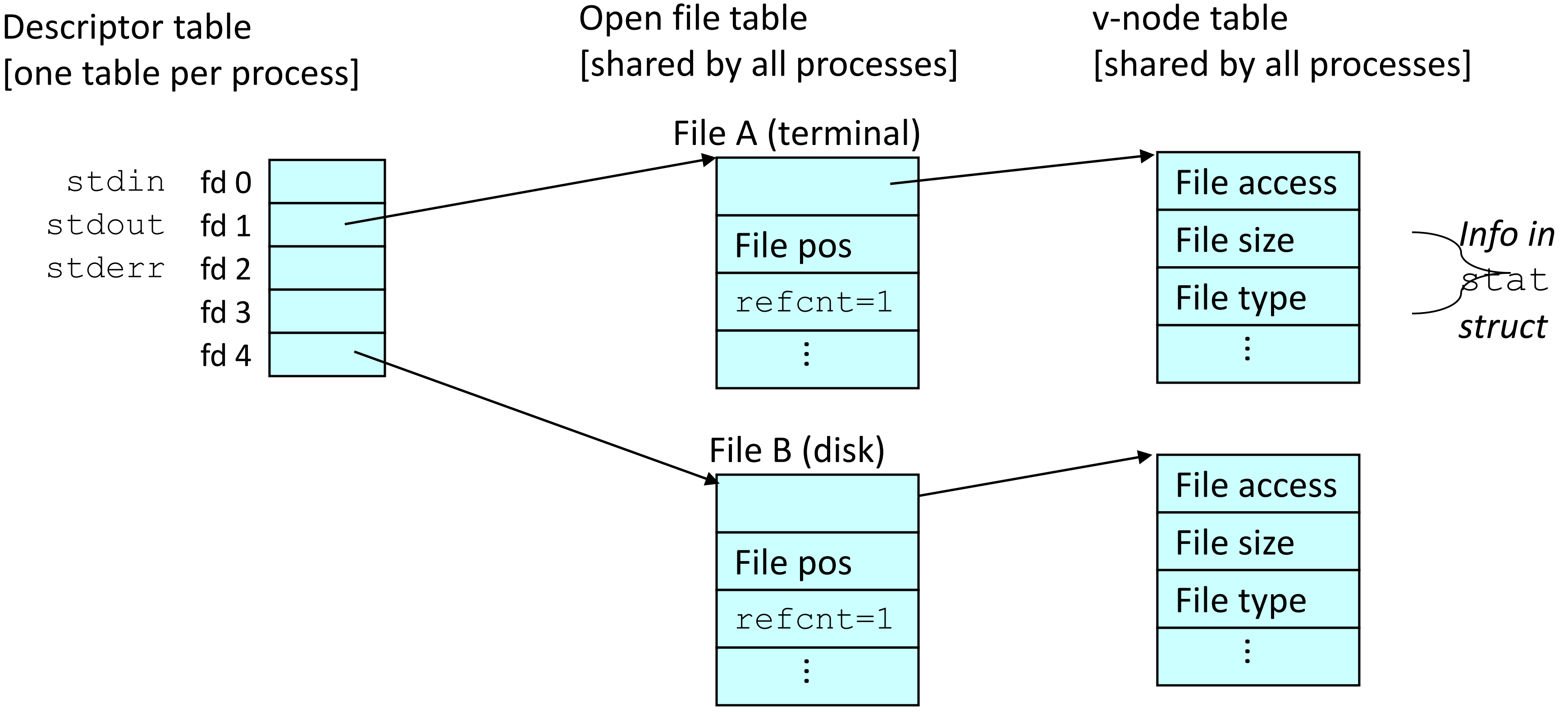
Example of Accessing File Metadata

```
/* statcheck.c - Querying and manipulating a file's meta data
*/
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* file type*/
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* OK to read?*/
        readok = "yes";
    else
        readok = "no";
    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

```
good> ./statcheck statcheck.c
type: regular, read: yes
good> chmod 000 statcheck.c
good> ./statcheck statcheck.c
type: regular, read: no
```

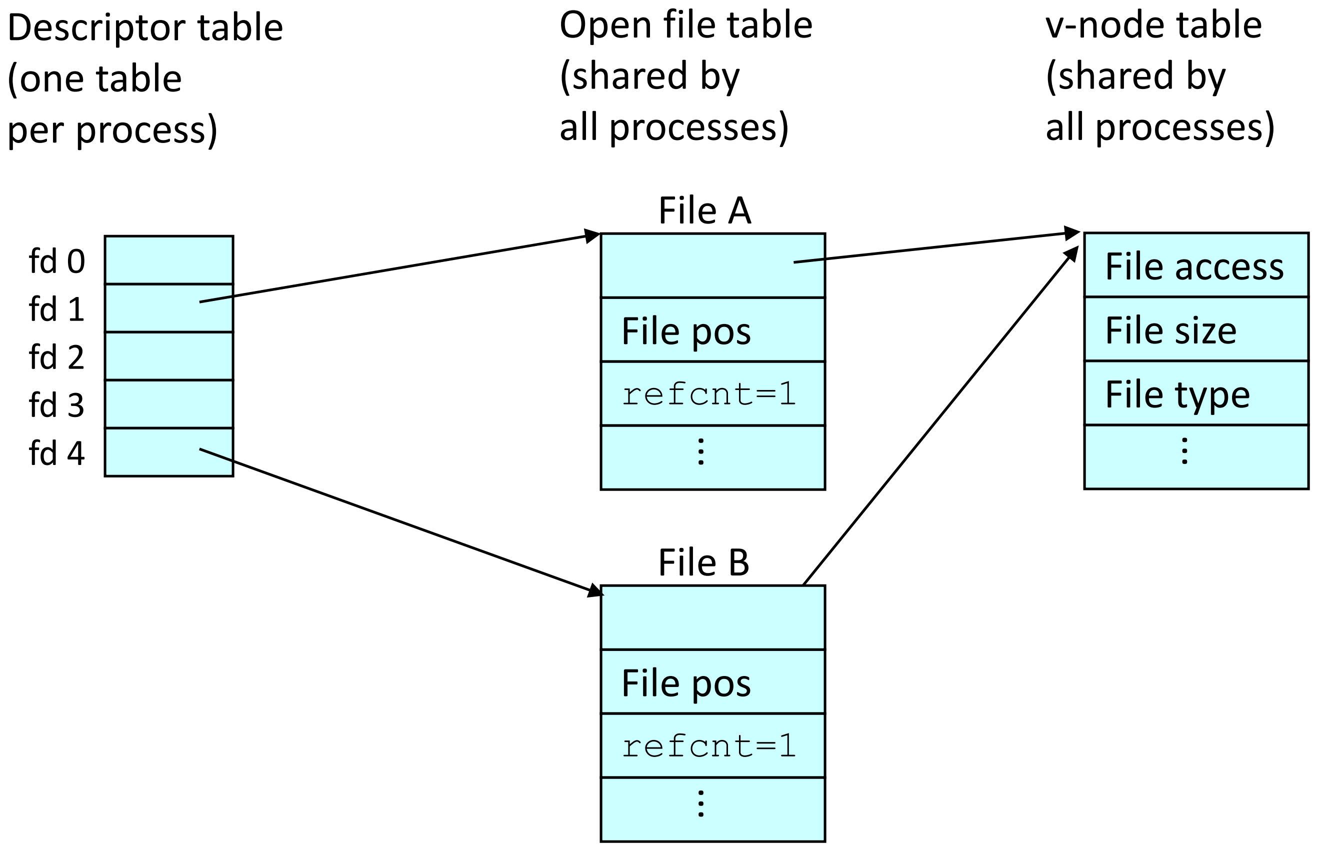
How the *nix Kernel Represents Open Files

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



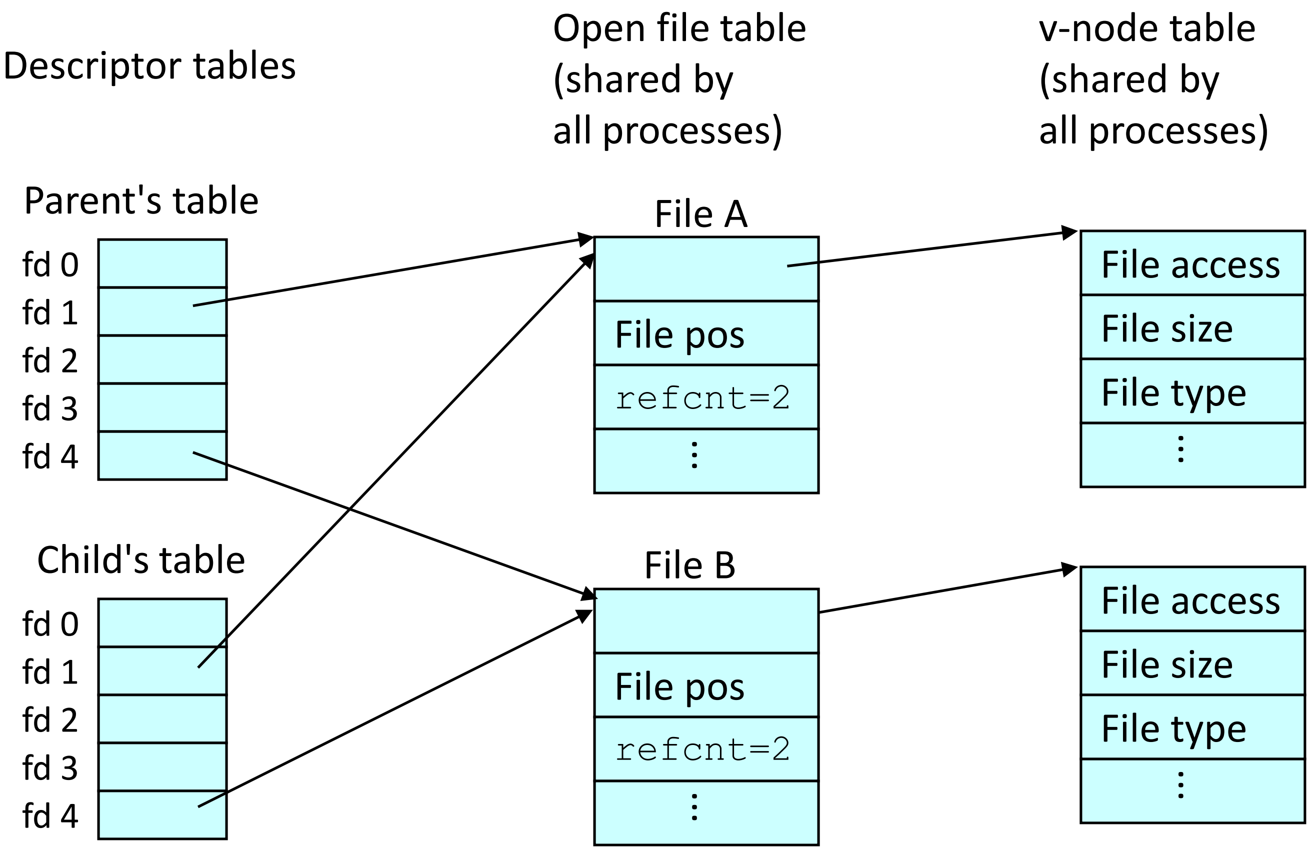
File Sharing

- 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

- A child process inherits its parent's open files. Here is the situation immediately after a `fork`

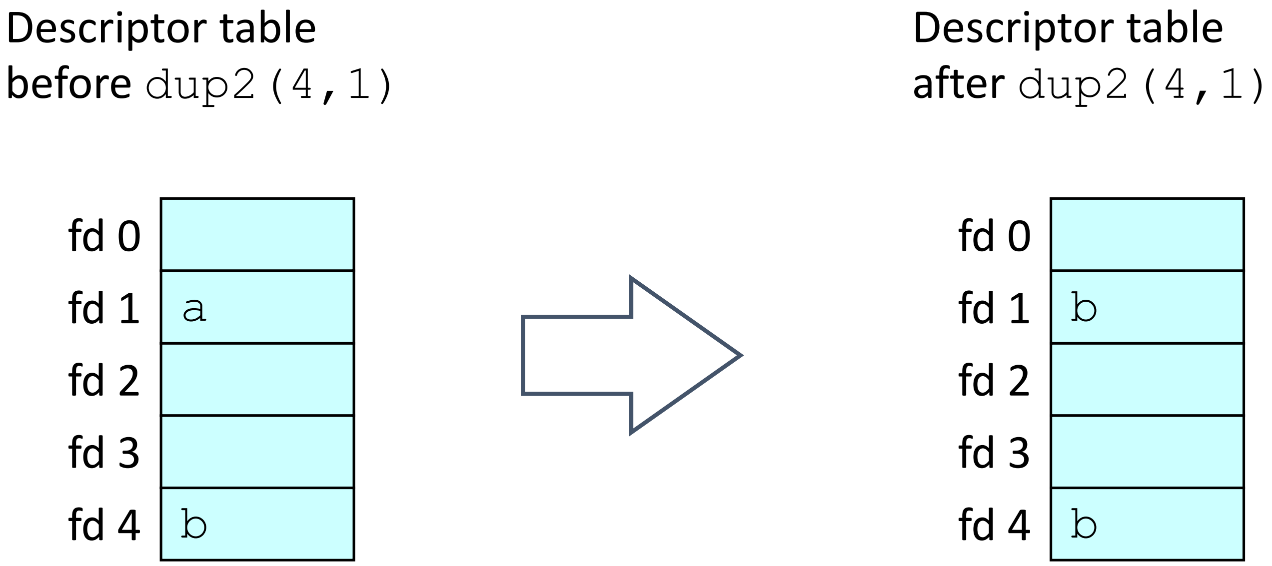


I/O Redirection

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

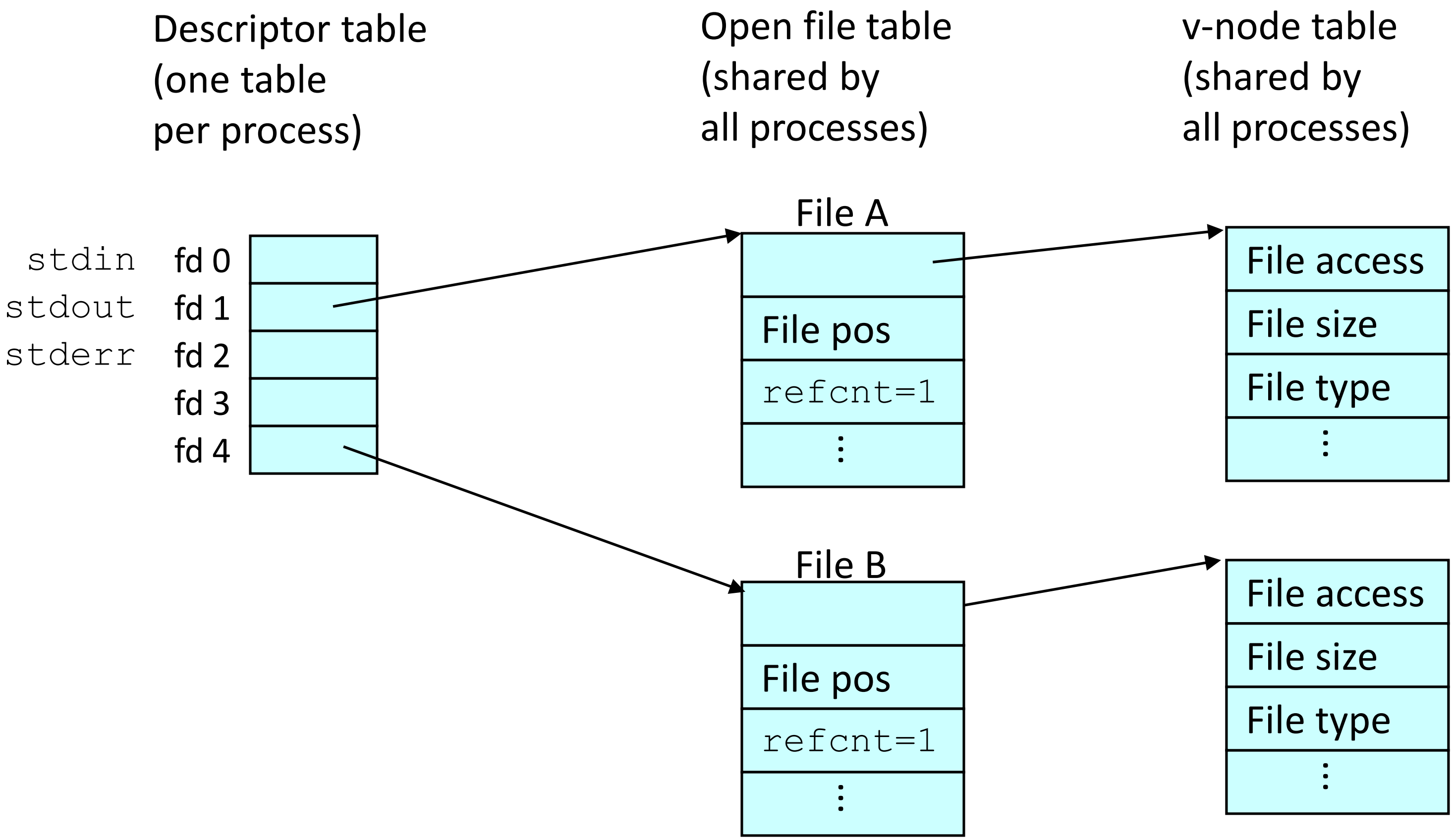
  `Linix> ls > foo.txt`

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



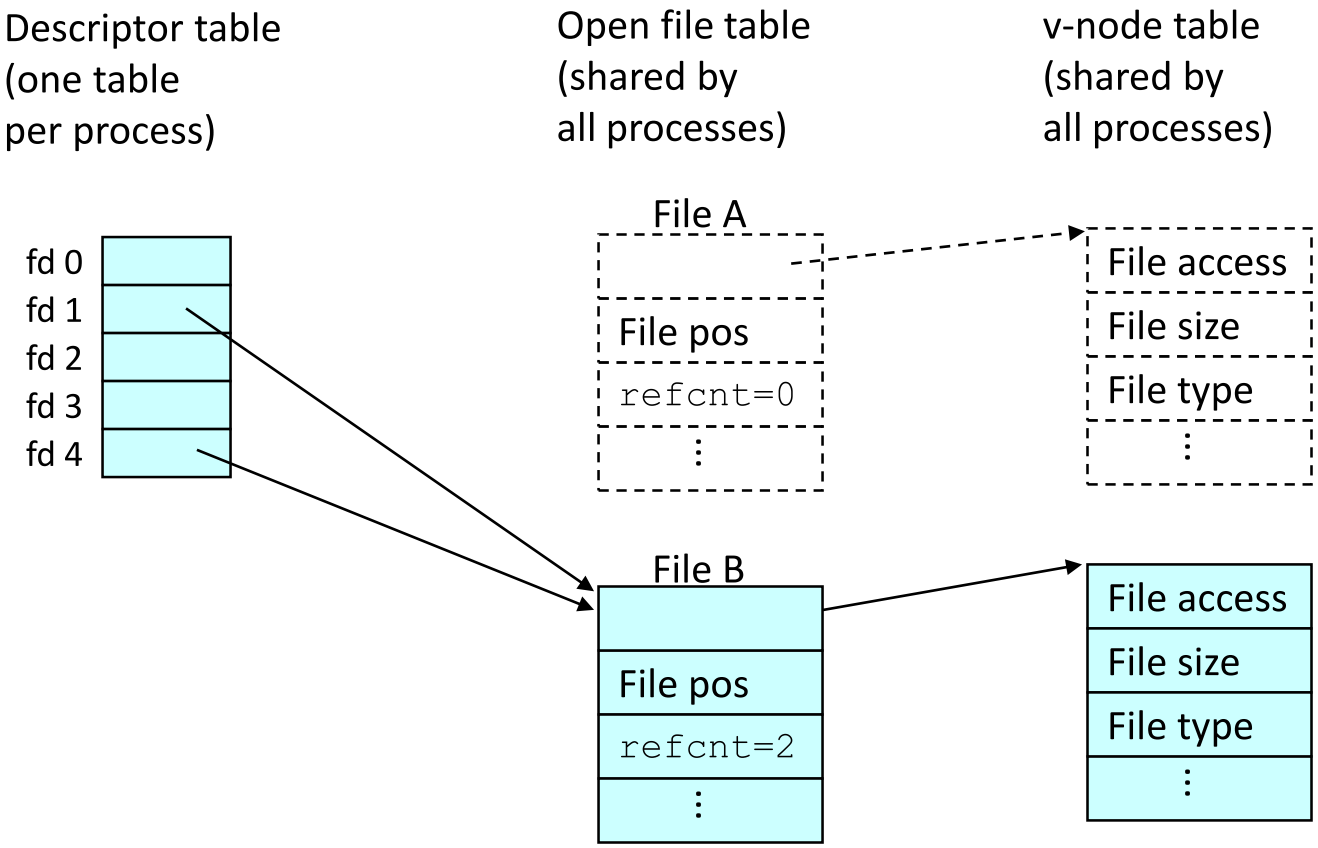
I/O Redirection Example

- Before calling `dup2(4,1)`, stdout (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.



I/O Redirection Example (cont)

- After calling `dup2(4,1)`, stdout is now redirected to the disk file pointed at by descriptor 4.



Standard I/O Functions

- The C standard library (`libc.a`) 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`)

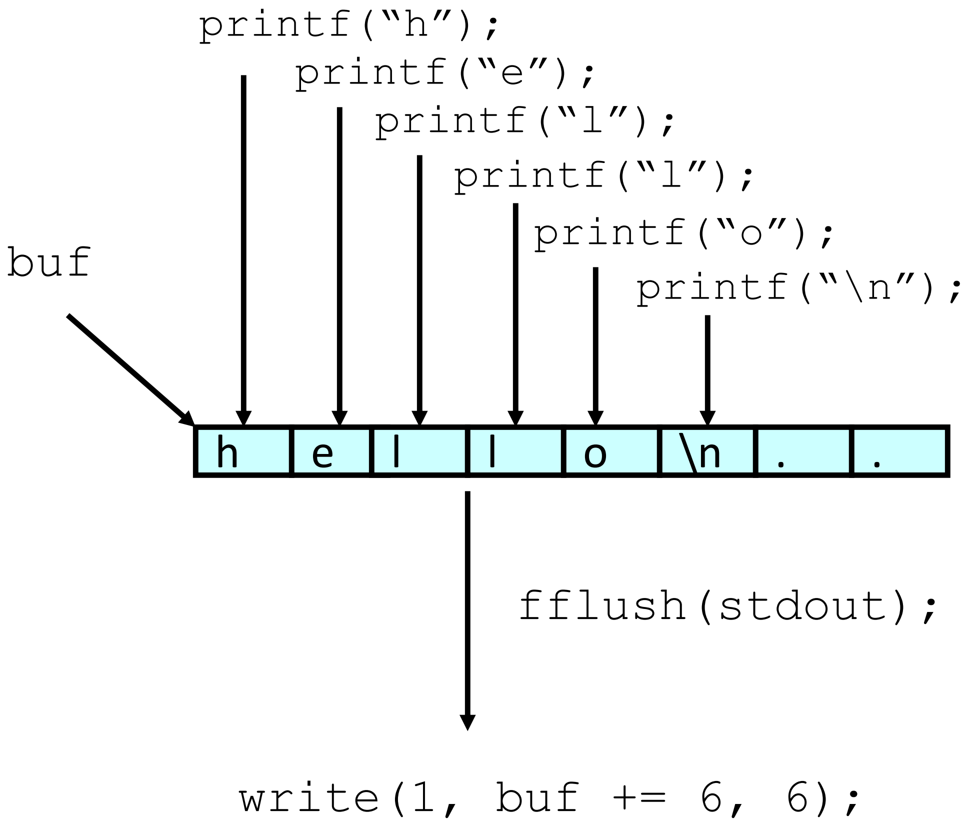
Standard I/O Streams

- 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



Standard I/O Buffering in Action

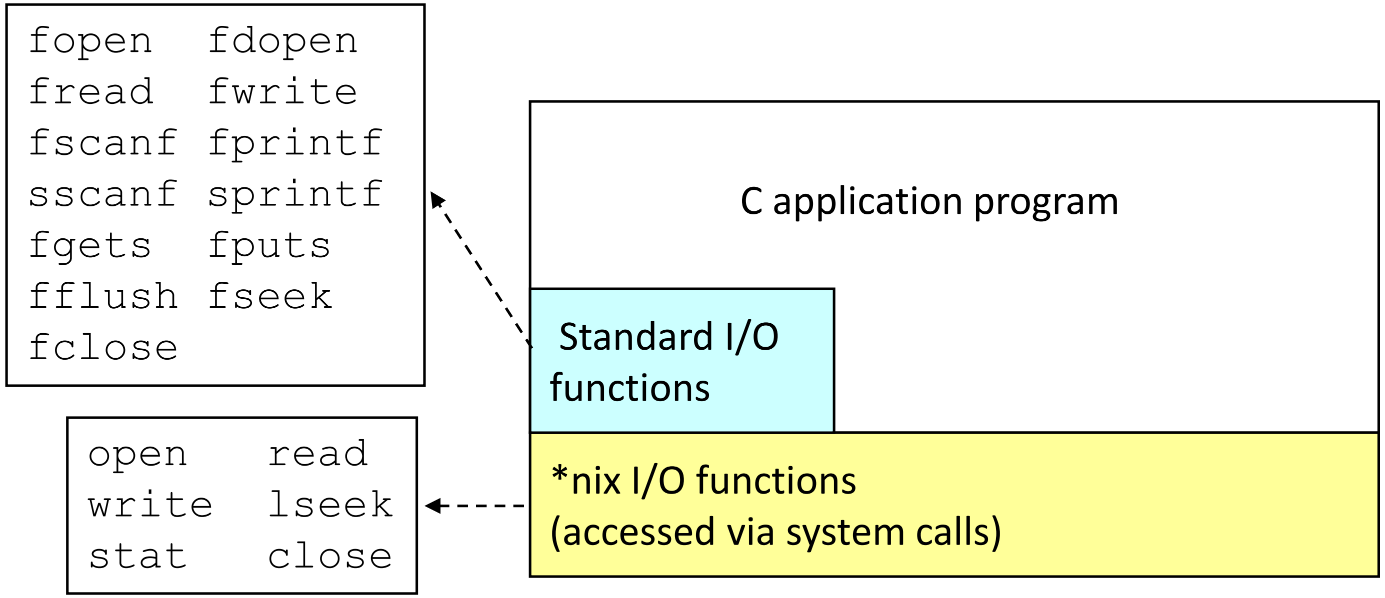
- You can see this buffering in action for yourself, using the always fascinating *nix `strace` program:

```
#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)                                 = ?
```

*nix 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 the standard I/O and RIO 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

Pros and Cons of Standard I/O (cont)

- Restrictions on streams:
  - Restriction 1: input function cannot follow output function without intervening call to `fflush`, `fseek`, `fsetpos`, or `rewind`.
    - Latter three functions all use `lseek` to change file position.
  - Restriction 2: output function cannot follow an input function with intervening call to `fseek`, `fsetpos`, or `rewind`.
- Restriction on sockets:
  - You are not allowed to change the file position of a socket.

Pros and Cons of Standard I/O (cont)

- Workaround for restriction 1:
  - Flush stream after every output.
- Workaround for restriction 2:
  - Open two streams on the same descriptor, one for reading and one for writing:

```
FILE *fpin, *fpout;

fpin = fdopen(sockfd, "r");
fpout = fdopen(sockfd, "w");
```

  - However, this requires you to close the same descriptor twice:

```
fclose(fpin);
fclose(fpout);
```

  - Creates a deadly race in concurrent threaded programs!

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 *nix I/O
  - When you need to fetch file metadata.
  - In rare cases when you need absolute highest performance.

Reference

- CMU Unix File system Course

http://csapp.cs.cmu.edu/2e/ch10-preview.pdf

Conclusion

- We have started on File I/O in Linux

- We have talked a lot on different file IO functions

- We will start on Linux Programming on IPC next week

- Reading Assignment: Chapter 5, 6, 7 in your textbook,
  CMU Robust I/O in our supplementary materials