I/O and File System

System level: below standard level

```c
#include <stdio.h>
int main(void) {
    FILE *fp = fopen("output.txt", "w");
    if (!fp) {
        perror("output.txt");
        return 1;
    }
    fputs("baby shark (do doo dooo)\n", fp);
    if (fclose(fp)) {
        perror("output.txt");
        return 1;
    }
    return 0;
}
```

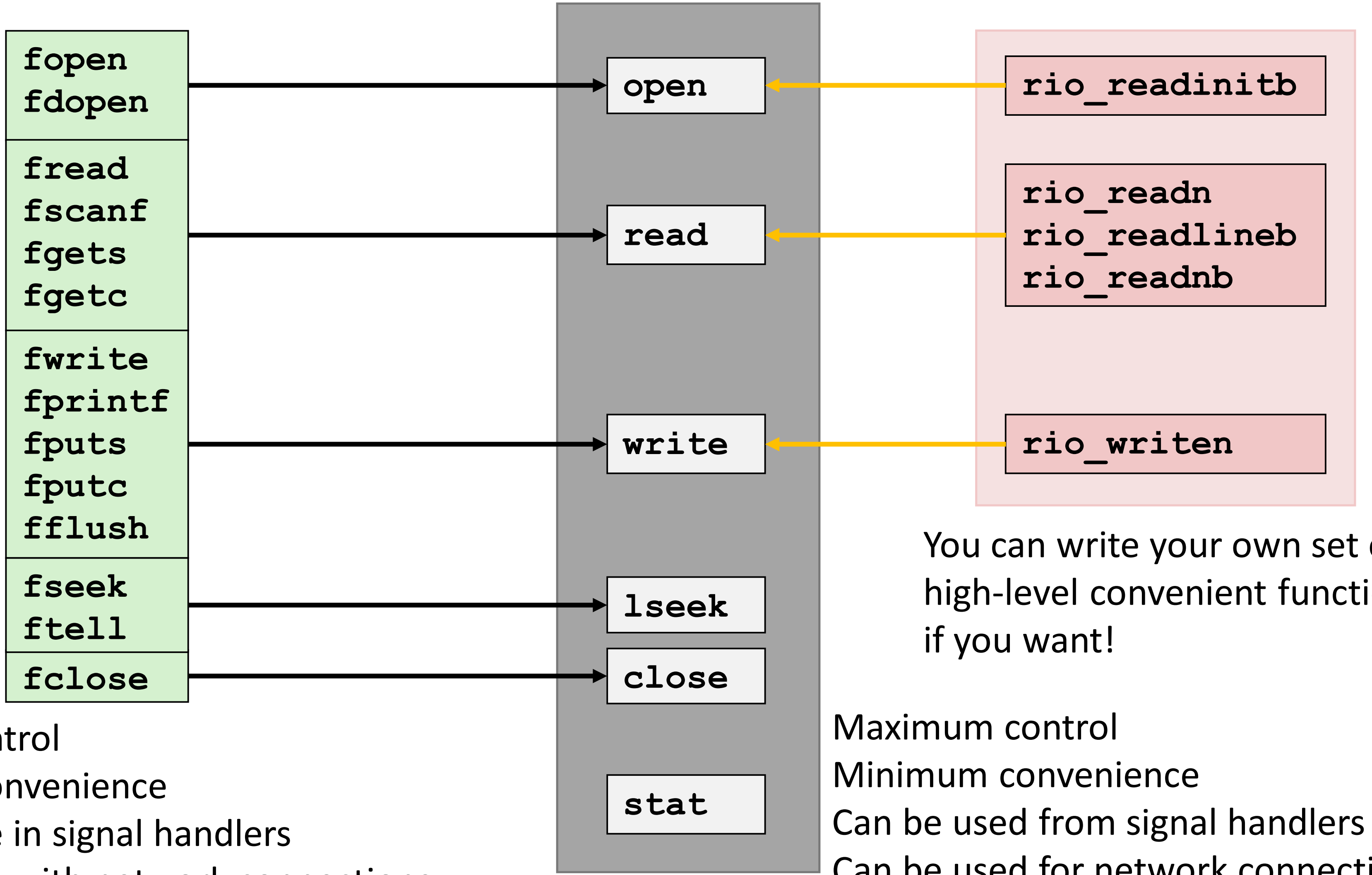
```c
FILE *fopen(const char *fname,
            const char *mode) {
    int fd = open(fname,
                  __mode2flags(mode),
                  DEFFILEPERMS);
    if (fd == -1) {
        return NULL;
    }
    return fdopen(fd, mode);
}
```

```c
int fputs(const char *s, FILE *fp) {
    size_t n = strlen(s);
    while (n > 0) {
        ssize_t written =
            write(fp->fd, s, n);
        if (written < 0) return EOF;
        n -= written;
        s += written;
    }
    return 0;
}
```

```c
int fclose(FILE *fp) {
    int rv = close(fp->fd);
    __ffree(fp);
    return rv;
}
```

```asm
.globl close
close:
    mov $3, %eax
    syscall
    cmp $-4096, %rax
    jae __syscall_error
    ret
```

Why do we have two sets?



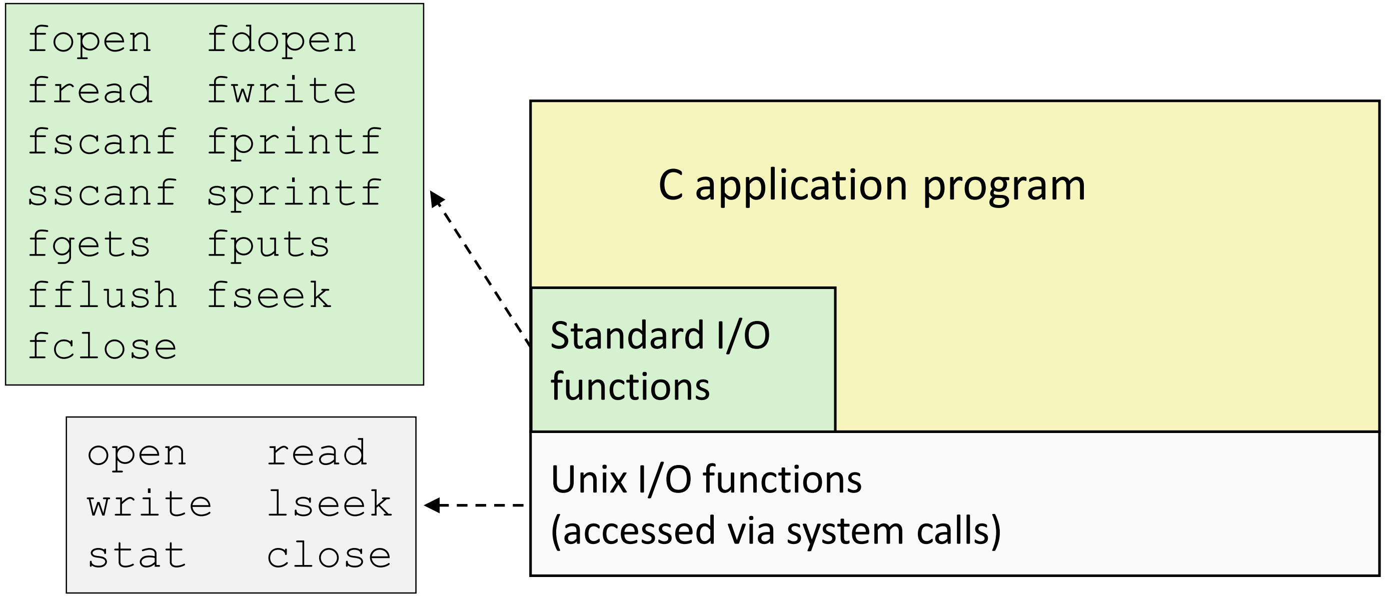
You can write your own set of high-level convenient functions if you want!

Less control
More convenience
Not safe in signal handlers
Not safe with network connections

Maximum control
Minimum convenience
Can be used from signal handlers
Can be used for network connections

Unix I/O and C Standard I/O

- Two sets: system-level and C level



Outline

- Unix I/O
- Metadata
- Standard I/O

Unix I/O Overview

- A Linux *file* is a sequence of *m* bytes:
  - $B_0, B_1, \ldots., B_k, \ldots., B_{m-1}$
  - Each *file* has some kind of low-level name: inode number

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

- Cool fact: Kernel data structures are exposed as files
  - **`cat /proc/$$/status`**
  - **`ls -l /proc/$$/fd/`**

Unix I/O Overview

- Two abstractions: File and Directory
- File
  - filename: human readable
  - low-level name: inode number
- Directory
  - low-level name: inode number
  - contents are quite specific: pairs of (filename, inode number) or (subdirectory, inode number)

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()`**

I/O Example

- > echo hello > foo
- > cat foo

  hello
- strace: trace every system call made by a program when it runs
- > strace cat foo

```
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
```

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()`**

File Types

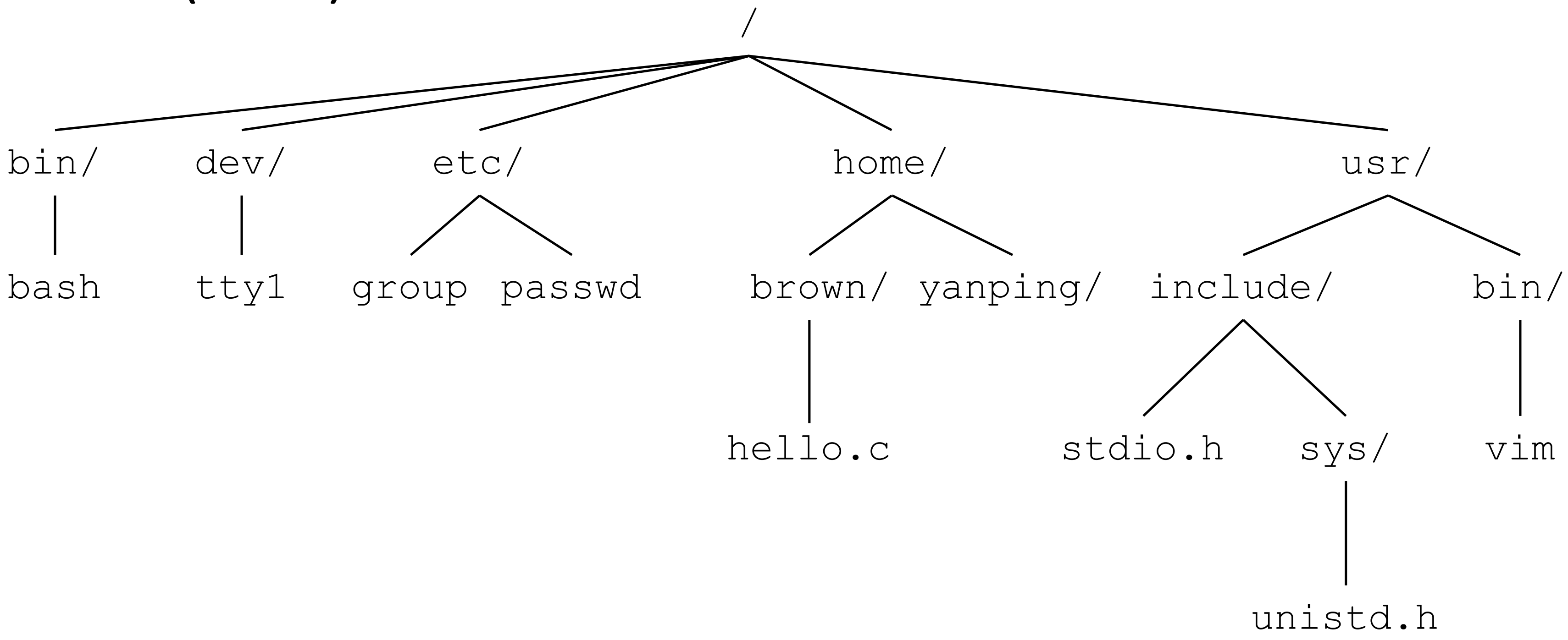
- Each file has a *type* indicating its role in the system
  - *Regular file:* Contains arbitrary data
  - *Directory:* Index for a related group of files
  - *Socket:* For communicating with a process on another machine

Directories

- Directory consists of an array of *links*
  - Each link maps a *filename* to a file
  - Pairs of (*filename*, inode number)
- Each directory contains at least two entries
  - `.` (dot) is a link to itself
  - `..` (dot dot) is a link to *the parent directory* in the *directory hierarchy* (next slide)
- Commands for manipulating directories
  - **`mkdir`**: create empty directory
  - **`ls`**: view directory contents
  - **`rmdir`**: delete empty directory

Directory Hierarchy

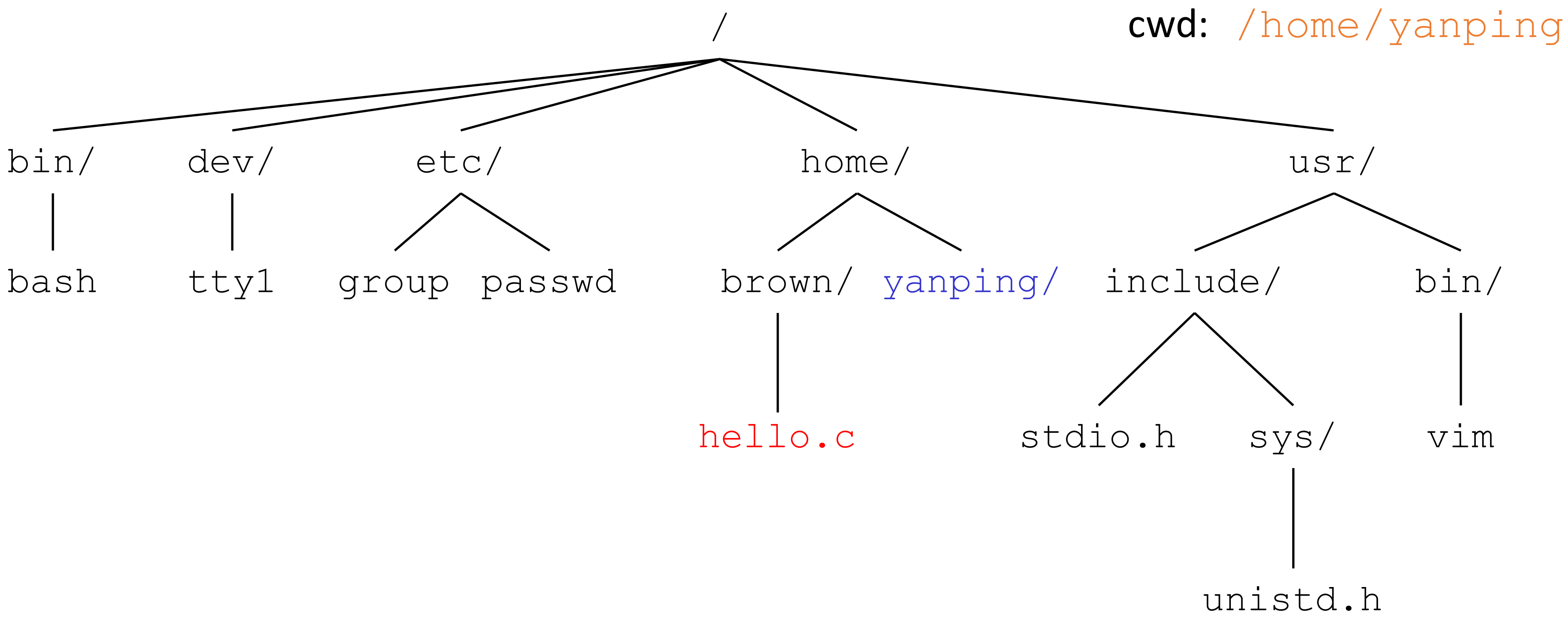
- All files are organized as a hierarchy anchored by root directory named / (slash)



- Kernel maintains *current working directory (cwd)* for each process
  - Modified using the **cd** command

Pathnames

- Locations of files in the hierarchy denoted by *pathnames*
  - *Absolute pathname* starts with '/' and denotes path from root
    - **`/home/brown/hello.c`**
  - *Relative pathname* denotes path from current working directory
    - **`../home/brown/hello.c`**



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

I/O: File Descriptor

- File descriptor:
  - an integer
  - like a “pointer” to an object of type file; you can call “methods” to access the file: read(), write()

I/O: File Descriptor

```
int main( ) {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
          write(2, note, strlen(note));
          exit(1);
    }
  return(0);
}
```

I/O: File Descriptor:

- File descriptor: when you first open a file, the file descriptor is 3
- Each running process already has three files open:
  - standard input (which the process can read to receive input)
  - standard output (which the process can write to in order to dump information to the screen)
  - standard error (which the process can write error messages to)
  - These are represented by file descriptors 0, 1, and 2, respectively.

Data Structures

```
// Per-process state
struct proc {
  uint sz;                      // Size of process memory (bytes)
  pde_t* pgdir;                 // Page table
  char *kstack;                 // Bottom of kernel stack for this process
  enum procstate state;         // Process state
  int pid;                      // Process ID
  struct proc *parent;          // Parent process
  struct trapframe *tf;         // Trap frame for current syscall
  struct context *context;      // swtch() here to run process
  void *chan;                   // If non-zero, sleeping on chan
  int killed;                   // If non-zero, have been killed
  struct file *ofile[NOFILE];   // Open files
  struct inode *cwd;            // Current directory
  char name[16];                // Process name (debugging)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
```

Data Structures

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

Closing Files

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

```
if (close(fd) < 0) {
   fprintf(stderr, "%s: write error: %s",
           filename, strerror(errno));
   exit(1);
}
```

- 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, 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 the 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!

Writing Files

- Writing a file copies bytes 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!

Today

- Unix I/O
- Metadata
- Standard I/O

File Metadata

- Each file has an inode that stores *Metadata* (data about file data)
- maintained by kernel

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Simplified Ext2 Inode

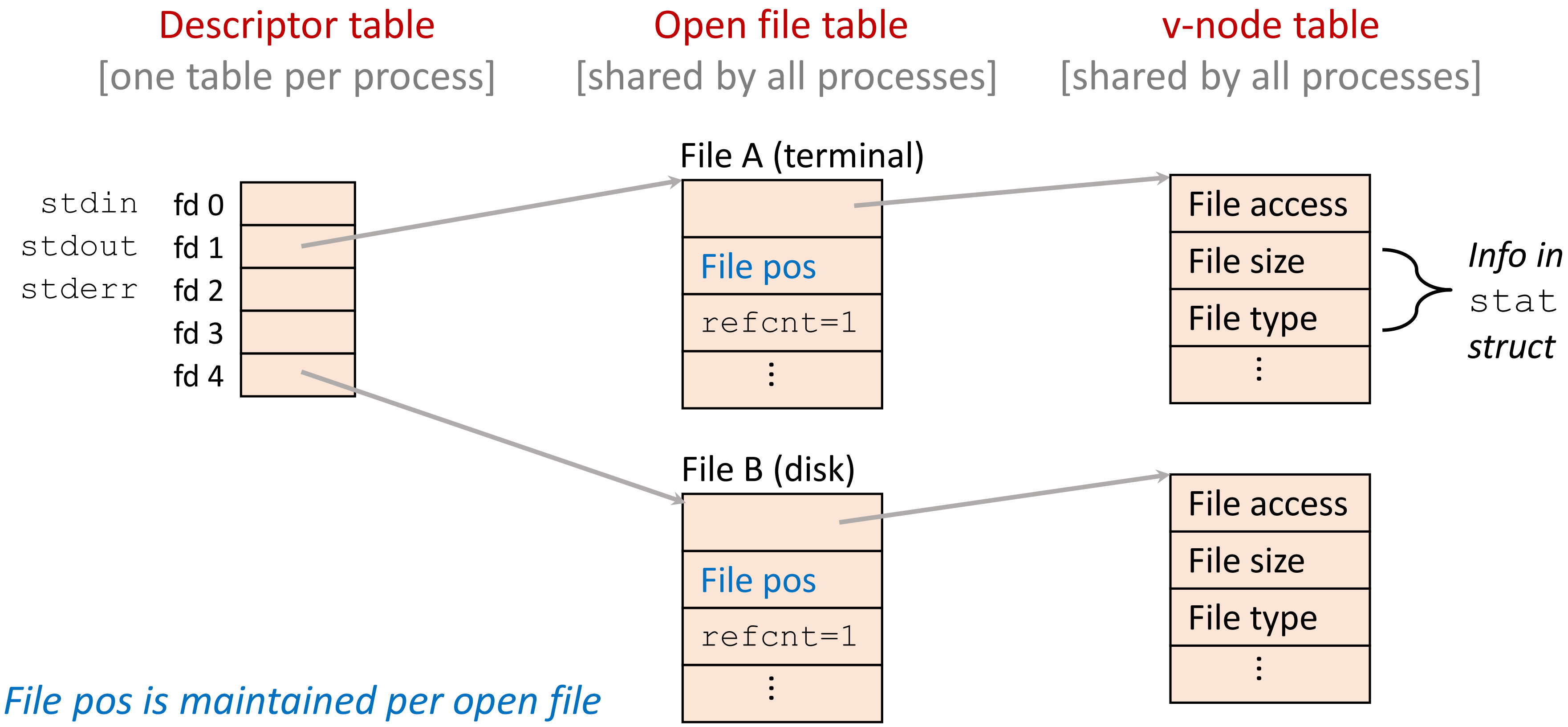
File Metadata

- *Metadata* can be retrieved by using "stat" (or related) calls, which returns a *stat* structure.

```
/* 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 */
};
```

How the Unix Kernel Represents Open Files

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



File pos is maintained per open 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