计算机系统基础12

I/O

Today

- **I/O Systems**
- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O

Overview

- **A computer's job is to process data**
  - Computer (CPU, cache, and memory)
  - Move data into and out of a system (between I/O devices and memory)
- **Challenges with I/O devices**
  - Different categories: storage, networking, displays, etc.
  - Large number of device drivers to support
  - Device driver run in kernel mode and can crash systems
    设备驱动程序

Overview (Cont.)

- **I/O management is a major component of operating system**
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent

- **Ports, busses, device controllers connect to various devices**

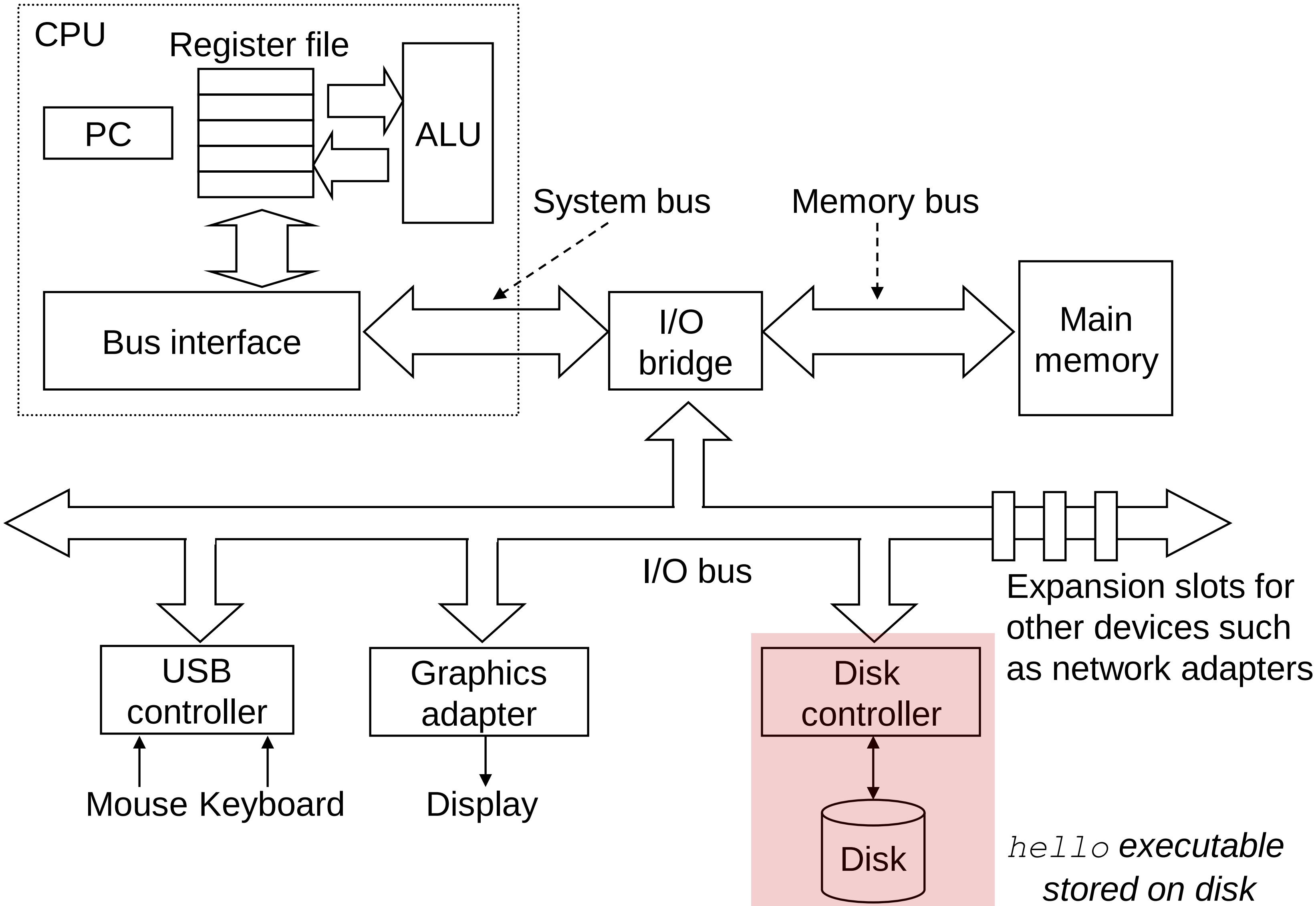
  端口 总线 设备控制器

- **Device drivers encapsulate device details**
  - Present uniform device-access interface to I/O subsystem

I/O Hardware

- **Incredible variety of I/O devices**
  - Storage
  - Transmission
  - Human-interface
- **Common concepts – signals from I/O devices interface with computer**
  - **Port** – connection point for device
  - **Bus** - daisy chain or shared direct access, e.g. PCI, PCIe
  - **Controller** – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board
    - Contains processor, microcode, private memory, bus controller, etc
      微码
      – Some talk to per-device controller with bus controller, microcode, memory, etc

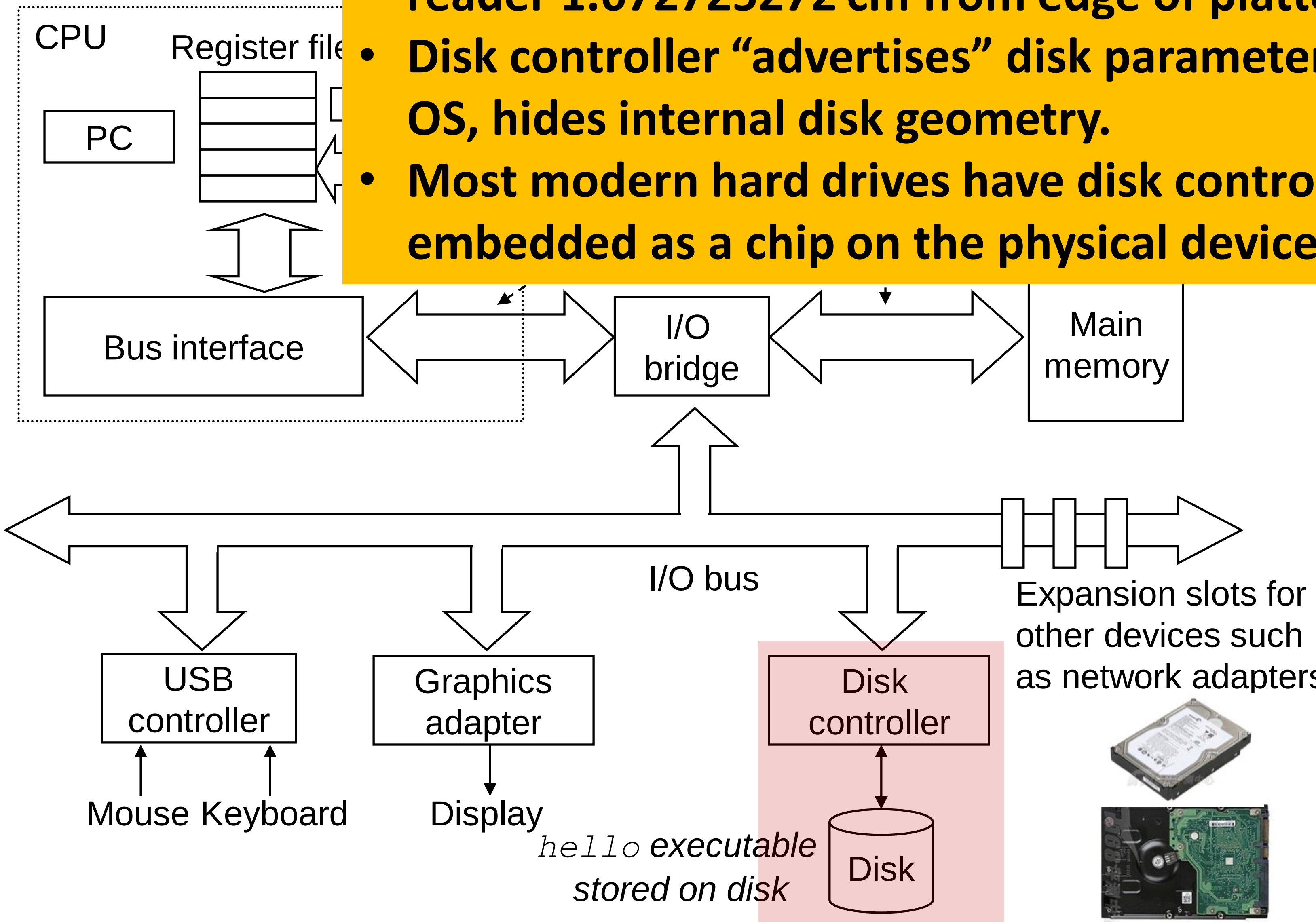
Typical Computer (PC) Today: HW Organization



Typical Comp

A disk controller:

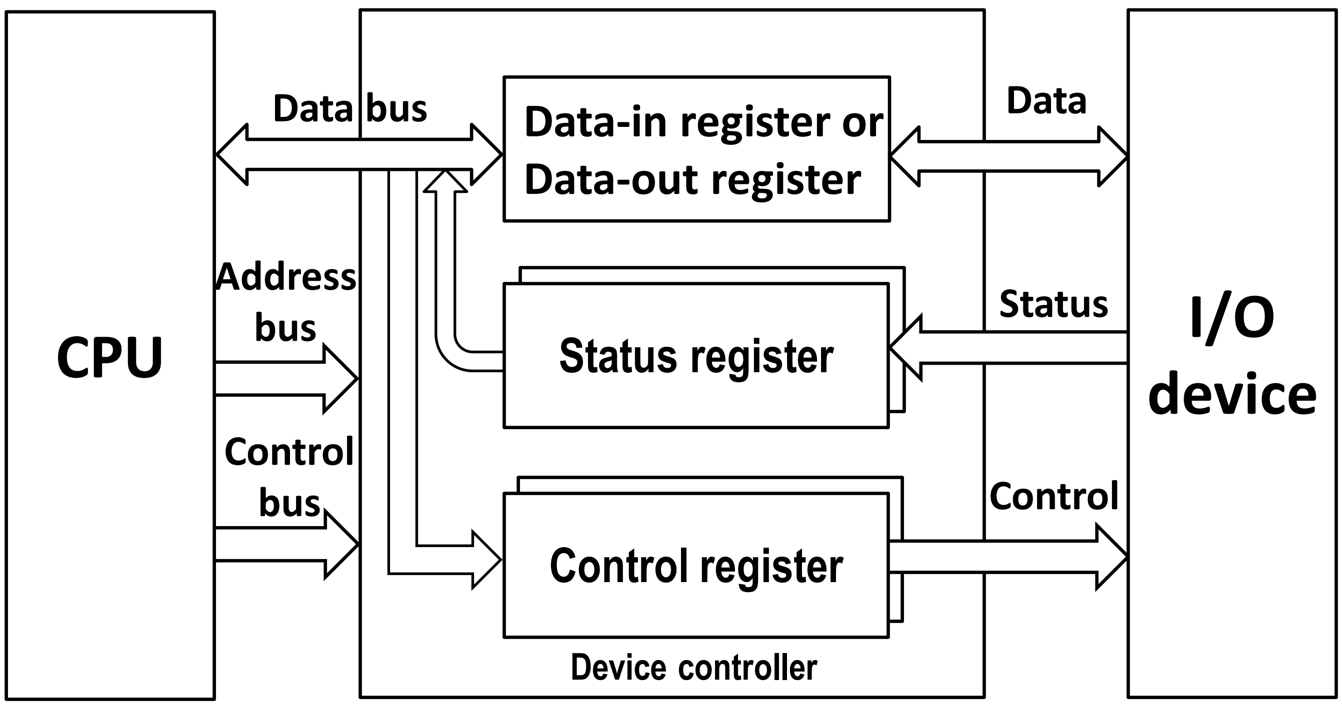
- **Translates "access sector 23" to "move head reader 1.672725272 cm from edge of platter"**
- **Disk controller "advertises" disk parameters to OS, hides internal disk geometry.**
- **Most modern hard drives have disk controller embedded as a chip on the physical device**



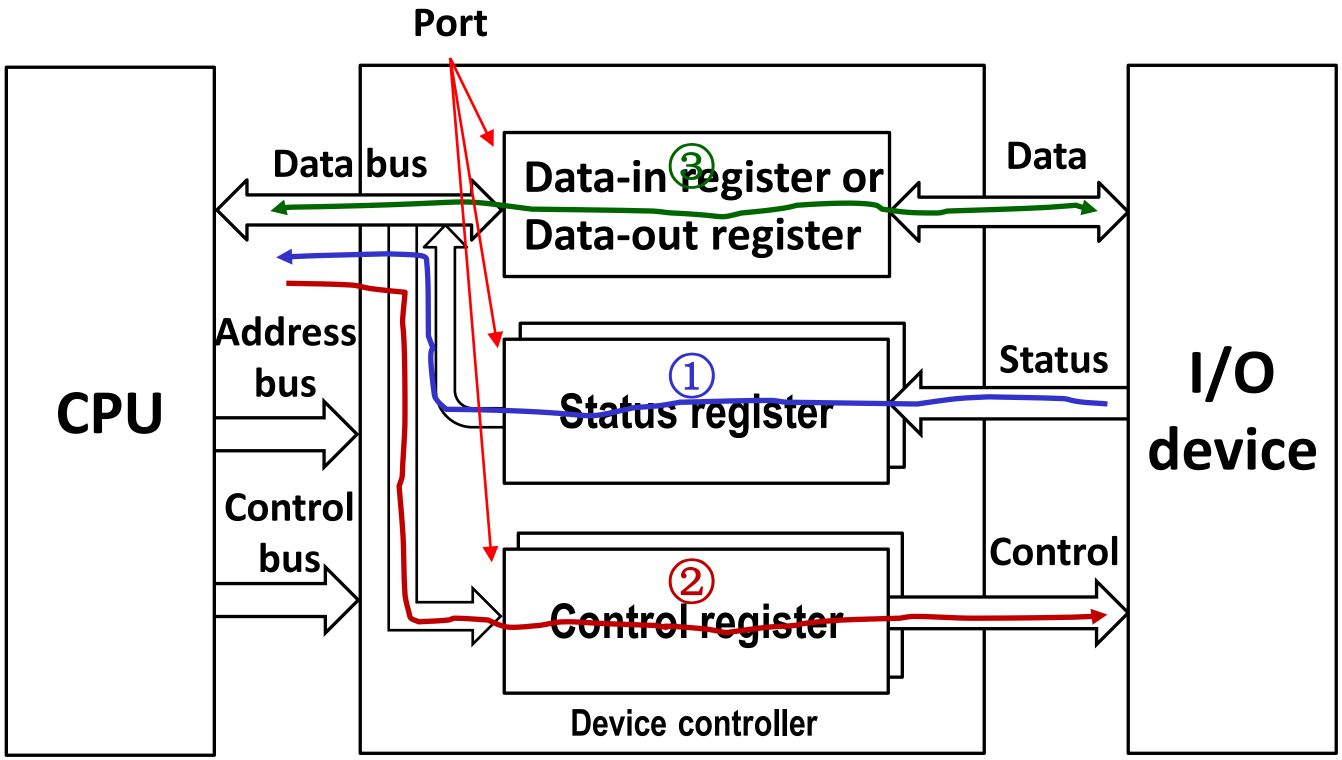
I/O Hardware (Cont.)

- **I/O instructions control devices**
- **Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution**
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO buffer
- **Devices have addresses, used by**
  - Direct I/O instructions
  - Memory-mapped I/O
    - Device data and command registers mapped to processor address space
    - Especially for large address spaces (graphics)

I/O Hardware (Cont.)



I/O Hardware (Cont.)



A typical sequence of I/O operations

① Read status information of the device from status registers

② Write commands to control registers for control device

③ Read data-in or write data-out register for exchange data with I/O device

Device Driver and I/O Instruction

- **The device drivers present a uniform device-access interface to the I/O subsystem , and encapsulate the details and oddities of different devices**
- **The driver designer should understand the device controller and the working principle of the device, including: which registers are accessible to the user in the device controller, the meaning of each bit in the control/status register. The driver performs I/O by accessing the I/O port control peripheral**
- **Access to I/O ports is done by I/O instructions, which are privileged instruction**

Three Types of I/O

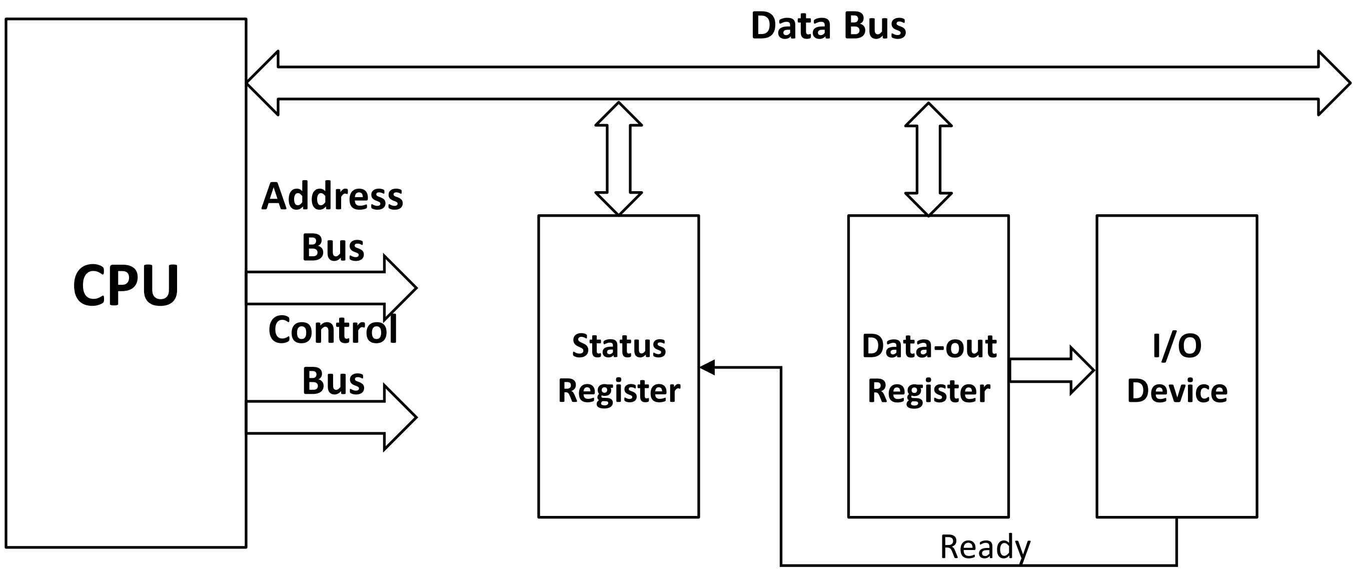
- Programmed I/O(Polling): continuous attention of the 查询 processor is required

- Interrupt driven I/O: processor launches I/O and can continue until interrupted

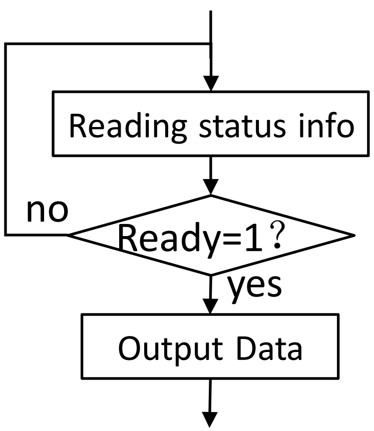
- Direct memory access(DMA): the dma module governs the exchange of data between the I/O unit and the main memory

Programmed I/O（Polling）

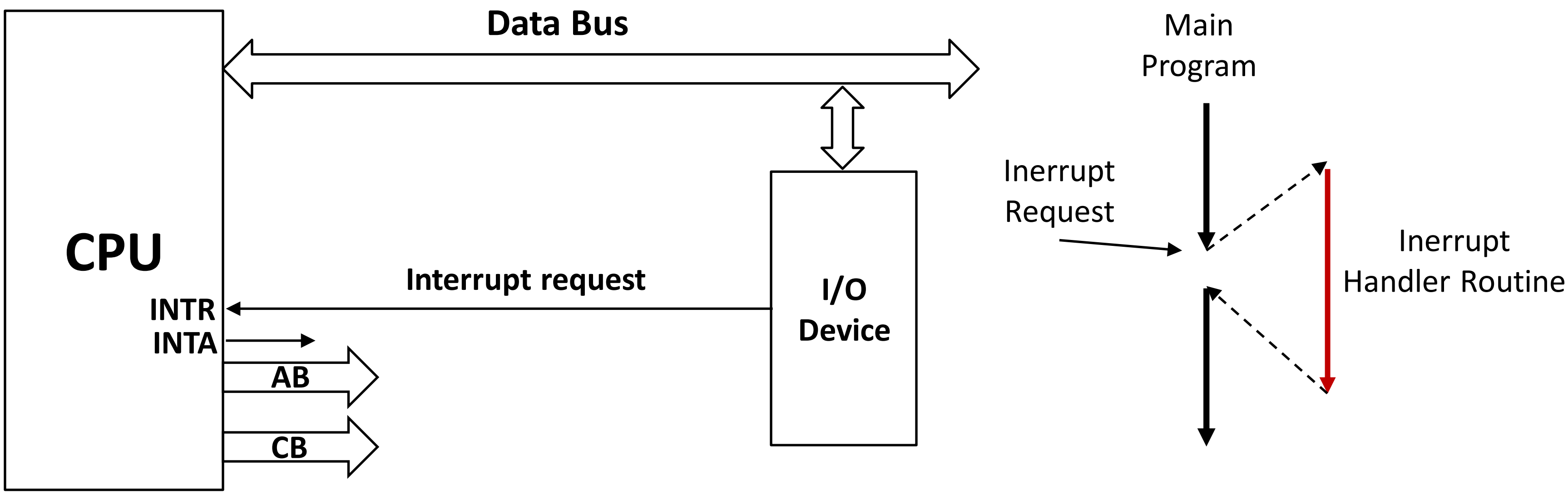
- **Output illustrates**



- The host reads the status register over and over until the ready bit becomes set
- If ready bit becomes set, the host writes data to data-out register so as to output data to the device
- If the device consumes data slowly the host is in busy-waiting state



Interrupts

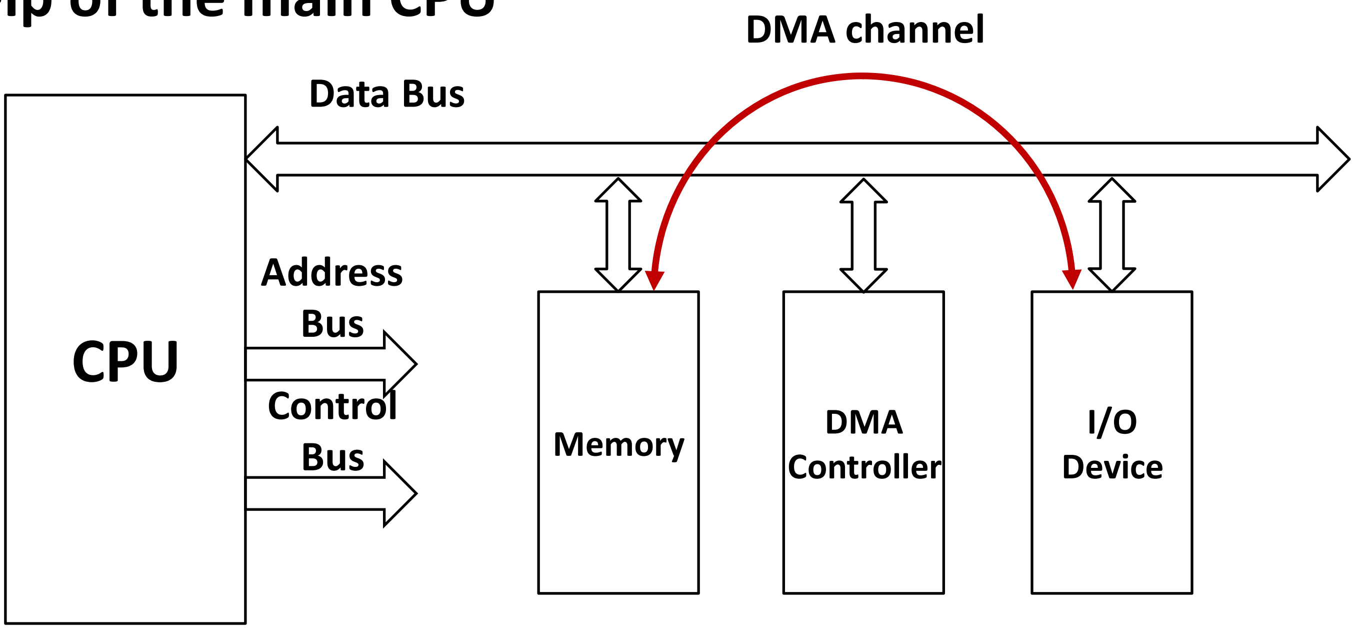


- An external pin INTR on the CPU chip triggered by I/O device
  - Checked by processor after each instruction
- Interrupt handler routine receives interrupts
  - Maskable to ignore or delay some interrupts
- Interrupt vector to dispatch interrupt to correct handler routine

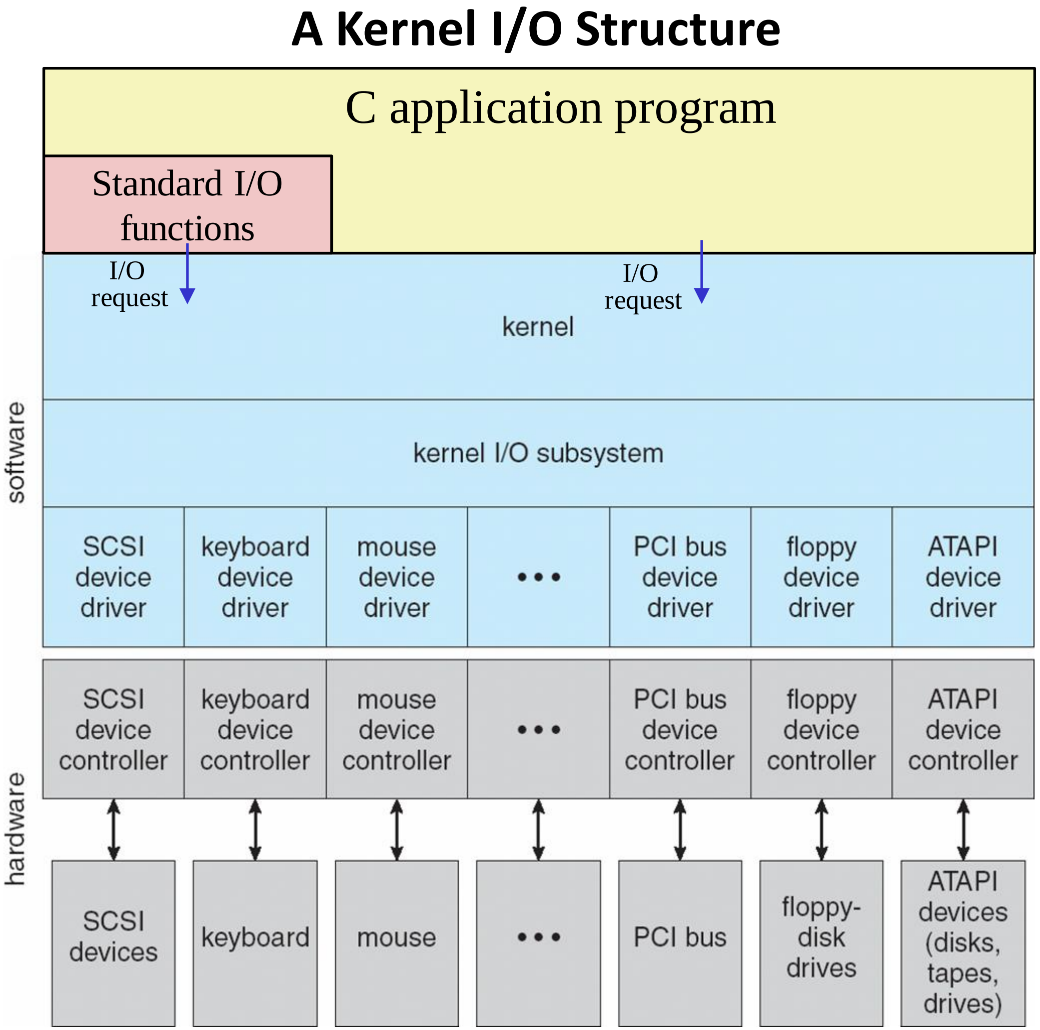
DMA（Direct Memory Access）

- Bypasses CPU to transfer data directly between I/O device and memory

  During data transfer period , the DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU



Architecture of Kernel I/O stack



Each OS has its own I/O subsystem structures and device driver frameworks

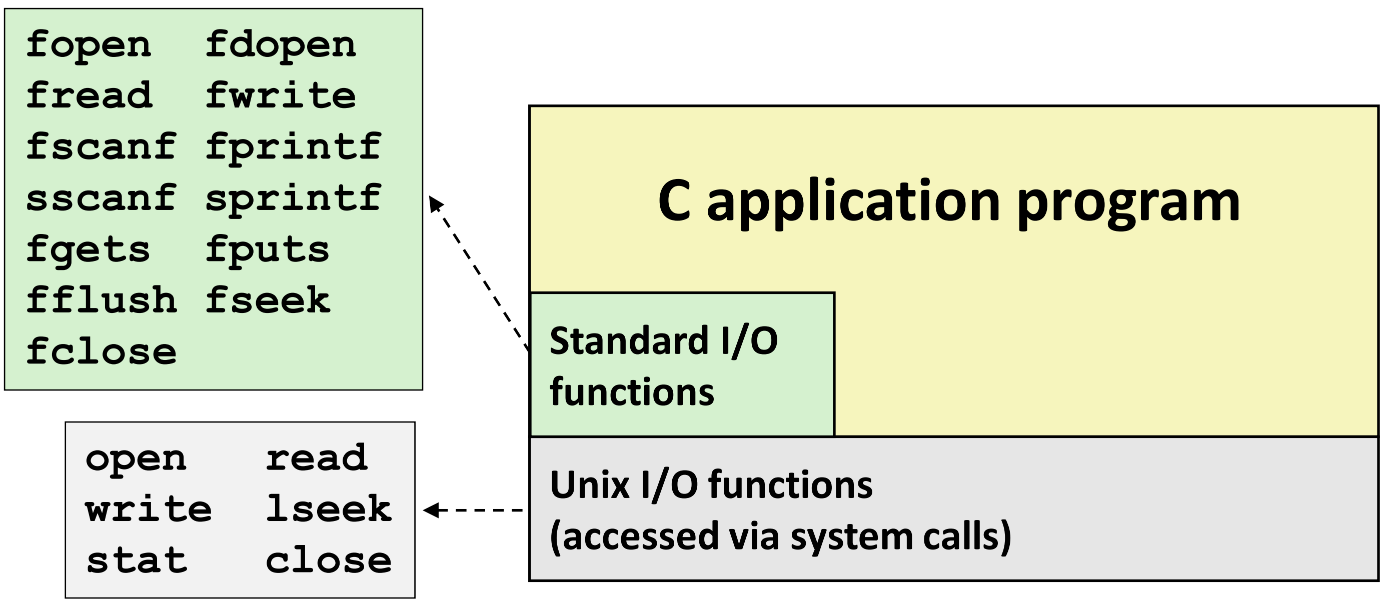
- **I/O system calls**
  1. Use the highest-level I/O functions
  2. Use raw system I/O
- **Device-driver layer hides differences among I/O controllers from kernel**
- **Controller – electronics that operate port, bus, device**
- **Devices vary in many dimensions**
  - Character-stream or block
  - Sequential or random-access
  - … …

Today

- I/O Systems
- **Unix I/O**
- Metadata, sharing, and redirection
- Standard I/O

Today: Unix I/O and C Standard I/O

- **Two sets: system-level and C level**

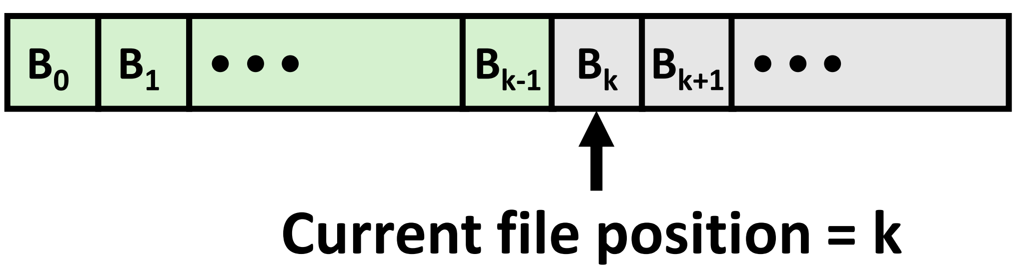


Unix I/O Overview

- 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()`



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

- **Other file types beyond our scope**
  - *Named pipes (FIFOs)*
  - *Symbolic links*
  - *Character and block devices*

Regular Files

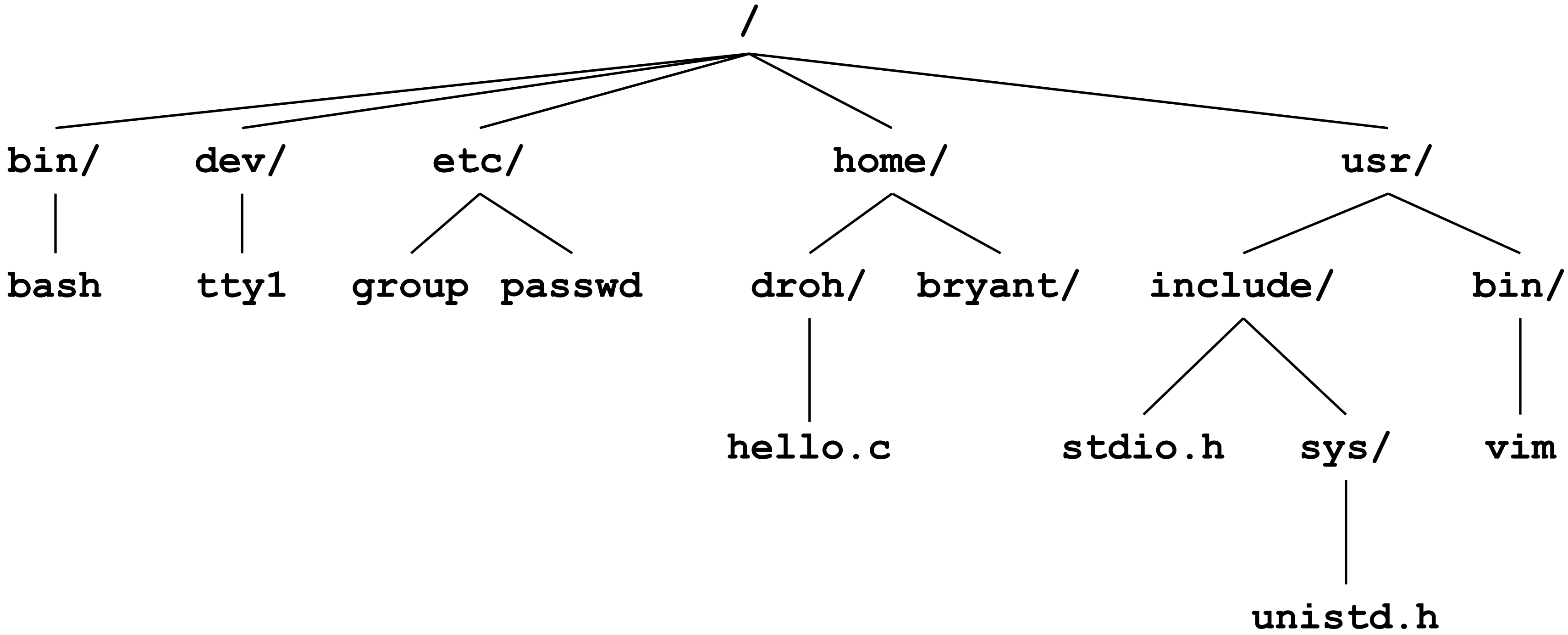
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't know the difference!
- **Text file is sequence of *text lines***
  - Text line is sequence of chars terminated by *newline char* (**'`\n`'**)
    - Newline is **`0xa`**, same as ASCII line feed character (LF)
- **End of line (EOL) indicators in other systems**
  - Linux and Mac OS: **'`\n`'** (**`0xa`**)
    - line feed (LF)
  - Windows and Internet protocols: **'`\r\n`'** (**`0xd 0xa`**)
    - Carriage return (CR) followed by line feed (LF)

Directories

- **Directory consists of an array of *links***
  - Each link maps a *filename* to a file
- **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

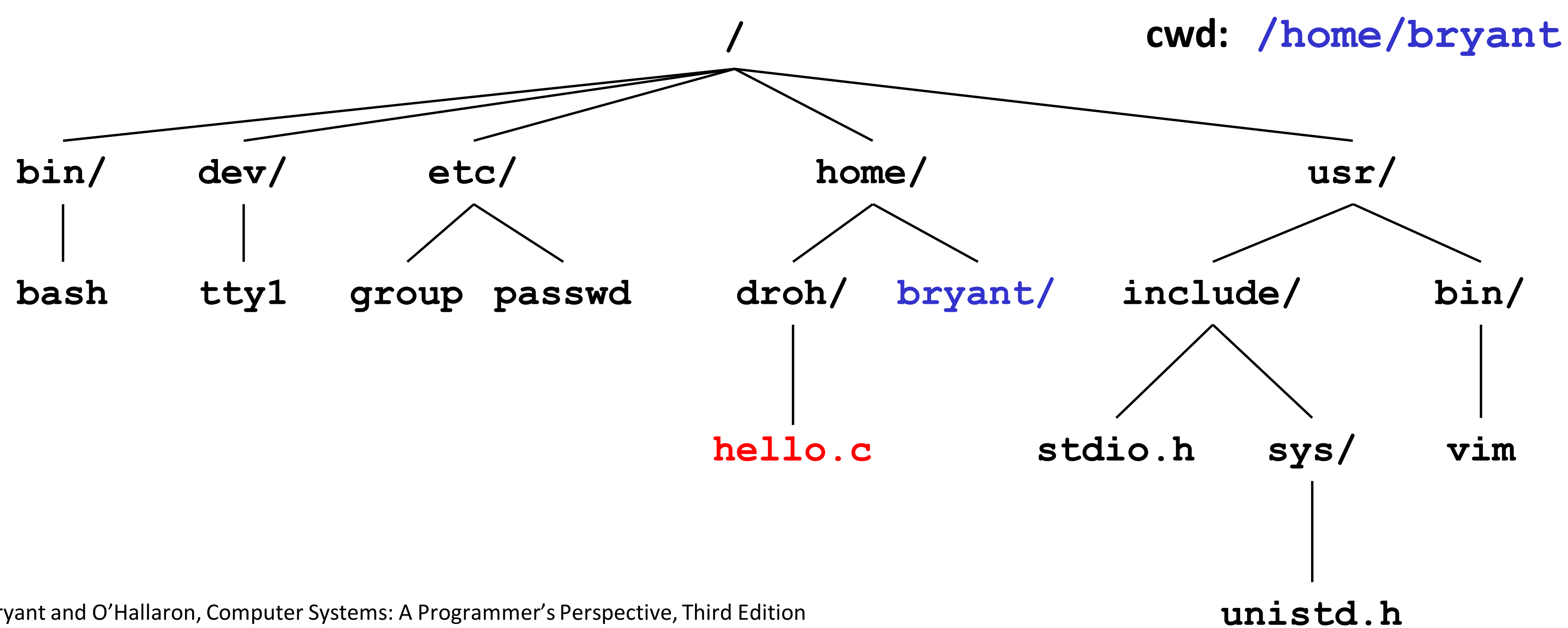
- All files are organized as a hierarchy anchored by root directory named / (slash)



- 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/droh/hello.c`**
  - *Relative pathname* denotes path from current working directory
    - **`../droh/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
- **Each process created by a Linux shell begins life with three open files associated with a terminal:**
  - 0: standard input (stdin)
  - 1: standard output (stdout)
  - 2: standard error (stderr)

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);
}
```

- 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`**
  - 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!
不足值

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!

Simple Unix I/O example

- Copying file to stdout, one byte at a time

```
#include "csapp.h"

int main(int argc, char *argv[])
{
    char c;
    int infd = STDIN_FILENO;
    if (argc == 2) {
        infd = open(argv[1], O_RDONLY, 0);
    }
    while(read(infd, &c, 1) != 0)
        write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

- Demo:

```
linux> strace ./showfile1_nobuf names.txt
```

On 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

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

- **Best practice is to always allow for short counts.**

Today

- I/O Systems
- Unix I/O
- **Metadata, sharing, and redirection**
- Standard I/O

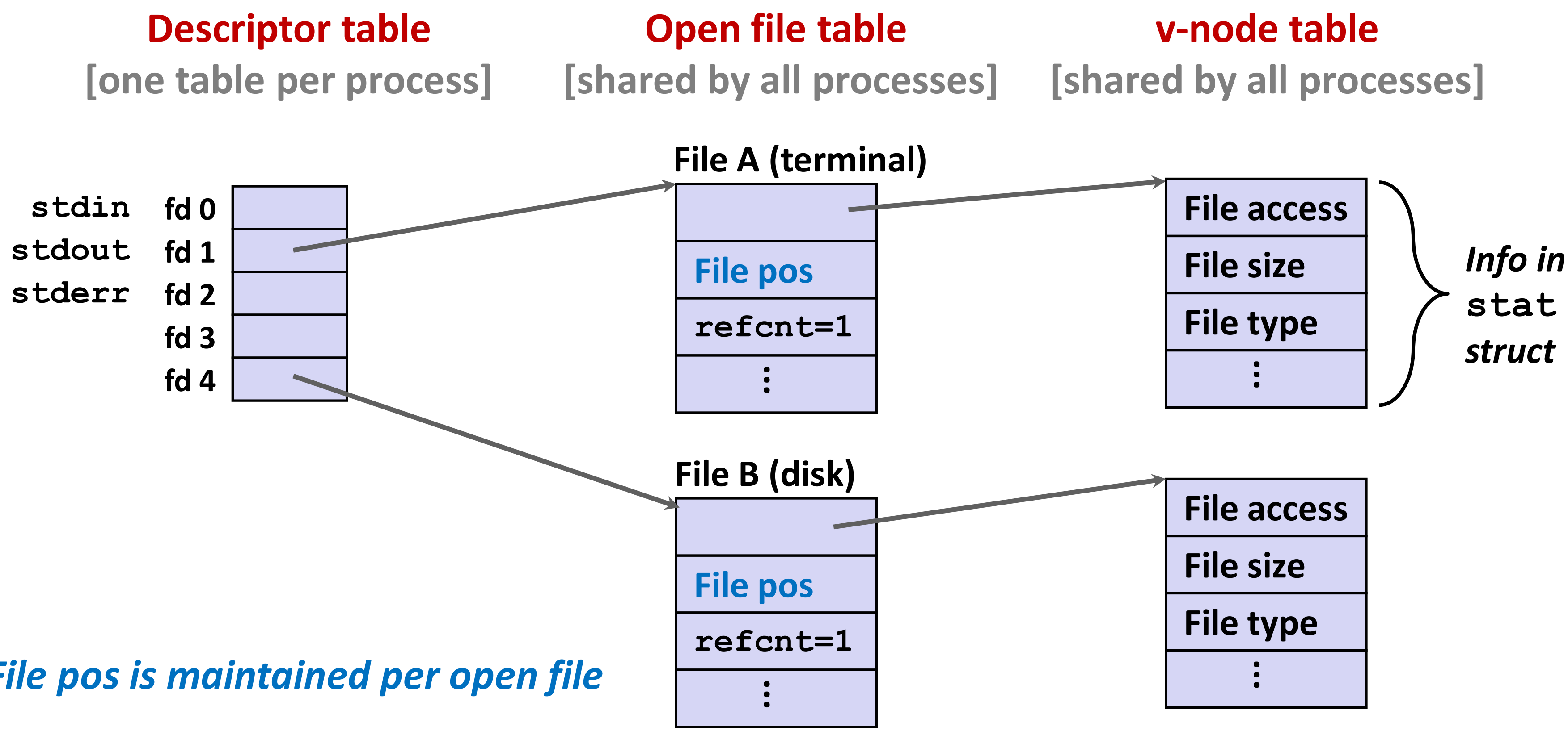
File Metadata

- *Metadata* is data about data, in this case file data
- Per-file metadata 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 */
};
```

How the Unix Kernel Represents Open Files

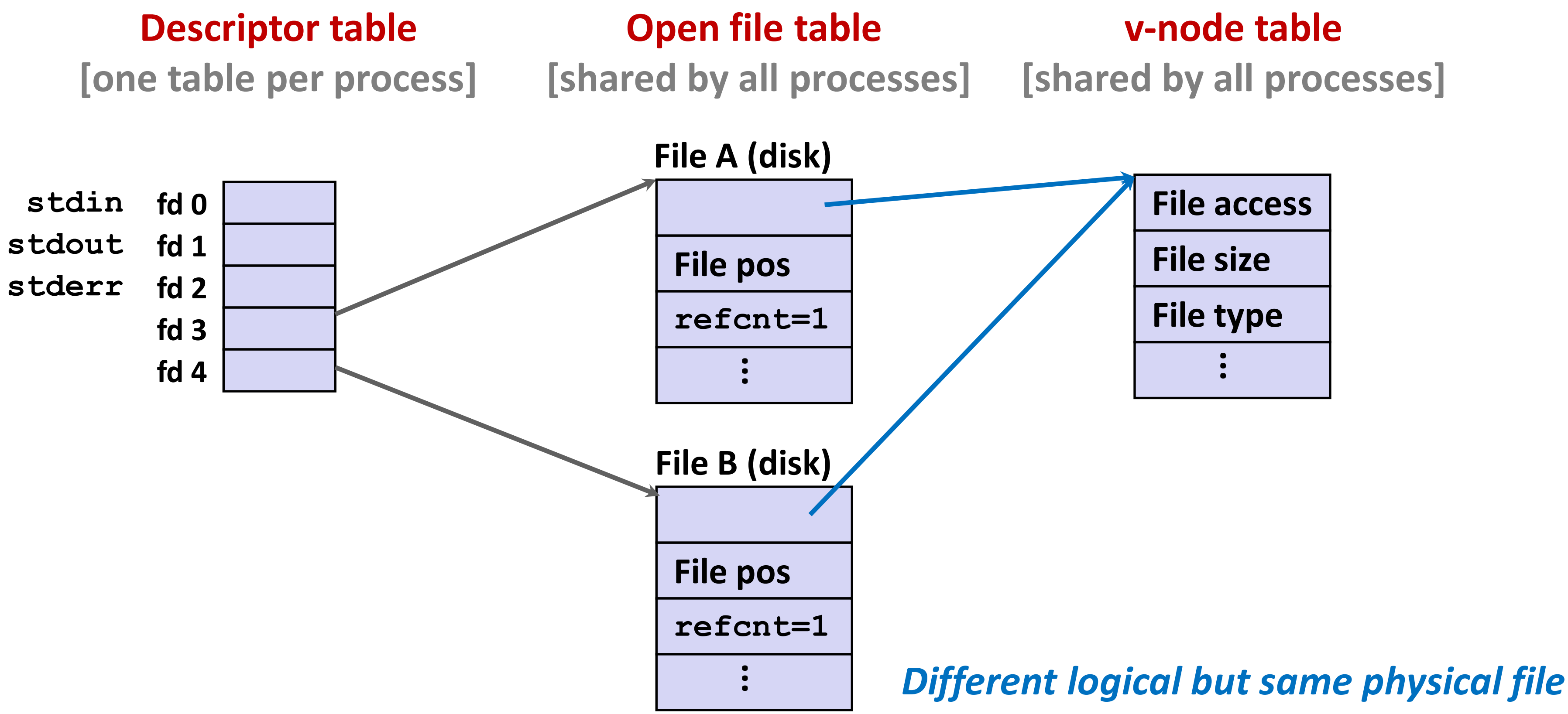
- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



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



Creating Processes

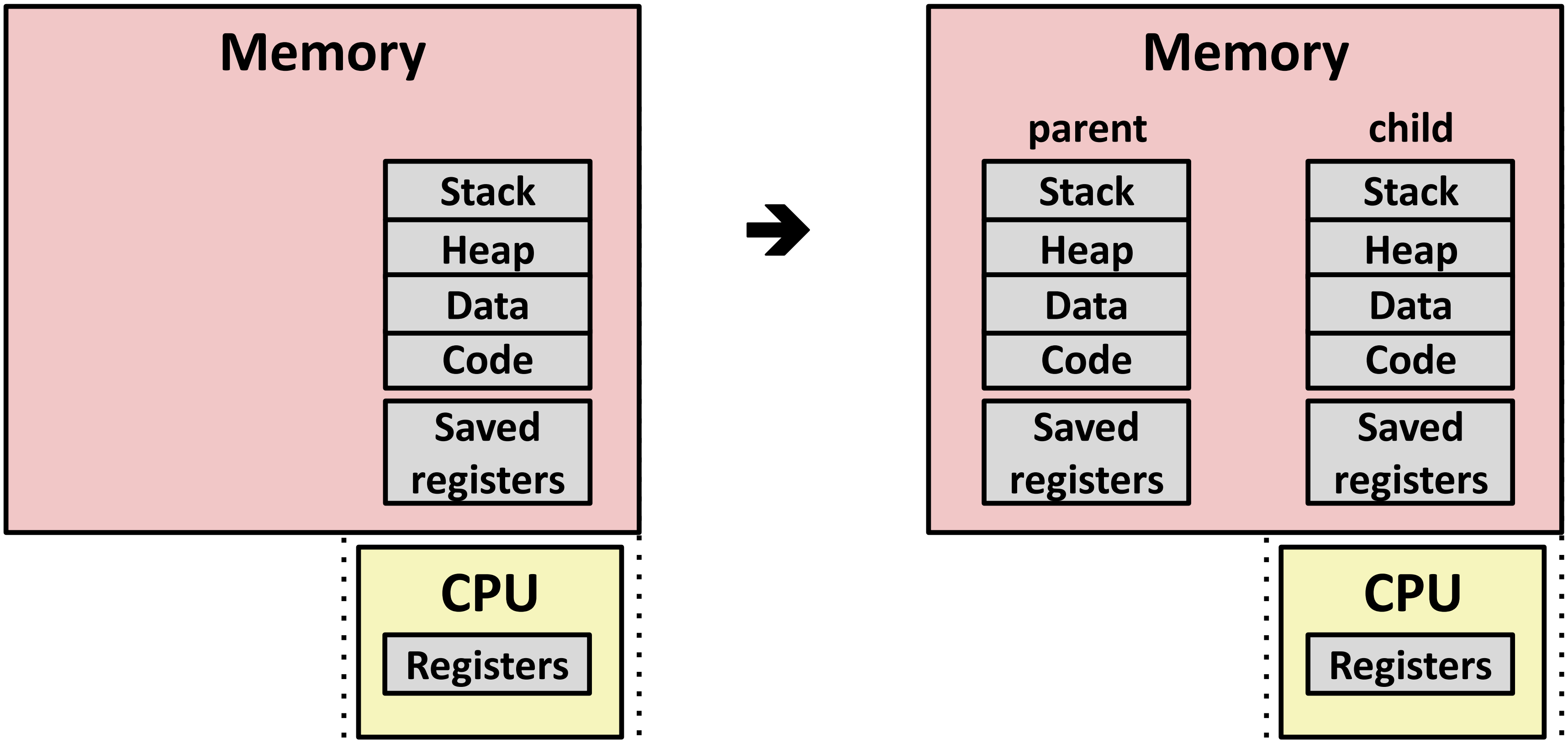
补充

- ***Parent process*** **creates a new running** ***child process*** **by calling `fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- **`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

Conceptual View of `fork` 补充



- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of parent or child

fork Example

补充

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child

```
linux> ./fork
parent: x=0
child : x=2
```

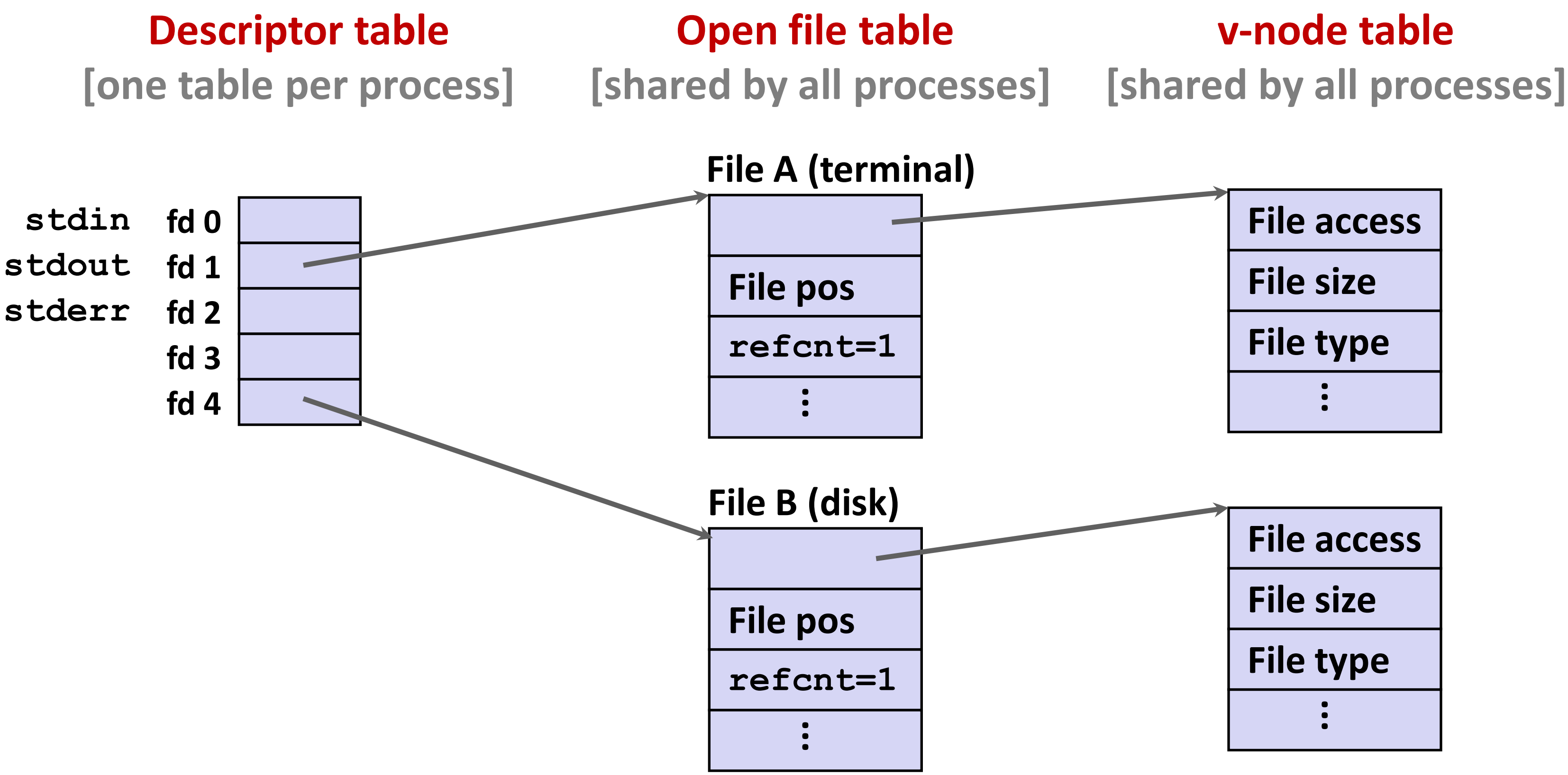
```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

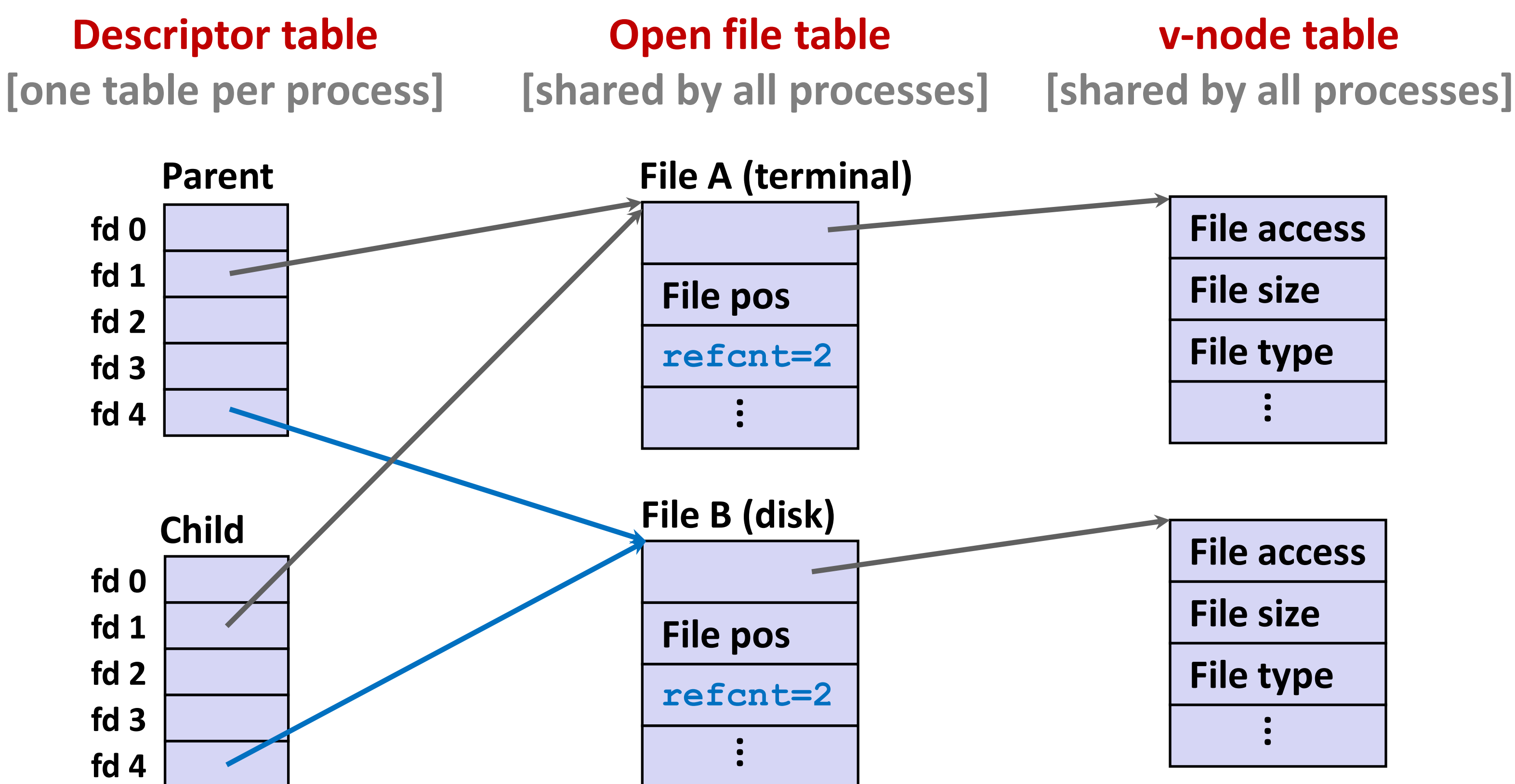
How Processes Share Files: `fork`

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



How Processes Share Files: `fork`

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



File is shared between processes

I/O Redirection

重定向

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

```
linux> 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(4,1)`

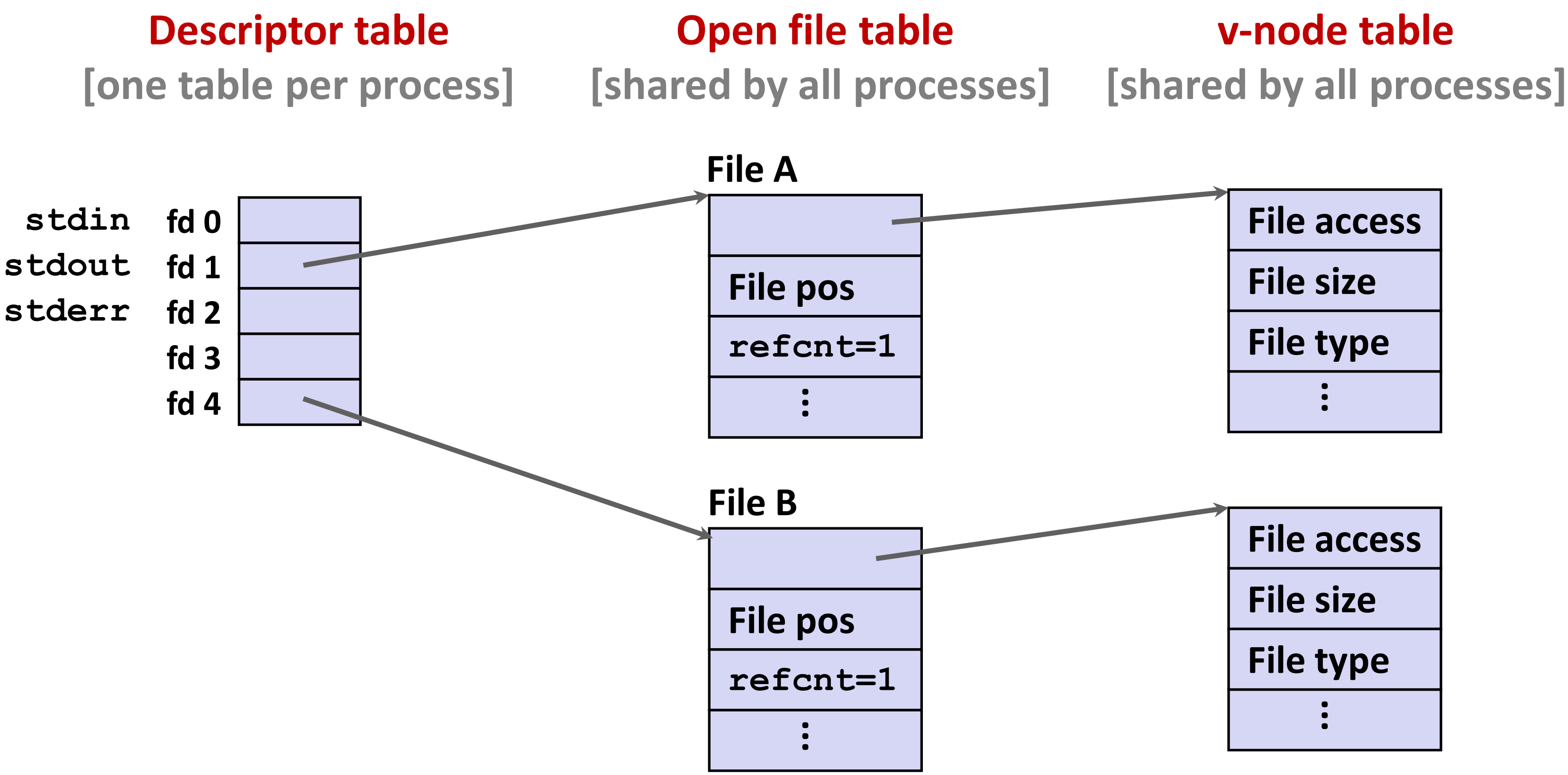
fd 0	
fd 1	a
fd 2	
fd 3	
fd 4	b

Descriptor table *after* `dup2(4,1)`

fd 0	
fd 1	b
fd 2	
fd 3	
fd 4	b

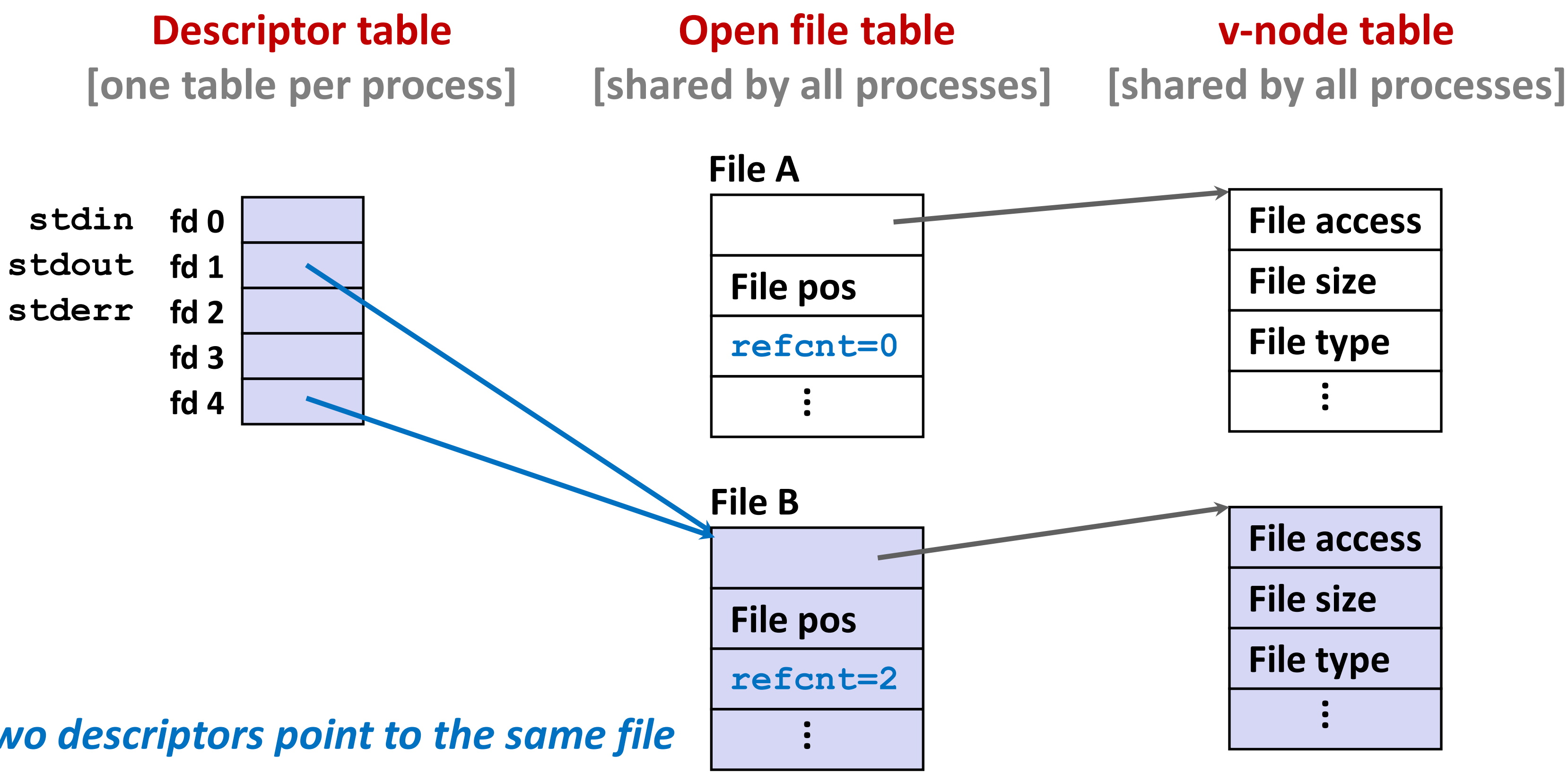
I/O Redirection Example

- **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



Two descriptors point to the same file

Warm-Up: 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"?**

Warm-Up: 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”?

Today

- I/O Systems
- Unix I/O
- Metadata, sharing, and redirection
- **Standard I/O**

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

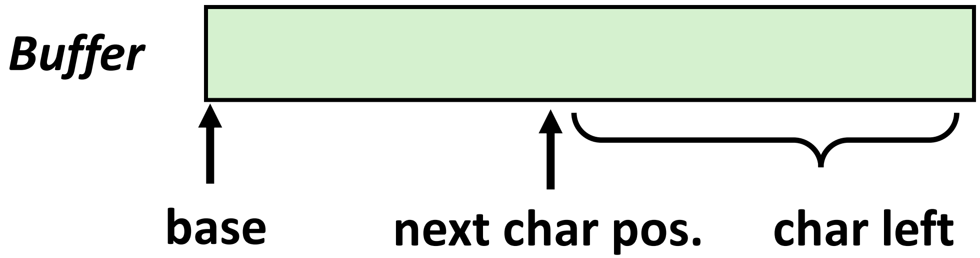
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - `stdin` (standard input)
  - `stdout` (standard output)
  - `stderr` (standard error)

```
#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");
}
```

Struct FILE

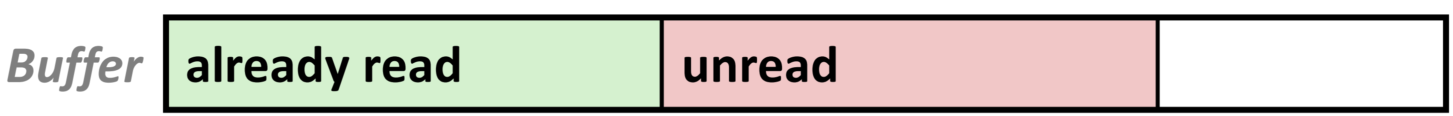
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory




```
/* stdio.h file */
typedef   struct _iobuf {
    int    cnt;        /* characters left */
    char   *ptr;       /* next characters position */
    char   *base;      /* location of buffers */
    int    flag;       /* mode of file access */
    int    fd;         /* file descripter */
} FILE
```

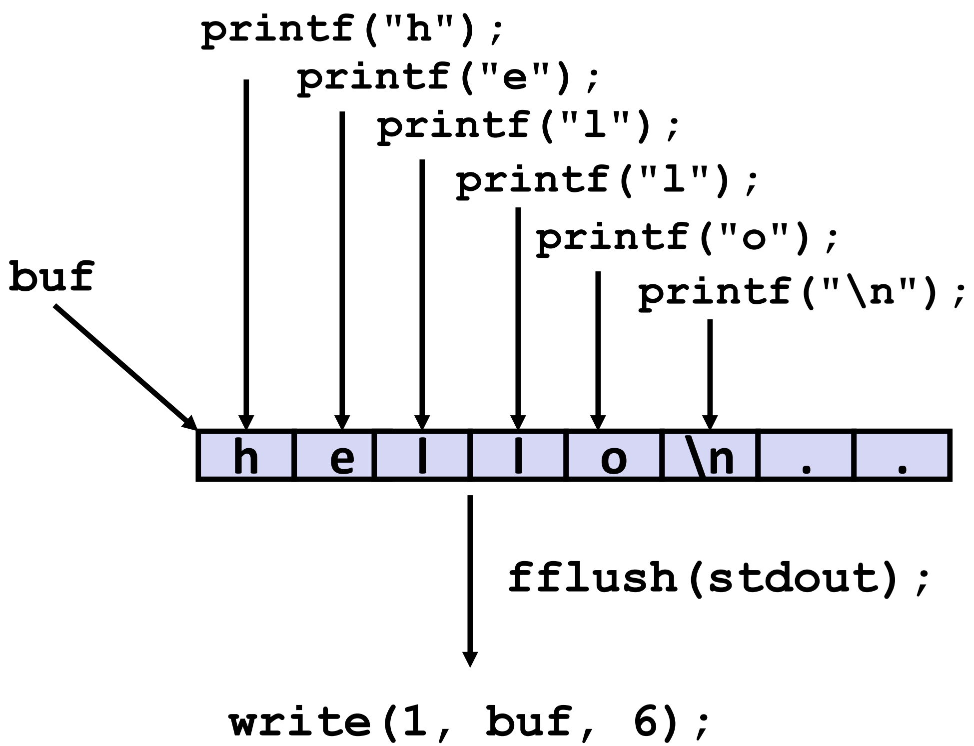
Buffered I/O: Motivation

- **Applications often read/write one character at a time**
  - `getc, putc, ungetc`
  - `gets, fgets`
    - Read line of text one character at a time, stopping at newline
- **Implementing as Unix I/O calls expensive**
  - **`read`** and **`write`** require Unix kernel calls
    - > 10,000 clock cycles
- **Solution: Buffered read**
  - Use Unix **`read`** to grab block of bytes
  - User input functions take one byte at a time from buffer
    - Refill buffer when empty



Buffering in Standard I/O

- **Standard I/O** functions use **buffered I/O**



- Buffer flushed to output fd on "\n", call to `fflush` or `exit`, or return from `main`.

Standard I/O Buffering in Action

- You can see this buffering in action for yourself, using the always fascinating Linux `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_group(0)                             = ?
```

Standard I/O Example

- **Copying file to stdout, line-by-line with stdio**

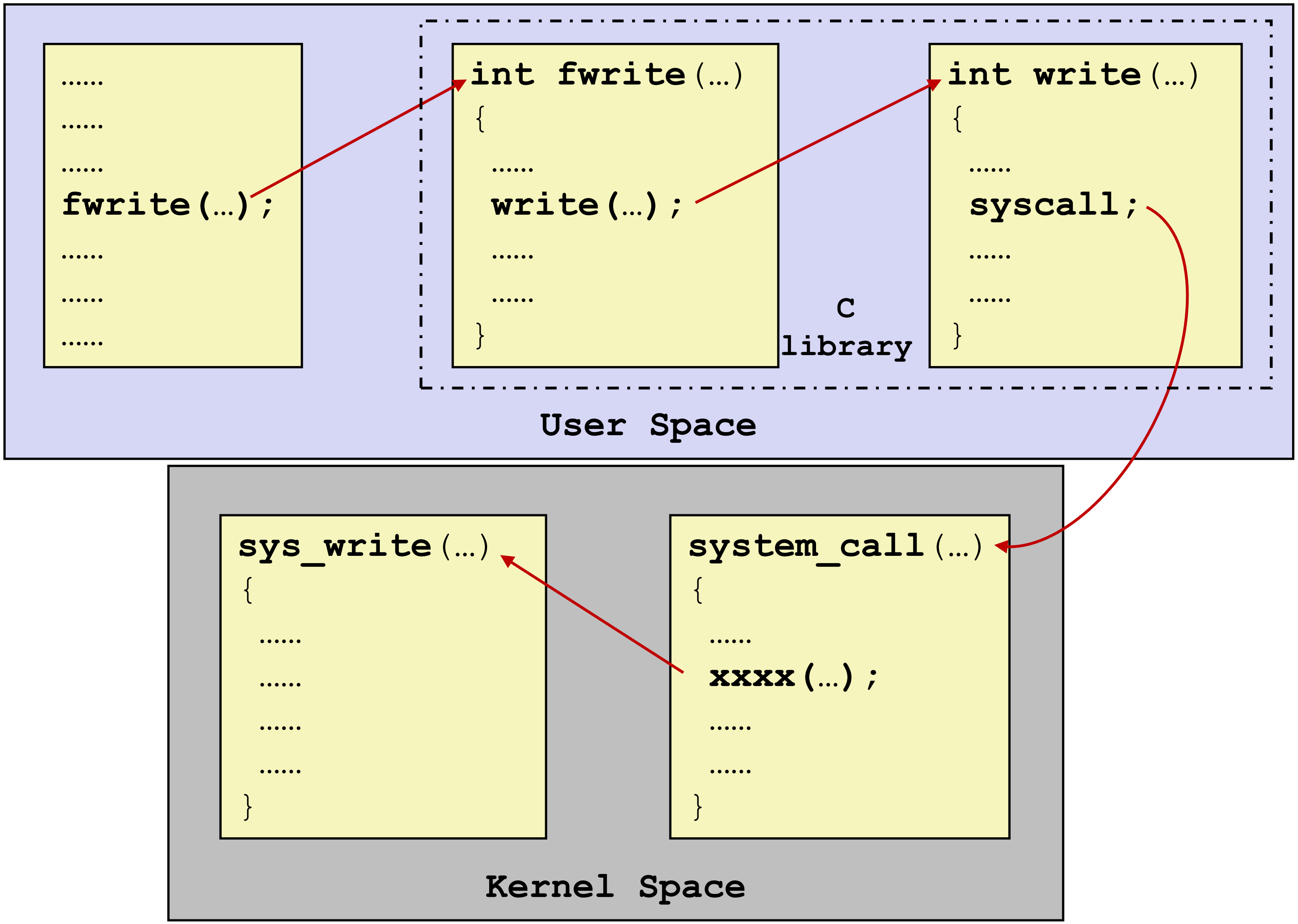
```
#include "csapp.h"
#define MLINE 1024

int main(int argc, char *argv[])
{
    char buf[MLINE];
    FILE *infile = stdin;
    if (argc == 2) {
        infile = fopen(argv[1], "r");
        if (!infile) exit(1);
    }
    while(fgets(buf, MLINE, infile) != NULL)
        fputs(buf, stdout);
    exit(0);
}
```

- **Demo:**

```
linux> strace ./showfile3_stdio names.txt
```

Standard I/O Functions→Unix I/O

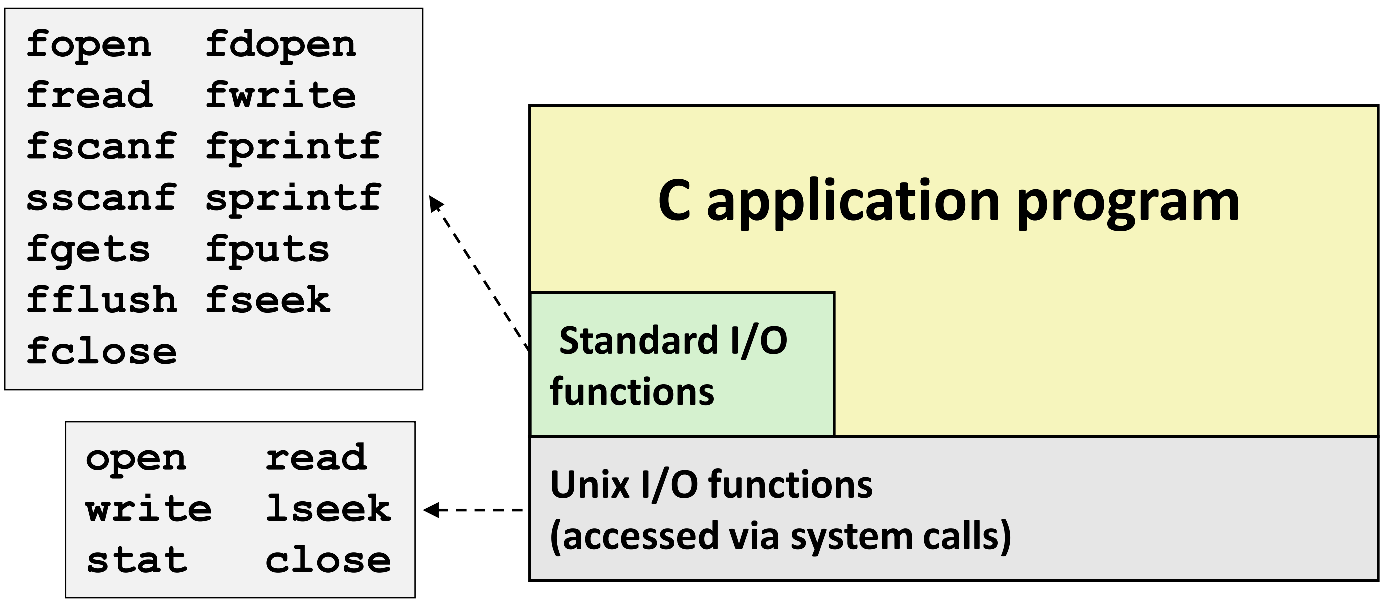


Today

- I/O Systems
- Unix I/O
- Metadata, sharing, and redirection
- Standard I/O
- **Summary**

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
  - Unix I/O functions are async-signal-safe and can be used safely in signal handlers

- **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

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 functions are not async-signal-safe, and not appropriate for signal handlers
  - 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:APP3e, Sec 10.11)

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
  - But, be sure to understand the functions you use!

- **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

Aside: Working with Binary Files

- **Binary File**
  - Sequence of arbitrary bytes
  - Including byte value 0x00
- **Functions you should *never* use on binary files**
  - **Text-oriented I/O:** such as **`fgets, scanf`**
    - Interpret EOL characters.
  - **String functions**
    - **`strlen, strcpy, strcat`**
    - Interprets byte value 0 (end of string) as special

教材阅读

- 第10章 10.1、10.2、10.3、10.4、10.6-10.12
- 参考书

  《计算机系统基础》； 袁春风；机械工业出版社

  参考阅读 第8章 8.1、8.2、8.3.4、8.4.1、8.4.2