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

Overview (Cont.)

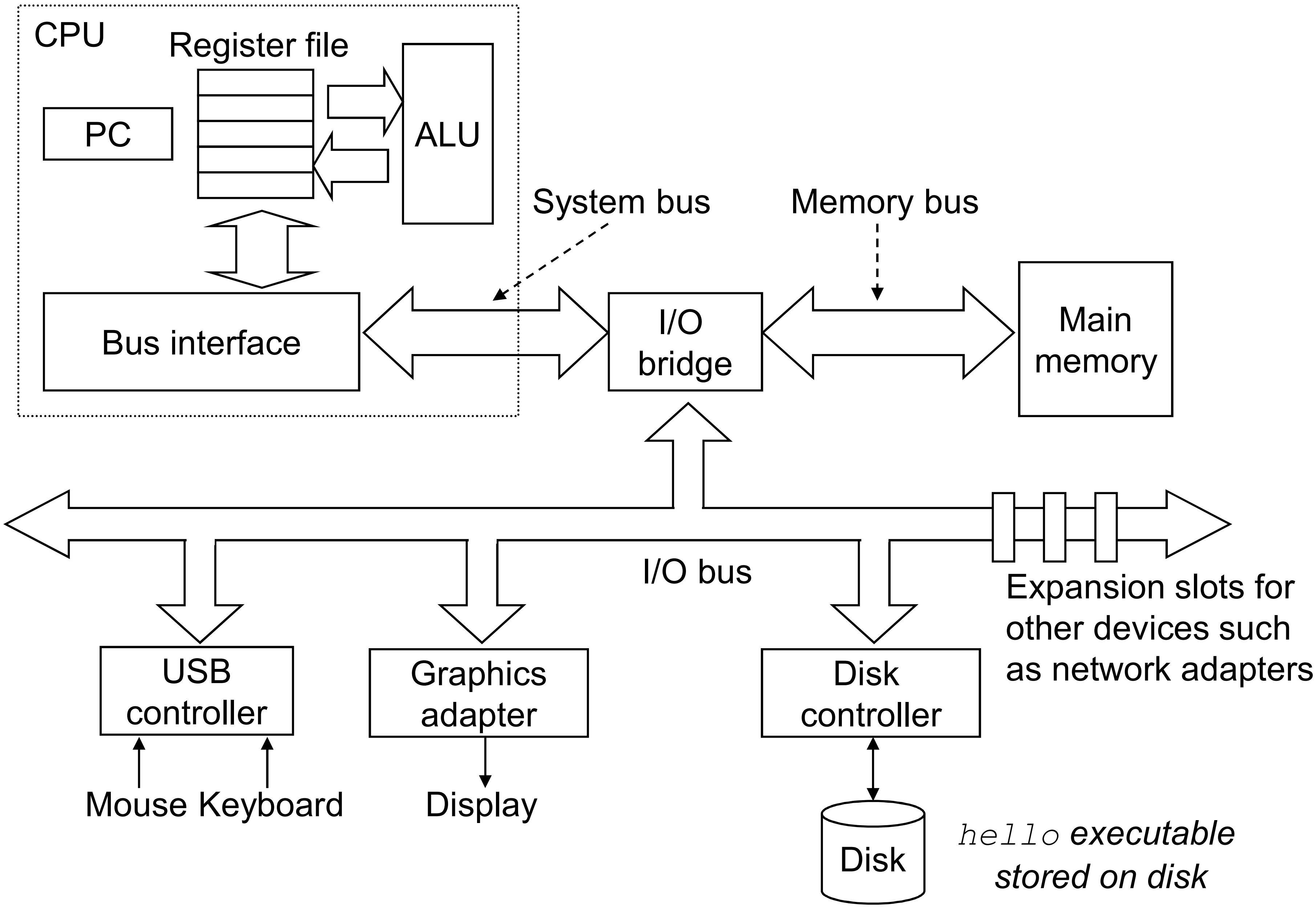
- **I/O management is a major component of operating system**
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- **Ports, busses, device controllers connect to various devices**
- **Device drivers encapsulate device details**
  - Present uniform device-access interface to I/O subsystem

How does the CPU talk to devices?

- **Device controller: Hardware that enables devices to talk to the peripheral bus**
- **Host adapter: Hardware that enables the computer to talk to the peripheral**
- **Bus: Wires that transfer data between components inside computer**
- **Device controller allows OS to specify simpler instructions to access data**
- **Example: 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

Typical Computer (PC) Today: HW Organization



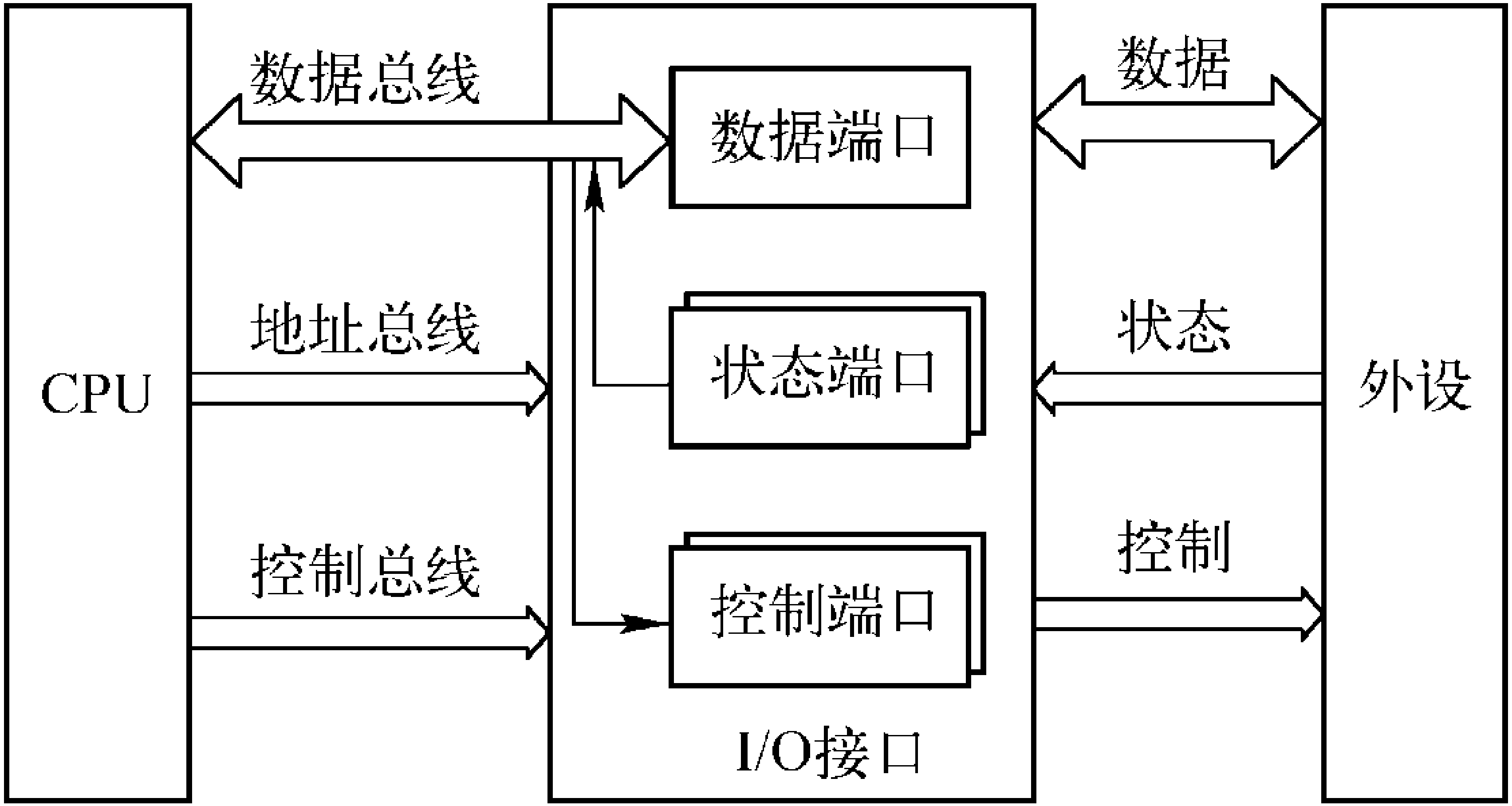
I/O Hardware

- **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
  - **Controller** (**host adapter**) – electronics that operate port, bus, device
    - Sometimes integrated
    - Sometimes separate circuit board (host adapter)
    - Contains processor, microcode, private memory, bus controller, etc
      - – Some talk to per-device controller with bus controller, microcode, memory, etc

I/O Hardware (Cont.)

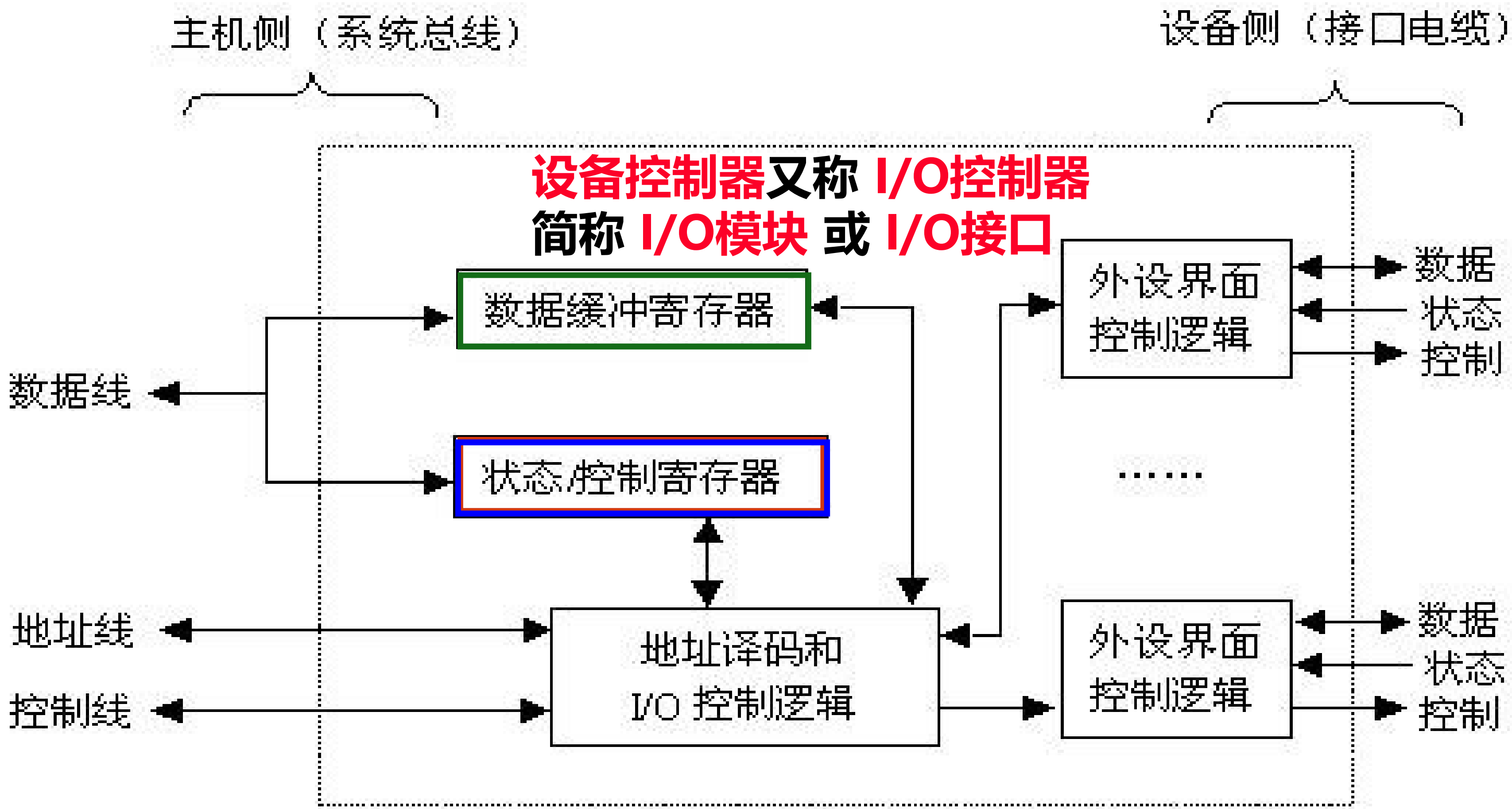
- **I/O instructions control devices**
- **Devices usually have registers where device driver places commands, addresses, and data to write, or read data from registers after command execution**
  - Data-in register, data-out register, status register, control register
  - Typically 1-4 bytes, or FIFO 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



设备控制器的结构

- 设备控制器的一般结构：不同I/O模块在复杂性和控制外设的数量上相差很大



通过发送命令字到I/O控制寄存器来向设备发送命令

通过从状态寄存器读取状态字来获取外设或I/O控制器的状态信息

通过向I/O控制器发送或读取数据来和外设进行数据交换

将I/O控制器中CPU能够访问的各类寄存器称为I/O端口

对外设的访问通过向I/O端口发命令、读状态、读/写数据来进行

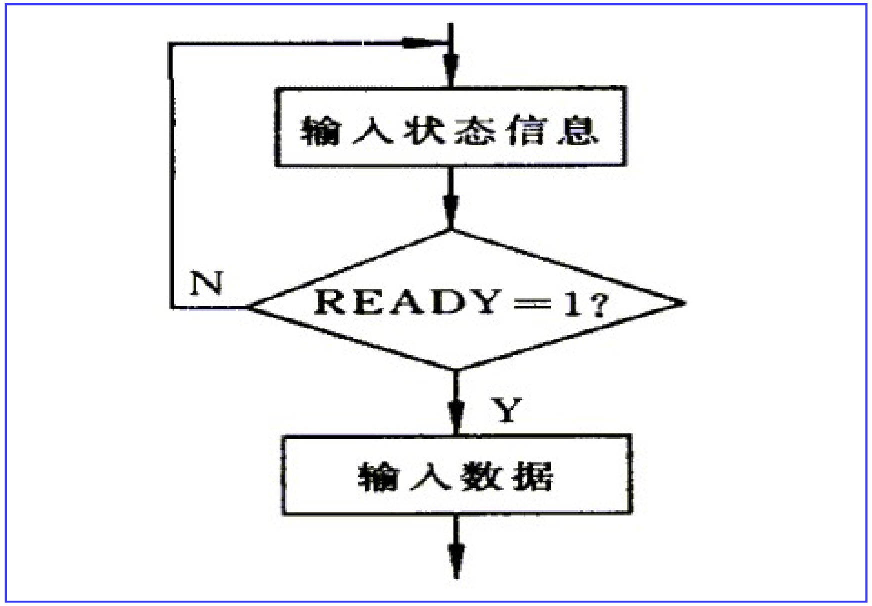
驱动程序与I/O指令

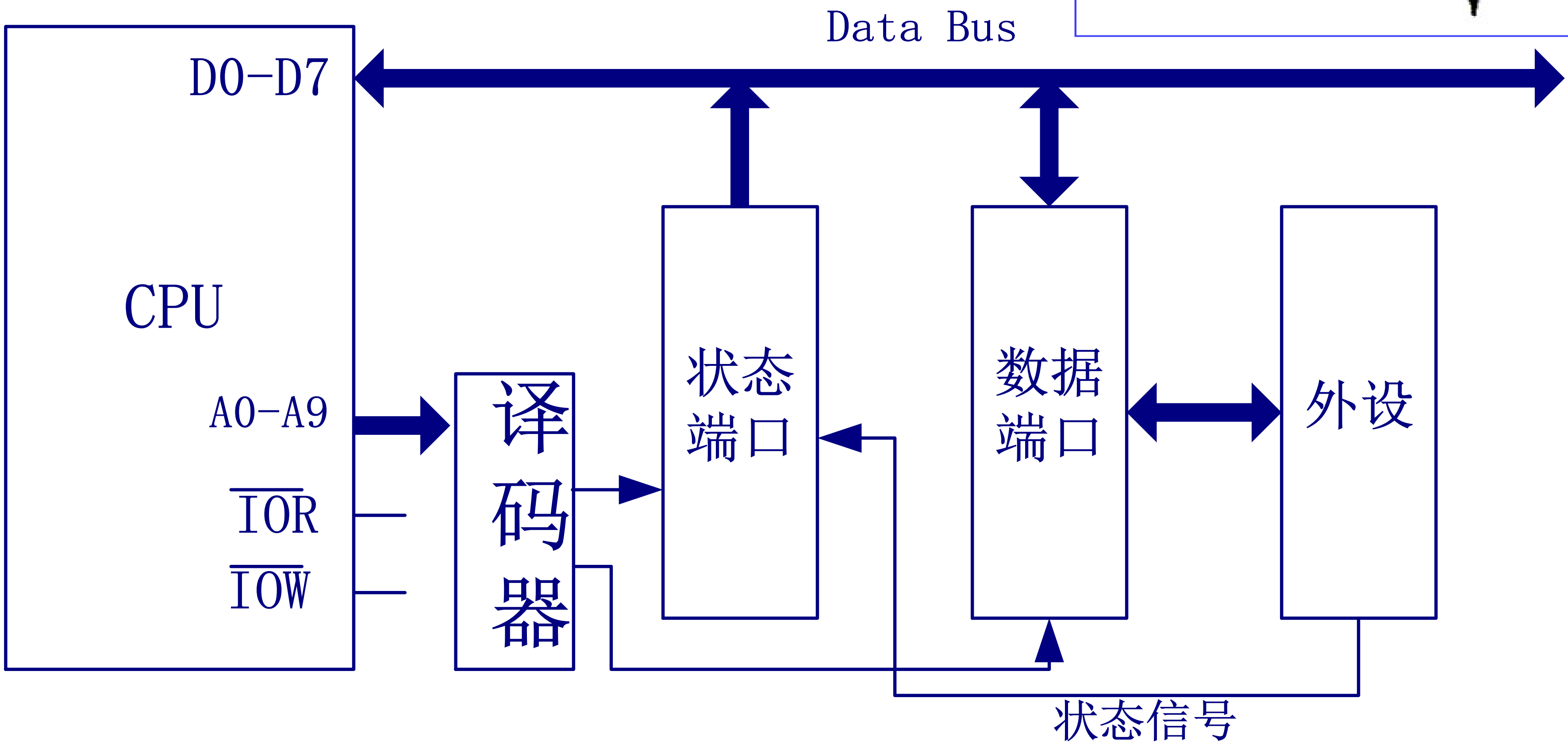
- 控制外设进行输入/输出的底层I/O软件是驱动程序
- 驱动程序设计者应了解设备控制器及设备的工作原理，包括：设备控制器中有哪些用户可访问的寄存器、控制/状态寄存器中每一位的含义、设备控制器与外设之间的通信协议等，而关于外设的机械特性，程序员则无需了解。驱动程序通过访问I/O端口控制外设进行I/O：
  - 将控制命令送到控制寄存器来启动外设工作；
  - 读取状态寄存器了解外设和设备控制器的状态；
  - 访问数据缓冲寄存器进行数据的输入和输出。
- 对I/O端口的访问操作由I/O指令完成，它们是一种特权指令

Three Types of I/O

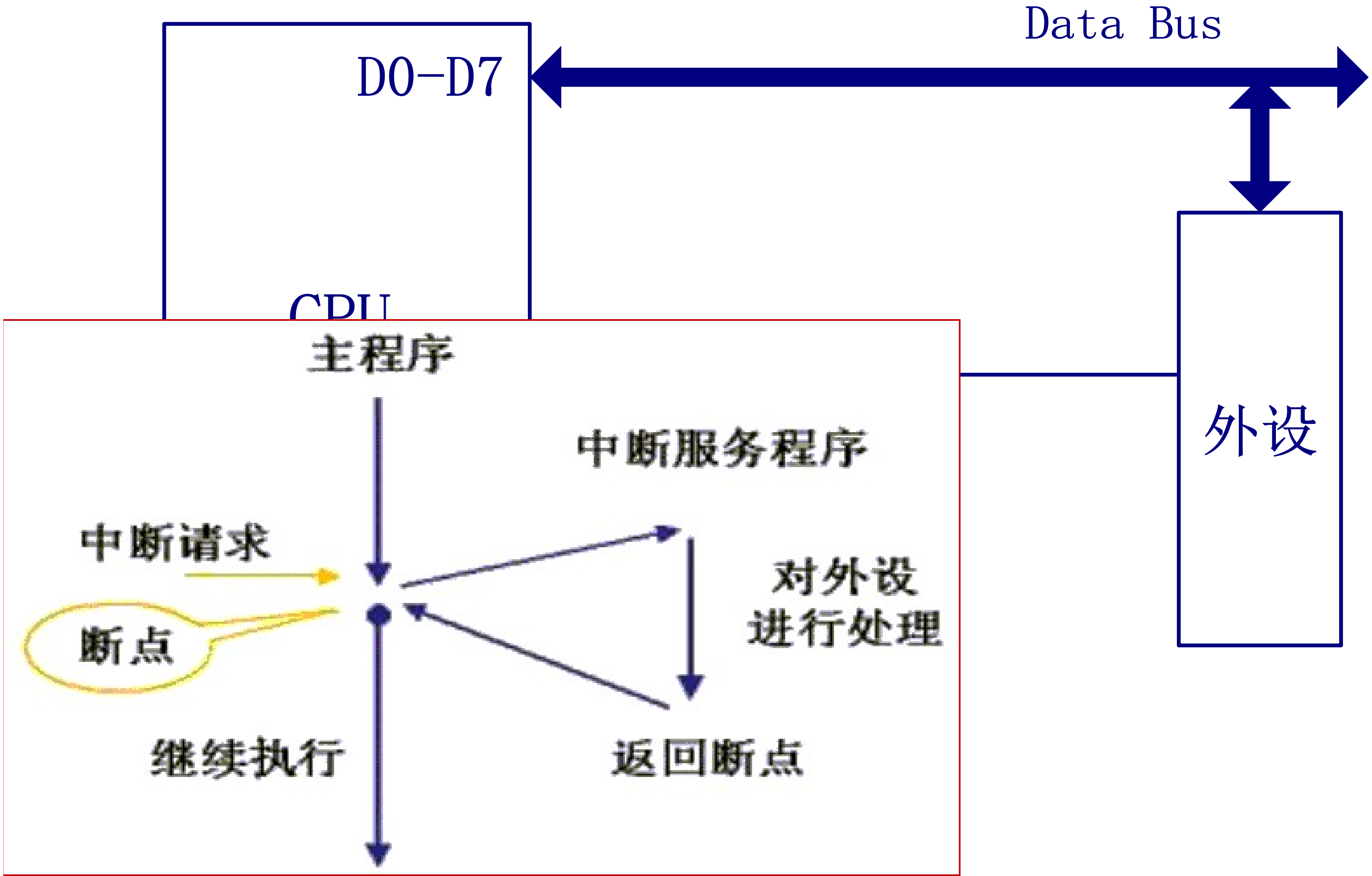
- **Programmed I/O(Polling): continuous attention of the processor is required**
- **Interrupt driven I/O: processor launches I/O and can continue until interrupted**
- **Direct memory access(DMA): the dma module governs the exchange of data between the I/O unit and the main memory**

查询方式

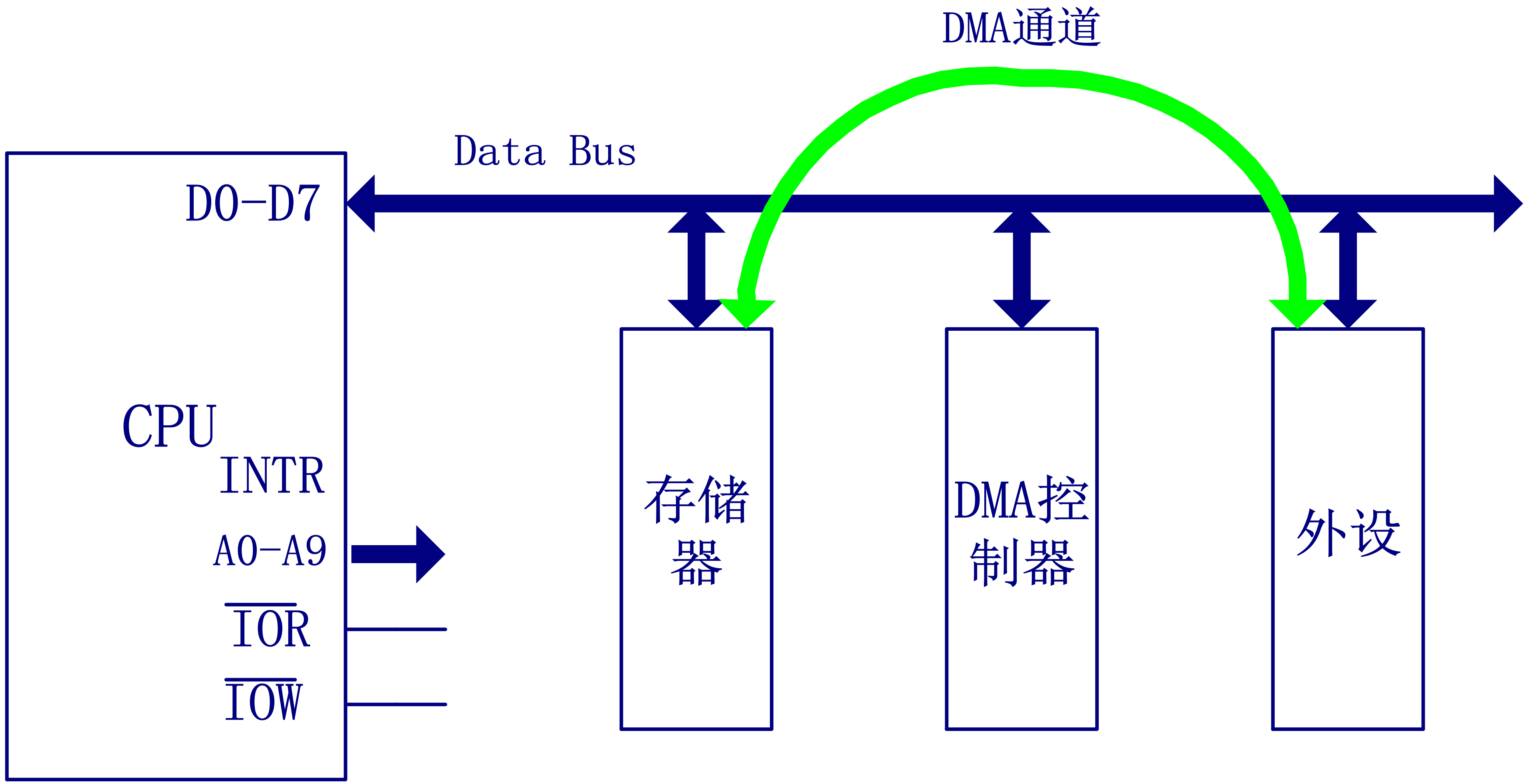

输入状态信息
READY＝1?
N
Y
输入数据


Data Bus
D0-D7
CPU
A0-A9
IOR
IOW
译码器
状态端口
数据端口
外设
状态信号

中断方式



DMA方式


DMA通道
Data Bus
D0-D7
CPU
INTR
A0-A9
$\overline{\text{IOR}}$
$\overline{\text{IOW}}$
存储器
DMA控制器
外设

I/O子系统概述

各类用户的I/O请求需要通过某种方式传给OS：

- 最终用户：键盘、鼠标通过操作界面传递给OS
- 用户程序：通过函数（高级语言）转换为系统调用传递给OS

I/O软件被组织成从高到低的四个层次，层次越低，则越接近设备而越远离用户程序。这四个层次依次为：

(1) 用户层I/O软件（I/O函数调用系统调用）

(2) 与设备无关的操作系统I/O软件

(3) 设备驱动程序

(4) I/O中断处理程序

OS

OS在I/O系统中极其重要！

大部分I/O软件都属于操作系统内核态程序，最初的I/O请求在用户程序中提出。

用户I/O软件

用户软件可用以下两种方式提出I/O请求：

（1）使用高级语言提供的标准I/O库函数。例如，在C语言程序中可以直接使用像fopen、fread、fwrite和fclose等文件操作函数，或printf、putc、scanf和getc等控制台I/O函数。 程序移植性很好！

但是，使用标准I/O库函数有以下几个方面的不足：

(a) 标准I/O库函数不能保证文件的安全性（无加/解锁机制）

(b) 所有I/O都是同步的，程序必须等待I/O操作完成后才能继续执行

(c) 有时不适合甚至无法使用标准I/O库函数实现I/O功能，如，不提供读取文件元数据的函数（元数据包括文件大小和文件创建时间等）

(d) 用它进行网络编程会造成易于出现缓冲区溢出等风险

（2）使用OS提供的API函数或系统调用。例如，在Windows中直接使用像CreateFile、ReadFile、WriteFile、CloseHandle等文件操作API函数，或ReadConsole、WriteConsole等控制台I/O的API函数。对于Unix或Linux用户程序，则直接使用像open、read、write、close等系统调用封装函数。

用户I/O软件

- 用户进程请求读磁盘文件操作
  - 用户进程使用标准C库函数fread，或Windows API函数ReadFile，或Unix/Linux的系统调用函数read等要求读一个磁盘文件块。
  - 用户程序中涉及I/O操作的函数最终会被转换为一组与具体机器架构相关的指令序列，这里我们将其称为I/O请求指令序列。
  - 每个指令系统中一定有一类陷阱指令（有些机器也称为软中断指令或系统调用指令），主要功能是为操作系统提供灵活的系统调用机制。
  - 在I/O请求指令序列中，具体I/O请求被转换为一条陷阱指令，在陷阱指令前面则是相应的系统调用参数的设置指令。

系统I/O软件

OS在I/O子系统中的重要性由I/O系统以下三个特性决定：

（1）共享性。I/O系统被多个程序共享，须由OS对I/O资源统一调度管理，以保证用户程序只能访问自己有权访问的那部分I/O设备，并使系统的吞吐率达到最佳。

（2）复杂性。I/O设备控制细节复杂，需OS提供专门的驱动程序进行控制，这样可对用户程序屏蔽设备控制的细节。

（3）异步性。不同设备之间速度相差较大，因而，I/O设备与主机之间的信息交换使用异步的中断I/O方式，中断导致从用户态向内核态转移，因此必须由OS提供中断服务程序来处理。

那么，如何从用户程序对应的用户进程进入到操作系统内核执行呢？

系统调用！

系统调用和API

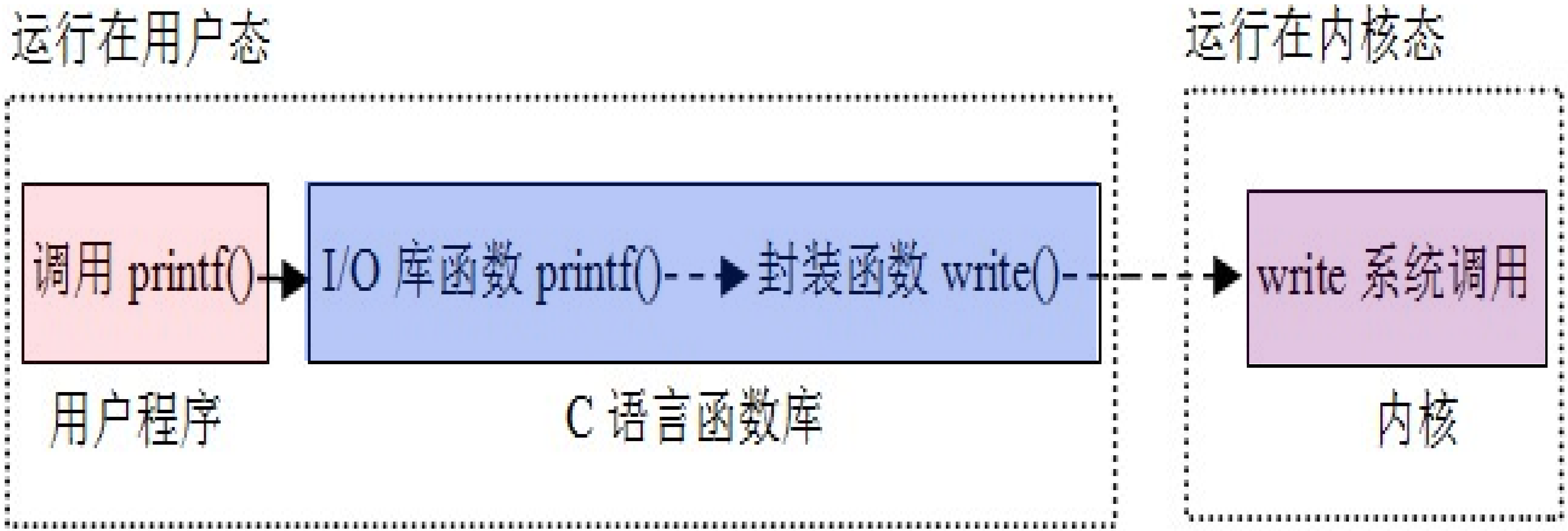
- OS提供一组系统调用为用户进程的I/O请求进行具体的I/O操作。
- 应用编程接口（API）与系统调用两者在概念上不完全相同，它们都是系统提供给用户程序使用的编程接口，但前者指的是功能更广泛、抽象程度更高的函数，后者仅指通过软中断（自陷）指令向内核态发出特定服务请求的函数。
- 系统调用封装函数是 API 函数中的一种。
- API 函数最终通过调用系统调用实现 I/O。一个API 可能调用多个系统调用，不同 API 可能会调用同一个系统调用。但是，并不是所有 API 都需要调用系统调用。
- 从编程者来看，API 和 系统调用之间没有什么差别。
- 从内核设计者来看，API 和 系统调用差别很大。API 在用户态执行，系统调用封装函数也在用户态执行，但具体服务例程在内核态执行。

用户程序、C函数和内核

- 用户程序总是通过某种I/O函数或I/O操作符请求I/O操作。

  例如，读一个磁盘文件记录时，可调用C标准I/O库函数fread()，也可直接调用系统调用封装函数read()来提出I/O请求。不管是C库函数、API函数还是系统调用封装函数，最终都通过操作系统内核提供的系统调用来实现I/O。

  例：printf()函数的调用过程如下：



以hello程序为例说明

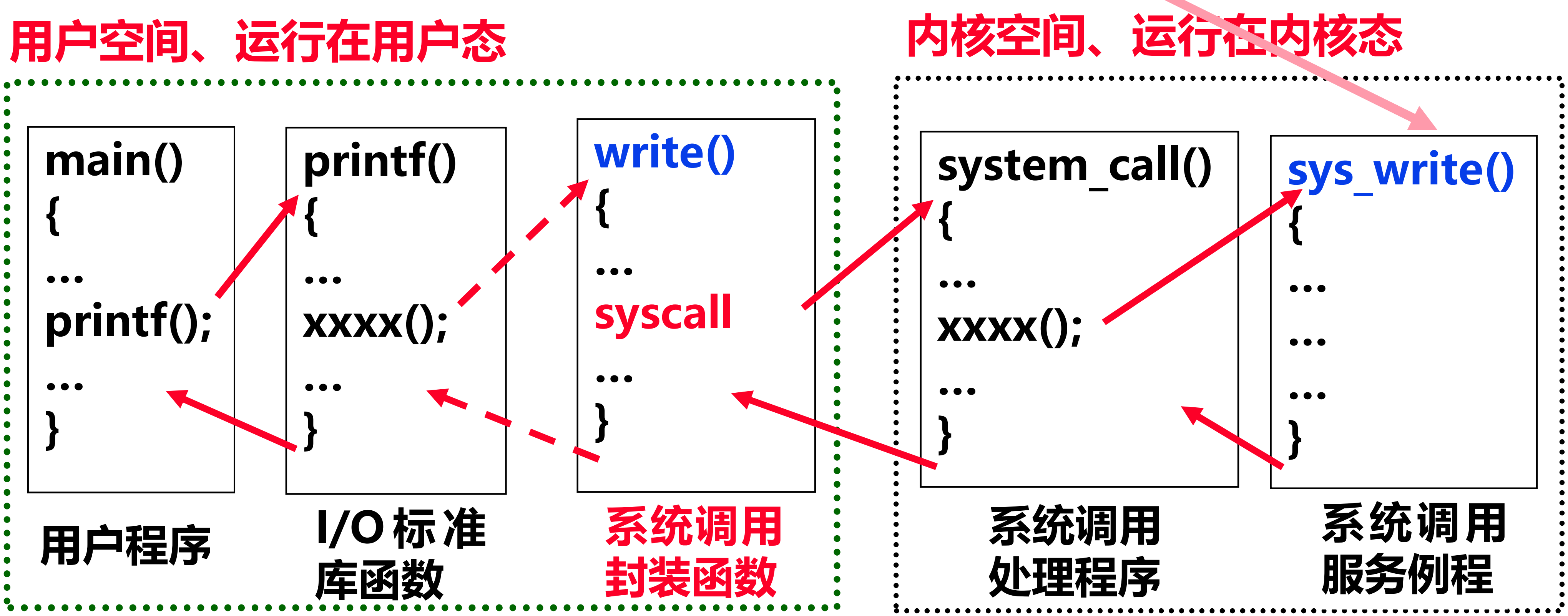
假定以下用户程序对应的进程为p

```
#include <stdio.h>
int main()
{
    printf("hello, world\n");
}
```

sys_write可用三种I/O方式实现：

程序查询、中断 和 DMA

可见：字符串输出最终是由内核中的sys_write系统调用服务例程实现

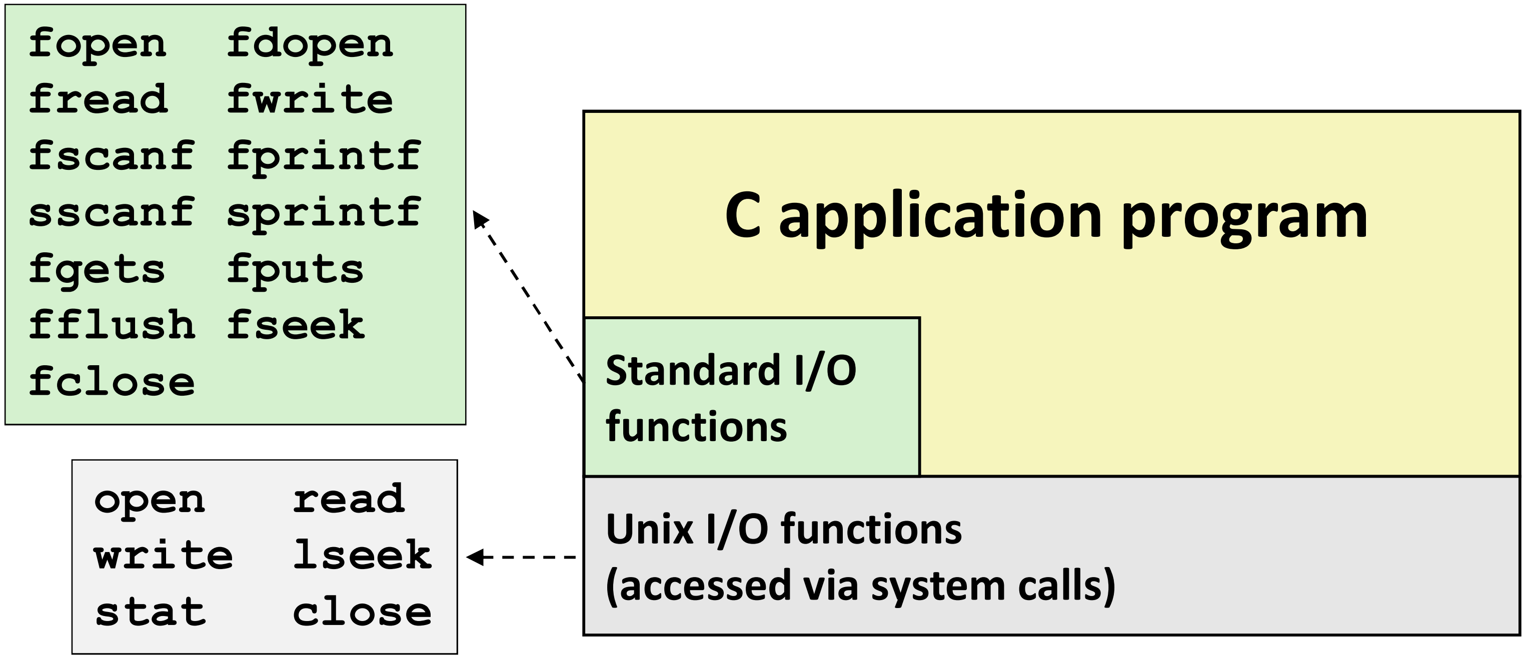


Today

- 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

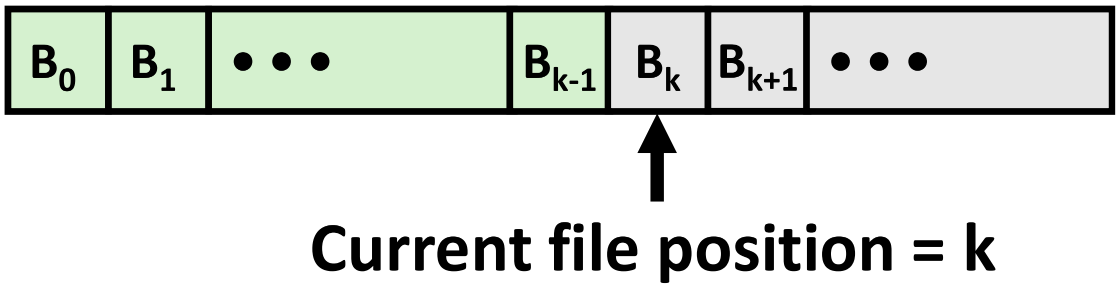


Unix I/O Overview

- 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

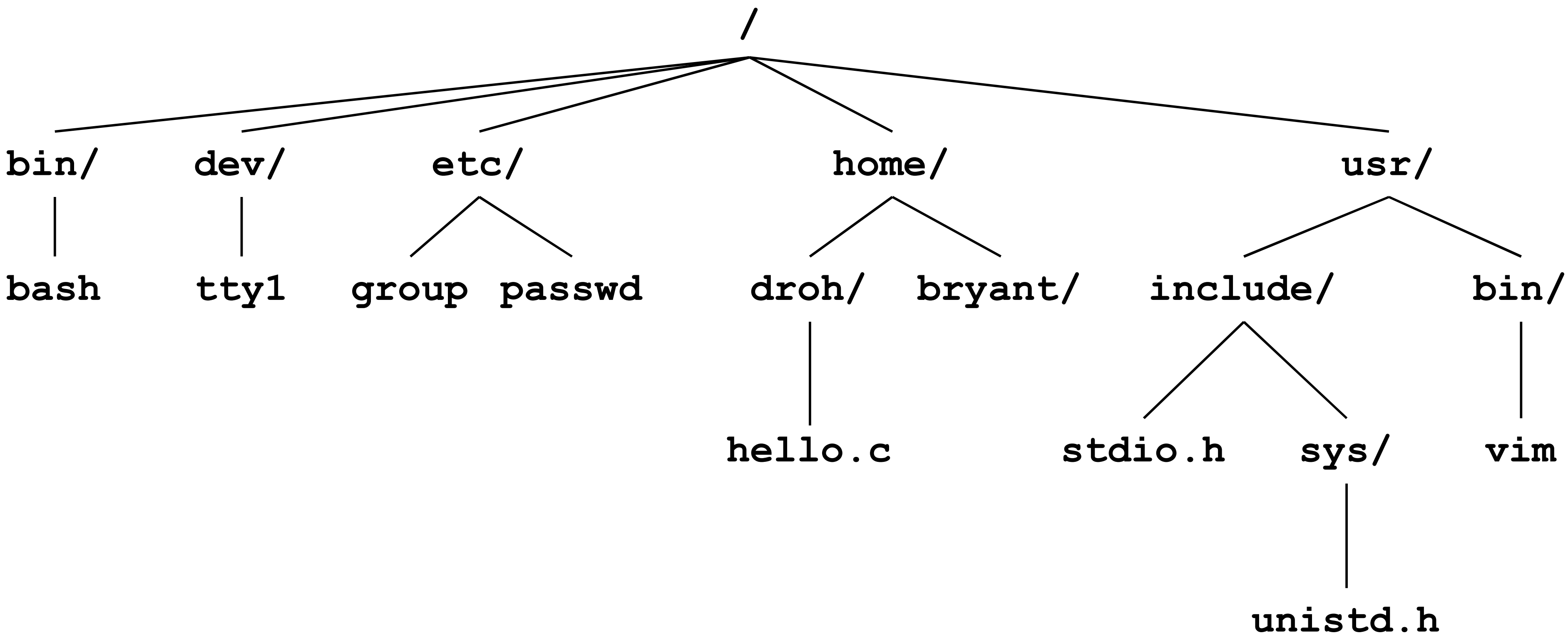
- **A regular file contains arbitrary data**
- **Applications often distinguish between *text files* and *binary files***
  - Text files are regular files with only ASCII or Unicode characters
  - Binary files are everything else
    - e.g., object files, JPEG images
  - Kernel doesn't know the difference!
- **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

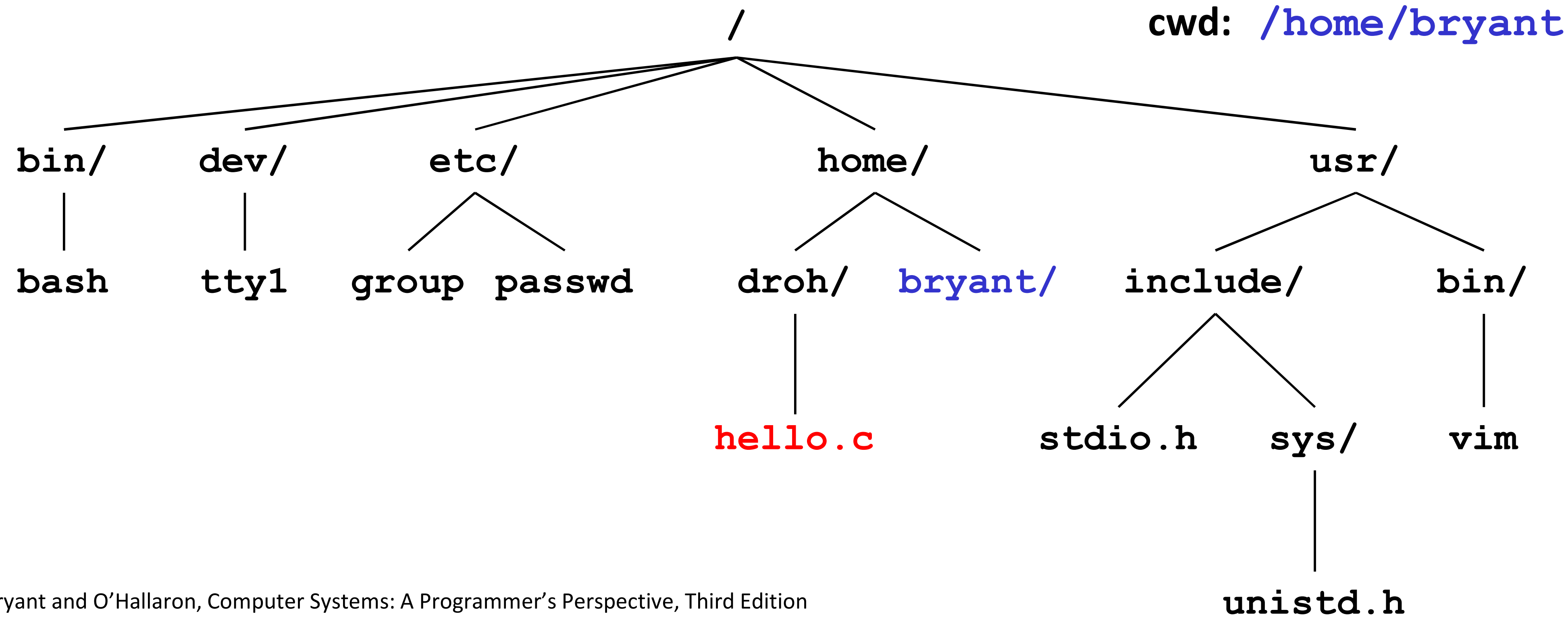
- **All files are organized as a hierarchy anchored by root directory named / (slash)**



- **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!

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 stdin to stdout, 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);
}
```

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

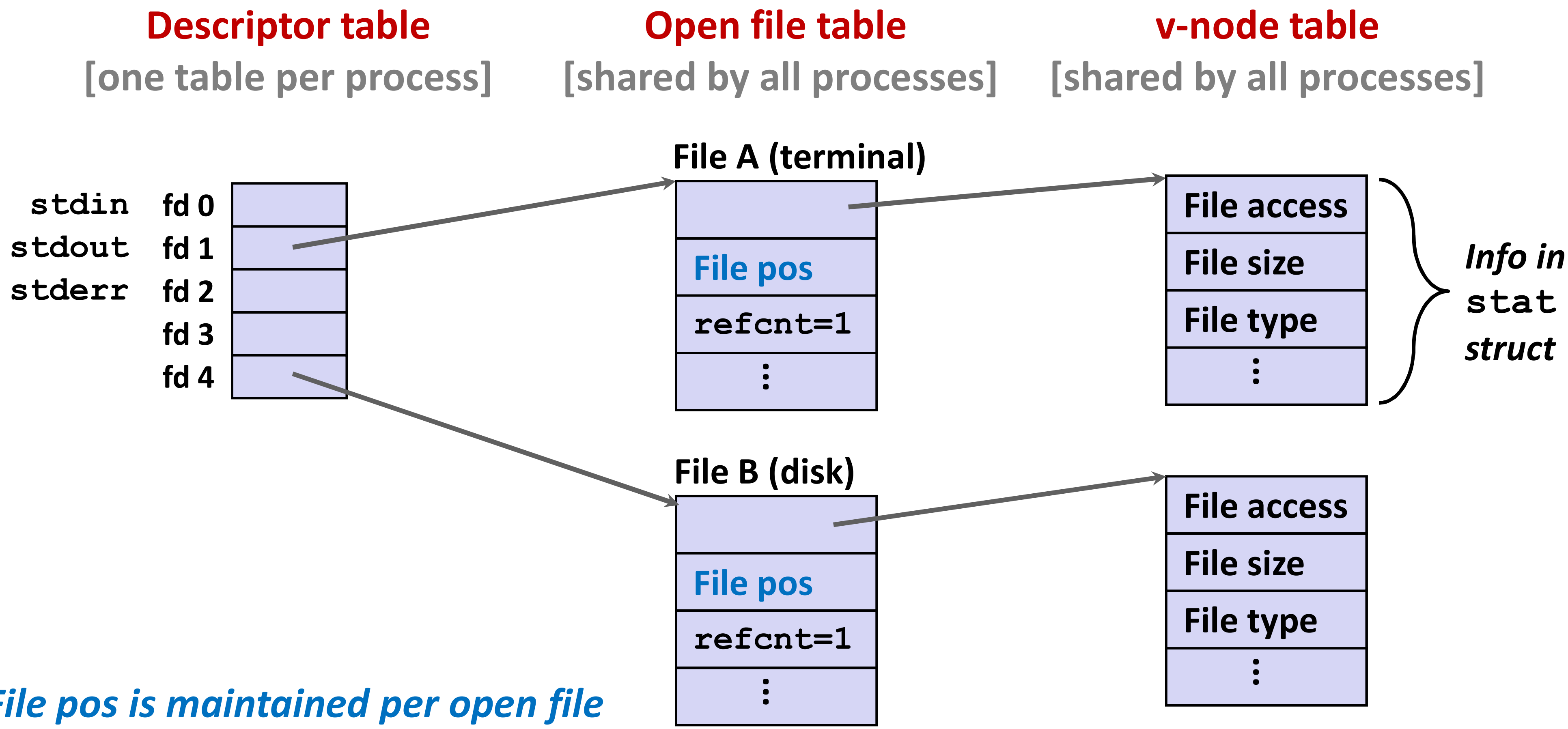
File Metadata

- *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

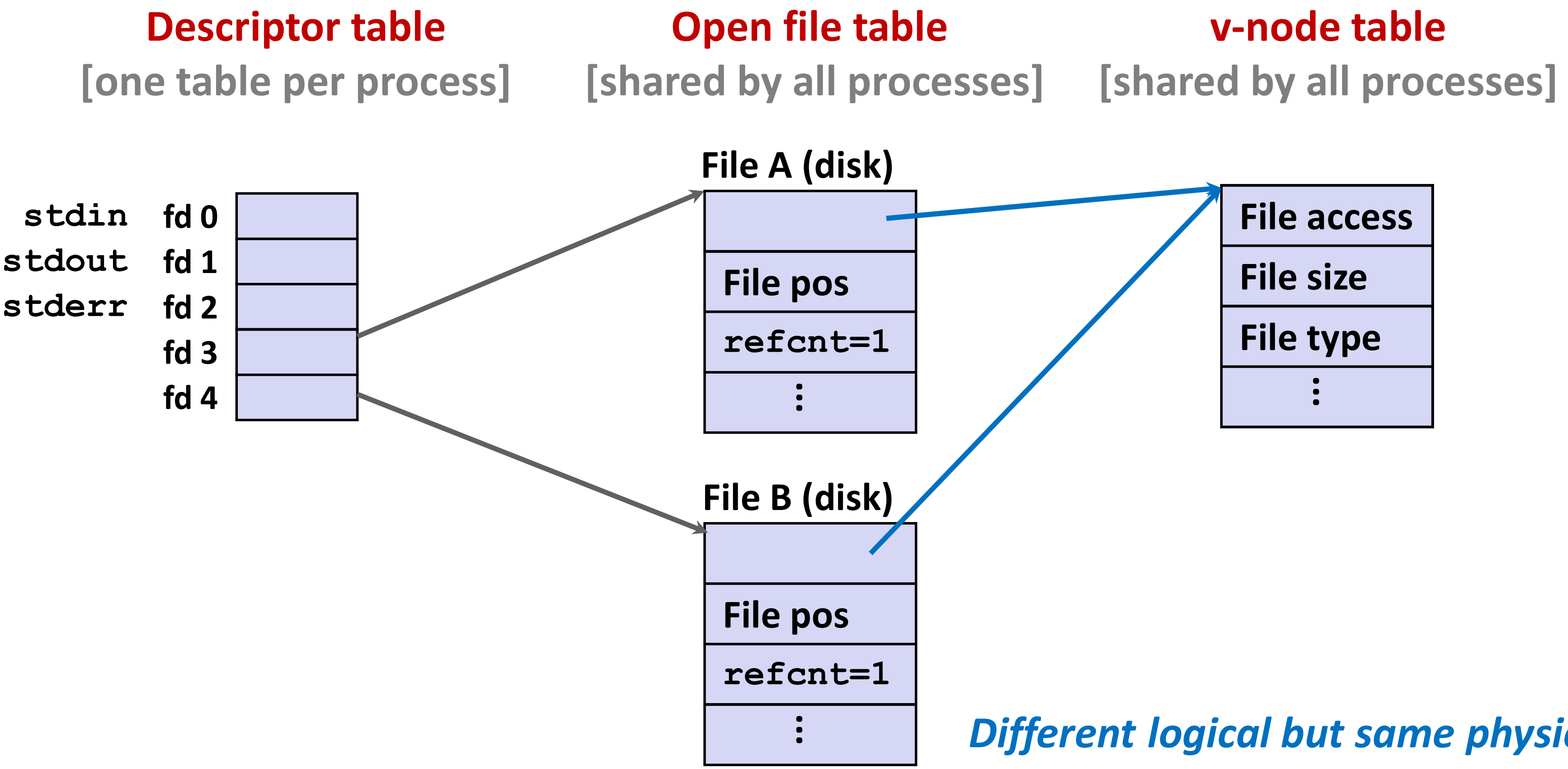
- Two descriptors referencing two distinct open files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file



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



Different logical but same physical file

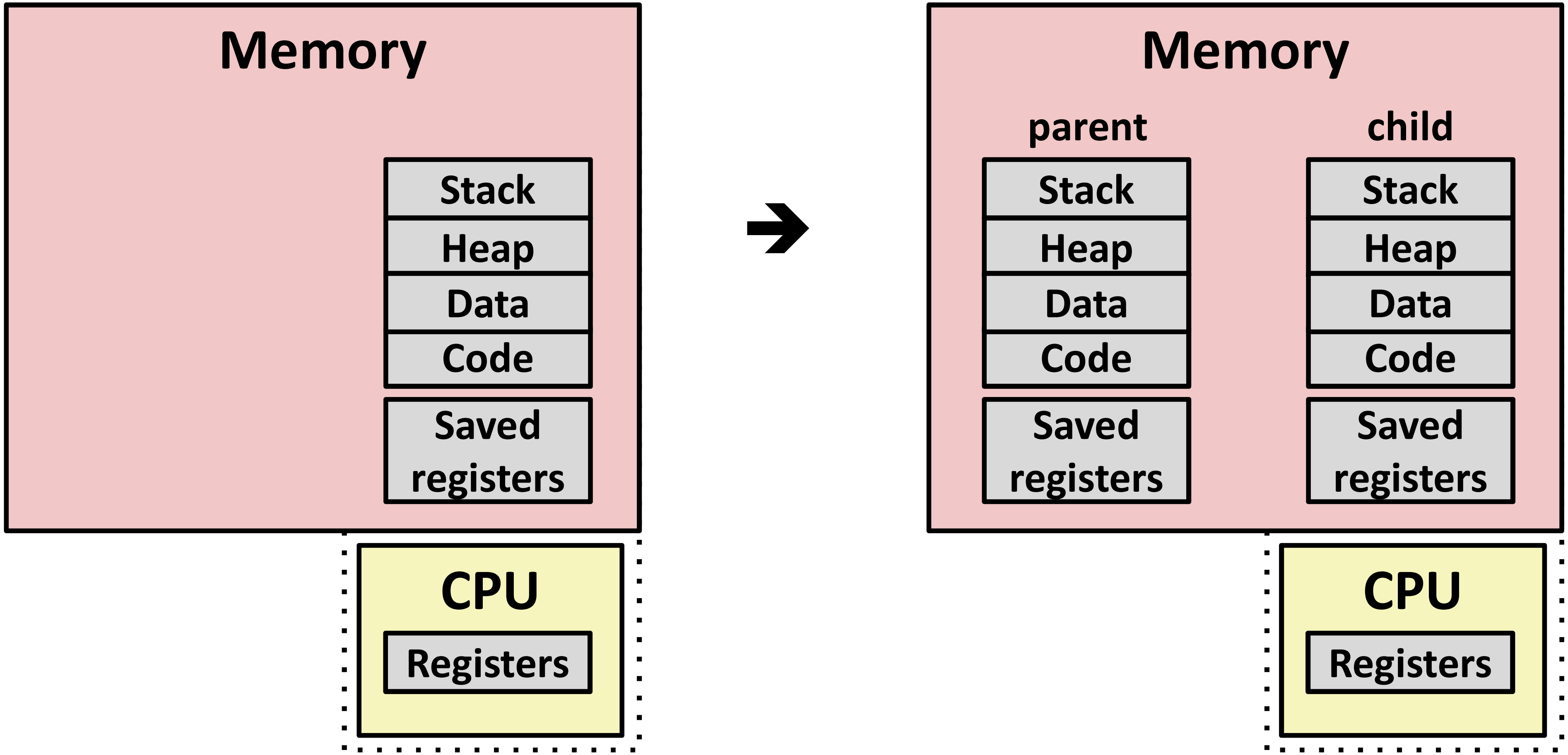
Creating Processes

- ***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`



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

fork Example

```c
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
```

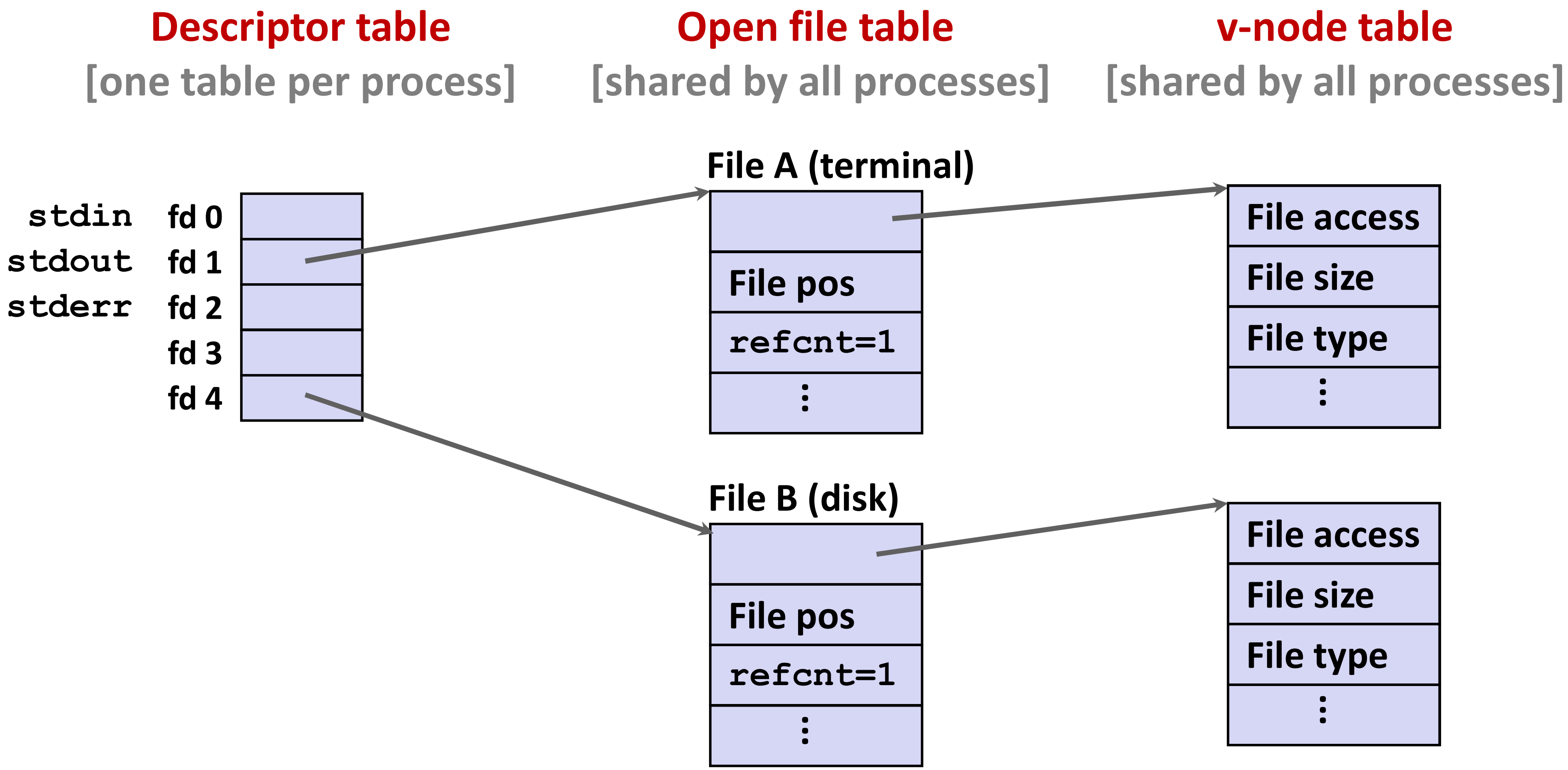
```
linux> ./fork
child : x=2
parent: x=0
```

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

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

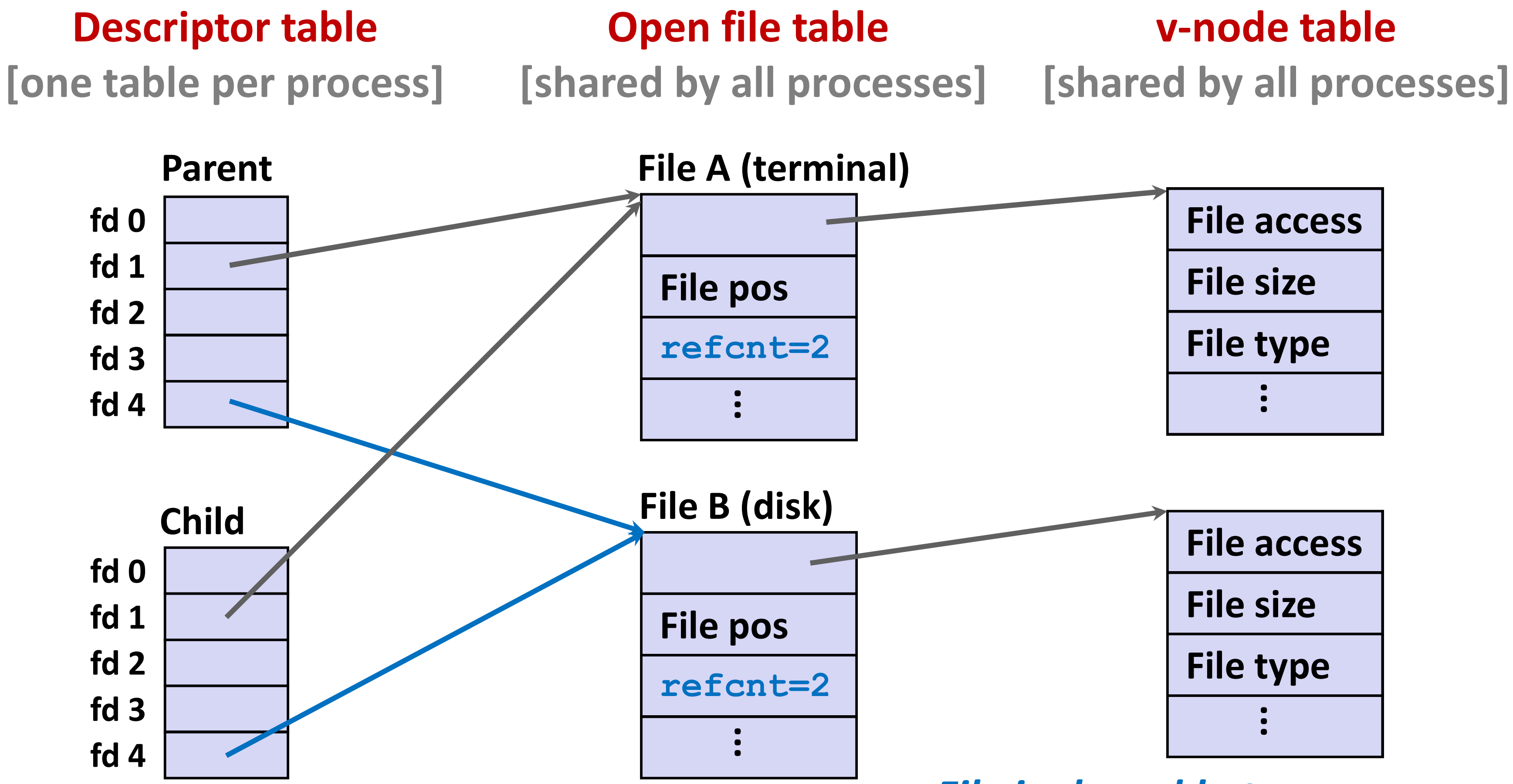
How Processes Share Files: `fork`

- **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

- **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)`

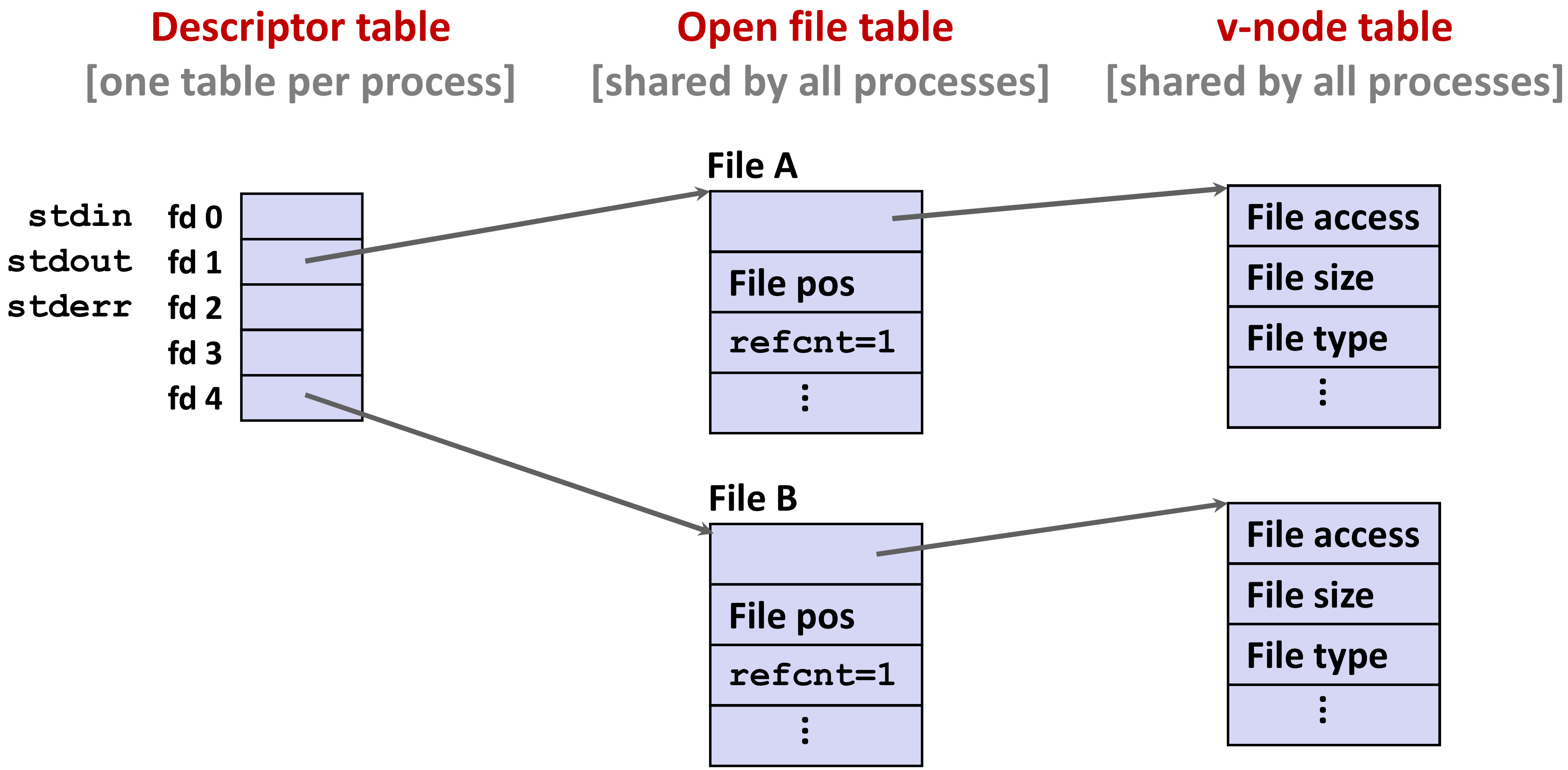
fd 0	
fd 1	a
fd 2	
fd 3	
fd 4	b

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

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

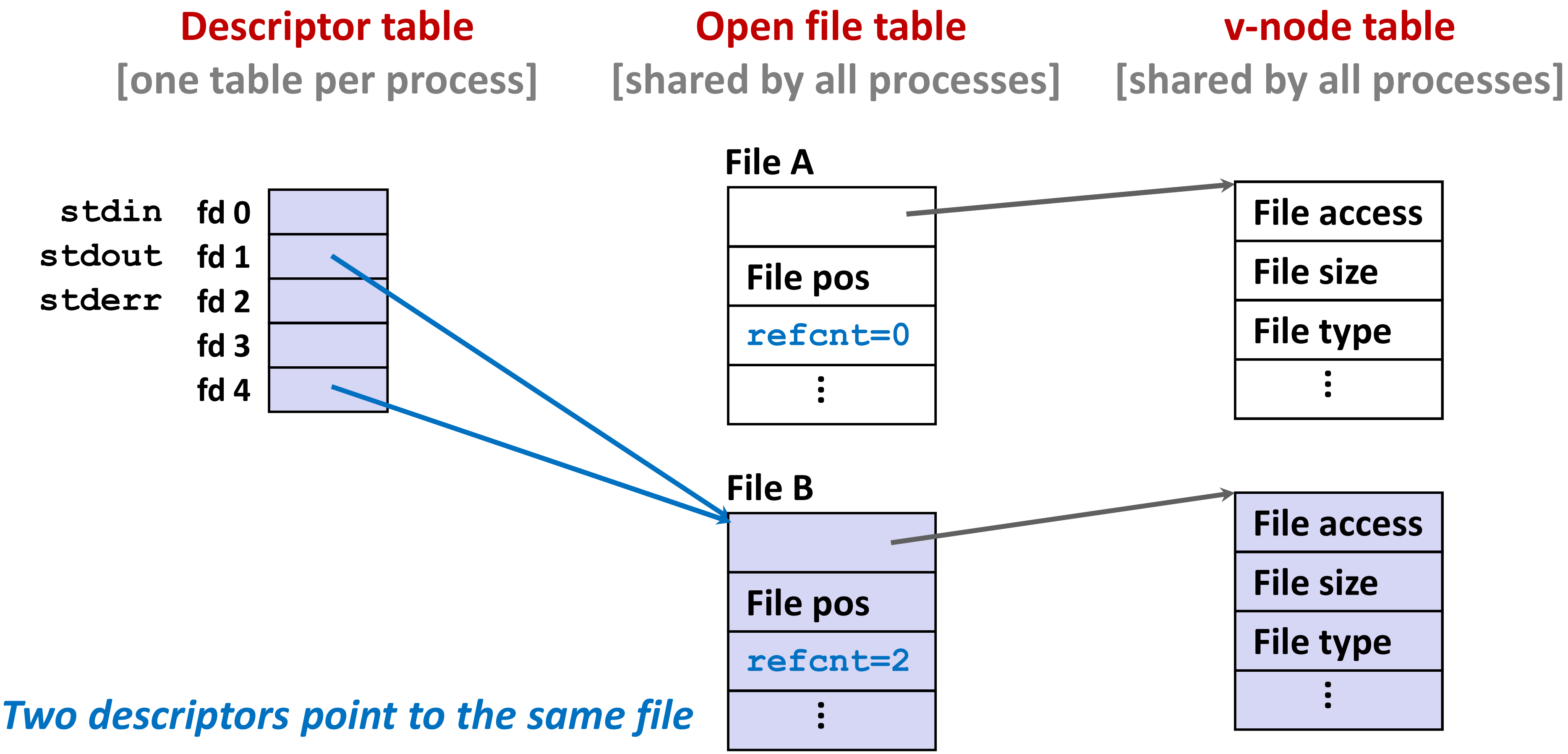
I/O Redirection Example

- **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



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

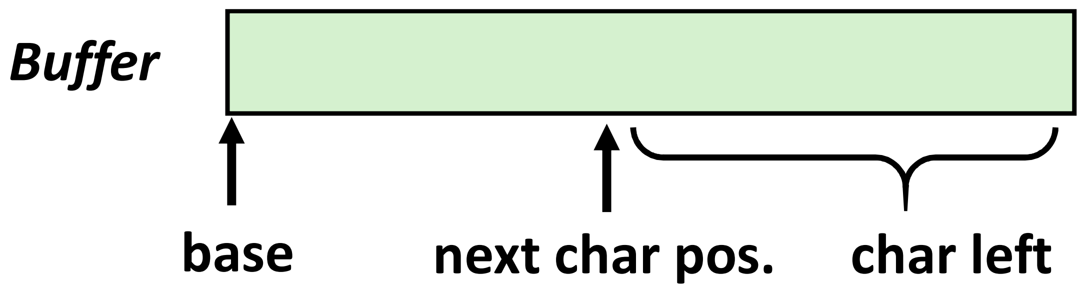
- **C programs begin life with three open streams (defined in `stdio.h`)**
  - `stdin` (standard input)
  - `stdout` (standard output)
  - `stderr` (standard error)

```
#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

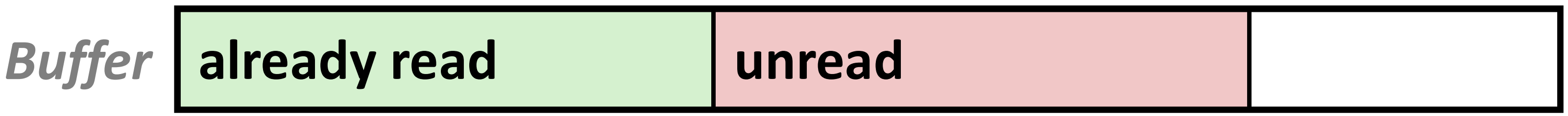
- **Standard I/O models open files as *streams***
  - Abstraction for a file descriptor and a buffer in memory




```
/* 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
```

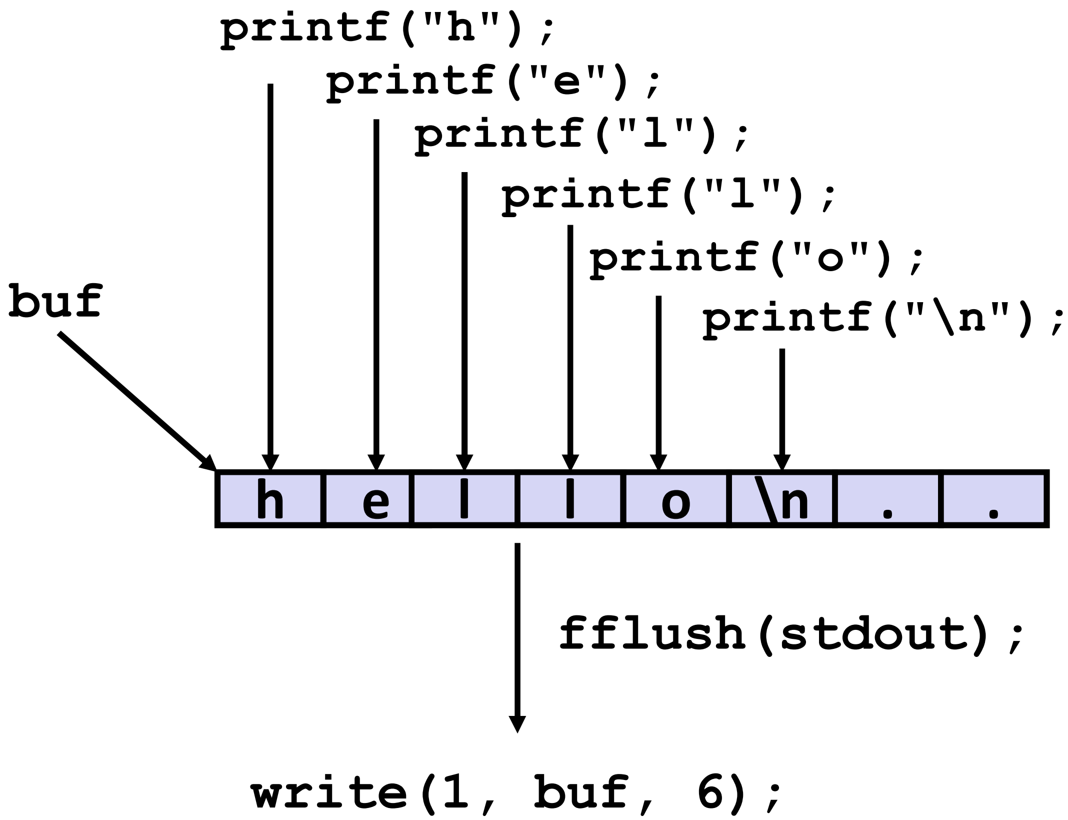
Buffered I/O: Motivation

- **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

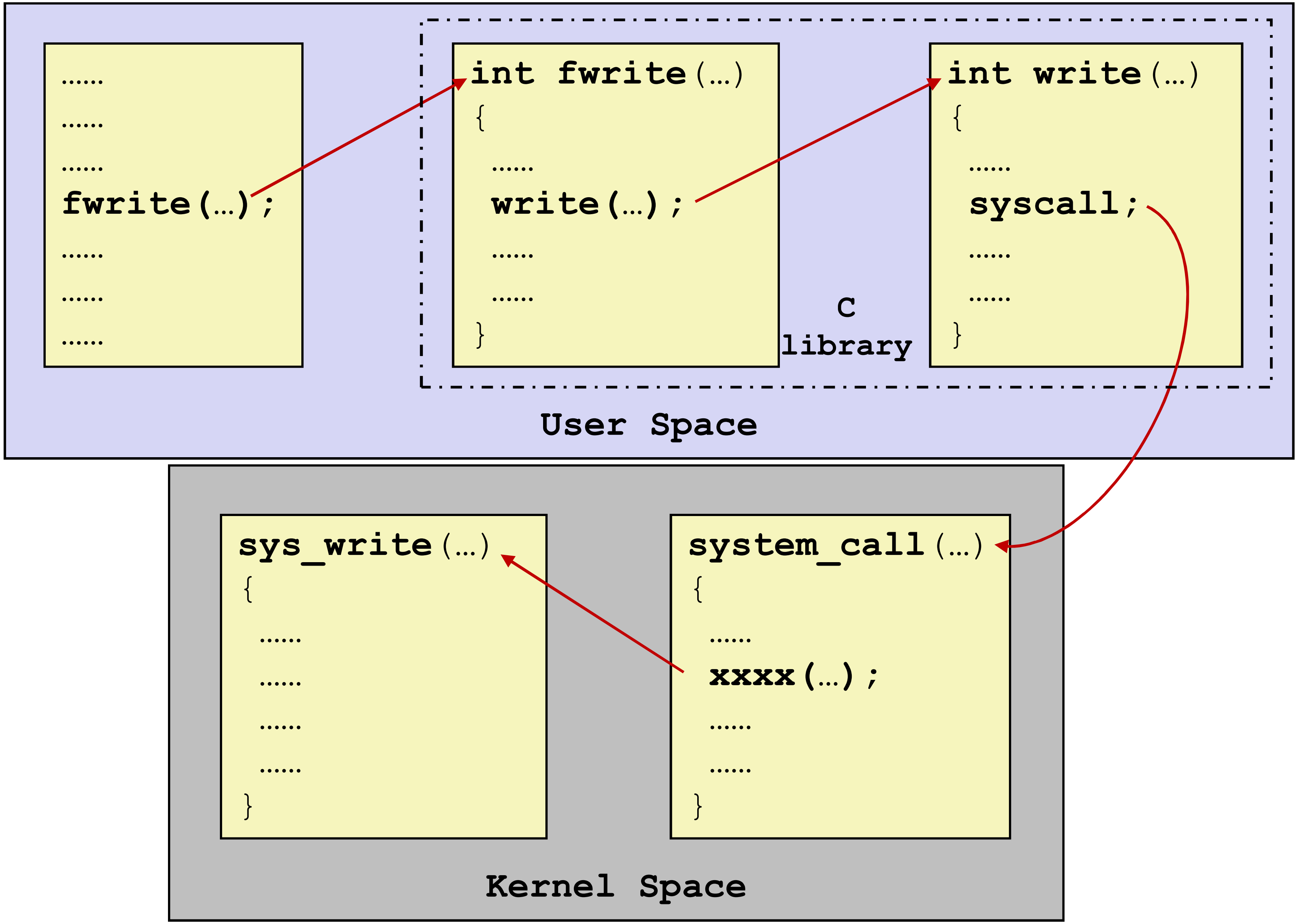
- You can see this buffering in action for yourself, using the always fascinating Linux `strace` program:

```
#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 Functions→Unix I/O

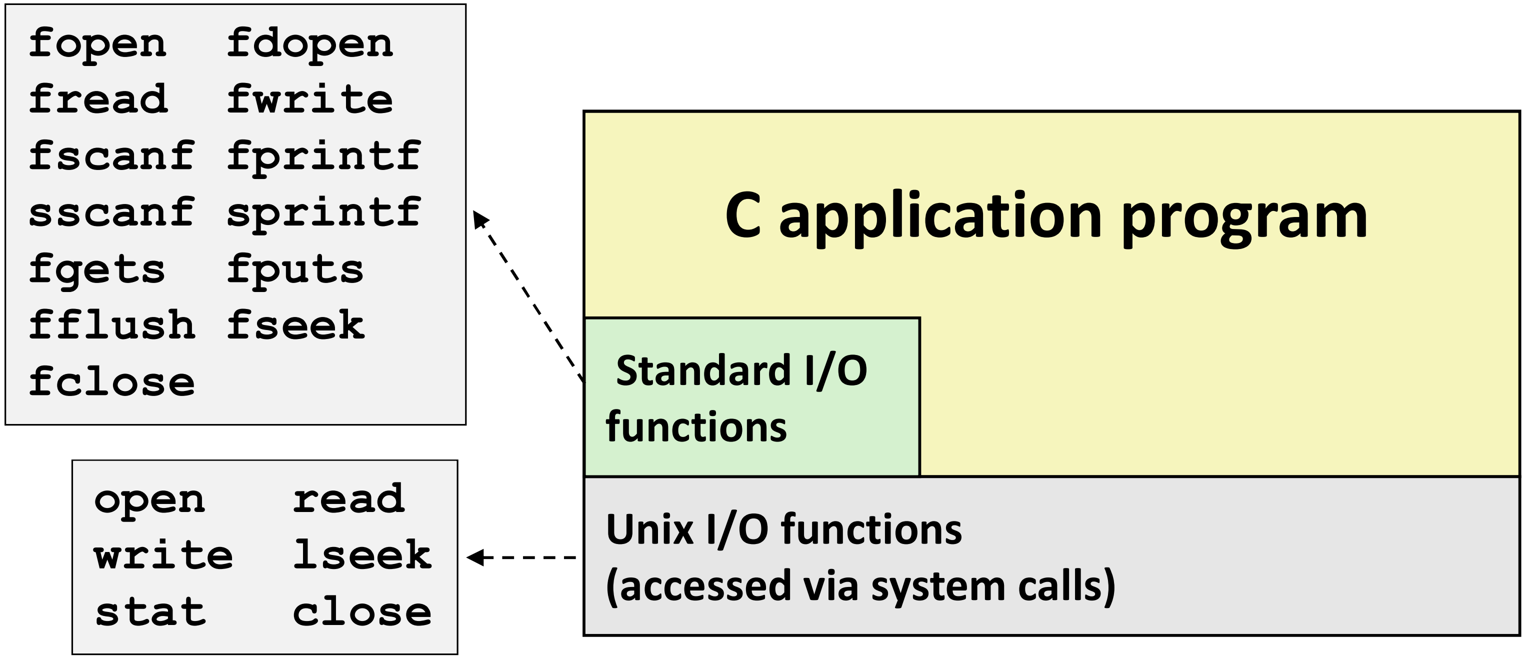


Today

- 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

- **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**