

I/O Systems

15-213: Introduction to Computer Systems
16th Lecture, October 19th, 2017

Instructor:

Randy Bryant

Today

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

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

操作系统的三个部分

□ 内核

- 操作系统五大管理功能一般都由操作系统内核负责。

□ 外壳

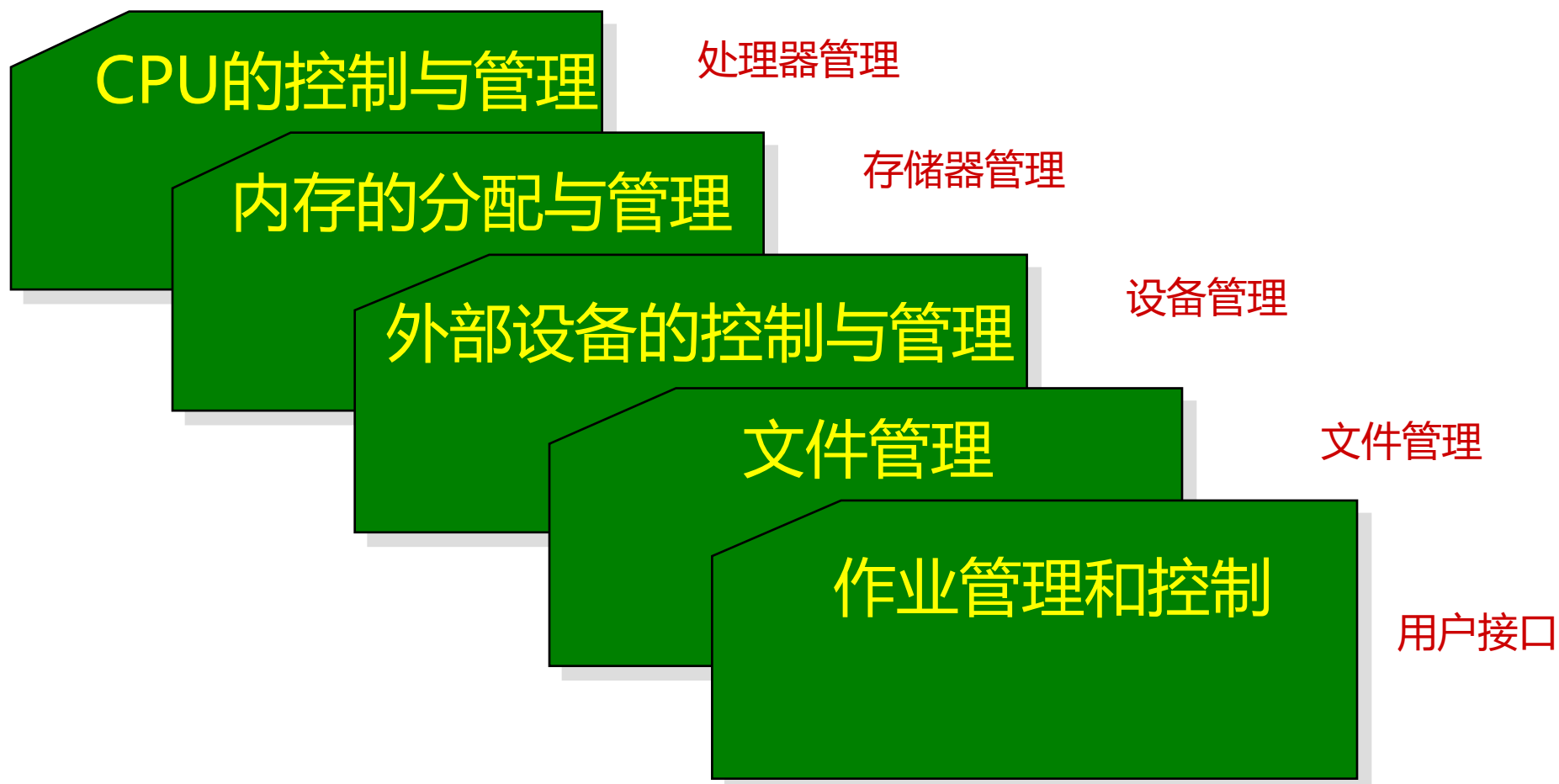
- 外壳程序负责接收用户操作，提供与用户的交互界面。
- 一般操作系统提供给一般用户的界面主要有两种：文本界面；GUI 图形界面。程序员接口：API

□ 管理工具和附属软件

- 一般是操作系统在发布时附带提供给用户的，用户安装完操作系统，就可以利用操作系统自带的管理工具和软件进行一些基本的操作。



操作系统的功能



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

回顾：Hello程序的数据流动过程

Unix> ./hello

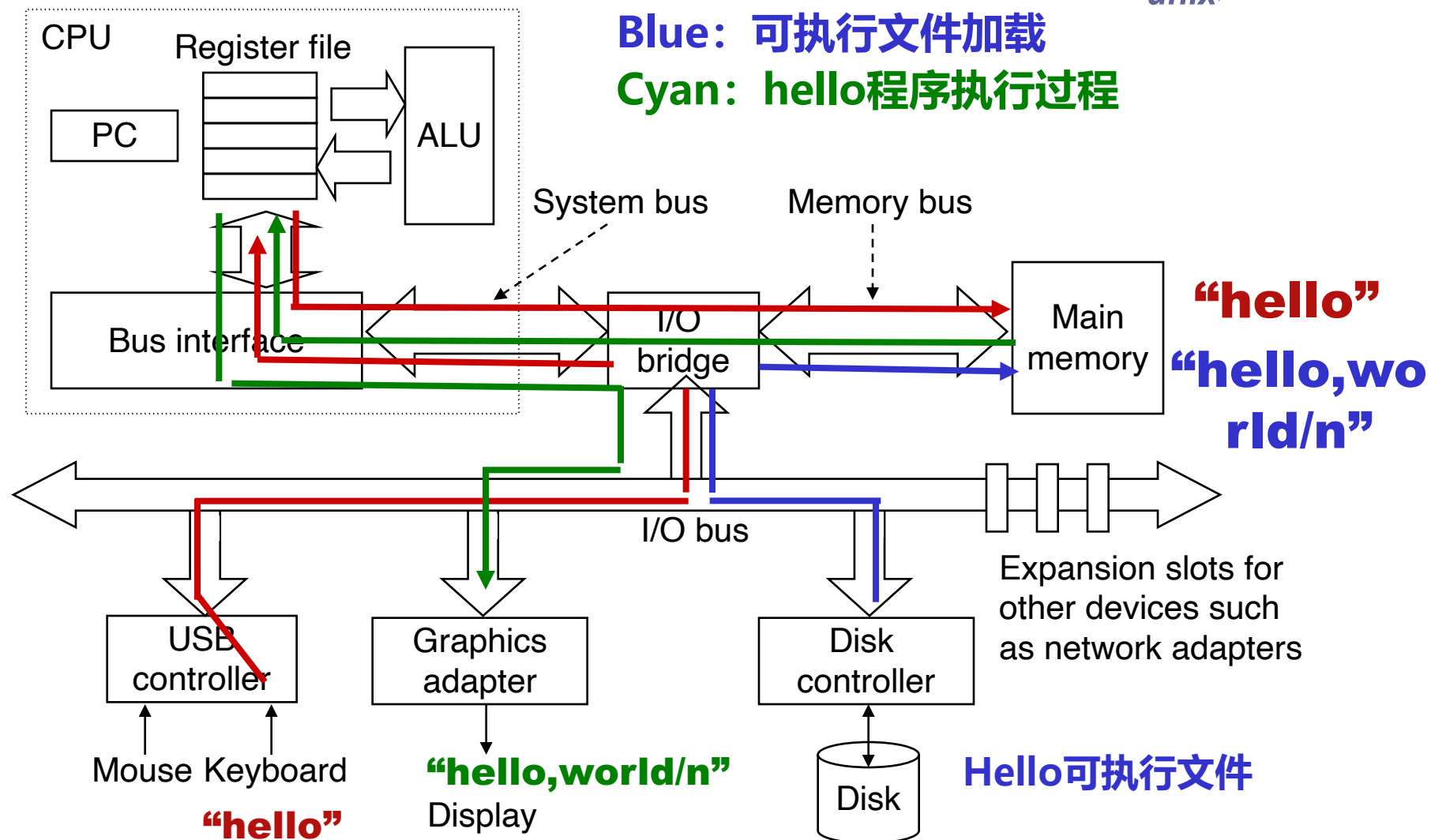
hello, world

unix>

Red: shell命令行处理

Blue: 可执行文件加载

Cyan: hello程序执行过程



操作系统在程序执行过程中的作用

- Shell进程生成子进程，子进程调用execve系统调用启动加载器，以装入Hello程序，并跳转到第一条指令执行
- 在Hello程序执行过程中，Hello本身不会直接访问键盘、显示器、磁盘和主存储器等硬件资源，而是依靠OS提供的服务来间接访问。

例如，利用printf()函数最终调出内核服务程序访问硬件。

- **操作系统**是在应用程序和硬件之间的**中间软件层**。
- 操作系统的两个主要的作用：
 - **硬件资源管理**，以达到以下两个目的：
 - **统筹安排和调度硬件资源**，以防止硬件资源被用户程序滥用
 - **对于广泛使用的复杂低级设备**，为用户程序提供一个简单一致的使用接口
 - **为用户（最终用户、用户程序）使用系统提供一个操作接口**

I/O子系统

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

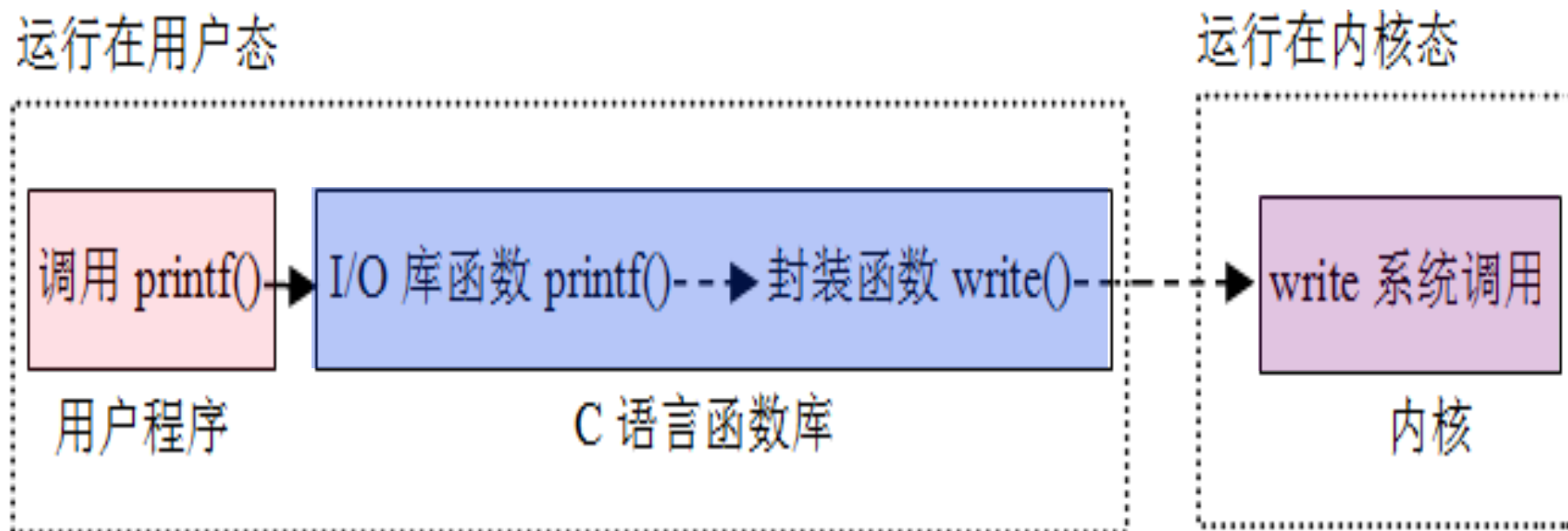
大部分I/O软件都属于操作系统内核态程序, 最初的I/O请求在用户程序中提出。

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。

printf()函数的调用过程如下：

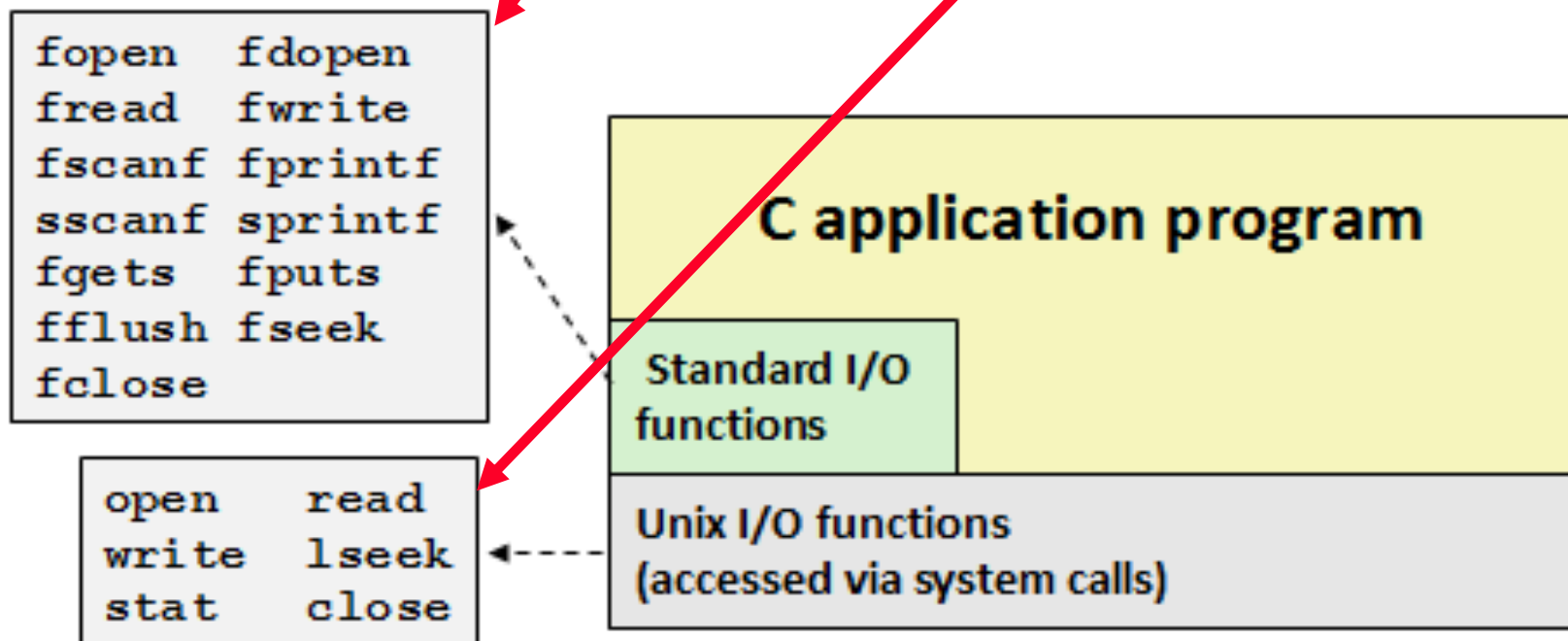


Today

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

UNIX/Linux的I/O

- 用户程序可通过调用特定的I/O函数的方式提出I/O请求。
- 在UNIX/Linux系统中，可以是**C标准I/O库函数**或**系统调用的封装函数**，前者如文件I/O函数fopen()、fread()、fwrite()和fclose()或控制台I/O函数printf()、putc()、scanf()和getc()等；后者如open()、read()、write()和close()等。
- 标准I/O库函数比系统调用封装函数抽象层次高，后者属于**系统级I/O函数**。与系统提供的**API函数**一样，前者是基于后者实现的。

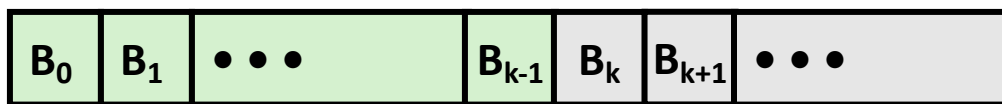


Unix I/O Overview

- A Linux *file* is a sequence of m bytes:
 - $B_0, B_1, \dots, B_k, \dots, B_{m-1}$
- 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()`



Current file position = k

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)

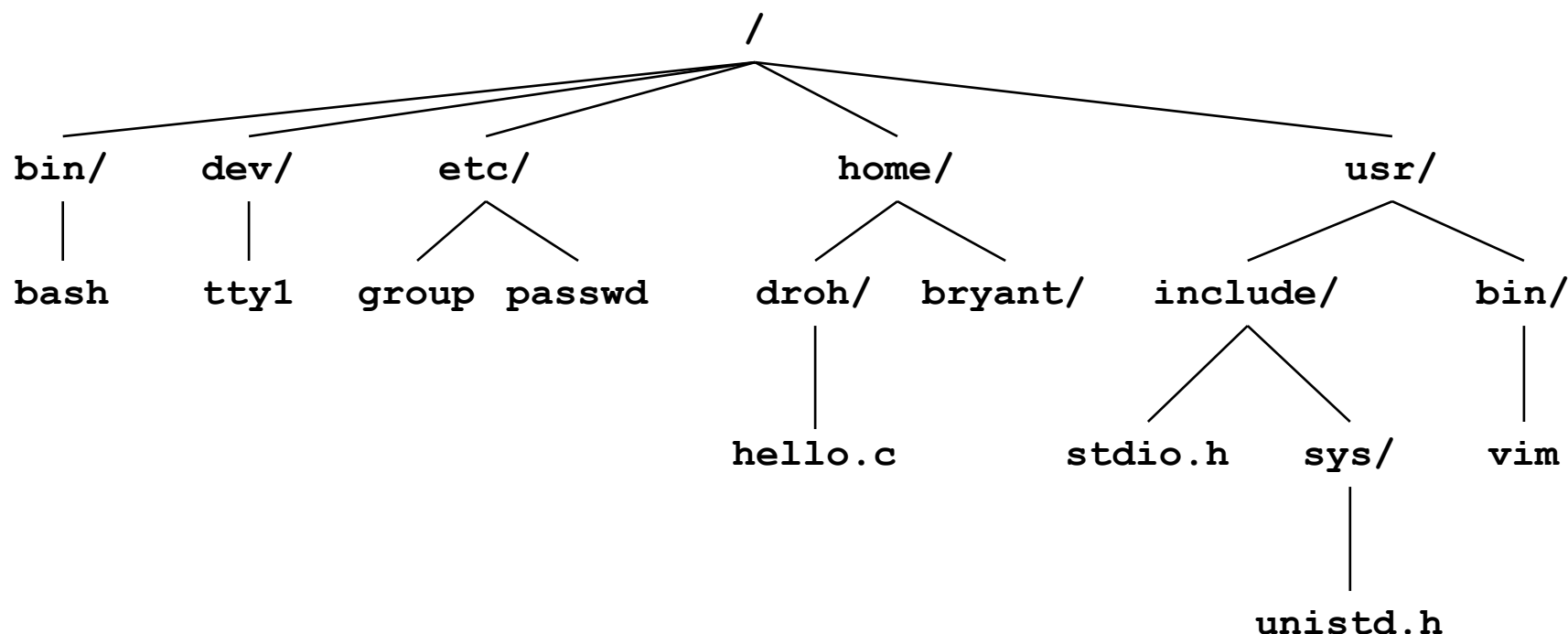


Directories

- **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)

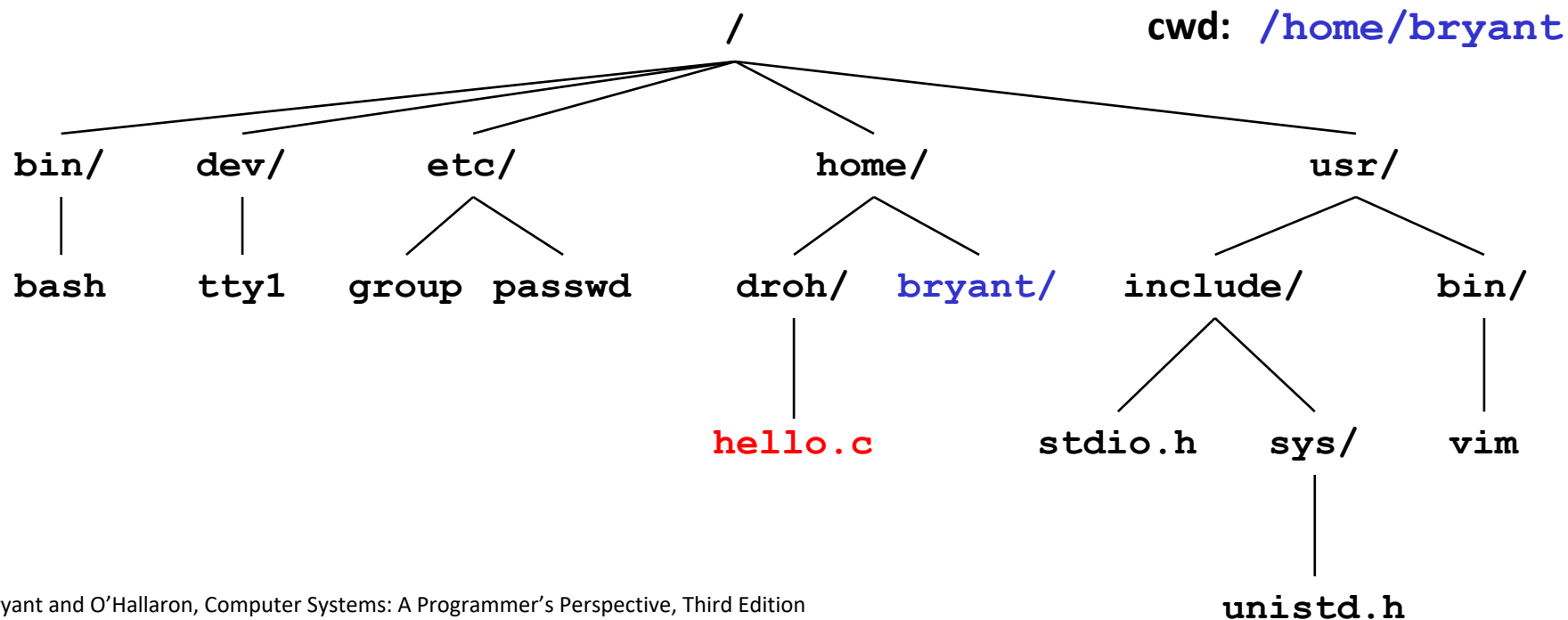


- Kernel maintains *current working directory (cwd)* for each process
 - Modified using the `cd` command

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
- **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
- Closing remarks

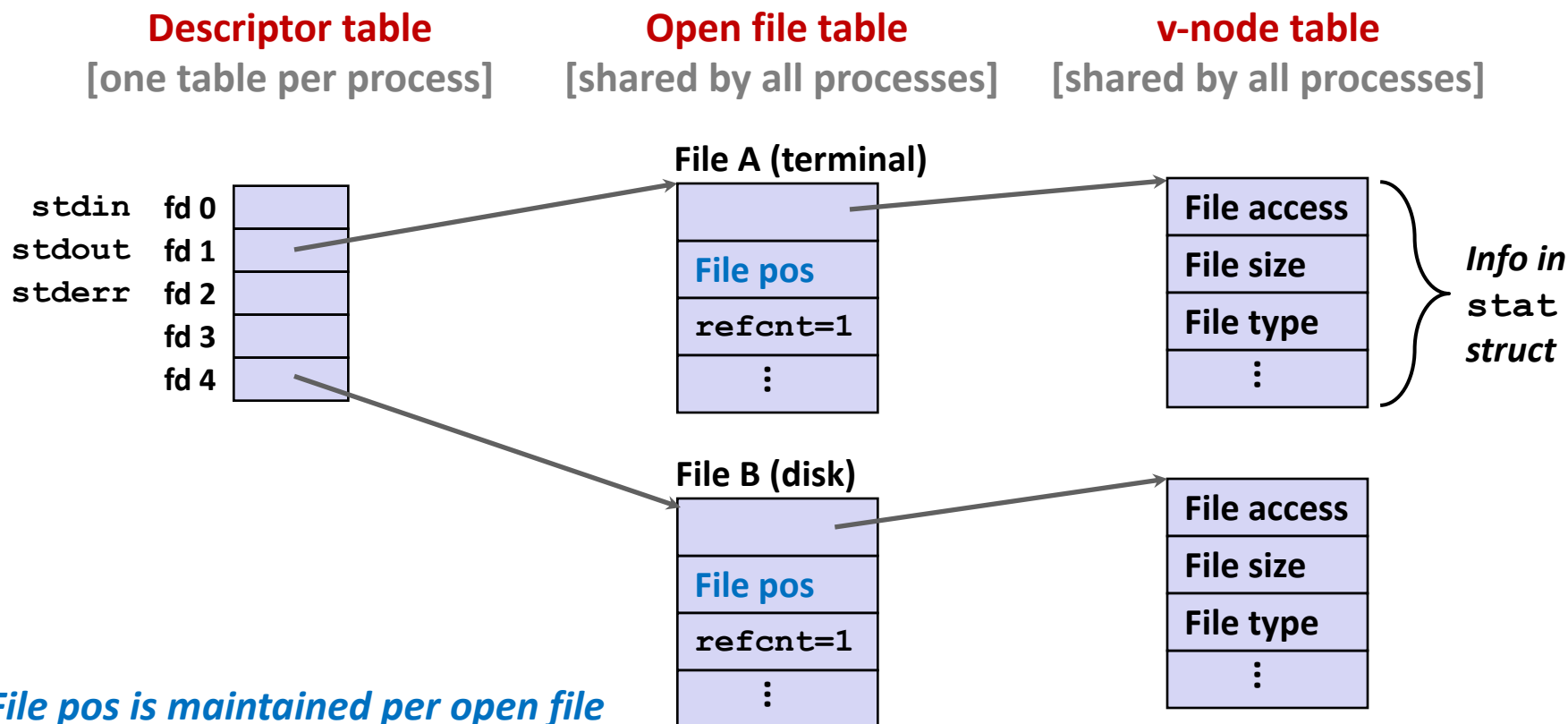
File Metadata

- **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 */
};
```

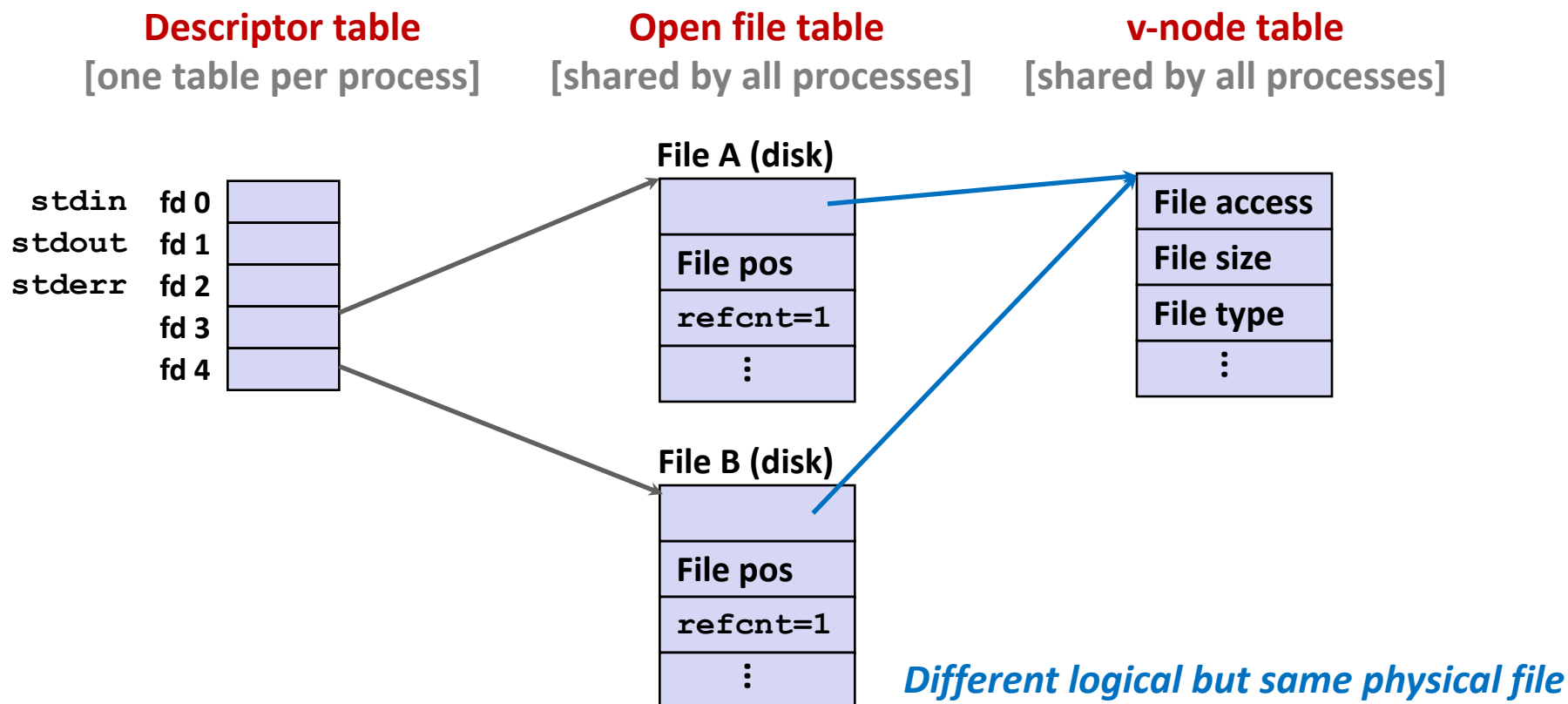
How the Unix Kernel Represents Open Files

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



File Sharing

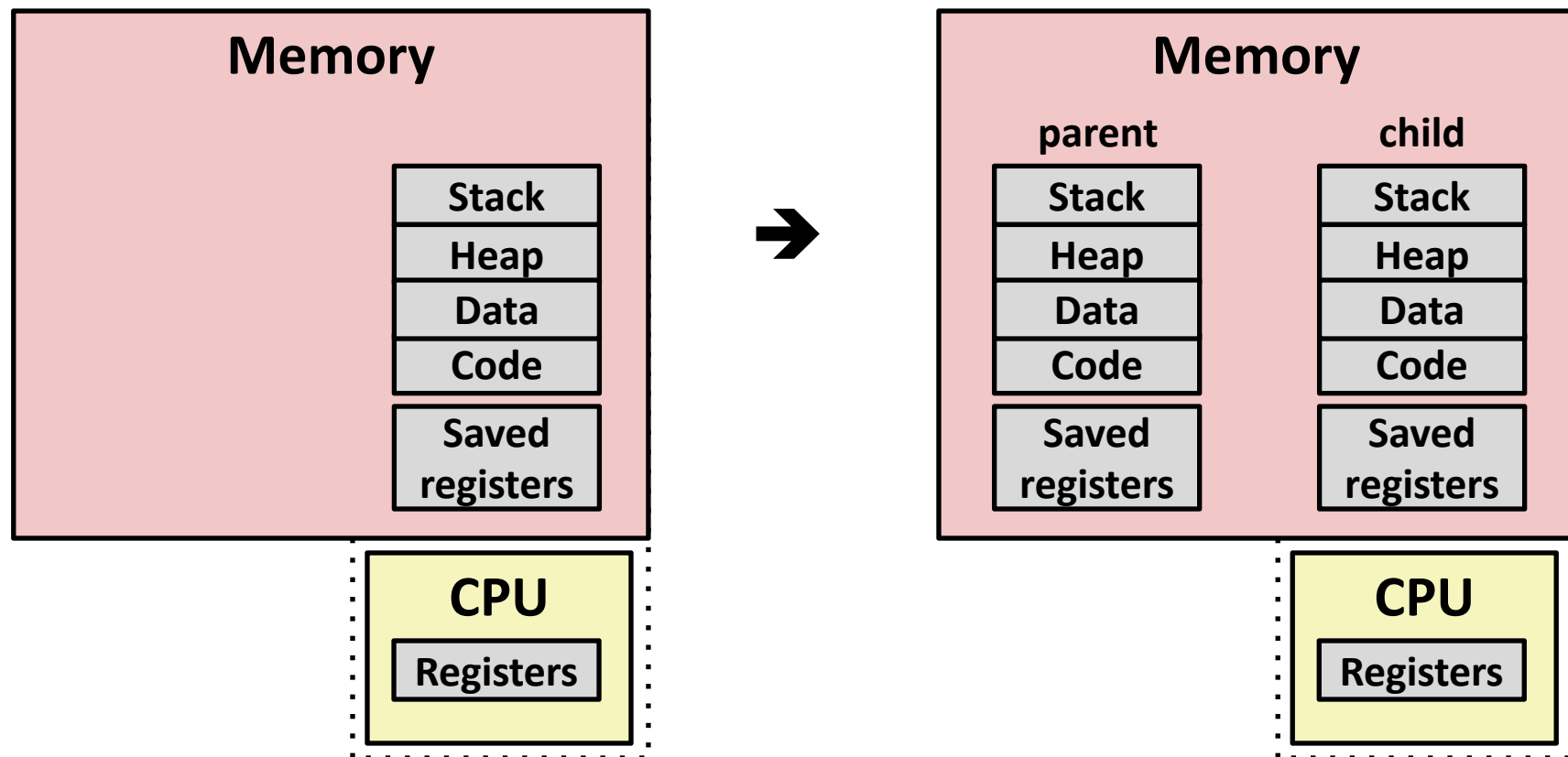
- Two distinct descriptors sharing the same disk file through two distinct open file table entries
 - E.g., Calling `open` twice with the same `filename` argument



Creating Processes

- *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

```
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
```


Modeling fork with Process Graphs

- **A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program:**
 - Each vertex is the execution of a statement
 - $a \rightarrow b$ means a happens before b
 - Edges can be labeled with current value of variables
 - `printf` vertices can be labeled with output
 - Each graph begins with a vertex with no inedges
- **Any *topological sort* of the graph corresponds to a feasible total ordering.**
 - Total ordering of vertices where all edges point from left to right

Process Graph Example

```

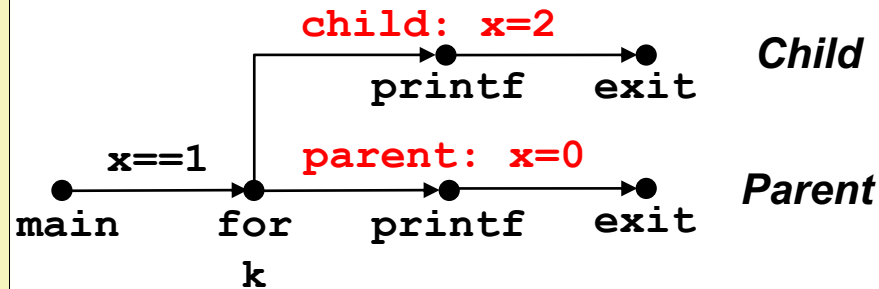
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

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

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

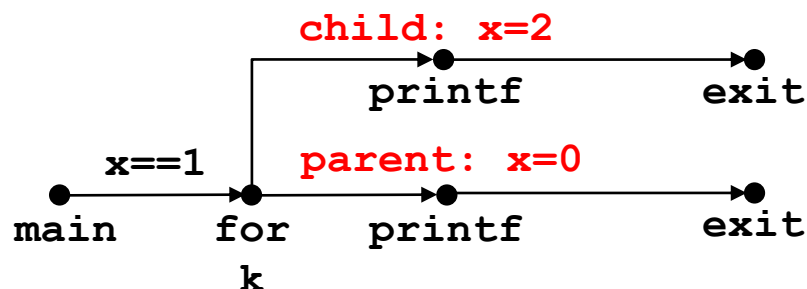
```

fork.c

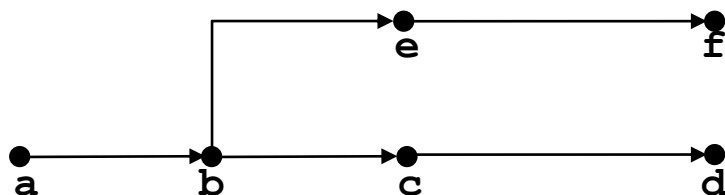


Interpreting Process Graphs

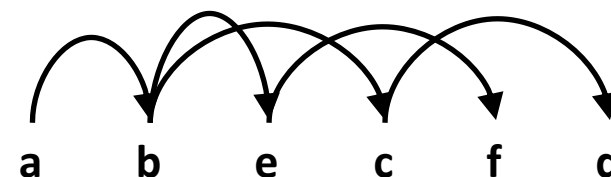
■ Original graph:



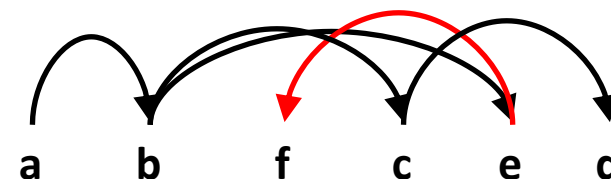
■ Relabled graph:



Feasible total ordering:

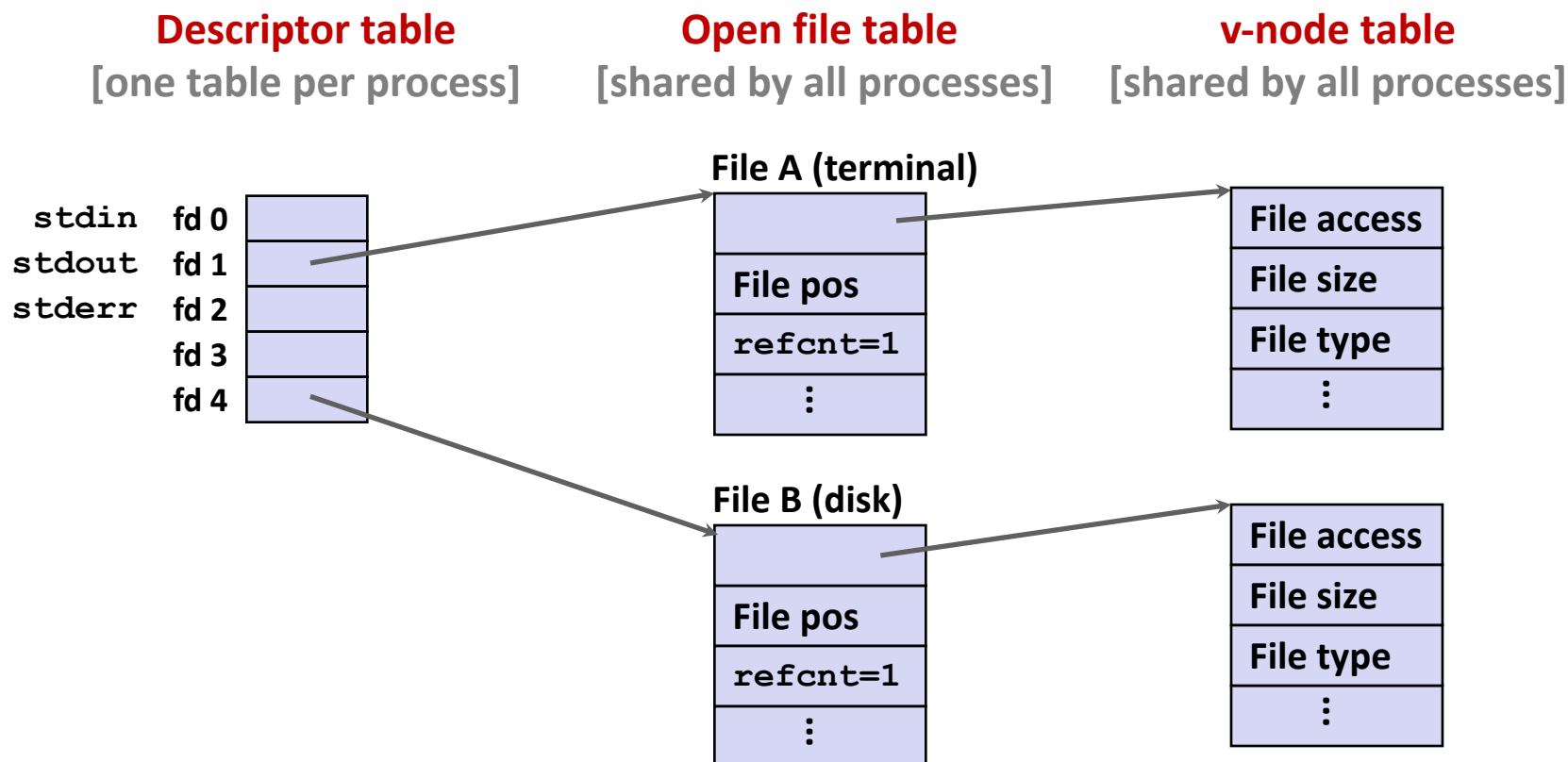


Infeasible total ordering:



How Processes Share Files: `fork`

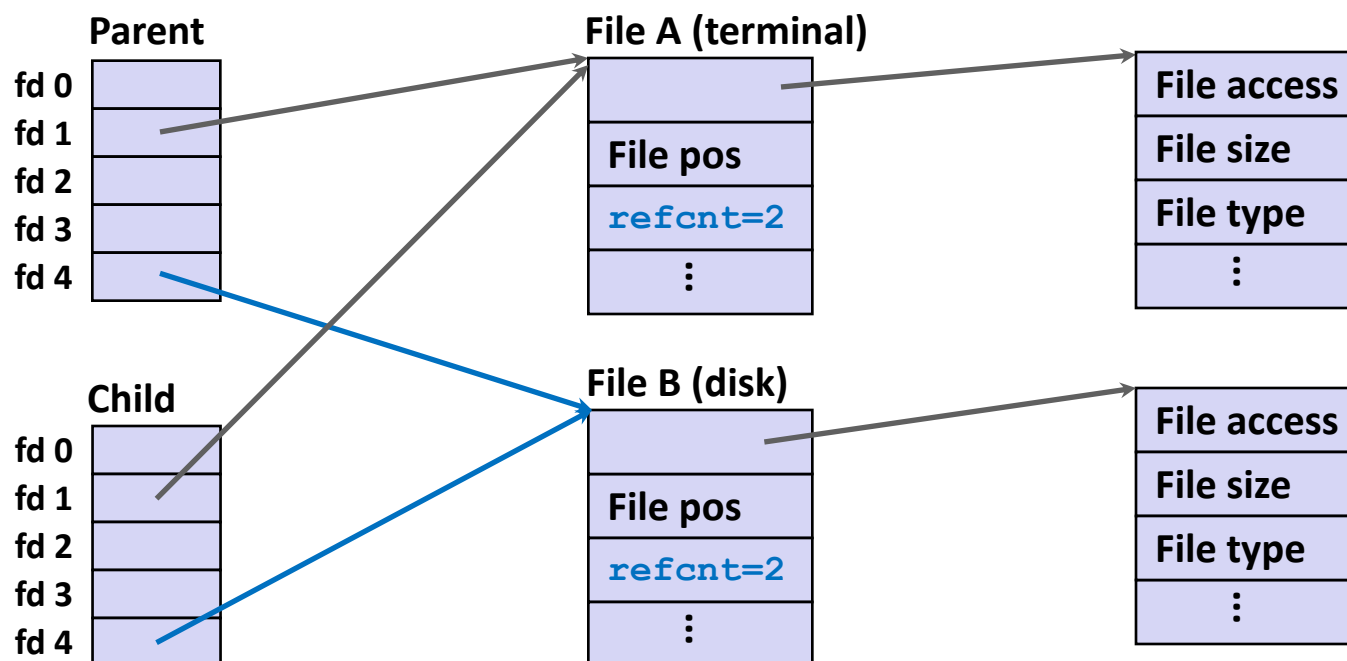
- A child process inherits its parent's open files
 - Note: situation unchanged by `exec` functions (use `fcntl` to change)
- *Before* `fork` call:



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

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



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

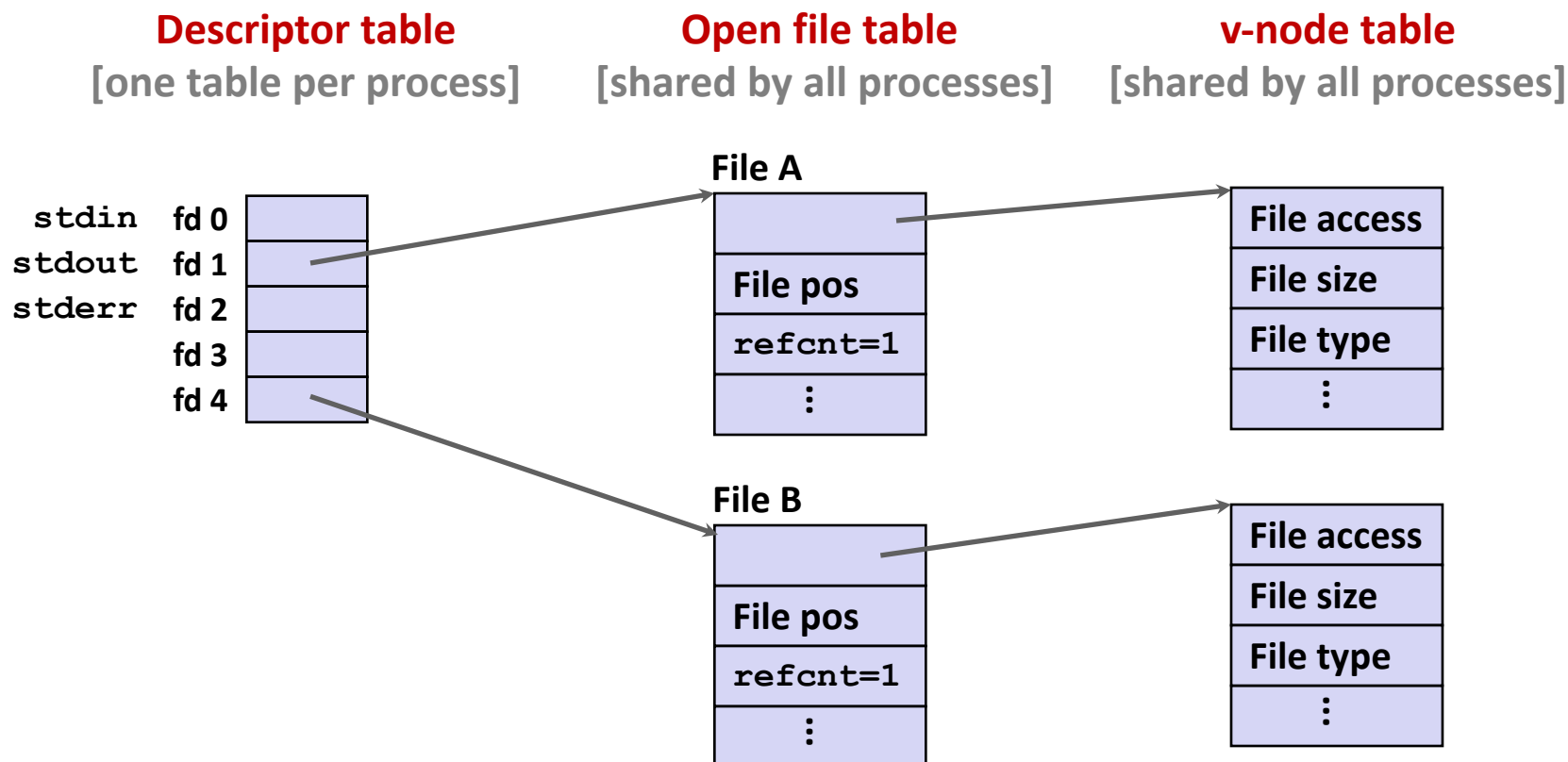


Descriptor table
after `dup2 (4, 1)`

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`

Descriptor table

[one table per process]

stdin	fd 0	
stdout	fd 1	
stderr	fd 2	
	fd 3	
	fd 4	

Open file table

[shared by all processes]

File A

File pos
refcnt=0
⋮

File B

File pos
refcnt=2
⋮

v-node table

[shared by all processes]

File access
File size
File type
⋮

File access
File size
File type
⋮

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**
- Closing remarks

I/O子系统

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。

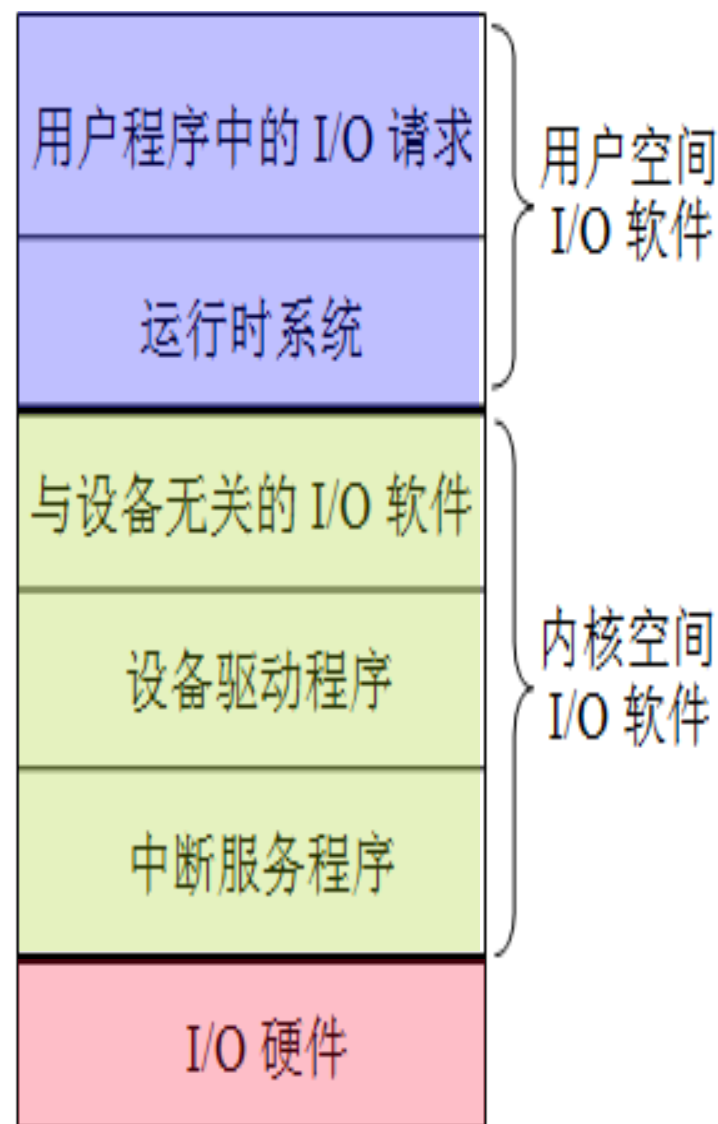
I/O子系统(软件观点)

- 所有高级语言的运行时 (runtime) 都提供了执行I/O功能的机制

例如, C语言中提供了包含像**printf()**和**scanf()**等这样的标准I/O库函数, C++语言中提供了如 **<< (输入)** 和 **>> (输出)** 这样的重载操作符。

- 从高级语言程序中通过I/O函数或I/O操作符提出I/O请求, 到设备响应并完成I/O请求, 涉及到多层次I/O软件和I/O硬件的协作。
- I/O子系统也采用层次结构

从用户I/O软件切换到内核I/O软件的唯一办法是“异常”机制: **系统调用 (自陷)**



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`)

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。

以hello程序为例说明

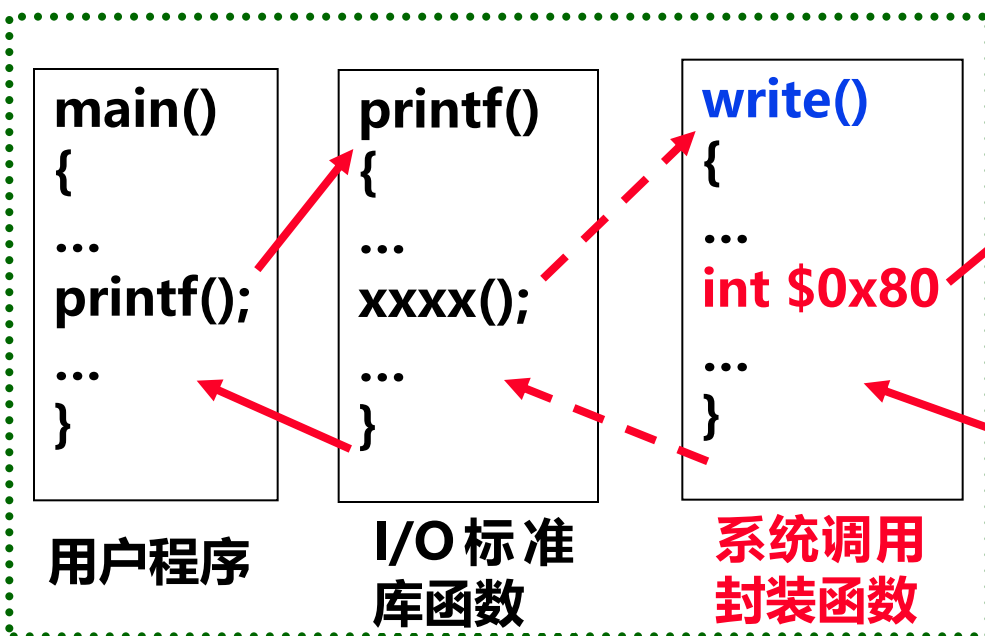
假定以下用户程序对应的进程为p

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

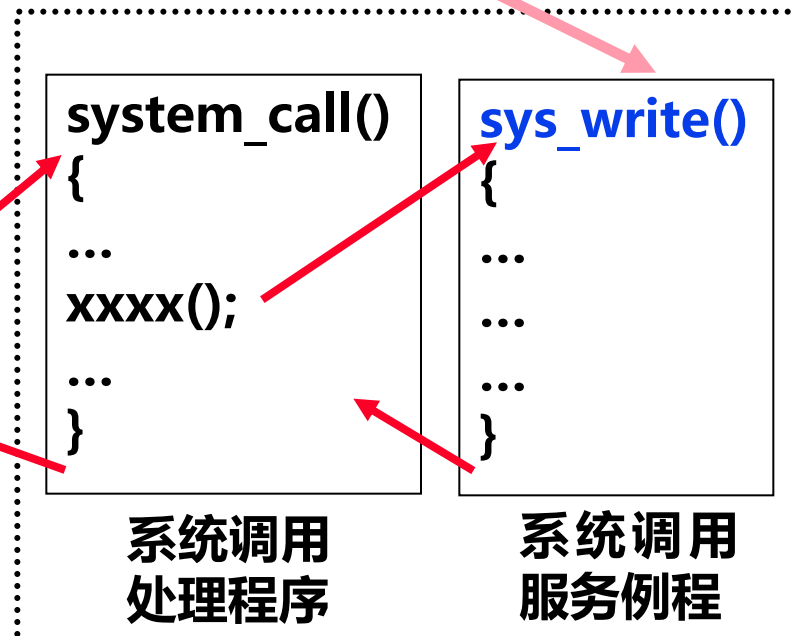
sys_write可用三种I/O方式实现：
程序查询、中断 和 DMA

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

用户空间、运行在用户态



内核空间、运行在内核态



Standard I/O Streams

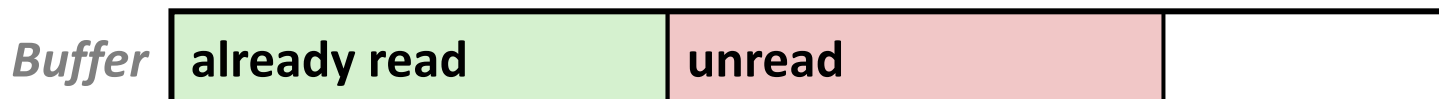
- Standard I/O models open files as *streams*
 - Abstraction for a file descriptor and a buffer in memory
- C programs begin life with three open streams (defined in `stdio.h`)
 - `stdin` (standard input)
 - `stdout` (standard output)
 - `stderr` (standard error)

```
#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");
}
```

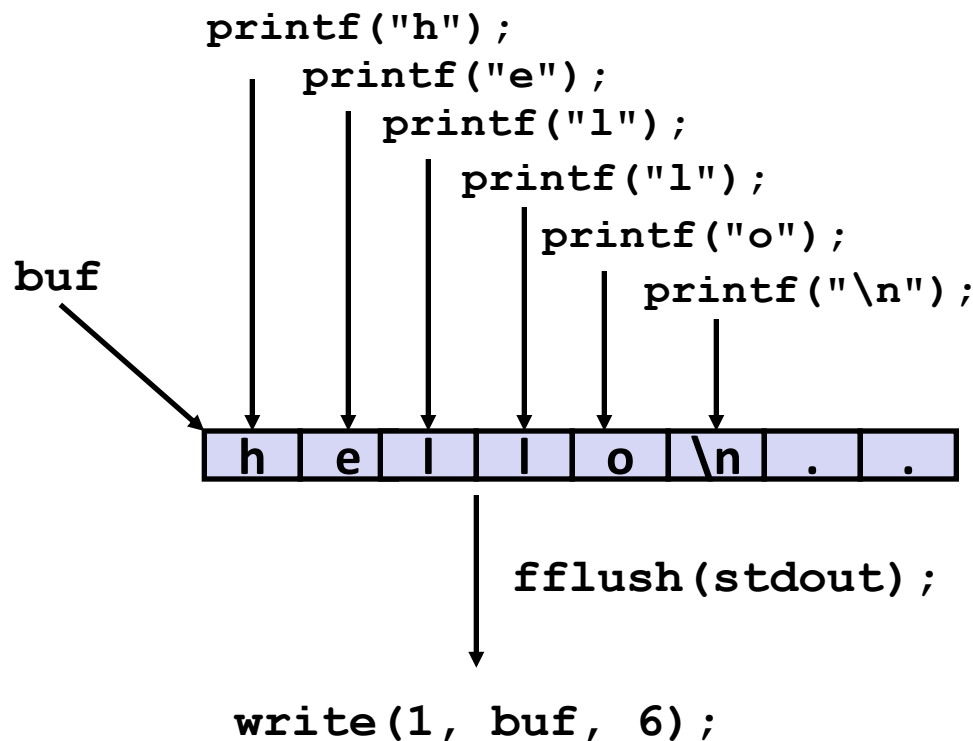
Buffered I/O: Motivation

- 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);
}
```

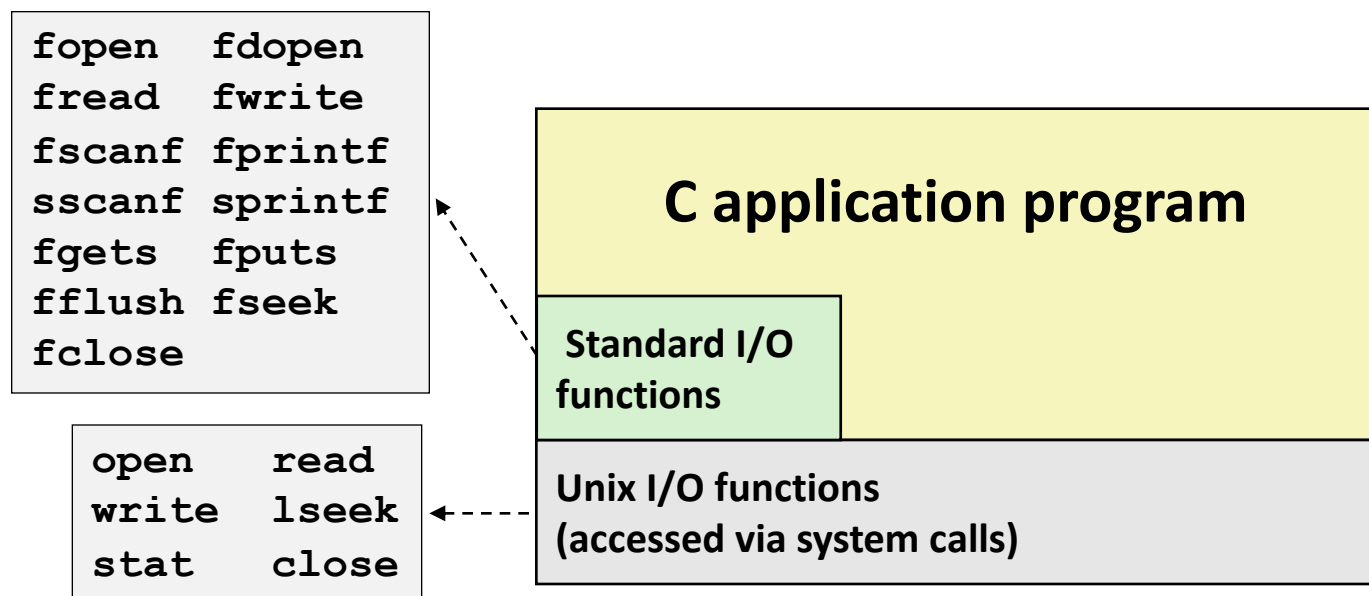
```
linux> strace ./hello
execve("./hello", ["hello"], [/* ... */]).
...
write(1, "hello\n", 6)                = 6
...
exit_group(0)                         = ?
```

Today

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

Standard I/O vs. Unix I/O

- Standard I/O is implemented using low-level Unix I/O



- Which ones should you use in your programs?

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。

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)

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

系统I/O软件

OS在I/O子系统的重要性由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。

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

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

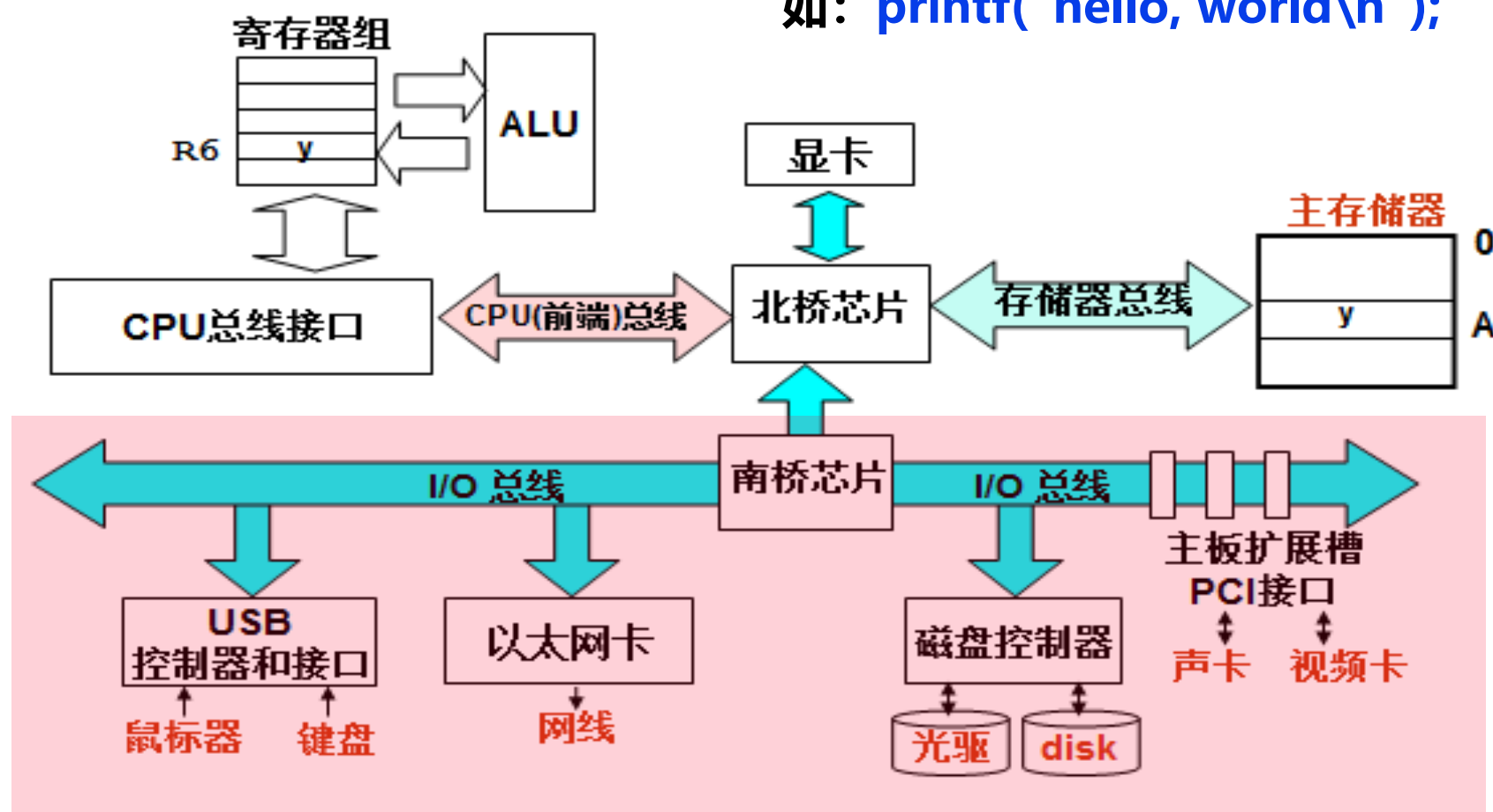
I/O硬件的组成

I/O硬件建立了外设与主机之间的“通路”：

主机----I/O总线（桥）----设备控制器----电缆----外设

如何把用户I/O请求转换为对设备的控制命令并完成设备I/O任务，需要I/O软件与I/O硬件之间的协调工作

如： `printf("hello, world\n");`



I/O Hardware

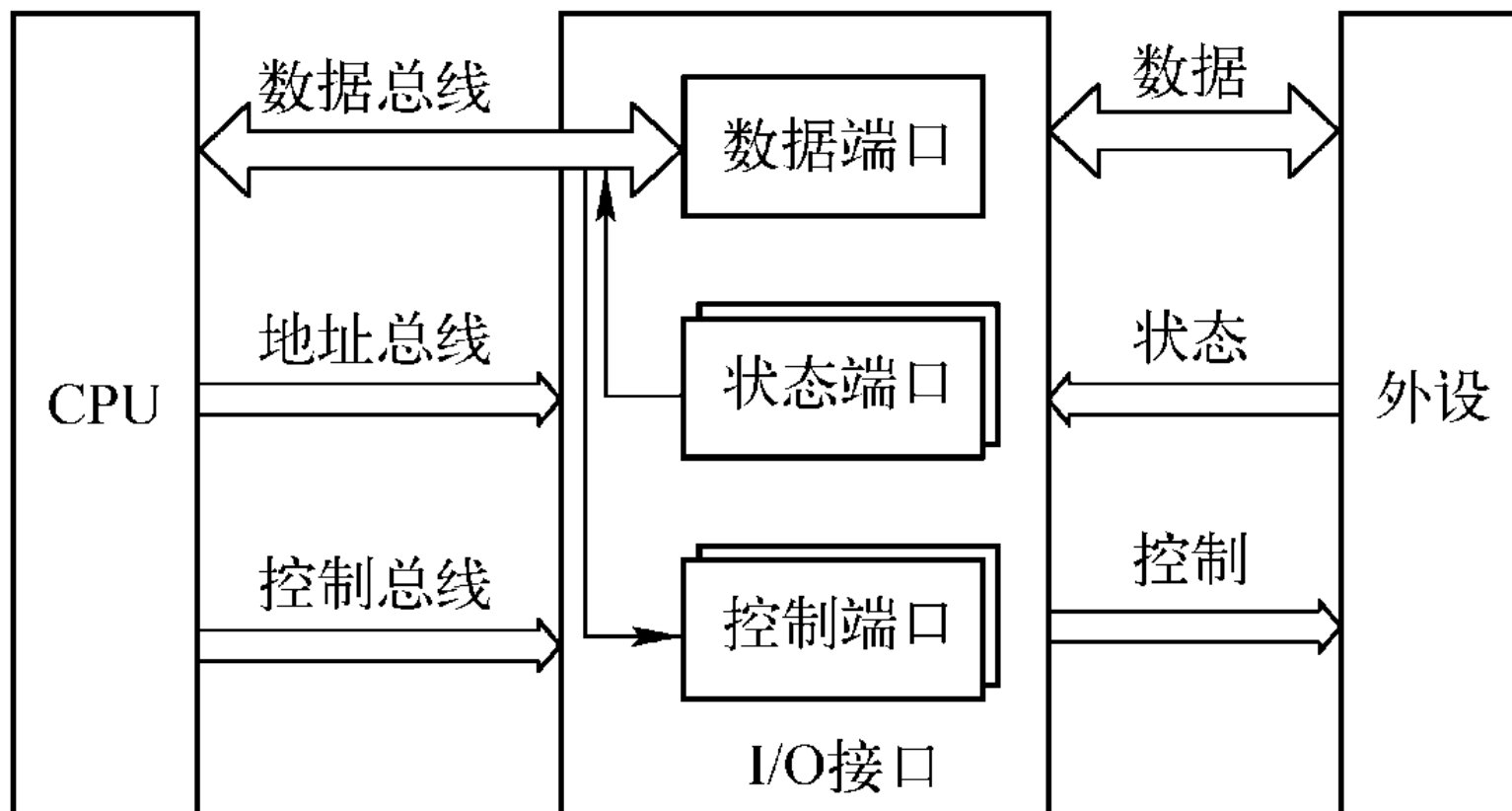
■ Incredible variety of I/O devices

- 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

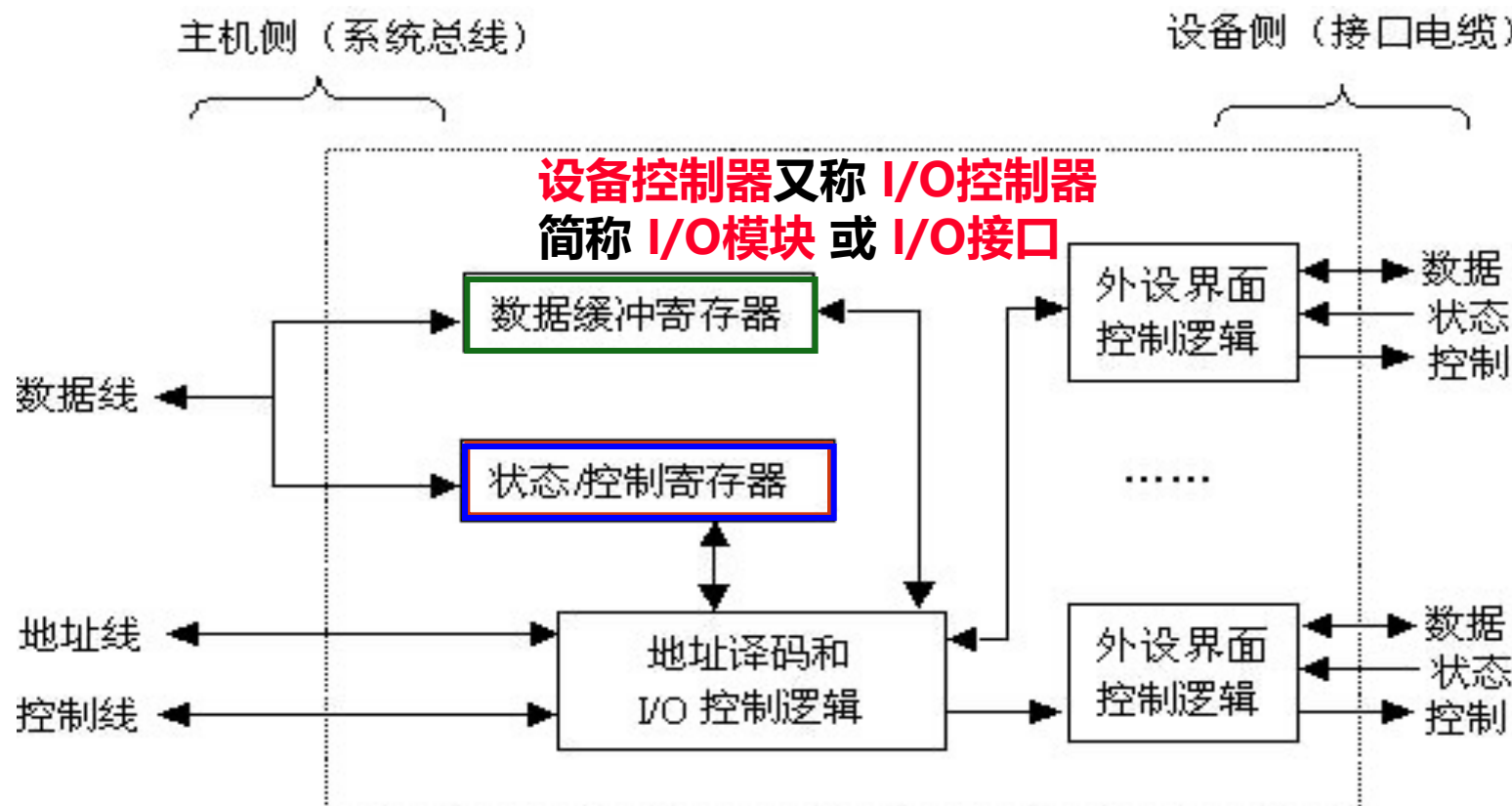


I/O Hardware (Cont.)

- 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)

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

I/O端口的寻址方式

- 对I/O端口读写就是向I/O设备送出命令或从设备读状态或读/写数据
 - 一个I/O控制器可能会占有多个端口地址
 - I/O端口必须编号后，CPU才能访问它
 - I/O设备的寻址方式就是I/O端口的编号方式
- 教室和办公室可以连号（统一编址），也可单独编号（独立编址）

(1) 统一编址方式（内存映射方式）

与主存空间统一编址，主存单元和I/O端口在同一个地址空间中。

（将I/O端口映射到某个主存区域，故也称“存储器映射方式”）

例如，RISC机器、Motorola公司的处理器等采用该方案

VRAM（显示存储器）通常也和主存统一编址

(2) 独立编址方式（特殊I/O指令方式）

单独编号，不和主存单元一起编，使成为一个独立的I/O地址空间

（因为需专门I/O指令，故也称为“特殊I/O指令方式”）

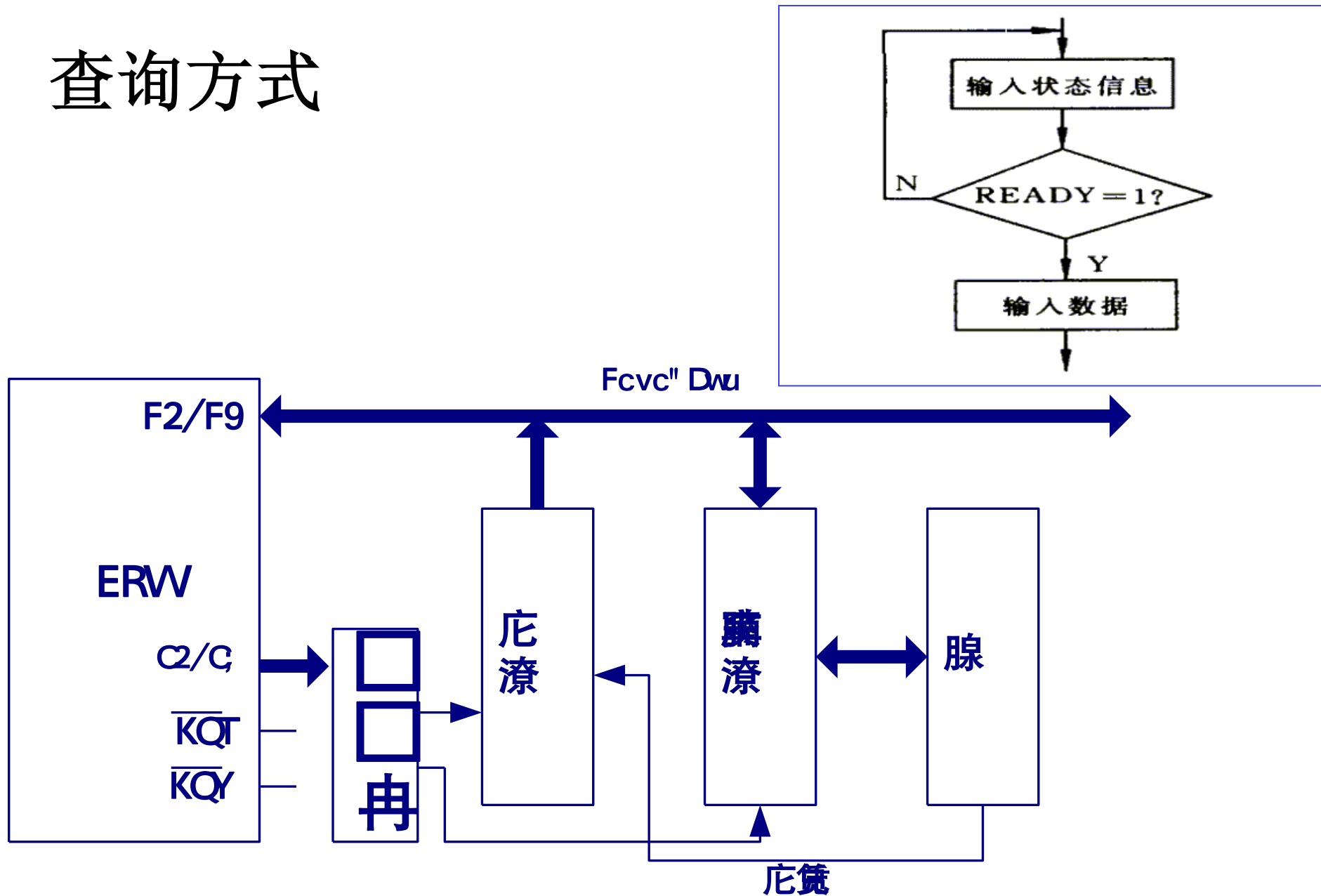
例如，Intel公司和Zilog公司的处理器就是独立编址方式

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
 - 无条件传统方式
 - 查询方式
- **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



程序查询（Polling）方式

- I/O设备（包括设备控制器）将自己的状态放到**状态寄存器**中
 - 打印缺纸、打印机忙、未就绪等都是状态
- OS阶段性地查询状态寄存器中的特定状态，以决定下一步动作
 - 如：未“就绪”时，则一直“等待”
- 例如：sys_write进行字符串打印的程序段大致过程如下：

```
copy_string_to_kernel ( strbuf, kernelbuf, n); // 将字符串复制到内核缓冲区
for (i=0; i < n; i++) {                          // 对于每个打印字符循环执行
    while ( printer_status != READY);              // 等待直到打印机状态为“就绪”
    *printer_data_port=kernelbuf[i];               // 向数据端口输出一个字符
    *printer_control_port=START;                   // 发送“启动打印”命令
}
return_to_user ( );                               // 返回用户态
```

如何判断“就绪”？如何“等待”？

读取状态寄存器，判断特定位（1-就绪；0-未就绪）是否为1

等待：读状态、判断是否为1；不是，则继续读状态、判断、……


打印输出标准子程序

功能：打印AL寄存器中的字符。

```

PRINT    PROC NEAR
          PUSH AX          ; 保留用到的寄存器
          PUSH DX          ; 保留用到的寄存器
          MOV DX, 378H      ; 数据锁存器口地址送DX
          OUT DX, AL        ; 输出要打印的字符到数据锁存器
          MOV DX, 379H      ; 状态寄存器口地址送DX
WAIT:    IN AL, DX          ; 读打印机状态位
          TEST AL, 80H      ; 检查忙位
          JE WAIT           ; 等待直到打印机不忙
          MOV DX, 37AH      ; 命令(控制)寄存器口地址送DX
          MOV AL, 0DH        ; 置选通位=1 (表示启动打印)
          OUT DX, AL        ; 使命令寄存器中选通位置1
          POP DX
          POP AX            ; 恢复寄存器
          RET
PRINT    ENDP

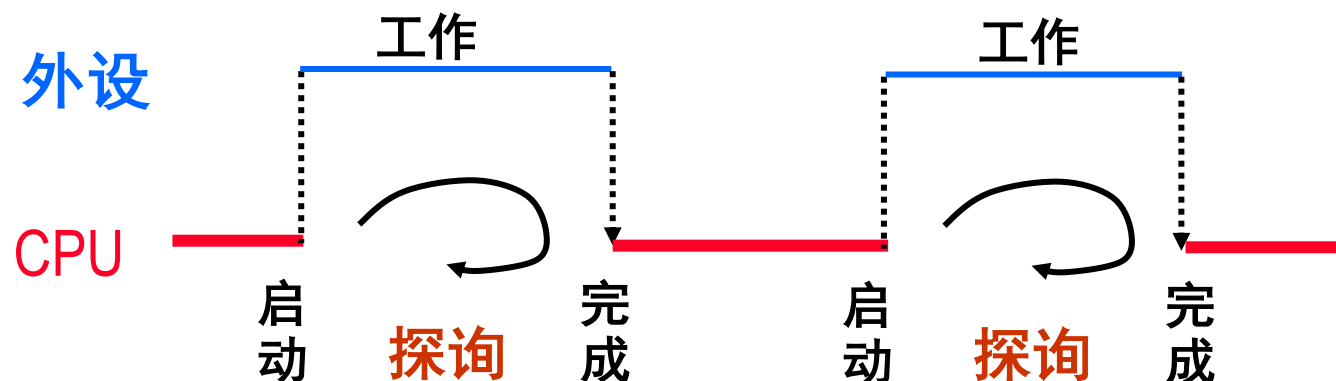
```



回顾：过程/函数/子程序中的开始总是先
要保护现场，最后总是要恢复现场！

程序查询I/O方式

sys_write系统调用服务例程



“踏步”现象

此时，CPU处于停止状态吗？

不是！只是不断执行“IN-TEST-JE”
3条指令，称为“忙等待”！

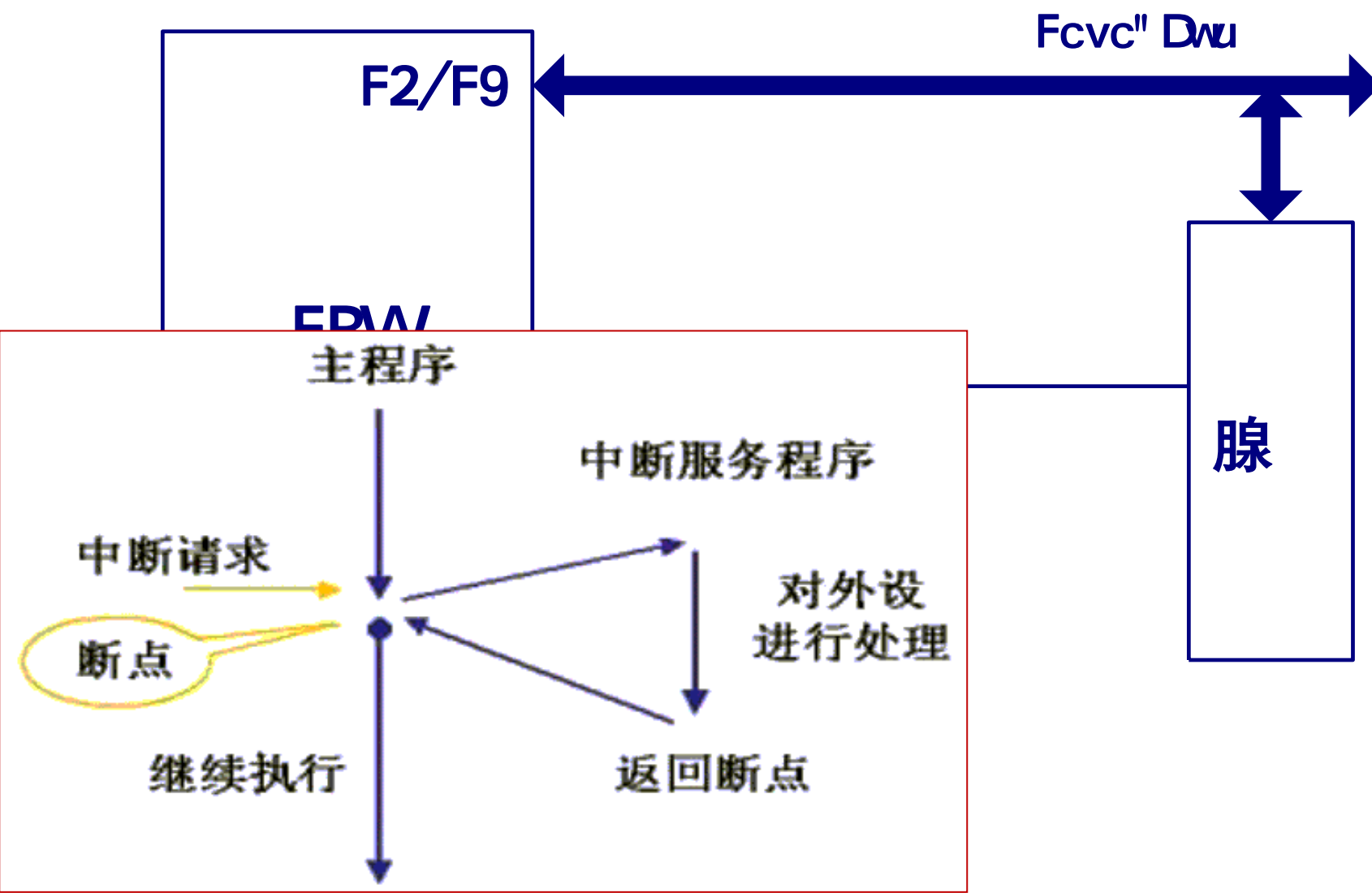
“查询”期间，可一直不断查询（独占查询），
也可定时查询（需保证数据不丢失！）。

特点：

- 简单、易控制、外围接口控制逻辑少；
- CPU与外设串行工作，效率低、速度慢，适合于慢速设备
- 查询开销极大（CPU完全在等待“外设完成”）

工作方式：完全串行或部分串行，CPU用100%的时间为I/O服务！

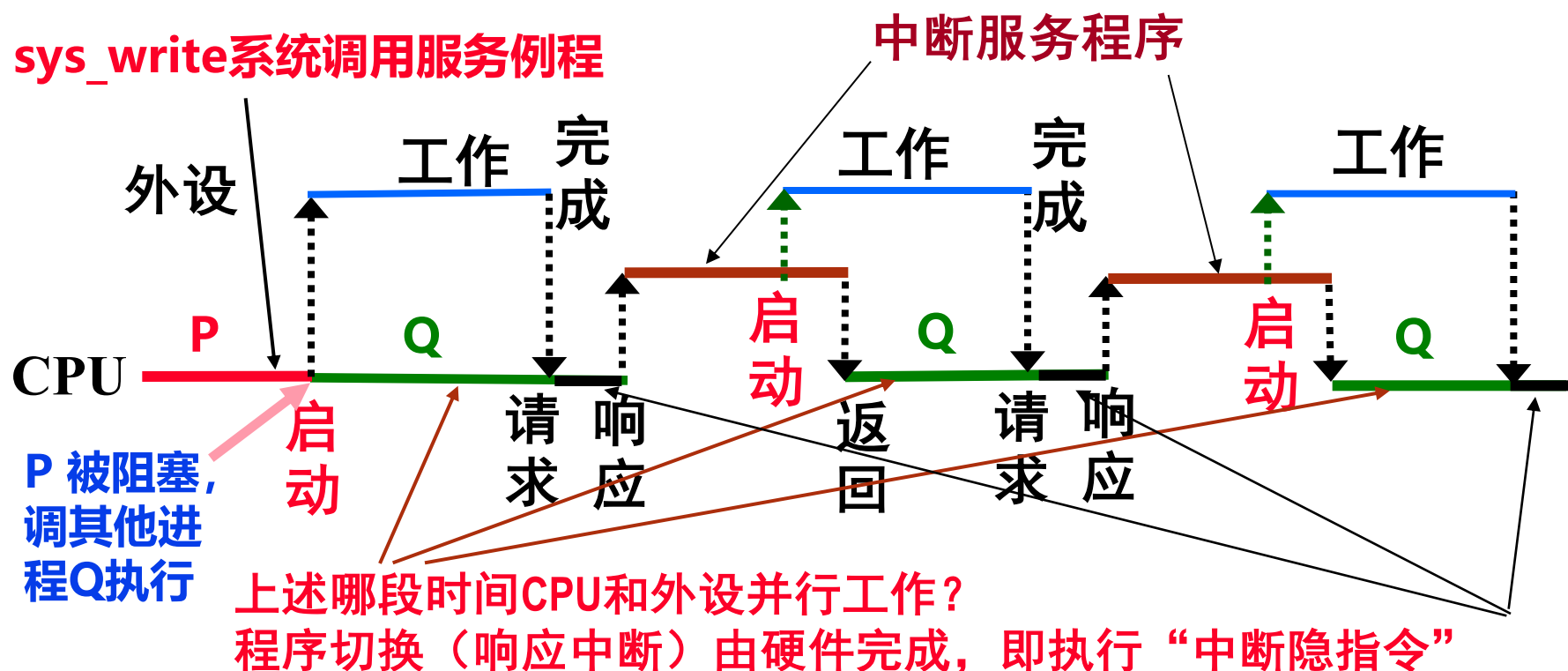
中断方式



中断I/O方式

基本思想：

当外设准备好 (ready) 时，便向CPU发中断请求，CPU响应后，中止现行政程序的执行，转入“**中断服务程序**”进行输入/出操作，以实现主机和外设接口之间的数据传送，并启动外设工作。“中断服务程序”执行完后，返回原被中止的程序断点处继续执行。此时，外设和CPU并行工作。



中断I/O方式

例子：采用中断方式进行字符串打印

sys_write进行字符串打印的程序段：

```
copy_string_to_kernel ( strbuf, kernelbuf, n); // 将字符串复制到内核缓冲区
enable_interrupts ( ); // 开中断，允许外设发出中断请求
while ( printer_status != READY); // 等待直到打印机状态为“就绪”
*printer_data_port=kernelbuf[i]; // 向数据端口输出第一个字符
*printer_control_port=START; // 发送“启动打印”命令
scheduler ( ); // 阻塞用户进程P，调度其他进程执行
```

“字符打印” 中断服务程序：

```
if (n==0) { // 若字符串打印完，则
    unblock_user ( ); // 用户进程P解除阻塞，P进就绪队列
} else {
    *printer_data_port=kernelbuf[i]; // 向数据端口输出一个字符
    *printer_control_port=START; // 发送“启动打印”命令
    n = n-1; // 未打印字符数减1
    i = i+1; // 下一个打印字符指针加1
}
acknowledge_interrupt(); // 中断回答（清除中断请求）
return_from_interrupt(); // 中断返回
```

**sys_write
是如何调出
来的？**

系统调用！

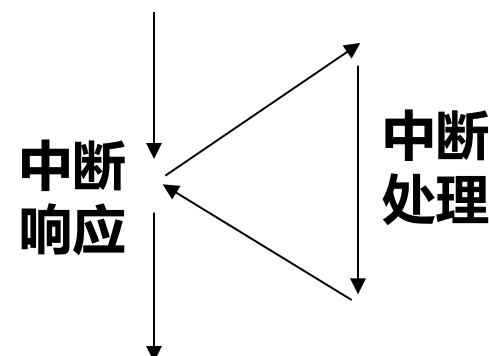
**中断服务程
序是如何调
出来的？**

**外设完成任
务！**

中断I/O方式

- 中断过程

- 中断检测（硬件实现）
- 中断响应（硬件实现）
- 中断处理（软件实现）



- 中断响应

- 中断响应是指主机发现外部中断请求，中止现行政程序的执行，到调出中断服务程序这一过程。

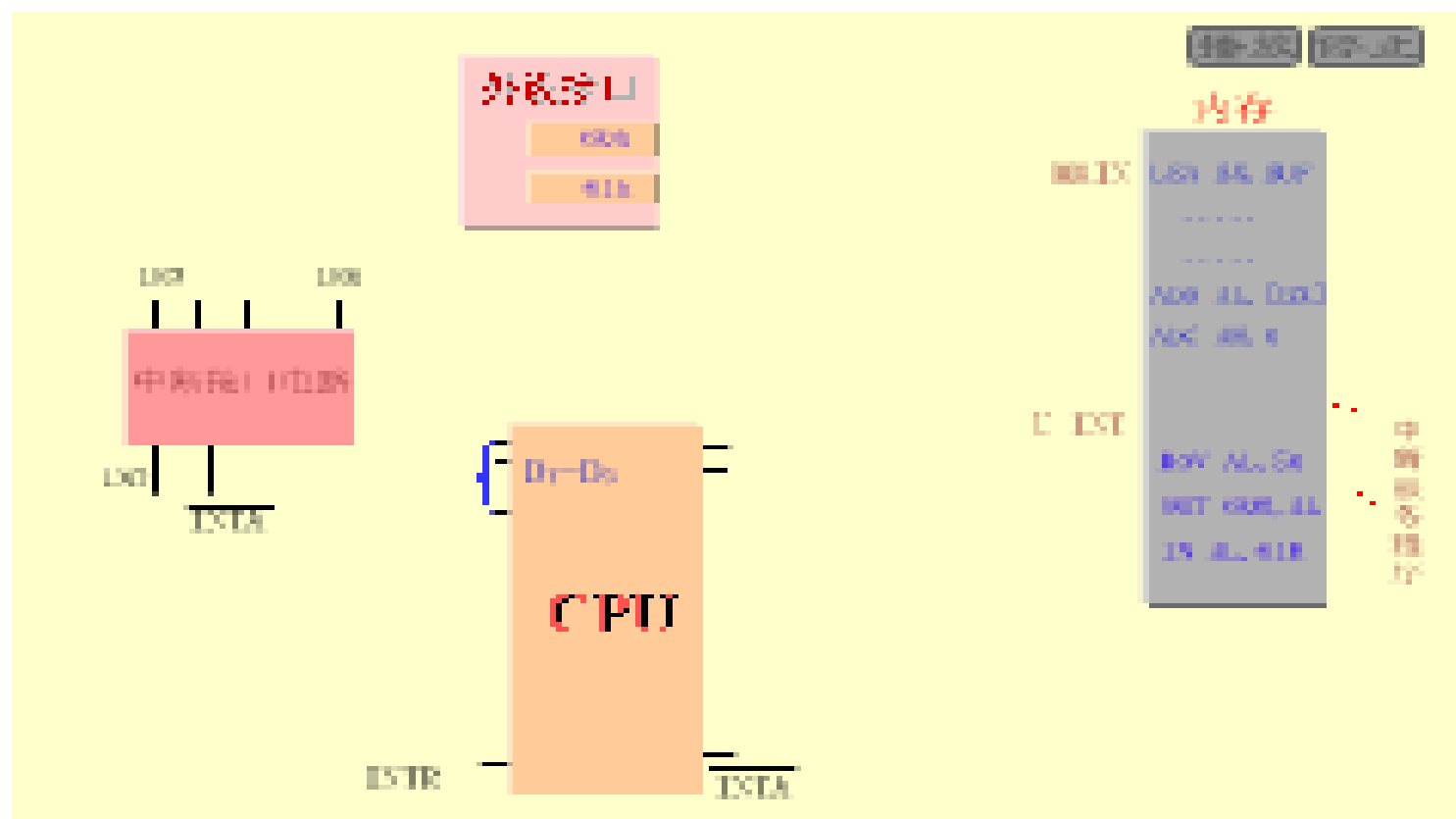
中断响应的条件

- ① CPU处于开中断状态
- ② 在一条指令执行完
- ③ 至少要有有一个未被屏蔽的中断请求

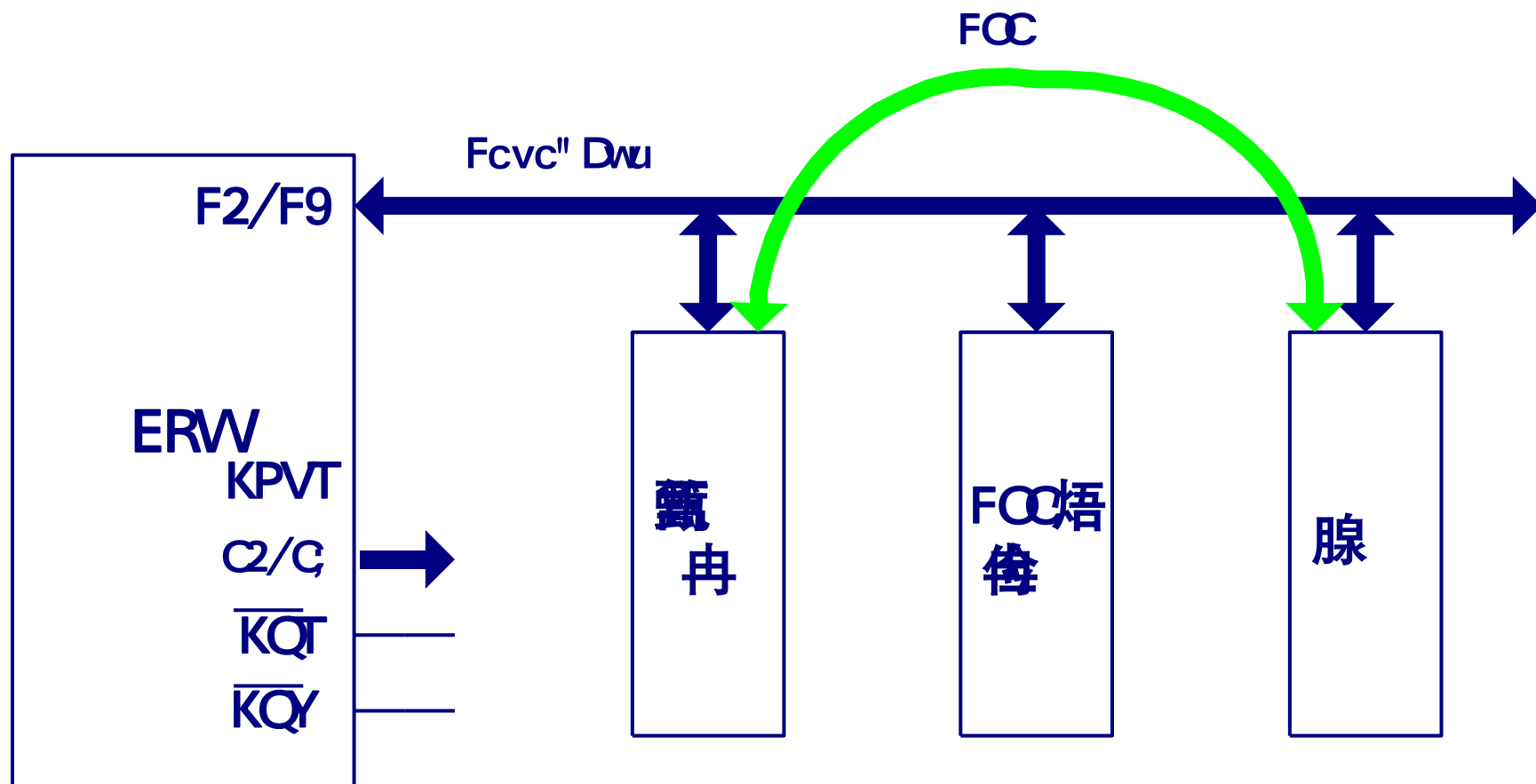
问题：中断响应的时点与异常处理的时点是否相同？为什么？

通常在指令执行结束时查询有无中断请求，有则立即响应；而异常发生在指令执行过程中，一旦发现则马上处理。

中断处理流程



DMA方式



DMA方式的基本要点

- DMA方式的基本思想
 - 在高速外设和主存间直接传送数据
 - 由专门硬件（即：DMA控制器）控制总线进行传输
- DMA方式适用场合
 - 高速设备（如：磁盘、光盘等）
 - 成批数据交换，且数据间间隔时间短，一旦启动，数据连续读写
- 采用“请求-响应”方式
 - 每当高速设备准备好数据就进行一次“DMA请求”，DMA控制器接受到DMA请求后，申请总线使用权
 - DMA控制器的总线使用优先级比CPU高，为什么？
- 与中断控制方式结合使用
 - 在DMA控制器控制总线进行数据传送时，CPU执行其他程序
 - DMA传送结束时，要通过“DMA结束中断”告知CPU

DMA方式下CPU的工作

例子：采用DMA方式进行字符串输出

sys_write进行字符串输出的程序段：

```
copy_string_to_kernel(strbuf, kernelbuf, n); // 将字符串复制到内核缓冲区
initialize_DMA ( );                        // 初始化DMA控制器（准备传送参数）
*DMA_control_port=START; // 发送 “启动DMA传送” 命令
scheduler ( );                          // 阻塞用户进程P，调度其他进程执行
```

DMA控制器接受到“启动”命令后，控制总线进行DMA传送。通常用“**周期挪用法**”：设备每准备好一个数据，挪用一次“存储周期”，使用一次总线事务进行数据传送，计数器减1。计数器为0时，发送**DMA结束中断请求**

“DMA结束”中断服务程序：

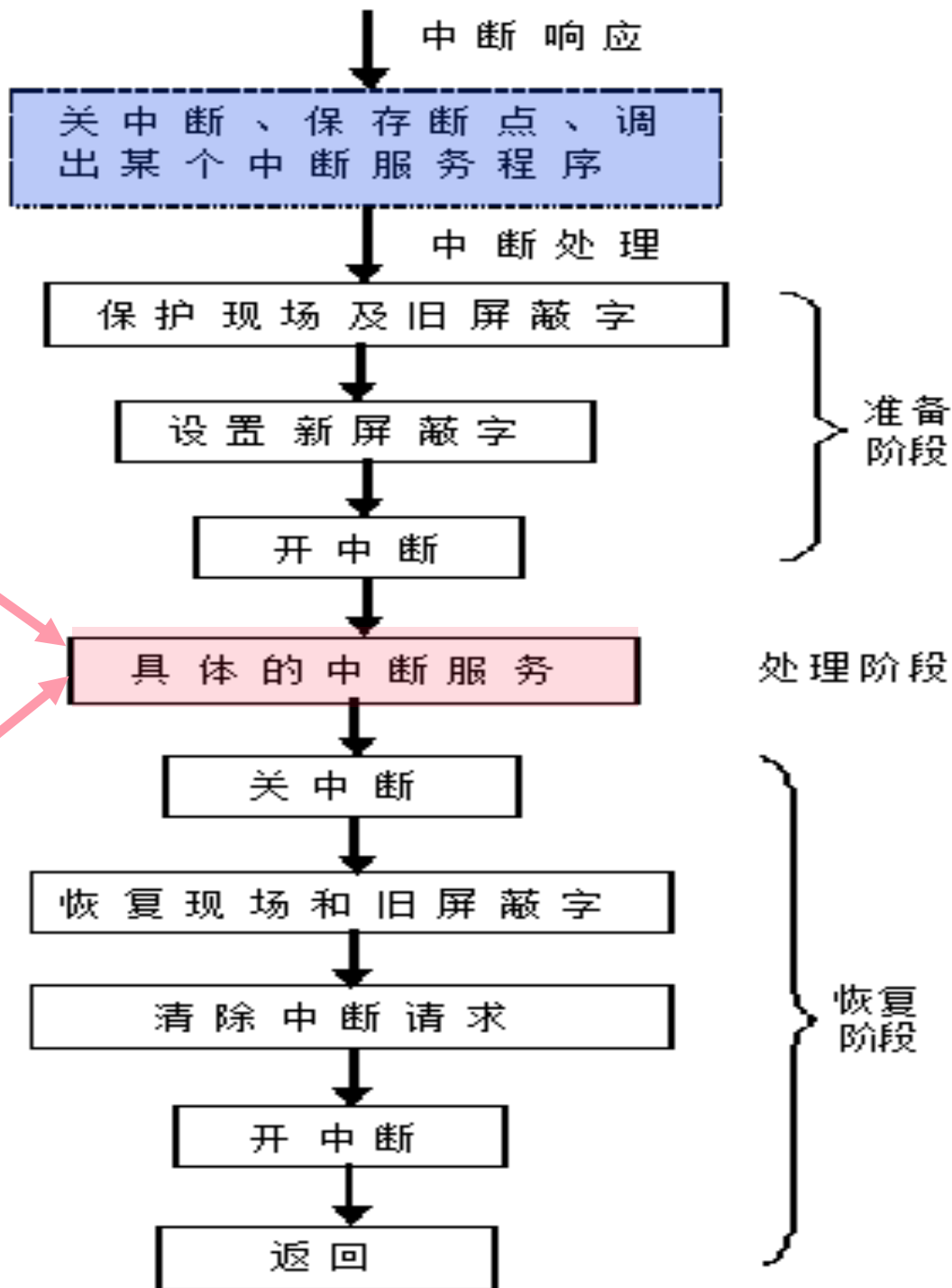
```
acknowledge_interrupt(); // 中断回答（清除中断请求）
unblock_user ( );        // 用户进程P解除阻塞，进入就绪队列
return_from_interrupt(); // 中断返回
```

CPU仅在DMA控制器初始化和处理“DMA结束中断”时介入，在DMA传送过程中不参与，因而CPU用于I/O的开销非常小。

中断服务程序

- 中断控制和DMA控制两种方式下都需进行中断处理
- 中断控制方式：**中断服务程序主要进行**从数缓冲器取数或写数据到数缓冲器**，然后启动外设工作
- DMA控制方式：**中断服务程序进行**数据校验**等后处理工作

在内核I/O软件中用到的I/O指令、“开中断”和“关中断”等指令都是特权指令，只能在操作系统内核程序中使用

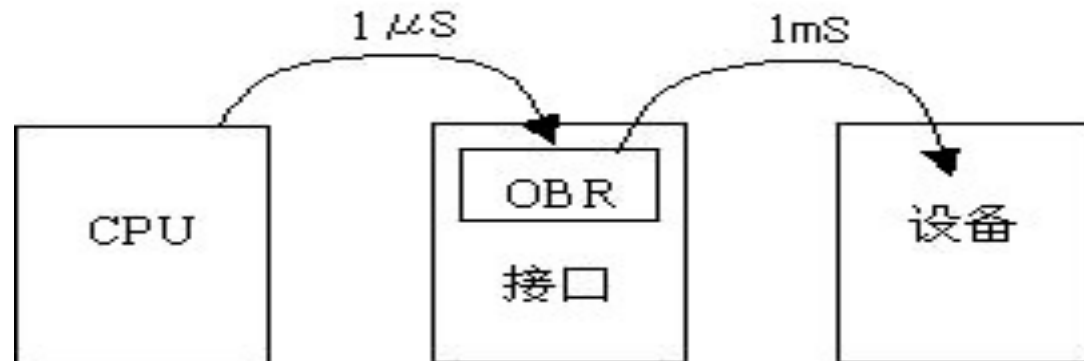


轮询方式和中断方式的比较

- 举例：假定某机控制一台设备输出一批数据。数据由主机输出到接口的数据缓冲器OBR，需要 $1\mu\text{s}$ 。再由OBR输出到设备，需要 1ms 。设一条指令的执行时间为 $1\mu\text{s}$ (包括隐指令)。试计算采用程序传送方式和中断传送方式的数据传输速度和对主机的占用率。

问题：CPU如何把数据送到OBR，I/O接口如何把OBR中的数据送到设备？

CPU执行I/O指令来将数据送OBR；而I/O接口则是自动把数据送到设备。



对主机占用率：

在进行I/O操作过程中，处理器有多少时间花费在输入/出操作上。

数据传送速度（吞吐量、I/O带宽）：

单位时间内传送的数据量。

假定每个数据的传送都要重新启动！即是字符型设备

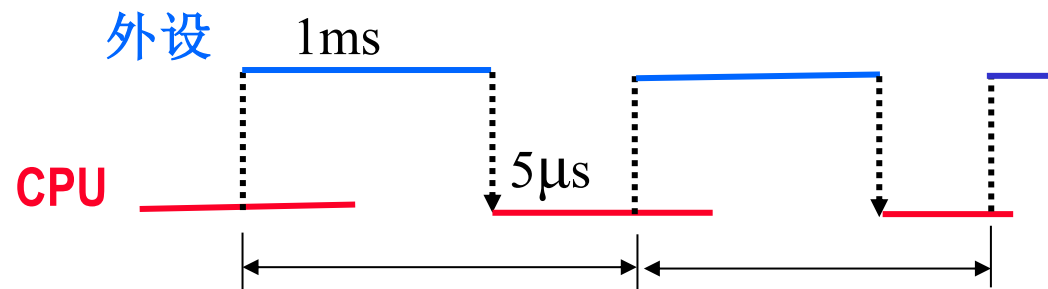
轮询方式和中断方式的比较

(1) 程序直接控制传送方式

若查询程序有10条，第5条为启动设备的指令，则：

数据传输率为： $1/(1000+5) \mu s$ ，约为每秒995个数据。

主机占用率=100%



轮询方式

(2) 中断传送方式

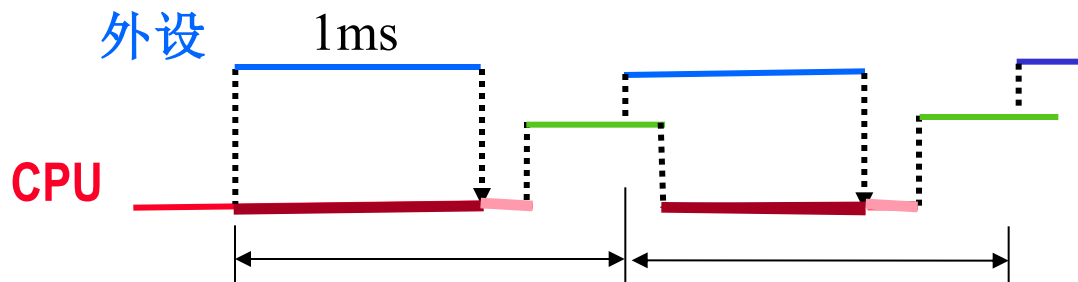
若中断服务程序有30条，在第20条启动设备，则：

数据传输率为：

$1/(1000+1+20)\mu s$ ，约为每秒979个数据。

主机占用率为：

$(1+30)/(1000+1+20)=3\%$



中断方式

为什么中断服务程序比查询程序长？

因为中断服务程序有额外开销，如：保存现场、保存旧屏蔽字、设置新屏蔽字、开中断、查询中断源等

Extra Slides

Fun with File Descriptors (1)

```
#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”?

Fun with File Descriptors (2)

```
#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”?

Fun with File Descriptors (3)

```
#include "csapp.h"
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);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1); /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}
```

ffiles3.c

- What would be the contents of the resulting file?

Accessing Directories

- **Only recommended operation on a directory: read its entries**
 - **dirent** structure contains information about a directory entry
 - **DIR** structure contains information about directory while stepping through its entries

```
#include <sys/types.h>
#include <dirent.h>

{
    DIR *directory;
    struct dirent *de;
    ...
    if (!(directory = opendir(dir_name)))
        error("Failed to open directory");
    ...
    while (0 != (de = readdir(directory))) {
        printf("Found file: %s\n", de->d_name);
    }
    ...
    closedir(directory);
}
```

Example of Accessing File Metadata

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

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

statcheck.c

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

/* Determine file type */

/* Check read access */

For Further Information

■ The Unix bible:

- W. Richard Stevens & Stephen A. Rago, ***Advanced Programming in the Unix Environment***, 2nd Edition, Addison Wesley, 2005
 - Updated from Stevens's 1993 classic text

■ The Linux bible:

- Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010
 - Encyclopedic and authoritative

I/O software in user space

- I/O software is not only in the OS, libraries are routinely linked with user programs, helper programs run outside the kernel.
 - `count = write(fd, buffer, nbytes)`
- Library routines typically transfer their parameters to the system calls, or do more work:
 - `printf("%d : %s, %ld\n", count, person, yearly_income);`
 - `cout << count << " : " << person << ", " << y_i << endl;`
 - `cin >> y_i;`
- Spooling systems: a daemon reads jobs from a spooling-directory and sends it to the printer (protected device, only spooler has access)

I/O software in user space (cont)

- Spooling may be used in networks, mailing systems,..., spoolers run outside the O.S.
- Overview:

user processes	Produce I/O call, format, spool
Device independent software	Naming, protection, blocking, buffering, allocating
Device drivers	Control device registers, status
Interrupt handlers	Wake up driver after I/O task
hardware	Perform I/O

本章小结

- 用户程序通常通过调用编程语言提供的库函数或操作系统提供的API函数来实现I/O操作
- I/O库函数最终都会调用系统调用的封装函数，通过封装函数中的陷阱指令使用户进程从用户态转到内核态执行
- 在内核态中执行的内核空间I/O软件主要包含三个层次：
 - 与设备无关的操作系统软件
 - 设备驱动程序
 - 中断服务程序
- 具体I/O操作通过设备驱动程序和中断服务程序控制I/O硬件来实现
- 设备驱动程序的实现主要取决于具体的I/O控制方式：
 - 程序查询方式、中断方式、DMA方式

练习

- 10.1、10.2、10.3