I/O操作的实现

用户空间I/O软件

I/O硬件与软件的接口

内核空间I/O软件

I/O和文件操作

- 主要教学目标
  - 通过揭示高级语言程序中的I/O及文件操作请求的底层实现机制，深刻理解OS在输入/输出系统中的重要作用；深刻理解计算机中硬件和软件如何协调工作以完成计算机功能。
- 主要教学内容
  - I/O子系统的组成和层次结构
  - 用户空间I/O软件
  - I/O硬件与软件的接口
  - 内核空间I/O软件

I/O操作的实现

- 分以下三个部分介绍
  - 第一讲：用户空间I/O软件
    - I/O子系统概述
    - 一切设备皆文件
    - 用户空间的I/O函数
  - 第二讲：I/O硬件和软件的接口
    - I/O设备和设备控制器
    - I/O端口及其编址方式
    - I/O控制方式
  - 第三讲：I/O硬件和软件的接口
    - 与设备无关的I/O软件
    - 设备驱动程序
    - 中断服务程序

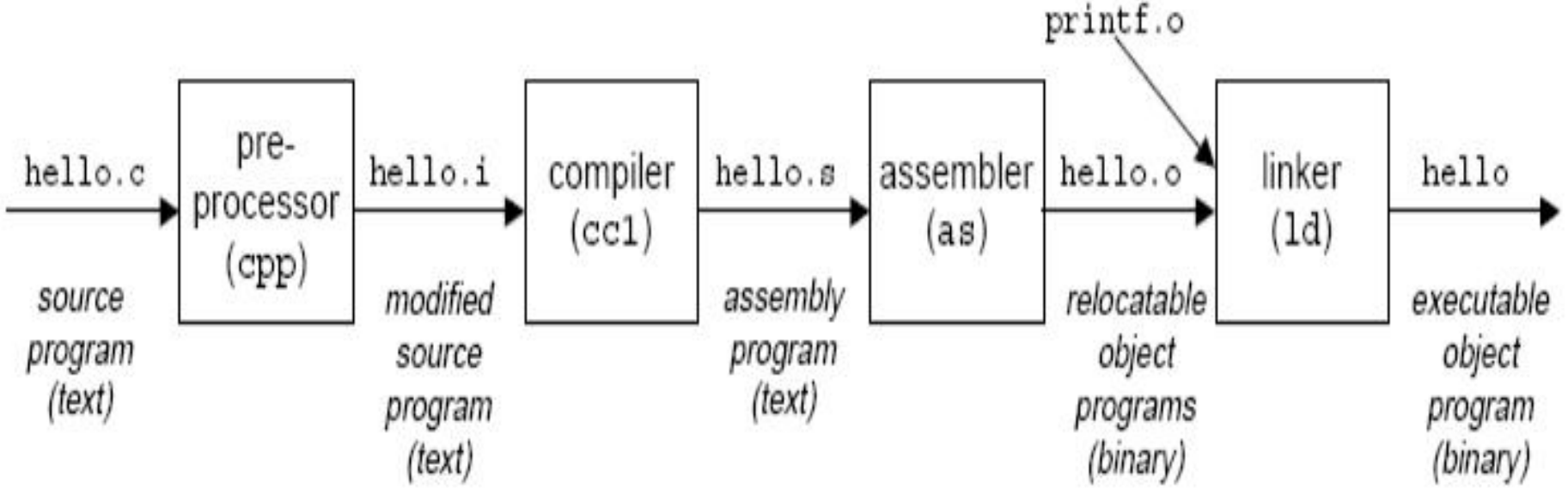
复习：一个典型程序的转换处理过程

经典的“ hello.c ”源程序

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

hello.c的ASCII文本表示

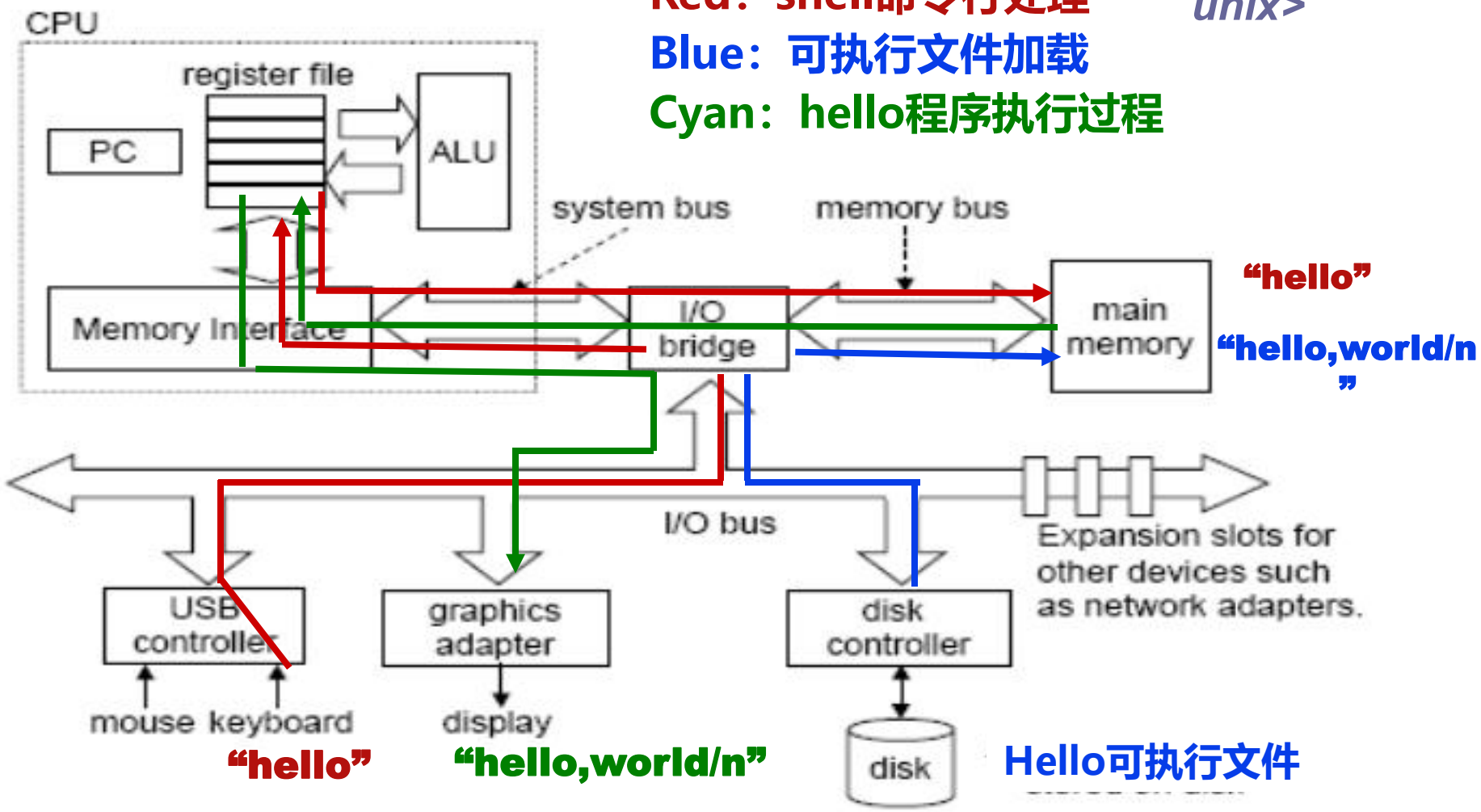
```
# i n c l u d e <sp> < s t d i o .
35 105 110 99 108 117 100 101 32 60 115 116 100 105 111 46
h > \n \n i n t <sp> m a i n ( ) \n {
104 62 10 10 105 110 116 32 109 97 105 110 40 41 10 123
\n <sp> <sp> <sp> <sp> p r i n t f ( " h e l
10 32 32 32 32 112 114 105 110 116 102 40 34 104 101 108
l o , <sp> w o r l d \ n " ) ; \n }
108 111 44 32 119 111 114 108 100 92 110 34 41 59 10 125
```



复习：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



问题：hello程序何时被装？谁来装入？被谁启动？每次是否被装到相同的地方？Hello程序是否能直接访问硬件资源？

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，最后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插入的一个中间软件层。
- 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子系统概述

- 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）

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：

- 用户层I/O软件（I/O函数调用系统调用）
- 与设备无关的操作系统I/O软件
- 设备驱动程序
- 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。
  - 例如，若用户程序在IA-32架构上执行，则I/O函数被转换为IA-32的指令序列。
  - 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，而系统调用封装函数在用户态执行，具体**服务例程**在内核态执行。

系统调用及其参数传递

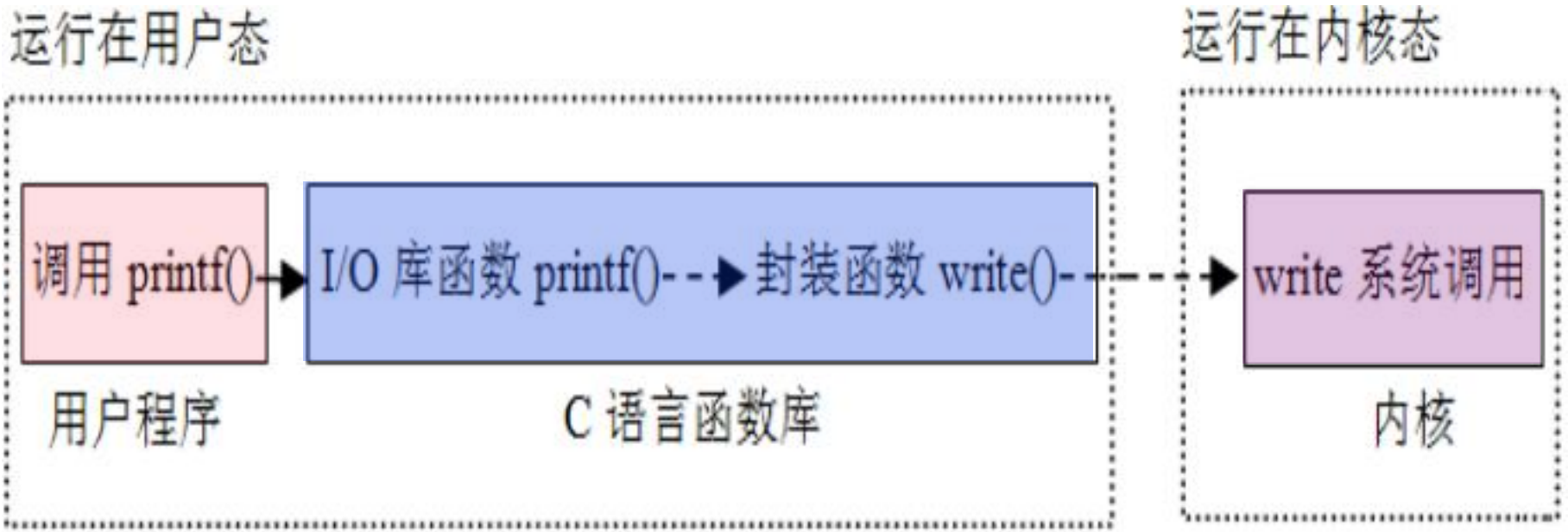
- 当用户态进程调用一个系统调用时，CPU切换到内核态，并开始执行一个被称为**系统调用处理程序**的内核函数
- 例如，IA-32中，可以通过两种方式调用Linux的系统调用
  - 执行软中断指令int 80
  - 执行指令sysenter（老的x86不支持该指令）
- 内核实现了许多系统调用，因此，用一个**系统调用号（存放在EAX中）**来标识不同系统调用
- 除了调用号以外，系统调用还需要其他参数，不同系统调用所需参数的个数和含义不同，**输入参数通过通用寄存器传递**，若参数个数超出寄存器个数，则将需传递参数块所在内存区首址放在寄存器中传递**（除调用号以外，最多6个参数）**
  - 传递参数的寄存器顺序：EAX（系统调用号）、EBX、ECX、EDX、ESI、EDI和EBP
- 返回参数为整数值。正数或0表示成功，负数表示出错码

用户程序、C函数和内核

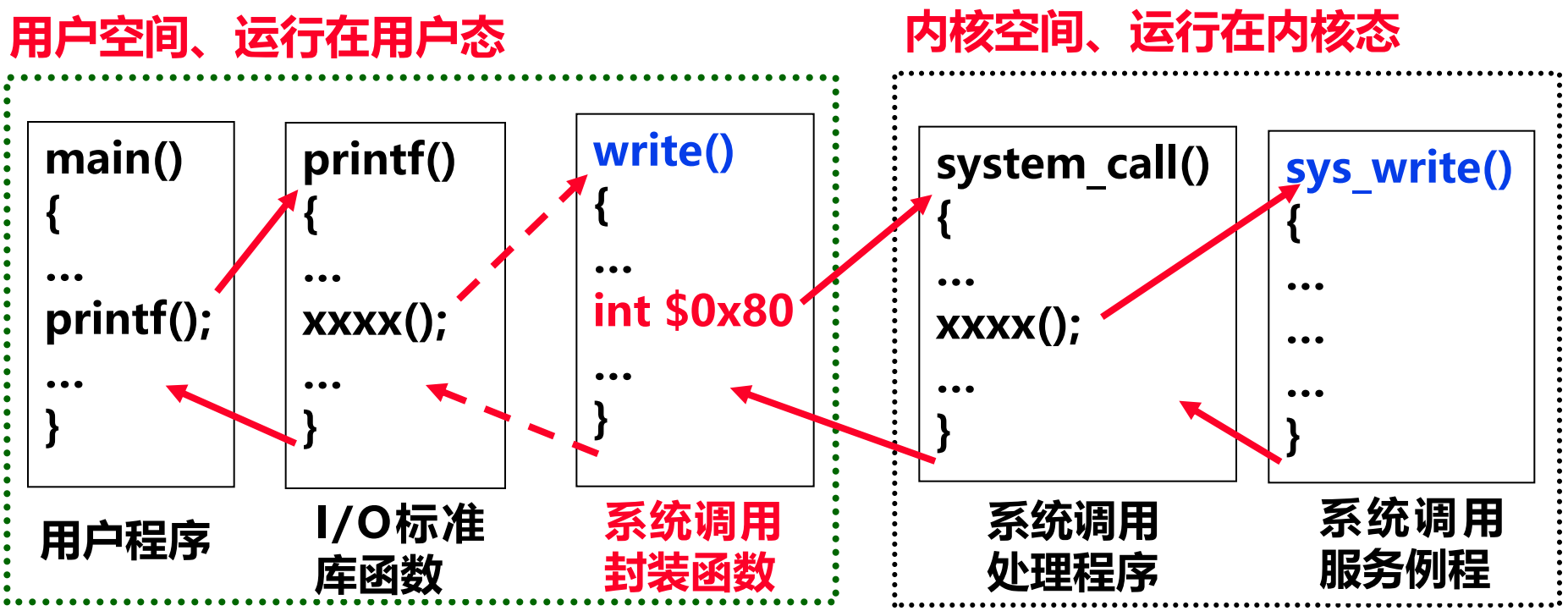
- 用户程序总是通过某种I/O函数或I/O操作符请求I/O操作。

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：



Linux系统中printf()函数的执行过程



- 某函数调用了printf()，执行到调用printf()语句时，便会转到C语言I/O标准库函数printf()去执行；
- printf()通过一系列函数调用，最终会调用函数write()；
- 调用write()时，便会通过一系列步骤在内核空间中找到write对应的系统调用服务例程sys_write来执行。

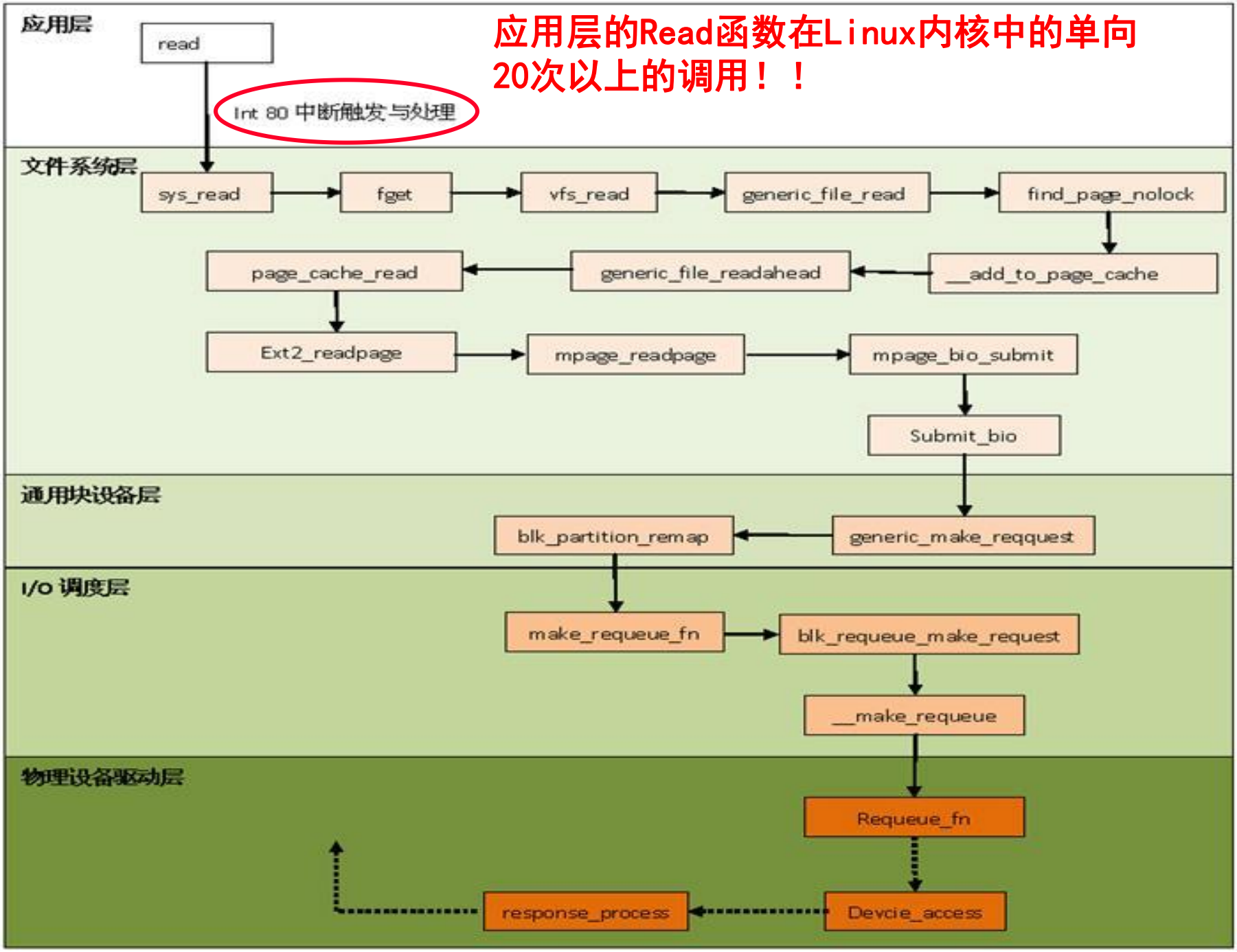
在system_call中如何知道要转到sys_write执行呢？ 根据系统调用号！

Linux系统下的write()封装函数

用法：ssize_t write(int fd, const void * buf, size_t n);

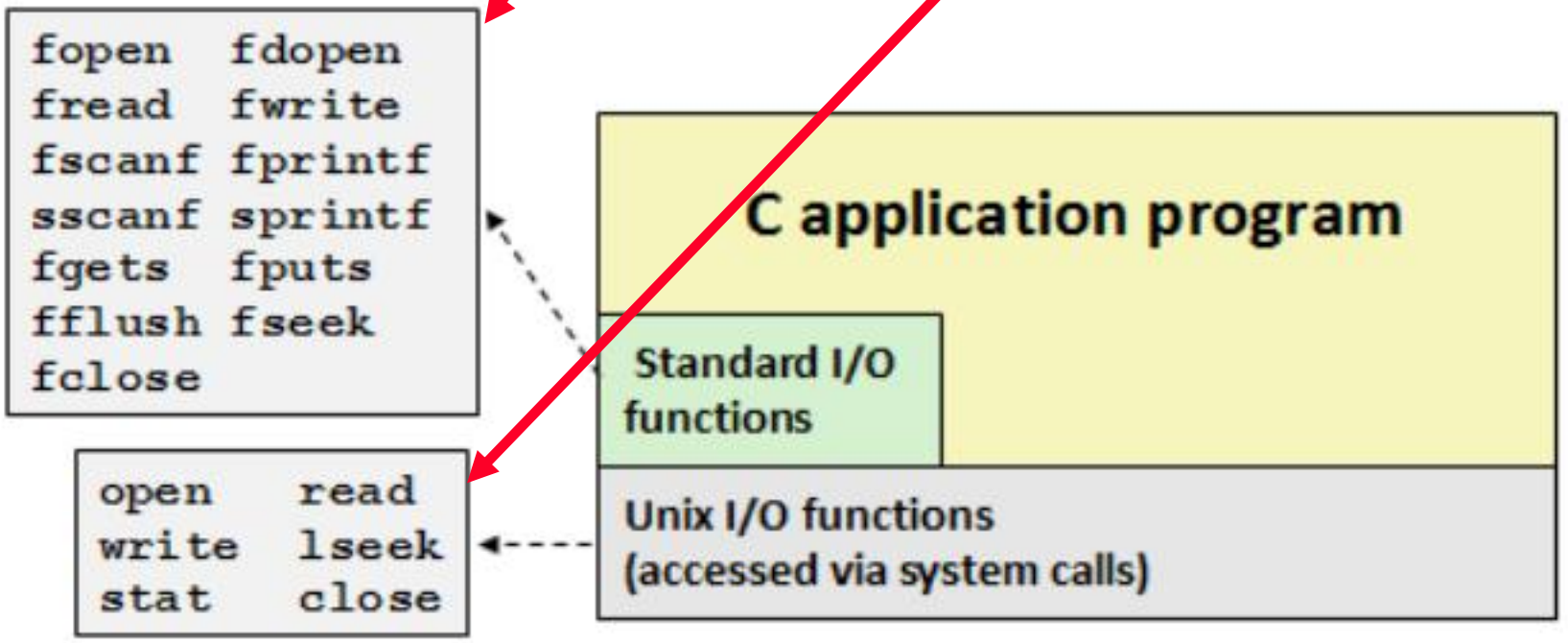
size_t 和 ssize_t 分别是 unsigned int 和 int，因为返回值可能是-1。

```
1 write:
2     pushl  %ebx              //将EBX入栈（EBX为被调用者保存寄存器）
3     movl  $4, %eax           //将系统调用号 4 送EAX
4     movl  8(%esp), %ebx      //将文件描述符 fd 送EBX
5     movl  12(%esp), %ecx     //将所写字符串首址 buf 送ECX
6     movl  16(%esp), %edx     //将所写字符个数 n 送EDX
7     int  $0x80               //进入系统调用处理程序system_call执行
8     cmpl  $-125, %eax        //检查返回值
9     jbe    .L1               //若无错误，则跳转至.L1（按无符号数比）
10    negl  %eax               //将返回值取负送EAX
11    movl   %eax, error       //将EAX的值送error
12    movl   $-1, %eax         //将write函数返回值置-1
13 .L1:
14     popl   %ebx             内核执行write的结果在EAX中返回，正确时
15     ret                     为所写字符数（最高位为0），出错时为错误
                               码的负数（最高位为1）
```


应用层
read
Int 80 中断触发与处理
应用层的Read函数在Linux内核中的单向
20次以上的调用！！
文件系统层
sys_read
fget
vfs_read
generic_file_read
find_page_nolock
page_cache_read
generic_file_readahead
__add_to_page_cache
Ext2_readpage
mpage_readpage
mpage_bio_submit
Submit_bio
通用块设备层
blk_partition_remap
generic_make_reqquest
I/O 调度层
make_requeue_fn
blk_requeue_make_request
__make_requeue
物理设备驱动层
Requeue_fn
response_process
Devcie_access

用户空间中的I/O函数

- 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。



用户空间中的I/O函数

C标准库	UNIX/Linux	Windows	功能描述
getc，scanf，gets	read	ReadConsole	从标准输入读取信息
fread	read	ReadFile	从文件读入信息
putc，printf，puts	write	WriteConsole	在标准输出上写信息
fwrite	write	WriteFile	在文件上写入信息
fopen	open，creat	CreateFile	打开/创建一个文件
fclose	close	CloseHandle	关闭文件（CloseHandle 不限于文件）
fseek	lseek	SetFilePointer	设置文件读写位置
rewind	lseek(0)	SetFilePointer(0)	将文件指针设置成指向文件开头
remove	unlink	DeleteFile	删除文件
feof	无对应	无对应	停留到文件末尾
perror	strerror	FormatMessage	输出错误信息
无对应	stat，fstat，lstat	GetFileTime	获取文件的时间属性
无对应	stat，fstat，lstat	GetFileSize	获取文件的长度属性
无对应	fcnt	LockFile / UnlockFile	文件的加锁、解锁

文件的基本概念

哪里遇过“文件”？ int fprintf(FILE *fp, char *format, [argument])

- 所有I/O操作通过读写文件实现，所有外设，包括网络、终端设备，都被看成文件。 printf在哪显示信息？ stdout文件！ 即终端显示器TTY
- 所有物理设备抽象成逻辑上统一的“文件”使得用户程序访问物理设备与访问真正的磁盘文件完全一致。例如，fprintf/fwrite(主要是磁盘文件) 和printf (stdout) 都通过统一的write函数陷入内核，差别则由内核处理！
- UNIX系统中，文件就是一个字节序列。 Stream！字节流
- 通常，将键盘和显示器构成的设备称为终端（terminal），对应标准输入、和标准（错误）输出文件；像磁盘、光盘等外存上的文件则是普通文件 。
- 根据文件的可读性，文件被分成ASCII文件和二进制文件两类。
- ASCII文件也称文本文件，可由多个正文行组成，每行以换行符‘\n’ 结束，每个字符占一个字节。标准输入和标准(错误)输出文件是ASCII文件。
- 普通文件可能是文本文件或二进制文件。

问题：.c、.cpp、.o、.txt、.exe文件各是什么类型文件?

文件的创建和打开

读写文件前，用户程序须告知将对文件进行何种操作：读、写、添加还是可读可写，通过打开或创建一个文件来实现。

- ✓ 已存在的文件：可直接打开
- ✓ 不存在的文件：则先创建

1. 创建文件：int creat(char *name, mode_t perms);

- ◆ 创建新文件时，应指定文件名和访问权限，系统返回一个非负整数，它被称为文件描述符fd (file descriptor)。
- ◆ 文件描述符用于标识被创建的文件，在以后对文件的读写等操作时用文件描述符代表文件。

2. 打开文件：int open(char *name, int flags, mode_t perms);

- ◆ 标准输入(fd=0)、标准输出(fd=1)和标准错误(fd=2)三种文件自动打开，其他文件须用creat或open函数显式创建或打开后才能读写
- ◆ 第三个参数perms用于指定所创建文件的访问权限，通常在open函数中该参数总是0，除非以创建方式打开，此时，参数flags中应带有O_CREAT标志。

文件的读/写

3. 读文件：size_t read(int fd, void *buf, size_t n);

◆将fd中当前位置k开始的n个字节读到buf中，读后当前位置为k+n。若文件长度为m，当k+n>m时，则读取字节数为m-k<n，读后当前位置为文件尾。返回实际字节数，当m=k（EOF）时，返回值为0。

4. 写文件：ssize_t write(int fd, const void *buf, size_t n);

◆将buf中n字节写到fd中，从当前位置k处开始写。返回实际写入字节数m，写后当前位置为k+m。对于普通文件，实际字节数等于n。

- 对于read和write系统调用，可以一次读/写任意字节。显然，按一个物理块大小读/写较好，可减少系统调用次数。
- 有些情况下，真正读/写字节数比设定所需字节数少，这并不是一种错误。在读/写磁盘文件时，除非遇到EOF，否则不会出现这种情况。但当读/写的是终端设备或网络套接字文件、UNIX管道、Web服务器等都可能出现这种情况。

文件的定位和关闭

5. 设置读写位置：long lseek(int fd, long offset, int origin);

◆offset指出相对字节数；origin指出基准，分别是开头（0）、当前位置（1）和末尾（2）。例如，lseek(fd, 0L, 2)表示定位到文件末尾。返回位置值，若发生错误，则返回-1。

6. 元数据统计：int stat(const *name, struct stat *buf);

int fstat(int fd, struct stat *buf);

◆文件的所有属性信息，包括文件描述符、文件名、文件大小、创建时间、当前读写位置等都由内核维护，这些信息称为文件的元数据（metadata）。

◆用户程序可通过stat()或fstat()函数查看文件元数据。

◆stat第一个参数是文件名，而fstat指出的是文件描述符，除第一个参数类型不同外，其他全部一样。

7. 关闭文件：close(int fd);

C标准I/O库函数

```
#define NULL        0
#define EOF         (-1)
#define BUFSIZ      1024
#define OPEN_MAX  20   /* 最多打开文件数 */
typedef struct  _iobuf     {
        int     cnt;      /* 未读写字节数  */
        char *ptr;        /*下一可读写位置 */
        char *base;       /* 起始位置 */
        int     flag;     /* 存取模式 */
        int     fd;       /*  文件描述符 */
} FILE;
extern  FILE  _iob[OPEN_MAX];

#define stdin     (&_iob[0])
#define stdout    (&_iob[1])
#define stderr    (&_iob[2])
enum  _flags {
        _READ= 01,    /* file open for reading */
        _WRITE= 02,   /* file open for writing */
        _UNBUF= 04,  /* file is unbuffered */
        _EOF= 010,    /* EOF has occurred on this file */
        _ERR= 020     /* error occurred on this file */
};
```

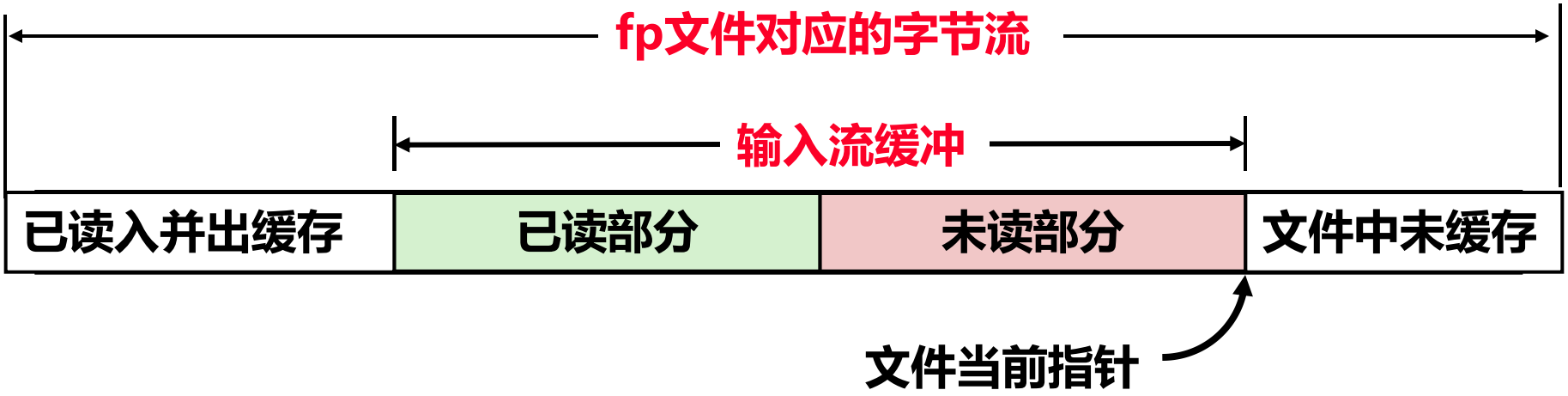
- C标准I/O库函数基于系统调用实现
- C标准I/O库函数将打开文件抽象为一个类型为FILE的“流”，它stdio.h中定义。

```
FILE _iob[OPEN_MAX] = {
  { 0, ( char * ) 0, ( char * ) 0, _READ, 0 },
  { 0, ( char * ) 0, ( char * ) 0, _WRITE, 1 },
  { 0, ( char * ) 0, ( char * ) 0, _WRITE | _UNBUF, 2 },
};
```

stdout和stderr都用于输出，但是，stderr为非缓存stdout为带缓存

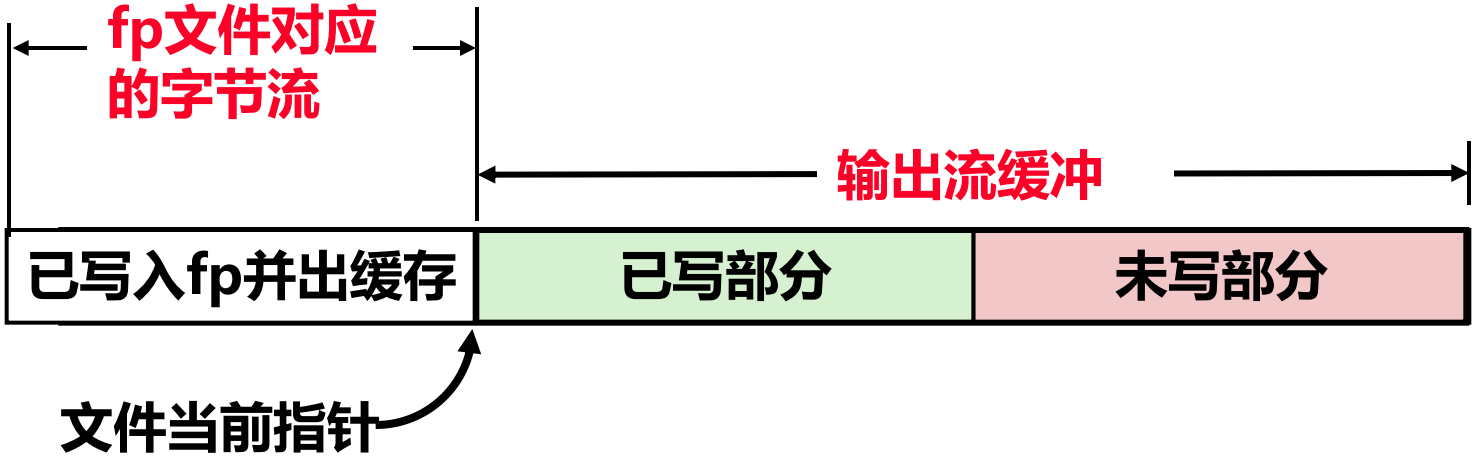
带缓冲I/O的实现

- 从文件fp中读数据时，FILE中定义的缓冲区为输入流缓冲
- 首先要从文件fp中读入1024（缓冲大小BUFSIZ=1024）个字节数据到缓存，然后，再按需从缓存中读取1个（如getc）或n个（如fread）字节并返回



带缓冲I/O的实现

- 向文件fp中写数据时，FILE中定义的缓冲区为**输出流缓冲**
- 先按需**不断地**向缓存写1个（如getc）或n个（如fread）字节，遇到换行符\n或缓存被写满1024（缓冲大小BUFSIZ=1024）个字节，则将缓存内容一次写入文件fp中



stdout和stderr的差别

猜一下以下程序的输出是什么?

```
#include<stdio.h>
int main()
{
    fprintf(stdout, “hello ");
    fprintf(stderr, “world!");
    return 0;
}
```

输出结果为：world!hello

stdout和stderr都用于标准输出，
但是，

stderr为 _WRITE | _UNBUF

stdout为 _WRITE

```
#include<stdio.h>
int main()
{
    fprintf(stdout, “hello ");
    fprintf(stderr, “world!\n");
    return 0;
}
```

输出结果为：world!
hello

```
#include<stdio.h>
int main()
{
    fprintf(stdout, “hello \n");
    fprintf(stderr, “world!");
    return 0;
}
```

输出结果为：hello
world!

有缓冲：遇到换行符\n或缓冲满（BUFSIZE=1024）才写文件!

stdout 和 stderr 的差别

网上的一个例子：

```
#include <stdio.h>
void main()
{
    fprintf(stdout, "from stdout\n") ;
    fprintf(stderr, "from stderr\n");
}
```

编译成 hello.exe后，执行命令行：

```
hello.exe > out.txt
```

会发现屏幕输出如下：

```
from stderr
```

同时out.txt中的内容为

```
from stdout
```

默认情况下，二者都指向标准输出，即显示器；

只有stdout设备文件的内容才可重定位到普通文件中！

stdio.h中更多的定义

- 在stdio.h中，还定义了feof()、ferror()、fileno()、getc()、putc()、getchar()、putchar()等标准I/O函数。
- 系统级I/O函数对文件的标识是文件描述符，C标准I/O库函数中对文件的标识是指向FILE结构的指针，FILE中定义了1024字节的流缓冲区。
- 使用流缓冲区可使文件内容缓存在用户缓冲区中，而不是每次都直接读/写文件，从而减少执行系统调用次数。

```
int _fillbuf( FILE *);  /*第一次调用getc()，需用_fillbuf()填充缓冲区*/
int _flushbuf( int, FILE *); /*遇换行或写缓冲区满，调用其将缓冲内容写文件*/

#define  feof(p)     (((p) ->flag & _EOF) != 0)
#define  ferror(p)   (((p) ->flag & _ERR) != 0)
#define  fileno(p)   ((p) ->fd)
#define  getc(p)     (--(p)->cnt>=0 ? (unsigned char)*(p)->ptr++ : _fillbuf(p))
#define  putc(x,p)  (--(p)->cnt>=0 ? *(p)->ptr++ = (x) : _flushbuf((x),p))
#define  getchar()    getc(stdin)
#define  putchar(x)   putc((x), stdout)
```

输入缓冲内容未读完。cnt为未读字符数，初值为0，调用_fillbuf()后值<=1023。

输出缓冲未写满。cnt为可写字符数，初值为0，调用_flushbuf()后，值为1024-1=1023。

_fillbuf()函数的实现

```
#include “syscalls.h”
/* _fillbuf: allocate and fill input buffer */
int _fillbuf(FILE *fp)
{
    int bufsize;
    if ((fp ->flag & ( _READ | _EOF | _ERR)) != _READ)
        return EOF;
    bufsize = (fp ->flag & _UNBUF) ? 1 : BUFSIZ;
    if ((fp -> base == NULL)                    /* 刚开始，还没有申请缓冲 */
        if (( fp -> base = (char *) malloc(bufsize)) == NULL)
        return EOF;                    /* 缓冲没有申请到 */
    fp -> ptr = fp -> base;
    fp -> cnt = read (fp->fd, fp->ptr, bufsize);    /* cnt<=1024 */
    if (--fp->cnt < 0) {                                /* cnt<=1023 */
        if (fp->cnt == -1) fp->flag | = _EOF;
        else fp->flag | = _ERR;
        fp -> cnt =0;
        return EOF;
    }
    return (unsigned char ) *fp->ptr++;
}
```

stderr没有缓冲
即bufsize=1

cnt减1

调用系统调用封装函数进行读文件操作，一次将输入缓冲读满

返回缓冲区当前字节，并ptr加1

_flushbuf()函数的实现

```
int _flushbuf(int x, FILE *fp)
{
    unsigned nc;
    int bufsize;
    if (fp < _iob || fp > _iob + OPEN_MAX)
        return EOF;
    if ((fp->flag & (_WRITE | _ERR)) != _WRITE)
        return EOF;
    bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
    if (fp->base == NULL) {          /* 刚开始，还没有申请缓冲 */
        if ((fp->base = (char *)malloc(bufsize)) == NULL) {
            fp->flag |= _ERR;
            return EOF;
        }
    } else {                         /* 已存在缓冲，且遇到换行符或缓冲已满 */
        nc = fp->ptr - fp->base;
        if (write(fp->fd, fp->base, nc) != nc) {
            fp->flag |= _ERR;
            return EOF;
        }
    }
    fp->ptr = fp->base;
    *fp->ptr++ = x;
    fp->cnt = bufsize - 1;
    return x;
}
```

举例：文件复制功能的实现

```
/* 方式一: getc/putc版本 */
void filecopy(FILE *infp, FILE
  *outfp)
{
  int c;
  while ((c=getc(infp)) != EOF)
        putc(c, outfp);
}
/* 方式二: read/write版本 */
void filecopy(int *infp, int *outfp)
{
  char c;
  while (read(infp,&c,1) != 0)
        write(outfp,&c,1);
}
```

实现一个功能有多种方式，但开销和性能不同，需要权衡！

哪种方式更好?

方式一更好！Why?

因其系统调用次数少！

对于方式二，若文件长度为n，则需执行2n次系统调用；

对于方式一，若文件长度为n，则执行系统调用的次数约为n/512。

为何要尽量减少系统调用次数?

系统调用的开销有多大? 相当大!

还有其他的实现方法吗?

SKIP

使用fread()和fwrite()

使用fgetc()和fputc()

使用WindowsAPI函数CopyFile()

Linux系统下的write()封装函数

用法：ssize_t write(int fd, const void * buf, size_t n);

size_t 和 ssize_t 分别是 unsigned int 和 int，因为返回值可能是-1。

```
1 write:
2     pushl  %ebx              //将EBX入栈（EBX为被调用者保存寄存器）
3     movl  $4, %eax           //将系统调用号 4 送EAX
4     movl  8(%esp), %ebx      //将文件描述符 fd 送EBX
5     movl  12(%esp), %ecx     //将所写字符串首址 buf 送ECX
6     movl  16(%esp), %edx     //将所写字符个数 n 送EDX
7     int  $0x80               //进入系统调用处理程序system_call执行
8     cmpl  $-125, %eax        //检查返回值
9     jbe    .L1               //若无错误，则跳转至.L1（按无符号数比）
10    negl  %eax               //将返回值取负送EAX
11    movl   %eax, error       //将EAX的值送error
12    movl   $-1, %eax         //将write函数返回值置-1
13 .L1:
14     popl   %ebx             内核执行write的结果在EAX中返回，正确时
15     ret                     为所写字符数（最高位为0），出错时为错误
                               码的负数（最高位为1）
```

软中断指令int $0x80的执行过程

它是陷阱类（编程异常）事件，因此它与异常响应过程一样。

1) 将IDTi（i=128）中段选择符（0x60）所指GDT中的内核代码段描述符取出， 其DPL=0，此时CPL=3（因为int $0x80指令在用户进程中执行），因而CPL>DPL且IDTi 的 DPL=CPL，故未发生13号异常。

2) 读 TR 寄存器，以访问TSS，从TSS中将内核栈的段寄存器内容和栈指针装入SS和ESP；

3) 依次将执行完指令int $0x80时的SS、ESP、EFLAGS、CS、EIP的内容（即断点和程序状态）保存到内核栈中，即当前SS：ESP所指之处；

4) 将IDTi（i=128）中段选择符（0x60）装入CS，偏移地址装入EIP。

这里，CS:EIP即是系统调用处理程序system_call（所有系统调用的入口程序）第一条指令的逻辑地址。

BACK

执行int $0x80需一连串的一致性和安全性检查，因而速度较慢。从Pentium II开始，Intel引入了指令sysenter和sysexit，分别用于从用户态到内核态、从用户态到内核态的快速切换。

I/O操作的实现

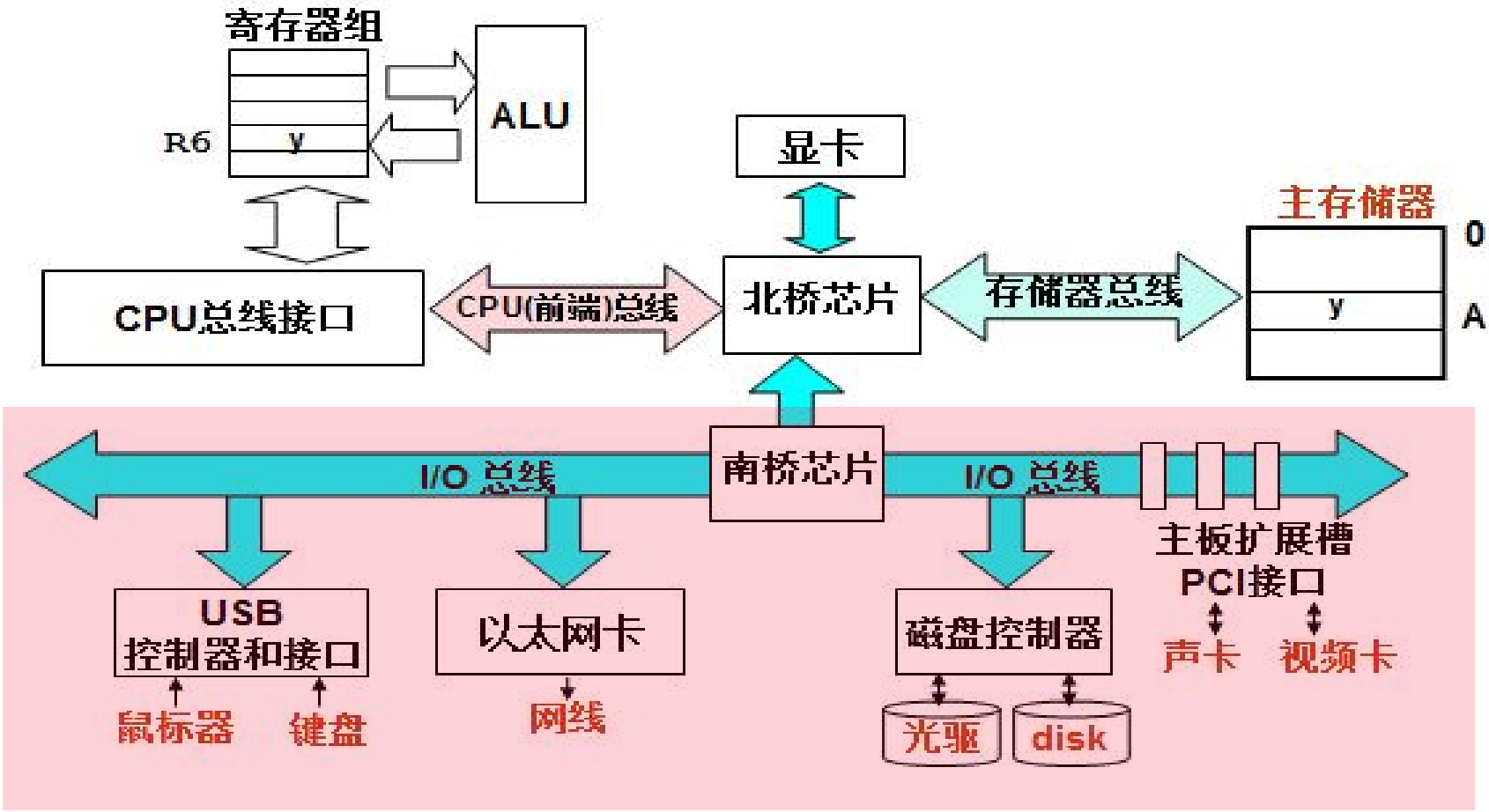
- 分以下三个部分介绍
  - 第一讲：用户空间I/O软件
    - I/O子系统概述
    - 一切设备皆文件
    - 用户空间的I/O函数
  - 第二讲：I/O硬件和软件的接口
    - I/O设备和设备控制器
    - I/O端口及其编址方式
    - I/O控制方式
  - 第三讲：内核空间I/O软件
    - 与设备无关的I/O软件
    - 设备驱动程序
    - 中断服务程序

I/O硬件的组成

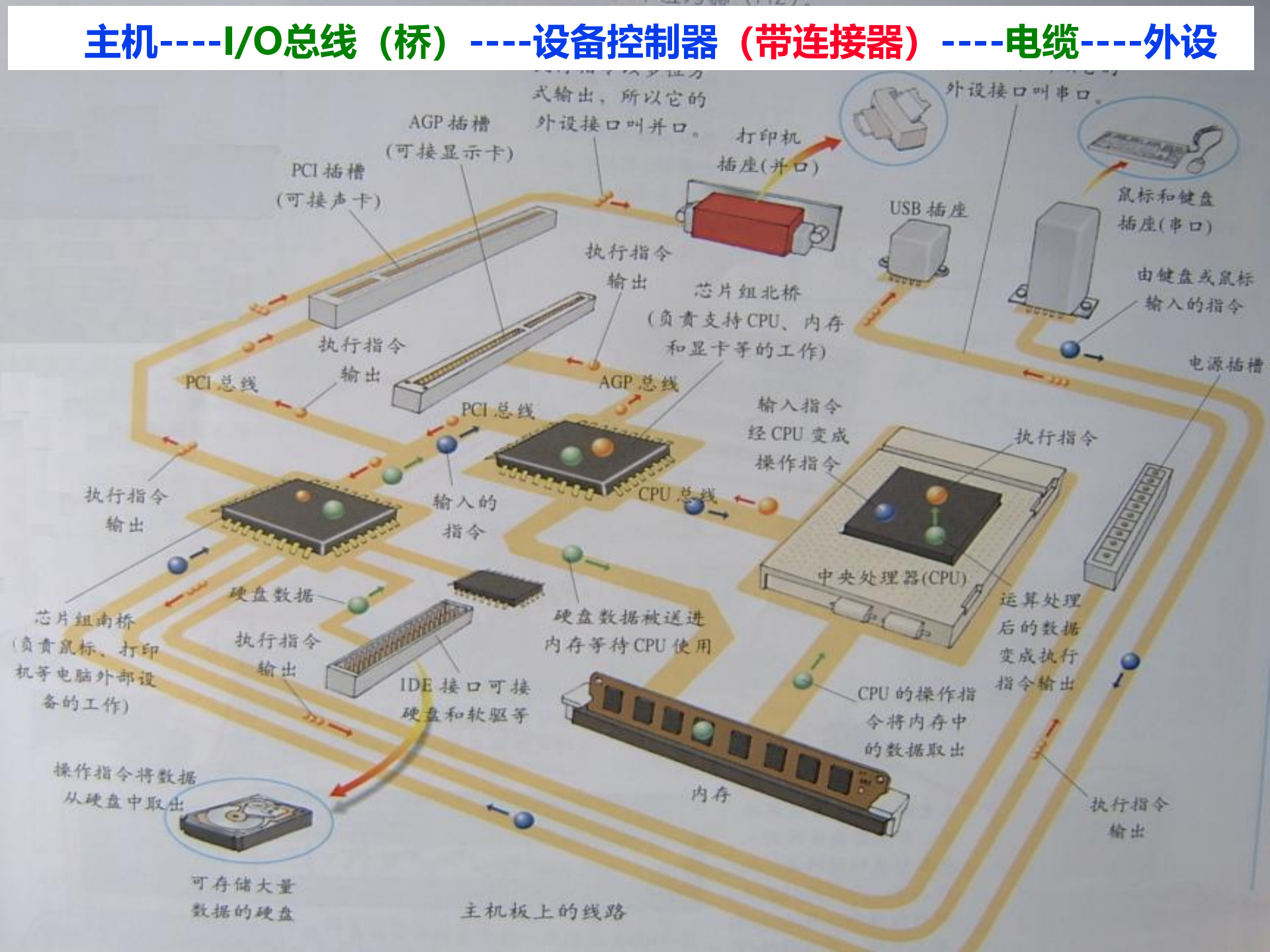
I/O硬件建立了外设与主机之间的“通路”：

主机----I/O总线（桥）----设备控制器----电缆----外设

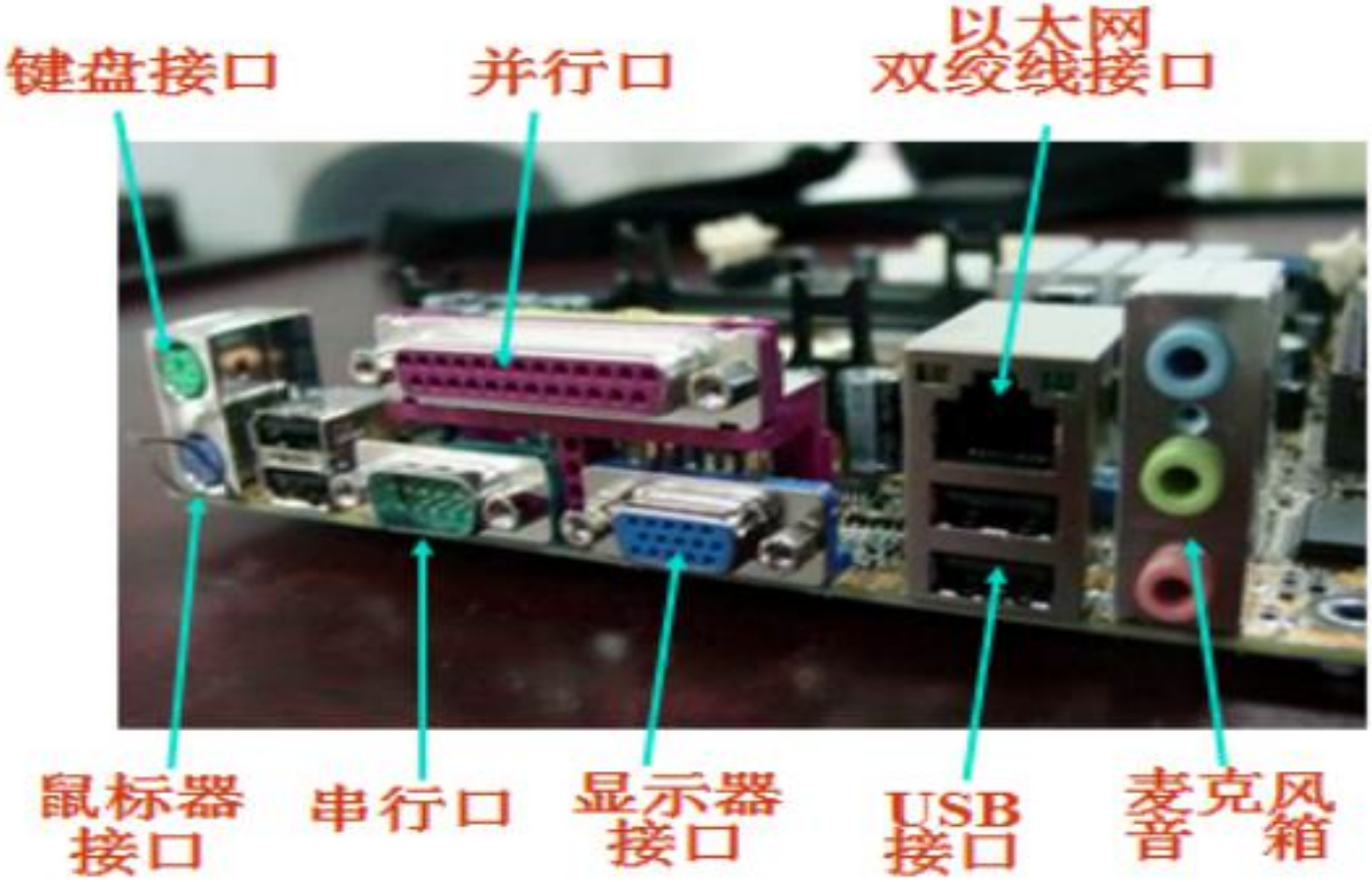
如何把用户I/O请求转换为对设备的控制命令并完成设备I/O任务，需要I/O软件与I/O硬件之间的协调工作



主机----I/O总线（桥）----设备控制器（带连接器）----电缆----外设



连接外部设备的连接器



主机----I/O总线（桥）----设备控制器（带连接器）----电缆----外设

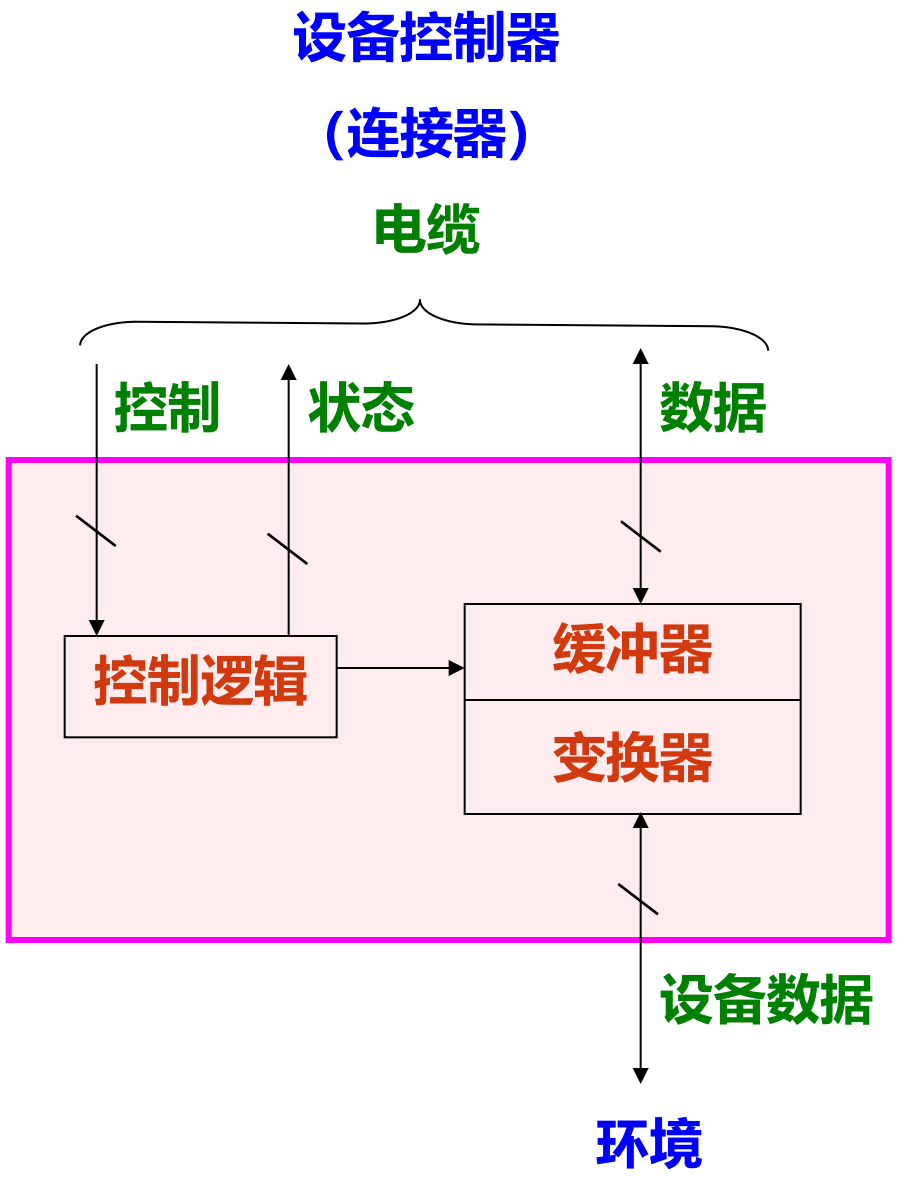
外部设备的通用模型

- 通过**电缆**与设备控制器（I/O接口）进行数据、状态和控制信息的传送
- **控制逻辑**根据控制信息控制设备的操作，并检测设备状态
- **缓冲器**用于保存交换的数据信息
- **变换器**用于在电信号形式（内部数据）和其他形式的设备数据之间进行转换

所有设备都可抽象成该通用模型！

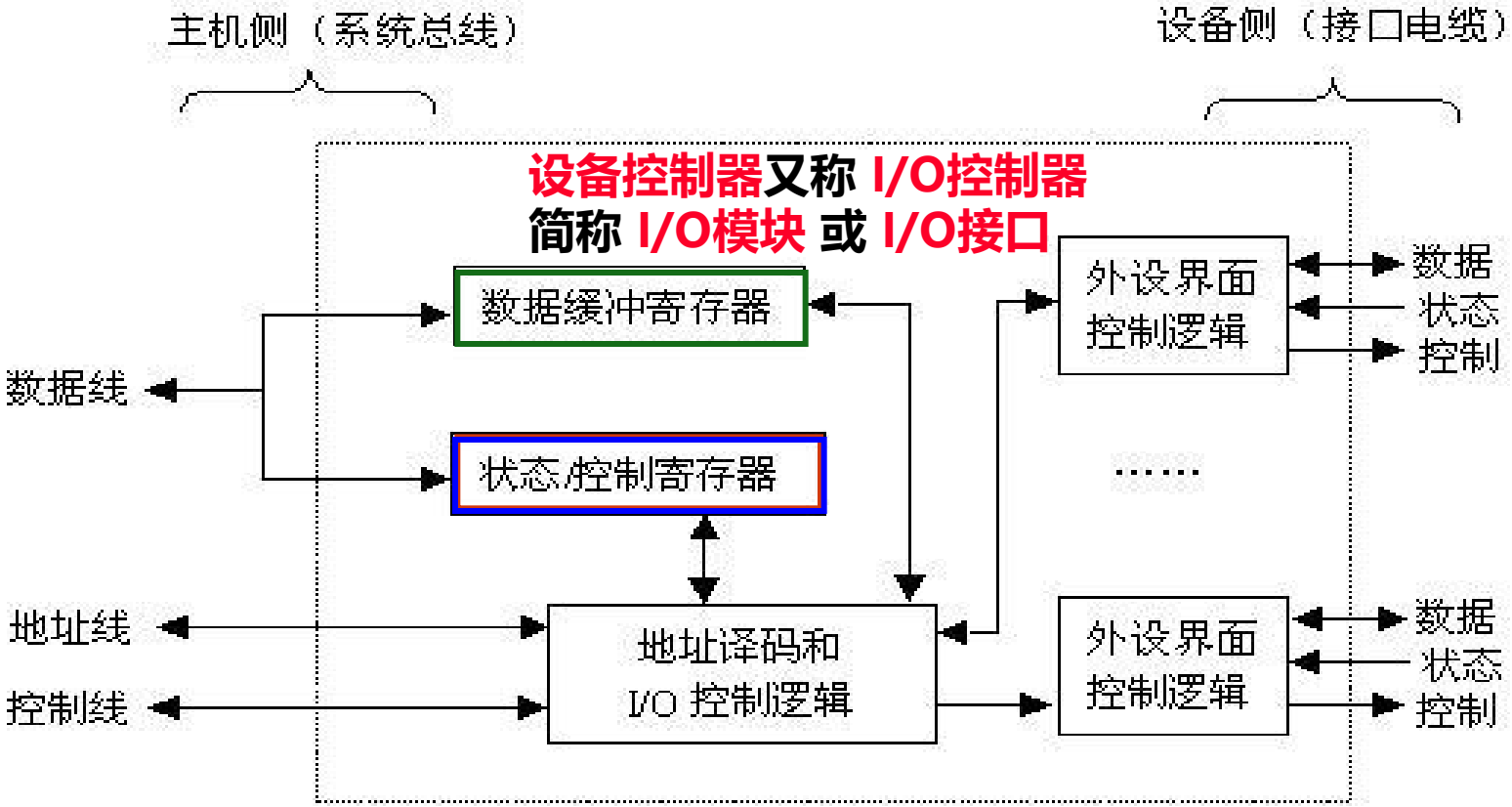
设备所用电缆中有三种信号线：

控制信号、状态信号、数据信号



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

（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

例如：教室和办公室可以连号（统一编址），也可单独编号（独立编址）

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
- IA-32中的I/O指令：in、ins、out和outs
  - in和ins用于将I/O端口的内容取到CPU内的通用寄存器中；
  - out和outs用于将通用寄存器内容输出到I/O端口。

如 IN AL, DX：DX中存放I/O端口地址，将I/O端口中的内容取到AL中

三种基本I/O方式

- 程序直接控制方式（最简单的I/O方式）
  - 无条件传送：对简单外设定时（同步）进行数据传送
  - 条件传送：CPU主动查询，也称程序查询或轮询（Polling）方式
- I/O Interrupt (中断I/O方式): 几乎所有系统都支持中断I/O方式
  - 若一个I/O设备需要CPU干预，它就通过中断请求通知CPU
  - CPU中止当前程序的执行，调出OS（中断处理程序）来执行
  - 处理结束后，再返回到被中止的程序继续执行
- Direct Memory Access (DMA方式): 磁盘等高速外设所用的方式
  - 磁盘等高速外设成批地直接和主存进行数据交换
  - 需要专门的DMA控制器控制总线，完成数据传送
  - 数据传送过程无需CPU参与

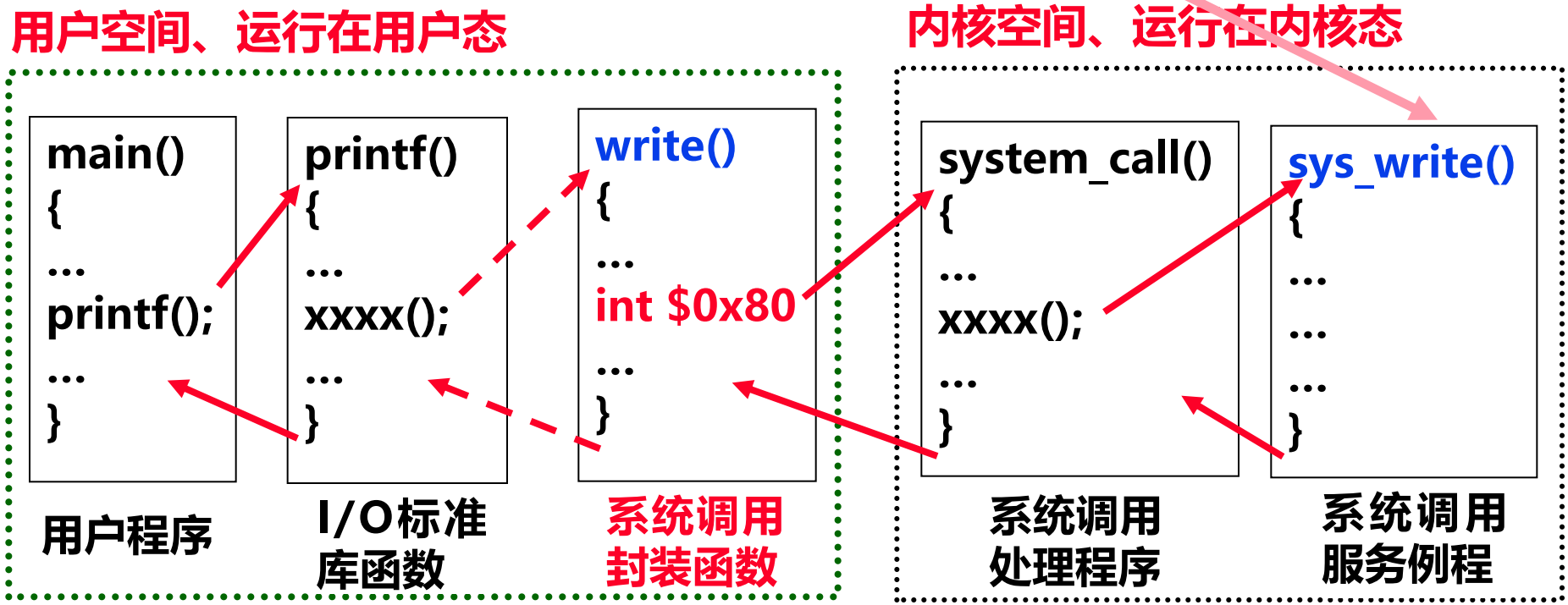
以hello程序为例说明

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



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

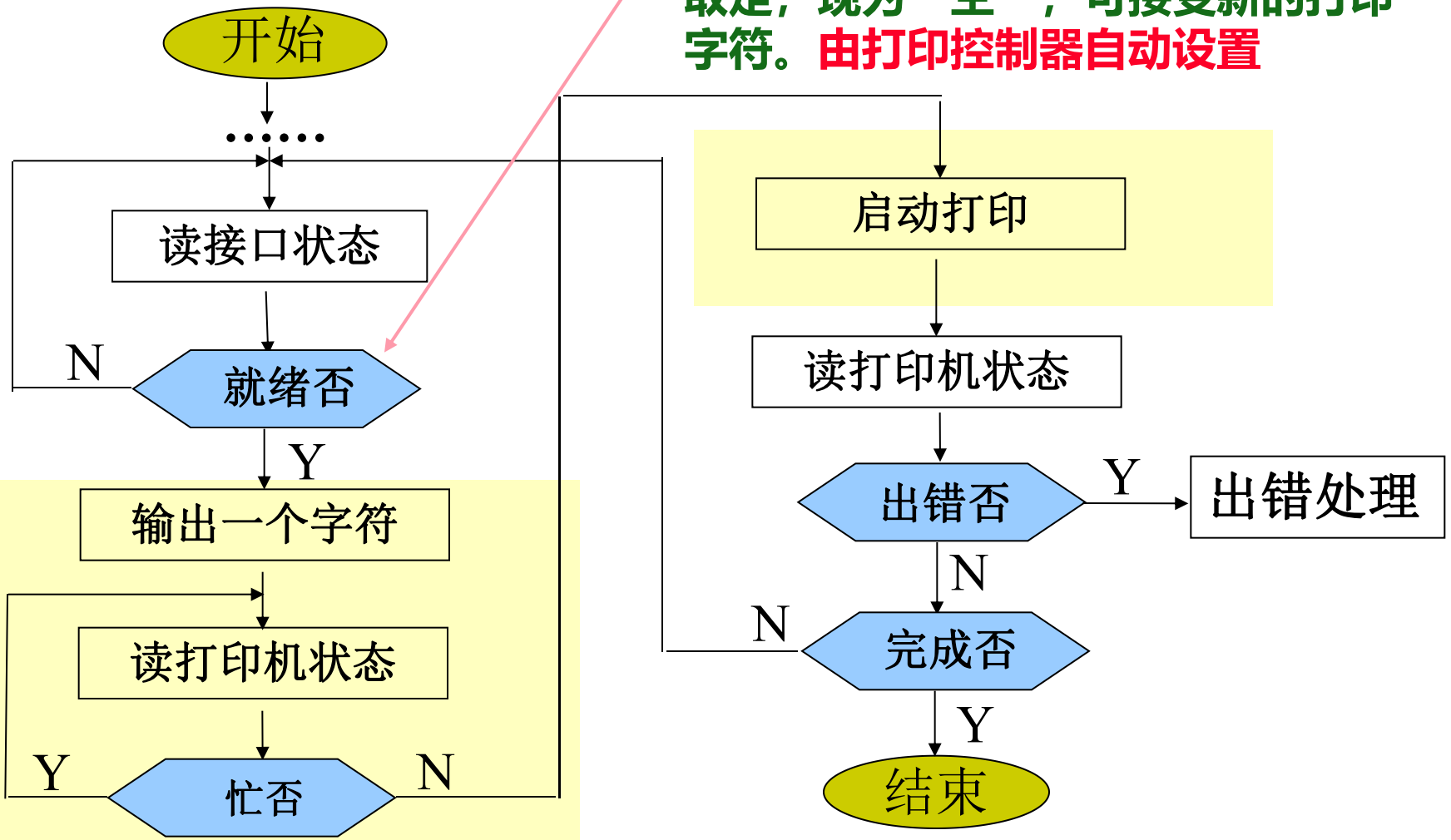
等待：读状态、判断是否为1；不是，则继续读状态、判断、……

程序查询（Polling）方式

- 举例：控制打印输出

这里“就绪”的含义是什么?

打印机控制器的数据缓冲中内容已被取走，现为“空”，可接受新的打印字符。由打印控制器自动设置



打印输出标准子程序

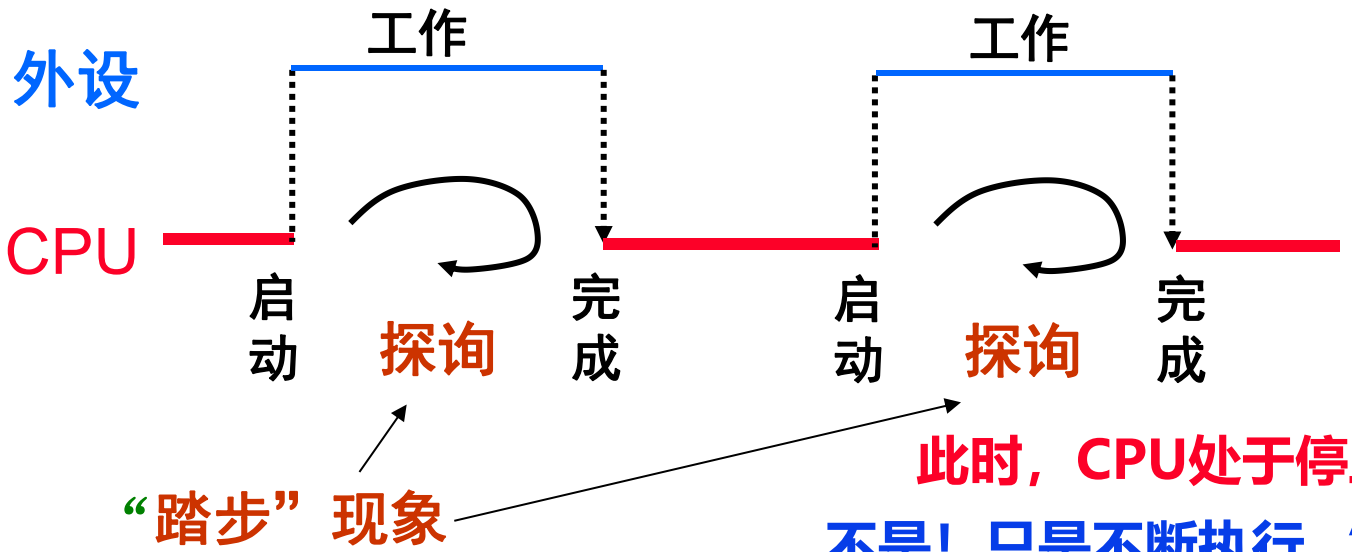
功能：打印AL寄存器中的字符（上一张PPT中黄色部分）。

```
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



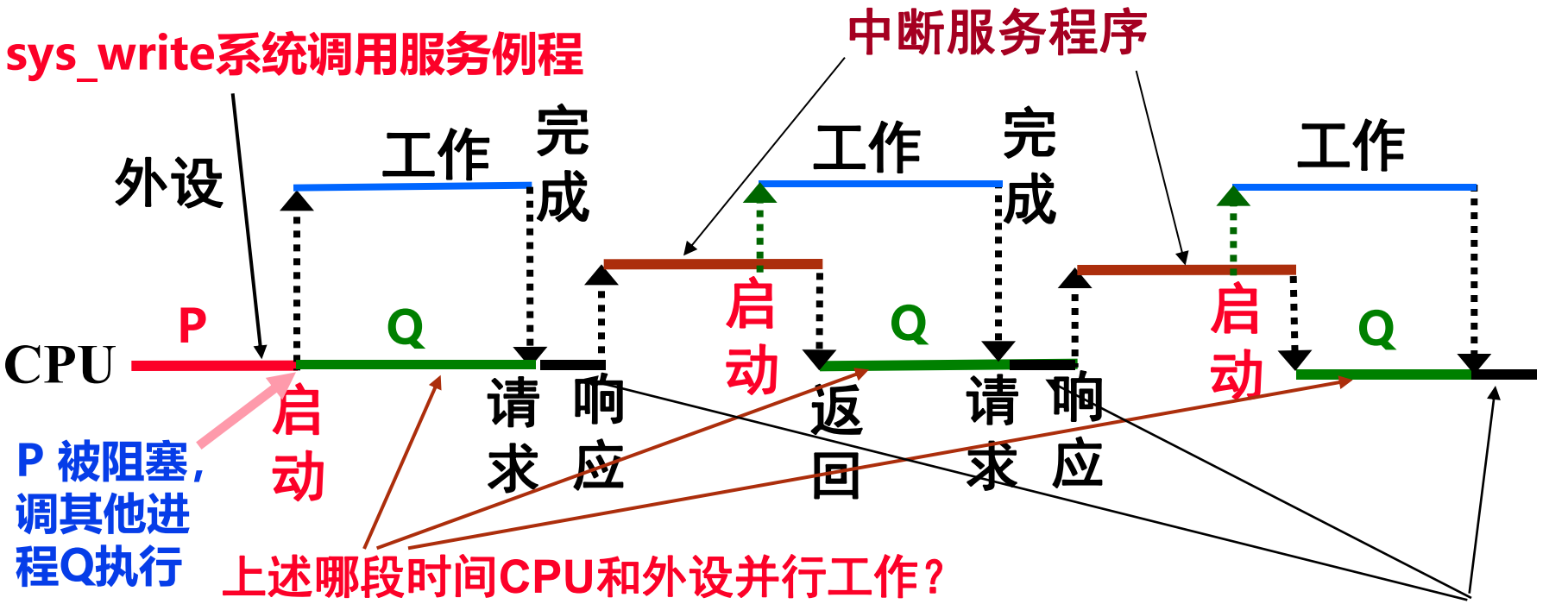
“探询”期间，可一直不断查询（独占查询），
也可定时查询（需保证数据不丢失！）。

- 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=kernbuf[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:

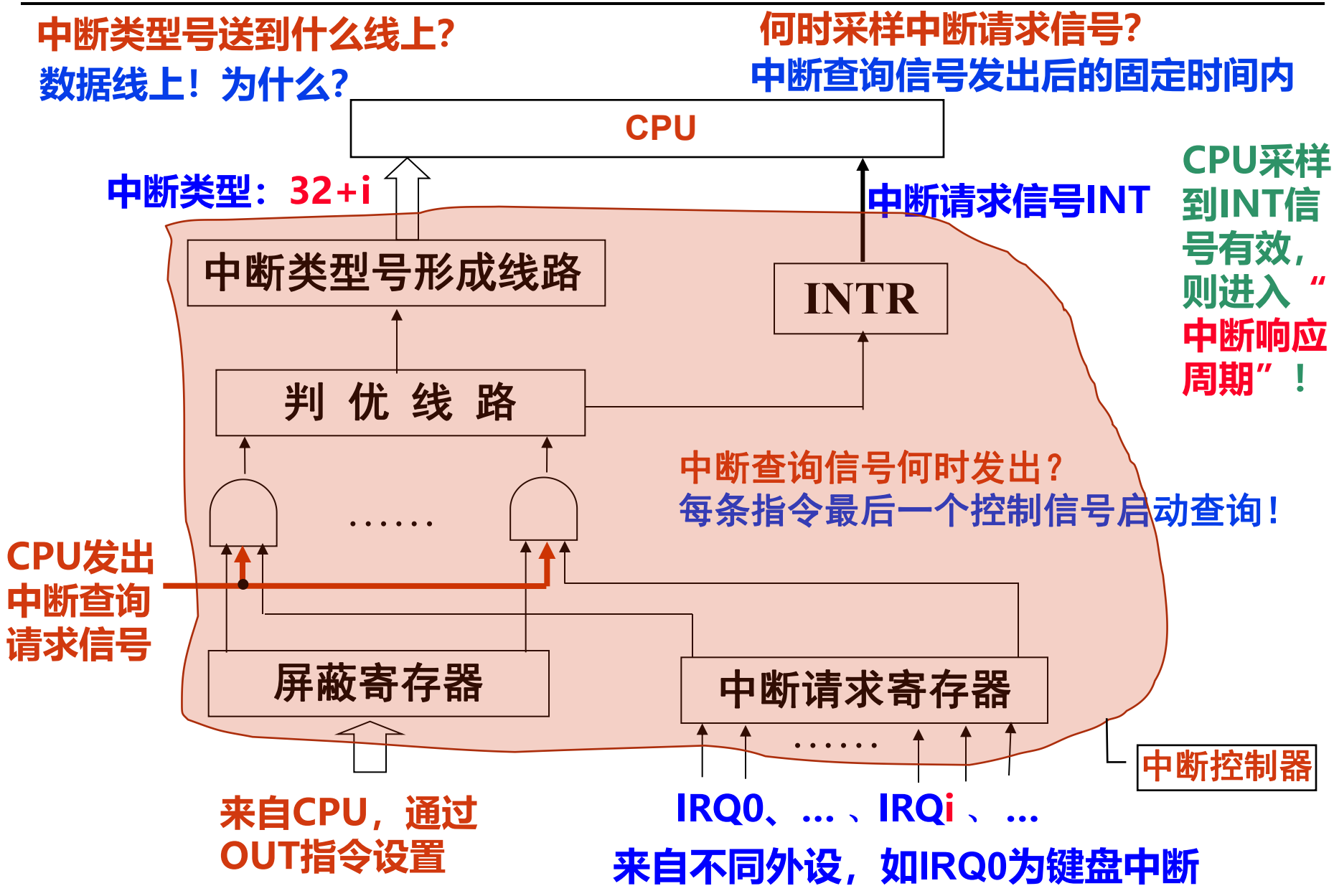
```
if (n==0) {                    // 若字符串打印完，则
unblock_user ( );              // 用户进程P解除阻塞，P进就绪队列
} else {
   *printer_data_port=kernelbuf[i]; // 向数据端口输出一个字符
   *printer_control_port=START;     // 发送“启动打印”命令
    n = n-1;                   // 未打印字符数减1
    i = i+1;                   // 下一个打印字符指针加1
}
acknowledge_interrupt();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

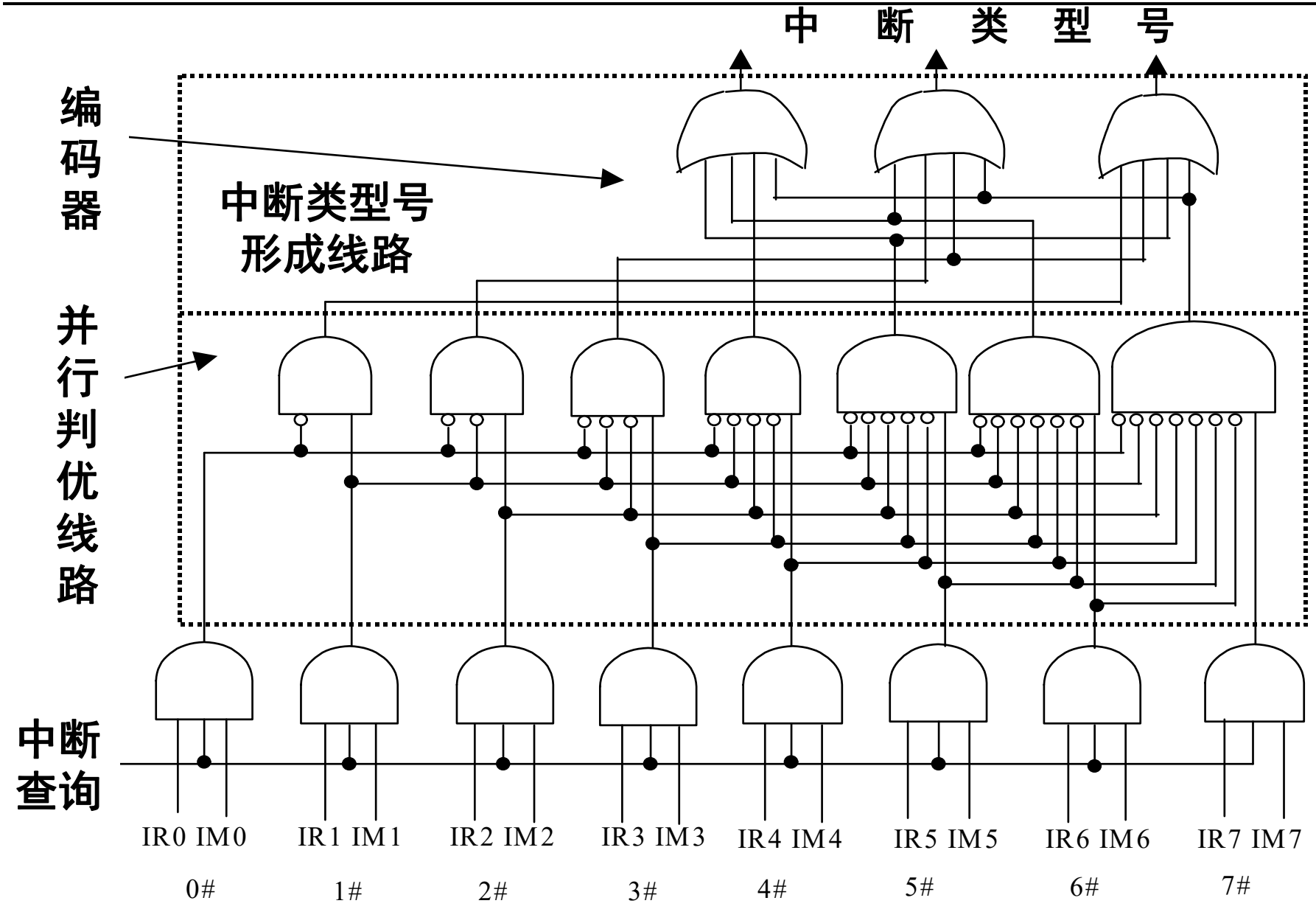
系统调用!

中断服务程序是如何调出来的?

中断控制器的基本结构

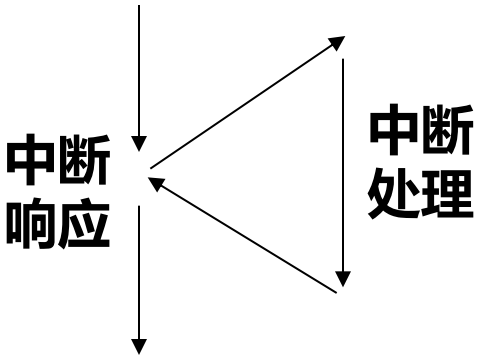

中断类型号送到什么线上?
数据线上！为什么?
何时采样中断请求信号?
中断查询信号发出后的固定时间内
CPU
中断类型：32+i
中断请求信号INT
CPU采样到INT信号有效，则进入“中断响应周期”！
中断类型号形成线路
INTR
判 优 线 路
中断查询信号何时发出?
每条指令最后一个控制信号启动查询!
CPU发出中断查询请求信号
屏蔽寄存器
中断请求寄存器
中断控制器
来自CPU，通过OUT指令设置
IRQ0、…、IRQi 、…
来自不同外设，如IRQ0为键盘中断

中断优先权编码器


中 断 类 型 号
编码器
中断类型号
形成线路
并行判优线路
中断查询
IR0 IM0
IR1 IM1
IR2 IM2
IR3 IM3
IR4 IM4
IR5 IM5
IR6 IM6
IR7 IM7
0#
1#
2#
3#
4#
5#
6#
7#

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。

中断处理过程

中断响应的结果就是调出相应的中断服务程序

中断处理是指执行相应中断服务程序的过程

- 不同的中断源其对应的中断服务程序不同。
- 典型的多重中断处理（中断服务程序）分为三个阶段：
  - 先行段（准备阶段）
    - 保护现场及旧屏蔽字
    - 查明原因（软件识别中断时）
    - 设置新屏蔽字

    处在“关中断”状态，不允许被打断
    - 开中断

    处在“开中断”状态，可被新的处理优先级更高的中断打断
  - 本体段（具体的中断处理阶段）
  - 结束段（恢复阶段）
    - 关中断
    - 恢复现场及旧屏蔽字
    - 清“中断请求”

    处在“禁止中断”状态，不允许被打断
    - 开中断
    - 中断返回

单重中断不允许在中断处理时被新的中断打断，因而直到中断返回前才会开中断。单重中断系统无需设置中断屏蔽字。

多重中断的概念

- 多重中断和中断处理优先权的动态分配
  - **多重中断的概念：**

    在一个中断处理（即执行中断服务程序）过程中，若又有新的中断请求发生，而新中断优先级高于正在执行的中断，则应立即中止正在执行的中断服务程序，转去处理新的中断。这种情况为多重中断，也称中断嵌套。

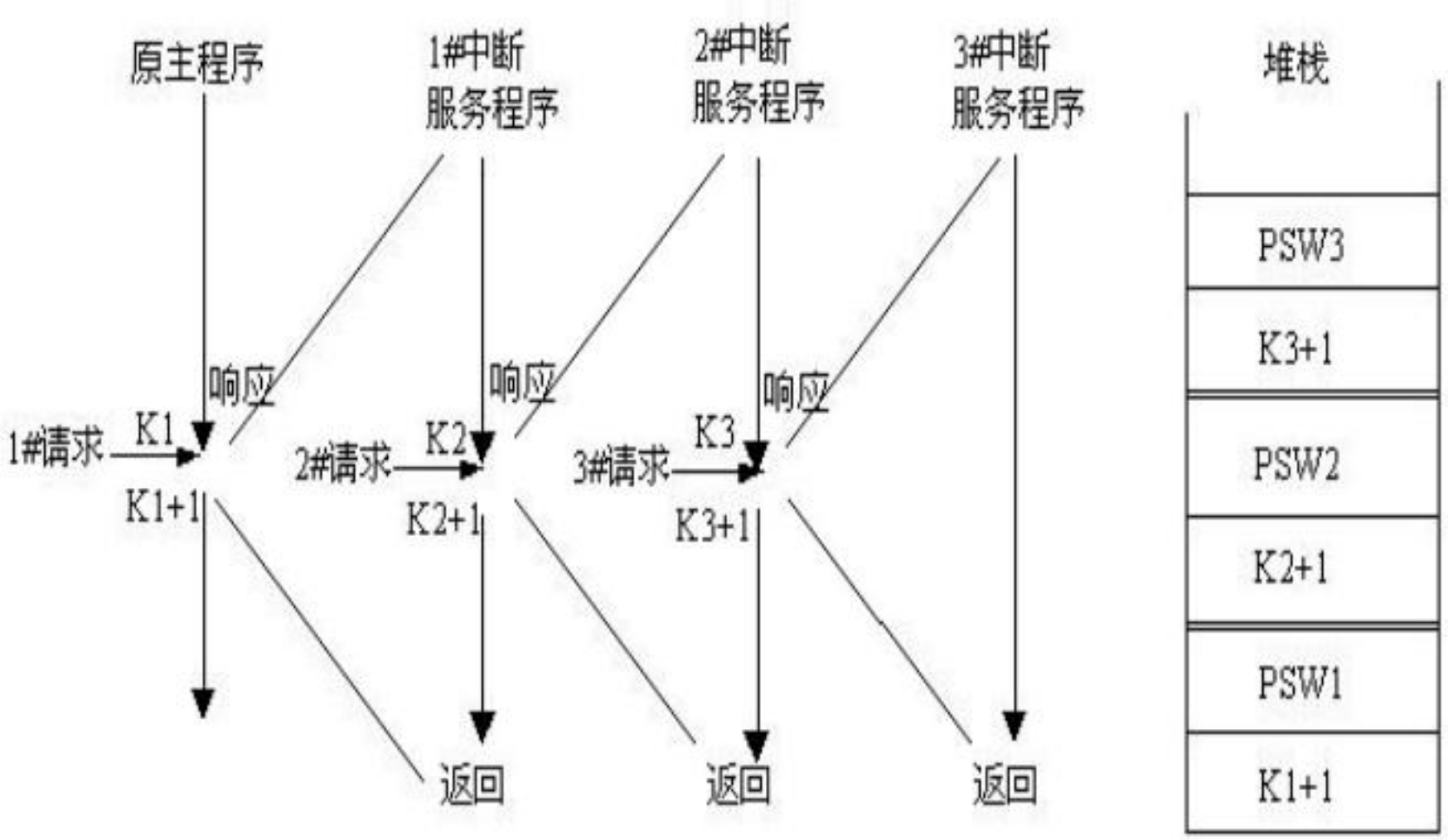
  - **中断优先级的概念：**

    中断响应优先级----由查询程序或硬联排队线路决定的优先权，反映多个中断同时请求时选择哪个响应。

    中断处理优先级----由各自的中断屏蔽字来动态设定，反映本中断与其它中断间的关系。

    回想一下，中断屏蔽字在何处用到的?

多重中断嵌套



中断处理优先级的顺序是：

3# > 2# > 1#

1# 对 2# 开放（不屏蔽）

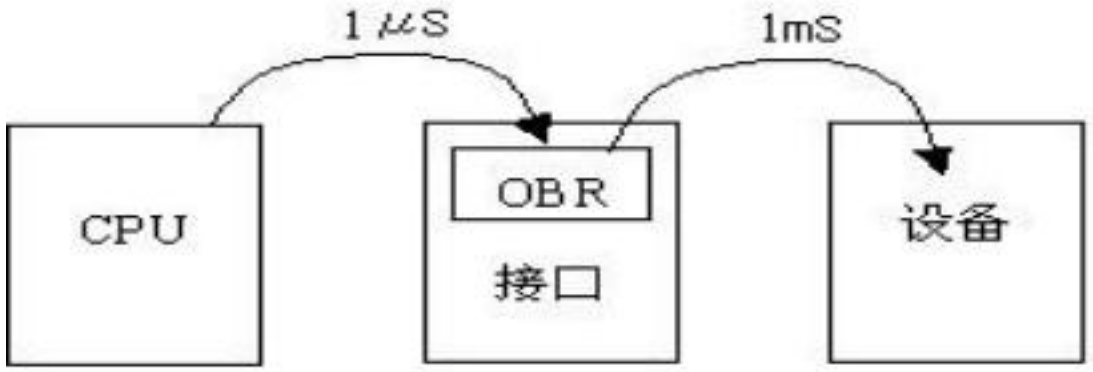
2# 对 3# 开放（不屏蔽）

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1μs。再由OBR输出到设备，需要1ms。设一条指令的执行时间为1μs(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

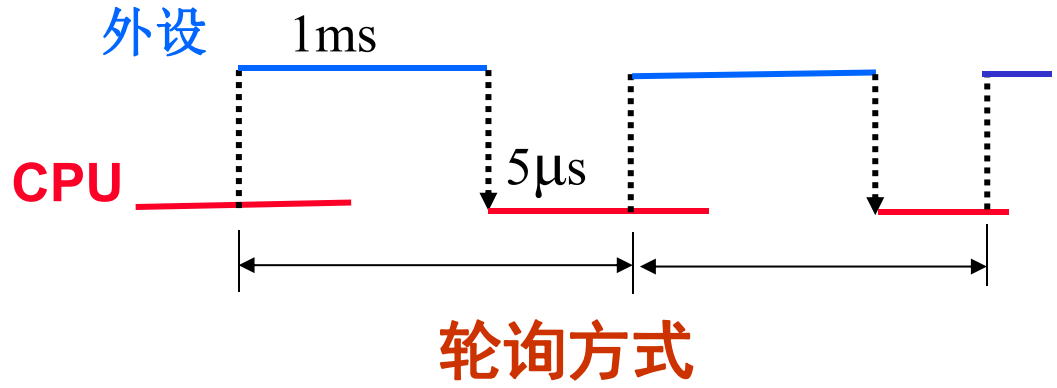
轮询方式和中断方式的比较

（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) μs，约为每秒995个数据。

主机占用率=100%



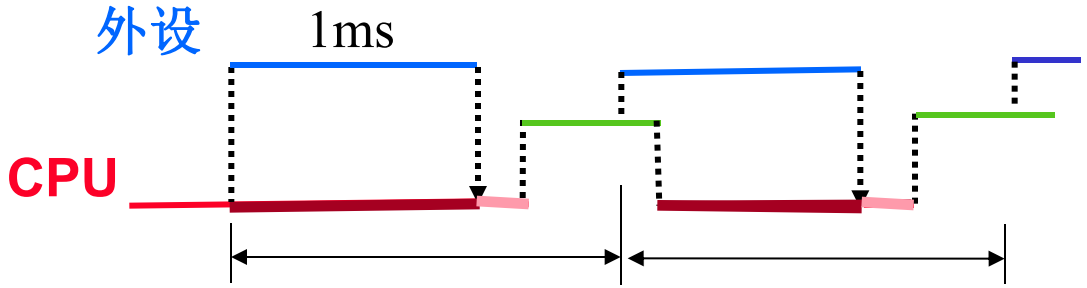
轮询方式

（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)μs，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

DMA方式的基本要点

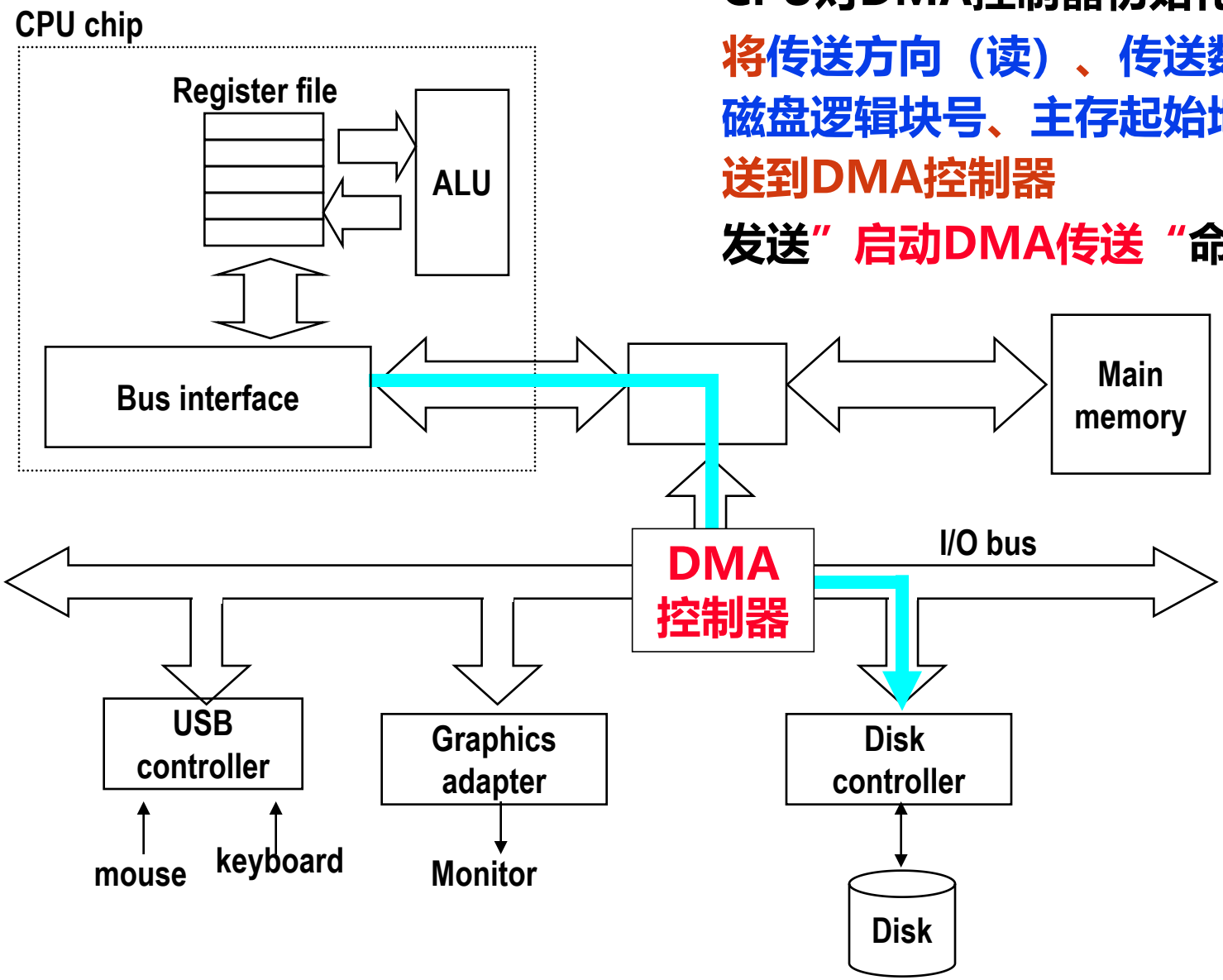
- DMA方式的基本思想
  - 在高速外设和主存间直接传送数据
  - 由专门硬件（即：DMA控制器）控制总线进行传输
- DMA方式适用场合
  - 高速设备（如：磁盘、光盘等）
  - 成批数据交换，且数据间间隔时间短，一旦启动，数据连续读写
- 采用“请求-响应”方式
  - 每当高速设备准备好数据就进行一次“DMA请求”，DMA控制器接受到DMA请求后，申请总线使用权
  - DMA控制器的总线使用优先级比CPU高，为什么?
- 与中断控制方式结合使用
  - 在DMA控制器控制总线进行数据传送时，CPU执行其他程序
  - DMA传送结束时，要通过“DMA结束中断”告知CPU

读一个磁盘扇区 – 第一步

CPU对DMA控制器初始化：

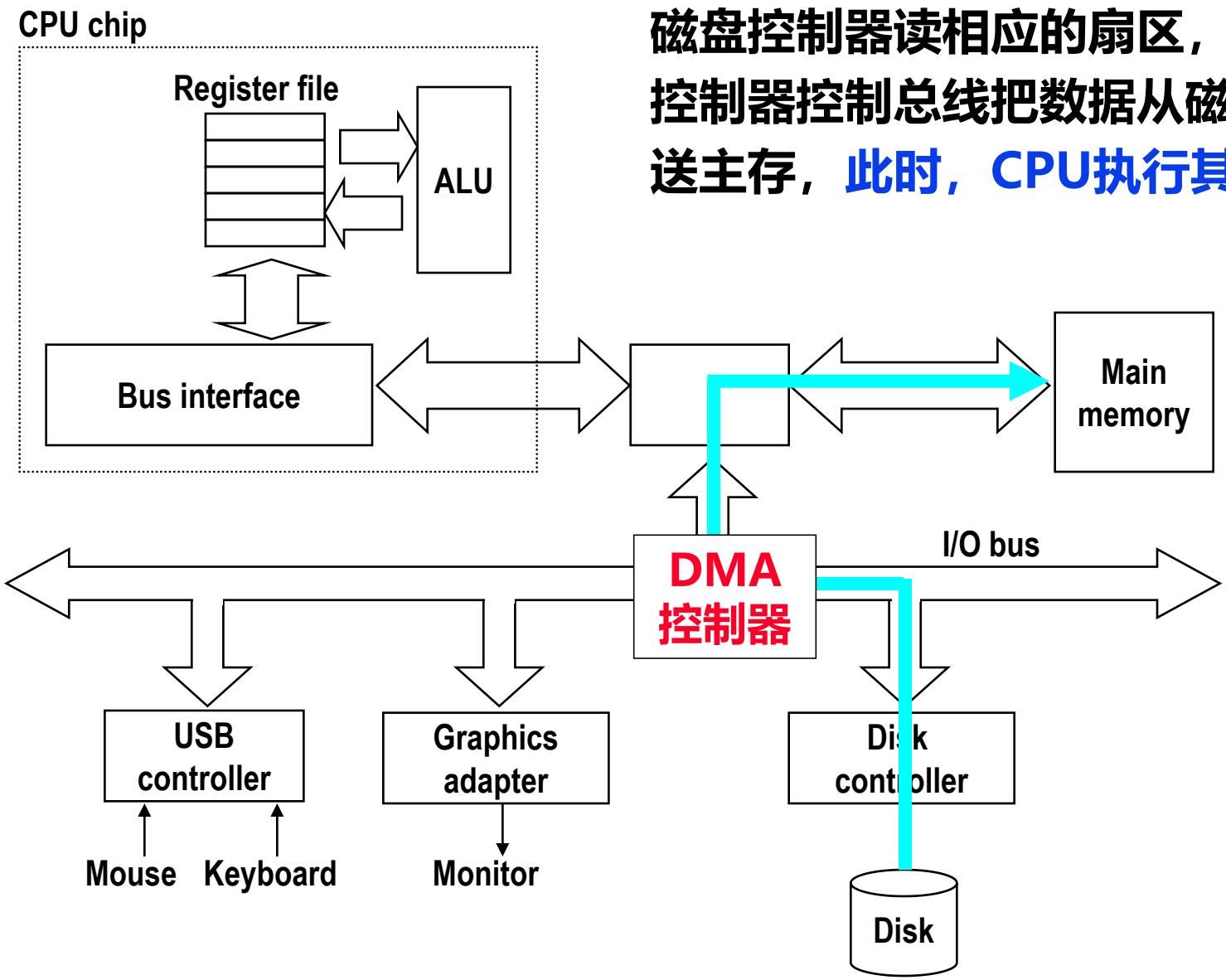
将传送方向（读）、传送数据个数、磁盘逻辑块号、主存起始地址等参数送到DMA控制器

发送”启动DMA传送“命令



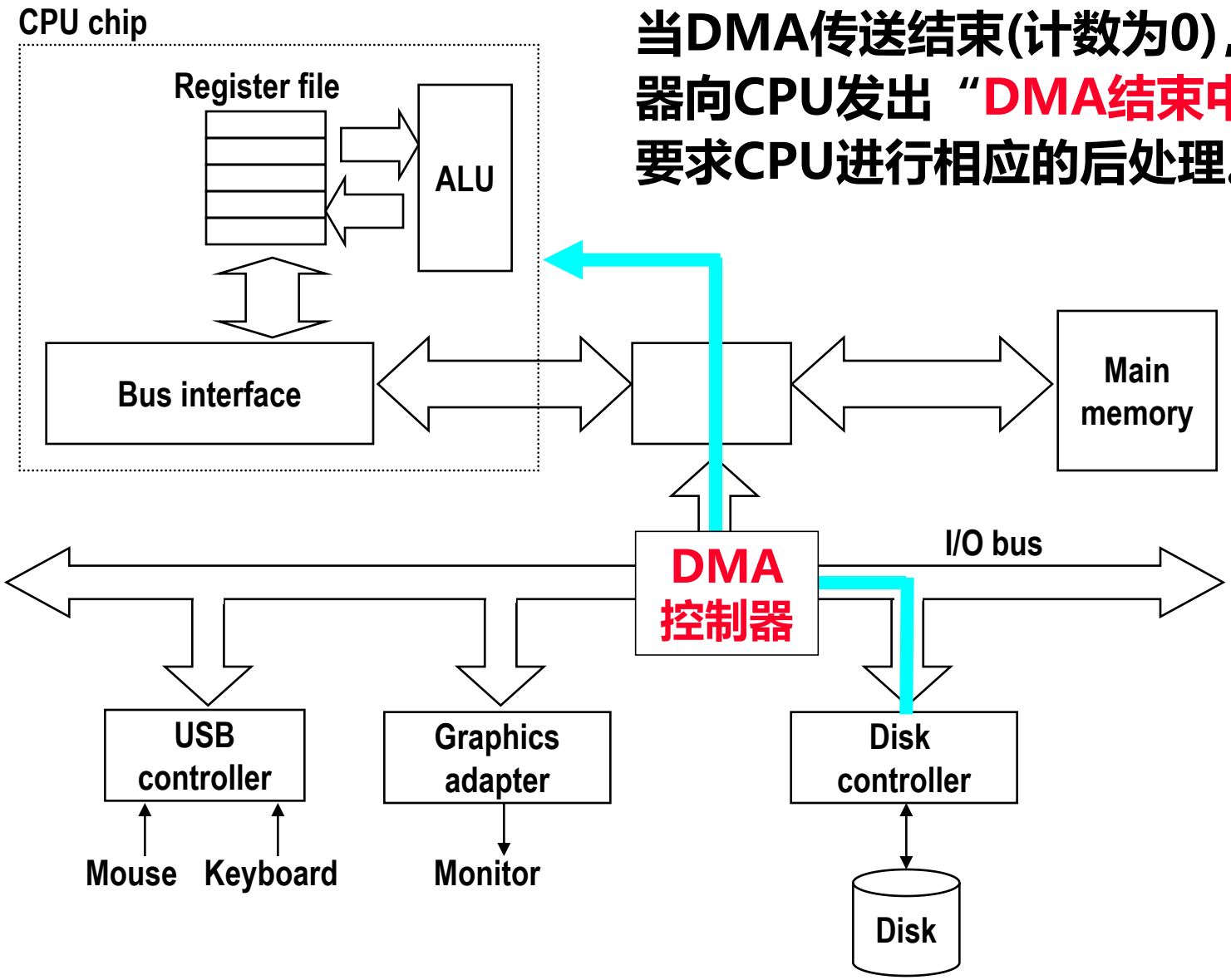
读一个磁盘扇区 - 第二步

磁盘控制器读相应的扇区，并由DMA控制器控制总线把数据从磁盘控制器送主存，此时，CPU执行其他进程



读一个磁盘扇区 - 第三步

当DMA传送结束(计数为0)，DMA控制器向CPU发出“DMA结束中断请求”，要求CPU进行相应的后处理。



DMA方式下CPU的工作

例子：采用中断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。

例：中断、DMA方式下CPU的开销

设处理器按500MHz的速度执行，硬盘控制器中有一个16B的数据缓存器，磁盘传输速率为4MB/Sec，在磁盘传输数据过程中，要求没有任何数据被错过，并假定CPU访存和DMA访存没有冲突。

(1) 若用中断方式，每次传送的开销（包括用于中断响应和处理的时间）是500个时钟周期。如果硬盘仅用5%的时间进行传送，那么处理器用在硬盘I/O操作上所花的时间百分比（主机占用率）为多少?

(2) 若用DMA方式，处理器用1000个时钟进行DMA传送初始化，在DMA完成后的中断处理需要500个时钟。如果每次DMA传送8000B的数据块，那么当硬盘进行传送的时间占100%（即：硬盘一直进行读写，并传输数据）时，处理器用在硬盘I/O操作上的时间百分比（主机占用率）为多少?

想象一下：假定大仓库门口有一个箱子，可放16个零件。要将大仓库中的一批零件运到小仓库中，可以有几种方法?

中断方式：每装满一个箱子就喊车床上的技工来运到车间，再从车间运到小仓库

DMA方式：车床技工停下来告诉搬运工说，一次要8000个零件放到小仓库固定的地方，然后回到车床工作；搬运工开始分两组工作，一组从大仓库搬货到箱子中，另一组将箱子直接运到小仓库指定地方，搬完8000个后， 搬运工告知技工已完成任务，技工进行相应处理。

上述两种方式中，哪种方式的生产效率更高呢?

例：中断、DMA方式下CPU的开销

一旦磁盘被启动传送，就以4MB/s的速度进行，主机要保证没有数据丢失！

- **中断传送：**
  - 硬盘每次中断，可以以16字节为单位进行传送，为保证没有任何数据被错过，应达到每秒4MB /16B=250k次中断的速度；
  - 每秒钟用于中断的时钟周期数为250kx500=$125x10^6$；
  - 在一次数据传输中，处理器花费在I/O上的时间的百分比为：$125x10^6/(500x10^6)$=25%；
  - 假定硬盘仅用其中5%的时间来传送数据，则处理器花费在I/O方面的百分比为25%x5%=1.25% 。
- **DMA传送：**
  - 每次DMA传送将花费8000B/(4MB/Sec)≈$2x10^{-3}$秒；
  - 一秒钟内有$1/(2x10^{-3})$=500次DMA传送；
  - 如果硬盘一直在传送数据的话，处理器必须每秒钟花(1000+500)x500=$750x10^3$个时钟周期来为硬盘I/O操作服务；
  - 在硬盘I/O操作上处理器花费的时间占：

$$750x10^3/(500x10^6)=1.5x10^{-3}=0.15\%$$ 。

I/O操作的实现

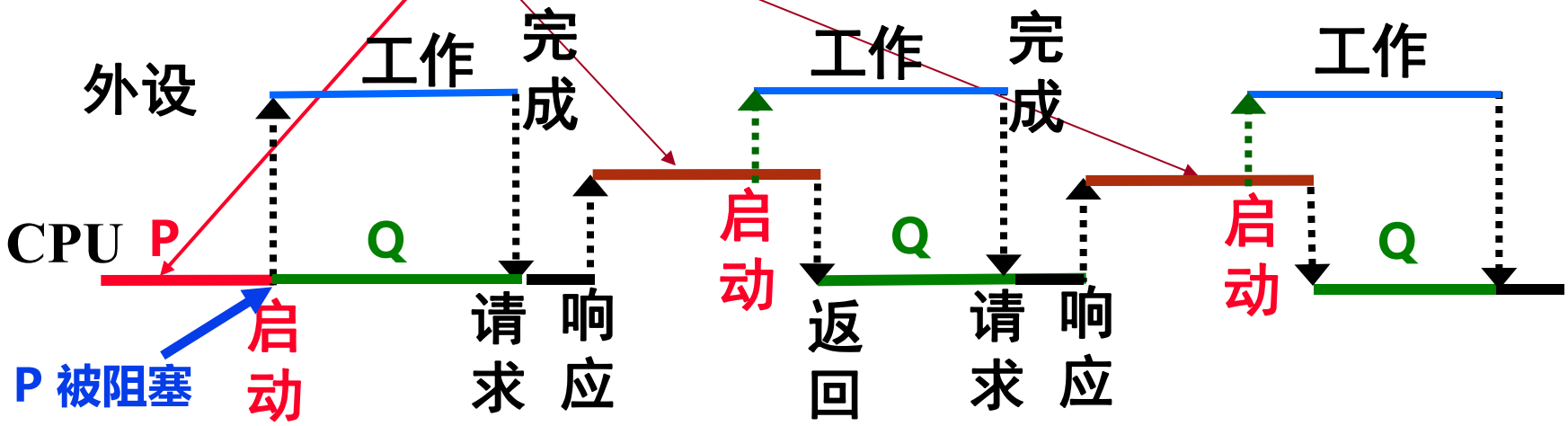
- 分以下三个部分介绍
  - 第一讲：用户空间I/O软件
    - I/O子系统概述
    - 一切设备皆文件
    - 用户空间的I/O函数
  - 第二讲：I/O硬件和软件的接口
    - I/O设备和设备控制器
    - I/O端口及其编址方式
    - I/O控制方式
  - 第三讲：内核空间I/O软件
    - 与设备无关的I/O软件
    - 设备驱动程序
    - 中断服务程序

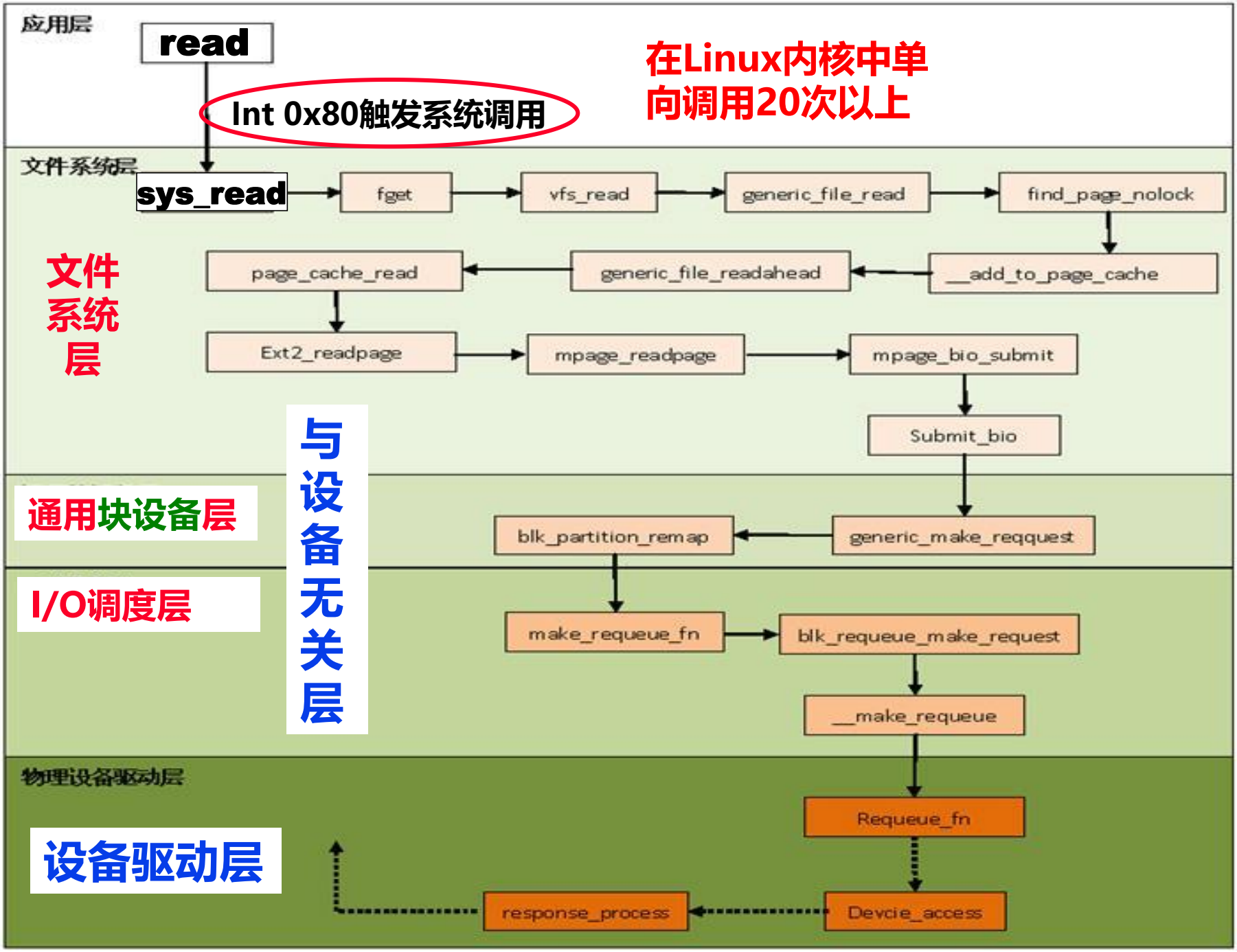
内核空间I/O软件

- 所有用户程序提出的I/O请求，最终都通过系统调用实现
- 通过系统调用封装函数中的陷阱指令转入内核I/O软件执行
- 内核空间I/O软件实现相应系统调用的服务功能
- 内核空间的I/O软件分三个层次
  - 设备无关软件层
  - 设备驱动程序层
  - 中断服务程序层

  } 系统调用服务例程，被陷阱指令调出执行，一旦发送“启动”命令，则所代表的进程被送等待队列（即被阻塞）

- 设备驱动程序层、中断服务程序层与I/O硬件密切相关




应用层
read
Int 0x80触发系统调用
在Linux内核中单向调用20次以上
文件系统层
sys_read
fget
vfs_read
generic_file_read
find_page_nolock
文件系统层
page_cache_read
generic_file_readahead
__add_to_page_cache
Ext2_readpage
mpage_readpage
mpage_bio_submit
Submit_bio
通用块设备层
与设备无关层
blk_partition_remap
generic_make_reqquest
I/O调度层
make_requeue_fn
blk_requeue_make_request
__make_requeue
物理设备驱动层
Requeue_fn
设备驱动层
response_process
Devcie_access

设备无关I/O软件层

- 设备驱动程序统一接口
  - 操作系统为所有外设的设备驱动程序规定一个统一接口，这样，新设备的驱动程序只要按统一接口规范来编制，就可在不修改操作系统的情况下，添加新设备驱动程序并使用新的外设进行I/O。
  - 所有设备都抽象成文件，设备名和文件名在形式上没有差别，设备和文件具有统一的接口，不同设备名和文件名被映射到对应设备驱动程序。
- 缓冲处理
  - 每个设备的I/O都需使用内核缓冲区，因而缓冲区的申请和管理等处理是所有设备公共的，可包含在与设备无关的I/O软件部分
- 错误报告
  - I/O操作在内核态执行时所发生的错误信息，都通过与设备无关的I/O软件返回给用户进程，也即：错误处理框架与设备无关。
  - 直接返回编程等错误，无需设备驱动程序处理，如，请求了不可能的I/O操作；写信息到一个输入设备或从一个输出设备读信息；指定了一个无效缓冲区地址或者参数；指定了不存在的设备等。
  - 有些错误由设备驱动程序检测出来并处理，若驱动程序无法处理，则将错误信息返回给设备无关I/O软件，再由设备无关I/O软件返回给用户进程，如写一个已被破坏的磁盘扇区；打印机缺纸；读一个已关闭的设备等。

设备无关I/O软件层

- 打开与关闭文件
  - 对设备或文件进行打开或关闭等I/O函数所对应的系统调用，并不涉及具体的I/O操作，只要直接对主存中的一些数据结构进行修改即可，这部分工作也由设备无关软件来处理。
- 逻辑块大小处理
  - 为了为所有的块设备和所有的字符设备分别提供一个统一的抽象视图，以隐藏不同块设备或不同字符设备之间的差异，与设备无关的I/O软件为所有块设备或所有字符设备设置统一的逻辑块大小。
  - 对于块设备，不管磁盘扇区和光盘扇区有多大，所有逻辑数据块的大小相同，这样，高层I/O软件就只需处理简化的抽象设备，从而在高层软件中简化了数据定位等处理

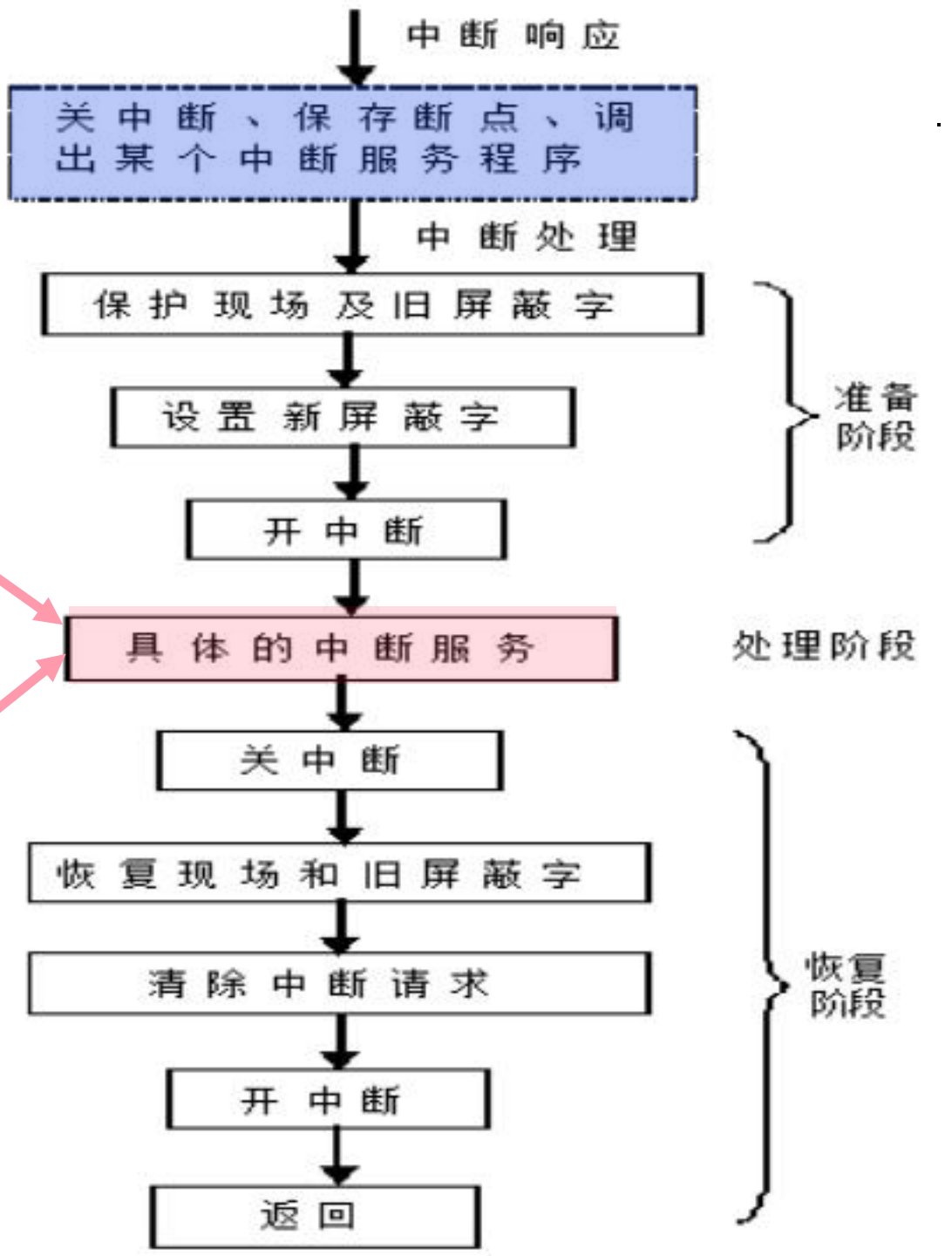
设备驱动程序

- 每个外设具体的I/O操作需通过执行设备驱动程序来完成
- 外设种类繁多、其控制接口不一，导致不同外设的设备驱动程序千差万别，因而设备驱动程序与设备相关
- 每个外设或每类外设都有一个设备控制器，其中包含各种I/O端口。 CPU通过执行设备驱动程序中的I/O指令访问个各种I/O端口
- 设备所采用的I/O控制方式不同，驱动程序的实现方式也不同
  - 程序直接控制：驱动程序完成用户程序的I/O请求后才结束。这种情况下，用户进程在I/O过程中不会被阻塞，内核空间的I/O软件一直代表用户进程在内核态进行I/O处理 。
  - 中断控制：驱动程序启动第一次I/O操作后，将调出其他进程执行，而当前用户进程被阻塞。在CPU执行其他进程的同时，外设进行I/O操作，此时，CPU和外设并行工作。外设完成I/O时，向CPU发中断请求，然后CPU调出相应中断服务程序执行。在中断服务程序中再次启动I/O操作。
  - DMA控制：驱动程序对DMA控制器初始化后，便发送“启动DMA传送”命令，外设开始进行I/O操作并在外设和主存间传送数据。同时CPU执行处理器调度程序，转其他进程执行，当前用户进程被阻塞。DMA控制器完成所有I/O任务后，向CPU发送一个“DMA完成”中断请求信号。

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**从数缓器取数或写数据到数缓器**，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



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