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
    - 文件的基本概念
    - 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

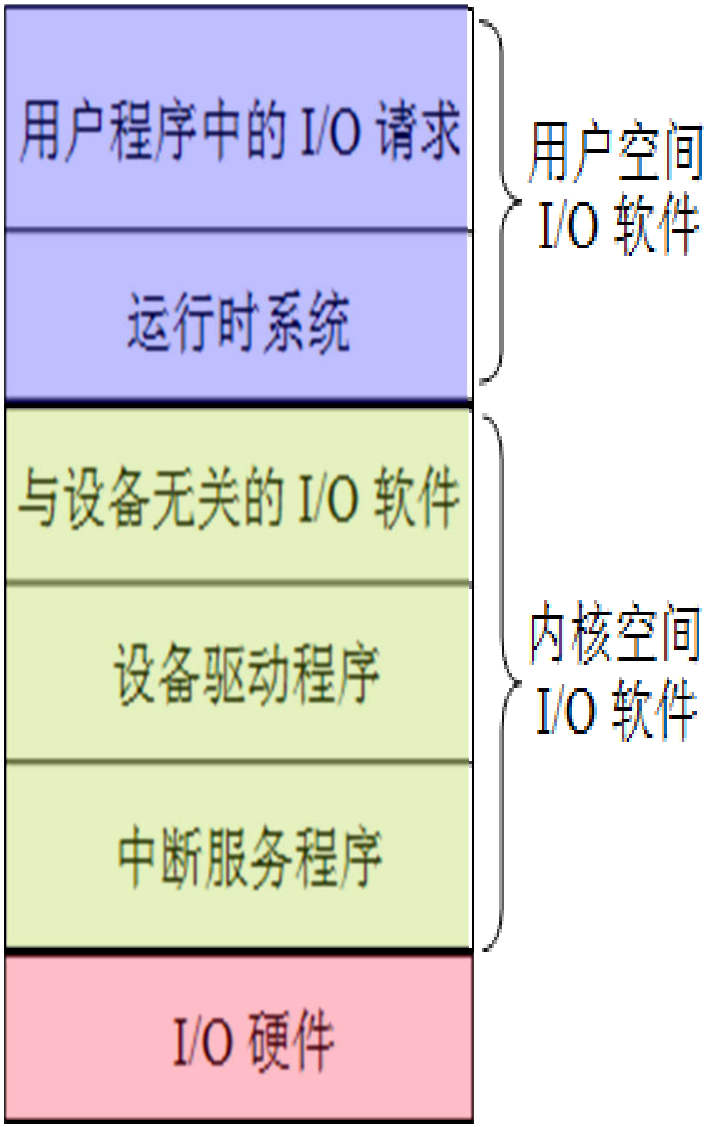
- 所有高级语言的运行时（runtime）都提供了执行I/O功能的机制

  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：

(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（串行）

(c) 有些I/O功能不适合甚至无法使用标准I/O库函数实现，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。

回顾：IA-32/Linux的系统调用

- 通常，系统调用被封装成用户程序能直接调用的函数，如exit()、read()和open()，这些是标准C库中系统调用对应的**封装函数**。
- Linux中系统调用所用参数通过寄存器传递，传递参数的寄存器顺序依次为：EAX（调用号）、EBX、ECX、EDX、ESI、EDI和EBP，除调用号以外，最多6个参数。
- 封装函数对应的机器级代码有一个统一的结构：
  - 总是若干条传送指令后跟一条陷阱指令。传送指令用来传递系统调用的参数，陷阱指令（如int $0x80）用来陷入内核进行处理。
- 例如，若用户程序调用系统调用write(1，“hello, world!\n”,14)，将字符串“hello, world!\n”中14个字符显示在标准输出设备文件stdout上，则其封装函数对应机器级代码（用汇编指令表示）如下：

```
movl  $4, %eax          //调用号为4，送EAX
movl  $1, %ebx   //标准输出设备stdout的文件描述符为1，送EBX
movl  $string, %ecx  //字符串“hello, world!\n”首址送ECX
movl  $14, %edx       //字符串的长度为14，送EDX
int    $0x80            //系统调用
```

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！

如：INT $0x80

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。

SKIP

- 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。

回顾：IA-32/Linux的系统调用

BACK

- 系统调用（陷阱）是特殊异常事件，是OS为用户程序提供服务的手段。
- Linux提供了几百种系统调用，主要分为以下几类：

  进程控制、文件操作、文件系统操作、系统控制、内存管理、网络管理、用户管理、进程通信等

- 系统调用号是系统调用跳转表索引值，跳转表给出系统调用服务例程首址

调用号	名称	类别	含义	调用号	名称	类别	含义
1	exit	进程控制	终止进程	12	chdir	文件系统	改变当前工作目录
2	fork	进程控制	创建一个新进程	13	time	系统控制	取得系统时间
3	read	文件操作	读文件	19	lseek	文件系统	移动文件指针
4	write	文件操作	写文件	20	getpid	进程控制	获取进程号
5	open	文件操作	打开文件	37	kill	进程通信	向进程或进程组发信号
6	close	文件操作	关闭文件	45	brk	内存管理	修改虚拟空间中的堆指针 brk
7	waitpid	进程控制	等待子进程终止	90	mmap	内存管理	建立虚拟页面到文件片段的映射
8	create	文件操作	创建新文件	106	stat	文件系统	获取文件状态信息
11	execve	进程控制	运行可执行文件	116	sysinfo	系统控制	获取系统信息

系统调用及其参数传递

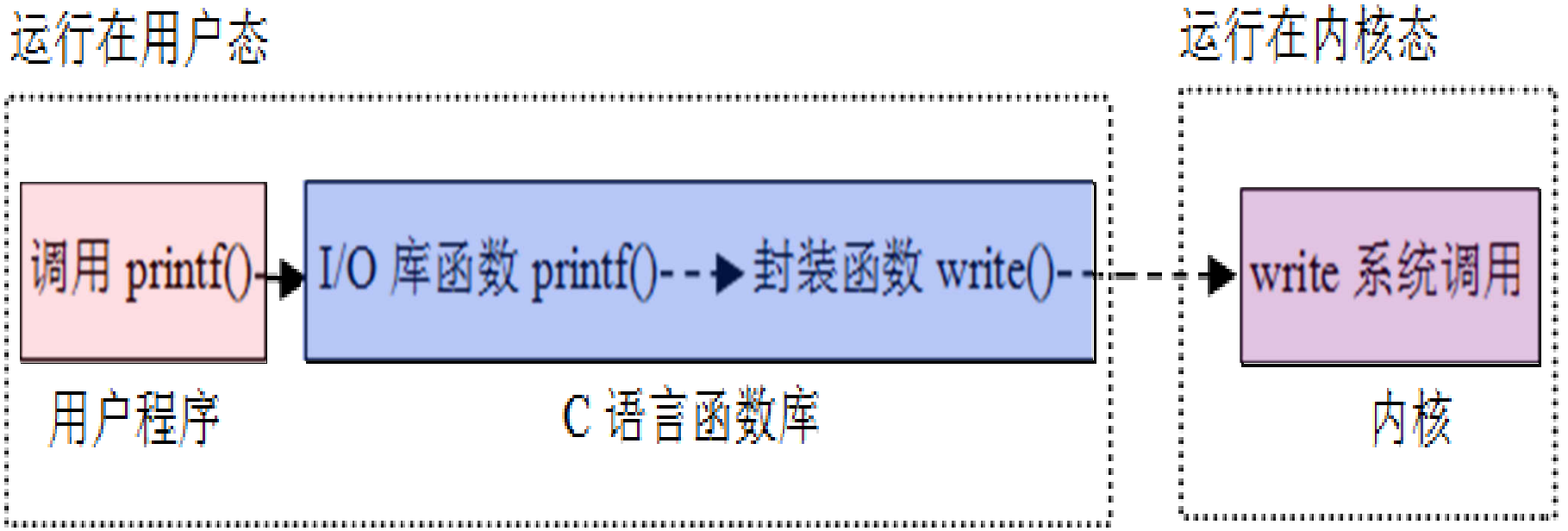
- 在用户态，当进程调用一个系统调用时，CPU切换到内核态，并开始执行一个被称为**系统调用处理程序**的内核函数
- 例如，IA-32中，可以通过两种方式调用Linux的系统调用
  - 执行软中断指令int $0x80
  - 执行指令sysenter（老的x86不支持该指令）
- 内核实现了许多系统调用，因此，用一个**系统调用号（存放在EAX中）**来标识不同的系统调用
- 除了调用号以外，系统调用还需要其他参数，不同系统调用所需参数的个数和含义不同，**输入参数通过通用寄存器传递**，若参数个数超出寄存器个数，则将需传递参数块所在内存区首址放在寄存器中传递**（除调用号以外，最多6个参数）**
  - 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库函数和内核

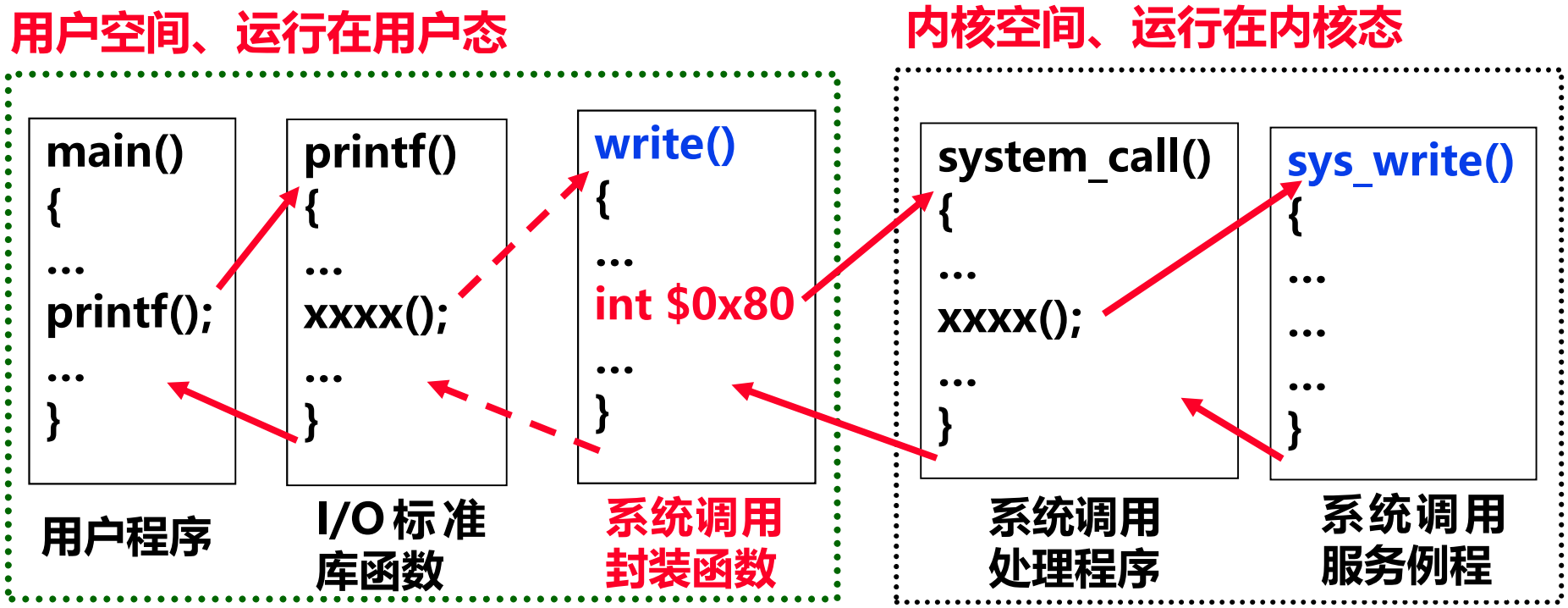
- 用户程序总是通过某种I/O函数或I/O操作符请求I/O操作。

  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：



回顾：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。

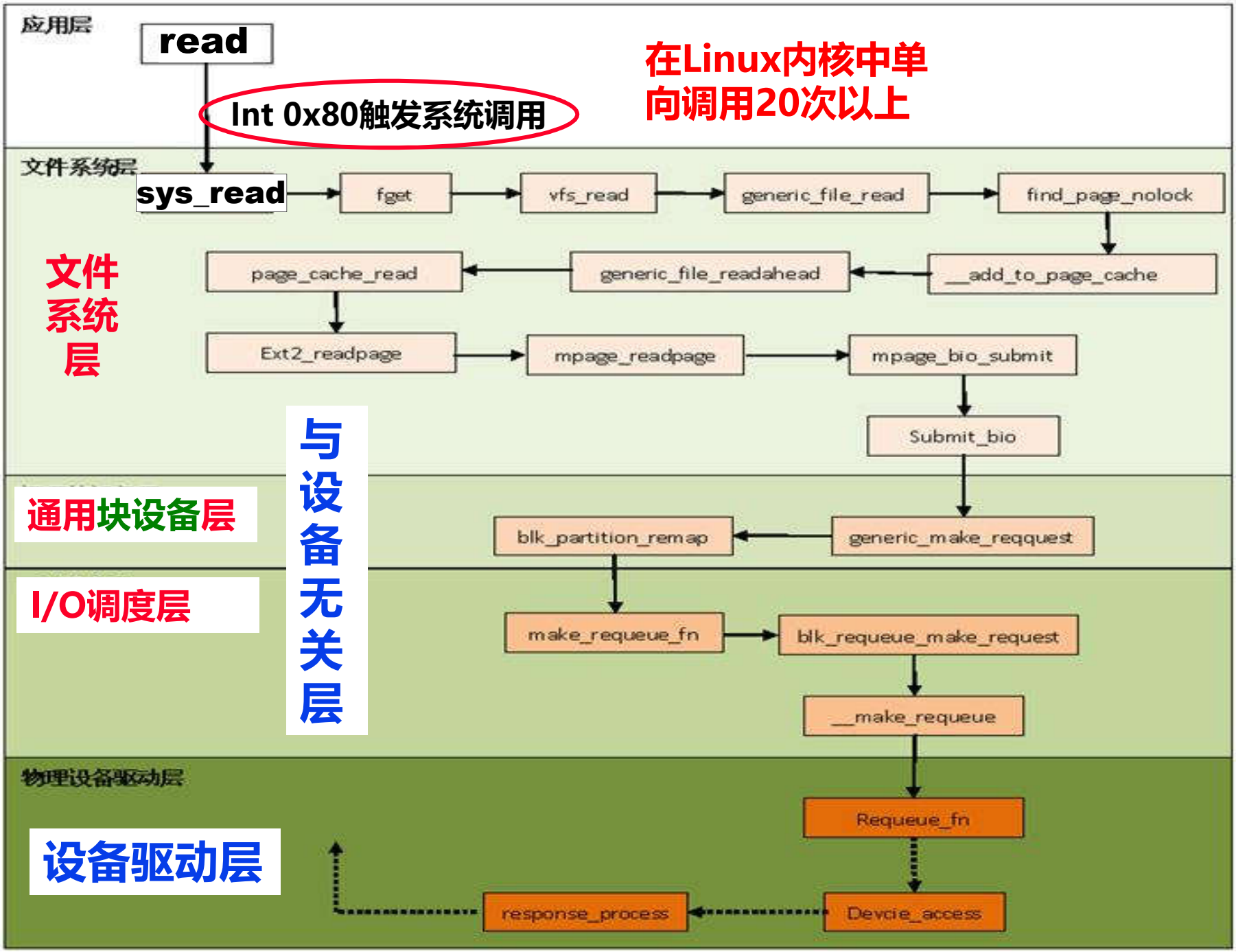
在system_call中如何知道要转到sys_write执行呢？ 根据系统调用号！

Linux系统下的write()封装函数

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

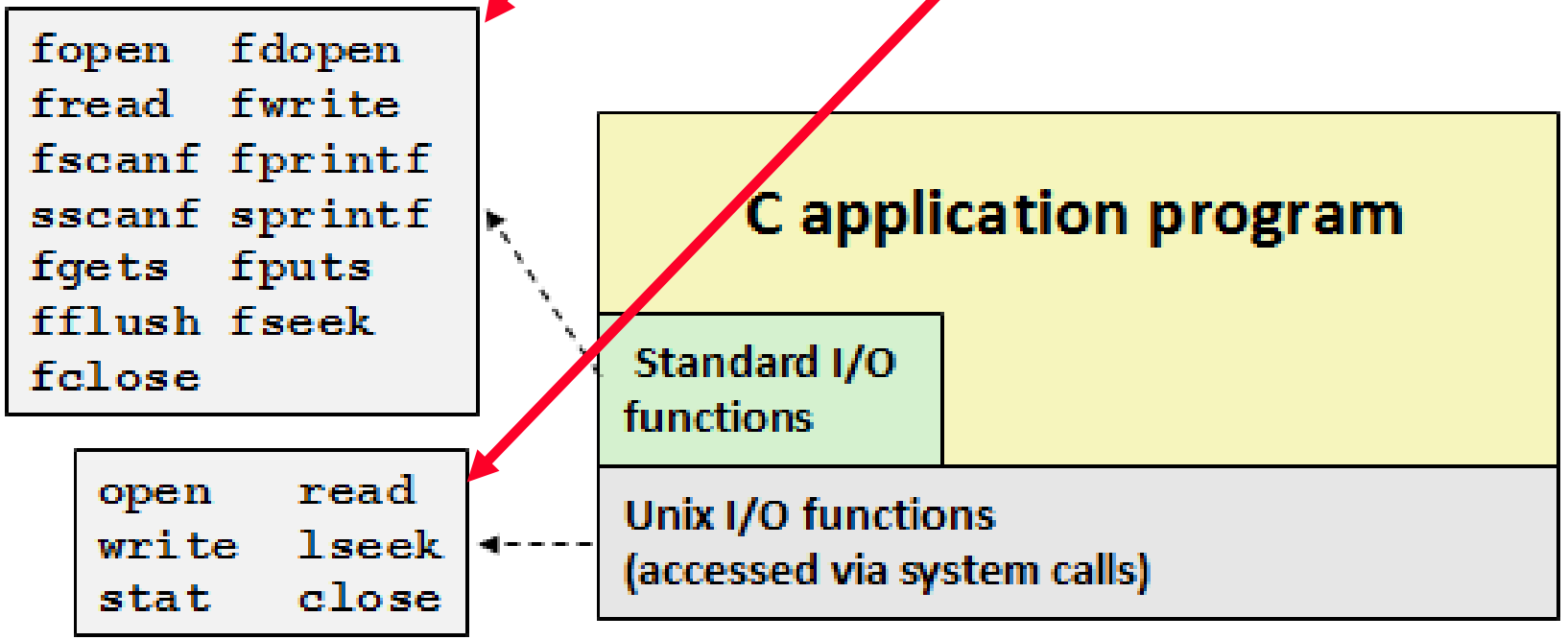
size_t 和 ssize_t 分别是 unsigned int 和 int，因为返回值可能是-1。

```
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（所有正数都小于FFFFFF83H)
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 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

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。



用户程序中的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
- ◆ 参数perms用于指定文件的访问权限，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。
- ◆ 参数flags：O_RDONLY, O_WRONLY|O_APPEND, O_RDWR等

例：fd=open（“test.txt”,O_RDONLY, 0);

文件的读/写

3. 读文件：ssize_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/写任意个字节。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。

是不带缓冲的读写：直接从（向）磁盘读（写），没有缓冲

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：开头（0）、当前位置（1）和末尾（2）

  例：lseek(fd,5L,0)；表示定位到文件开始后的第5字节

  lseek(fd, 0L, 2)；表示定位到文件末尾
- 返回的是位置值，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，由内核维护，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);

典型的stdio.h的部分内容

```
#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”，它在stdio.h中定义。

用数组实现I/O(文件)缓冲

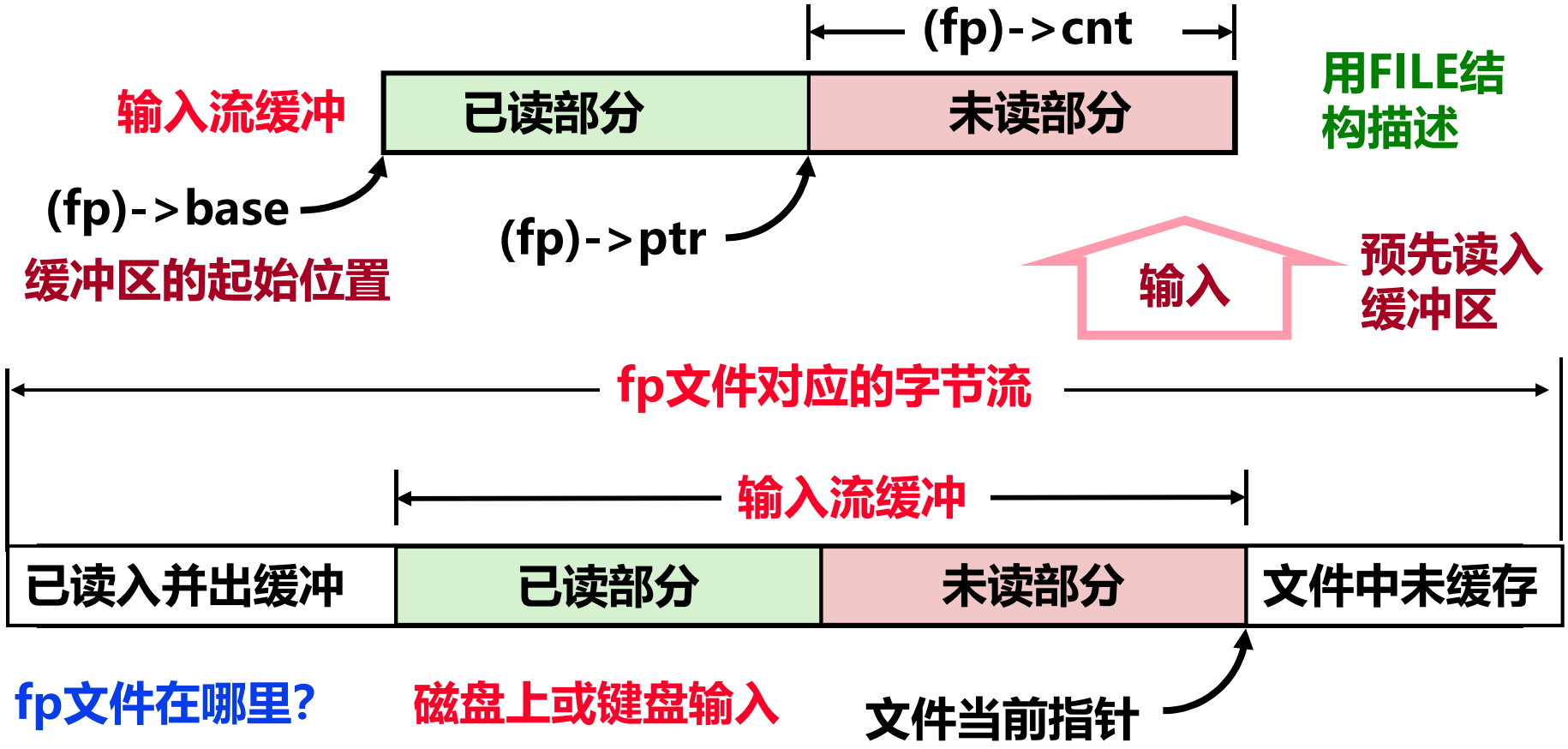
```
FILE _iob[OPEN_MAX] = {
{ 0, ( char * ) 0, ( char * ) 0, _READ, 0 },
{ 0, ( char * ) 0, ( char * ) 0, _WRITE, 1 },
{ 0, ( char * ) 0, ( char * ) 0, _WRITE | _UNBUF, 2 },
};
```

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

带缓冲有何好处?

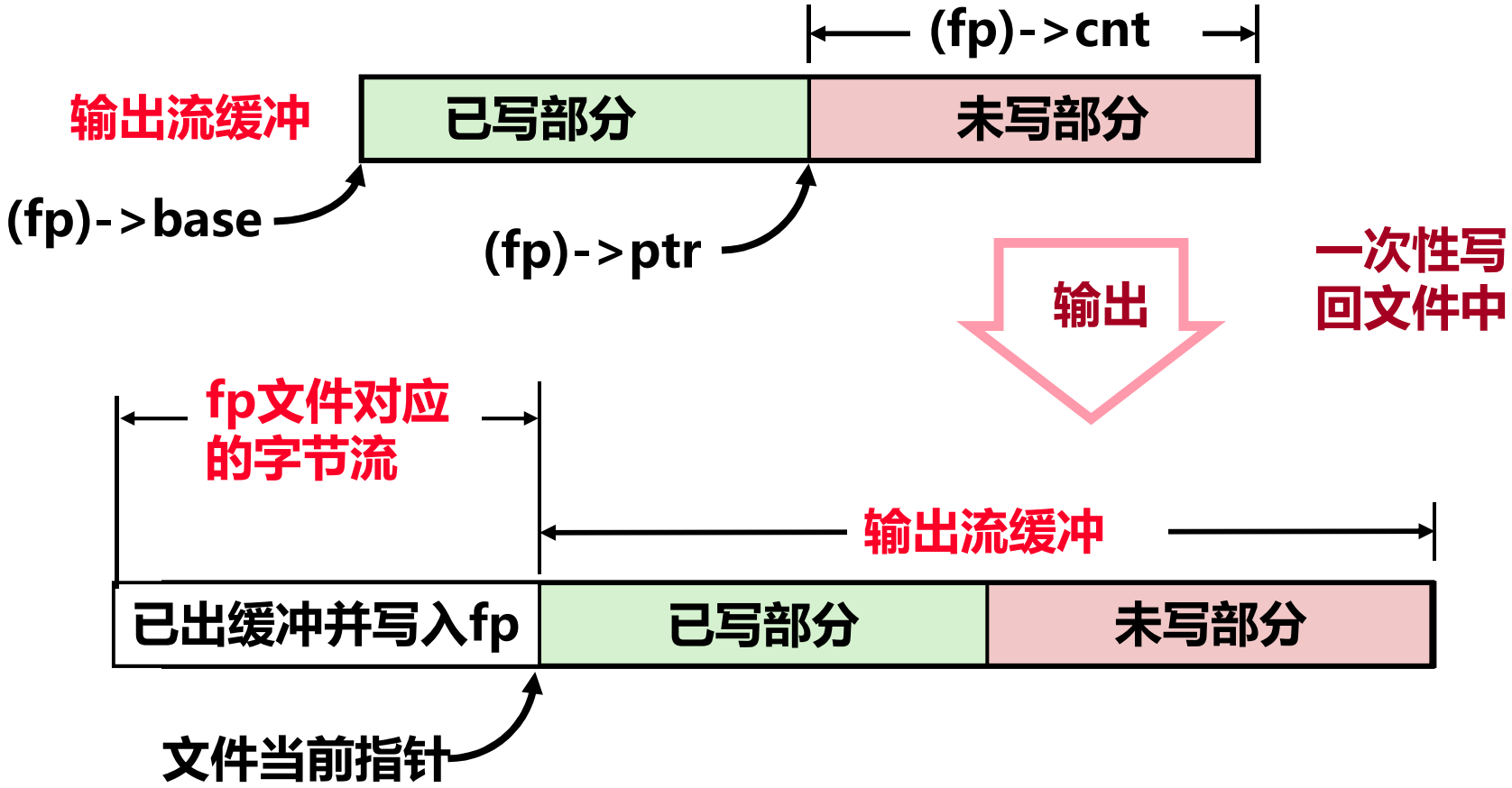
带缓冲I/O的实现

- 从文件fp中读数据时，FILE中定义的缓冲区为输入流缓冲（在内存）
- 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putc）或n个（如fwrite）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

猜一下在Linux中以下程序输出什么?

```
#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

例子（可执行文件为hello）

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

二者都默认指向标准输出，即显示器；也都可重定位到普通文件中！

执行结果如下：

./hello > out.txt：stdout送out.txt, stderr送屏幕

./hello 2 > err.txt：stdout送屏幕, stderr送err.txt

./hello > out.txt 2> err.txt：stdout送out.txt，stderr送err.txt

./hello > combine.txt 2>&1：stdout和stderr都送combine.txt

./hello > combine.txt 2> combine.txt：

stdout和stderr都送combine.txt

stdio.h中更多的定义

- 在stdio.h中，还有feof()、ferror()、fileno()、getc()、putc()、getchar()、putchar()等宏定义。
- 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。

系统调用的开销很大！

```
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。调用_fillbuf()后，cnt<=1023。

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

SKIP

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

BACK

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
- ◆ 参数perms用于指定文件的访问权限，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。
- ◆ 参数flags：O_RDONLY, O_WRONLY|O_APPEND, O_RDWR等

例：fd=open（“test.txt”,O_RDONLY, 0);

_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=0 */
        if (fp->cnt == -1) fp->flag | = _EOF;
        else fp->flag | = _ERR;
        fp -> cnt =0;
        return EOF;
    }
    return (unsigned char ) *fp->ptr++;   /* 0<cnt<=1023 */
}
```

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(FILE *infp, FILE *outfp)
{
   char c;
   while (read(infp->fd,&c,1) != 0)
            write(outfp->fd,&c,1);
}
```

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!

还有其他的实现方式吗?

SKIP

使用fread()和fwrite()

使用fgetc()和fputc()

使用WindowsAPI函数CopyFile()

实现一个功能有多种方式，但开销和性能不同，需要权衡！

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）
```

回顾：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。

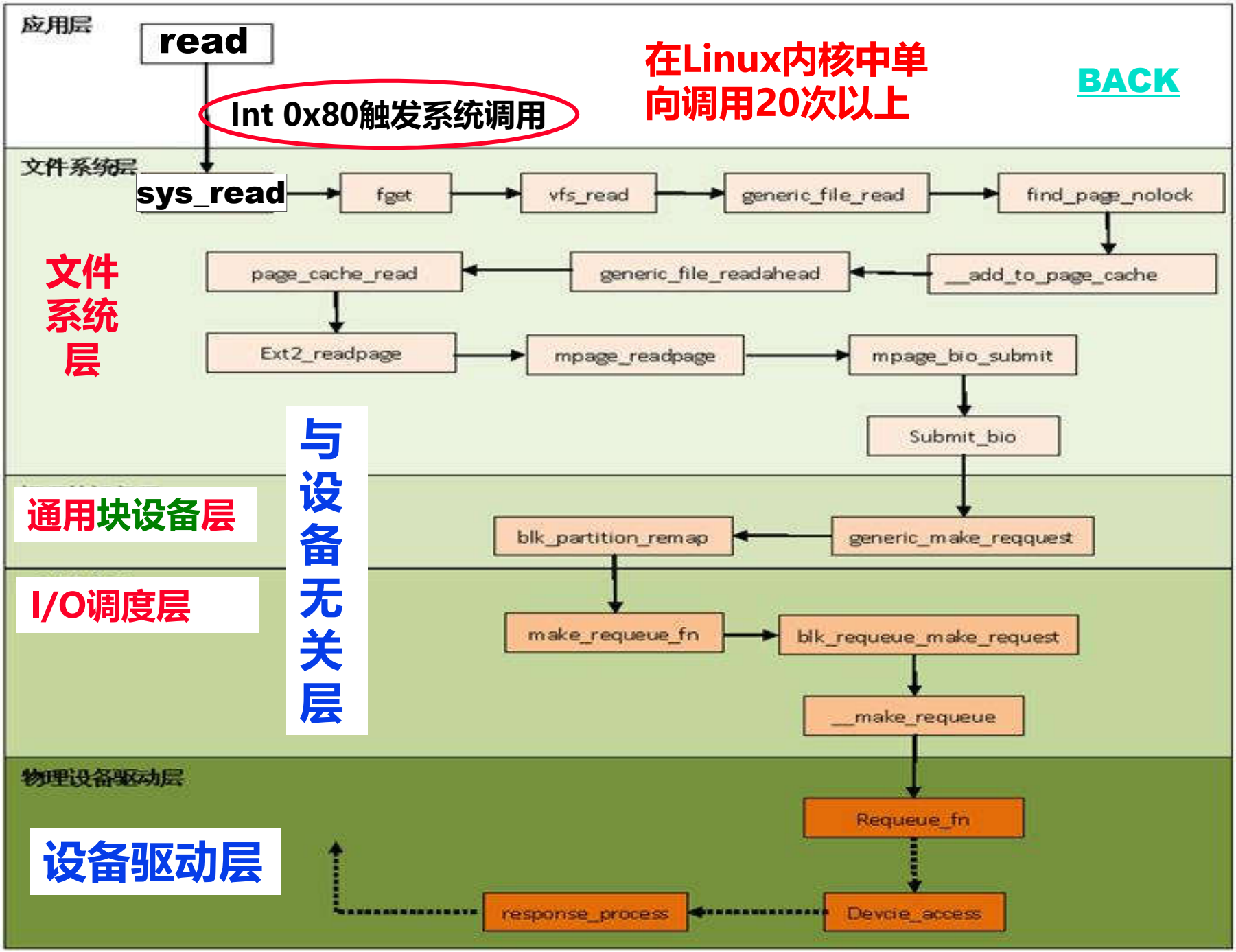
2) 读 TR 寄存器，以访问TSS，从TSS中将内核栈的段寄存器内容和栈指针装入SS和ESP；

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。

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。


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
BACK
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