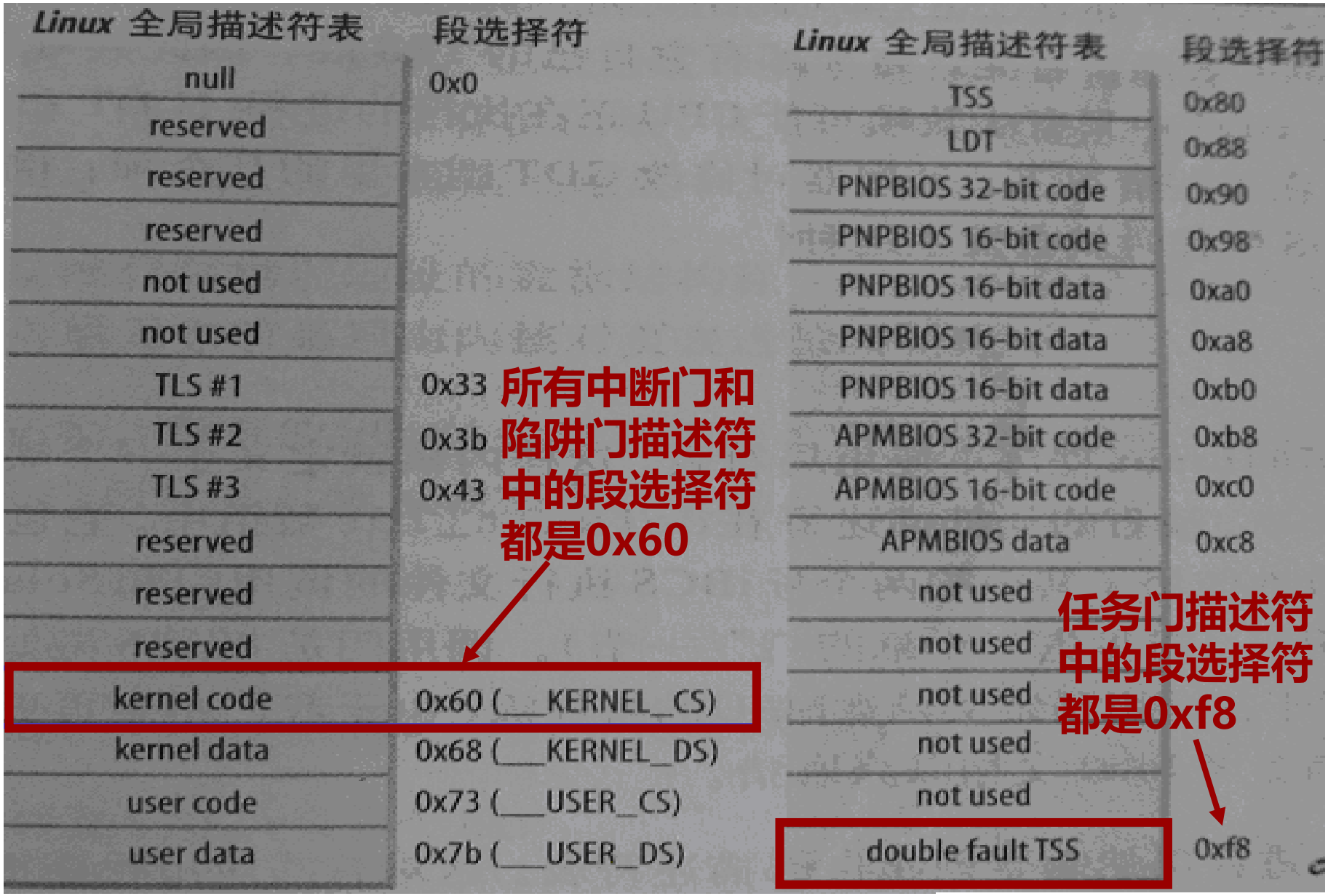
Linux中的异常和中断处理

- Linux利用陷阱门来处理异常，利用中断门来处理中断。
- 异常和中断对应处理程序都属于内核代码段，所以，所有中断门和陷阱门的段选择符(0x60)都指向 GDT 中的“内核代码段”描述符。
- 通过中断门进入到一个中断服务程序时，CPU 会清除 EFLAGS 寄存器中的 IF 标志，即关中断；通过陷阱门进入一个异常处理程序时，CPU 不会修改 IF 标志。也就是说，外部中断不支持嵌套处理，而内部异常则支持嵌套处理。 Why？试想一下处理缺页时又有非法指令或栈溢出等！
- 任务门描述符中不包含偏移地址，只包含 TSS 段选择符，这个段选择符指向 GDT 中的一个 TSS 段描述符，CPU 根据 TSS 段中的相关信息装载 SS 和 ESP 等寄存器，从而执行相应的异常处理程序。
- Linux中，将类型号为8的双重故障（#DF）用任务门实现，而且是唯一通过任务门实现的异常。
- 双重故障 TSS 段描述符在 GDT 中位于索引值为 0x1f 的表项处，即13位索引为0 0000 0001 1111，且其TI=0（指向 GDT），RPL=00（内核级代码），即任务门描述符中的段选择符为00F8H。

Linux中的中断门、陷阱门和任务门



Linux中中断描述符表的初始化

CPU负责对异常和中断的检测与响应，而操作系统则负责初始化 IDT 以及编制好异常处理程序或中断服务程序。Linux运用提供的三种门描述符格式，构造了以下5种类型的门描述符。

(1) 中断门：DPL=0，TYPE=1110B。激活所有中断

(2) 系统门：DPL=3，TYPE=1111B。激活4、5和128三个陷阱异常，分别对应指令into、bound和int $0x80三条指令。因DPL为3，CPL≤DPL，故在用户态下可使用这三条指令

(3) 系统中断门：DPL=3，TYPE=1110B。激活3号中断（即调试断点），对应指令int 3。因DPL为3，CPL≤DPL，故用户态下可使用int 3指令。

(4) 陷阱门：DPL=0，TYPE=1111B。激活所有内部异常，并阻止用户程序使用INT n（n≠128或3）指令模拟非法异常来陷入内核态运行。

(5) 任务门：DPL=0，TYPE=0101B。激活8号中断（双重故障）。

Linux内核在启用异常和中断机制之前，先设置好 IDT 的每个表项，并把 IDT 首址存入 IDTR。系统初始化时，Linux完成对 GDT、GDTR、IDT 和 IDTR 等的设置，以后一旦发生异常或中断，CPU就可通过异常和中断响应机制调出异常或中断处理程序执行。

Linux中对异常的处理

- 异常处理程序发送相应的信号给发生异常的当前进程，或者进行故障恢复，然后返回到断点处执行。

  例如，若执行了非法操作，CPU就产生6号异常（#UD），在对应的异常处理程序中，向当前进程发送一个SIGILL信号，以通知当前进程中止运行。

- 采用向发生异常的进程发送信号的机制实现异常处理，可尽快完成在内核态的异常处理过程，因为异常处理过程越长，嵌套执行异常的可能性越大，而异常嵌套执行会付出较大的代价。

- 并不是所有异常处理都只是发送一个信号到发生异常的进程。

  例如，对于14号页故障异常（#PF），需要判断是否访问越级、越权或越界等，若发生了这些无法恢复的故障，则页故障处理程序发送SIGSEGV信号给发生页故障异常的进程；若只是缺页，则页故障处理程序负责把所缺失页面从磁盘装入主存，然后返回到发生缺页故障的指令继续执行。

Linux中对异常的处理

所有异常处理程序的结构是一致的，都可划分成以下三个部分：

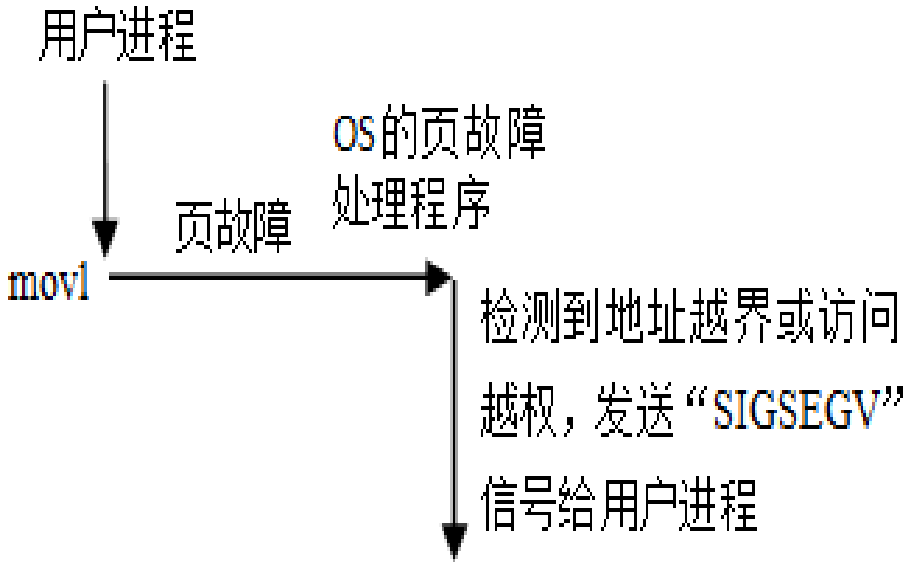
（1）**准备阶段：**在内核栈保存通用寄存器内容（称为现场信息），这部分大多用汇编语言程序实现。

（2）**处理阶段：**采用C函数进行具体处理。函数名由do_前缀和处理程序名组成，如 do_overflow 为溢出处理函数。

大部分函数的处理方式：保存硬件出错码（如果有的话）和异常类型号，然后，向当前进程发送一个信号。

当前进程接受到信号后，若有对应信号处理程序，则转信号处理程序执行；若没有，则调用内核abort例程执行，以终止当前进程。

（3）**恢复阶段：**恢复保存在内核栈中的各个寄存器的内容，切换到用户态并返回到当前进程的断点处继续执行。



Linu

Linux中异常对应的信号名和处理程序名

异常处理在内核态

信号处理在用户态

为何除法错显示却是“浮点异常”的原因

类型号	助记符	含义描述	处理程序名	信号名
0	#DE	除法出错	divide_error( )	SIGFPE
1	#DB	单步跟踪	debug( )	SIGTRAP
2		NMI 中断	nmi( )	无
3	#BP	断点	int3( )	SIGTRAP
4	#OF	溢出	overflow( )	SIGSEGV
5	#BR	边界检测（BOUND）	bounds( )	SIGSEGV
6	#UD	无效操作码	invalid( )	SIGILL
7	#NM	协处理器不存在	device_not_available( )	无
8	#DF	双重故障	doublefault( )	无
9	#MF	协处理器段越界	coprocessor_segment_overrun( )	SIGFPE
10	#TS	无效 TSS	invalid_tss( )	SIGSEGV
11	#NP	段不存在	segment_not_present( )	SIGBUS
12	#SS	栈段错	stack_segment( )	SIGBUS
13	#GP	一般性保护错（GPF）	general_protecton( )	SIGSEGV
14	#PF	页故障	page_fault( )	SIGSEGV
15		保留	无	无
16	#MF	浮点错误	coprocessor_error( )	SIGFPE
17	#AC	对齐检测	alignment_check( )	SIGSEGV
18	#MC	机器检测异常	machine_check( )	无
19	#XM	SIMD 浮点异常	simd_coprocessor_error( )	SIGFPE

回顾：用“系统思维”分析问题

代码段一：

```
int a = 0x80000000;
int b = a / -1;
printf("%d\n", b);
```

运行结果为-2147483648

objdump反汇编代码，得知除以 -1 被优化成取负指令neg，故未发生除法溢出

代码段二：

```
int a = 0x80000000;
int b = -1;
int c = a / b;
printf("%d\n", c);
```

a/b用除法指令IDIV实现，但它不生成OF标志，那么如何判断溢出异常的呢?

实际上是“除法错”异常#DE（类型0）

Linux中，对#DE类型发SIGFPE信号

运行结果为“Floating point exception”，显然CPU检测到了溢出异常

为什么两者结果不同！

Linux中对异常的处理

```
sigjmp_buf  buf;

void  FLPhandler(int  sig)
{
      printf("error  type  is  SIGFPE!\n");
      siglongjmp(buf,1);
}

int  main()
{
      int  a,t;
//    signal(SIGFPE,FLPhandler);

      if(!sigsetjmp(buf,1))  {
            printf("starting\n");
            a=100;
            t=0;
            a=a/t;
      }
      printf("I  am  still  alive.....\n");
      exit(0);
}
```

Linux中对异常的处理

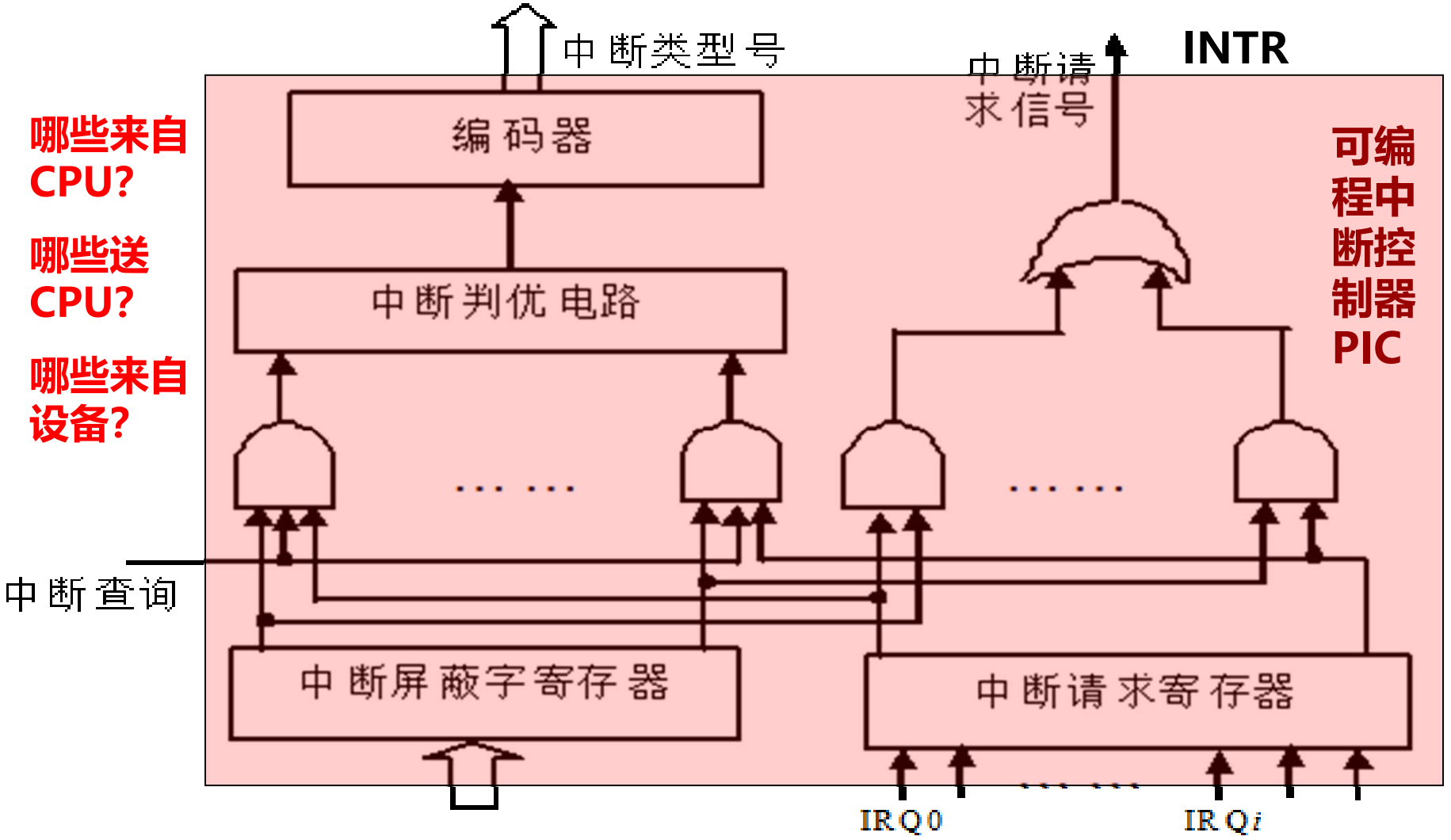
```
//   signal(SIGFPE,FLPhandler);

     if(!sigsetjmp(buf,1))
     {
          printf("starting\n");
          a=100;
          t=0;
          a=a/t;
     }

     printf("I am still alive.....\n");

     exit(0);
}

linuxer@debian:~$ gcc -o sigtest sigtest.c
sigtest.c: In function ‘main’:
sigtest.c:29:5: warning: incompatible implic
‘exit’
      exit(0);
      ^
linuxer@debian:~$ ./sigtest
starting
Floating point exception
linuxer@debian:~$ _
```

Linux中对异常的处理

```
sigjmp_buf buf;

void FLPhandler(int sig)
{
    printf("error type is SIGFPE!\n");
    siglongjmp(buf,1);
}

int main()
{
    int a,t;
    signal(SIGFPE,FLPhandler);

    if(!sigsetjmp(buf,1)) {
        printf("starting\n");
        a=100;
        t=0;
        a=a/t;
    }
    printf("I am still alive.....\n");
    exit(0);
}
```

```
linuxer@debian:~$ ./sigtest
starting
error type is SIGFPE!
I am still alive.....
linuxer@debian:~$ _
```

Linux中对中断的处理



哪些来自CPU?

哪些送CPU?

哪些来自设备?

都有一个编号，如IRQ0、IRQ1、…、IRQi、…，将与 IRQi 关联的中断类型号设定为32+i。

IA-32的中断类型

- **用户自定义类型号为32~255，部分用于可屏蔽中断，部分用于软中断**
- **可屏蔽中断通过CPU的INTR 引脚向CPU发出中断请求**

  中断类型号为32+i

  (i为中断请求号IRQi)
- **软中断指令 INT n 被设定为一种陷阱异常，例如，Linux通过int $0x80指令将128号设定为系统调用，而Windows通过int $0x2e指令将46号设定为系统调用。**

类型号	助记符	含义描述	起因或发生源
0	#DE	除法出错	div 和 idiv 指令
1	#DB	单步跟踪	任何指令和数据引用
2		NMI 中断	不可屏蔽外部中断
3	#BP	断点	int 3 指令
4	#OF	溢出	into 指令
5	#BR	边界检测（BOUND）	bound 指令
6	#UD	无效操作码	不存在的指令操作码
7	#NM	协处理器不存在	浮点或 wait/fwait 指令
8	#DF	双重故障	处理一个异常时发生另一个
9	#MF	协处理器段越界	浮点指令
.....			
19	#XM	SIMD 浮点异常	SIMD 浮点指令
20-31		保留	
32-255		可屏蔽中断和软中断	INTR 中断或 INT *n* 指令

Linux中对中断的处理

- PIC需对所有外设来的 IRQ请求**按优先级排队**，若至少有一个IRQ线有请求且未被屏蔽，则 PIC向 CPU的 INTR引脚**发中断请求**。
- CPU每执行完一条指令都会查询 INTR，若发现有中断请求，则进入**中断响应过程（关中断、保护断点和现场、发中断查询信号）**，**调出中断服务程序**执行。

所有中断服务程序的结构类似，都划分为以下三个阶段。

① **准备阶段**：在内核栈中保存各通用寄存器的内容（称为现场信息）以及所请求 IRQi 的值等，并给PIC回送应答信息，允许其发送新的中断请求信号。

② **处理阶段**：执行 IRQi 对应的中断服务例程 ISR （Interrupt Server Routine）。中断类型号为**32+i**

③ **恢复阶段**：恢复保存在内核栈中的各个寄存器的内容，切换到用户态并返回到当前进程的逻辑控制流的断点处继续执行。

IA-32/Linux的系统调用

- 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

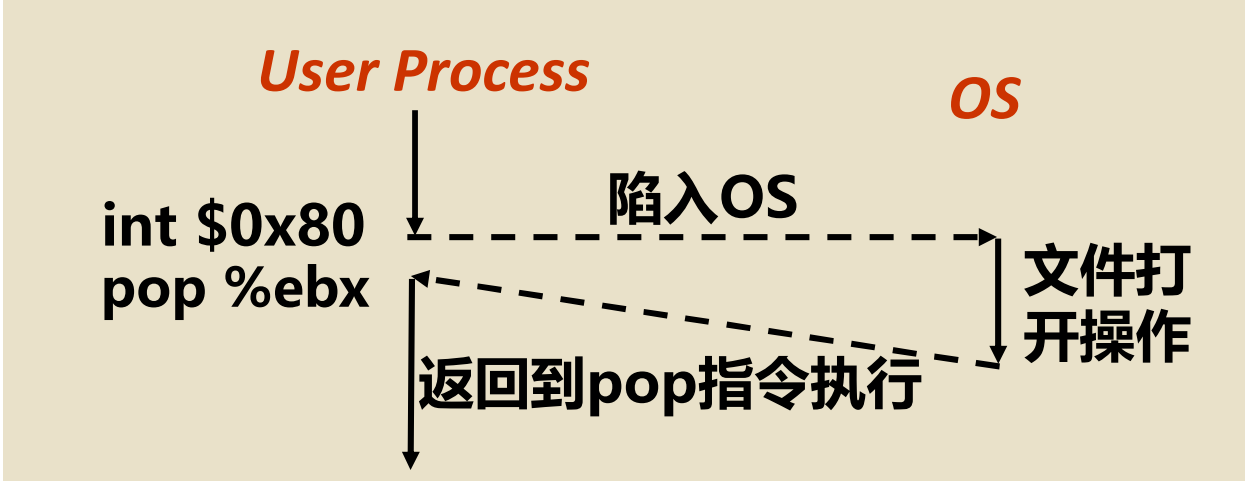
调用号	名称	类别	含义	调用号	名称	类别	含义
1	exit	进程控制	终止进程	12	chdir	文件系统	改变当前工作目录
2	fork	进程控制	创建一个新进程	13	time	系统控制	取得系统时间
3	read	文件操作	读文件	19	lseek	文件系统	移动文件指针
4	write	文件操作	写文件	20	getpid	进程控制	获取进程号
5	open	文件操作	打开文件	37	kill	进程通信	向进程或进程组发信号
6	close	文件操作	关闭文件	45	brk	内存管理	修改虚拟空间中的堆指针 brk
7	waitpid	进程控制	等待子进程终止	90	mmap	内存管理	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

Trap举例: Opening File

- 用户程序中调用函数 open(filename, options)
- open函数执行陷阱指令（即系统调用指令“int”）

这种“地雷”一定“爆炸”

```
0804d070 <__libc_open>:
 . . .
 804d082:      cd 80          int    $0x80
 804d084:      5b             pop    %ebx
 . . .
```



通过执行“int $0x80”指令，调出OS完成一个具体的“服务”（称为系统调用）

Open系统调用（system call）：OS must find or create file, get it ready for reading or writing，Returns integer file descriptor

IA-32/Linux的系统调用

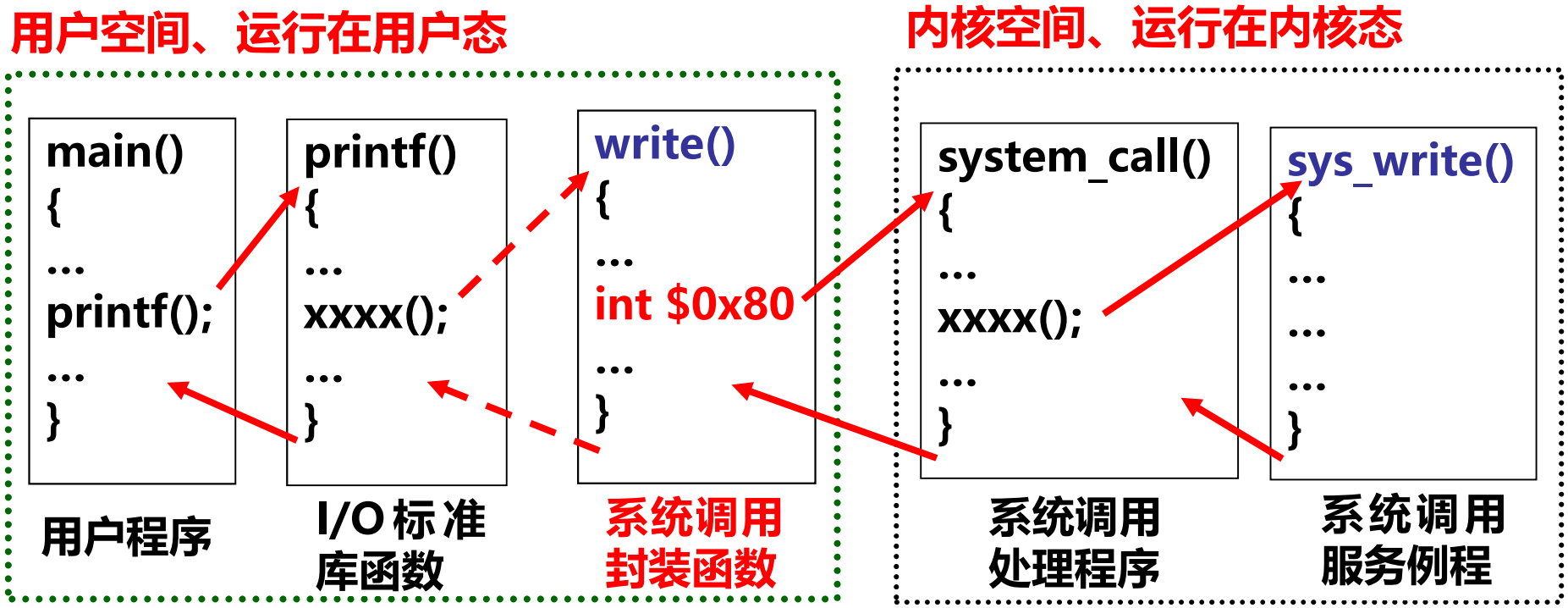
- 通常，系统调用被封装成用户程序能直接调用的函数，如exit()、read()和open()，这些是标准C库中系统调用对应的封装函数 。
- Linux中系统调用所用参数通过寄存器传递，传递参数的寄存器顺序依次为：EAX（调用号）、EBX、ECX、EDX、ESI、EDI和EBP，除调用号以外，最多6个参数。
- 封装函数对应的机器级代码有一个统一的结构：
  - 总是若干条传送指令后跟一条陷阱指令。传送指令用来传递系统调用的参数，陷阱指令（如int $0x80）用来陷入内核进行处理。
- 例如，若用户程序调用系统调用write(1，“hello, world!\n”,14)，将字符串“hello,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

IA-32/Linux的系统调用

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

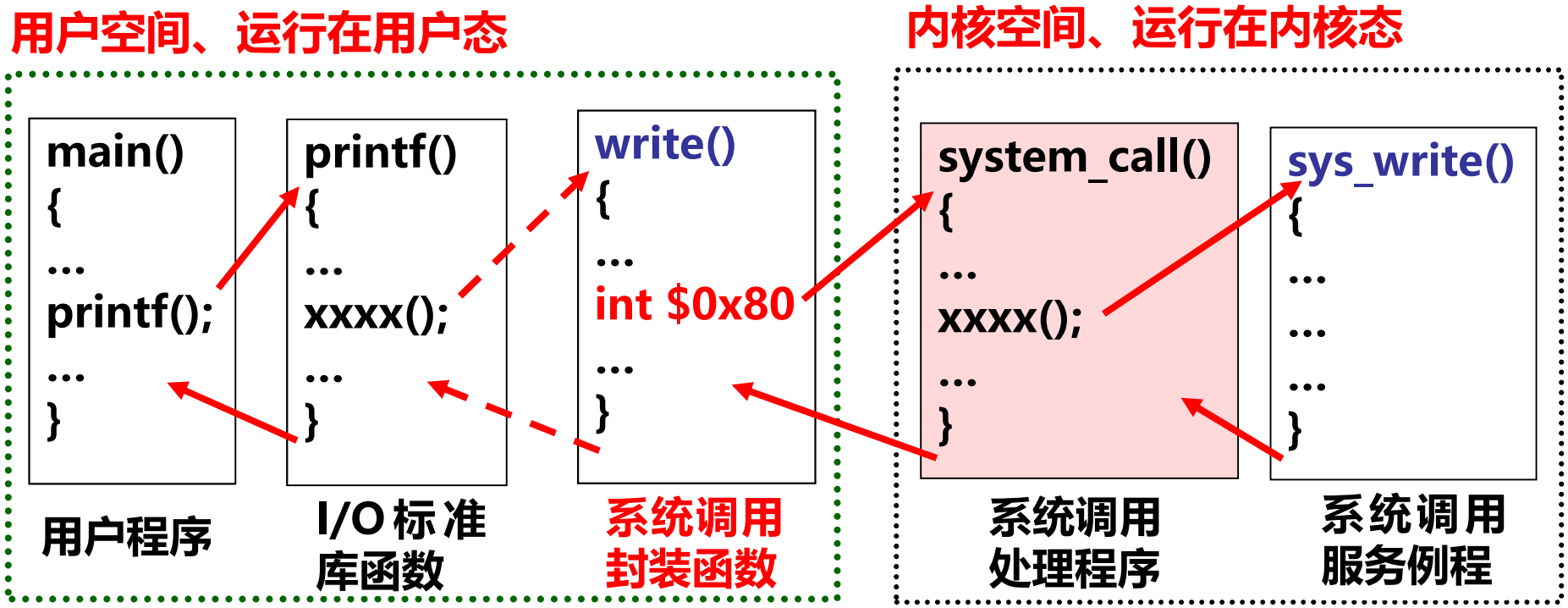
Linux系统中printf()函数的执行过程



- 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?

BACK

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。

SKIP

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。

Linux中中断描述符表的初始化

CPU负责对异常和中断的检测与响应，而操作系统则负责初始化 IDT 以及编制好异常处理程序或中断服务程序。Linux运用提供的三种门描述符格式，构造了以下5种类型的门描述符。

(1) 中断门：DPL=0，TYPE=1110B。激活所有中断

(2) 系统门：DPL=3，TYPE=1111B。激活4、5和128三个陷阱异常，分别对应指令into、bound和int $0x80三条指令。因DPL为3，CPL≤DPL，故在用户态下可使用这三条指令

(3) 系统中断门：DPL=3，TYPE=1110B。激活3号中断（即调试断点），对应指令int 3。因DPL为3，CPL≤DPL，故用户态下可使用int 3指令。

(4) 陷阱门：DPL=0，TYPE=1111B。激活所有内部异常，并阻止用户程序使用INT n（n≠128或3）指令模拟非法异常来陷入内核态运行。

(5) 任务门：DPL=0，TYPE=0101B。激活8号中断（双重故障）。

Linux内核在启用异常和中断机制之前，先设置好 IDT 的每个表项，并把 IDT 首址存入 IDTR。系统初始化时，Linux完成对 GDT、GDTR、IDT 和 IDTR 等的设置，以后一旦发生异常或中断，CPU就可通过异常和中断响应机制调出异常或中断处理程序执行。

BACK

回顾：IA-32/Linux中的分段机制

- 为使能移植到绝大多数流行处理器平台， Linux简化了分段机制
- RISC对分段支持非常有限，因此Linux仅使用IA-32的分页机制，而对于分段，则通过在初始化时将所有段描述符的基址设为0来简化
- 若把运行在用户态的所有Linux进程使用的代码段和数据段分别称为**用户代码段和用户数据段**；把运行在内核态的所有Linux进程使用的代码段和数据段分别称为**内核代码段和内核数据段**，则Linux初始化时，将上述4个段的段描述符中各字段设置成下表中的信息：

段	基地址	G	限界	S	TYPE	DPL	D	P
用户代码段	0x0000 0000	1	0xFFFFF	1	10	3	1	1
用户数据段	0x0000 0000	1	0xFFFFF	1	2	3	1	1
内核代码段	0x0000 0000	1	0xFFFFF	1	10	0	1	1
内核数据段	0x0000 0000	1	0xFFFFF	1	2	0	1	1

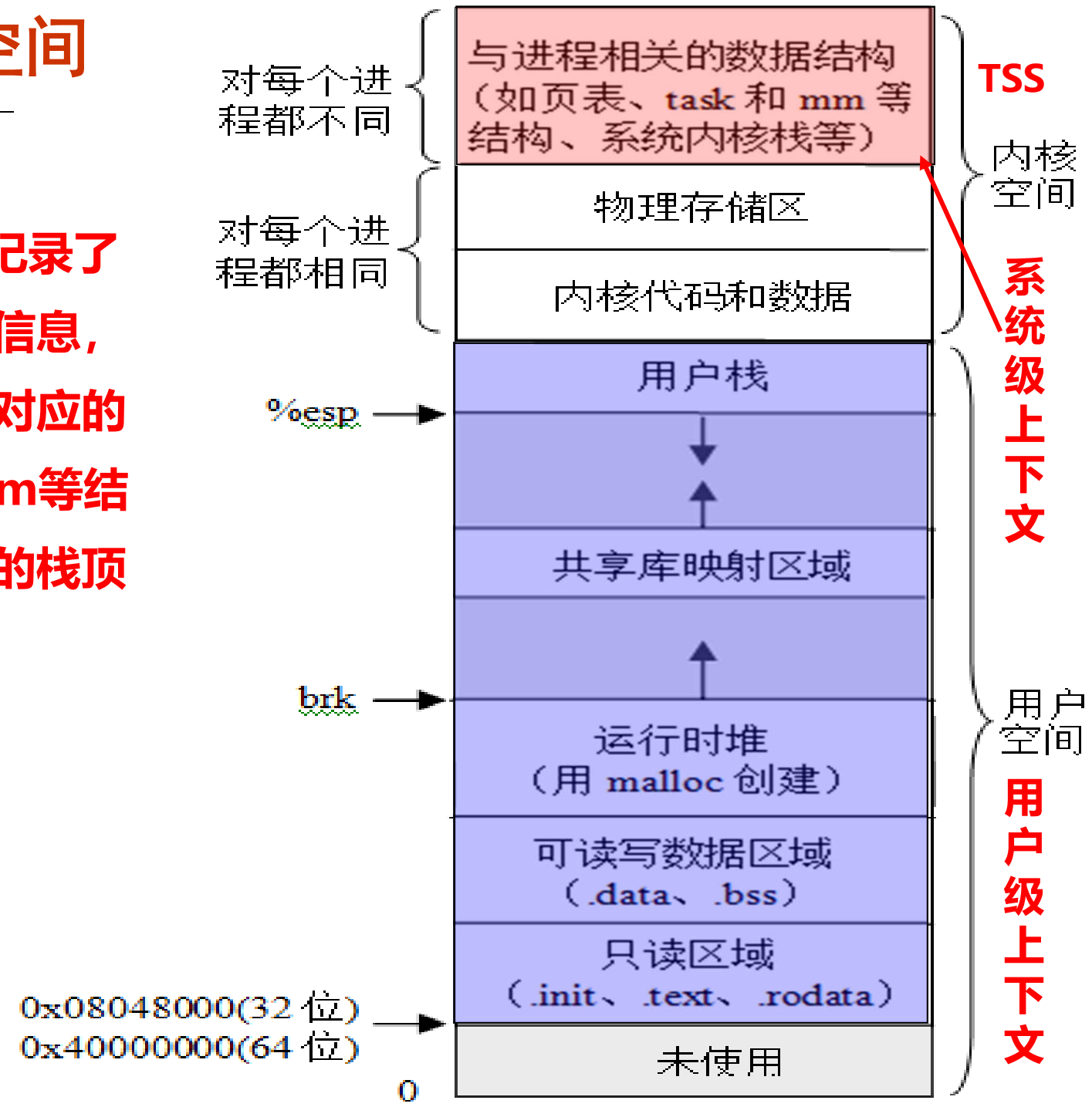
初始化时，上述4个段描述符被存放在GDT中

BACK

进程的地址空间

内核中的TSS段记录了每个进程的状态信息，例如，每个进程对应的页表、task和mm等结构信息、内核栈的栈顶指针SS:ESP 等

BACK



IA-32中异常和中断响应过程

(1) 确定中断类型号 i，从 IDTR 指向的 IDT 中取出第 i 个表项 IDTi。

(2) 根据 IDTi 中段选择符，从 GDTR 指向的 GDT 中取出相应段描述符，得到对应异常或中断处理程序所在段的 DPL、基地址等信息。Linux下中断门和陷阱门对应的即为内核代码段，所以DPL为0，基地址为0。

(3) 若CPL<DPL或编程异常 IDTi 的 DPL<CPL，则发生13号异常。Linux下，前者不会发生。后者用于防止恶意程序模拟 INT n 陷入内核进行破坏性操作。

(4) 若CPL≠DPL，则从用户态换至内核态，以使用内核栈。切换栈的步骤：

① 读 TR 寄存器，以访问正在运行的用户进程的 TSS段；

② 将 TSS段中保存的内核栈的段选择符和栈指针分别装入寄存器 SS 和 ESP，然后在内核栈中保存原来用户栈的 SS 和 ESP。

(5) 若是故障，则将发生故障的指令的逻辑地址写入 CS 和 EIP，以使处理后回到故障指令执行。其他情况下，CS 和 EIP 不变，使处理后回到下条指令执行。

(6) 在当前栈中保存 EFLAGS、CS 和 EIP 寄存器的内容（断点和程序状态）。

(7) 若异常产生了一个硬件出错码，则将其保存在内核栈中。

(8) 将IDTi中的段选择符装入CS，IDTi中的偏移地址装入EIP，它们是异常处理程序或中断服务程序第一条指令的逻辑地址（Linux中段基址=0）。

BACK

Linux中的中断门、陷阱门和任务门 BACK