

# 第七章 异常控制流

**CPU控制流的概念**

**进程上下文切换**

**异常和中断的基本概念**

**异常和中断的响应和处理**

# 异常控制流

---

- **主要教学目标**

- 使学生了解程序执行过程中正常的控制流和异常控制流的区别
- 了解在较低层次上如何实现异常控制流
- 初步理解硬件如何和操作系统协调工作，从而为将来理解和掌握操作系统核心内容打下良好基础。

- **主要教学内容**

- CPU控制流、异常控制流
- 进程和进程上下文切换
- 异常和中断的基本概念
- 异常和中断的响应和处理

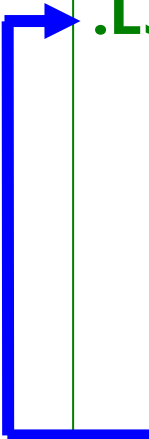
# 异常控制流

---

- 分以下两个部分介绍
  - **第一讲：进程与进程的上下文切换**
    - CPU的控制流、异常控制流
    - 程序和进程、引入进程的好处
    - 逻辑控制流和物理控制流
    - 进程与进程的上下文切换
    - 程序的加载和运行
  - **第二讲：异常和中断**
    - 异常和中断的基本概念
    - 异常和中断的响应、处理
    - IA-32/Linux下的异常/中断机制

# 回顾：程序的机器级表示与执行

```
int sum(int a[ ], unsigned len)
{
    int i, sum = 0;
    for (i = 0; i <= len-1; i++)
        sum += a[i];
    return sum;
}
```



```
sum:
    ...
.L3:
    ...
    movl -4(%ebp), %eax
    movl 12(%ebp), %edx
    subl $1, %edx
    cmpl %edx, %eax
    jbe .L3
    ...
```

程序的正常执行顺序有哪两种？

(1) 按顺序取下一条指令执行

(2) 通过CALL/RET/Jcc/JMP等指令跳转到转移目标地址处执行

CPU所执行的**指令的地址序列**称为**CPU的控制流**，通过上述两种方式得到的控制流为**正常控制流**。

# 重定位后

程序始终按正常  
控制流执行吗？

08048380 <main>:

```
8048380: 55          push %ebp
8048381: 89 e5       mov  %esp,%ebp
8048383: 83 e4 f0    and  $0xffffffff,%esp
8048386: e8 09 00 00 00 call 8048394 <swap>
804838b: b8 00 00 00 00 mov  $0x0,%eax
```

08048394 <swap>:

```
8048394: 55          push %ebp
8048395: 89 e5       mov  %esp,%ebp
8048397: 83 ec 10    sub  $0x10,%esp
804839a: c7 05 00 97 04 08 24 mov  $0x8049624,0x8049700
80483a1: 96 04 08
80483a4: a1 28 96 04 08 mov  0x8049628,%eax
80483a9: 8b 00       mov  (%eax),%eax
80483ab: 89 45 fc    mov  %eax,-0x4(%ebp)
80483ae: a1 28 96 04 08 mov  0x8049628,%eax
80483b3: 8b 15 00 97 04 08 mov  0x8049700,%edx
80493b9: 8b 12       mov  (%edx),%edx
80493bb: 89 10       mov  %edx,(%eax)
80493bd: a1 00 97 04 08 mov  0x8049700,%eax
80493c2: 8b 55 fc    mov  -0x4(%ebp),%edx
80493c5: 89 10       mov  %edx,(%eax)
80493c7: c9          leave
80493c8: c3          ret
```

假定每个函数  
要求4字节边界  
对齐,故填充两  
条nop指令

R[eip]=0x804838b

- 1) R[esp] ← R[esp]-4
- 2) M[R[esp]] ← R[eip]
- 3) R[eip] ← R[eip]+0x9

[BACK](#)

# 异常控制流

- CPU会因为遇到**内部异常**或**外部中断**等原因而打断程序的正常控制流，转去执行操作系统提供的针对这些特殊事件的处理程序。
- 由于某些特殊情况**引起用户程序的正常执行被打断**所形成的意外控制流称为**异常控制流**（Exceptional Control of Flow, ECF）。
- 异常控制流的形成原因：
  - **内部异常**（缺页、越权、越级、整除0、溢出等）
  - **外部中断**（Ctrl-C、打印缺纸、DMA结束等）
  - **进程的上下文切换**（发生在操作系统层）
  - **一个进程直接发送信号给另一个进程**（发生在应用软件层）

} 发生在  
硬件层

本章主要介绍发生在OS层和硬件层的异常控制流

# “程序” 和 “进程”

---

**程序 (program)** 指按某种方式组合形成的代码和数据集合，代码即是机器指令序列，因而程序是一种**静态**概念。

**进程 (process)** 指程序的一次运行过程。更确切说，进程是具有独立功能的一个程序关于某个数据集合的一次运行活动，因而进程具有**动态**含义。同一个程序处理不同的数据就是不同的进程

- 进程是OS对CPU执行的程序的运行过程的一种抽象。进程有**自己的生命周期**，它由于任务的启动而创建，随着任务的完成（或终止）而消亡，它所占用的资源也随着进程的终止而释放。
- 一个可执行目标文件（即程序）可被加载执行多次，也即，一个程序可能对应多个不同的进程。
  - 例如，用word程序编辑一个文档时，相应的用户进程就是winword.exe，如果多次启动同一个word程序，就得到多个winword.exe进程，**处理不同的数据**。

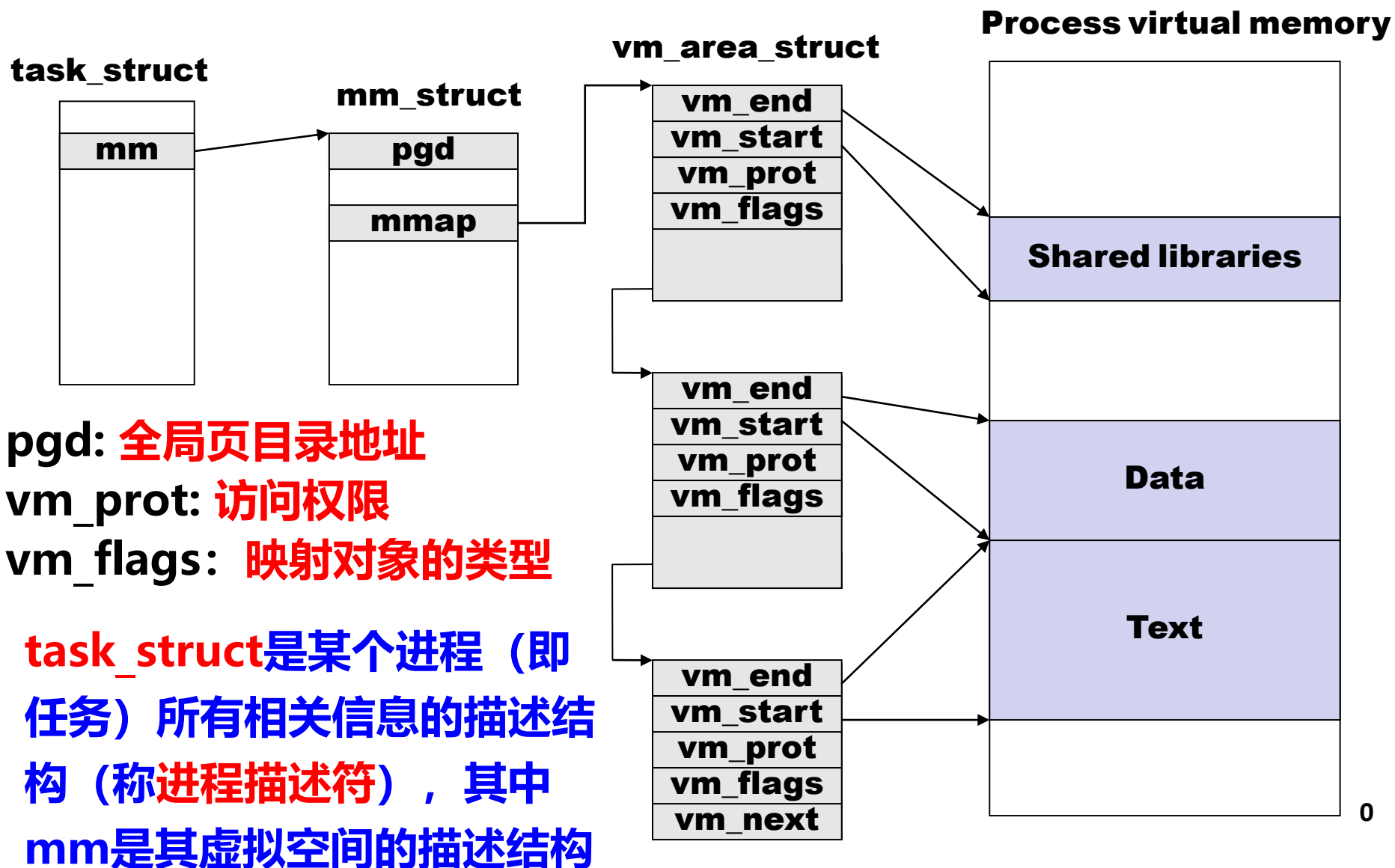
# 进程的概念

---

- 操作系统（管理任务）以外的都属于“用户”的任务。
- 计算机处理的所有“用户”的任务由进程完成。
- 为强调进程完成的是用户的任务，通常将进程称为用户进程。
- 计算机系统中的任务通常就是指进程。例如，
  - Linux内核中通常把进程称为任务，每个进程主要通过一个称为进程描述符（process descriptor）的结构来描述，其结构类型定义为task\_struct，包含了一个进程的所有信息。
  - 所有进程通过一个双向循环链表实现的任务列表（task list）来描述，任务列表中每个元素是一个进程描述符。
  - IA-32中的任务状态段（TSS）、任务门（task gate）等概念中所称的任务，实际上也是指进程。



# 回顾：Linux将虚存空间组织成“区域”的集合



# 引入“进程”的好处

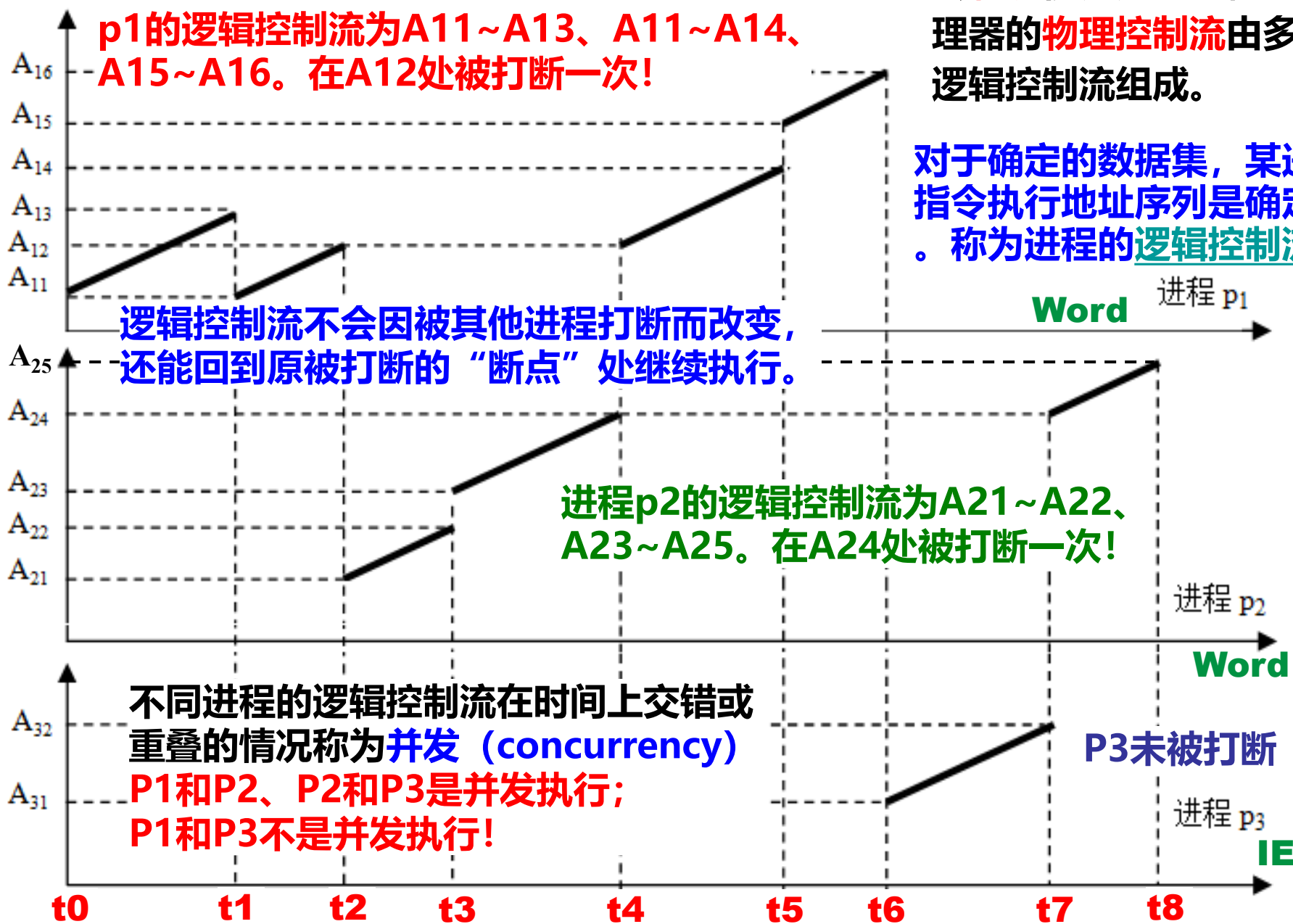
---

- “进程”的引入为应用程序提供了以下两方面的抽象：
  - 一个**独立的逻辑控制流**
    - 每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中**独占使用处理器**
  - 一个**私有的虚拟地址空间**
    - 每个进程拥有一个私有的虚拟地址空间，使得程序员以为自己的程序在执行过程中**独占使用存储器**
- “进程”的引入简化了程序员的编程以及语言处理系统的处理，即**简化了编程、编译、链接、共享和加载等整个过程。**

# 逻辑控制流

对于单处理器系统，进程会轮流使用处理器，即处理器的物理控制流由多个逻辑控制流组成。

对于确定的数据集，某进程指令执行地址序列是确定的。称为进程的逻辑控制流。



# “进程” 与 “上下文切换”

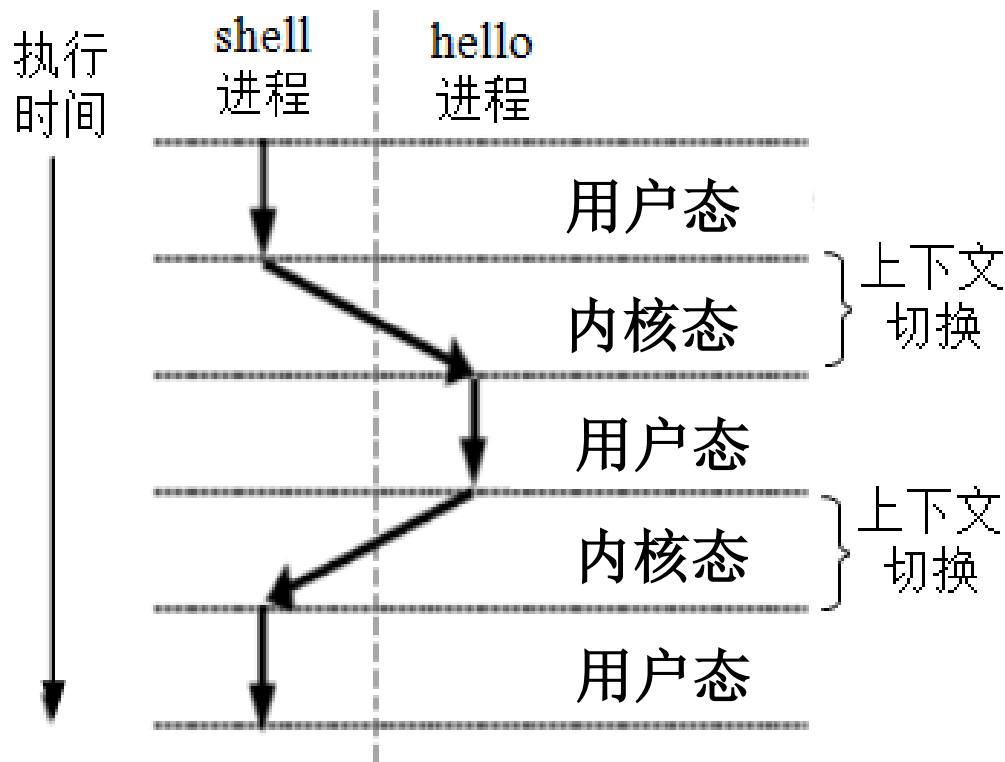
OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为**进程的上下文切换 (context switching)**

```
$. /hello  
hello, world  
$
```

“\$” 是shell命令行提示符，说明正在运行shell进程。

在一个进程的生命周期中，可能会有其他不同进程在处理器上交替运行！

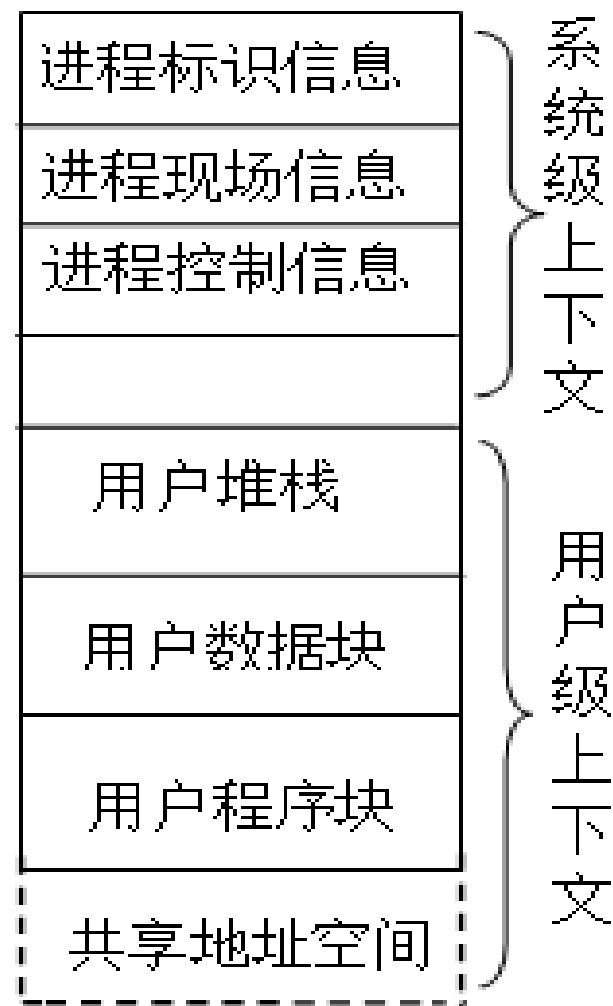
感觉到的运行时间比真实执行时间要长！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。

# “进程” 的 “上下文”

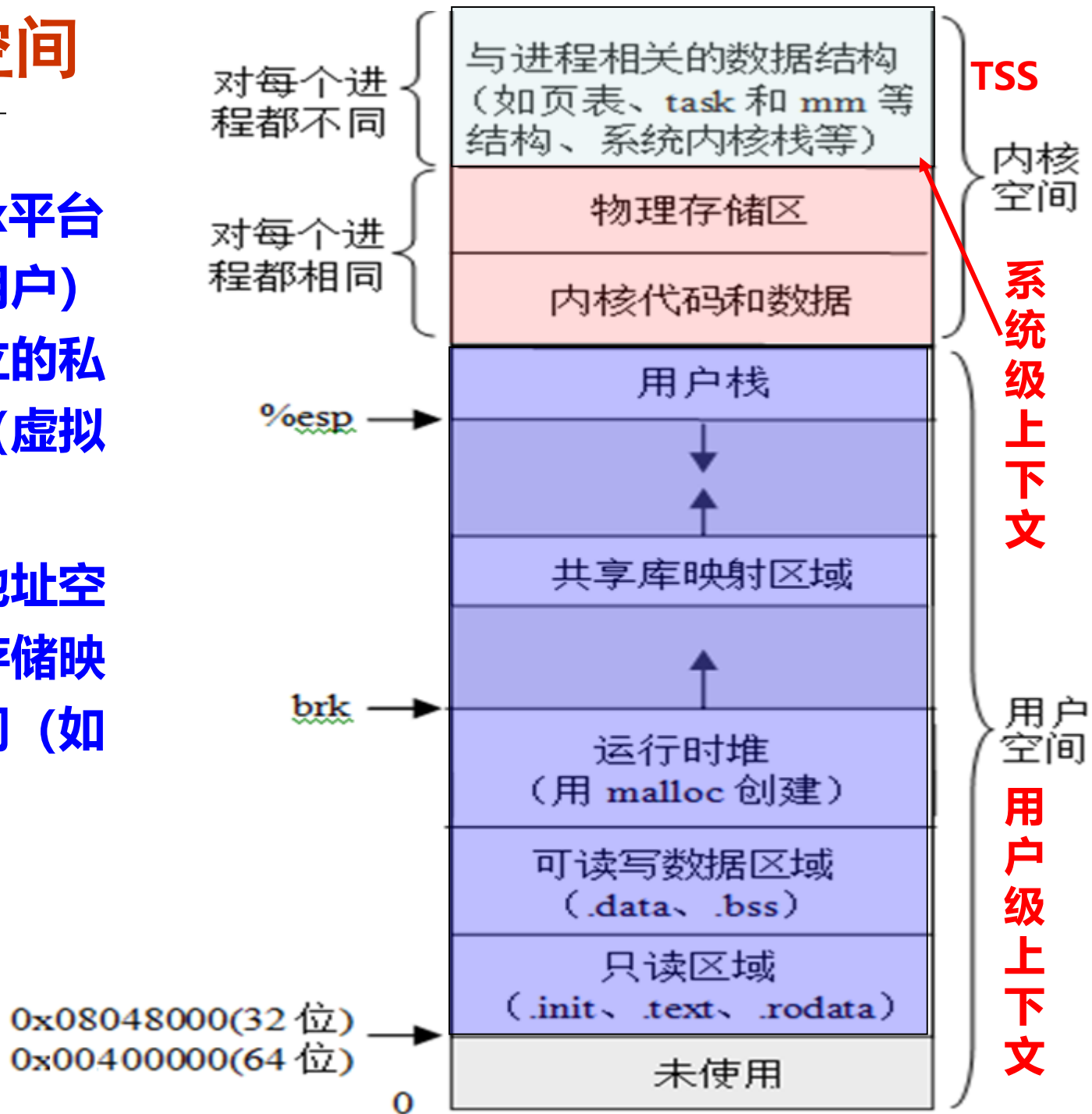
- 进程的物理实体（代码和数据等）和支持进程运行的环境合称为**进程的上下文**。
- 由进程的**程序块**、**数据块**、运行时的**堆**和**用户栈**（两者通称为**用户堆栈**）等组成的**用户空间**信息被称为**用户级上下文**；
- 由**进程标识信息**、**进程现场信息**、**进程控制信息**和**系统内核栈**等组成的**内核空间**信息被称为**系统级上下文**；
- 处理器中各寄存器的内容被称为**寄存器上下文**（也称**硬件上下文**），即进程的**现场信息**。
- 在进行进程上下文切换时，操作系统把换下进程的寄存器上下文保存到系统级上下文中的现场信息位置。
- 用户级上下文地址空间和系统级上下文地址空间一起构成了一个**进程的整个存储器映像**



进程的存储器映像

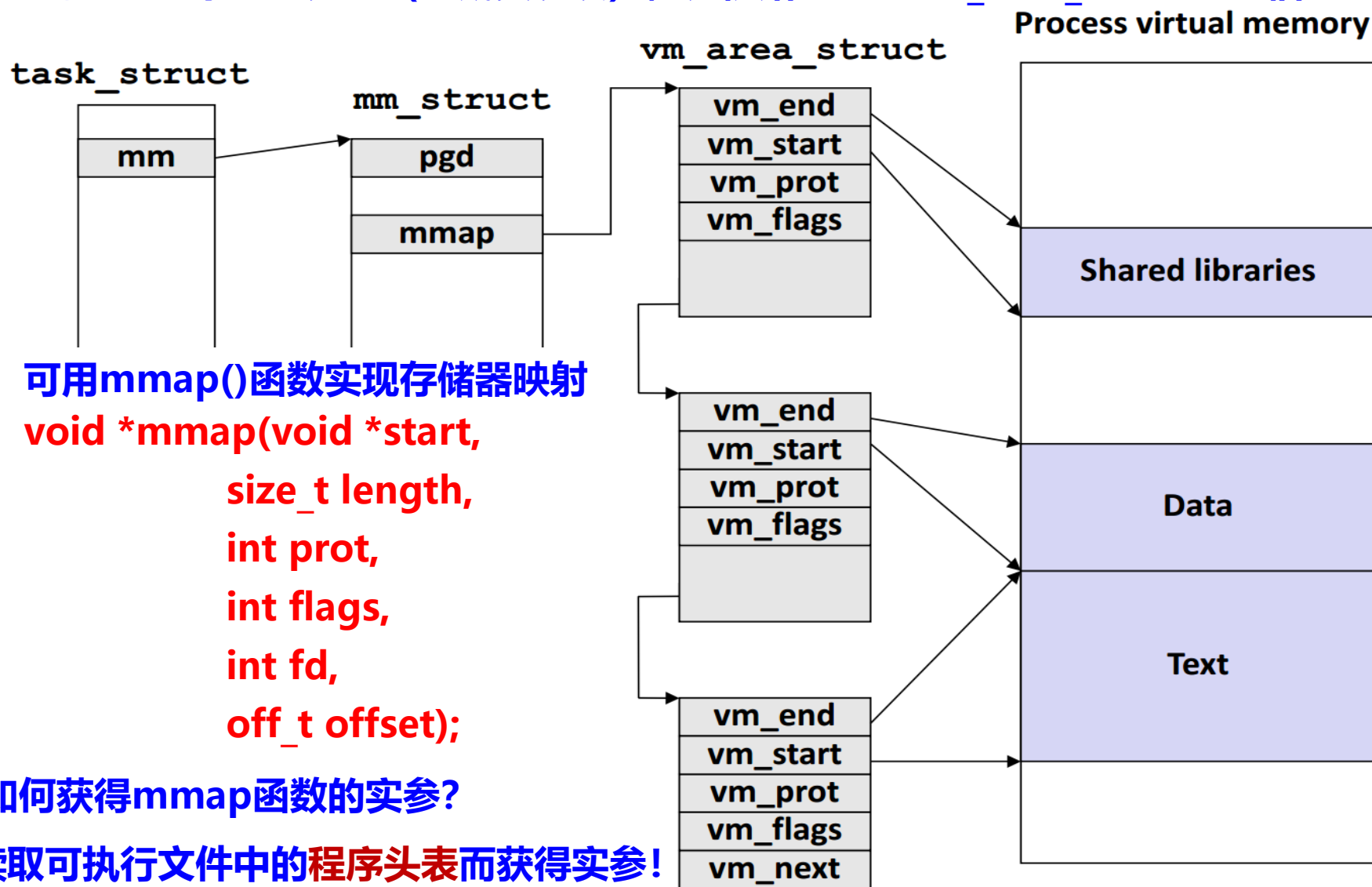
# 进程的地址空间

- IA-32/Linux平台下, 每个 (用户) 进程具有独立的私有地址空间 (虚拟地址空间)
- 每个进程的地址空间划分 (即存储映像) 布局相同 (如右图)



# 进程的存储器映射

**存储器映射 (memory mapping)** 是指将进程虚拟地址空间中的一个区域与硬盘上的一个对象建立关联 (生成页表项)，并初始化一个 `vm_area_struct` 结构信息



## 回顾：存储管理全局图

## task\_struct 进程控制块

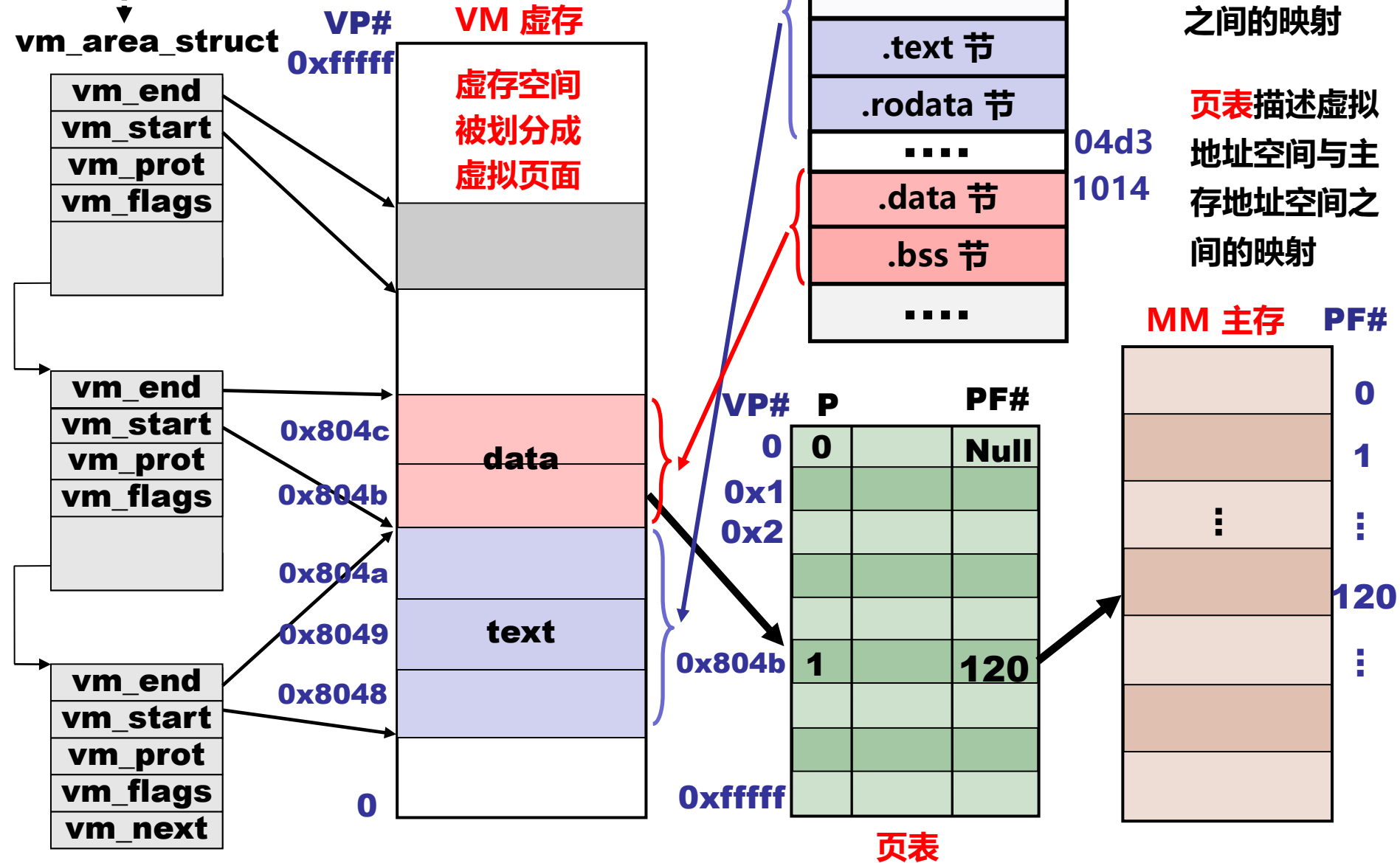
页大小: 4KB

## 可执行目标文件

## 程序头表描述可执行文件与虚拟地址空间之间的映射

## 页表描述虚拟地址空间与主存地址空间之间的映射

MM 主存 PF#





# 进程的存储器映射

可用mmap()函数实现存储器映射

**void \*mmap(void \*start, size\_t length, int prot, int flags, int fd, off\_t offset);**

**功能：**将指定文件fd中偏移量offset开始的长度为length个字节的一块信息映射到虚拟空间中起始地址为start、长度为length个字节的一块区域，得到vm\_area\_struct结构的信息，并生成相应页表项，建立文件地址和区域之间的映射关系。

**prot**指定该区域内页面的访问权限位，对应vm\_area\_struct结构中的vm\_prot字段

**PROT\_EXE：**区域内页面由指令组成

**PROT\_READ：**区域内页面可读

**PROT\_WRITE：**区域内页面可写

**PROT\_NONE：**区域内页面不能被访问

**flags**指定所映射的对象的类型，对应vm\_area\_struct结构中的vm\_flags字段

**MAP\_PRIVATE：**私有的写时拷贝对象，对应可执行文件中只读代码区域（.init、.text、.rodata）和已初始化数据区域（.data）

**MAP\_SHARED：**共享对象，对应共享库文件中的信息

**MAP\_ANON：**请求0的页，对应内核创建的匿名文件，相应页框用0覆盖并驻留内存

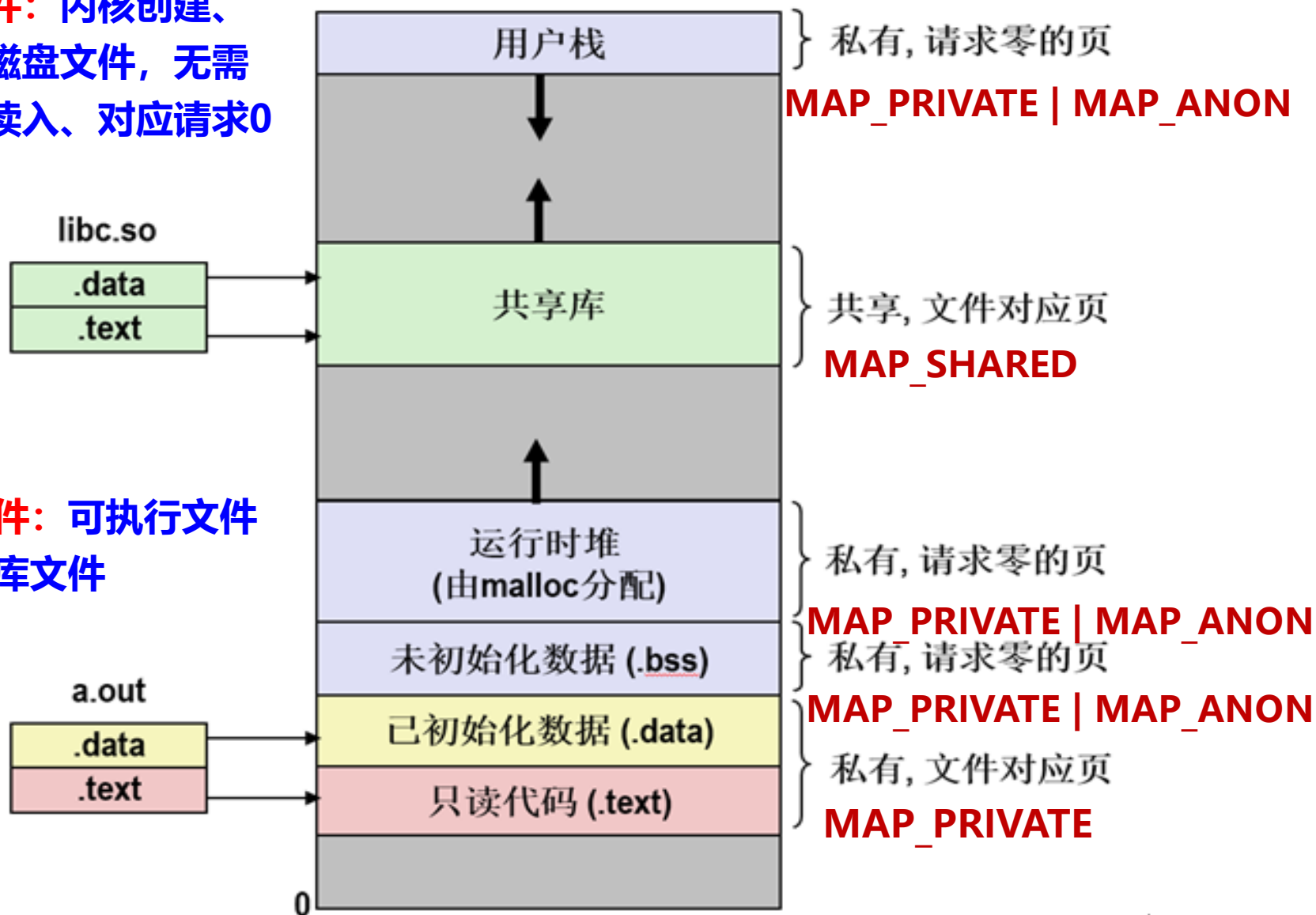
**MAP\_PRIVATE | MAP\_ANON：**未初始化数据（.bss）、堆和用户栈等对应区域

虚页第一次被装入内存后，不管是用普通文件还是匿名文件对其进行初始化，以后都是在主存页框和硬盘中**交换文件（swap file）**间进行调进调出。交换文件由内核管理和维护，称为**交换分区（swap area）**或**交换空间（swap space）**。

# Linux中虚拟地址空间中的区域

**匿名文件：**内核创建、  
无实际磁盘文件，无需  
从磁盘读入、对应请求0  
的页面

**普通文件：**可执行文件  
和共享库文件



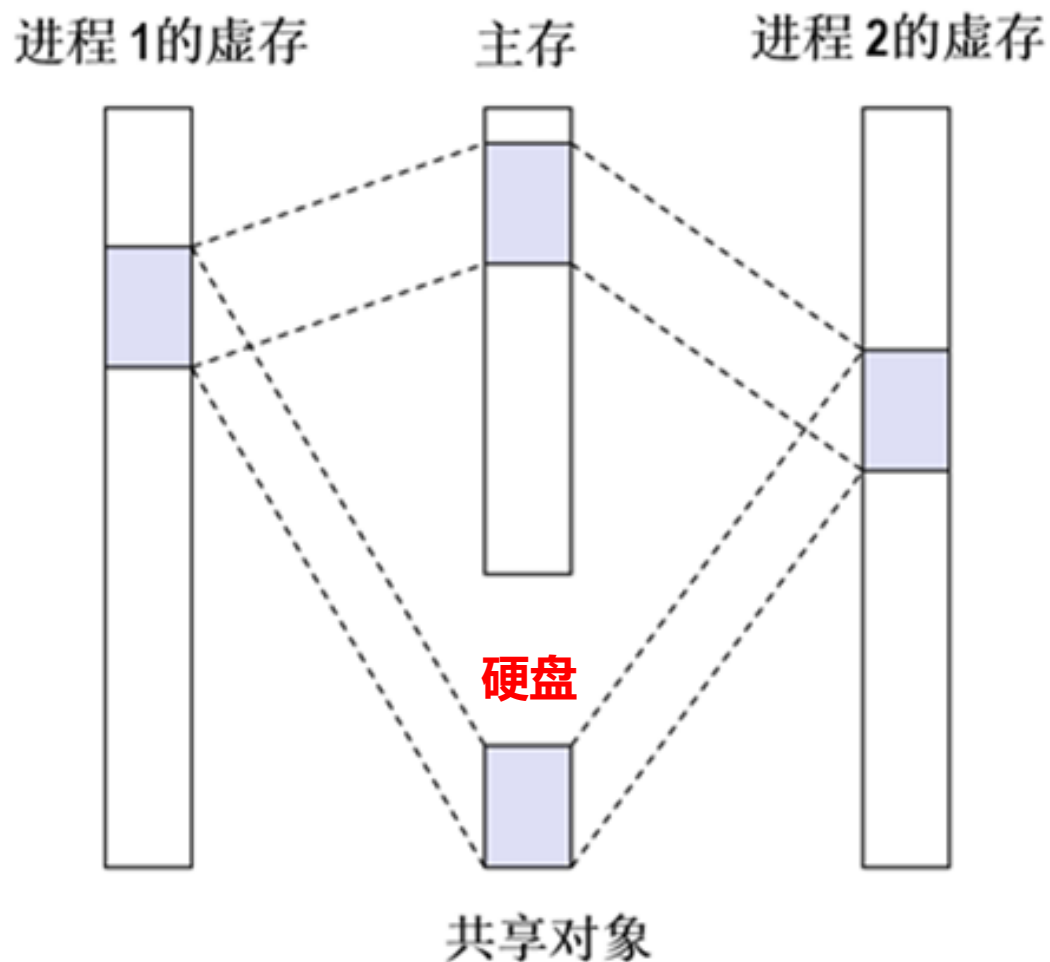
# 共享库文件中的共享对象

**多个进程调用共享库文件中的代码，但共享库代码在内存和硬盘都只需要一个副本**

**进程1运行过程中，内核为共享对象分配若干页框**

**进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上即可**

**一个进程对共享区域进行的写操作结果，对于所有共享同一个共享对象的进程都是可见的，而且结果也会反映在硬盘上对应的共享对象中**



**所分配的页框在主存不一定连续，为简化示意图，这里图中所示页框是连续的**

# 私有的写时拷贝对象

同一个可执行文件对应不同进程时，只读代码区一样，可读可写数据区开始也一样，但属于私有对象

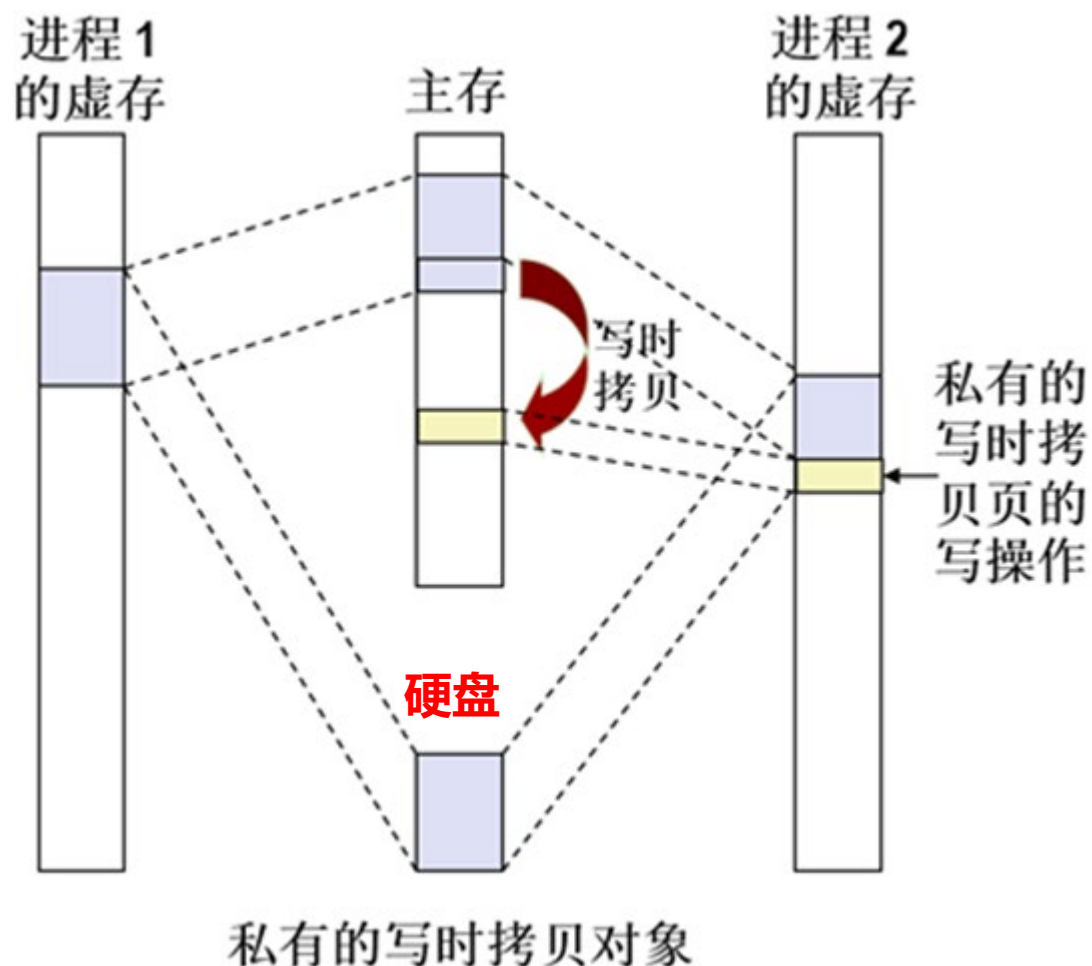
为节省主存，多采用写时拷贝技术

进程1运行过程中，内核为对象分配若干页框，并标记为只读

进程2运行过程中，内核只要将进程2对应区域内页表项中的页框号直接填上，并标记为只读

若两个进程都只是读或执行，则在内存只有一个副本，节省主存；

若进程2进行写操作，则发生访问违例，此时，内核判断异常原因是进程试图写私有的写时拷贝页，就会分配一个新页框，把内容拷贝到新页框，并修改进程2的页表项



所分配的页框在主存不一定连续，为简化示意图，这里图中所示页框是连续的

# 回顾：用户模式和内核模式

---

- 为了使OS能够起到管理程序执行的目的，在一些时候处理器中必须运行**内核代码**
- 为了区分处理器运行的是用户代码还是内核代码，必须有一个状态位来标识，这个状态位称为**模式位**
- **处理器模式**分**用户模式（用户态）**和**内核模式（核心态）**
- 用户模式（也称**目态、用户态**）下，处理器运行用户进程，此时不允许使用特权指令
- 内核模式（有时称**系统模式、管理模式、超级用户模式、管态、内核态、核心态**）下处理器运行内核代码，允许使用**特权指令**，例如：停机指令、开/关中断指令、Cache冲刷指令等。

# 程序的加载和运行

- UNIX/Linux系统中，可通过调用execve()函数来启动加载器。
- execve()函数的功能是在当前进程上下文中加载并运行一个新程序。  
execve()函数的用法如下：

`int execve(char *filename, char *argv[], *envp[]);`

filename是加载并运行的可执行文件名(如./hello)，可带参数列表argv和环境变量列表envp。若错误（如找不到指定文件filename），则返回-1，并将控制权交给调用程序；若函数执行成功，则不返回，最终将控制权传递到可执行目标中的主函数main。

- 主函数main()的原型形式如下：

`int main(int argc, char **argv, char **envp);` 或者：

`int main(int argc, char *argv[], char *envp[]);`

argc指定参数个数，参数列表中第一个总是命令名（可执行文件名）

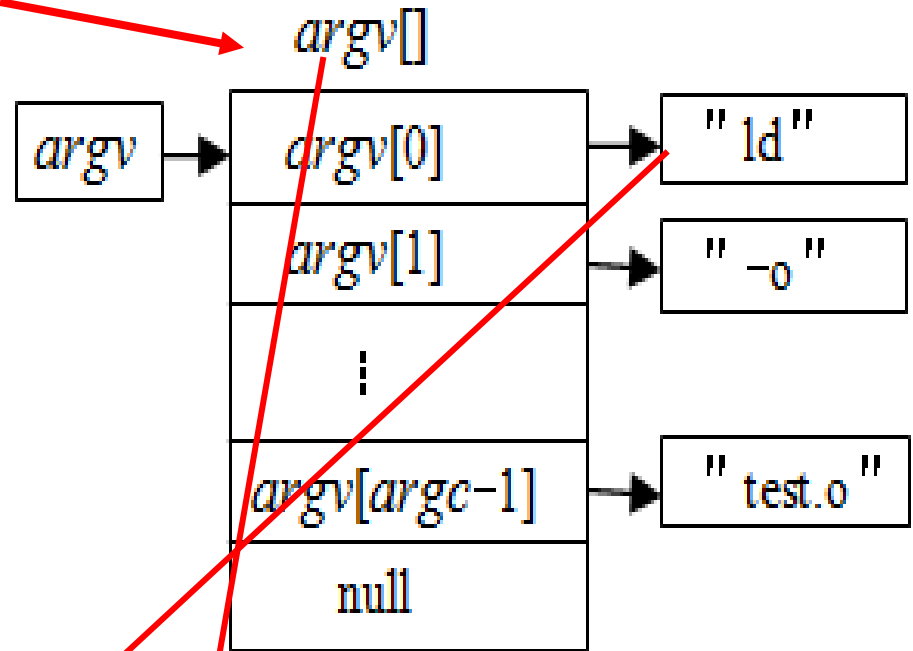
例如：命令行为“ld -o test main.o test.o” 时，argc=5

# 回顾：程序的加载和运行

若在shell命令行提示符下输入以下命令

Unix>**ld -o test main.o test.o**

**ld**是可执行文件名（即命令名），随后是命令的若干参数，**argv**是一个以null结尾的指针数组，**argc=5**



在shell命令行提示符后键入命令并按“enter”键后，便构造**argv**和**envp**，然后调用**execve()**函数来启动加载器，最终转**main()**函数执行

**int execve(char \*filename, char \*argv[], \*envp[]);**

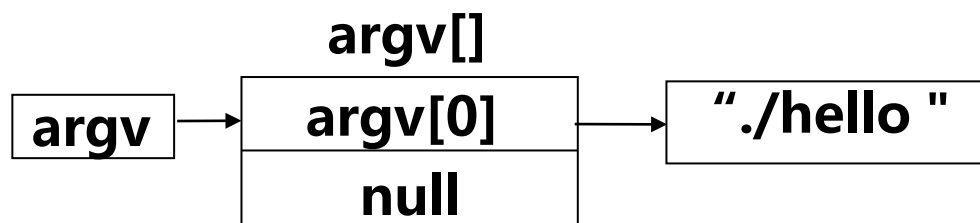
**int main(int argc, char \*argv[], char \*envp[]);**

# 回顾：程序的加载和运行

问题：hello程序的加载和运行过程是怎样的？

Step1: 在shell命令行提示符后输入命令：`$./hello[enter]`

Step2: shell命令行解释器构造argv和envp



Step3: 调用**fork()**函数，创建一个子进程，与父进程shell完全相同（只读/共享），包括只读代码段、可读写数据段、堆以及用户栈等。

Step4: 调用**execve()**函数,在当前进程（新创建的子进程）的上下文中加载并运行hello程序。将hello中的.text节、.data节、.bss节等内容加载到当前进程的虚拟地址空间（仅修改当前进程上下文中关于存储映像的一些数据结构，不从磁盘拷贝代码和数据等内容）

Step5: 调用hello程序的**main()**函数，hello程序开始在一个进程的上下文中运行。  
`int main(int argc, char *argv[], char *envp[]);`



# 回顾：可执行文件的加载

- 通过调用execve系统调用函数来调用加载器
- 加载器 (loader) 根据可执行文件的程序 (段) 头表中的信息, 将可执行文件的代码和数据从磁盘“拷贝”到存储器中 (实际上不会真正拷贝, 仅建立一种映像, 这涉及到许多复杂的过程和一些重要概念, 将在后续课上学习)
- 加载后, 将PC (EIP) 设定指向 Entry point (即符号\_start处), 最终执行main函数, 以启动程序执行。

程序被启动  
如 \$ ./P

调用fork()

以构造的argv和envp  
为参数调用execve()

execve()调用加载器  
进行可执行文件加载,  
并最终转去执行main

\_start: \_\_libc\_init\_first → \_init → atexit → main → \_exit

# ELF文件信息举例

**\$ readelf -h main**

可执行目标文件的ELF头

ELF Header:

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00

Class: ELF32

Data: 2's complement, little endian

Version: 1 (current)

OS/ABI: UNIX - System V

ABI Version: 0

Type: EXEC (Executable file)

Machine: Intel 80386

Version: 0x1

**Entry point address: x8048580**

Start of program headers: 52 (bytes into file)

Start of section headers: 3232 (bytes into file)

Flags: 0x0

Size of this header: 52 (bytes)

Size of program headers: 32 (bytes)

Number of program headers: 8

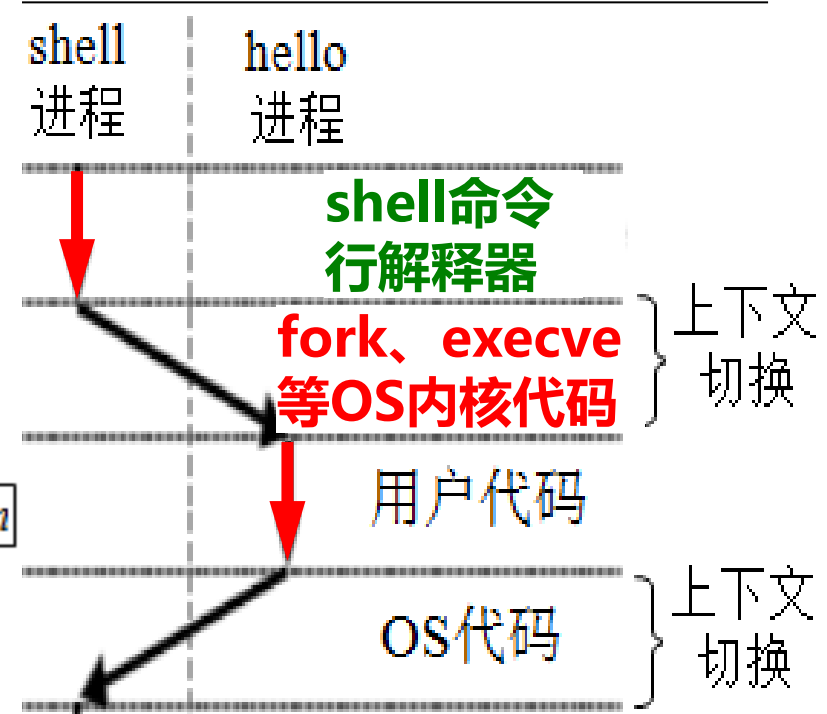
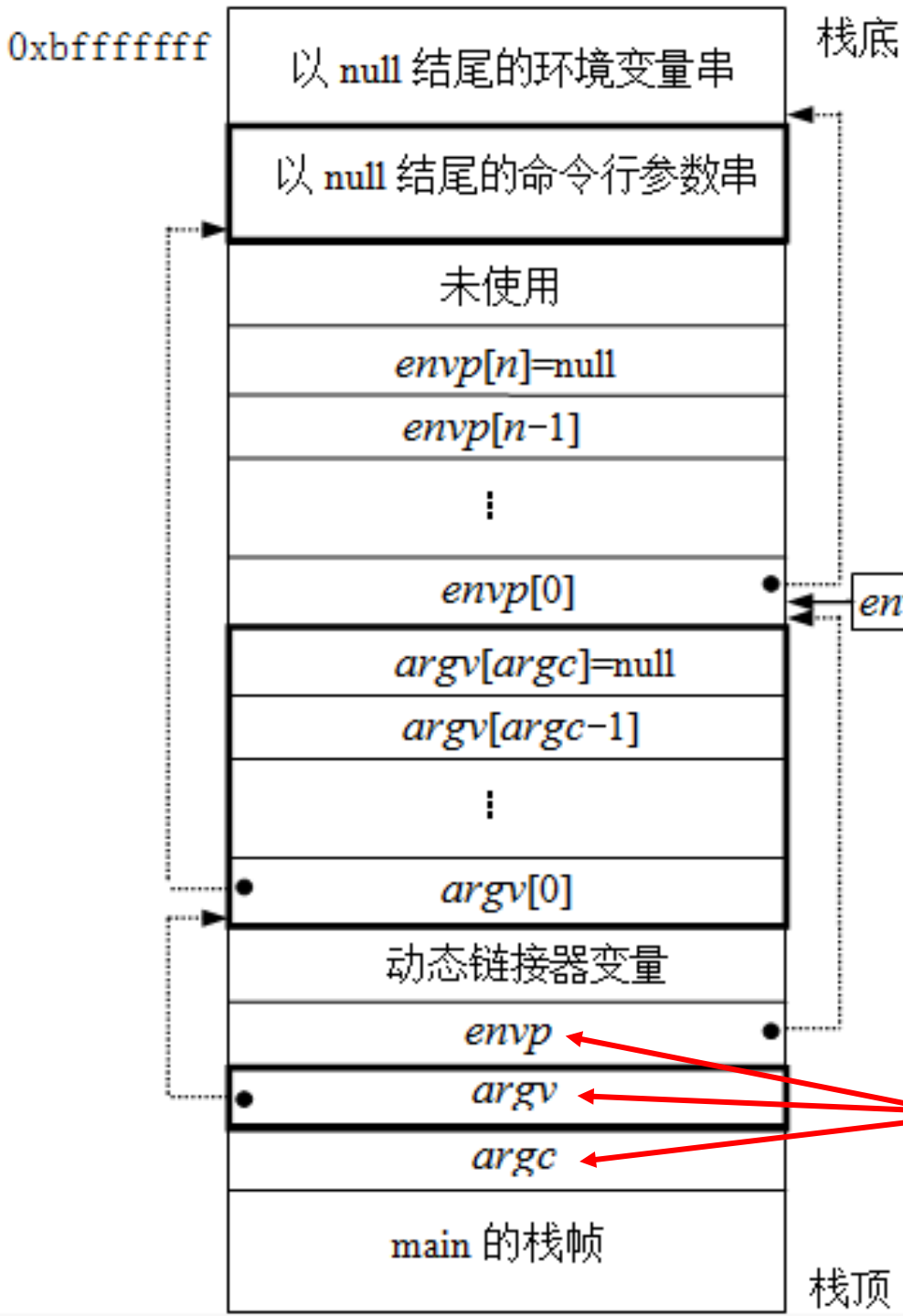
Size of section headers: 40 (bytes)

Number of section headers: 29

Section header string table index: 26

ELF 头
程序头表
.init 节
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.debug 节
.line 节
.strtab 节
节头表

# 程序加载和运行



当IA-32/Linux系统开始执行 `main()` 函数时，在虚拟地址空间的 **用户栈** 中的结构如左图所示

```
int main(int argc,  
char *argv[],  
char *envp[]);
```

# 异常控制流

---

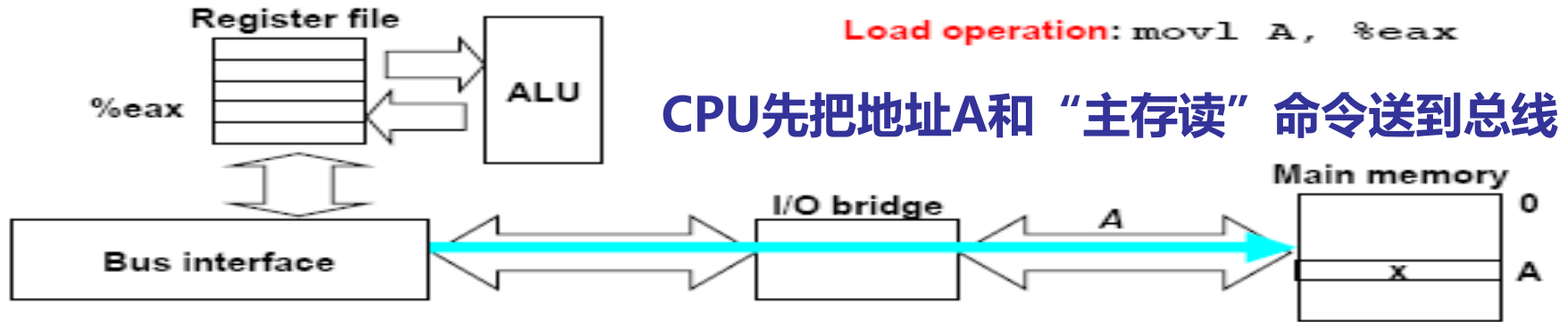
- 分以下两个部分介绍
  - **第一讲：进程与进程的上下文切换**
    - CPU的控制流、异常控制流
    - 程序和进程、引入进程的好处
    - 逻辑控制流和物理控制流
    - 进程与进程的上下文切换
    - 程序的加载和运行
  - **第二讲：异常和中断**
    - 异常和中断的基本概念
    - 异常和中断的响应、处理
    - IA-32/Linux下的异常/中断机制

# 异常和中断

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“中断”
  - CPU中止原来正在执行的程序，转到处理异常情况或特殊事件的程序去执行，结束后再返回到原被中止的程序处（断点）继续执行。
- 程序执行被“中断”的事件（在硬件层面）有两类
  - 内部“异常”：在CPU内部发生的意外事件或特殊事件  
按发生原因分为硬故障中断和程序性中断两类  
**硬故障中断**：如电源掉电、硬件线路故障等  
**程序性中断**：执行某条指令时发生的“例外(Exception)”事件，如溢出、缺页、越界、越权、越级、非法指令、除数为0、堆/栈溢出、访问超时、断点设置、单步、系统调用等
  - 外部“中断”：在CPU外部发生的特殊事件，通过“中断请求”信号向CPU请求处理。如实时钟、控制台、打印机缺纸、外设准备好、采样计时到、DMA传输结束等。

# 回顾：指令“`movl 8(%ebp), %eax`”操作过程

由`8(%ebp)`得到主存地址A的过程较复杂，涉及MMU、TLB、页表等许多重要概念！



- IA-32中，执行“`movl 8(%ebp), %eax`”中取数操作的大致过程如下：
  - 若 $CPL > DPL$ 则越级，否则计算有效地址 $EA = R[ebp] + 0 \times 0 + 8$
  - 通过段寄存器找到段描述符以获得段基址，线性地址 $LA = \text{段基址} + EA$
  - 若“ $LA > \text{段限}$ ”则越界，否则将LA转换为主存地址A
    - 若访问TLB命中则地址转换得到A；否则处理TLB缺失（硬件/OS）
    - 若缺页或越权(R/W不符)则调出OS内核；否则地址转换得到A
    - 根据A先到Cache中找，若命中则取出A在Cache中的副本
    - 若Cache不命中，则再到主存取A所在主存块送对应Cache行

# 异常和中断的处理

- 发生**异常(exception)**和**中断(interrupt)**事件后，系统将进入OS内核态对相应事件进行处理，即改变处理器状态（**用户态→内核态**）



中断或异常处理执行的代码不是一个进程，而是“**内核控制路径**”，它代表异常或中断发生时正在运行的当前进程在内核态执行一个独立的指令序列。内核控制路径比进程更“轻”，其上下文信息比进程上下文信息少得多。而**上下文切换后CPU执行的是另一个用户进程**。

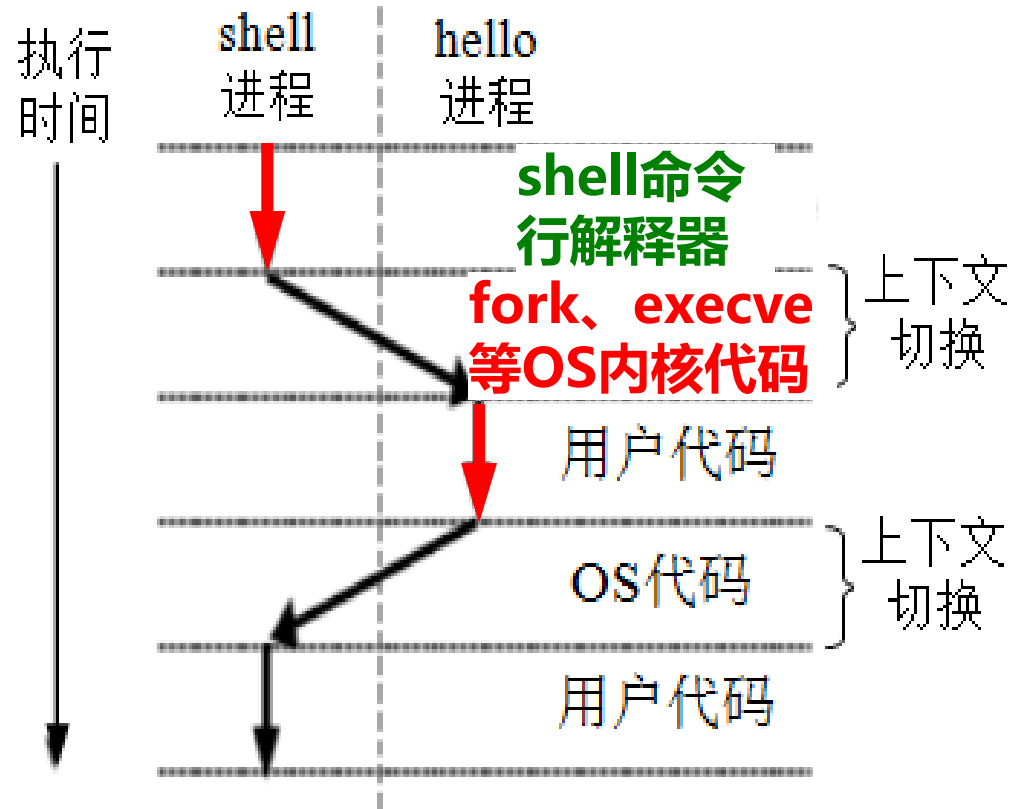
# 回顾：“进程”与“上下文切换”

OS通过处理器调度让处理器轮流执行多个进程。实现不同进程中指令交替执行的机制称为**进程的上下文切换 (context switching)**

```
$. /hello  
hello, world  
$
```

“\$”是shell命令行提示符，说明正在运行shell进程。

上下文切换后，是另一个用户进程！



处理器调度等事件会引起用户进程正常执行被打断，因而形成异常控制流。进程的上下文切换机制很好地解决了这类异常控制流，实现了从一个进程安全切换到另一个进程执行的过程。



# 异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

**故障(fault)**：执行指令引起的异常事件，如溢出、非法指令、缺页、访问越权等。 “断点” 为发生故障指令的地址

**自陷(Trap)**：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。

“断点” 为自陷指令下条指令地址

**终止(Abort)**：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点” 是什么？ 随便！

**思考1**：自陷处理完成后回到哪条指令执行？ 回到下条指令

**思考2**：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法指令、内存保护错等：终止当前进程。

“断点”：异常处理结束后回到原来被“中断”的程序执行时的起始指令。

# 异常举例—页故障

“页故障”事件何时发现？如何发现？

执行每条指令都要**访存**（取指令、取操作数、存结果）

在保护模式下，每次访存都要进行**逻辑地址向物理地址转换**

在地址转换过程中会发现是否发生了“页故障”！

“页故障”事件是软件发现的还是硬件发现的？

逻辑地址向物理地址的转换由硬件（MMU）实现，故“页故障”事件由硬件发现。**所有异常和中断事件都由硬件检测发现！**

- 以下几种情况都会发生“页故障”

- 缺页：页表项有效位为0 ← 可通过读磁盘恢复故障
- 地址越界：地址大于最大界限
- 访问越级或越权（保护违例）： } 不可恢复，称为“段故障（segmentation fault）”
  - 越级：用户进程访问内核数据（CPL=3 / DPL=0）
  - 越权：读写权限不相符（如对只读段进行了写操作）

# 异常举例一页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1 int a[1000];
```

```
2 int x;
```

```
3 main( )
```

```
4 {
```

```
5     a[10]=1;
```

```
6     a[1000]=3;
```

```
7     a[10000]=4;
```

```
8 }
```

正常的控制流为

...、0x8048300、0x8048309、0x8048313、...

可能的异常控制流是什么？

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5 8048300: c7 05 28 90 04 08 01 00 00 00 movl $0x1, 0x8049028
```

```
6 8048309: c7 05 a0 9f 04 08 03 00 00 00 movl $0x3, 0x8049fa0
```

```
7 8048313: c7 05 40 2c 05 08 04 00 00 00 movl $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

(1) 对于上述三条指令的执行，在取指令时是否可能发生页故障？

(2) 在数据访问时分别会发生什么问题？

(3) 哪些问题是可恢复的？哪些问题是不可恢复的？

# 异常举例一页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1  int a[1000];
2  int x;
3  main( )
4  {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8  }
```

**三条指令在读指令时都不会发生缺页，Why?**

它们都位于起始地址为0x08048000（是一个4KB页面的起始位置）的同一个页面，执行这三条指令之前，该页已经调入内存。因为没有其他进程在系统中运行，所以不会因为执行其他进程而使得调入主存的页面被调出到磁盘。因而都不会在取指令时发生页故障。

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5  8048300: c7 05 28 90 04 08 01 00 00 00  movl  $0x1, 0x8049028
6  8048309: c7 05 a0 9f 04 08 03 00 00 00  movl  $0x3, 0x8049fa0
7  8048313: c7 05 40 2c 05 08 04 00 00 00  movl  $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

- (1) 对于上述三条指令的执行，在取指令时是否可能发生页故障？
- (2) 在数据访问时分别会发生什么问题？
- (3) 哪些问题是可恢复的？哪些问题是不可恢复的？

# 异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1  int a[1000];
2  int x;
3  main( )
4  {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8  }
```

**第5行指令数据访问时是否发生页故障，Why?**

**对a[10]（地址0x8049028）的访问是对所在页面（首址为0x08049000）的第一次访问，故不在主存，缺页处理结束后，再回到这条movl指令重新执行，再访问数据就没有问题了。**

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5  8048300: c7 05 28 90 04 08 01 00 00 00  movl  $0x1, 0x8049028
6  8048309: c7 05 a0 9f 04 08 03 00 00 00  movl  $0x3, 0x8049fa0
7  8048313: c7 05 40 2c 05 08 04 00 00 00  movl  $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

- (1) 对于上述三条指令的执行，在取指令时是否可能发生页故障？
- (2) 在数据访问时分别会发生什么问题？
- (3) 哪些问题是可恢复的？哪些问题是不可恢复的？

# 异常举例一页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1  int a[1000];
2  int x;
3  main( )
4  {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8  }
```

**第6行指令数据访问时是否发生页故障，Why?**

**对a[1000]（地址0x8049fa0）的访问是对所在页面（首址为0x08049000）的第2次访问，故在主存，不会发生缺页。但a[1000]实际不存在，只不过编译器未检查数组边界，0x8049fa0处可能是x的地址，故该指令执行结果可能是x被赋值为3**

假设编译、汇编和链接后，第5、6和7行源代码对应的指令序列如下：

```
5  8048300: c7 05 28 90 04 08 01 00 00 00  movl  $0x1, 0x8049028
6  8048309: c7 05 a0 9f 04 08 03 00 00 00  movl  $0x3, 0x8049fa0
7  8048313: c7 05 40 2c 05 08 04 00 00 00  movl  $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

- (1) 对于上述三条指令的执行，在取指令时是否可能发生页故障？
- (2) 在数据访问时分别会发生什么问题？
- (3) 哪些问题是可恢复的？哪些问题是不可恢复的？

# 异常举例—页故障

假设在IA-32/linux系统中一个C语言源程序 P 如下：

```
1  int a[1000];
2  int x;
3  main( )
4  {
5      a[10]=1;
6      a[1000]=3;
7      a[10000]=4;
8  }
```

第7行指令数据访问时是否发生页故障，Why?

地址0x8052c40偏离数组首址0x8049000已达 $4 \times 10000 + 4 = 40004$ 个单元，即偏离了9个页面，很可能超出可读写区范围，故执行该指令时可能会发生保护违例。页故障处理程序发送一个“段错误”信号 (SIGSEGV) 给用户进程，用户进程接受到该信号后就调出一个信号处理程序执行，该信号处理程序根据信号类型，在屏幕上显示“段故障 (segmentation fault)”信息，并终止用户进程。

假设编译、汇编和链接

```
5  8048300: c7 05 28 90 04 08 01 00 00 00  movl  $0x1, 0x8049028
6  8048309: c7 05 a0 9f 04 08 03 00 00 00  movl  $0x3, 0x8049fa0
7  8048313: c7 05 40 2c 05 08 04 00 00 00  movl  $0x4, 0x8052c40
```

已知页大小为4KB，若在运行P对应的进程时，系统中无其他进程在运行，则：

- (1) 对于上述三条指令的执行，在取指令时是否可能发生页故障？
- (2) 在数据访问时分别会发生什么问题？
- (3) 哪些问题是可恢复的？哪些问题是不可恢复的？

# 异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

**故障(fault)**：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。  
“断点” 为发生故障指令的地址

**自陷(Trap)**：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。

“断点” 为自陷指令下条指令地址

**终止(Abort)**：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。  
“断点” 是什么？ 随便！

**思考1**：自陷处理完成后回到哪条指令执行？ 回到下条指令

**思考2**：哪些故障补救后可继续执行，哪些只好终止当前进程？

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

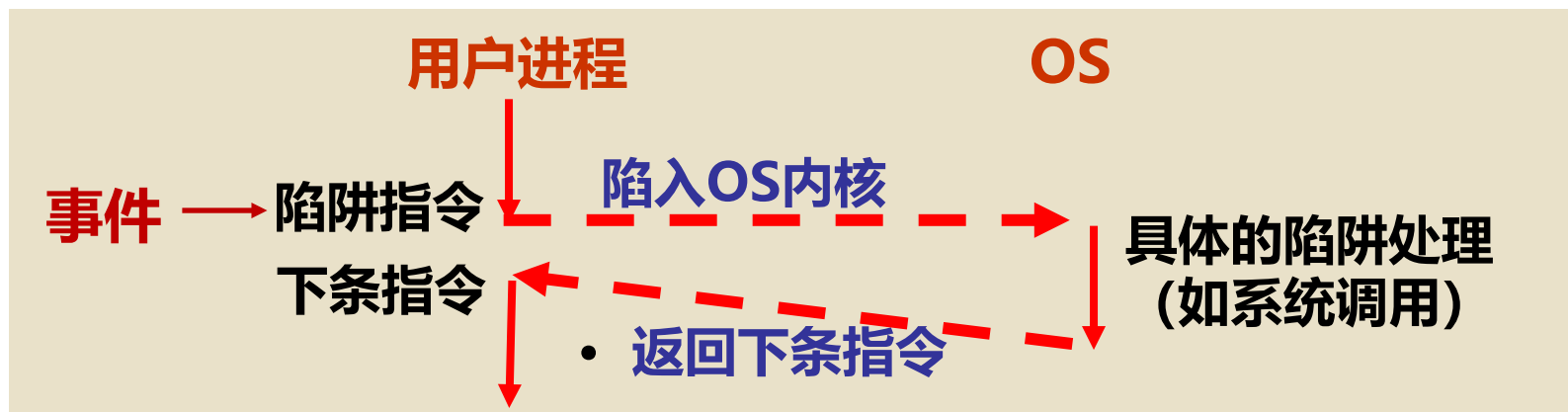
溢出、除数为0、非法操作、内存保护错等：终止当前进程。

- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！



# 陷阱 (Trap) 异常

- 陷阱也称自陷或陷入，执行陷阱指令（自陷指令）时，CPU调出特定程序进行相应处理，处理结束后返回到陷阱指令下一条指令执行。



- 陷阱的作用之一是在用户和内核之间提供一个像过程一样的接口，这个接口称为系统调用，用户程序利用这个接口可方便地使用操作系统内核提供的一些服务。操作系统给每个服务编一个号，称为系统调用号。例如，Linux系统调用fork、read和execve的调用号分别是1、3和11。
- IA-32处理器中的 int 指令和 sysenter 指令、MIPS处理器中的 syscall 指令等都属于陷阱指令（相当于“地雷”）。有条件“爆炸”
- 陷阱指令异常称为编程异常 (programmed exception)，这些指令包括 INT n、int 3、into (溢出检查)、bound (地址越界检查) 等

# 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”

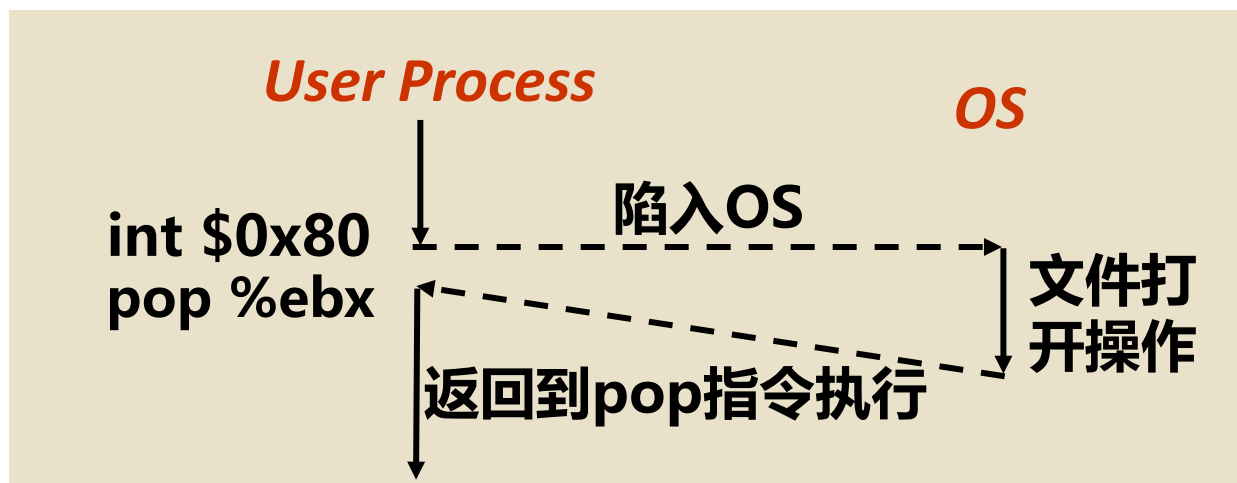
```
0804d070 <__libc_open>:
```

```
...
```

```
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

# 陷阱 (Trap) 异常

问题：你用过单步跟踪、断点设置等调试功能吗？你知道这些功能是如何实现的吗？ 通过“埋地雷”的方式实现

- 利用陷阱机制可实现程序调试功能，包括设置断点和单步跟踪
  - IA-32中，当CPU处于单步跟踪状态 ( $TF=1$ 且 $IF=1$ ) 时，每条指令都被设置成了陷阱指令，执行每条指令后，都会发生中断类型为1的“调试”异常，从而转去执行“单步跟踪处理程序”。
  - 注意：当陷阱指令是转移指令时，不能返回到转移指令的下条指令执行，而是返回到转移目标指令执行。
  - (在一定的条件下，每条指令都变成“地雷”)
  - IA-32中，用于程序调试的“断点设置”陷阱指令为int 3，对应机器码为CCH。若调试程序在被调试程序某处设置了断点，则调试程序就在该处“加”一条int 3指令（该首字节）。执行到该指令时，会暂停被调试程序的运行，并发出“EXCEPTION\_BREAKPOINT”异常，以调出调试程序执行，执行结束后回到被调试程序执行。

(int 3是一定爆炸的“地雷”)

SKIP

# 陷阱（Trap）异常

❓ 求 int 3 这条汇编指令的解释? 🍷 20

艺术家o417 | 浏览 2289 次



我有更好的答案 ✓

🏆 最佳答案

发布于2011-11-07 20:42 | 邀请更新

int 3机器码0cch，是无数人的最爱，你不见VC++不管是链接还是初始化，都用这个 0CCh来填，你不见所有运行在ring3的debug都使用这个0cc来插入你想中断调试的位置，以至在不去掉断点的情况下写盘，在你的代码中会发现这个莫名其妙的0CCh。

int 3不过是一个软件断点中断，你自己就可以任意修改这个中断，在anti-debug中，你可以通过修改这个中断给调试者带来巨大的惊喜，如果你能更深入一步的了解int 3的机制，你甚至可以指挥debug运行，即别人调试你的程序，单步往前走，你可以让他按一个单步执行键后倒退几步，哈哈（这些都是善意的，千万不要学将那个倒退的指令改成将别人硬盘的启动分区给删除了，将别人的文件分配表给删除了，或将人家硬盘给硬格式化了）。

int 3只不过是一个软件调试中断，通过他，你可以进入ring0，那个许多人都想进入的神秘世界。int 3只不过是一个简单的中断，利用他，你可以完成你许多过去不敢想象的事，只要你敢去尝试

[BACK](#)

# IA-32的标志寄存器

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	0	D	I	T	S	Z	0	A	0	P	1	C

- 6个条件标志

- OF、SF、ZF、CF各是什么标志（条件码）？
- AF：辅助进位标志（BCD码运算时才有意义）
- PF：奇偶标志

BACK

- 3个控制标志

- DF（Direction Flag）：方向标志（自动变址方向是增还是减）
- IF（Interrupt Flag）：中断允许标志（仅对外部可屏蔽中断有用）
- TF（Trap Flag）：陷阱标志（是否是单步跟踪状态）

- .....

# 异常的分类

“异常” 按处理方式分为故障、自陷和终止三类

**故障(fault)**：执行指令引起的异常事件，如溢出、缺页、堆栈溢出、访问超时等。 “断点” 为发生故障指令的地址

**自陷(Trap)**：预先安排的事件（“埋地雷”），如单步跟踪、断点、系统调用（执行访管指令）等。是一种自愿中断。

“断点” 为自陷指令下条指令地址

**终止(Abort)**：硬故障事件，此时机器将“终止”，调出中断服务程序来重启操作系统。 “断点” 是什么？ 随便！

**思考1：自陷处理完成后回到哪条指令执行？ 回到下条指令**

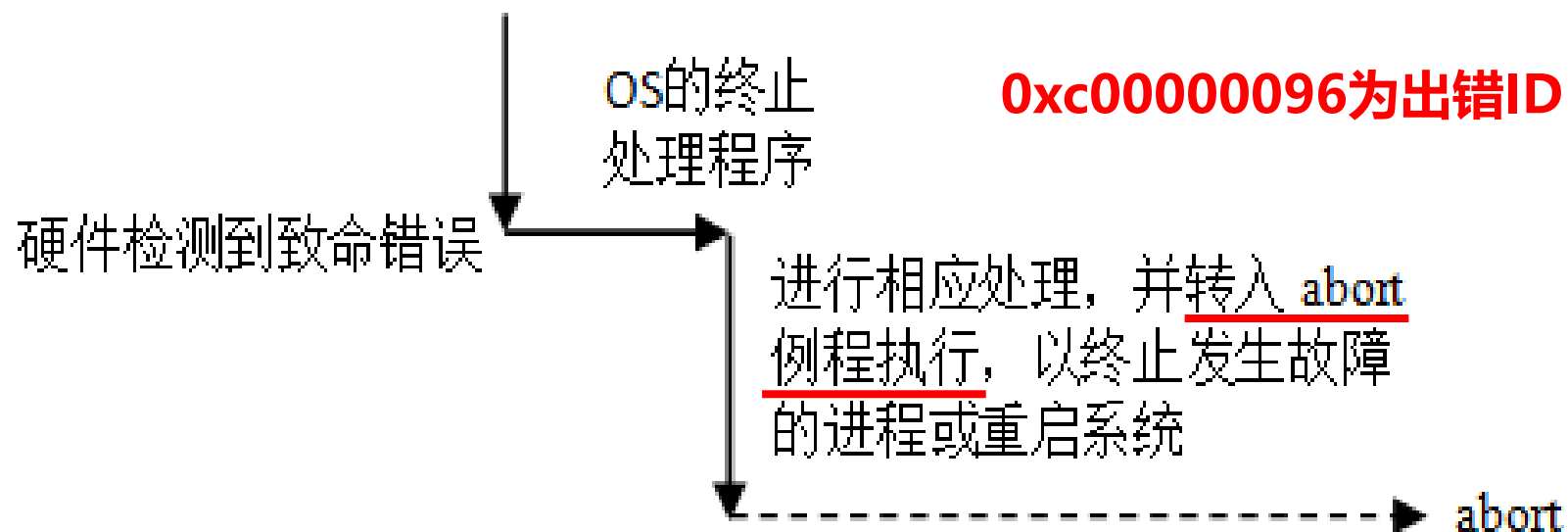
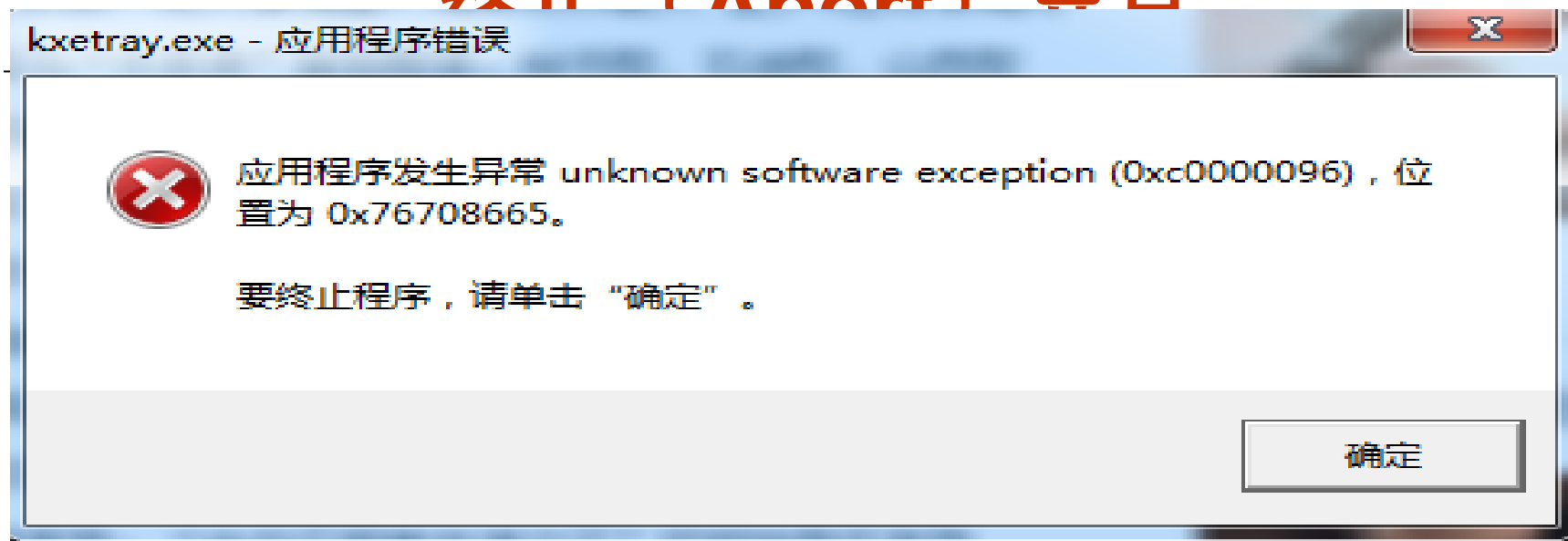
**思考2：哪些故障补救后可继续执行，哪些只好终止当前进程？**

缺页、TLB缺失等：补救后可继续，回到发生故障的指令重新执行。

溢出、除数为0、非法操作、内存保护错等：终止当前进程。

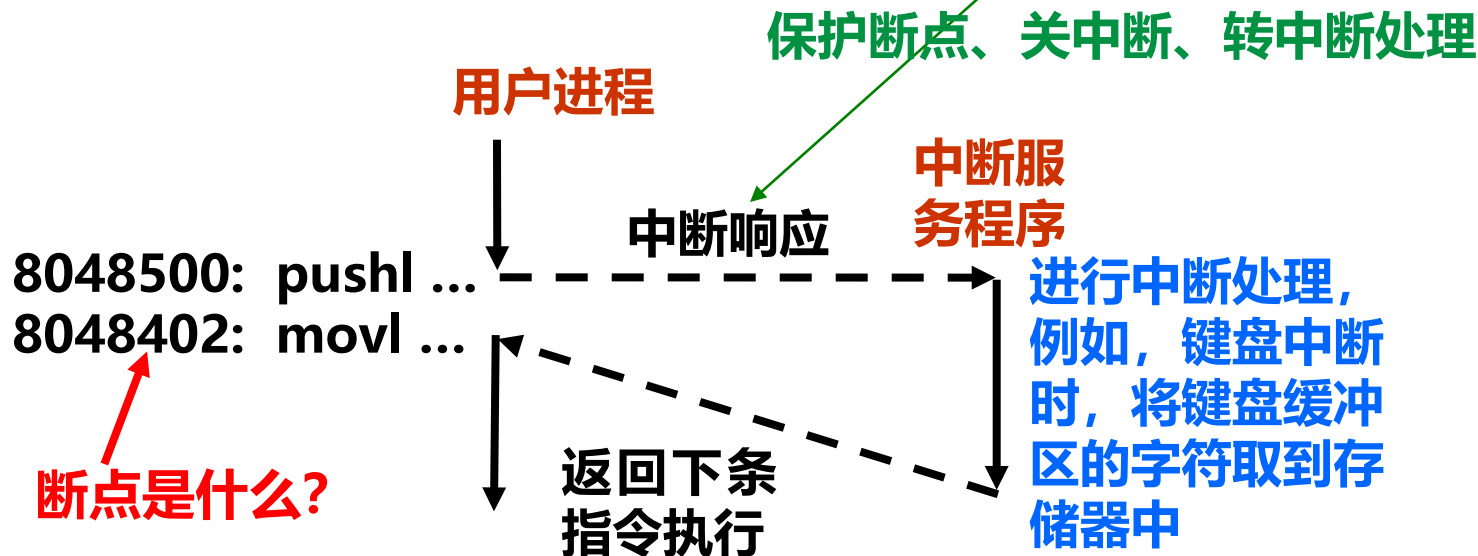
- 不同体系结构和教科书对“异常”和“中断”定义的内涵不同，在看书时要注意！

# 终止 (Abort) 异常



# 中断的概念

- 外设通过**中断请求信号线**向CPU提出“中断”请求，不由指令引起，故中断也称为**异步异常**。
- 事件：**Ctrl-C**、**DMA传送结束**、**网络数据到达**、**打印缺纸**、.....
- 每执行完一条指令，CPU就查看中断请求引脚，若**引脚的信号有效**，则进行**中断响应**：将当前PC（断点）和当前机器状态保存到栈中，并“关中断”，然后，从数据总线读取中断类型号，根据中断类型号跳转到对应的中断服务程序执行。**中断检测及响应过程由硬件完成**。
- 中断服务程序执行具体的中断处理工作，中断处理完成后，再回到被打断程序的“断点”处继续执行。

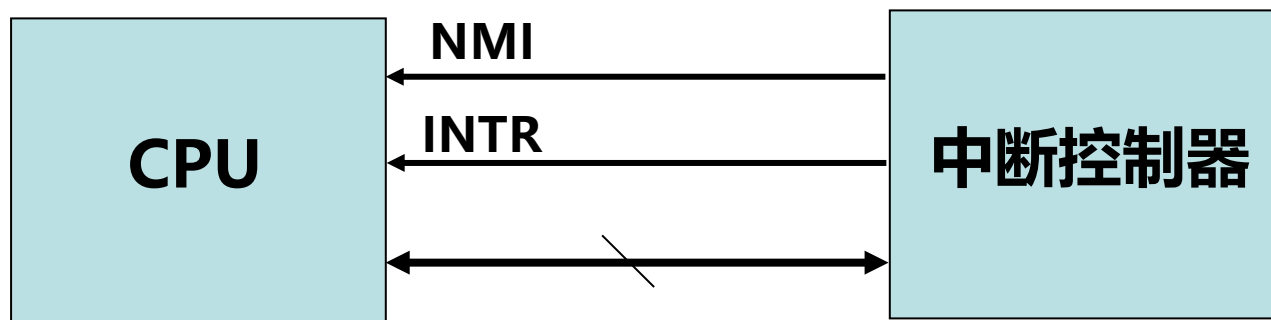


溢出、整除0、缺页等异常和外部中断都是由硬件检测并响应的!



# 中断的分类

- Intel将中断分成**可屏蔽中断**（maskable interrupt）和**不可屏蔽中断**（nonmaskable interrupt, NMI）。
  - **可屏蔽中断**：通过 INTR 向CPU请求，可通过设置**屏蔽字**来屏蔽请求，若中断请求被屏蔽，则不会被送到CPU。
  - **不可屏蔽中断**：非常紧急的硬件故障，如：电源掉电，硬件线路故障等。通过 NMI 向CPU请求。一旦产生，就被立即送CPU，以便快速处理。这种情况下，中断服务程序会尽快保存系统重要信息，然后在屏幕上显示相应的消息或直接重启系统。



# 异常/中断响应过程

检测到异常或中断时，CPU须进行以下基本处理：

① 关中断（“中断允许位”清0）：使CPU处于“禁止中断”状态，以防止新中断破坏断点（PC）、程序状态（PSW）和现场（通用寄存器）。

② 保护断点和程序状态：将断点和程序状态保存到栈或特殊寄存器中

PC→栈 或 EPC（专门存放断点的寄存器）

PSWR →栈 或 EPSWR（专门保存程序状态的寄存器）

PSW（Program Status Word）：程序状态字

PSWR（PSW寄存器）：如IA-32中的EFLAGS寄存器

③ 识别中断事件

有软件识别和硬件识别（向量中断）两种不同的方式。

IA-32中，响应异常时不关中断，只在响应中断时关中断

# 异常/中断响应过程

有两种不同的识别方式：软件识别和硬件识别（向量中断）。

## (1) 软件识别（MIPS采用）

设置一个异常状态寄存器（MIPS中为Cause寄存器），用于记录异常原因。操作系统使用一个统一的异常处理程序，该程序按优先级顺序查询异常状态寄存器，识别出异常事件。

（例如：MIPS中位于内核地址0x8000 0180处有一个专门的异常处理程序，用于检测异常的具体原因，然后转到内核中相应的异常处理程序段中进行具体的处理）

## (2) 硬件识别（向量中断）（IA-32采用）

用专门的硬件查询电路按优先级顺序识别中断，得到“中断类型号”，根据此号，到中断向量表中读取对应的中断服务程序的入口地址。

所有事件都被分配一个“中断类型号”，每个中断都有相应的“中断服务程序”，可根据中断类型号找到中断服务程序的入口地址。

中断类型号相当于中断向量表的索引，表中存放中断服务程序首地址

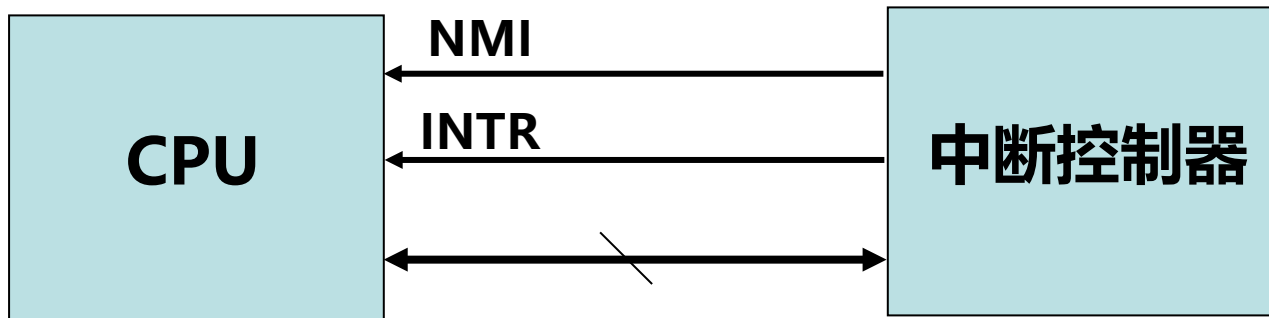
# IA-32的向量中断方式

- 有256种不同类型的异常和中断
- 每个异常和中断都有唯一编号，称之为中断类型号（也称向量号）。如类型0为“除法错”，类型2为“NMI中断”，类型14为“缺页”
- 每个异常和中断有与其对应的异常处理程序或中断服务程序，其入口地址放在一个专门的中断向量表或中断描述符表中。
- 前32个类型（0~31）保留给CPU使用，剩余的由用户自行定义（这里的用户指机器硬件的用户，即操作系统）
- 通过执行INT n（指令第二字节给出中断类型号n，n=32~255）使CPU自动转到OS给出的中断服务程序执行
- 实模式下，用中断向量表描述
- 保护模式下，用中断描述符表描述

SKIP

# 回顾：中断的分类

- Intel将中断分成**可屏蔽中断**（maskable interrupt）和**不可屏蔽中断**（nonmaskable interrupt, NMI）。
  - **可屏蔽中断**：通过 INTR 向CPU请求，可通过设置**屏蔽字**来屏蔽请求，若中断请求被屏蔽，则不会被送到CPU。
  - **不可屏蔽中断**：非常紧急的硬件故障，如：电源掉电，硬件线路故障等。通过 NMI 向CPU请求。一旦产生，就被立即送CPU，以便快速处理。这种情况下，中断服务程序会尽快保存系统重要信息，然后在屏幕上显示相应的消息或直接重启系统。



[BACK](#)

# 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
- **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

# 实地址模式下的中断向量表

**实地址模式 (Real Mode)** 是Intel为80286及其之后的处理器提供的一种8086兼容模式。寻址空间1MB，指令地址=CS<<4+IP。中断向量表位于0000H~03FFH。共256组，每组占四个字节CS:IP。

例1：除法错的中断类型号为0，故其向量地址为：0x4=0

除法错  
单步

例2：NMI的中断类型号为2，故其向量地址为：2x4=8

NMI  
⋮

CS:IP	000~003H
CS:IP	004~007H
CS:IP	008~00BH
⋮	
CS:IP	
CS:IP	3FC~3FFH

**实地址模式下没有分页管理机制！**

中断向量表中每一项是对应中断服务程序或异常处理程序的入口地址，被称为**中断向量**(Interrupt Vector)。

# 实地址模式下的中断向量表

- 开机后系统首先在实地址模式下工作（只有1MB的寻址空间）
- 开机过程中，需要先准备在实模式下的中断向量表和中断服务程序。通常，由固化在主板上一块ROM芯片中的**BIOS程序**完成
- BIOS程序检测显卡、键盘、内存等，并在00000H ~ 003FFH区建立中断向量表，在中断向量所指主存区建立相应的中断服务程序
- BIOS**利用INT指令**执行**特定的中断服务程序**把OS从磁盘加载到内存中。例如，BIOS可通过执行int 0x19指令来调用中断向量0x19对应的中断服务程序，将**启动盘上的0号磁头对应盘面的0磁道1扇区中的引导程序装入内存**
- BIOS（Basic Input/Output System）是**基本输入/输出系统**的简称，是针对具体主板设计的，**与安装的操作系统无关**。
- BIOS包含各种**基本设备驱动程序**，通过执行BIOS程序，基本设备驱动程序以中断服务程序的形式被加载到内存，以提供基本I/O系统调用。
- 一旦进入保护模式，就不再使用BIOS。



# 保护模式下的中断描述符表

- 保护模式下，通过中断描述符表获异常处理或中断服务程序入口地址
- 中断描述符表** (Interrupt Descriptor Table, **IDT**) 是OS内核中的一个表，共有256个表项，每个表项占8个字节，IDT共占用2KB
- IDTR**中存放 IDT在内存的首地址
- 每一个表项是一个**中断门描述符**、**陷阱门描述符**或**任务门描述符**

**段选择符**用来指示异常处理程序或中断服务程序所在段的段描述符在GDT中的位置，其RPL=0；

**偏移地址**则给出异常处理程序或中断服务程序第一条指令所在偏移量。

## 中断门描述符格式：

偏移地址 (A31-A16)															
P	DPL	0	1	1	1	0	0	0	0	0	0	0	0	0	0
段选择符															
偏移地址 (A15-A0)															

P: Linux总把P置1。DPL: 访问本段要求的最低特权级。主要用于防止恶意应用程序通过 **INT n** 指令模拟非法异常而进入内核态执行破坏性操作

TYPE: 标识门的类型。TYPE=1110B: 中断门; TYPE=1111B: 陷阱门;  
TYPE=0101B : 任务门

# IA-32中异常和中断的处理

- 引导程序被读到内存后，开始执行引导程序，以装入操作系统内核，并对GDT、IDT等进行初始化，系统启动后，进入保护模式
- IA-32中，每条指令执行后，下条指令的**逻辑地址（虚拟地址）由CS和EIP指示**  
    实地址模式下：指令地址=CS<<4+IP
- 每条指令执行过程中，CPU会根据执行情况判定是否发生了某种内部异常事件，并在每条指令执行结束时判定是否发生了外部中断请求  
    （由此可见，**异常事件和中断请求的检测都是在某一条指令执行过程中进行的，显然由硬件完成**）
- 在CPU根据CS和EIP取下条指令之前，会根据检测的结果判断是否进入**中断响应阶段**  
    （**异常和中断的响应也都是在某一条指令执行过程中或执行结束时进行的，显然也由硬件完成**）

# IA-32中异常和中断响应过程

SKIP

- (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  $CPL \neq DPL$ ，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）。

**下个时钟周期开始，从CS:EIP所指处开始执行异常或中断处理程序！**

# 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

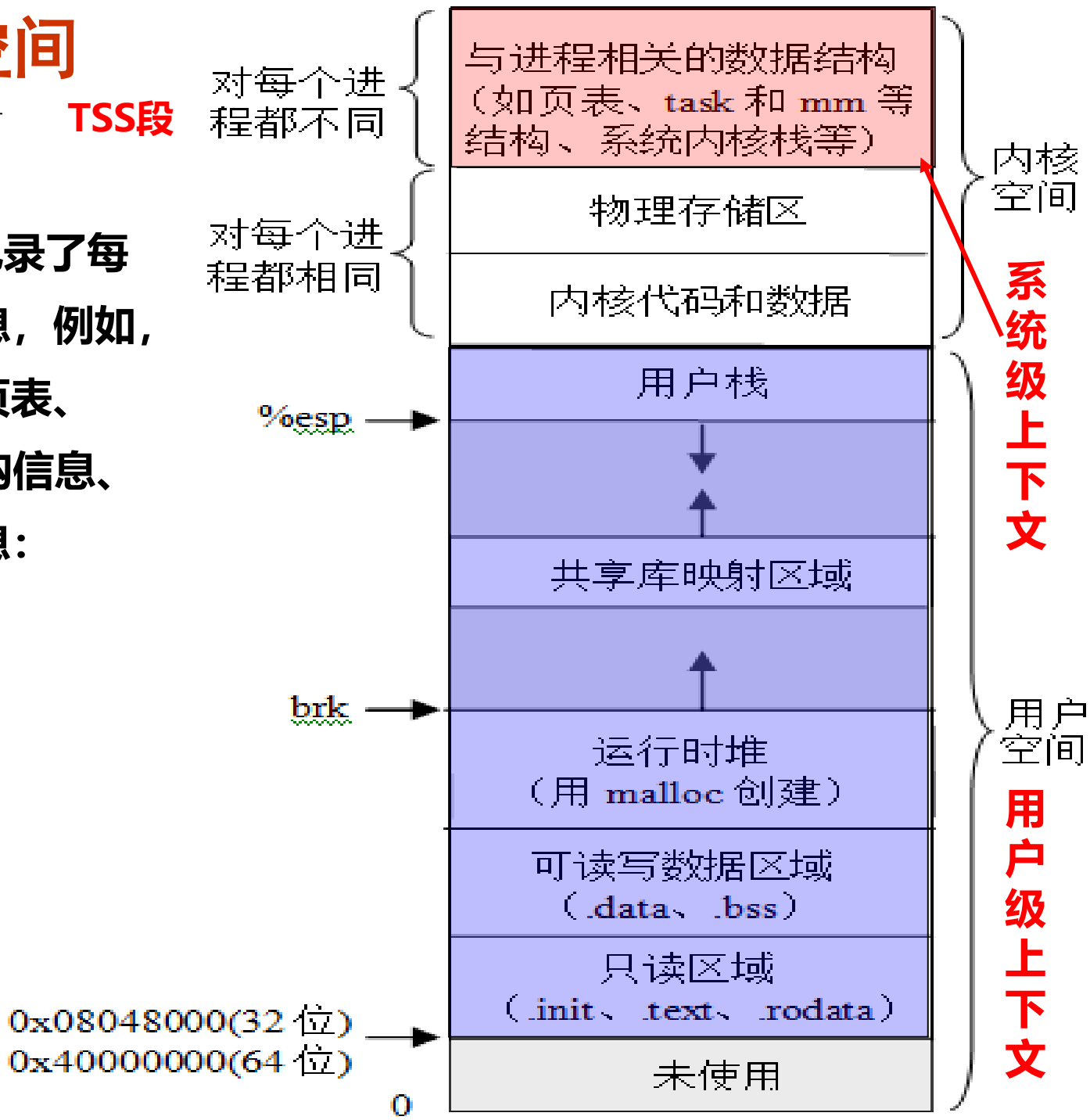
初始化时，上述4个段描述符被存放在GDT中

[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、内核栈的栈顶信息：SS:ESP 等

[BACK](#)



# IA-32中异常和中断返回过程

**中断或异常处理程序最后一条指令是IRET。CPU在执行IRET指令过程中完成以下工作：**

- (1) 从栈中弹出硬件出错码（保存过的话）、EIP、CS和EFLAGS**
- (2) 检查当前异常或中断处理程序的CPL是否等于CS中最低两位，若是则说明异常或中断响应前、后都处于同一个特权级，此时，IRET指令完成操作；否则，再继续完成下一步工作。**
- (3) 从内核栈中弹出SS和ESP，以恢复到异常或中断响应前的特权级进程所使用的栈。**
- (4) 检查DS、ES、FS和GS段寄存器的内容，若其中有某个寄存器的段选择符指向一个段描述符且其DPL小于CPL，则将该段寄存器清0。这是为了防止恶意应用程序（CPL=3）利用内核以前使用过的段寄存器（DPL=0）来访问内核地址空间。**

**执行完IRET指令后，CPU回到原来发生异常或中断的进程继续执行**

# 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。
- **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 EIP 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 全局描述符表	段选择符	Linux 全局描述符表	段选择符
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APMBIOS 32-bit code	0xb8
TLS #3	0x43	APMBIOS 16-bit code	0xc0
reserved		APMBIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 ( __KERNEL_CS )	not used	
kernel data	0x68 ( __KERNEL_DS )	not used	
user code	0x73 ( __USER_CS )	not used	
user data	0x7b ( __USER_DS )	not used	
		double fault TSS	0xf8

所有中断门和陷阱门描述符中的段选择符都是0x60

任务门描述符中的段选择符都是0xf8



# 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

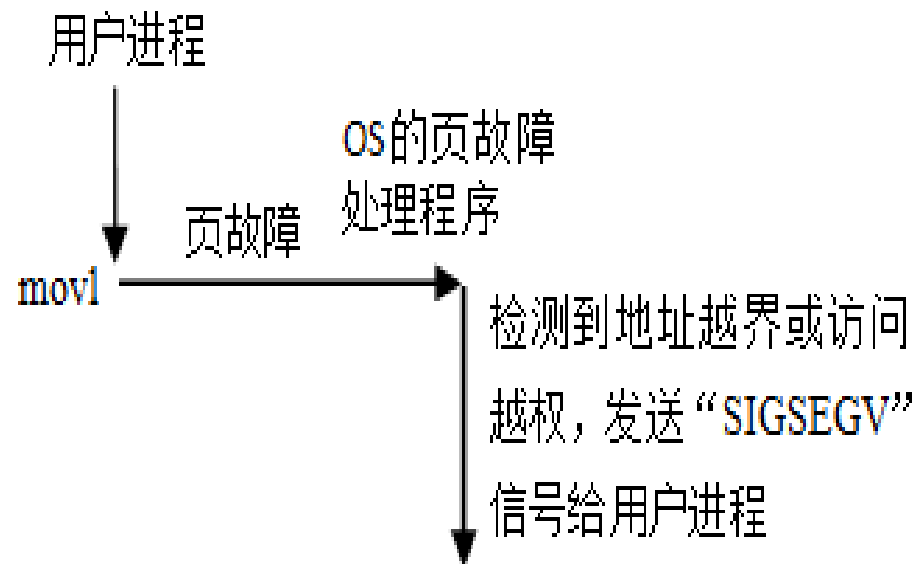
所有异常处理程序的结构是一致的，都可划分成以下三个部分：

- (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（可恢复）或信号处理程序（不可恢复）执行。



# Linux

Linux中  
异常对应的  
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  
法错误  
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);
```

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！

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

**sigsetjmp:** 若直接调用则返回0, 若从siglongjmp调用返回则返回非0值

```
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);
}
```

默认的SIGFPE信号处理程序显示:  
"Floating point exception"

# 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 implicit declaration of function 'exit'
    exit(0);
    ^
linuxer@debian:~$ ./sigtest
starting
Floating point exception
linuxer@debian:~$ _
```

**sigsetjmp: 若直接调用则返回0, 若从siglongjmp调用返回则返回非0值**

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

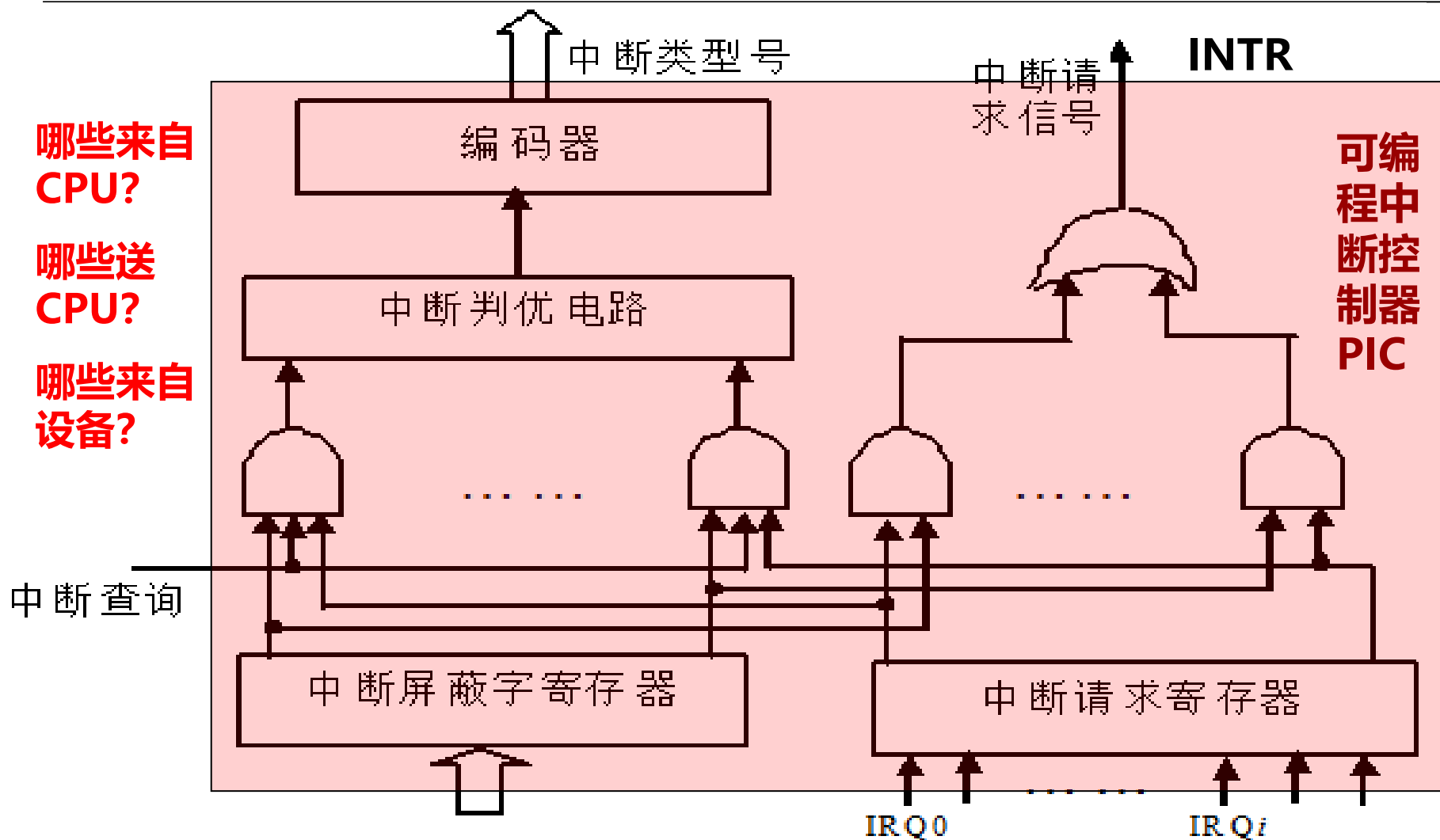
int main()
{
    int a, t;
    signal(SIGFPE, FLPhandler);

    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



每个中断源都有一个编号，如 $IRQ_0$ 、 $IRQ_1$ 、...、 $IRQ_i$ 、...，可将与 $IRQ_i$  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**（**发中断查询信号、关中断、保护断点和机器状态**），**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 IRQ<sub>i</sub> 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 IRQ<sub>i</sub> 对应的中断服务例程 ISR（Interrupt Server Routine）。中断型号为**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”

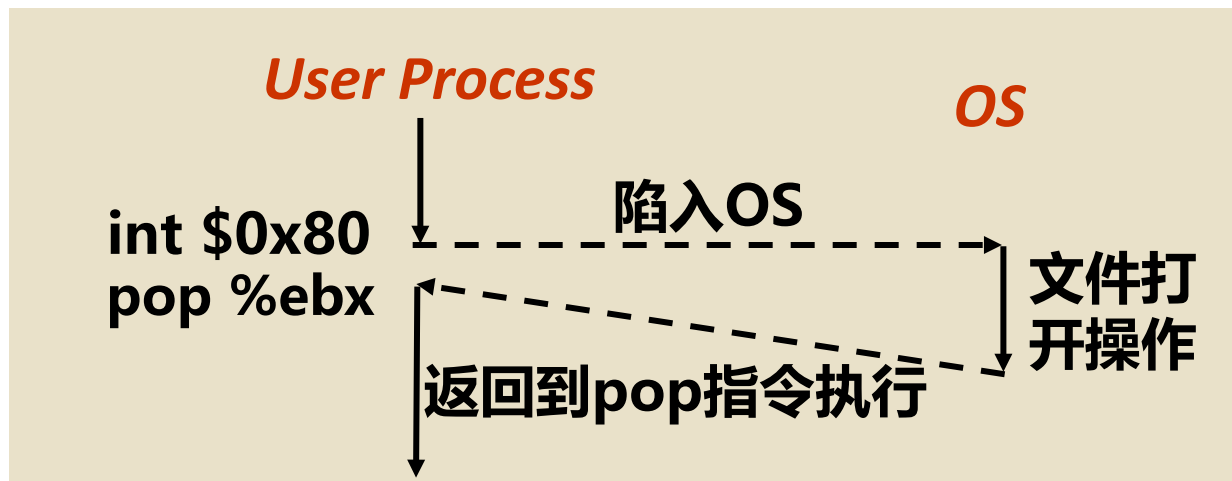
```
0804d070 <__libc_open>:
```

```
...
```

```
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**

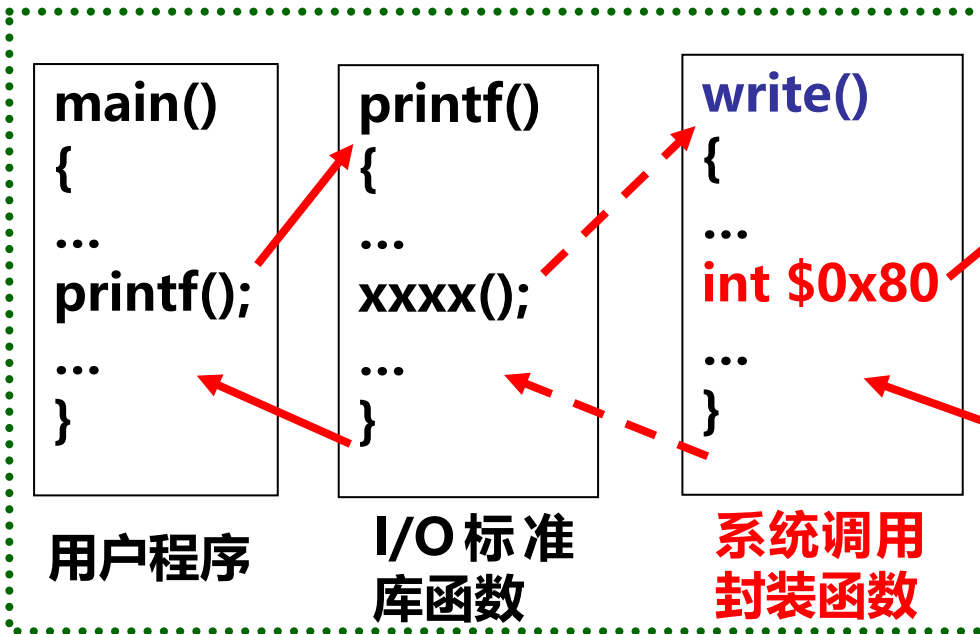
**int \$0x80 //系统调用**

# IA-32/Linux的系统调用

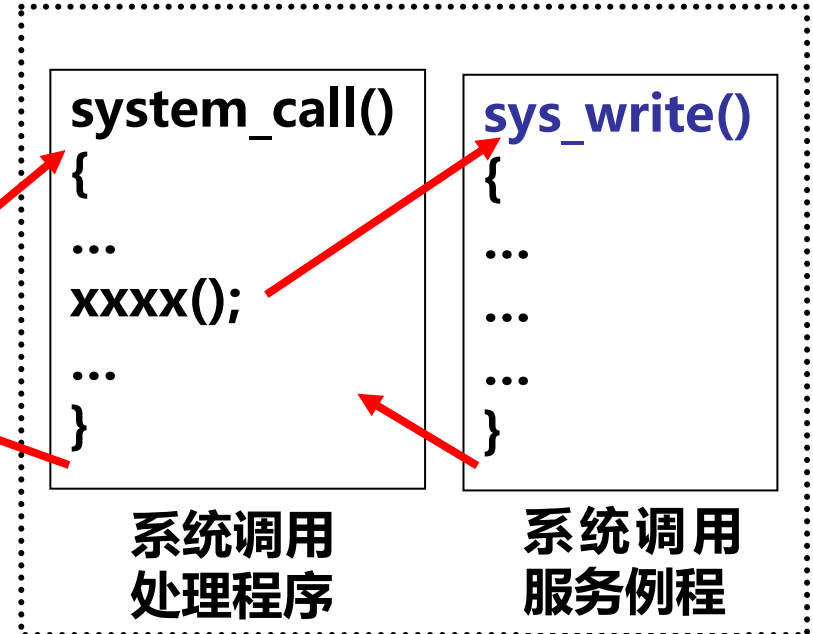
调用号	名称	类别	含义	调用号	名称	类别	含义
1	exit	进程控制	终止进程	12	chdir	文件系统	改变当前工作目录
2	fork	进程控制	创建一个新进程	13	time	系统控制	取得系统时间
3	read	文件操作	读文件	19	lseek	文件系统	移动文件指针
4	write	文件操作	写文件	20	getpid	进程控制	获取进程号
5	open	文件操作	打开文件	37	kill	进程通信	向进程或进程组发信号
6	close	文件操作	关闭文件	45	brk	内存管理	修改虚拟空间中的堆指针 brk
7	waitpid	进程控制	等待子进程终止	90	mmap	内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

用户空间、运行在用户态



内核空间、运行在内核态



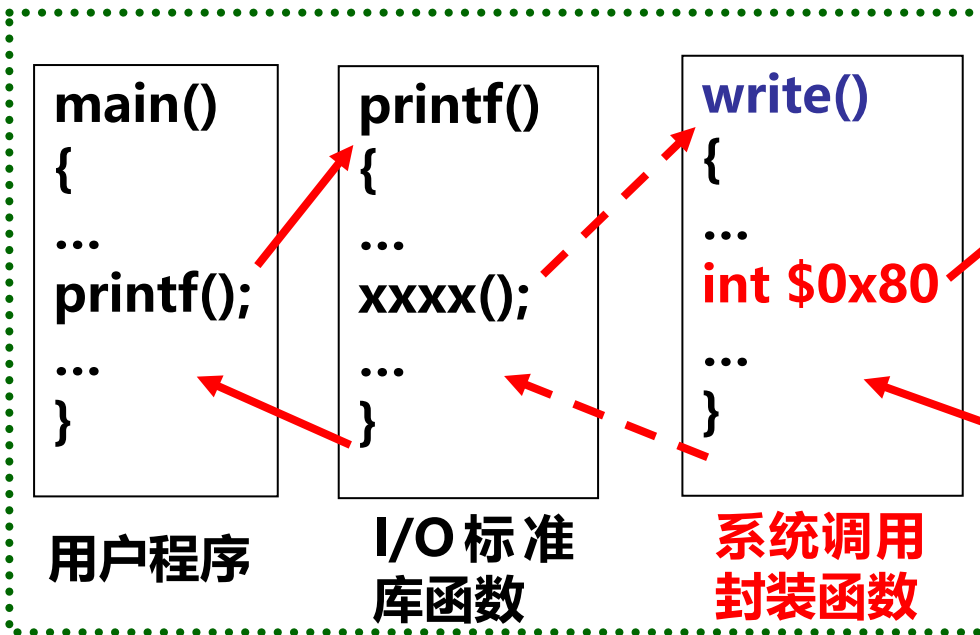
- 某函数调用了`printf()`，执行到调用`printf()`语句时，便会转到C语言I/O标准库函数`printf()`去执行；
- `printf()`通过一系列函数调用，最终会调用函数`write()`；
- 调用`write()`时，便会通过一系列步骤在内核空间中找到`write`对应的系统调用服务例程`sys_write`来执行。

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

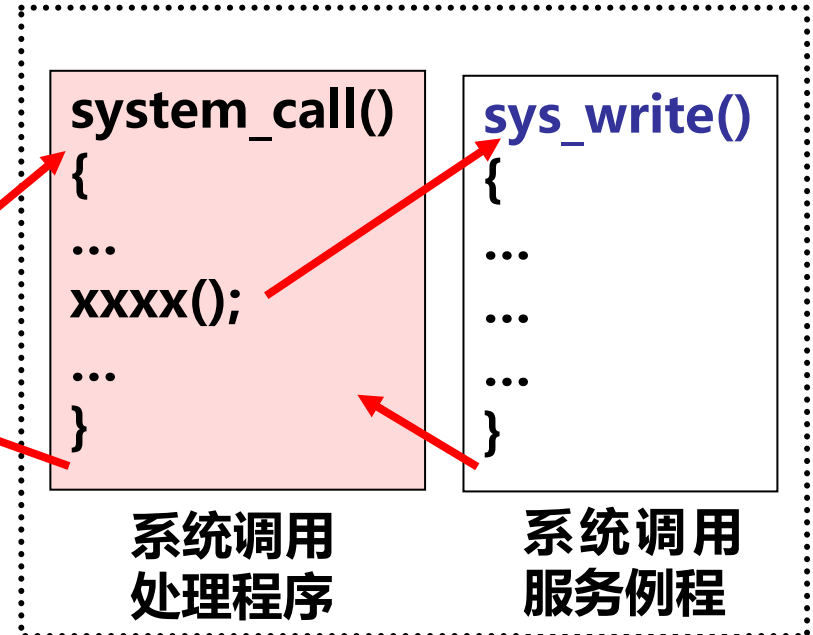


# Linux系统中printf()函数的执行过程

用户空间、运行在用户态



内核空间、运行在内核态



- 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

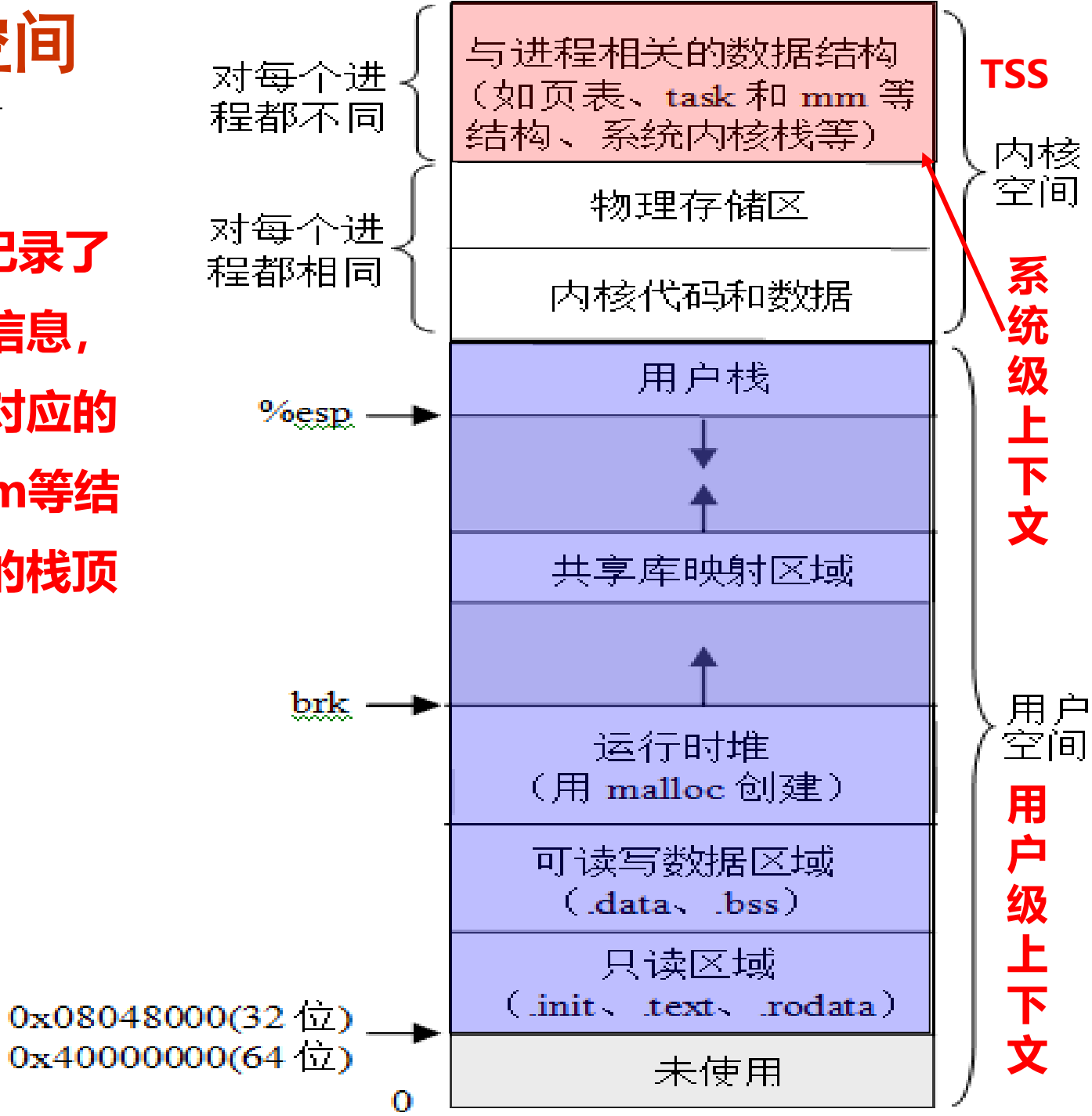
初始化时，上述4个段描述符被存放在GDT中

[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、基址址等信息。Linux下中断门和陷阱门对应的即为内核代码段，所以DPL为0，基址址为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  $CPL \neq DPL$ ，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](#)

Linux 全局描述符表	段选择符	Linux 全局描述符表	段选择符
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APMBIOS 32-bit code	0xb8
TLS #3	0x43	APMBIOS 16-bit code	0xc0
reserved		APMBIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 ( __KERNEL_CS )	not used	
kernel data	0x68 ( __KERNEL_DS )	not used	
user code	0x73 ( __USER_CS )	not used	
user data	0x7b ( __USER_DS )	not used	
		double fault TSS	0xf8

所有中断门和陷阱门描述符中的段选择符都是0x60

任务门描述符中的段选择符都是0xf8

# 总 结

---

- 每个被打断的逻辑控制流处都发生了一个异常控制流
- 异常控制流的原因有多种：
  - 操作系统进行进程的处理器调度，进行进程上下文切换
  - 硬件在执行指令时检测到有异常或中断事件
  - 一个进程利用信号机制向另一个进程发送信号
- 进程的引入给程序员两个假象，简化了编程、编译、链接、装入执行整个过程
  - 独占使用处理器、独占使用存储器
- 硬件在执行一条指令过程中检测到异常或中断，硬件会通过响应过程调出内核中相应处理程序，整个内核程序不是一个进程，而是一个“内核控制路径”，它代表当前进程在内核态执行单独的一个指令序列。
- 不同类型的异常或中断，其处理方式不同：
  - 对于陷阱指令（埋地雷），相当于一个过程调用（引发特定处理）；
  - 对于无法恢复的故障类异常，则向当前进程发送一个特定信号，当前进程接受到信号后，调用相应的信号处理程序执行或调用内核的abort例程终止当前进程（如果没有对应的信号处理程序的话）；
  - 对于可恢复故障类异常，则相应的异常处理程序处理完故障后，会回到当前进程的故障指令继续执行；
  - 对于外部中断，则在相应的中断服务程序执行后，回到当前进程的下一条指令继续执行。