Chapter7 异常控制流

第一讲：进程与进程的上下文切换

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。

1.1 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、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）

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）

1.2 “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

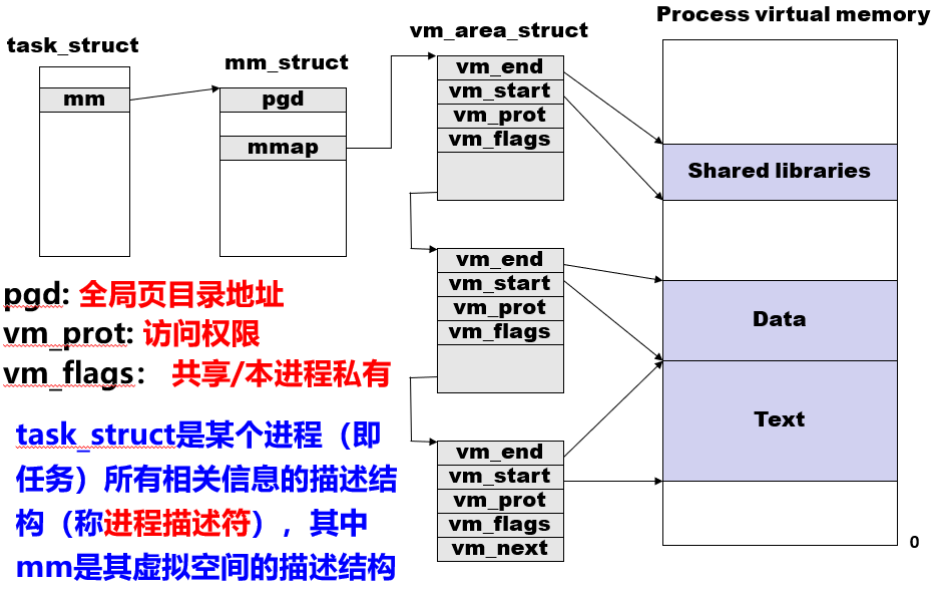
- 进程是OS对CPU执行的程序的运行过程的一种抽象。进程有自己的生命周期，它由于任务的启动而创建，随着任务的完成（或终止）而消亡，它所占用的资源也随着进程的终止而释放。
- 一个可执行目标文件（即程序）可被加载执行多次，也即，一个程序可能对应多个不同的进程。

  — 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。

Linux将虚存空间组织成“区域”的集合



pgd: 全局页目录地址
vm_prot: 访问权限
vm_flags: 共享/本进程私有

task_struct是某个进程（即任务）所有相关信息的描述结构（称进程描述符），其中mm是其虚拟空间的描述结构

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

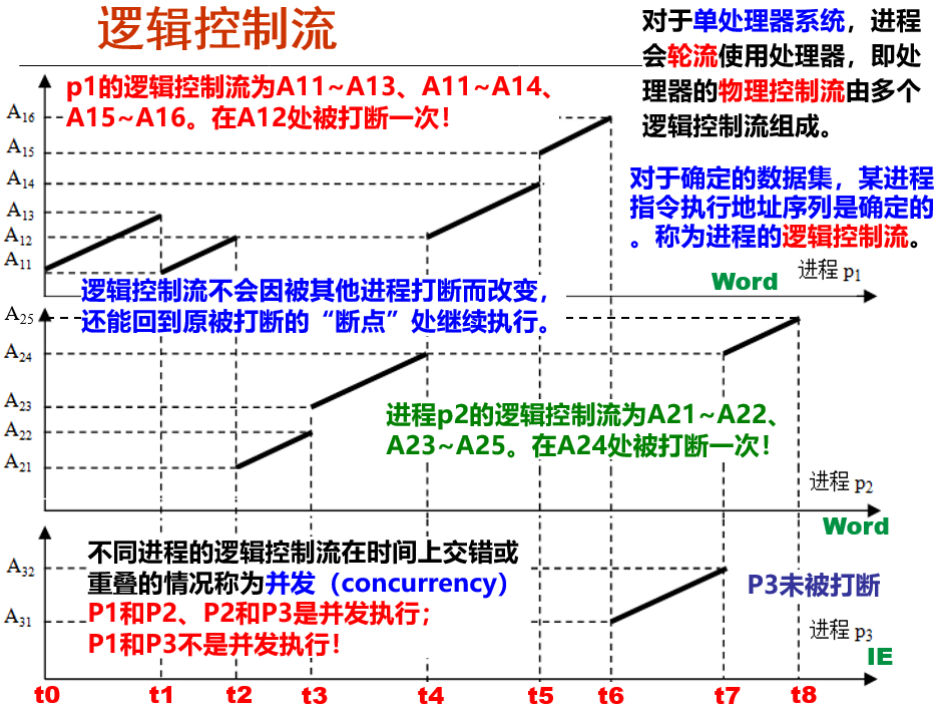
  每个进程拥有一个独立的逻辑控制流，使得程序员以为自己的程序在执行过程中独占使用处理器

  —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

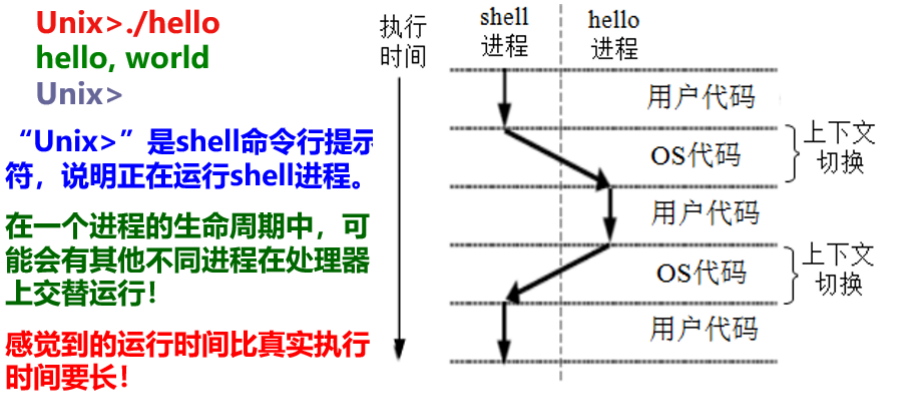
进程”的引入简化了程序员的编程以及语言处理系统的处理，即简化了编程、编译、链接、共享和加载等整个过程。

1.3 逻辑控制流



1.4 “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）



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。

进程切换的时候，主要切换的是CPU里的东西；shell也是用户进程

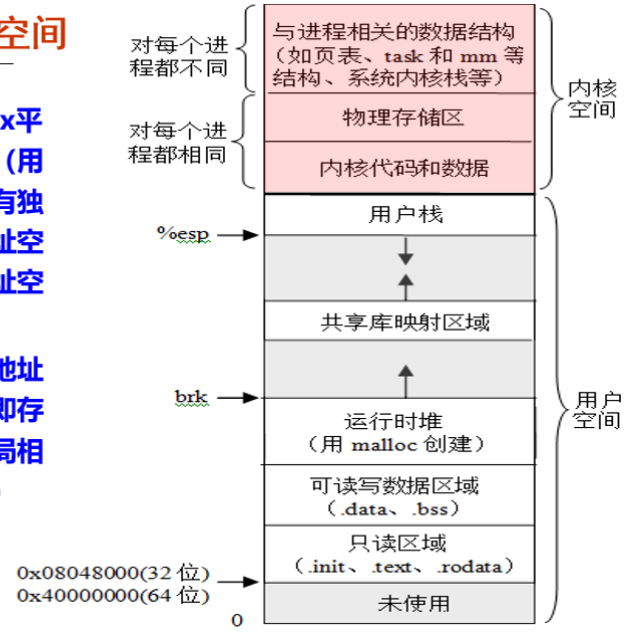
“进程” 的“上下文”

- 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）**



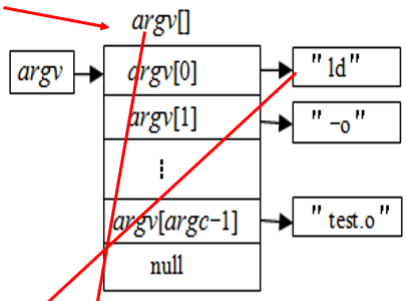
1.5 程序的加载和运行

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

若在shell命令行提示符下输入以下命令行

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

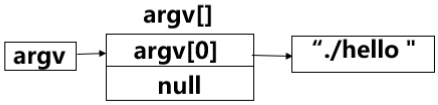
在shell命令行提示符后键入命令并按“enter”键后，便构造argv和envp，然后调用execve()函数来启动加载器，最终转main()函数执行

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

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：Unix>./hello[enter]

Step2：shell命令行解释器构造argv和envp



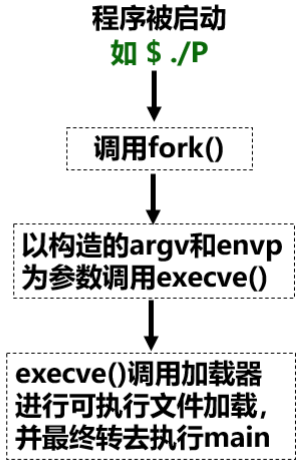
Step3：调用fork()函数，创建一个子进程，与父进程shell完全相同（只读/共享），包括只读段、可读写数据段、堆以及用户栈等。

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、数据等内容）

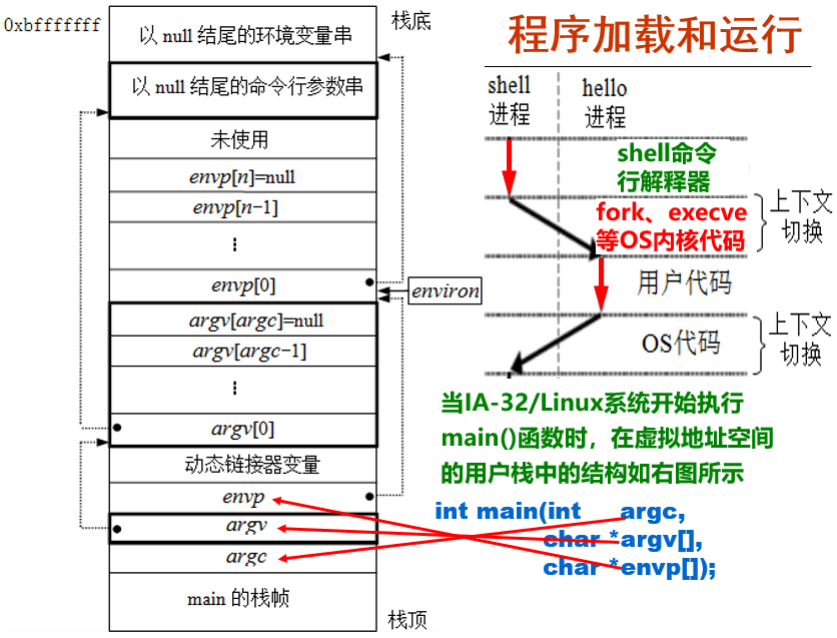
Step5：调用hello程序的main()函数，hello程序开始在一个进程的上下文中运行。 int main(int argc, char *argv[], char *envp[]);

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）
- 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。



_start: __libc_init_first → _init → atexit → main → _exit



第二讲：异常和中断的基本概念

2.1 基本概念

- 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

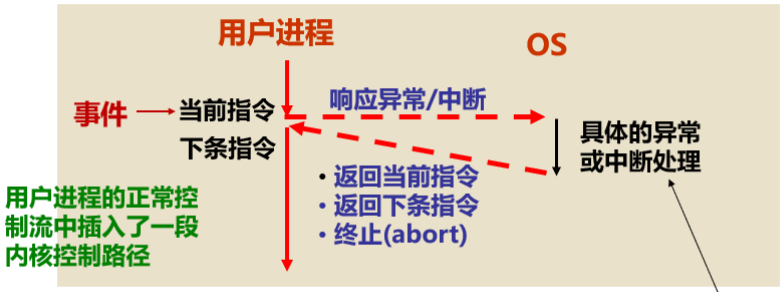
  硬故障中断：如电源掉电、硬件线路故障等

  程序性中断：执行某条指令时发生的“例外(Exception)”，如溢出、缺页、越界、越权、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。

2.2 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。

2.3 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，如单步跟踪、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。

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。

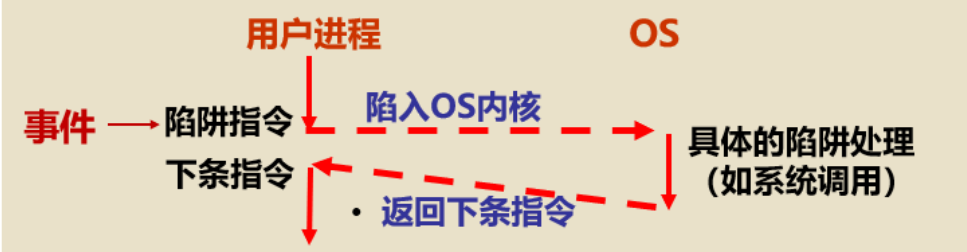
溢出、除数为0、非法操作、内存保护错等：终止当前进程。

故障（fault）异常

见PPT “异常举例—页故障”

陷阱（Trap）异常

- 陷阱也称自陷或陷入，**执行陷阱指令（也称为自陷指令）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的下一条指令执行。**



- 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**。
- **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

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？

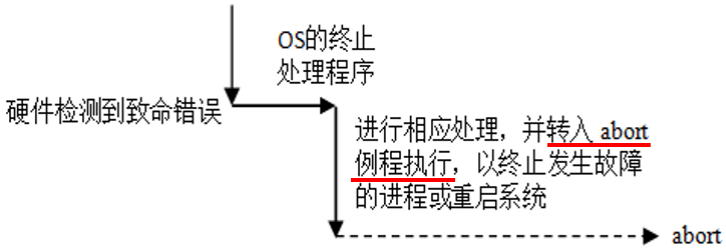
- **利用陷阱机制可实现程序调试功能，包括设置断点和单步跟踪**
  - **IA-32中，当CPU处于单步跟踪状态（TF=1且IF=1）时，每条指令都被设置成了陷阱指令，执行每条指令后，都会发生中断类型号为1的“调试”异常，从而转去执行“单步跟踪处理程序”。**

    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。
  - **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入一条int 3指令。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。**
- **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）**

终止（Abort）异常

如果在执行指令过程中发生了严重错误，例如，控制器出现问题，访问DRAM或SRAM时发生校验错等，则程序将无法继续执行，**只好终止发生问题的进程**，在有些严重的情况下，甚至要重启系统。

这种异常是随机发生的，无法确定发生异常的是哪条指令。



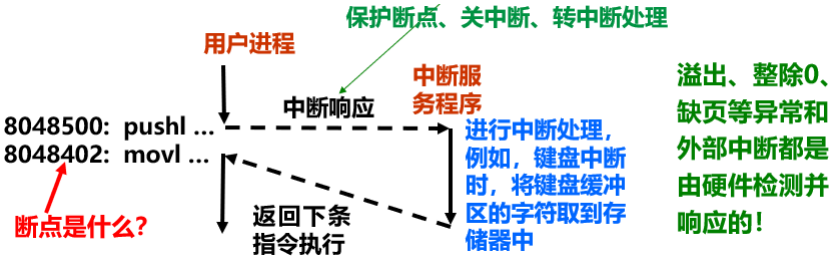
2.4 中断的概念与分类

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。



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。

第三讲：异常和中断的响应、处理

3.1 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中的的EFLAGS寄存器

③ 识别异常事件

有软件识别和硬件识别（向量中断）两种不同的方式。

（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异常，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

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。

3.2 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。

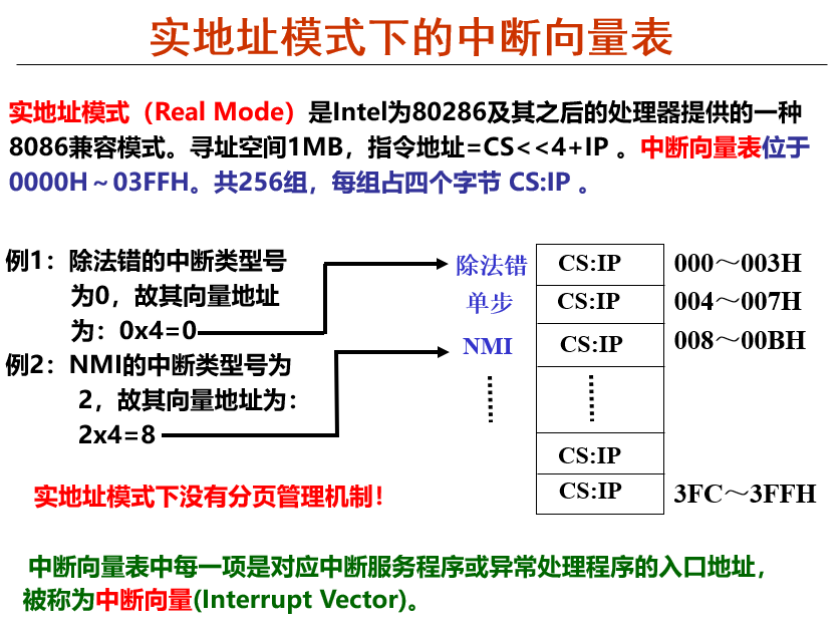
前32个类型（0~31）保留给CPU使用，剩余的由用户自行定义（这里的用户指机器硬件的用户，即操作系统）

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

实地址模式下的中断向量表



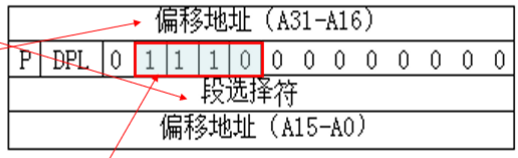
保护模式下的中断描述符表

- **保护模式下，通过中断描述符表获取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**
- **寄存器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：



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

系统启动过程中，操作系统内核被装入，并对GDT、IDT等进行了初始化，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**

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进行的，显然也由硬件完成**）

3.3 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）。

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！

3.4 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，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 < 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

第四讲：IA-32/Linux下的异常/中断机制

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**

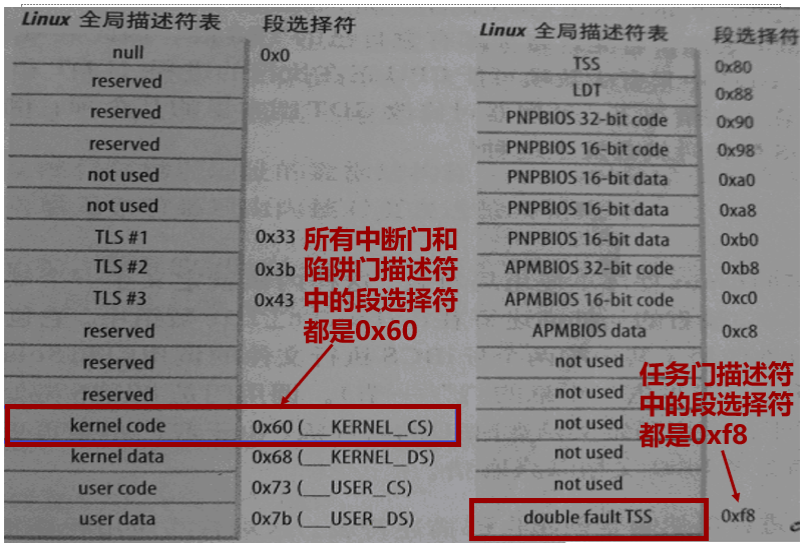
试想一下处理缺页时又有非法指令或栈溢出等！

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 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



4.1 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（用户可以用的陷阱）：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。

4.2 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。

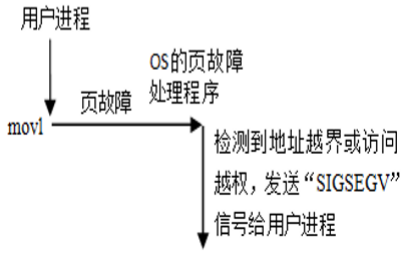
所有异常处理程序的结构是一致的，都可划分成以下三个部分：

（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。

4.3 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，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。

4.4 IA-32/Linux的系统调用

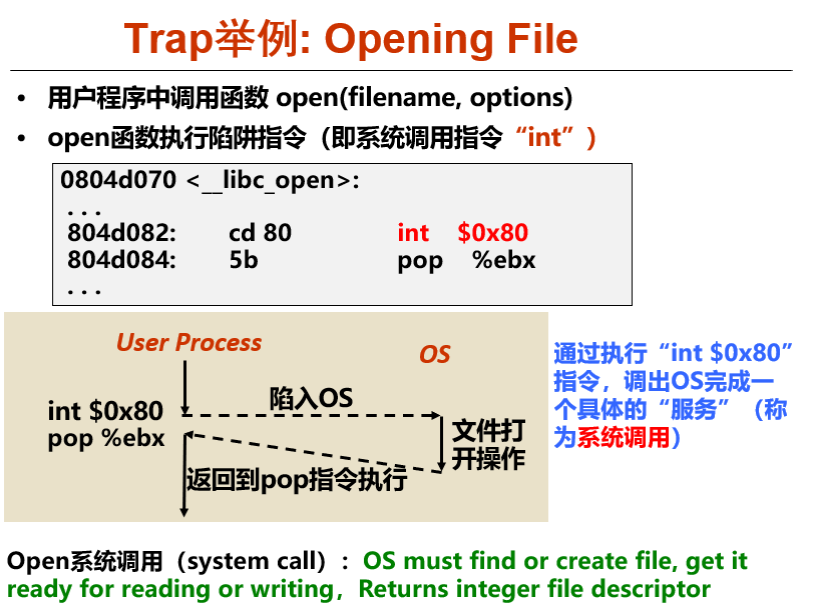
系统调用是特殊异常事件，是OS为用户程序提供服务的手段。

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，表中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



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
```

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。

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，相当于一个过程调用；

  —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。