基于x86处理器

例子：LINUX系统调用实现

下面我们再以 Linux 为例，看一看基于 x86 处理器的 Linux

LINUX的系统调用实现
——基于X86处理器

- 陷入指令选择128号

  int $0x80

sched_init()中
set_system_gate(0
x80, &system_call)

- 门描述符
  - 系统初始化时：对IDT表中的128号门初始化
  - 门描述符的2、3两个字节：内核代码段选择符

    0、1、6、7四个字节：偏移量（指向system_call()）
  - 门类型：15，陷阱门，为什么？
  - DPL：3，与用户级别相同，允许用户进程使用该门描述符

系统调用号示例

(INCLUDE/ASM-I386/UNISTD.H)

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid       7
#define __NR_creat         8
#define __NR_link          9
#define __NR_unlink        10
#define __NR_execve        11
#define __NR_chdir         12
#define __NR_time          13
...
```

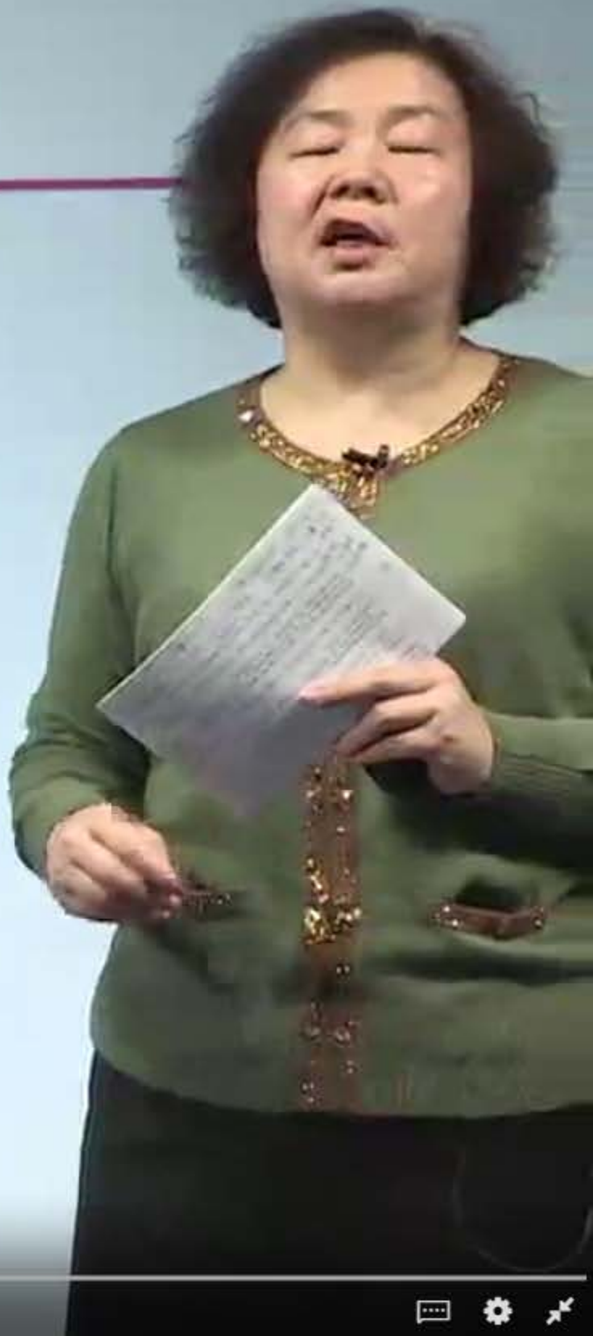
系统执行INT $0X80指令

- 由于特权级的改变，要切换栈

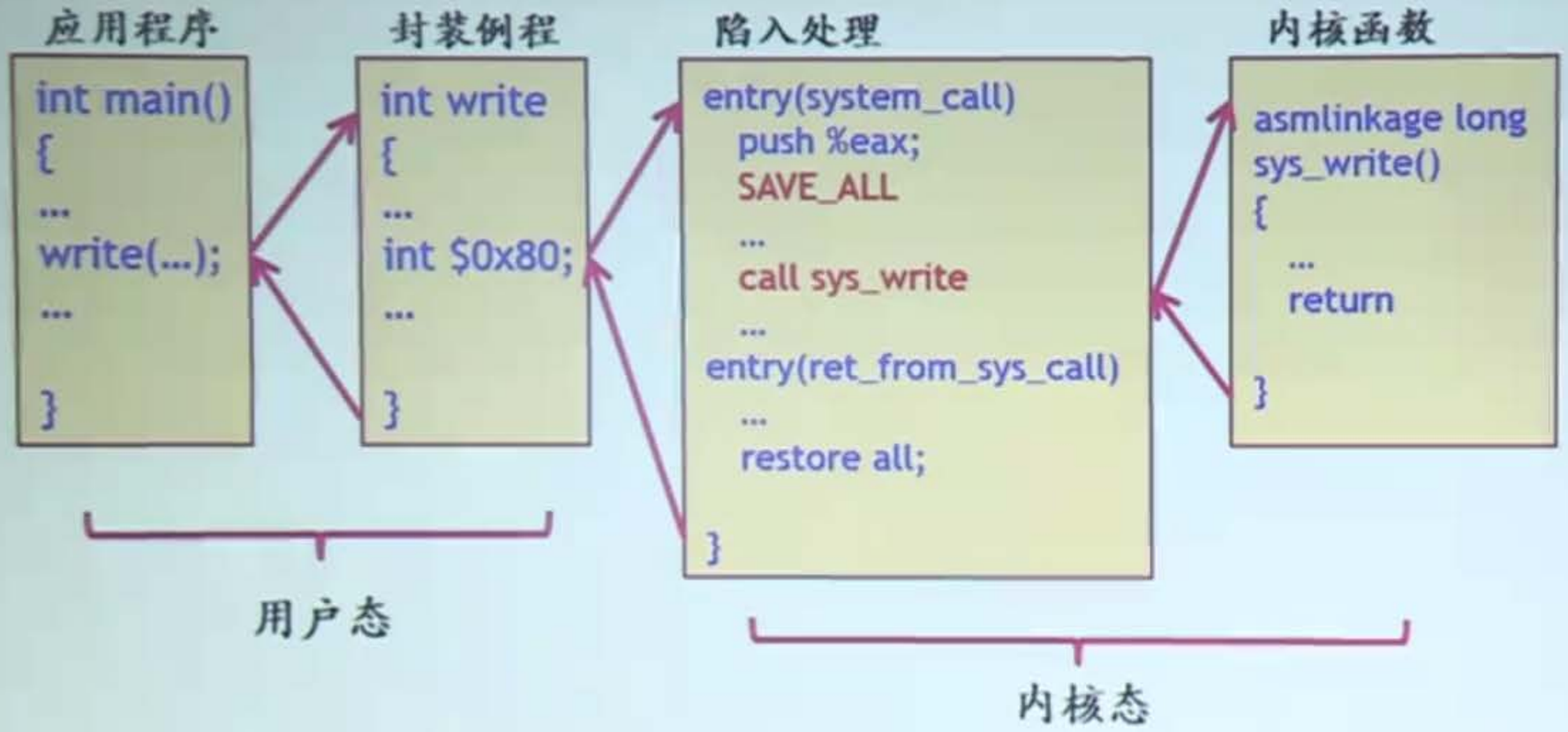
  用户栈 → 内核栈

  CPU从任务状态段TSS中装入新的栈指针（SS：ESP），指向内核栈

- 用户栈的信息（SS：ESP）、EFLAGS、用户态CS 、EIP 寄存器的内容压栈（返回用）
- 将EFLAGS压栈后，复位TF，IF位保持不变
- 用128在IDT中找到该门描述符，从中找出段选择符装入代码段寄存器CS
- 代码段描述符中的基地址 + 陷阱门描述符中的偏移量 → 定位 system_call()的入口地址

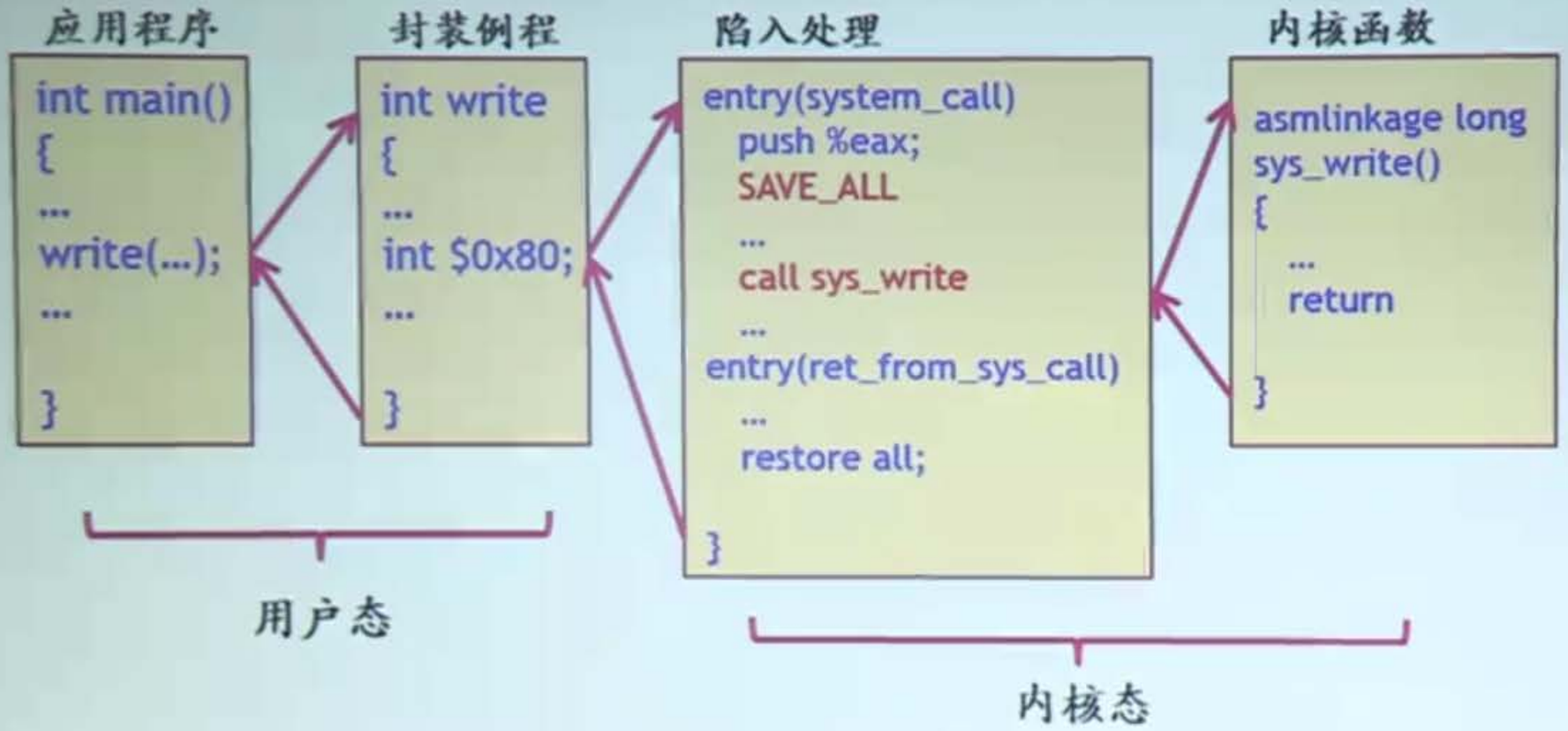


LINUX系统调用执行流程



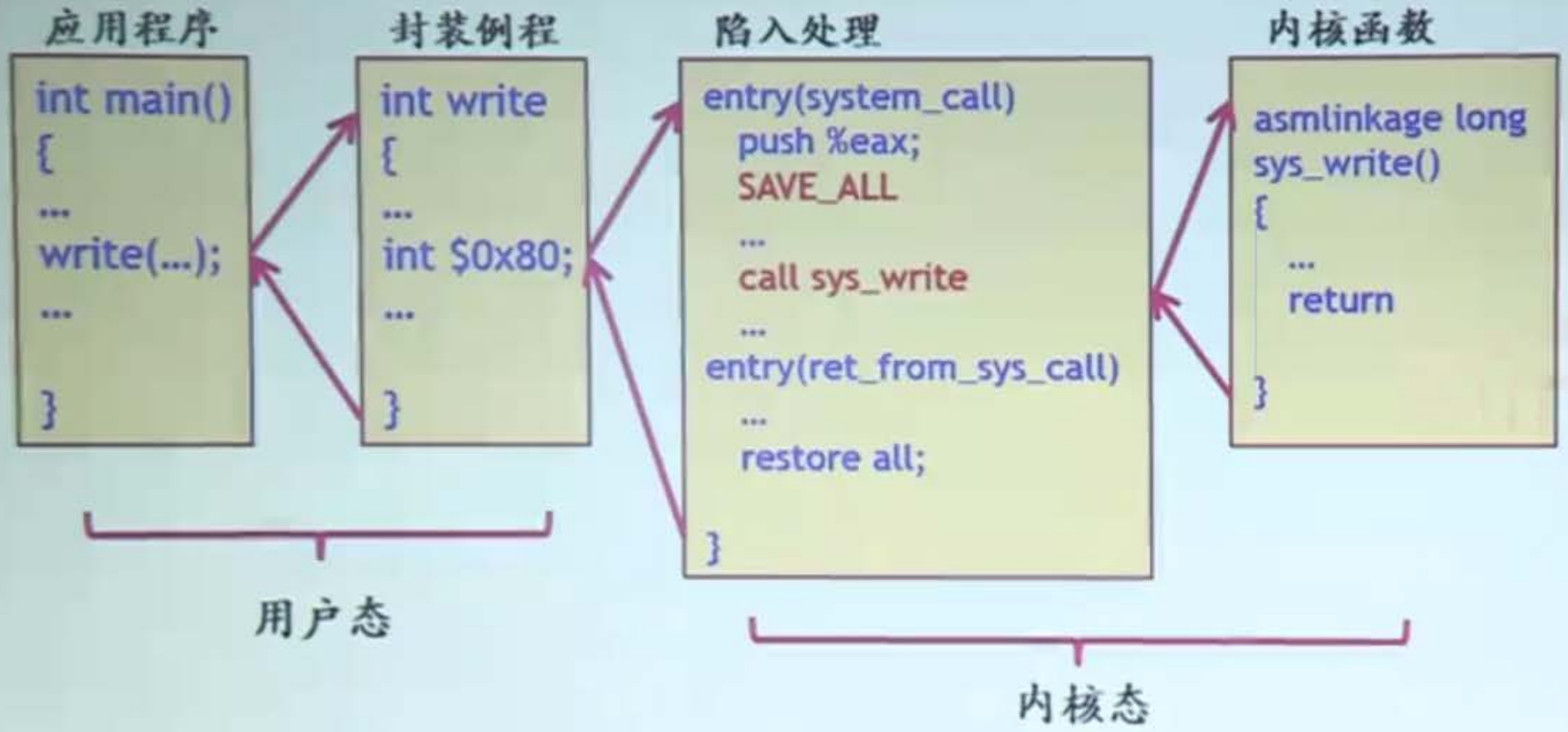
用户态下调用C库的库函数，比如write()

LINUX系统调用执行流程

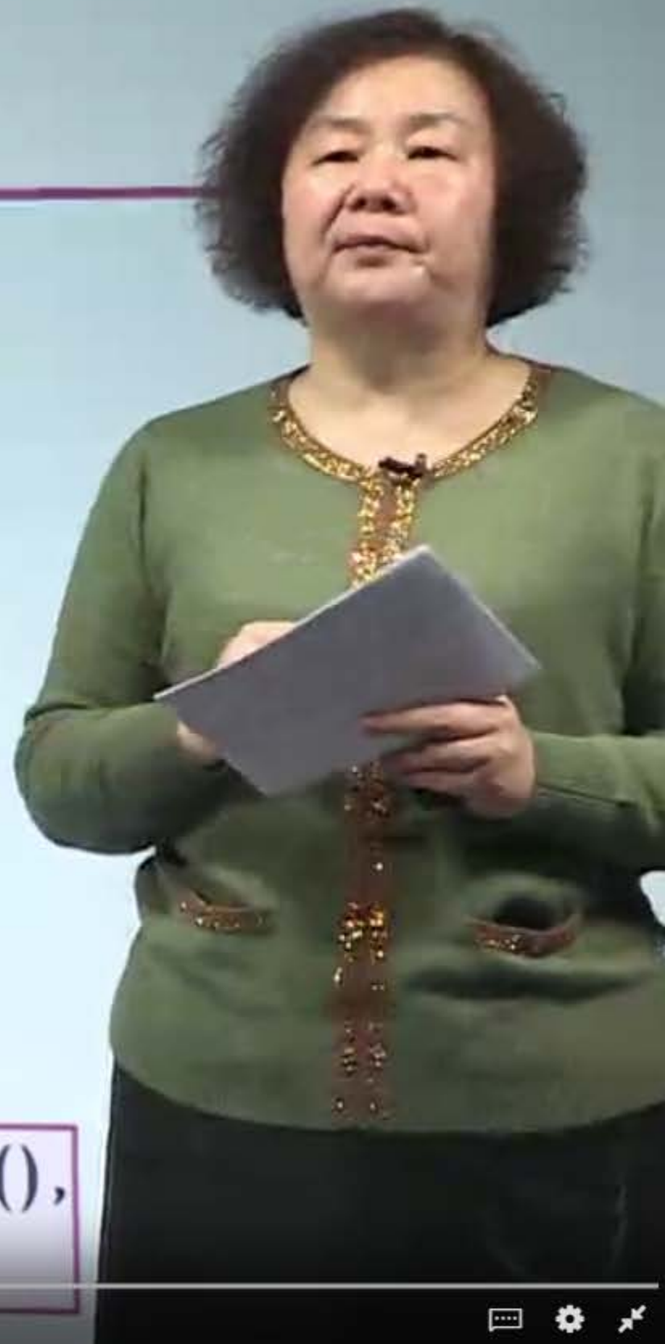


封装后的write()先做好参数传递工作，然后使用int 0x80指令产生一次异常

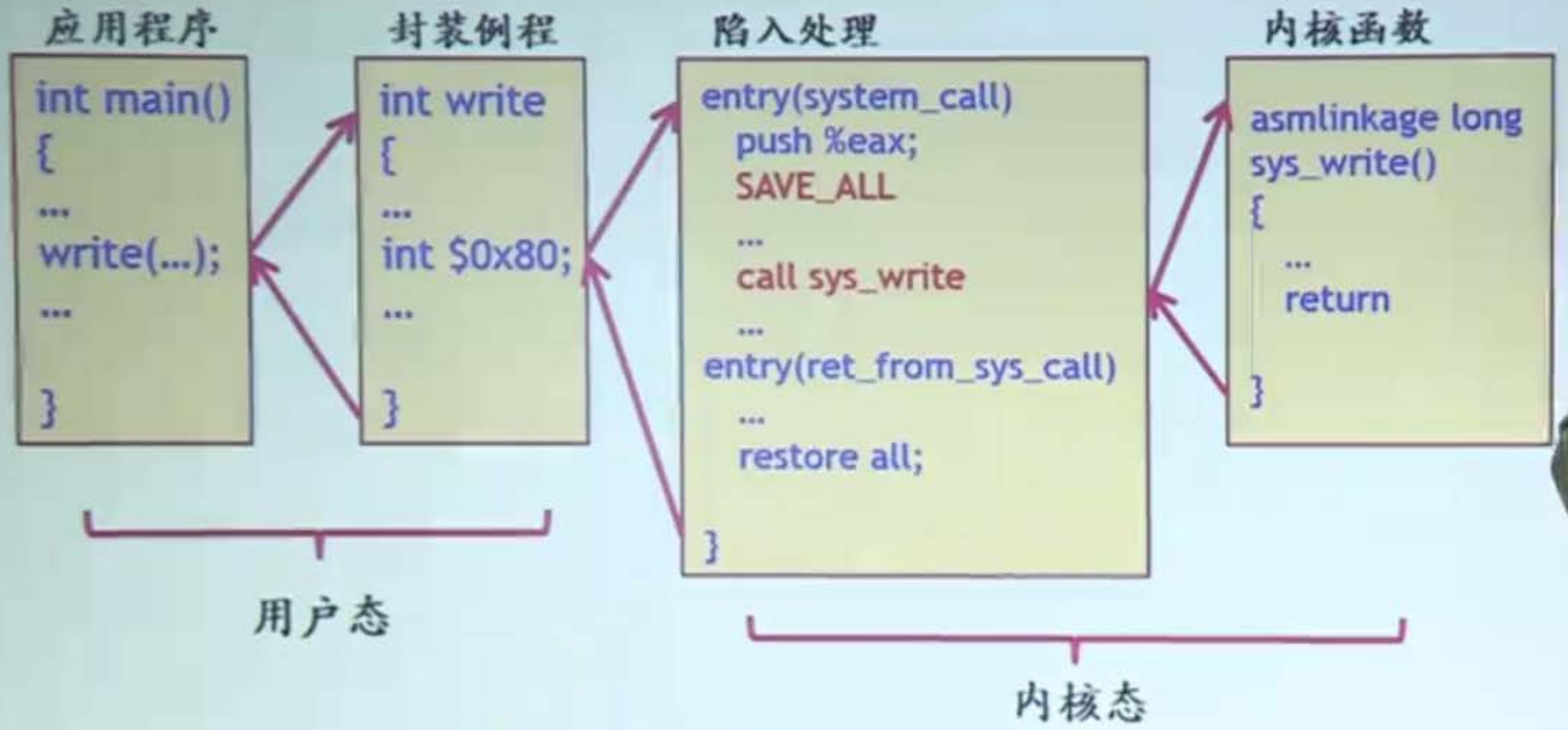
LINUX系统调用执行流程



CPU通过0x80号在IDT中找到对应的服务例程system_call()，并调用之

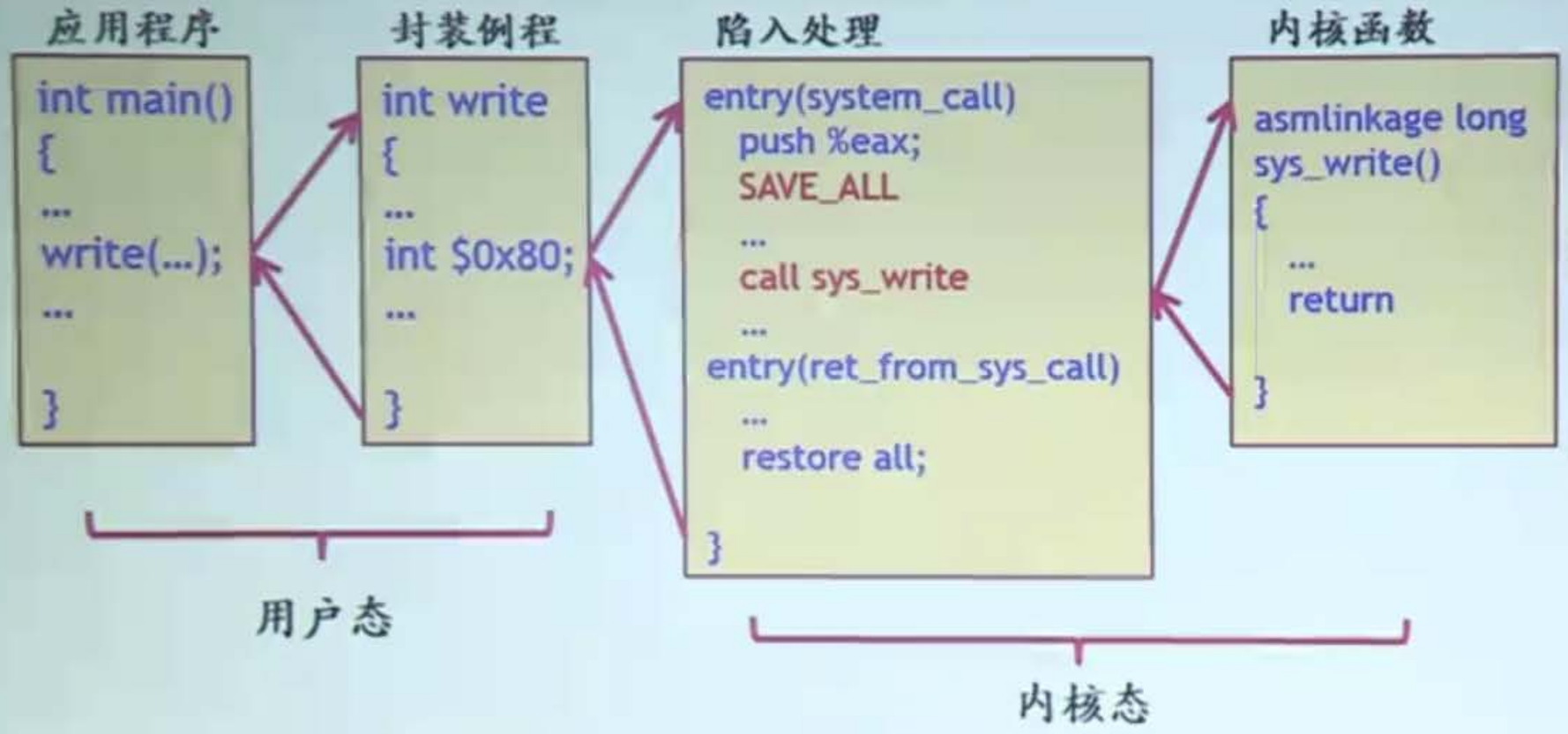


LINUX系统调用执行流程



system_call()：将参数保存在内核栈；根据系统调用号索引系统调用表，找到系统调用程序入口，比如sys_write()

LINUX系统调用执行流程

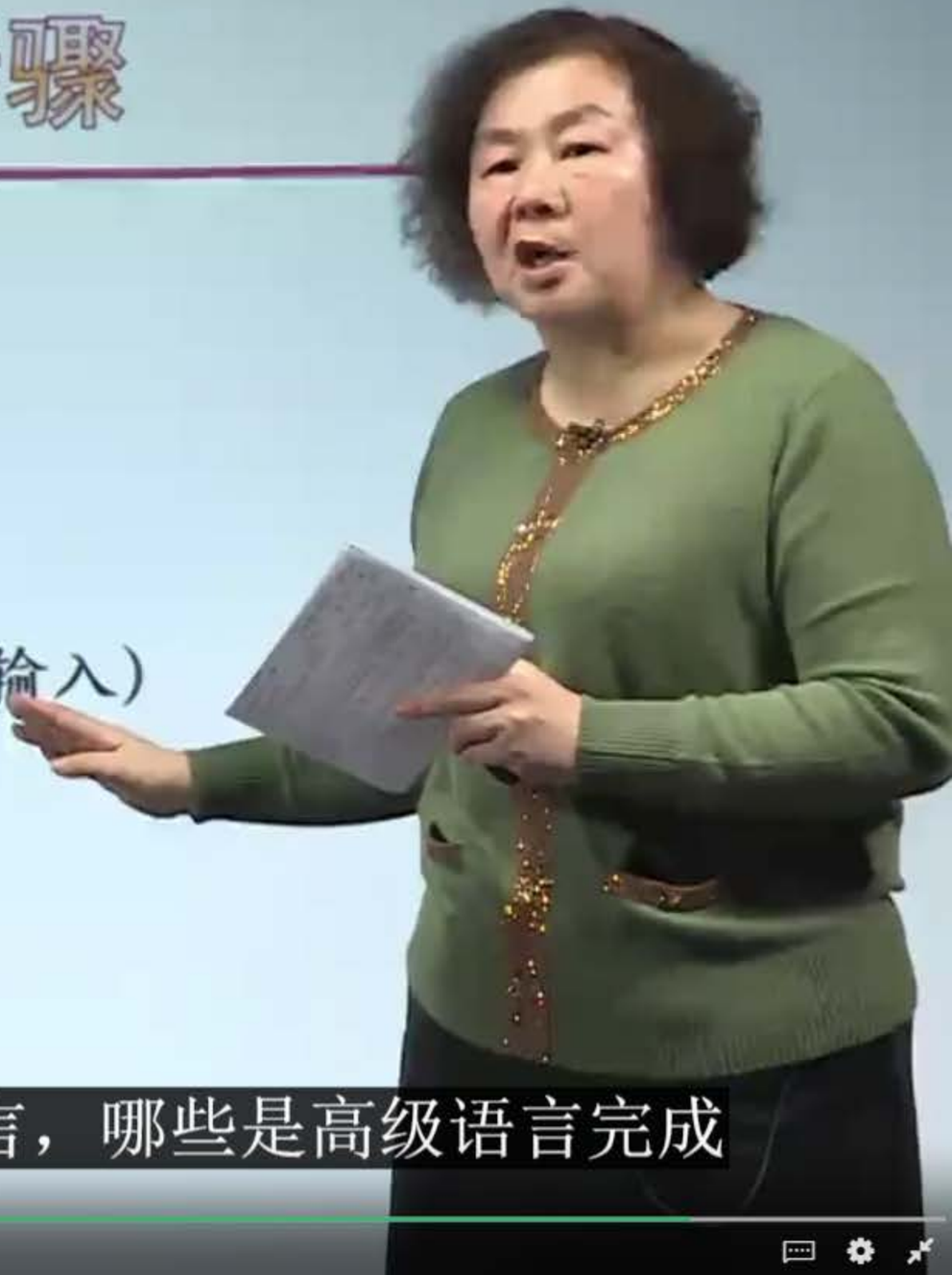


sys_write()执行完后，经过ret_from_sys_call()例程返回用户程序


```
#define SAVE_ALL \
        cld; \
        pushl %es; \
        pushl %ds; \
        pushl %eax; \
        pushl %ebp; \
        pushl %edi; \
        pushl %esi; \
        pushl %edx; \
        pushl %ecx; \
        pushl %ebx; \
        movl $(__USER_DS), %edx; \
        movl %edx, %ds; \
        movl %edx, %es;
```

中断发生后OS低层工作步骤

1. 硬件压栈：程序计数器等
2. 硬件从中断向量装入新的程序计数器等
3. 汇编语言过程保存寄存器值
4. 汇编语言过程设置新的堆栈
5. C语言中断服务程序运行（例：读并缓冲输入）
6. 进程调度程序决定下一个将运行的进程
7. C语言过程返回至汇编代码
8. 汇编语言过程开始运行新的当前进程

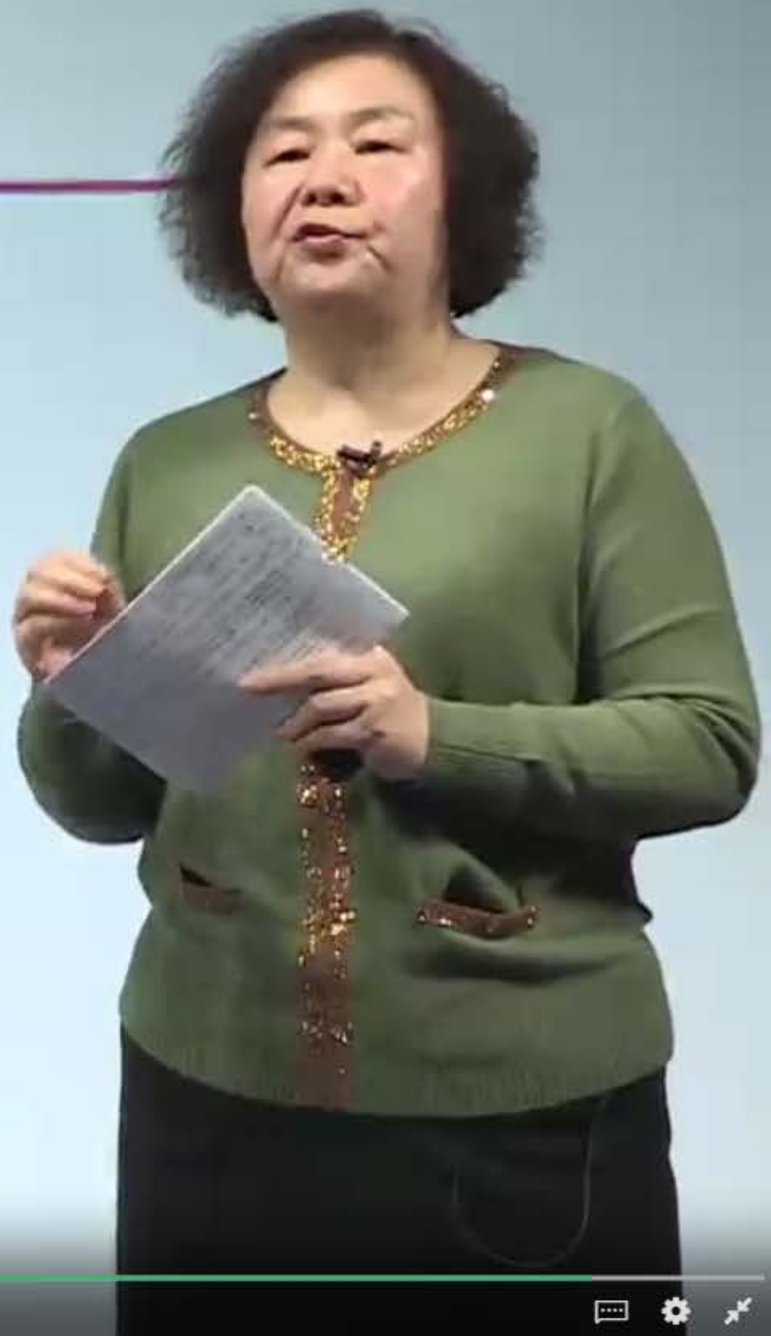


哪些是硬件，哪些是软件，哪些是汇编语言，哪些是高级语言完成

教材第32页的图2-5

本讲重点

- 理解计算机系统的保护机制
  - 掌握处理器状态
  - 掌握特权指令与非特权指令
- 掌握中断/异常机制
  - 掌握中断/异常的基本概念
  - 理解中断/异常机制的工作原理
- 掌握系统调用机制
  - 掌握系统调用设计原理
  - 掌握系统调用执行过程



本周要求

- 重点阅读教材

  第1章相关内容：1.3、1.6

  第2章 第52页 图2-5及说明该图思路的段落

- 重点概念

CPU状态　内核态/用户态　特权指令/非特权指令

中断　异常　中断响应　中断向量　中断处理程序

系统调用　陷入指令　系统调用号　系统调用表

0:00 下面我们再以 Linux 为例，看一看基于 x86 处理器的 Linux 的系统调用是怎么实现的 当然这部分内容呢是希望通过这个例子 加深对系统调用整个过程的一个理解，包括怎么去设计，包括它的执行过程 并不要求大家全搞得非常清楚，因为很多细节我们没有介绍 好我们来看一下 基于 x86 处理器的 Linux 的系统调用的实现 那么当然了，首先我们要选择一条陷入指令 那么在这里呢， Linux 选择了 128 号中断向量，那么这是十进制 那么十六进制表示呢，就是 80 口，是 int $0x80，是这个我们特殊的这个陷入指令 当然除了这个以外，那么 Linux 还利用了 sysenter sysexit 这一套指令，当然我们这里头介绍其中一种方法 那么中断向量表中的门描述符 它是怎么样来初始化的呢？我们来看一下，在系统初始化的时候呢 对于中断描述符表当中的 128 号这个门描述符呢，我们要对它进行初始化 首先呢，先把这个门描述符的 从右边数的 2、 3 两个字节设置成段选择符 然后 0、 1、 6、 7 这 4 个字节设置成了一个偏移量 那么通过这个段选择符和这个偏移量 能够最终使得硬件找到 中断处理程序也就是 system_call() 这个处理程序 那么这一段的设置，如果大家去读代码的话呢，可以在 这条语句当中看到，set_system_gate 80 口指向 system 口，是通过这个 做好设置。 那么在中断描述符当中，门描述符当中呢门的类型是采用什么类型呢？ 那么它的类型采用的是 15，15 类型呢实际上是陷阱门，那么为什么用陷阱门呢？ 是因为在执行系统调用过程中，我们还允许接收中断的话 那我们就用陷阱门进来，就不自动关闭中断了。 那么 特权级设置成多少为好呢？合适呢？ 我们看到，特权级设置为 3 设置为 3。 因为我们知道，在用户态进入到内核态的时候要经过这个门 那么用户态的特权级是 3，R3，对吧，是 3。 那么 在经过这个门的时候，要求当前运行程序 的这个特权级要等于或者高于什么呀？ 要执行的这个代码段。 你要执行的中断处理程序如果它的特权级 不是 3，那么那就是 0，那么 3 比 0 的 级别要低，注意这个数字啊，这 3 比 0 的级别要低，它就进不来了。 所以要保证 用户态能进到核心态，所以这个门描述符一定要是什么呀？是 3，与用户态的 这个特权级是一致的，相同的 这样就允许用户可以通过这个门描述符 那么我们来看看在 Linux 当中到底有哪些系统调用和它的编号呢？这里没列举所有的，列举了几个，我们大致有个印象 比如说 exit ，这个系统调用实际上就是 1 号系统调用 fork 创建进程是 2 号系统调用 read、 write 是 3 号、 4 号系统调用 我们可以看到不同的系统调用都有一个编号，有个感性的认识 好，那么下面，当系统执行过程中 执行到了 int

$0x80 这个指令以后 要做哪些事情呢？我们来看一下。 那么由于特权级发生了改变 所以这时候要切换栈，要不从用户 栈切换到内核栈，怎么切换栈呢？是 CPU 从任务状态段 TSS 表当中 装入新的栈指针，指向内核栈 那么内核栈有了以后，剩下 就是压栈，把一些信息往栈里压。 那么是由硬件 自动依次地把用户栈的信息 SS：ESP 标志状态字，呃，标志寄存器的信息 EFLAGS 还有返回的地址用户态的 CS 和 EIP 寄存器的内容 依次压栈，依次压栈。 压完栈 之后，特别是把 EFLAGS 压完栈之后，就复位 TF 位 然后呢，IF 位保持不变，保持不变。 下面 硬件用 128 在 中断描述符表当中找到了刚才我们初始化好的门描述符 从当中呢取到了段选择符，装到了代码段寄存器 CS 当中 而代码段描述符当中的基地址 和陷阱门描述符当中的偏移，由这两部分就能够 定位我们系统调用的一个总的入口 地址，总的入口地址。当然这个过程中有一些硬件的工作 我们之前介绍过了，这里就简单地说一下 我们再 复习一下刚才的过程。 那我们来看这张图 左边是应用程序，那么应用程序调了个库函数 write 所以应用程序在用户态下调用了一个 C 库的库函数，这里比如说是 write 那么封装后的这个 write ，就在库里封装后的 write 它主要做好的是什么呀？参数的传递工作，然后呢，设置了一个 int $0x80 这条指令，用它呢来产生一次异常。 所以我们看 封装例程当中就变成了若干条参数 推送到寄存器的指令以及 一个 int $0x80 作为陷入的这么一个特殊的指令 好，当执行这条指令的时候呢，我们知道就陷入了内核态，就是陷入后的工作 那么 CPU 执行到 0x80，那么它通过 0x80 号在 IDT 表中，中断描述符表当中找到了对应的服务例程 system_call()，这个是总的，系统调用的总的入口地址，并且调用它 调用它之后我们可以看到，之前啊，在 调用它之前其实已经做好了一些压栈的工作，按照我们刚才叙述，把一些 重要的寄存器、 这个用户栈的信息、EFLAGS 信息和返回地址都压好栈了 那么陷入之后的这个 system_call() 主控程序还要把 eax 再压栈 再把剩余的其它的寄存器的内容呢 压栈，那么叫 SAVE_ALL ，SAVE_ALL ，把它全都压在堆栈里头 压完栈之后再去调用查这个 系统调用表去调用对应的那个内核函数 内核函数，这个内核函数比如说 sys_wirte()

对应的内核函数 所以我们看看就是 system_call() 是将参数 存在内核栈，然后再根据系统编，这个编号调用编号再去查系统调用表，找到 这个系统调用内核函数的入口，入口 再执行这个函数。 执行完了之后，那么 通过执行这个 ret_from_system_call() 就是返回 用户程序，从这返回，返回到用户程序 这就是一个 Linux 的系统调用的一个执行过程 那这里头简单地啊给大家看一下，这个 SAVE_ALL SAVE_ALL 的一段代码。 那么 SAVE_ALL 呢是把其他的一些剩余的寄存器内容压栈 那么这个栈，压完栈之后，栈的状态，栈的这个布局就是这样的 那么前面的信息，栈前面的信息 是硬件压的，然后系统调用号是 这个 system_call() 这个程序推送的，剩下的 从 es 开始，这些呢是 SAVE_ALL 完成的，那么完成之后 就把这个栈增长了，压栈的顺序是这样一个顺序 这是一个小例子，大家看懂这个就可以了 好，那么我们在讲完了这个中断 异常机制还有这个系统调用之后呢，我们再来 归纳一下，在中断发生之后 那么操作系统底层的一些工作的一些主要步骤 那么这张，我们要介绍的内容呢是在教材的 52 页的这个图 2-5。 第一步 硬件压栈主要的内容是程序计数器 PSW 等 硬件从中断向量装入新的程序计数器 汇编语言过程来保存一些寄存器的值 汇编语言过程呢设置新的堆栈 那么 C 语言 中断服务程序呢来运行，可能是读 读文件啊，或者是读盘啊，缓冲输入等等 进程调度程序决定下一个 将要运行的进程，也就是说这件事情完了之后呢，转向进程调度程序转 由它来决定下一个要运行的程序 C 语言过程呢，通过 C 语言过程呢返回到汇编代码然后汇编代码呢，汇编代码的一个过程呢来开始新的 运行新的这个进程。 那么这个过程当中呢其实我们主要要强调 是谁来完成什么样的事。 也就是说前两步是硬件做的工作 后面呢是软件做的事情，但是软件做的事情 呢，我们来看有些呢是必须用汇编语言来完成 有些呢，是可以用 C 语言来完成 那是，因为跟体系结构相关的就要用汇编语言 那么进程调度程序中间还要发挥它的作用 所以从这样一个例子当中呢，我们既 把刚才的过程又重复了一遍，更多的呢我们是通过这个里头看到了这个过程当中 哪些是硬件，哪些是软件，哪些是汇编语言，哪些是高级语言完成 那么本讲的内容呢其实就讲了三件事 第一件事呢，就讲了 CPU 的状态，这是计算机系统的一种保护机制 第二件事情呢，讲了中断/异常机制 这是硬件的一个机制。 第三个呢，是讲了系统 调用机制，是利用了硬件机制完成一个向 用户提供服务的这么一个接口 回去之后呢希望大家呢能够去阅读相关的教材 1.3、 1.6，那么还有呢第 2 章的第 52 页的这张图，以及

说明该图思路的一些段落 本章的重点概念呢有以下一些： CPU 的状态 内核态/用户态，特权指令/非特权指令 中断，异常，中断响应，中断向量 中断处理程序，系统调用，陷入指令 系统调用号和系统调用表。希望大家把这些重要的概念掌握了 那么今天的课程就讲到这里，谢谢大家