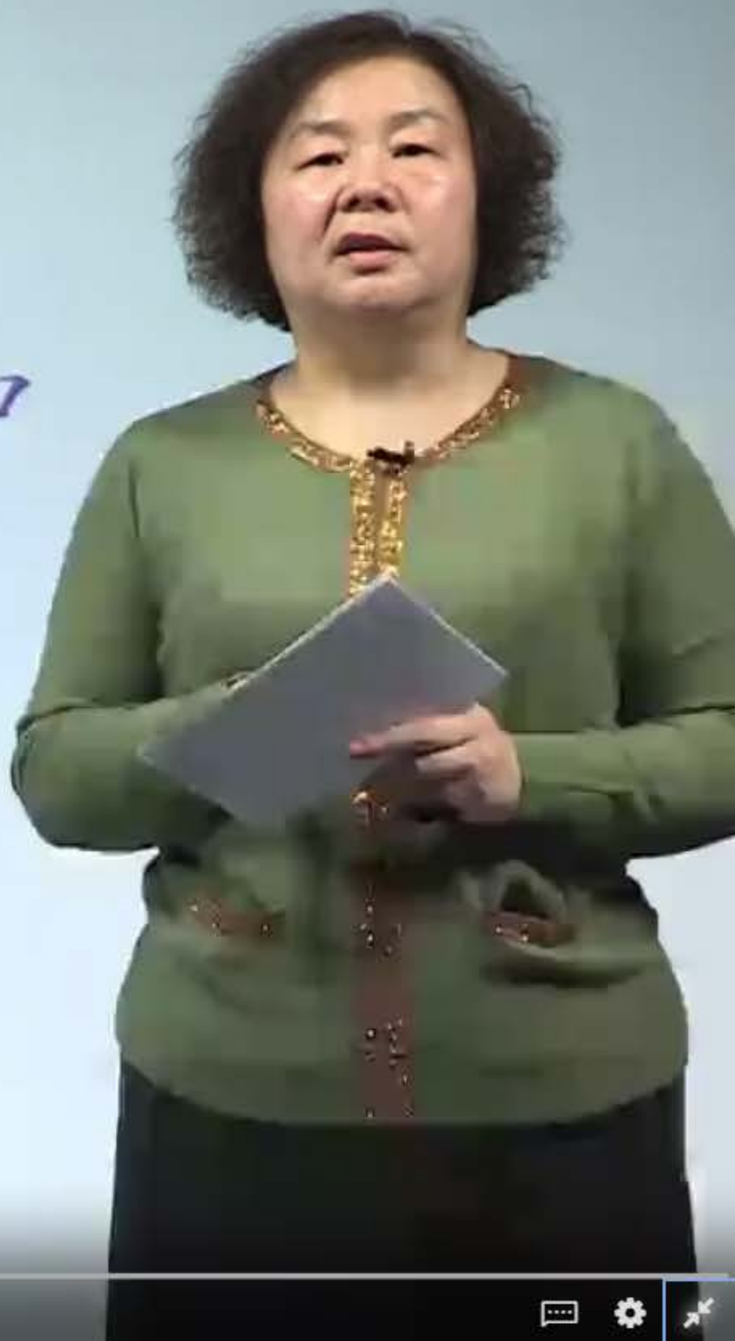
操作系统向用户程序提供的接口

系统调用机制



系统调用(SYSTEM

Linux操作系统提供多少个系统调用?

系统调用是什么?

系统调用的作用

典型系统调用举例

系统调用：用户在编程时可以调用的操作系统功能

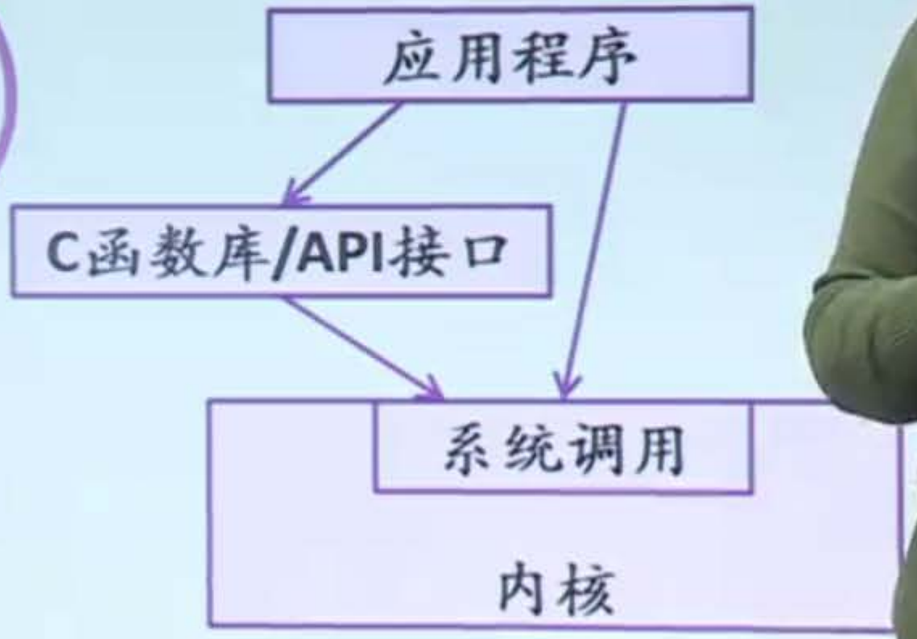
- 系统调用是操作系统提供给编程人员的唯一接口
- 使CPU状态从用户态陷入内核态

每个操作系统都提供几百种系统调用（进程控制、进程通信、文件使用、目录操作、设备管理、信息维护等）

系统调用、库函数、API、内核函数

系统调用与C库函数/API函数的关系

系统调用与内核函数的关系



那么这就是系统调用库函数、 API 和内核函数的一个 简单的区别。

系统调用机制设计与执行过程

那么下面我们来设计一下系统调用机制 来看一下执行系统调用的过程是什么样的

4:58 / 15:35

系统调用机制的设计

中断/异常机制

支持系统调用服务的实现 ①

选择一条特殊指令：陷入指令(亦称访管指令)

引发异常，完成用户态到内核态的切换 ②

系统调用号和参数

每个系统调用都事先给定一个编号(功能号) ③

系统调用表

存放系统调用服务例程的入口地址 ④

所以在设计一个系统调用的时候呢，我们要有这样一些要素

参数传递过程问题

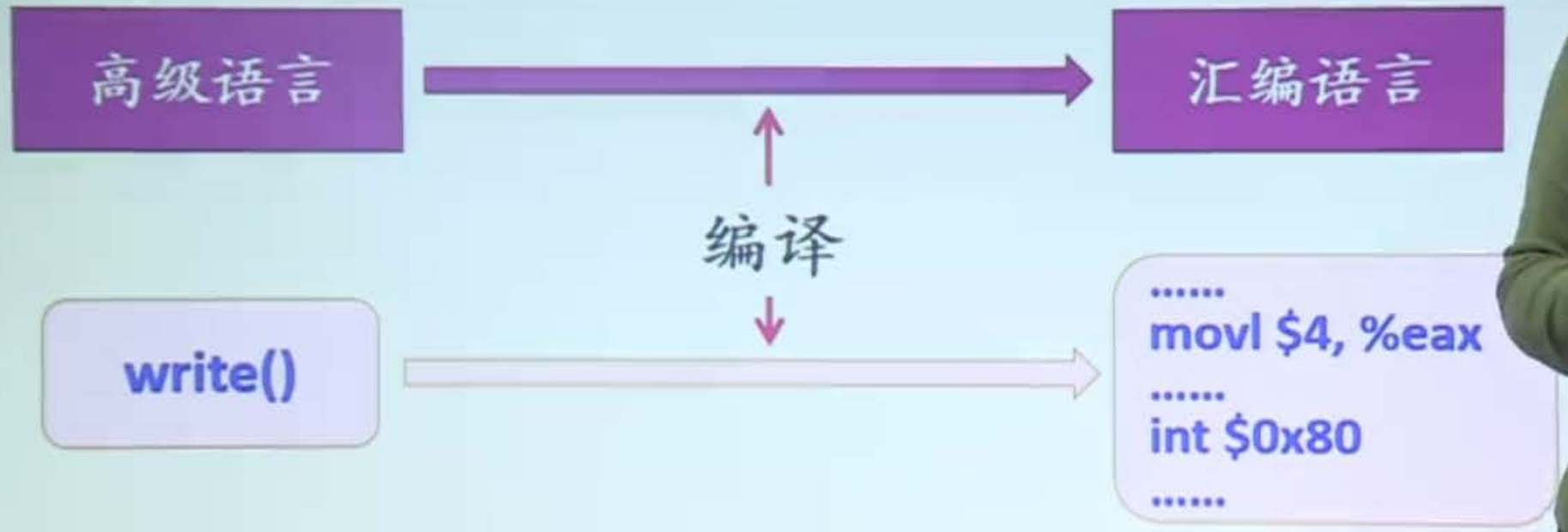
- 怎样实现用户程序的参数传递给内核？

常用的3种实现方法：

- 由陷入指令自带参数：陷入指令的长度有限，且还要携带系统调用功能号，只能自带有限的参数
- 通过通用寄存器传递参数：这些寄存器是操作系统和用户程序都能访问的，但寄存器的个数会限制传递参数的数量
- 在内存中[illegible]传递参数

通常呢，大部分情况下，我们会采用第二种方案

系统调用举例(1/3)



好，那么我们进一步来讨论这个例子。

系统调用举例(2/3)

```
#include <unistd.h>
int main(){
    char string[5] = {'H', 'e', 'l', 'l', 'o', '!', '\n'};
    write(1, string, 7);
    return 0;
}
```

输出结果：Hello!

高级语言视角

那么我们来看一下， 这段 C 代码，编译成了汇编指令之后是个什么样子的。

系统调用举例(3/3)

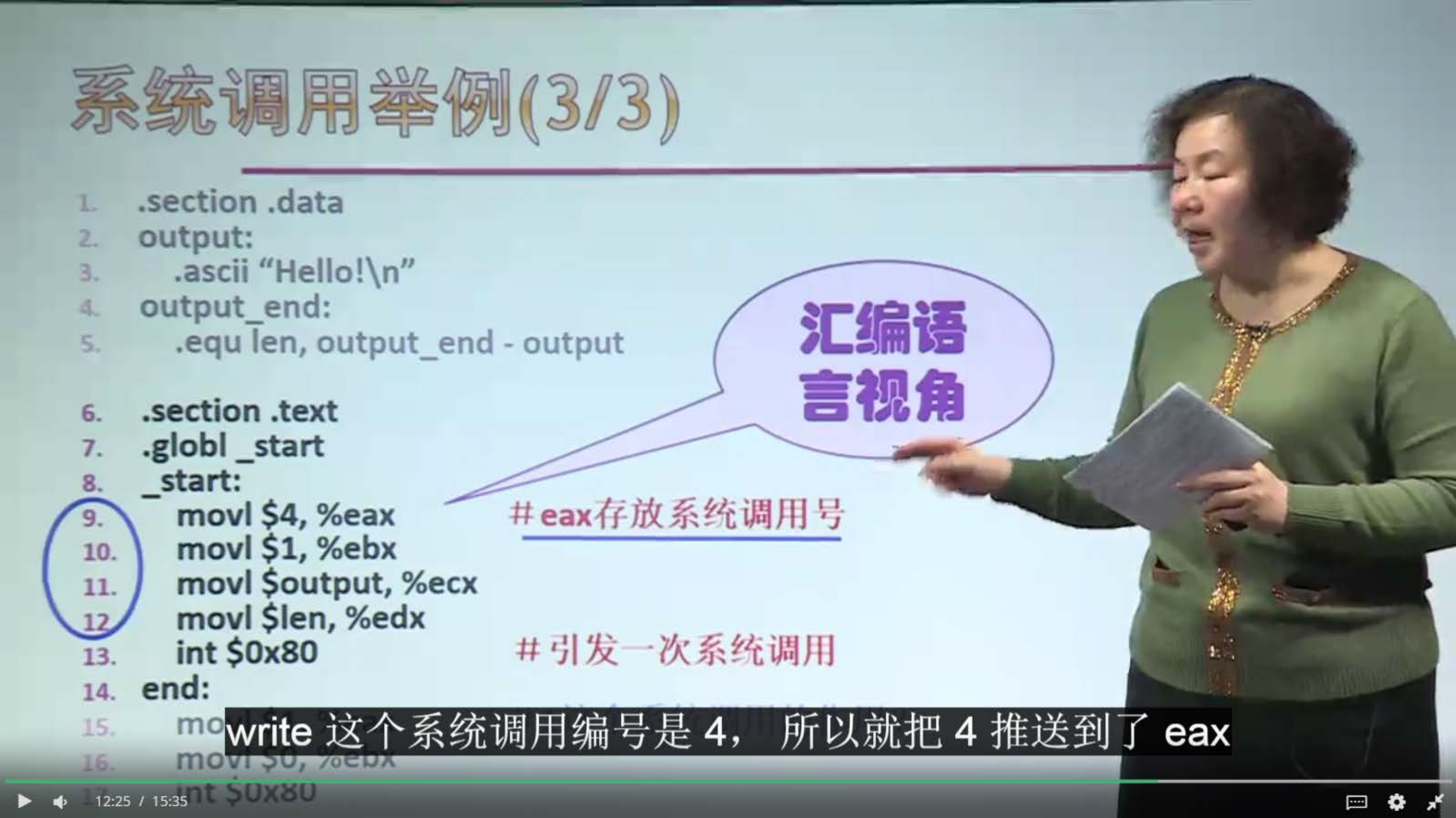
```
1.  .section .data
2.  output:
3.      .ascii “Hello!\n”
4.  output_end:
5.      .equ len, output_end - output

6.  .section .text
7.  .globl _start
8.  _start:
9.      movl $4, %eax
10.     movl $1, %ebx
11.     movl $output, %ecx
12.     movl $len, %edx
13.     int $0x80
14. end:
15.     mo
16.     movl $0, %ebx
17.     int $0x80
```

汇编语言视角

#eax存放系统调用号

#引发一次系统调用

write 这个系统调用编号是 4， 所以就把 4 推送到了 eax



系统调用举例(3/3)

```
1.  .section .data
2.  output:
3.      .ascii “Hello!\n”
4.  output_end:
5.      .equ len, output_end - output

6.  .section .text
7.  .globl _start
8.  _start:
9.      movl $4, %eax
10.     movl $1, %ebx
11.     movl $output, %ecx
12      movl $len, %edx
13.     int $0x80
14. end:
15.     movl $1, %eax
16.     movl $0, %ebx
17.     int $0x80
```

汇编语言视角

#eax存放系统调用号

#引发一次系统调用

#1这个系统调用的作用？

系统调用的执行过程

当CPU执行到特殊的陷入指令时：

- 中断/异常机制：硬件保护现场；通过查中断向量表把控制权转给系统调用总入口程序
- 系统调用总入口程序：保存现场；将参数保存在内核栈里；通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- 执行系统调用例程

那这里头我们可以看到，其实有一个表是中断向量表，这是硬件来访问的，然后

15:11 / 15:35

0:00 下面呢，我们来介绍系统调用机制 那么这是操作系统向用户程序提供的接口 什么是系统调用？ 系统调用有什么作用？有哪些典型的系统调用呢？ 我们来看一下。 系统调用呢 是用户在编程时可以调用的操作系统功能 系统调用是一个简称 它的全称应该是操作系统功能调用，简称系统调用 那么有了系统调用 操作系统就可以给编程人员，提供了一个提出服务请求的这么一个接口 通过这样一个系统调用 使得 CPU 的状态从用户态陷入了内核态 前面我们已经介绍过了，当用户要做一些事情，而这些事情呢 是需要用特权指令来完成的。 因此用户不能直接执行特权指令 所以就通过了一个接口向操作系统提出请求 由操作系统来完成这个过程。 那么就需要这个接口，通过这个接口 从用户态陷入内核态 这就是操作系统当中系统调用 所起的作用。 当然了有很多的系统调用 每个操作系统呢都提供了几百种系统调用 那么我们来简，简单来看一下啊，进程控制类 比如说创建进程啊，撤销进程 进程的通信类，那么可以一个进程给另外一个进程发消息 还有文件使用的类。 比如说创建 撤销、 打开、 关闭、 读、 写文件 还有对目录的操作、对设备的管理 以及一些信息维护，那么这些呢都是 典型的系统调用的例子。 那这里呢也给大家提一个小小的问题 那么 Linux 操作系统当中，提供了多少个系统调用呢？ 大家不妨呢，根据不同的版本去查一下，看看某个版本提供了多少个系统调用 那这里的我们要区分系统调用 库函数、 API、 内核函数这样几个不同的概念 那么系统调用和我们说的 C 库函数 或者是 API 函数有什么关系呢？那么系统调用和内核函数又有什么关系呢？ 我们来看这张图。 那么应用程序 可以直接调系统调用 但是呢通常情况下，应用程序都是 通过了 C 函数库或者是 API 的接口来间接地调用系统调用 那么在操作系统内核当中，提供了很多的内核函数 那么这些内核函数呢，经过了封装，经过了封装 实际上呢把它提供到了 C 函数库，或者是 API 接口 所以系统调用呢 对内，对于内核而言，那么内核函数就是这个系统调用的处理程序，处理程序 而这些处理程序呢通过封装 在 C 函数库或者 API 接口呢提供给用户来使用 但是呢，C 函数库里头或者是 API 接口里头还有一些函数 它们呢不是系统调用，它们就是一些普通的函数在完成一些功能 而且一些函数 通过系统调用对应到了多个内核函数 当然也可能是某一个函数 通过系统调用呢对应内核的一个函数 这样呢都

是不太一样的。 内核函数当中 也有一些函数呢是不开放给用户使用的 那么这就是系统调用库函数、 API 和内核函数的一个 简单的区别。 那么应用程序往往，大部分情况下是通过调用函数 调用函数。 那么这个函数执行过程中呢再去 变成系统调用来进入内核来完成，基本上是这样一个情况 那么下面我们来设计一下系统调用机制 来看一下执行系统调用的过程是什么样的 那么我们要设计 系统调用，首先 我们要利用硬件给我们提供的支持 就是中断异常机制 通过这个机制可以支持系统调用服务的一个实现。 所以我们要利用硬件的这个机制 那么然后呢，我们要选择一条特殊的指令 那么指令用作我们的陷入指令，也称之为访管指令 那么这条指令呢作用呢，是通过这条指令的执行呢引发一个异常 完成从用户态到内核态的切换工作 那我们看到只有一条指令 也就是说所有的系统调用，都是通过这条指令来进入内核的 在中断向量表或中断描述符表当中 有一行专门用于系统调用 那么在初始化，在操作系统初始化的时候，要把这，这个 表的这一项设置好。 然后呢操作系统呢要为 每一个系统调用事先先给它一个编号，确定一个 编号，有的时候呢叫系统调用号，有的时候呢叫功能号 那么为什么要有编号呢？因为我们刚才说的所有的 系统调用都通过这一条指令进来 那么到底哪一个系统调用呢？我们要通过编号来区分 那么每个系统调用其实还有不同的参数 有的系统有参数，有的系统调用没有参数，所以呢我们还要设计相应的参数 那么这些工作除了操作系统的设计之外 还需要编译器来帮忙，那么编译器呢 会把这个封装的系统调用把它展开 展开的话呢，在过程中呢生成这条特殊的 陷入指令，以及这些参数的 推送寄存器的这些指令，要生成这些 这是编译器要在这过程当中，也要参与进来 那么每一个系统调用 其实都有一段内核函数，或者是一段代码来对应 那么怎么样找到对应的这个内核函数呢？我们就需要设计一张系统调用表 这张表就把系统调用的各项服务的入口地址 填在这张表里头，那这张表也是在这个初始化的时候 设置好了。 所以在设计一个系统调用的时候呢，我们要有这样一些要素 那么这个过程中呢，我们还有一件事情要说明一下 就是参数传递的这个问题，那么 用户程序在执行的时候呢，是用户栈。 那我们知道 一般的函数调用呢是通过这个栈来传递参数的 但是现在我们的面临的问题呢是用户和系统 那么用户程序不能够把它的这些参数呢，推到系统栈里头去，这是不允许的 所以我们现在要解决的问题呢，就是怎么样来实现用户 程序把它的参数，调用函数的参数传递给内核 通常呢有三种方法，可以由陷入指令自带参数 也可以通过通用寄存器来传递

参数，因为这些通用寄存器是 操作系统和用户都能访问的 那么也还可以在内存中开辟一个专门的 专用的区域来传递参数。 通常呢， 大部分情况下，我们会采用第二种方案 通过通用寄存器来传递参数 但是由于寄存器的个数有限 会限制传递参数的数量 我们举一个 例子来说明系统调用的这样一个过程 那么高级语言编写程序通过了编译之后呢，比如说变成了汇编 语言，好，我们在高级语言当中调用了一个库函数 write 它和系统调重名，经常是封装成重名的。 那么，经过编译之后呢，实际上是要编译成这样一个汇编指令。 那么要把 write 的这个函数对应的系统调用号，推到寄存 器里头，要在这里安排一条特殊的陷入指令。 好，那么我们进一步来讨论这个例子。 这是 C 代码，我们从高级语言 的角度来看这段代码，调用了一个函数 write。 write 有三个参数：第一个参数是 1 ，表示的是要把结果送到标准输出设备上；然后，送的内容是放在了字符串里头。 长度是 7，就是 write 这个函数表达了这样一层含义，有三个参数。 还有一个 return 也是一个 函数，它也是一个系统调用，啊。 那么我们来看一下， 这段 C 代码，编译成了汇编指令之后是个什么样子的。 我们首先要说明一下，那么上述的代码，刚才的 C 代码 把它编译之后，变成了汇编的代码， 现在的系、 当前的系统当中，真实的这个编译器 编译的结果和我们这个显示的内容呢，有点不太一样。 主要的原因是，在这个过程当中 C 做了很多层的封装， 啊，所以这样的话呢，结果可能不太一样。 但是呢，这一部分， 和系统调用相关的这一部分， 和我们上、 上面的那个 C 代码的那个相关的调用呢， 是比较一致的，我们只是一个示意啊。 不是、不完全是一个真实的编译后的内容，但是 主要的这个步骤是一样的，我们来看一下这几个步骤： 那么这几个步骤实际上呢，就是把一些数据推送到寄存器里头。 那么这些数据包括什么呢？首先，先包括了系统 调用的编号，而系统调用编号，我们 write 这个系统调用编号是 4， 所以就把 4 推送到了 eax 寄存器， 这个是规定的，eax 寄存器存放了系统调用的编号。 剩下的 write 的三个参数呢，依次送到不同的寄存器。 然后在这里安排了一条特殊的指令，就是我们说的那个陷入指令。 通过这个陷入指令呢，在执行这段代码的时候呢，会引发一次系统调用。 我们刚才看到，在刚才那段代码当中，还有一个系统调用，我们再来看一下，啊。 那这个系统调用，它的作用是什么呢？它的作用是返回、 退出。 那么退出的系统调用编号呢是 1 号，1 号系统调用。 那么它的参数呢是 0，返回 0，然后 也安排了一个特殊的陷入指令，所以， 只要是系统调用，都

是这样一个基本的过程。 好，那么，我们知道了 有这么一条特殊的指令，当执行这条指令的时候，会引发一个系统调用。 那么系统调用的执行过程呢，我们来看一下。 当 CPU 执行到了特殊的 陷入指令的时候，那么各个部件、 硬件 部件和软件就开始工作了，首先呢是 硬件的中断／异常机制工作，它的工作呢， 保存现场；查找中断向量表， 并且把 CPU 的控制权转交给 中断处理程序，而这个中断处理程序，因为我们是 系统调用就是叫做系统调用的一个总入口程序。 因为我们知道所有的系统调用，都是通过这个中断向量进来， 都是执行这个总的入口程序，总控程序，可以看成是。 然后下面，就是这个系统调用的总控程序执行了， 它呢，也要保存现场，然后呢，把参数 保存在内核的堆栈当中，下面呢它要去查找另外一张表 系统调用表，把控制权交给 对应的内核函数，或者是系统调用的处理程序。 所以要查这张表。 接着就去执行系统调用的过程， 然后呢，再恢复现场，返回用户程序。 那这里头我们可以看到，其实有一个表是中断向量表，这是硬件来访问的，然后 主要的作用是找到系统调用的总控程序。 还有一张表呢，是系统调用表。 那么这是总控程序里查这张表，根据 功能号，根据系统调用编号，来把 CPU 控制权转交给对应的内核函数或者是系统调用处理程序。