《openEuler内核编程》

课程讲稿

第5章 第5讲

系统调用

软件所制

第五章 第5讲 系统调用

**学时：1**学时

**教学目的：**在之前对中断机制的了解上，学习使用中断机制来完成的系统调用机制……

**课程时间线：**

章节简介

系统调用的意义

初始化

系统调用的处理过程

5min

15min

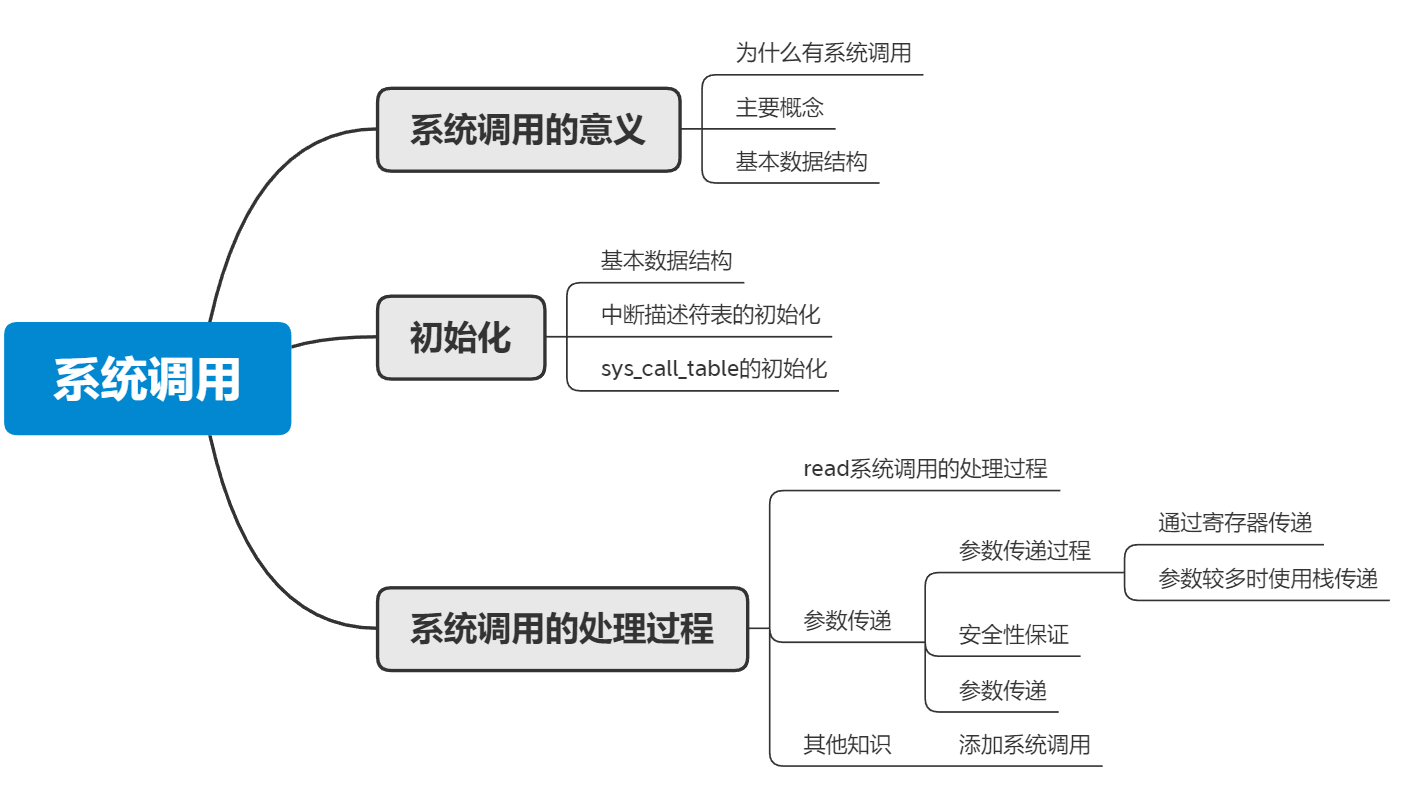
35min

45 min

**课外参考读物：**

《深入理解LINUX内核》

**知识框图：**

****

**PPT讲稿：**

1. 到今天我们已经学习完了中断机制的一大半内容，今天我们再来看看一个和中断机制紧密结合的部分——系统调用。
2. 为什么要讲系统调用呢。因为系统调用是依托中断机制来实现的，通过学习系统调用我们可以对中断机制形成更深的了解。
3. 本节中我们将分为三块内容来介绍系统调用，分别是系统调用的意义、初始化和系统调用的处理过程，我们先来看看下系统调用的意义这一部分。
4. 按照惯例，在学习一个东西之前我们先来了解一下为什么有这个东西。这节课我们学习Linux中的系统调用，为什么会有系统调用呢？首先我们思考一下，我们写好了应用程序肯定要操作各个计算机资源，但是如果任由用户进程随意操作这些计算机资源可能会造成混乱甚至崩溃。因此Linux为用户进程提供了系统调用来替用户操作这些计算机资源，并且通过内核对这些操作进行管理，例如权限控制、互斥机制等。为了向用户进程提供这些功能，就要提供一套接口来让用户进程使用内核的这些功能。透过这些接口，应用程序就可以访问硬件设备和其他操作系统资源了。
5. 我们先来看看一点典型的系统调用fork是怎么工作的。一个用户进程需要fork一个新进程，它会在内核里面查找是否有足够的pid来分配给新进程，如果有则分配，否则fork失败。Fork的一个功能就是防止进程过度创建新的进程。试想一下如果fork不是系统调用，而是用户空间实现的函数的话，用户进程可以随意创建新进程，这时pid的数量急剧上涨甚至超过了系统所能承受的最大进程数。这就会带来ppt上的后果，甚至还有其他更严重的后果。
6. 系统调用的意义其实就是系统调用的功能
   1. 系统调用为用户空间提供了一种硬件的抽象接口，也就是上层的应用软件不用去管底层的硬件是如何工作的，软件和硬件相结合的部分就由操作系统去完成。
   2. 系统调用保证了系统的稳定和安全。
   3. 系统调用为用户进程与操作系统之间提供了一种交流方式
7. 在知道系统调用的意义之后，我们正式开始系统调用的学习，首先从涉及的一些概念开始。
   1. 系统调用号：内核为每种系统调用都分配了对应的号码，这个号码在用户调用库函数的时候由库函数给出，传入内核。也就是说应用程序可以直接使用封装好的库函数来使用系统调用，当然用户也可以直接给出系统调用号来使用系统调用，只不过这个过程稍微繁琐。
   2. 系统调用表：内核为所有系统调用函数构建了一个表，表项是分别指向这些函数的入口地址，系统调用号就作为下标来找到对应的系统调用函数
   3. 系统调用处理函数：内核为不同的系统调用实现的处理函数。
8. 系统调用的一些基本情况我们已经了解了，下面开始学习系统调用最开始的部分——初始化。
9. 看看系统调用机制里面的数据结构。在这里我们只介绍一个最重要的数据结构——系统调用表。X86下的系统调用表定义在这个文件下。可以看到，在32位和64位下的定义都是一样的，这说明在这两种架构下系统调用的实现方式是一样的。初始化的时候默认创建了\_\_NR\_syscall\_max个系统调用，但是将所有表项都指向了sys\_ni\_syscall函数。\_\_NR\_syscall\_max在各个版本中都不同，总体来说版本越高，系统调用数目越多，而在最新的Linux4.19里面达到了386。在这里要注意一下sys\_call\_table和idt\_table是两个不同的概念！！！
10. 下面通过对中断描述符表以及系统调用表的初始化的学习我们就可以知道sys\_call\_table和idt\_table在哪里不同了。在初始化过程中，将def\_idts里的多项gate\_desc拷贝到idt\_table里面，这里的def\_idts是内核程序员写好的一段idt表内容，直接用于填充idt表。而def\_idts定义了idt\_table里第0x80（第128项）项，这个写法非常有意思。
11. idt\_table里的第0x80（128）项为一个入口地址，它指示了中断处理函数的入口地址，在这个用汇编语言写的函数里面，完成了如下的两步：
    1. 首先保存了各寄存器的值
    2. 调用do\_int80\_syscall\_32

然后会根据系统调用号，也就是eax寄存器里的值在来调用对应的系统调用函数

1. 与idt表类似，系统调用表的默认填充数据已经由汇编代码写好放在了内核的数据区。当然我们还可以用重新编译内核或者使用模块机制的方法来向系统调用表添加自己新实现的系统调用。简单的思路就是首先取得系统调用表在内核中的地址，然后把其中未使用的某一项（下标即为系统调用号）指向自己实现的系统调用处理函数。而事实上Linux的程序员也是这么实现这个功能的。
2. 初始化完成后，就要正式开始使用系统调用了，所以最后我们来看看系统调用的处理过程。
3. 下面我们来以read这个系统调用为例来看看系统调用的过程。当用户程序要读取一个文件时，最终肯定是使用内核提供的read系统调用，而为了使用方便C函数库将这个系统调用封装成了一个便于使用的函数read（），直接供用户程序使用。而这个read（）函数做的就是2.这个活：首先准备系统调用函数sys\_read所需要的参数，写入栈中或者寄存器中，其中系统调用号写入了eax中，最后调用int 0x80发送中断进入内核。
4. 在read函数调用int 0x80后进入内核，内核查idt\_table的第0x80项，进入中断处理函数，在这个函数中，首先将各寄存器的值入栈（保存所要传递的参数），然后调用do\_int80\_syscall\_32函数。这个函数可以理解为一个通用的系统调用处理函数，因为所有系统调用的执行过程都要先通过这个函数的处理。
5. 接下来do\_int80\_syscall\_32函数要做的事情就和系统调用表的功能息息相关了。它会根据eax中的值（系统调用号，此时已经保存在栈中），查询sys\_call\_table表，找到对应的系统调用函数sys\_read的入口地址，跳到sys\_read执行，并且将之前保存在栈中的参数作为sys\_read的参数。
6. 最后就是系统调用的结束阶段了，系统调用其实就是一种中断，所以系统调用的返回过程就是一种中断的返回过程。在这里我们思考一个问题：系统调用处理函数能不能调用库函数？例如使用printf，系统调用是库函数实现的基础，因此必须要先有系统调用才能实现库函数，不能本末倒置。
7. 在系统调用的执行过程中我们要重点注意一下参数的传递过程，一般情况下参数是通过寄存器传递的，例如一些数据寄存器和eax，但参数数目过多时（例如结构体、数组），就只能通过指针来传递了。但是，系统调用的参数是用户态下给出的，而使用是在内核态下，因此一些安全因素我们必须就要考虑在内。
8. 正因为参数传递的不安全性，内核在接收一个用户空间的指针之前必须保证如下的几点：
   1. 指针指向的内存区域属于用户空间，进程决不能误导内核去读用户空间的数据。
   2. 指针指向的内存区域在进程的地址空间里，进程决不能误导内核去读其他进程的数据。
   3. 如果是读，该内存应被标记为可读。如果是写，该内存应被标记为可写。进程决不能绕过内存访问限制。

故我们简单总结一下，参数传递的原则：内核无论何时都不能轻易地接受来自用户空间的指针!

1. 对于这些安全问题，当然有方法来尽量避免。这个方法就叫做参数验证，Linux提供如下的两个函数来保证安全性：
   1. copy\_to\_user，从内核拷贝数据到用户空间。
   2. copy\_from\_user，从用户空间拷贝数据到内存。

这两个函数都会在内部会对合法性作一些验证，保证传入内核空间或传出用户空间的参数合法。

1. 最后我们来看看如何添加自己实现的系统调用。在前面我们已经简要介绍了添加系统调用的涉及思想，也就是以下三步：
   1. 1. 实现新的系统调用函数syscall\_xxx(...)
   2. 2. 修改sys\_call\_table，选择为被分配且数值小于\_\_NR\_syscall\_max的项，指向刚实现的syscall\_xxx(...)
   3. 3. 编译内核