# 系统调用：

Unix系统通过向内核发出系统调用（system call）实现了用户态进程和硬件设备之间的大部分接口。

## PUSIX API和系统调用：

1. 应用编程接口（API）与系统调用之不同：前者只是一个函数定义，说明了如何获取一个给定的服务；后者通过软中断向内核态发出一个明确的请求。
2. Unix系统给程序员提供了很多API的库函数。libc的标准C库所定义的一些API引用了封装例程，其唯一目的就是发布系统调用。通常情况下，每个系统调用对应一个封装例程，而封装例程定义了应用程序使用的API。
3. 系统调用属于内核，而用户态函数不属于内核。

## 系统调用处理程序及服务例程：

1. 当用户态的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。最终会跳转到所谓系统调用处理程序的汇编语言函数。
2. 系统调用处理程序与其他异常处理程序的结构类似，执行以下操作：
3. 在内核栈保存大多数寄存器的内容（这个操作对所有系统调用都是通用的，并用汇编语言编写）。
4. 调用名为系统调用服务例程的相应C函数来处理系统调用。
5. 退出系统调用处理程序：用保存在内核栈中的值加载寄存器，CPU从内核态切换回到用户态（所有的系统调用都要执行这一相同的操作，该操作用汇编语言代码实现）。
6. 为了把系统调用号与相应的服务例程关联起来，内核利用了一个系统调用分派表。这个表存放在sys\_call\_table数组中，有NR\_syscalls个表项（在Linux2.6.11内核中是289），第n个表项包含系统调用号为n的服务例程的地址。

## 进入和退出系统调用：

1. 调用和退出系统调用的两种方式：P415
2. 通过int $0x80指令发出系统调用 P415
3. system\_call()函数：P416
4. sysenter指令提供了一种从用户态到内核态的快速切换的方法。P418

## 参数传递：

1. 普通的C函数的参数传递是通过把参数值写入活动的程序栈（用户态栈或内核态栈）实现的。因为系统调用是一种横跨用户和内核两个大陆的特殊函数，所以既不能使用用户态栈也不能使用内核态栈。在发出系统调用之前，系统调用的参数被写入CPU寄存器，然后在调用系统调用服务例程之前，内核把存放在CPU中的参数拷贝到内核态堆栈中，这是因为系统调用服务例程是普通的C函数。

## 异常表：

1. 决定异常表的来源的关键在于内核使用有限的范围访问进程的地址空间。只有少数的函数和宏用来访问进程的地址空间；因此，如果异常是由一个无效的参数引起的，那么引起异常的的指令一定包含在其中的一个函数中或又展开的宏产生。这些对用户空间的寻址的指令数量是非常少的。
2. 因此，把访问进程地址空间的每条内核指令的地址放到一个叫异常表（exception table）的结构中并不用费太多功夫。当在内核态发生缺页异常时，do\_page\_fault()处理程序检查异常表：如果表中包含产生异常的指令地址，那么这个错误就是由非法的系统调用参数引起的（地址放在表中的指令，都是系统调用指令），否则，就是由某一更严重的bug引起的。
3. Linux定义了几个异常表。主要的异常表在建立内核程序映像时由C编译器自动产生。它存放在内核代码段的\_\_ex\_table节，其起始与终止地址由C编译器产生的两个符号\_\_stat\_ex\_table和\_\_stop\_ex\_table来标识。
4. 此外，每个动态装载的内核模块都包含有自己的局部异常表。这个表是在建立模块映像时由C编译器自动产生的，当把模块插入运行中的内核时，把这个表装入内存。
5. 每一个异常表的表项是一个exception\_table\_entry结构，它有两个字段：

insn 访问进程地址空间的指令的线性地址

fixup 当存放在insn单元中的指令所触发的缺页异常发生时，fixup就是要调用的汇编语言代码的地址

// 修正代码由几条汇编指令组成，用以解决由缺页异常所引起的问题。这个指令序列强制服务例程返回一个出错码给用户态进程。这些指令通常在访问进程地址空间的同一函数或宏中定义，由C编译器把它们放置在内核代码段的一个叫做.fixup的独立部分。

1. search\_exception\_tables()函数用来在所有异常表中查找一个指定地址 P429

## 生成异常表和修正代码：

## 内核封装例程

1. 系统调用主要由用户态进程使用，但也可以被内核线程调用，内核线程不能使用库函数函数。为了简化相应封装例程的声明，Linux定义了7个\_sysycall0到\_\_syscall6的一组宏。
2. 每个宏名字中的数组0~6对应着系统调用所用的参数个数（系统调用号除外）。P431