1. 什么是系统调用

所谓系统调用是指操作系统提供给用户程序调用的一组“特殊”接口，用户程序可以通过这组“特殊”接口来获得操作系统内核提供的服务。例如用户可以通过进程控制相关的系统调用来创建进程、实现进程调度、进程管理等。

在这里，为什么用户程序不能直接访问系统内核提供的服务呢？这是由于在 Linux 中，为了更好地保护内核空间，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，在逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。 但是，在有些情况下，用户空间的进程需要获得一定的系统服务（调用内核空间程序），这时操作系统就必须利用系统提供给用户的“特殊接口”——系统调用规定用户进程进入内核空间的具体位置。进行系统调用时，程序运行空间需要从用户空间进入内核空间，处理完后再返回到用户空间。

二．应用编程接口(API)与系统调用

API只是定义了一个函数，二系统调用利用软中断机制向内核发出服务请求。从用户编程者看，API和系统调用都具有函数名，参数以及返回代码的格式。从内核观点看，系统调用属于内核，API并不属于内核部分。

三．实现系统调用的需求

1. 当用户进程调用一个系统调用时，CPU会切换到内核态执行内核函数即(跳转到系统调用处理程序），因而需要一套用户态到内核态的切换的机制。

2. 因为内核能提供丰富的资和服务，所以必定会提供很多不同的系统调用，因此进程必须传递一个系统调用号作为参数来识别所需的系统调用，如何传递系统调用号也是系统调用需要考虑的一个问题。

3. 由于系统调用具有函数一样的格式，有些内核服务需要传递参数，因而参数的传递也是系统调用需要考虑的问题。

4. 同上面类似，系统调用的返回值传递方式。

四．系统调用的实现

1. 用户态到内核态的切换

Linux利用软中断的方式实现上述转换,利用指令int $0x80 进入内核向量128（0x80）的中断门函数: system\_call（系统调用处理程序）

在内核初始化的时候建立中断向量0x80的中断描述表项：set\_system\_gate(0x80, &system\_call)

当用户进行系统调用时触发中断向量0x80从而切换到内核态。

2. system\_call函数：系统调用号和参数的传递

我们知道传统函数通过栈的方式传递参数，系统调用与用户函数具有相同的格式，那么我们的系统调用号和参数也能通过用户栈方式传递吗？不能，因为用户程序和内核程序使用不同的栈，用户栈和内核栈是相互隔离的，系统调用属于内核程序，必须使用内核栈。直接操作两个栈数据拷贝是非常复杂的事情，因而也不适合直接从用户栈到到内核栈的拷贝。

Linux利用寄存器方式传递系统调用号和参数

存放系统调用号：eax

存放参数：ebx, ecx, edx, esi, edi, epb

ENTRY(system\_call)

RING0\_INT\_FRAME # can't unwind into user space anyway

pushl\_cfi %eax # save orig\_eax

SAVE\_ALL #(1)

GET\_THREAD\_INFO(%ebp) # system call tracing in operation / emulation

testl $\_TIF\_WORK\_SYSCALL\_ENTRY,TI\_flags(%ebp)

jnz syscall\_trace\_entry

cmpl $(nr\_syscalls), %eax #(2)

jae syscall\_badsys

syscall\_call:

call \*sys\_call\_table(,%eax,4) #(3)

movl %eax,PT\_EAX(%esp) #(4)

……………

……………

syscall\_badsys:

movl $-ENOSYS,PT\_EAX(%esp)

jmp resume\_userspace

END(syscall\_badsys)

由于发出int0x80指令后，进程通过中断方式陷入内核，栈指针由用户栈变向内核栈，代码行(1)利用宏SAVE\_ALL把用户程序传递下给寄存器调用号和参数依次压入内核栈，供系统调用服务程序使用。

代码行(2),将系统调用号与总的系统调用数比较，如果大于它说明该系统调用号错误。返回错误ENOSYS给系统调用返回值。

代码行(3), 如果调用号正确 就通过系统调用表sys\_call\_table转向相应的系统调用服务函数。

call \*sys\_call\_table(,%eax,4):系统调用表每个表项占4字节，因此把系统调用号（eax）乘以4，再加上sys\_call\_table的起始地址，然后从这个地址获取只想服务例程的指针，内核就找到了要调用的服务例程。，并执行。

SAVE\_ALL

cld

PUSH\_GS

pushl\_cfi %fs

/\*CFI\_REL\_OFFSET fs, 0;\*/

pushl\_cfi %es

/\*CFI\_REL\_OFFSET es, 0;\*/

pushl\_cfi %ds

/\*CFI\_REL\_OFFSET ds, 0;\*/

pushl\_cfi %eax

CFI\_REL\_OFFSET eax, 0

pushl\_cfi %ebp

CFI\_REL\_OFFSET ebp, 0

pushl\_cfi %edi

CFI\_REL\_OFFSET edi, 0

pushl\_cfi %esi

CFI\_REL\_OFFSET esi, 0

pushl\_cfi %edx

CFI\_REL\_OFFSET edx, 0

pushl\_cfi %ecx

CFI\_REL\_OFFSET ecx, 0

pushl\_cfi %ebx

CFI\_REL\_OFFSET ebx, 0

……..

3. 系统调用的退出：返回值的传递

从上面system\_call代码行（4），看出内核的系统调用服务例程通函数过eax返回值，然后再把返回值放于保存用户堆栈eax寄存器的那个栈单元的位置上。用于后面的restore\_all, pop给eax寄存器作为系统调用的返回。

syscall\_exit:

LOCKDEP\_SYS\_EXIT

DISABLE\_INTERRUPTS(CLBR\_ANY) # make sure we don't miss an interrupt

# setting need\_resched or sigpending

# between sampling and the iret

TRACE\_IRQS\_OFF

movl TI\_flags(%ebp), %ecx

testl $\_TIF\_ALLWORK\_MASK, %ecx # current->work

jne syscall\_exit\_work

restore\_all:

popl\_cfi %ebx

CFI\_RESTORE ebx

popl\_cfi %ecx

CFI\_RESTORE ecx

popl\_cfi %edx

CFI\_RESTORE edx

popl\_cfi %esi

CFI\_RESTORE esi

popl\_cfi %edi

CFI\_RESTORE edi

popl\_cfi %ebp

CFI\_RESTORE ebp

popl\_cfi %eax

CFI\_RESTORE eax

………………

restore\_all:将保存用户栈寄存器的值重新加载给寄存器，eax由于来的系统调用号变为返回值，供用户程序获取。