# 1、系统调用简介

系统调用在用户空间进程和硬件设备之间添加的一个中间层。主要有三个作用：

A．它为用户空间提供了一种硬件的抽象接口

B．隔离用户态和内核态，保证系统的稳定和安全

C．每个进程都运行在虚拟地址，实现进程独立和方便虚拟内存管理

在linux系统中，系统调用是用户空间主动访问内核的唯一方法；除异常和中断外，是内核唯一的合法入口。下图是简单的调用流程（以getpid为例）：

A. 每个系统调用都有一个调用号（getpid=20），目前，内核已经有383个系统调用（部分废除）

B．通过swi指令产生软中断进入内核态，后面的调用号存放在swi指令的低24位（即最大支持16777215个系统调用号）

C．每个系统调用参数一般不能超过6个。

系统调用命名规则：在用户空间定义成getpid则内核加前缀编程sys\_getpid（SYSCALL\_DEFINE宏定义），展开后的定义如下：

asmlinkage long sys\_getpid(void)，asmlinkage一个编译指令，通知编译器仅从栈中提取该函数的参数。

#define SYSCALL\_DEFINE0(sname)

#define SYSCALL\_DEFINE1(name, ...) SYSCALL\_DEFINEx(1, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE2(name, ...) SYSCALL\_DEFINEx(2, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE3(name, ...) SYSCALL\_DEFINEx(3, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE4(name, ...) SYSCALL\_DEFINEx(4, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE5(name, ...) SYSCALL\_DEFINEx(5, \_##name, \_\_VA\_ARGS\_\_)

#define SYSCALL\_DEFINE6(name, ...) SYSCALL\_DEFINEx(6, \_##name, \_\_VA\_ARGS\_\_)

宏定义后面的0~6表示系统调用的函数参数个数

Linux系统调用比其他许多操作系统执行的要快，linux很短的上下文切换时间是一个重要的原因，进出内核都被优化的简介高效。另外一个原因是系统调用处理程序和每个系统调用本身也都非常简洁。

# 2、系统调用服务例程

上一节已经提到系统调用通过软中断实现：通过swi产生一个软中断促使系统切换到内核态去执行中断处理程序（vector\_swi）。

（1）swi指令

指令格式如下：SWI {cond} immed\_24

Cond域：是可选的条件码 (参见 ARM汇编指令条件执行详解).

immed\_24域：24位立即数(即0-16777215)，立即数作用：区分用户不同操作，执行不同内核函数。如果用户程序调用系统调用时传递参数，**根据ATPCSC语言与汇编混合编程规则将参数放入R0~R4即可。**

使用SWI指令时，通常使用以下两种方法进行传递参数，SWI异常中断处理程序要通过读取引起软件中断的SWI指令，以取得24为立即数。

A．指令中的24位立即数指定了用户请求的服务类型，参数通过通用寄存器传递．

mov  r0,#34   ;设置子功能号位34

SWI   12      ;调用12号软中断

B．指令中的24位立即数被忽略，用户请求的服务类型有寄存器R0的值决定，参数通过其他的通用寄存器传递．

　mov r0,#12     ;调用12号软中断

　mov r1,#34     ;设置子功能号位34

　SWI　　0

在SWI异常中断处理程序中，取出SWI立即数的步骤为：首先确定引起软中断的SWI指令是ARM指令还是Thunb指令，这可通过对SPSR访问得到；然后取得该SWI指令的地址，这可通过访问LR寄存器得到；接着读出指令，分解出立即数。

（2）OABI和EABI

CONFIG\_OABI\_COMPAT代表old ABI：通过跟随在swi指令中的调用号来进行。

swi (#num | 0x900000)

CONFIG\_AEABI代表新的方式为EABI，根据r7的值

mov r7, #num

swi 0

两种宏可以同时配置，可以不配，也可以配置任何一种。以old ABI方式的系统调用会执行sys\_oabi\_call\_table表中的系统调用函数，EABI方式的系统调用会用sys\_call\_table中的函数指针。因此配置存在一下4中情况：

第一种 两个宏都配置，行为就是上面说的

第二种 只配置了CONFIG\_OABI\_COMPAT，那么以old ABI方式调用的会用sys\_oabi\_call\_table，以EABI方式调用的用sys\_call\_table，和情况1实质相同，只是情况1更加明确

第三种 只配置CONFIG\_AEABI系统中不存在sys\_oabi\_call\_table，对于old ABI方式调用不兼容，只能以EABI方式调用，用sys\_call\_table

第四种 两个都没有配置，系统默认会只允许old ABI方式，但是不存在sys\_oabi\_call\_table，最终会通过sys\_call\_table完成函数调用

uClibc库处理代码：

/\* Call a given syscall, with arguments loaded. For EABI, we must

save and restore r7 for the syscall number. Unlike the DO\_CALL

macro in glibc, this macro does not load syscall arguments. \*/

#undef DO\_CALL

#if defined(\_\_ARM\_EABI\_\_)

#define DO\_CALL(syscall\_name) \

mov ip, r7; \

ldr r7, =SYS\_ify (syscall\_name); \

swi 0x0; \

mov r7, ip;

#else

#define DO\_CALL(syscall\_name) \

swi SYS\_ify (syscall\_name);

#endif

（3）源码分析

源码目录：arch/arm/kernel/entry-common.S。该中断处理程序在《linux内核之进程详解》一文中讲解过部分，这里主要简介系统调用部分。

.align 5

ENTRY(vector\_swi) //中断服务例程

sub sp, sp, #S\_FRAME\_SIZE

stmia sp, {r0 - r12} @ Calling r0 - r12

ARM( add r8, sp, #S\_PC )

ARM( stmdb r8, {sp, lr}^ ) @ Calling sp, lr

THUMB( mov r8, sp )

THUMB( store\_user\_sp\_lr r8, r10, S\_SP ) @ calling sp, lr

mrs r8, spsr @ called from non-FIQ mode, so ok.

str lr, [sp, #S\_PC] @ Save calling PC

str r8, [sp, #S\_PSR] @ Save CPSR

str r0, [sp, #S\_OLD\_R0] @ Save OLD\_R0

zero\_fp

#ifdef CONFIG\_ALIGNMENT\_TRAP

ldr ip, \_\_cr\_alignment

ldr ip, [ip]

mcr p15, 0, ip, c1, c0 @ update control register

#endif

enable\_irq

ct\_user\_exit

get\_thread\_info tsk

//获取系统调用号

#if defined(CONFIG\_OABI\_COMPAT)

//查看swi指令值判断是EABI还是old ABI

#ifdef CONFIG\_ARM\_THUMB

tst r8, #PSR\_T\_BIT

movne r10, #0 @ no thumb OABI emulation

USER( ldreq r10, [lr, #-4] ) //获取swi指令

#else

USER( ldr r10, [lr, #-4] ) //获取swi指令

#endif

#elif defined(CONFIG\_AEABI)

//如果只定义了AEABI，则系统调用号存放在r7

#elif defined(CONFIG\_ARM\_THUMB)

/\* Legacy ABI only, possibly thumb mode. \*/

tst r8, #PSR\_T\_BIT @ this is SPSR from save\_user\_regs

addne scno, r7, #\_\_NR\_SYSCALL\_BASE @ put OS number in

USER( ldreq scno, [lr, #-4] )

#else

/\* Legacy ABI only. \*/

USER( ldr scno, [lr, #-4] ) @ get SWI instruction

#endif

adr tbl, sys\_call\_table //加载EABI方式的系统调用表

#if defined(CONFIG\_OABI\_COMPAT)

如果swi低24位参数为0，则是EABI方式

如果不为0，则计算并存放系统调用号到scno中，同时获取系统调用表sys\_oabi\_call\_table

bics r10, r10, #0xff000000 //获取swi低24位立即数

eorne scno, r10, #\_\_NR\_OABI\_SYSCALL\_BASE //去除幻数0x900000

ldrne tbl, =sys\_oabi\_call\_table //获取old ABI的系统调用表

#elif !defined(CONFIG\_AEABI)

bic scno, scno, #0xff000000 @ mask off SWI op-code

eor scno, scno, #\_\_NR\_SYSCALL\_BASE @ check OS number

#endif

local\_restart:

/\*查看是否需要跟踪系统调用\*/

ldr r10, [tsk, #TI\_FLAGS] @ check for syscall tracing

stmdb sp!, {r4, r5} @ push fifth and sixth args

tst r10, #\_TIF\_SYSCALL\_WORK @ are we tracing syscalls?

bne \_\_sys\_trace

cmp scno, #NR\_syscalls //判读系统调用号是否超出总系统调用号

adr lr, BSYM(ret\_fast\_syscall) //系统调用返回地址

ldrcc pc, [tbl, scno, lsl #2] //调用相应的系统调用sys\_\*

//下面是对系统调用号异常处理（未识别，或者系统调用使用默认处理方式）

add r1, sp, #S\_OFF

2: mov why, #0 @ no longer a real syscall

cmp scno, #(\_\_ARM\_NR\_BASE - \_\_NR\_SYSCALL\_BASE)

eor r0, scno, #\_\_NR\_SYSCALL\_BASE @ put OS number back

bcs arm\_syscall

b sys\_ni\_syscall @ not private func

#if defined(CONFIG\_OABI\_COMPAT) || !defined(CONFIG\_AEABI)

/\*

\* We failed to handle a fault trying to access the page

\* containing the swi instruction, but we're not really in a

\* position to return -EFAULT. Instead, return back to the

\* instruction and re-enter the user fault handling path trying

\* to page it in. This will likely result in sending SEGV to the

\* current task.

\*/

9001:

sub lr, lr, #4

str lr, [sp, #S\_PC]

b ret\_fast\_syscall //调用相应的系统调用sys\_\*

#endif

ENDPROC(vector\_swi)

# 3、系统调用实现

一个linux的系统调用在实现时不需要太关心它和系统调用处理程序之间的关系。添加一个新的系统调用是件相对容易的工作，怎样设计和实现一个系统调用处理程序是难题所在，基本原则如下：

A．每个系统调用都应该有一个明确的用途

B．系统调用的接口应该力求简洁、参数尽可能少，明确返回值和错误码。

C．必须仔细检查所有的参数，通过copy\_to\_user和copy\_from\_user来实现用户和内核数据拷贝

D．检查操作的合法权限

E．系统调用必需是可以重入的。

实现系统调用基本步骤：

A．arch/arm/kernel/calls.S的系统调用表最后添加如表项

CALL(sys\_seccomp)

CALL(sys\_mysyscall)

B．arch/arm/include/uapi/asm/unistd.h添加调用号

#define \_\_NR\_seccomp (\_\_NR\_SYSCALL\_BASE+383)

#define \_\_NR\_mysyscall (\_\_NR\_SYSCALL\_BASE+384)

arch/arm/include/asm/unistd.h总系统调用数：

#define \_\_NR\_syscalls (388)

这里为啥是+4，而不是加1，是因为calls.S中会自动填充系统调用表

C．在源文件kernel/sys.c中实现系统调用处理程序

SYSCALL\_DEFINE0(mysyscall)

{

printk("test syscall\n");

return THREAD\_SIZE;

}

D．include/linux/syscalls.h申明函数

asmlinkage long sys\_mysyscall (void);

E．用户态使用

he\_liangbin@ubuntu-hz:~/test$ vi mysyscall.c

#include<stdio.h>

#include<unistd.h>

#include<sys/syscall.h>

//#define \_\_NR\_mysyscall 384

//\_\_syscall0(long, mysyscall)

int main()

{

long stack\_size;

stack\_size = syscall(384);

printf("the kernel stack size: %#x\n", stack\_size);

return 0;

}