<https://blog.csdn.net/feixin620/article/details/78416560>

引言：分析Android源码的过程中，要想从上至下完全明白一行代码，往往涉及app、framework、native一直到kernel，可能迷失到代码世界，明白了系统调用原理，或许能帮你峰回路转，找到进入kernel函数的入口。本文主要讲解ARM架构相关源码：

1. /bionic/libc/kernel/uapi/asm-arm/asm/unistd.h
2. /bionic/libc/arch-arm/syscalls/kill.S
3. /kernel/arch/arm/kernel/calls.S
4. /kernel/arch/arm/include/Uapi/asm/unistd.h
5. /kernel/include/uapi/asm-generic/unistd.h
6. /kernel/include/linux/syscalls.h
7. /kernel/kernel/signal.c
8. /kernel/arch/arm/kernel/entry-common.S
9. /kernelarch/arm/kernel/entry-armv.S

## 一、Syscall意义

内核提供用户空间程序与内核空间进行交互的一套标准接口，这些接口让用户态程序能受限访问硬件设备，比如申请系统资源，操作设备读写，创建新进程等。用户空间发生请求，内核空间负责执行，这些接口便是用户空间和内核空间共同识别的桥梁，这里提到两个字“受限”，是由于为了保证内核稳定性，而不能让用户空间程序随意更改系统，必须是内核对外开放的且满足权限的程序才能调用相应接口。

在用户空间和内核空间之间，有一个叫做Syscall(系统调用, system call)的中间层，是连接用户态和内核态的桥梁。这样即提高了内核的安全型，也便于移植，只需实现同一套接口即可。Linux系统，用户空间通过向内核空间发出Syscall，产生软中断，从而让程序陷入内核态，执行相应的操作。对于每个系统调用都会有一个对应的系统调用号，比很多操作系统要少很多。

安全性与稳定性：内核驻留在受保护的地址空间，用户空间程序无法直接执行内核代码，也无法访问内核数据，通过系统调用

性能：Linux上下文切换时间很短，以及系统调用处理过程非常精简，内核优化得好，所以性能上往往比很多其他操作系统执行要好。

## 二、Syscall查找方式

这里以文章[理解杀进程的实现原理](http://gityuan.com/2016/04/16/kill-signal/)中的kill()方法为例子，来找一找kill()方法系统调用的过程。

Tips 1： 用户空间的方法xxx，对应系统调用层方法则是sys\_xxx；   
Tips 2： unistd.h文件记录着系统调用中断号的信息。

故用户空间kill方法则对应系统调用层便是sys\_kill，这个方法去哪里找呢？从/kernel/include/uapi/asm-generic/unistd.h等还有很多unistd.h去慢慢查看，查看关键字sys\_kill，便能看到下面几行：

1. */\* kernel/signal.c \*/*
2. **\_\_SYSCALL(\_\_NR*\_kill, sys\_kill)***

根据这个能得到一丝线索，那就是kill对应的方法sys\_kill位于/kernel/signal.c文件。

Tips 3： 宏定义SYSCALL\_DEFINEx(xxx,…)，展开后对应的方法则是sys\_xxx；   
Tips 4： 方法参数的个数x，对应于SYSCALL\_DEFINEx。

kill(int pid, int sig)方法共两个参数，则对应方法于SYSCALL\_DEFINE2(kill,...)，进入signal.c文件，再次搜索关键字，便能看到方法：

1. SYSCALL\_DEFINE2(kill, pid\_t, pid, int, sig)
2. {
3. struct siginfo info;
4. info.si\_signo = sig;
5. info.si\_errno = 0;
6. info.si\_code = SI\_USER;
7. info.si\_pid = task\_tgid\_vnr(current);
8. info.si\_uid = from\_kuid\_munged(current\_user\_ns(), current\_uid());
9. return kill\_something\_info(sig, &info, pid);
10. }

SYSCALL\_DEFINE2(kill, pid\_t, pid, int, sig) 基本等价于 asmlinkage long sys\_kill(int pid, int sig)，这里用的是基本等价，往下看会解释原因。

#### 实用技巧

比如kill命令, 有两个参数. 则可以直接在kernel目录下搜索 “SYSCALL\_DEFINE2(kill”,即可直接找到,所有对应的Syscall方法位于signal.c

## 三、Syscall流程

Syscall是通过中断方式实现的，ARM平台上通过swi中断来实现系统调用，实现从用户态切换到内核态，发送软中断swi时，从中断向量表中查看跳转代码，其中异常向量表定义在文件/kernelarch/arm/kernel/entry-armv.S(汇编语言文件)。当执行系统调用时会根据系统调用号从系统调用表中来查看目标函数的入口地址，在calls.S文件中声明了入口地址信息。

总体流程：kill() -> kill.S -> swi陷入内核态 -> 从sys\_call\_table查看到sys\_kill -> ret\_fast\_syscall -> 回到用户态执行kill()下一行代码。 下面介绍部分核心流程：

3.1： 用户程序通过软中断swi指令切入内核态，执行vector\_swi处的指令。vector\_swi在文件/kenel/arch/arm/kernel/entry-common.S中定义，此处省略。像每一个异常处理程序一样，要做的第一件事当然就是保护现场了。紧接着是获得系统调用的系统调用号

3.2： 仍以kill()函数为例，来详细说说Syscall调用流程，用户空间kill()定义位于文件kill.S。

1. *#include <private/bionic\_asm.h>*
2. ENTRY(kill)
3. mov ip, r7
4. ldr r7, =\_\_NR\_kill
5. swi *#0*
6. mov r7, ip
7. cmn r0, *#(MAX\_ERRNO + 1)*
8. bxls lr
9. neg r0, r0
10. b \_\_set\_errno\_internal
11. END(kill)

当调用kill时, 系统先保存r7内容, 然后将\_\_NR\_kill值放入r7, 再执行swi软中断指令切换进内核态。

3.3： Linux内核中，每个Syscall都有唯一的系统调用号对应，kill的系统调用号为\_\_NR\_kill，用户空间的系统调用号定义于/bionic/libc/kernel/uapi/asm-generic/unistd.h，如下：

1. #define \_\_NR\_kill (\_\_NR\_SYSCALL\_BASE + 37)

其中\_\_NR\_SYSCALL\_BASE=0，也就是\_\_NR\_kill系统调用号=37。

3.4： 在内核中有与系统调用号对应的系统调用表，定义在文件/kernel/arch/arm/kernel/calls.S，如下：

1. */\* 35 \*/ CALL(sys\_ni\_syscall) /\* was sys\_ftime \*/*
2. CALL(sys\_sync)
3. CALL(sys\_kill) *//此处为37号*
4. CALL(sys\_rename)
5. CALL(sys\_mkdir)

到这里可知37号系统调用对应sys\_kill()，该方法所对应的函数声明在syscalls.h文件

3.5： 文件/kernel/include/linux/syscalls.h中有如下声明：

1. asmlinkage long sys\_kill(int pid, int sig);

asmlinkage是gcc标签，代表函数读取的参数来自于栈中，而非寄存器。

#### 3.1 SYSCALL\_DEFINE

sys\_kill()定义在内核源码找不到直接定义，而是通过syscalls.h文件中的SYSCALL\_DEFINE宏定义。前面已经讲过sys\_kill是通过语句SYSCALL\_DEFINE2(kill, pid\_t, pid, int, sig)来定义，下面来一层层剖开，这条宏定义的真面目：

等价 1：

syscalls.h中有大量如下宏定义：

1. #define SYSCALL\_DEFINE0(sname) \
2. SYSCALL\_METADATA(\_##sname, 0); \
3. asmlinkage long sys\_##sname(void)
4. #define SYSCALL\_DEFINE1(name, ...) SYSCALL\_DEFINEx(1, \_##name, \_\_VA\_ARGS\_\_)
5. #define SYSCALL\_DEFINE2(name, ...) SYSCALL\_DEFINEx(2, \_##name, \_\_VA\_ARGS\_\_)
6. #define SYSCALL\_DEFINE3(name, ...) SYSCALL\_DEFINEx(3, \_##name, \_\_VA\_ARGS\_\_)
7. #define SYSCALL\_DEFINE4(name, ...) SYSCALL\_DEFINEx(4, \_##name, \_\_VA\_ARGS\_\_)
8. #define SYSCALL\_DEFINE5(name, ...) SYSCALL\_DEFINEx(5, \_##name, \_\_VA\_ARGS\_\_)
9. #define SYSCALL\_DEFINE6(name, ...) SYSCALL\_DEFINEx(6, \_##name, \_\_VA\_ARGS\_\_)

可得出原语句等价：

1. SYSCALL\_DEFINEx(2, \_kill, pid\_t, pid, int, sig)

等价 2：

syscalls.h中有如下宏定义：

1. #define SYSCALL\_DEFINEx(x, sname, ...) \
2. SYSCALL\_METADATA(sname, x, \_\_VA\_ARGS\_\_) \
3. \_\_SYSCALL\_DEFINEx(x, sname, \_\_VA\_ARGS\_\_)

可得出原语句等价：

1. SYSCALL\_METADATA(\_kill, 2, pid\_t, pid, int, sig)
2. \_\_SYSCALL\_DEFINEx(2, \_kill, pid\_t, pid, int, sig)

define \_\_SYSCALL\_DEFINEx(x, name, …)

等价 3：

syscalls.h中有如下宏定义：

1. #define \_\_SYSCALL\_DEFINEx(x, name, ...) \
2. asmlinkage long sys##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)) \
3. \_\_attribute\_\_((alias(\_\_stringify(SyS##name)))); \
4. static inline long SYSC##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)); \
5. asmlinkage long SyS##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)); \
6. asmlinkage long SyS##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)) \
7. { \
8. long ret = SYSC##name(\_\_MAP(x,\_\_SC\_CAST,\_\_VA\_ARGS\_\_)); \
9. \_\_MAP(x,\_\_SC\_TEST,\_\_VA\_ARGS\_\_); \
10. \_\_PROTECT(x, ret,\_\_MAP(x,\_\_SC\_ARGS,\_\_VA\_ARGS\_\_)); \
11. return ret; \
12. } \
13. static inline long SYSC##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_))

可得出原语句等价：

1. asmlinkage long sys\_kill(\_\_MAP(2,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)) \
2. \_\_attribute\_\_((alias(\_\_stringify(SyS\_kill)))); \
3. static inline long SYSC\_kill(\_\_MAP(2,\_\_SC\_DECL,\_\_VA\_ARGS\_\_)); \
4. asmlinkage long SyS\_kill(\_\_MAP(2,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)); \
5. asmlinkage long SyS\_kill(\_\_MAP(2,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)) \
6. { \
7. long ret = SYSC\_kill(\_\_MAP(2,\_\_SC\_CAST,\_\_VA\_ARGS\_\_)); \
8. \_\_MAP(2,\_\_SC\_TEST,\_\_VA\_ARGS\_\_); \
9. \_\_PROTECT(2, ret,\_\_MAP(2,\_\_SC\_ARGS,\_\_VA\_ARGS\_\_)); \
10. return ret; \
11. } \
12. static inline long SYSC\_kill(\_\_MAP(2,\_\_SC\_DECL,\_\_VA\_ARGS\_\_))

这里\_\_VA\_ARGS\_\_等于 pid\_t, pid, int, sig。

等价 4:

先说说这里涉及的宏定义

\_\_MAP宏定义：

1. *#define \_\_MAP0(m,...)*
2. *#define \_\_MAP1(m,t,a) m(t,a)*
3. *#define \_\_MAP2(m,t,a,...) m(t,a), \_\_MAP1(m,\_\_VA\_ARGS\_\_)*
4. *#define \_\_MAP3(m,t,a,...) m(t,a), \_\_MAP2(m,\_\_VA\_ARGS\_\_)*
5. *#define \_\_MAP4(m,t,a,...) m(t,a), \_\_MAP3(m,\_\_VA\_ARGS\_\_)*
6. *#define \_\_MAP5(m,t,a,...) m(t,a), \_\_MAP4(m,\_\_VA\_ARGS\_\_)*
7. *#define \_\_MAP6(m,t,a,...) m(t,a), \_\_MAP5(m,\_\_VA\_ARGS\_\_)*
8. *#define \_\_MAP(n,...) \_\_MAP##n(\_\_VA\_ARGS\_\_)*

相关宏定义：

1. *#define \_\_SC\_DECL(t, a) t a*
2. *#define \_\_SC\_LONG(t, a) \_\_typeof(\_\_builtin\_choose\_expr(\_\_TYPE\_IS\_LL(t), 0LL, 0L)) a*
3. *#define \_\_SC\_CAST(t, a) (t) a*
4. *#define \_\_SC\_ARGS(t, a) a*
5. *#define \_\_SC\_TEST(t, a) (void)BUILD\_BUG\_ON\_ZERO(!\_\_TYPE\_IS\_LL(t) && sizeof(t) > sizeof(long))*

展开：

1. \_\_MAP(2,\_\_SC\_DECL, pid\_t, pid, int, sig) *//等价于 pid\_t pid, int sig*
2. \_\_MAP(2,\_\_SC\_LONG,\_\_VA\_ARGS\_\_) *//等价于 long pid, long sig*
3. \_\_MAP(2,\_\_SC\_CAST,\_\_VA\_ARGS\_\_) *//等价于 (pid\_t) pid, (int)sig*
4. \_\_MAP(2,\_\_SC\_ARGS,\_\_VA\_ARGS\_\_) *//等价于 pid, sig*

可得出原语句等价：

1. *//函数声明sys\_kill()，并别名指向SyS\_kill*
2. asmlinkage long sys\_kill(pid\_t pid, int sig) \_\_attribute\_\_((alias(\_\_stringify(SyS\_kill))));
3. static inline long SYSC\_kill(pid\_t pid, int sig);
5. *//函数声明SyS\_kill()*
6. asmlinkage long SyS\_kill(long pid, long sig);
7. asmlinkage long SyS\_kill(long pid, long sig)
8. {
9. long ret = SYSC\_kill((pid\_t) pid, (int)sig);
10. BUILD\_BUG\_ON\_ZERO(sizeof(pid\_t) > sizeof(long));
11. BUILD\_BUG\_ON\_ZERO(sizeof(int) > sizeof(long));
12. \_\_PROTECT(2, ret, pid, sig);
13. return ret;
14. }
15. static inline long SYSC\_kill(pid\_t pid, int sig)

通过以上分析过程：

* kill添加了sys\_前缀，声明sys\_kill()函数；
* 定义SYSC\_kill()函数和SyS\_kill()函数；
* sys\_kill，通过别名机制等同于SyS\_kill().

看到这或许很多人(包括我)会觉得诧异，为何要如此复杂呢，后来查资料，发现这是由于之前64位Linux存在CVE-2009-2009的漏洞，简单说就是32位参数存放在64位寄存器，修改符号扩展可能导致产生一个非法内存地址，从而导致系统崩溃或者提升权限。 为了修复这个问题，把寄存器高位清零即可，但做起来比较困难，为了做尽可能少的修改，将调用参数统一采用使用long型来接收，再强转为相应参数。 窥见一斑，可见Linux大师们精湛的宏定义，已经用得出神入化。

如果觉得很复杂，那么可以忽略这个宏定义，只要记住SYSCALL\_DEFINE2(kill, pid\_t, pid, int, sig) 基本等价于 asmlinkage long sys\_kill(int pid, int sig) 就足够了。

## 四、总结

#### 4.1 内核空间

1. 系统调用的函数原型的指针：位于文件/kernel/arch/arm/kernel/calls.S，格式为CALL(sys\_xxx)，指定了目标函数的入口地址。
2. 系统调用号的宏定义：位于文件/kernel/arch/arm/include/Uapi/asm/unistd.h，记录着内核空间的系统调用号，格式为#define\_\_NR\_xxx (\_\_NR\_SYSCALL\_BASE+[num])
3. 系统调用的函数声明：位于文件/kernel/include/linux/syscalls.h，格式为asmlinkage long sys\_xxx(args ...);
4. 系统调用的函数实现：不同函数位于不同文件，比如kill()位于/kernel/kernel/signal.c文件，格式为SYSCALL\_DEFINEx(x, sname, ...)

前面这4步都是在内核空间相关的文件定义，有了这些，那么内核就可以使用相应的系统调用号。

#### 4.2 用户空间

1. 系统调用号的宏定义：位于文件/bionic/libc/kernel/uapi/asm-arm/asm/unistd.h，记录着用户空间的系统调用号，格式为#define\_\_NR\_xxx (\_\_NR\_SYSCALL\_BASE+[num])。这个文件就是由内核空间同名的头文件自动生成的，所以该文件与内核空间的系统调用号是完全一致。
2. 汇编定义相关函数的中断调用过程：位于文件/bionic/libc/arch-arm/syscalls/xxx.S，比如kill()位于kill.S，格式为：
   1. ENTRY(xxx)
   2. mov ip, r7
   3. ldr r7, =\_\_NR\_xxx
   4. swi *#0*
   5. *mov r7, ip*
   6. *cmn r0, #(MAX\_ERRNO + 1)*
   7. *bxls lr*
   8. *neg r0, r0*
   9. *b \_\_set\_errno\_internal*
   10. *END(xxx)*

当然kill()方法还有函数声明，有了这些，用户空间也能在程序中使用系统调用。明白了这些过程，那么自己新添加系统调用其实也并不是多困难的一件事，新增系统调用号还需要修改syscalls总个数，但强烈不建议自己新增系统调用号，尽量保持与linux kernel主线一致，兼容性更好，所以就不进一步介绍新增流程了。

转自：<http://gityuan.com/2016/05/21/syscall/>