openEuler内核编程技术

实训指导书

第四章 第5讲

《系统调用》

软件所制

**术语表**

|  |  |  |  |
| --- | --- | --- | --- |
| 英文 | 中文 | 含义 | 别名 |
|  |  |  |  |
|  |  |  |  |

第四章 第5讲 创建并运行内核线程

**任务1：添加一个新的系统调用，newbaby()（120min）**

**相关知识**

**一、系统调用介绍**

Linux内核中设置了一组用于实现各种系统功能的子程序，称为系统调用。用户可以通过系统调用命令在自己的应用程序中调用它们。从某种角度来看，系统调用和普通的函数调用非常相似。区别仅仅在于，系统调用由操作系统核心提供，运行于核心态；而普通的函数调用由函数库或用户自己提供，运行于用户态。二者在使用方式上也有相似之处。Linux系统的核心部分即是Linux内核，是一系列设备的驱动程序。系统调用是Linux内核提供的功能十分强大的一系列的函数。这些函数是在内核中实现的，它们是应用程序和内核交互的接口，系统调用在Linux系统中发挥着巨大的作用，如果没有系统调用，那么应用程序就失去了内核的支持。

在ARM Linux内核中，系统调用是一种特殊的异常，通常被归于同步异常的范畴，同步异常是由正在运行的指令或指令运行的结果造成的异常。系统调用是操作系统内核为用户程序提供系统服务的接口，操作系统将一些需要在内核态运行的公共服务通过系统调用封装并提供给应用程序。用户程序在使用系统调用时将陷入内核态，并调用系统用的处理函数。不同的系统调用有不同的编号，它们被称为系统调用号。ARM64处理器中使用SVC指令触发系统调用的约定如下:

64位用户程序使用寄存器x8传递系统调用号，32位用户程序使用寄存器x7传递系统调用号；

使用寄存器x0-x6传递系统调用所需参数，最多可传递7个参数；

系统调用执行完后，用寄存器x0存放返回值。

ARM Linux内核中使用SYSCALL\_DEFINEn（...）宏来定义一个系统调用，其中n为非负整数，表示后面括号中参数的数目。以内核中用于向进程发送信号的kill系统调用为例，其定义代码在openeuler/kernel/blob/kernel-4.19/kernel/signal.c文件中可以找到：

SYSCALL\_DEFINE2(kill, pid\_t, pid, int, sig)

{

    struct siginfo info;

    clear\_siginfo(&info);

    info.si\_signo = sig;

    info.si\_errno = 0;

    info.si\_code = SI\_USER;

    info.si\_pid = task\_tgid\_vnr(current);

    info.si\_uid = from\_kuid\_munged(current\_user\_ns(), current\_uid());

    return kill\_something\_info(sig, &info, pid);

}

SYSCALL\_DEFINE的相关宏定义可以在openeuler/kernel/blob/kernel-4.19/include/linux/syscalls.h中找到（以下相关汇编代码在同一文件中）：

#ifndef SYSCALL\_DEFINE0

#define SYSCALL\_DEFINE0(sname)                  \

    SYSCALL\_METADATA(\_##sname, 0);              \

    asmlinkage long sys\_##sname(void);          \

    ALLOW\_ERROR\_INJECTION(sys\_##sname, ERRNO);      \

    asmlinkage long sys\_##sname(void)

#endif /\* SYSCALL\_DEFINE0 \*/

#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\_\_)

在C语言宏定义中##被解释成分隔与连接一个符号，符号可以是一个变量，也就是说“\_##name”被编译器理解为两段：“\_”和“name”，而name和前面的输入参数可以匹配，所以“\_##name”实际上是将输入参数name前面加了“\_”，例如输入“kill”就会变成“\_kill”。\_\_VA\_ARGS\_\_关键字等价于可变参数列表...中省略的内容。

SYSCALL\_DEFINEx宏则会调用SYSCALL\_METADATA宏对系统调用名称进行进一步变化：

#define SYSCALL\_DEFINEx(x, sname, ...)              \

    SYSCALL\_METADATA(sname, x, \_\_VA\_ARGS\_\_)         \

    \_\_SYSCALL\_DEFINEx(x, sname, \_\_VA\_ARGS\_\_)

SYSCALL\_METADATA中设置了与系统调用有关的参数，例如系统调用的参数类型列表、系统调用的参数值列表和系统调用的基本信息结构体等：

#define SYSCALL\_METADATA(sname, nb, ...)            \

    static const char \*types\_##sname[] = {          \

        \_\_MAP(nb,\_\_SC\_STR\_TDECL,\_\_VA\_ARGS\_\_)        \

    };                          \

    static const char \*args\_##sname[] = {           \

        \_\_MAP(nb,\_\_SC\_STR\_ADECL,\_\_VA\_ARGS\_\_)        \

    };                          \

    SYSCALL\_TRACE\_ENTER\_EVENT(sname);           \

    SYSCALL\_TRACE\_EXIT\_EVENT(sname);            \

    static struct syscall\_metadata \_\_used           \

      \_\_syscall\_meta\_##sname = {                \

        .name       = "sys"#sname,          \

        .syscall\_nr = -1,   /\* Filled in at boot \*/ \

        .nb\_args    = nb,               \

        .types      = nb ? types\_##sname : NULL,    \

        .args       = nb ? args\_##sname : NULL, \

        .enter\_event    = &event\_enter\_##sname,     \

        .exit\_event = &event\_exit\_##sname,      \

        .enter\_fields   = LIST\_HEAD\_INIT(\_\_syscall\_meta\_##sname.enter\_fields), \

    };                          \

    static struct syscall\_metadata \_\_used           \

      \_\_section("\_\_syscalls\_metadata")          \

     \*\_\_p\_syscall\_meta\_##sname = &\_\_syscall\_meta\_##sname;

例如系统调用的名字.name变成了“sys”#sname。宏定义中#表示字符串化，也就是说当sname为\_kill时.name的值就是“sys”“\_kill”。\_\_MAP宏可以将多个参数分解为参数对，其参数对数目由输入参数nb决定：

#define \_\_MAP0(m,...)

#define \_\_MAP1(m,t,a,...) m(t,a)

#define \_\_MAP2(m,t,a,...) m(t,a), \_\_MAP1(m,\_\_VA\_ARGS\_\_)

#define \_\_MAP3(m,t,a,...) m(t,a), \_\_MAP2(m,\_\_VA\_ARGS\_\_)

#define \_\_MAP4(m,t,a,...) m(t,a), \_\_MAP3(m,\_\_VA\_ARGS\_\_)

#define \_\_MAP5(m,t,a,...) m(t,a), \_\_MAP4(m,\_\_VA\_ARGS\_\_)

#define \_\_MAP6(m,t,a,...) m(t,a), \_\_MAP5(m,\_\_VA\_ARGS\_\_)

#define \_\_MAP(n,...) \_\_MAP##n(\_\_VA\_ARGS\_\_)

\_\_SYSCALL\_DEFINEx宏则定义了函数的调用接口：

#ifndef \_\_SYSCALL\_DEFINEx

#define \_\_SYSCALL\_DEFINEx(x, name, ...)                 \

    \_\_diag\_push();                          \

    \_\_diag\_ignore(GCC, 8, "-Wattribute-alias",          \

              "Type aliasing is used to sanitize syscall arguments");\

    asmlinkage long sys##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_))   \

        \_\_attribute\_\_((alias(\_\_stringify(\_\_se\_sys##name))));    \

    ALLOW\_ERROR\_INJECTION(sys##name, ERRNO);            \

    static inline long \_\_do\_sys##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_));\

    asmlinkage long \_\_se\_sys##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_)); \

    asmlinkage long \_\_se\_sys##name(\_\_MAP(x,\_\_SC\_LONG,\_\_VA\_ARGS\_\_))  \

    {                               \

        long ret = \_\_do\_sys##name(\_\_MAP(x,\_\_SC\_CAST,\_\_VA\_ARGS\_\_));\

        \_\_MAP(x,\_\_SC\_TEST,\_\_VA\_ARGS\_\_);             \

        \_\_PROTECT(x, ret,\_\_MAP(x,\_\_SC\_ARGS,\_\_VA\_ARGS\_\_));   \

        return ret;                     \

    }                               \

    \_\_diag\_pop();                           \

    static inline long \_\_do\_sys##name(\_\_MAP(x,\_\_SC\_DECL,\_\_VA\_ARGS\_\_))

#endif /\* \_\_SYSCALL\_DEFINEx \*/

这段代码首先声明了一个函数sys##name，然后使该函数成为\_\_se\_sys##name函数的别名（也就是说调用这两个函数是等价的）。但是sys##name和\_\_se\_sys##name两个函数之间存在一个类型转换，在sys##name中的原输入类型在\_\_se\_sys##name的参数列表中都变为了long类型，这个类型转换是通过\_\_SC\_DECL宏和\_\_SC\_LONG宏实现的。接着函数\_\_se\_sys##name调用了函数\_\_do\_sys##name，实际上\_\_do\_sys##name函数的内容就是我们对系统调用的定义。在这段代码的最后部分，\_\_do\_sys##name的函数声明后面没有加分号，这样它就能和内核中SYSCALL\_DEFINE宏后面的函数体构成函数定义。根据前文说述的调用关系，使用sys##name就能调用\_\_do\_sys##name定义的函数。我们可以看到\_\_do\_sys##name函数又将long类型的输入转换为了原输入的参数类型。之所以要做这种转换是为了对可能会放在64位寄存器中的输入参数（例如32位输入参数）进行符号扩展，从而防止用户程序通过传入特定的参数取得权限提升或导致系统崩溃。这种情况曾经在CVE-2009-0029漏洞中出现，如果传入系统调用的参数的高32位为非法值，就有可能访问到非法的地址。

\_\_SC\_DECL宏、\_\_SC\_CAST和\_\_SC\_LONG宏的定义如下：

#define \_\_SC\_DECL(t, a) t a

#define \_\_TYPE\_AS(t, v) \_\_same\_type((\_\_force t)0, v)

#define \_\_TYPE\_IS\_L(t)  (\_\_TYPE\_AS(t, 0L))

#define \_\_TYPE\_IS\_UL(t) (\_\_TYPE\_AS(t, 0UL))

#define \_\_TYPE\_IS\_LL(t) (\_\_TYPE\_AS(t, 0LL) || \_\_TYPE\_AS(t, 0ULL))

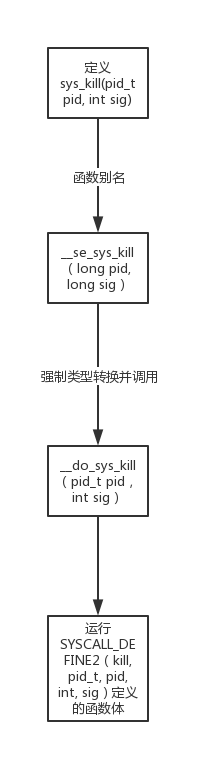
#define \_\_SC\_LONG(t, a) \_\_typeof(\_\_builtin\_choose\_expr(\_\_TYPE\_IS\_LL(t), 0LL, 0L)) a

#define \_\_SC\_CAST(t, a) (\_\_force t) a

它们的作用分别是使参数保持输入参数列表中的类型、对参数列表中参数的类型进行强制类型装换和将参数列表中的参数定义为long类型。

以kill系统调用为例，其定义语句为SYSCALL\_DEFINE2（kill, pid\_t, pid, int, sig）

说明该系统调用有两个参数pid和sig，其类型分别为pid\_t和int。pid是目标进程的进程号，sig是要发送的信号。kill系统调用的定义过程如下图所示：



\_\_do\_sys##name函数对应的函数声明为\_\_do\_sys\_kill（pid\_t pid，int sig），其函数体为SYSCALL\_DEFINE2宏下面定义的函数体。\_\_do\_sys\_kill（）函数是在\_\_se\_sys\_kill（）函数中被调用的，\_\_se\_sys\_kill（）函数的声明为\_\_se\_sys\_kill（long pid，long sig）。\_\_se\_sys\_kill（）函数为sys\_kill（）函数的别名，sys\_kill（）函数的声明为sys\_kill（pid\_t pid，int sig），通过调用sys\_kill（）函数可其使用kill系统调用。

系统调用号与系统调用处理函数的映射关系被保存在系统调用表中，系统调用表的相关代码在openeuler/kernel/blob/kernel-4.19/arch/arm64/kernel/sys.c文件中可以找到：

#undef \_\_SYSCALL

#define \_\_SYSCALL(nr, sym)  [nr] = \_\_arm64\_##sym,

const syscall\_fn\_t sys\_call\_table[\_\_NR\_syscalls] = {

    [0 ... \_\_NR\_syscalls - 1] = \_\_arm64\_sys\_ni\_syscall,

#include <asm/unistd.h>

};

\_\_SYSCALL宏用于注册系统调用表。syscall\_fn\_t是系统调用处理函数的函数指针类型，其定义在syscalls.h文件中：

typedef long (\*syscall\_fn\_t)(const struct pt\_regs \*regs);

pt\_regs结构体包含异常发生时栈上寄存器中保存的信息，其代码在openeuler/kernel/blob/kernel-4.19/arch/arm64/include/asm/ptrace.h文件中：

struct pt\_regs {

    union {

        struct user\_pt\_regs user\_regs;

        struct {

            u64 regs[31];

            u64 sp;

            u64 pc;

            u64 pstate;

        };

    };

    u64 orig\_x0;

#ifdef \_\_AARCH64EB\_\_

    u32 unused2;

    s32 syscallno;

#else

    s32 syscallno;

    u32 unused2;

#endif

    u64 orig\_addr\_limit;

    u64 unused; // maintain 16 byte alignment

    u64 stackframe[2];

};

从该结构体的定义代码中我们可以看到它包含了31个通用寄存器、栈指针寄存器SP、程序计数器PC和处理器状态PSTATE等信息。

系统调用表中#include了asm/unistd.h文件，该文件在openeuler/kernel/blob/kernel-4.19/arch/arm64/include/asm/unistd.h，它包含了系统调用号等信息。如果定义了\_\_COMPAT\_SYSCALL\_NR宏，使用Compat系统调用号：

#define \_\_NR\_compat\_restart\_syscall 0

#define \_\_NR\_compat\_exit        1

#define \_\_NR\_compat\_read        3

#define \_\_NR\_compat\_write       4

#define \_\_NR\_compat\_sigreturn       119

#define \_\_NR\_compat\_rt\_sigreturn    173

否则最终使用openeuler/kernel/blob/kernel-4.19/include/uapi/asm-generic/unistd.h文件中定义的系统调用映射关系：

#define \_\_NR\_io\_setup 0//系统调用号0

\_\_SC\_COMP(\_\_NR\_io\_setup, sys\_io\_setup, compat\_sys\_io\_setup)//系统调用号0对应的系统调用为sys\_io\_setup或compat\_sys\_io\_setup

#define \_\_NR\_io\_destroy 1

\_\_SYSCALL(\_\_NR\_io\_destroy, sys\_io\_destroy)

#define \_\_NR\_io\_submit 2

\_\_SC\_COMP(\_\_NR\_io\_submit, sys\_io\_submit, compat\_sys\_io\_submit)

#define \_\_NR\_io\_cancel 3

\_\_SYSCALL(\_\_NR\_io\_cancel, sys\_io\_cancel)

#define \_\_NR\_io\_getevents 4

\_\_SC\_COMP(\_\_NR\_io\_getevents, sys\_io\_getevents, compat\_sys\_io\_getevents)

……

#undef \_\_NR\_syscalls

#define \_\_NR\_syscalls 294

该文件中共定义了294个系统调用。\_\_SC\_COMP宏的定义如下：

#ifdef \_\_SYSCALL\_COMPAT

#define \_\_SC\_COMP(\_nr, \_sys, \_comp) \_\_SYSCALL(\_nr, \_comp)

#define \_\_SC\_COMP\_3264(\_nr, \_32, \_64, \_comp) \_\_SYSCALL(\_nr, \_comp)

#else

#define \_\_SC\_COMP(\_nr, \_sys, \_comp) \_\_SYSCALL(\_nr, \_sys)

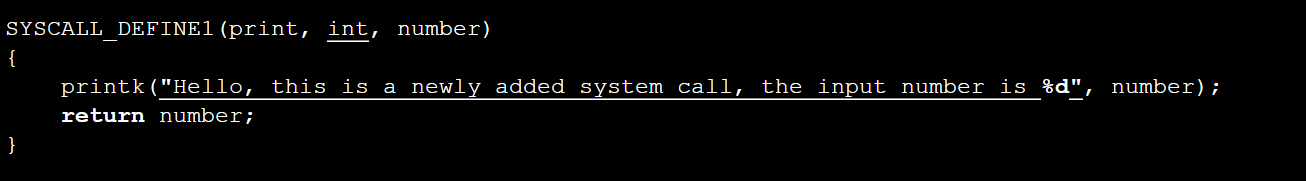
#define \_\_SC\_COMP\_3264(\_nr, \_32, \_64, \_comp) \_\_SC\_3264(\_nr, \_32, \_64)

#endif

当定义了\_\_SYSCALL\_COMPAT宏时，系统调用号对应的系统调用为宏定义的第三个输入参数，否则为第二个输入参数。

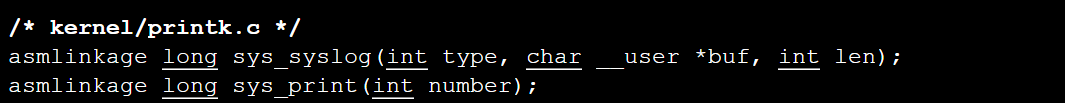
本节我们将详细讲解使用树莓派向openEuler内核中增加一个系统调用。

我们向ARM Linux内核中增加一个简单的打印相关信息的系统调用，该系统调用能打印出用户输入的数字。我们首先用SYSCALL\_DEFINEx宏定义该系统调用的函数体，由于我们用到了打印函数printk()，而该函数的定义在/kernel/printk文件中，因此我们直接在该文件中增加print系统调用的定义：

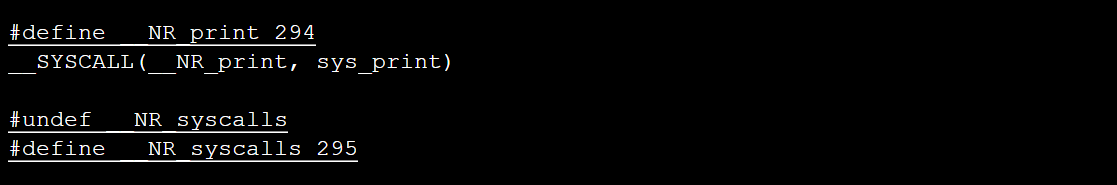


**增加一个叫sys\_print的系统调用**

该系统调用的将会被扩展为sys\_print()，其输入参数为一个整型，我们在include/linux/syscalls.h文件中声明sys\_print()函数：

**增加sys\_print()函数原型**

我们还需要在系统调用表中添加该函数，我们增加一个系统调用号，定义为\_\_NR\_print 294，其对应的系统调用为sys\_print，该修改在include/uapi/asm-generic/unistd.h 文件中实现：



**增加新的系统调用号**

然后编译并安装修改后的内核：

编译后的内核名称在编译后的屏幕上可以看到，或者使用如下命令查看：

cat /boot/grub2/grub.cfg |grep “menuentry”

安装好新的内核后我们需要编写一个用户态的测试程序print\_syscall\_test.c文件：

#define \_GNU\_SOURCE

#include <unistd.h>

#include <sys/syscall.h>

#include <stdio.h>

#define SYS\_PRINT 294

int main() {

int ret = syscall(SYS\_PRINT, 100);

printf("ret is %d\n", ret);

return ret;

}

这里的syscall()函数是用户态程序调用系统调用所用的函数名称，其输入参数为系统调用号和系统调用函数的输入参数，这里100是sys\_print()函数的输入参数。使用gcc print\_syscall\_test.c -o print\_syscall\_test将测试程序编译为目标代码，然后用sudo ./ print\_syscall\_test 命令运行测试程序，屏幕打印如下：



**屏幕打印**

使用sudo dmesg | tail -20命令查看内核日志，可以看到我们在系统调用中加的打印信息：

preview

**内核日志打印**

**任务描述**

添加一个新的系统调用，newbaby()，检查调用者的附近吃是否有超级权限。

其系统调用参数和返回定义如下：

Long newbaby(pid\_t pid, uid\_t \*uid);

1、如果给出的pid不是调用者父进程的pid，返回-1，设置EINVAL为errno。同时，打印出Not father 到内核日志

2、如果给出的pid是其父进程pid，

若uid不为NULL，其父进程的EUID被赋给uid

若父进程effective user 为root，返回0

若父进程effective user不为root，返回-1，并设EACCES为errno

3、测试，写一个应用调用新增的系统调用。

**审核要求**

1. 正确、成功地完成任务。
2. 提交每一步操作以及结果显示的截图。

**参考答案**

**一、操作步骤**

一、系统备份

|  |
| --- |
| # cd ~  # dnf install lrzsz # rz和sz可以在终端下很方便的传输文件  # tar czvf boot\_origin.tgz /boot/  # sz boot\_origin.tgz # 将备份文件发送到本地 |

二、内核源码下载

|  |
| --- |
| # 下载源码  # wget https://gitee.com/openeuler/raspberrypi-kernel/repository/archive/openEuler-20.03-LTS-raspi.zip  # 解压  # unzip openeuler-raspberrypi-kernel-openEuler-20.03-LTS-raspi.zip  # 进入内核源码根目录  # cd raspberrypi-kernel |

三、添加系统调用

|  |
| --- |
| # 添加系统调用, 增加系统总调用数+1  # vim include/uapi/asm-generic/unistd.h  # 声明  # vim include/linux/syscalls.h  # 实现  # vim kernel/sys.c |

diff in raspberrypi-kernel /include/uapi/asm-generic/unistd.h

++ #define \_\_NR\_newbaby 294

++ \_\_SYSCALL(\_\_NR\_newbaby, sys\_newbaby)

#undef \_\_NR\_syscalls

-- #define \_\_NR\_syscalls 294

++ #define \_\_NR\_syscalls 295

diff in raspberrypi-kernel / include/linux/syscalls.h

++ asmlinkage long sys\_newbaby(pid\_t pid, uid\_t \*uid);

#endif

diff in raspberrypi-kernel / kernel/sys.c

++SYSCALL\_DEFINE2(newbaby, pid\_t, pid, uid\_t \*, uid)

++{

++ int errno;

++ pid\_t pid\_cur = task\_tgid\_vnr(current);

++ struct task\_struct \*ts\_cur = NULL;

++ struct task\_struct \*ts\_par = NULL;

++ struct task\_struct \*tmp\_p;

++ for\_each\_process(tmp\_p)

++ {

++ if (tmp\_p->pid == pid\_cur)

++ ts\_cur = tmp\_p;

++ }

++ ts\_par = ts\_cur->real\_parent;

++ if (ts\_par->pid != pid)

++ {

++ errno = -EINVAL;

++ printk("Not father");

++ return -1;

++ }

++ if (uid != NULL)

++ {

++ \*uid = ts\_par->cred->euid.val;

++ if (\*uid == 0)

++ return 0;

++ else

++ {

++ errno = -EACCES;

++ return -1;

++ }

++ }

++}

四、编译内核

|  |
| --- |
| # 载入默认配置，openeuler-raspi\_defconfig配置文件 在 /arch/arm64/configs 下。  # make openeuler-raspi\_defconfig  # 编译构建并安装内核、模块和设备树  # make -j4 Image modules dtbs #耗时长，在树莓派中编译历时3-4小时  （使用时，先用 make help | grep Image 查看支持的Image，并替代命令中的Image）  # make modules\_install |

五、安装、升级内核

|  |
| --- |
| # 将内核放进引导  # cp arch/arm64/boot/Image /boot/kernel8.img  # 将设备树文件放进引导  # cp arch/arm64/boot/dts/broadcom/\*.dtb /boot/  # cp arch/arm64/boot/dts/overlays/\*.dtb\* /boot/overlays/  # cp arch/arm64/boot/dts/overlays/README /boot/overlays/ |

六、重启系统

|  |
| --- |
| # reboot |

七、测试验证新增的系统调用

syscal.c源码

#define \_GNU\_SOURCE

#include <unistd.h>

#include <sys/syscall.h>

#include<sys/types.h>

#include <stdio.h>

#define SYS\_NEWBABY 294

int main() {

uid\_t uid = getuid();

pid\_t pid = getpid();

pid\_t ppid = getppid();

long ret1 = syscall(SYS\_NEWBABY, pid, &uid);

long ret2 = syscall(SYS\_NEWBABY, ppid, &uid);

printf("ret1 is %ld, ret2 is %ld\n", ret1, ret2);

return 0;

}

|  |
| --- |
| # 创建并编译测试文件  # mkdir syscall && cd syscall  # vim syscall.c  # gcc -o syscall syscall.c  #执行syscall验证结果  # ./syscall  #查看系统日志  # dmesg | tail -n 5 |

