

项目一:向鸿蒙 Liteos 中加入一个自定义的系统调用

22920212204396

黄子安

一、实验目的

1、理解系统调用，向原操作系统中写入一个系统调用

二、实验环境

1.物理机: windows 操作系统

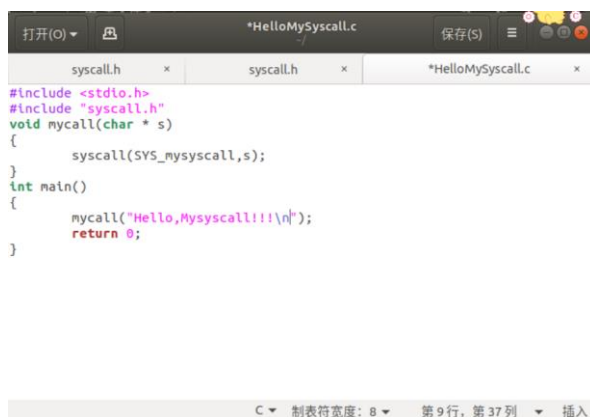
2.VMware 虚拟机: ubuntu 18.04.6

3.开发板: imx6ull Mini

三、实验内容

1、创建用户程序

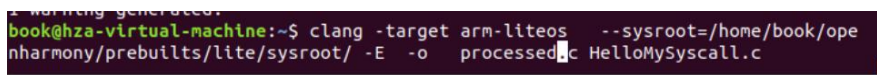
我们先创建一个用户态的程序，为了方便实现系统调用做一定的简化，直接将调用 `syscall` 的过程放在该程序当中，而不是通过 POSIX 接口加在 C 语言的库文件当中，其中 `SYS_mycall` 是系统调用号，将在后续的步骤中进行定义，`s` 作为参数传递到后续过程中



```
打开(O)  *HelloMySyscall.c 保存(S)
syscall.h x syscall.h x *HelloMySyscall.c x
#include <stdio.h>
#include "syscall.h"
void mycall(char * s)
{
    syscall(SYS_mysyscall,s);
}
int main()
{
    mycall("Hello,Mysyscall!!!\n");
    return 0;
}
```

2、增加对应的系统调用号

上述程序执行之后会运行 `syscall`，之后完成将参数传递，在此之前我们先寻找应该在哪个文件中定义新的系统调用号。为了方便起见，可以直接使用 `clang` 的 `-E` 参数输出预处理后的源文件，从而查看头文件中包含的文件具体是哪一个。



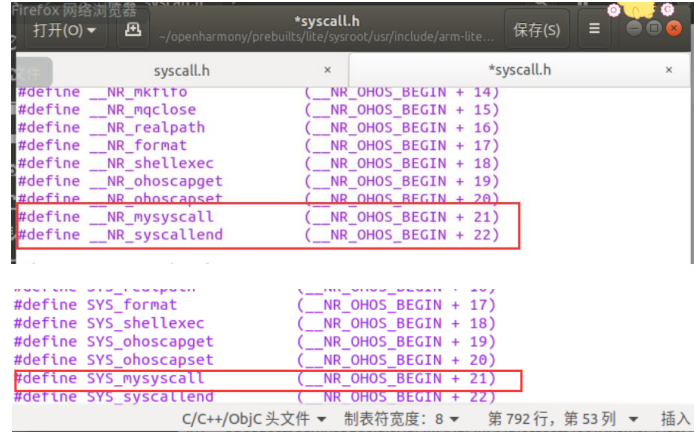
```
book@hza-virtual-machine:~$ clang -target arm-liteos --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/ -E -o processed.c HelloMySyscall.c
```

该命令生成了一个 `processed.c` 文件，打开这个文件拉到最底下，可以看到所包含的 `syscall.h` 的绝对路径



```
1 warning generated.
# 1 "/home/book/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/
bits/syscall.h" 1 3 4
# 5 "/home/book/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/
sys/syscall.h" 2 3 4
# 2 "/home/book/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/
syscall.h" 2 3 4
# 3 "HelloMySyscall.c" 2
void mycall(char * s)
{
    syscall(__NR_mysyscall,s);
}
int main()
{
    printf("%s","HelloMySyscall.c");
    mycall("Hello,Mysyscall!!!");
    return 0;
}
```

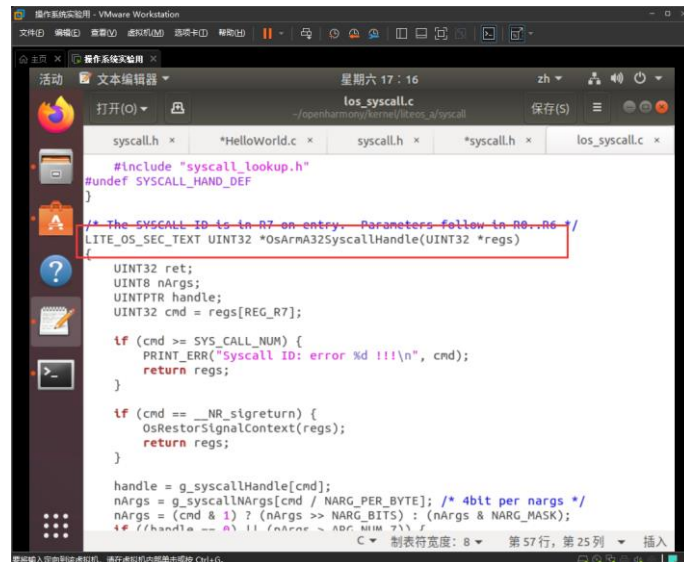
打开这个文件可以发现其中定义了大量的系统调用号，拉到最底下可以发现从 500 后面开始就是鸿蒙自己定义的系统调用号，和 ARM EABI 是不兼容的，我们将自己的系统调用号新建在此处，这里定义两个宏，一个是__NR 开头的用于内核，另一个 SYS 开头用于用户，两者的调用号相同，除此之外需要修改__NR_syscallend 和 SYS_syscallend，这两者会在后续步骤判断系统调用号是否越界



```
#define __NR_mkdir    (__NR_OHOS_BEGIN + 14)
#define __NR_mqclose  (__NR_OHOS_BEGIN + 15)
#define __NR_realpath (__NR_OHOS_BEGIN + 16)
#define __NR_format   (__NR_OHOS_BEGIN + 17)
#define __NR_shellexec (__NR_OHOS_BEGIN + 18)
#define __NR_ohoscapget (__NR_OHOS_BEGIN + 19)
#define __NR_ohoscapset (__NR_OHOS_BEGIN + 20)
#define __NR_mysyscall (__NR_OHOS_BEGIN + 21)
#define __NR_syscallend (__NR_OHOS_BEGIN + 22)

#define SYS_format    (__NR_OHOS_BEGIN + 17)
#define SYS_shellexec (__NR_OHOS_BEGIN + 18)
#define SYS_ohoscapget (__NR_OHOS_BEGIN + 19)
#define SYS_ohoscapset (__NR_OHOS_BEGIN + 20)
#define SYS_mysyscall (__NR_OHOS_BEGIN + 21)
#define SYS_syscallend (__NR_OHOS_BEGIN + 22)
```

接下来验证宏定义__NR_syscallend 为什么就是系统调用号的最大边界值，进入到./openharmy/kernel/liteos_a/syscall 目录中，之后打开 los_syscall.c 文件，在这里面定义了函数 OsArmA32SyscallHandle，该函数的作用是根据系统调用号查询对应的系统调用处理函数的具体实现并调用，传入的参数 regs 是汇编子程序段 osExceptSwtHdl 将寄存器的值写入到内核栈之后使用 TaskContext 这个结构体将该内存片段进行强制类型转换获得的，也就是说汇编程序把寄存器的值保存到了内核栈，然后为了方便访问内部数据使用结构体进行强转并使用结构体指针进行直接访问，其中 R7 寄存器存储的是系统调用号，所以 cmd 获得的就是系统调用号



```
/* The SYSCALL ID is in R7 on entry. Parameters follow in R0..R6 */
LITE_OS_SEC_TEXT UINT32 *OsArmA32SyscallHandle(UINT32 *regs)
{
    UINT32 ret;
    UINT8 nArgs;
    UINTPTR handle;
    UINT32 cmd = regs[REG_R7];

    if (cmd >= SYS_CALL_NUM) {
        PRINT_ERR("Syscall ID: error %d !!!\n", cmd);
        return regs;
    }

    if (cmd == __NR_sigreturn) {
        OsRestorSignalContext(regs);
        return regs;
    }

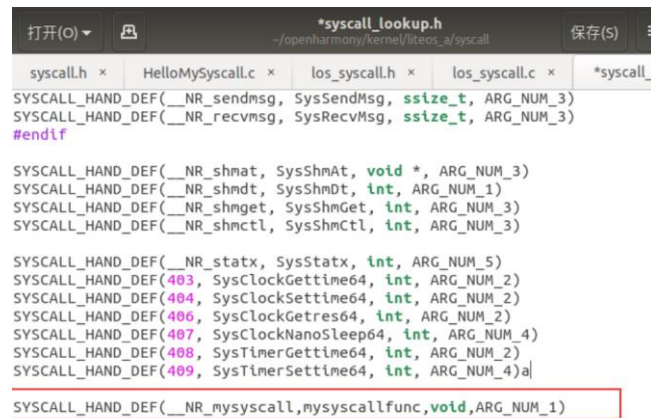
    handle = g_syscallHandle[cmd];
    nArgs = g_syscallNArgs[cmd / NARG_PER_BYTE]; /* 4bit per nargs */
    nArgs = (cmd & 1) ? (nArgs >> NARG_BITS) : (nArgs & NARG_MASK);
    if (!handle || !nArgs > 0) {
        PRINT_ERR("Syscall ID: error %d !!!\n", cmd);
        return regs;
    }
}
```

查看 SYS_CALL_NUM 的定义，也就证明了__NR_syscallend 是用于边界的判断，因此增加系统调用号的同时要给它加一

```
#define SYS_CALL_NUM    (__NR_syscallend + 1)
#define NARG_BITS       4
#define NARG_MASK       0x0F
#define NARG_PER_BYTE   2
```

3、建立映射

创建了系统调用号之后要做的就是创建对应的系统调用处理函数，进入到./openharmy/kernel/liteos_a/syscall 目录下所在的 syscall_lookup.h 文件中可以看到一个映射表，其中实现了系统调用编号和系统调用函数的映射关系，在此增加上我们的系统调用处理定义声明，如下图所示，返回值类型为 void，参数个数为 1，其中 mysyscallfun 是系统调用的具体实现部分



```
打开(O) 保存(S)
*syscall_lookup.h
~/openharmy/kernel/liteos_a/syscall

syscall.h x HelloMySyscall.c x los_syscall.h x los_syscall.c x *syscall_l

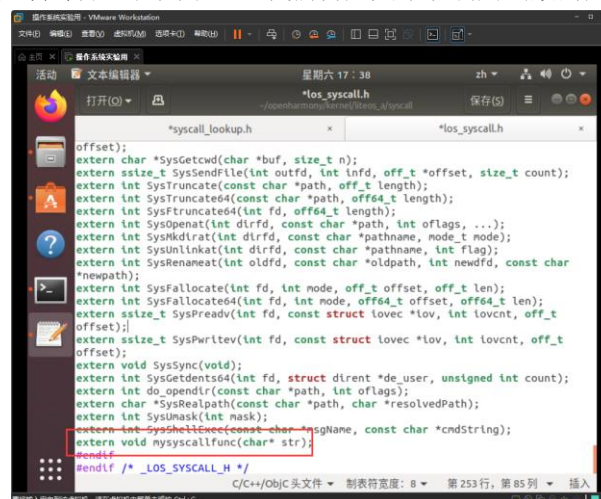
SYSCALL_HAND_DEF(_NR_sendmsg, SysSendMsg, ssize_t, ARG_NUM_3)
SYSCALL_HAND_DEF(_NR_recvmsg, SysRecvMsg, ssize_t, ARG_NUM_3)
#endif

SYSCALL_HAND_DEF(_NR_shmat, SysShmAt, void *, ARG_NUM_3)
SYSCALL_HAND_DEF(_NR_shmdt, SysShmDt, int, ARG_NUM_1)
SYSCALL_HAND_DEF(_NR_shmget, SysShmGet, int, ARG_NUM_3)
SYSCALL_HAND_DEF(_NR_shmctl, SysShmCtl, int, ARG_NUM_3)

SYSCALL_HAND_DEF(_NR_statx, SysStatx, int, ARG_NUM_5)
SYSCALL_HAND_DEF(403, SysClockGettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(404, SysClockSettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(406, SysClockGetres64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(407, SysClockNanoSleep64, int, ARG_NUM_4)
SYSCALL_HAND_DEF(408, SysTimerGettime64, int, ARG_NUM_2)
SYSCALL_HAND_DEF(409, SysTimerSettime64, int, ARG_NUM_4)

SYSCALL_HAND_DEF(_NR_mysyscall,mysyscallfunc,void,ARG_NUM_1)
```

进入到./openharmy/kernel/liteos_a/syscall 目录下的 los_syscall.h 文件中，可以看到声明了系统调用具体函数的声明部分，在其中加上我们自定义系统调用函数的声明

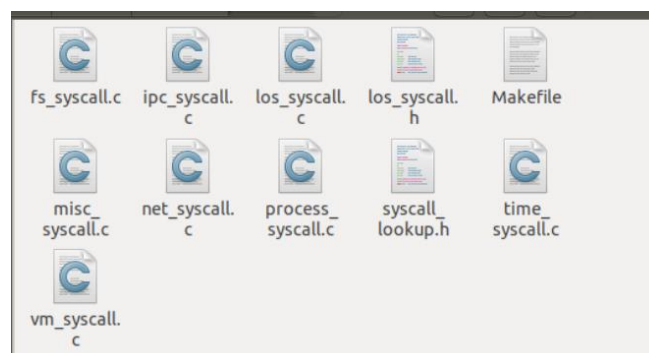


```
活动 文本编辑器
星期六 17:38 zh
打开(O) 保存(S)
*syscall_lookup.h x *los_syscall.h x

offset);
extern char *SysGetcwd(char *buf, size_t n);
extern ssize_t SysSendFile(int outfd, int infd, off_t *offset, size_t count);
extern int SysTruncate(const char *path, off_t length);
extern int SysFtruncate64(const char *path, off64_t length);
extern int SysOpenat(int dirfd, const char *path, int oflags, ...);
extern int SysMkdirat(int dirfd, const char *pathname, mode_t mode);
extern int SysUnlinkat(int dirfd, const char *pathname, int flag);
extern int SysRenameat(int oldfd, const char *oldpath, int newfd, const char *newpath);
extern int SysFallocate(int fd, int mode, off_t offset, off_t len);
extern int SysFallocate64(int fd, int mode, off64_t offset, off64_t len);
extern ssize_t SysPreadv(int fd, const struct iovec *iov, int iovcnt, off_t offset);
extern ssize_t SysPwritev(int fd, const struct iovec *iov, int iovcnt, off_t offset);
extern void SysSync(void);
extern int SysGetdents64(int fd, struct dirent *de_user, unsigned int count);
extern int do_opendir(const char *path, int oflags);
extern char *SysRealpath(const char *path, char *resolvedPath);
extern int SysUnmask(int mask);
extern int SysShellExec(const char *msgName, const char *cmdString);
extern void mysyscallfunc(char* str);
#endif
#endif /* _LOS_SYSCALL_H */

C/C++/ObjC 头文件 制表符宽度: 8 第 253 行, 第 85 列 插入
```

之后通过翻看同一级目录下的各个文件并根据文件的名称以及 extern 修饰，可以知道这些函数的具体实现都在这些文件当中，



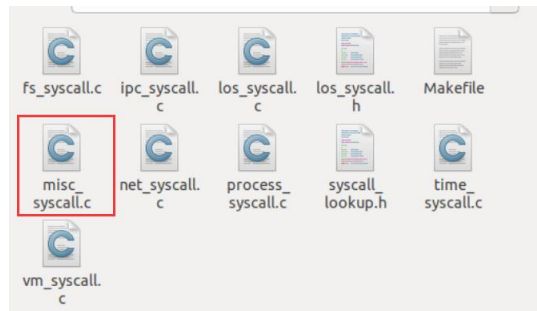
重新查看 los_syscall.c 文件，可以看到这里的 handle 作为一个函数指针，通过一个全局数组和系统调用号获得具体系统调用处理函数的指针，这里的 g_syscallHandl 数组保存了所有系统调

用入口函数的指针，该数组会在初始化的时候被赋值，最后根据参数个数通过 `switch` 语句完成调用，所以这里也就是具体函数的入口。

初始化的函数也位于该文件内，该函数会将其系统调用处理函数的指针存储到数组当中，这里还有一个数组 `g_syscallNArgs` 记录了参数的个数，因为参数的个数最多为 6 个，所以将一个 8 位的数组元素分成了两个 4 位来使用，起到一个哈希表的作用，快速获得当前系统调用处理函数的参数个数

```
void SyscallHandleInit(void)
{
#define SYSCALL_HAND_DEF(id, fun, rType, nArg) \
    if ((id) < SYS_CALL_NUM) \
    { \
        g_syscallHandle[(id)] = (UINTPTR) \
        (fun); \
        g_syscallNArgs[(id) / NARG_PER_BYTE] |= ((id) & 1) ? (nArg) << \
        NARG_BITS : (nArg); \
    } \
\
#include "syscall_lookup.h"
#undef SYSCALL_HAND_DEF
}
```

接下来去实现系统调用处理函数，在 `los_syscall.h` 中使用了 `extern` 修饰这些函数，代表函数实现在该文件外面，在编译的时候不会去检查对应的函数实现部分，而是在链接的时候由链接器来实现函数关联，这里有两种选择可以，一种是直接将函数实现部分放到同级目录里某个 `.c` 文件中，这样就无需修改构建时的设置；另一种是自己建一个 `.c` 文件但需要将其增加到对应的编译链接设置中使得可以找到新加的文件。这里没有找到 `BUILD.gn` 文件，暂时先考虑直接加在某个文件里，这里我们加到 `misc`（杂项）文件里，因为该文件就是用来存放不属于传统文件、网络、进程的调用



为了实现输出字符串，先引入下图中的头文件，之后在函数体中输出 `str` 字符串即可，这里的 `PRINTK` 调用类似于 `printf` 函数，在内核代码中可以使用它来输出各种调试信息、警告或错误消息。

```
#include "sys/utsname.h"
#include "user_copy.h"
#include "los_strncpy_from_user.h"
#include "capability_type.h"
#include "capability_api.h"
#include "los_printf.h"

return -EFAULT;
}
ret = uname(&tnpName);
if (ret < 0) {
    return ret;
}
ret = LOS_ArchCopyToUser(name, &tnpName, sizeof(struct utsname));
if (ret != 0) {
    return -EFAULT;
}
return ret;
}

void mysyscallfunc(char* str)
{
    PRINTK(str);
}
```

4.编译内核

同实验二使用 `make clean`、`make -j 8` 命令进行内核编译，之后使用 `clang` 交叉编译 `HelloMySyscall.c` 程序，将得到的结果复制到 `out/imx6ull/rootfs/bin` 中，使用 `make rootfs` 构建后将 `liteos.bin` 和 `rootfs.jffs2` 拖动到烧写工具中之后烧写到开发板内即可

```
book@hza-virtual-machine:~$ cd ~
book@hza-virtual-machine:~$ clang -target arm-liteos --sysroot=/home/book/ope
nharmony/prebuilts/lite/sysroot/ -o HelloMySyscall HelloMySyscall.c
HelloMySyscall.c:5:2: warning: implicit declaration of function 'syscall' is in
valid in C99 [-Wimplicit-function-declaration]
    syscall(SYS_mysyscall,s);
    ^
1 warning generated.
```

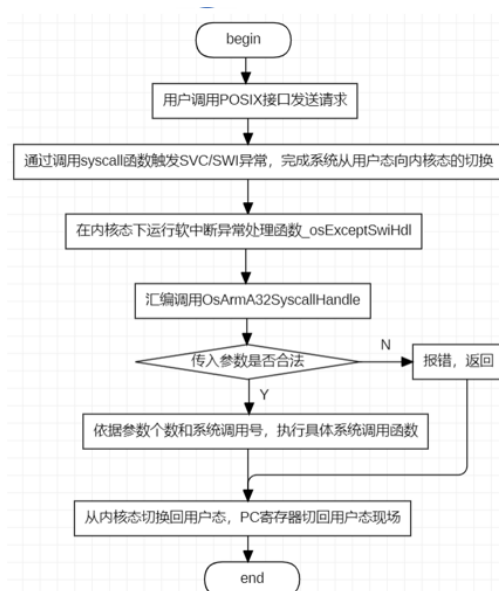
四、实验结果

启动开发板，之后在 MobaXterm 中连接对应的串口，运行对应的程序，可以看到成功输出了 `Hello, MySyscall!!!`，说明系统调用创建成功。

```
DeviceManagerStart end ...
[ERR]No console dev used.
[ERR]No console dev used.
OHOS # cd ./bin
OHOS # ls
Directory ./bin:
-rwxrwxr-x 13900 u:1001 g:1001 init
-rwxrwxr-x 26236 u:1001 g:1001 shell
-rwxrwxr-x 15116 u:1001 g:1001 HelloMySyscall
OHOS # cd ./HelloMySyscall
cd: Not a directory
OHOS # ./HelloMySyscall
OHOS # Hello,Mysyscall!!!
```

五、实验分析

本次实验的步骤分为编写用户程序、增加系统调用号、声明系统调用处理函数、实现系统调用处理函数，按照下面流程图所示的系统调用处理流程，逐步进行可以使得思路清晰步骤有序，关键还是在于理解系统调用、内核态用户态切换的过程。



六、实验总结

本次实验的重点是充分理解系统调用的过程，之后具体的步骤就会比较清晰，另外本实验

另一个难点就是找文件，比如 `syscall.h` 文件就有很多个，一开始将系统调用号加错了位置就导致没有成功，直到后面使用 `clang -E` 输出了预处理后的源文件找对了文件才成功完成，这次源码阅读与修改也感受到了宏定义高级使用的强大与简洁，在编译阶段就完成对应部分实现而不是使用函数调用大大加快了操作系统的运行速度，充分展现了 C 语言的简洁又强大。

七、参考文献

- [1] [OpenHarmony LiteOS-A 内核文档之学习--系统调用-开源基础软件社区-51CTO.COM](#)
- [2] [鸿蒙 OS 的系统调用是如何实现的？ | 解读鸿蒙源码-开源基础软件社区-51CTO.COM](#)
- [3] 理论课授课 PPT

八、附录

```
1. #include <stdio.h>
2. #include <syscall.h>
3. void mycall(char *s)
4. {
5.     syscall(SYS_mysyscall,s);
6. }
7. int main()
8. {
9.     mycall("Hello,Mysyscall!!!\n");
10.    return 0;
11.}
```