Project1 添加系统调用

姓名：任宇　　学号：33920212204567

一、 实验目的

向鸿蒙 Liteos 中加入一个自定义的系统调用。

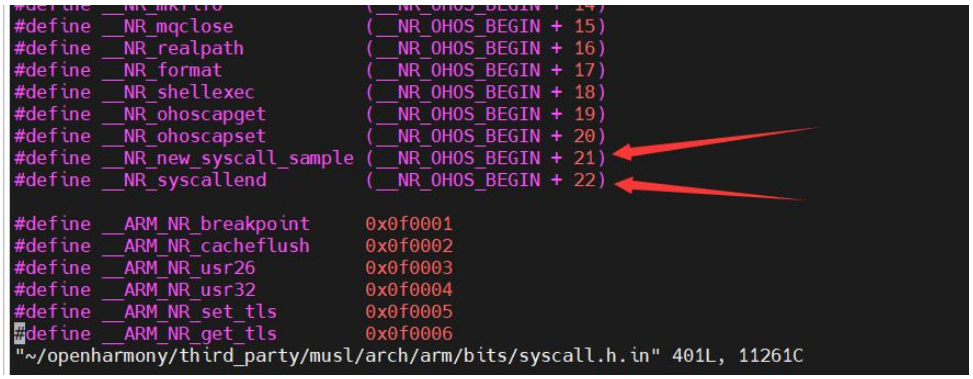
二、 实验环境

- 操作系统：
  - 主机：Windows 10
  - 虚拟机：Ubuntu 18.04
- 开发板：IMAX6ULL MIN
- 文件传输工具：FileZilla
- 终端工具：MobaXterm

三、 实验内容

1. **在 LibC 库中确定并添加新增的系统调用号：**

   编辑文件openharmony/third_party/musl/arch/arm/bits/syscall.h.in，如下所示，__NR_new_syscall_sample为新增系统调用号。需要注意同时更新系统调用的编号。



2. **在 LibC 库中新增用户态的函数接口声明及实现：**

在现成的一个源文件里增加函数实现代码，如在/openharmony/third_party/musl/src/aio/aio.c 文件中增加，如下图所示：

```
void newSyscallSample(int num)
{
     printf("user mode: num = %d\n",num);
     __syscall(SYS_new_syscall_sample,num);
     return;
}

static void *io_thread_func(void *ctx)
{
        struct aio_thread at, *p;

        struct aio_args *args = ctx;
```

3. **在内核系统调用头文件中新增系统调用号：**

如下所示，在/openharmony/third_party/musl/kernel/obj/include/bits/syscall.h 文件中，__NR_new_syscall_sample 为新增系统调用号。用户态代码和内核态代码增加系统调用号方式相同，编号相同。

```
#define __NR_OHOS_BEGIN          500
#define __NR_pthread_set_detach (__NR_OHOS_BEGIN + 0)
#define __NR_pthread_join       (__NR_OHOS_BEGIN + 1)
#define __NR_pthread_deatch     (__NR_OHOS_BEGIN + 2)
#define __NR_creat_user_thread  (__NR_OHOS_BEGIN + 3)
#define __NR_processcreat       (__NR_OHOS_BEGIN + 4)
#define __NR_processtart        (__NR_OHOS_BEGIN + 5)
#define __NR_printf             (__NR_OHOS_BEGIN + 6)
#define __NR_dumpmemory         (__NR_OHOS_BEGIN + 13)
#define __NR_mkfifo             (__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_new_syscall_sample (__NR_OHOS_BEGIN + 21)
#define __NR_syscallend         (__NR_OHOS_BEGIN + 22)

#define __ARM_NR_breakpoint    0x0f0001
#define __ARM_NR_cacheflush    0x0f0002
#define __ARM_NR_usr26        0x0f0003
```

在/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos/bits/syscall.h 中做同样修改，如下：

```
#define SYS_clone3              435
#define SYS_OHOS_BEGIN          500
#define SYS_pthread_set_detach  (__NR_OHOS_BEGIN + 0)
#define SYS_pthread_join        (__NR_OHOS_BEGIN + 1)
#define SYS_pthread_deatch      (__NR_OHOS_BEGIN + 2)
#define SYS_creat_user_thread   (__NR_OHOS_BEGIN + 3)
#define SYS_processcreat        (__NR_OHOS_BEGIN + 4)
#define SYS_processtart         (__NR_OHOS_BEGIN + 5)
#define SYS_printf              (__NR_OHOS_BEGIN + 6)
#define SYS_dumpmemory          (__NR_OHOS_BEGIN + 13)
#define SYS_mkfifo              (__NR_OHOS_BEGIN + 14)
#define SYS_mqclose             (__NR_OHOS_BEGIN + 15)
#define SYS_realpath            (__NR_OHOS_BEGIN + 16)
#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_new_syscall_sample  (__NR_OHOS_BEGIN + 21)
#define SYS_syscallend          (__NR_OHOS_BEGIN + 22)
```

在/openharmony/kernel/liteos_a/syscall/syscall_lookup.h 中，增加一行，如下。

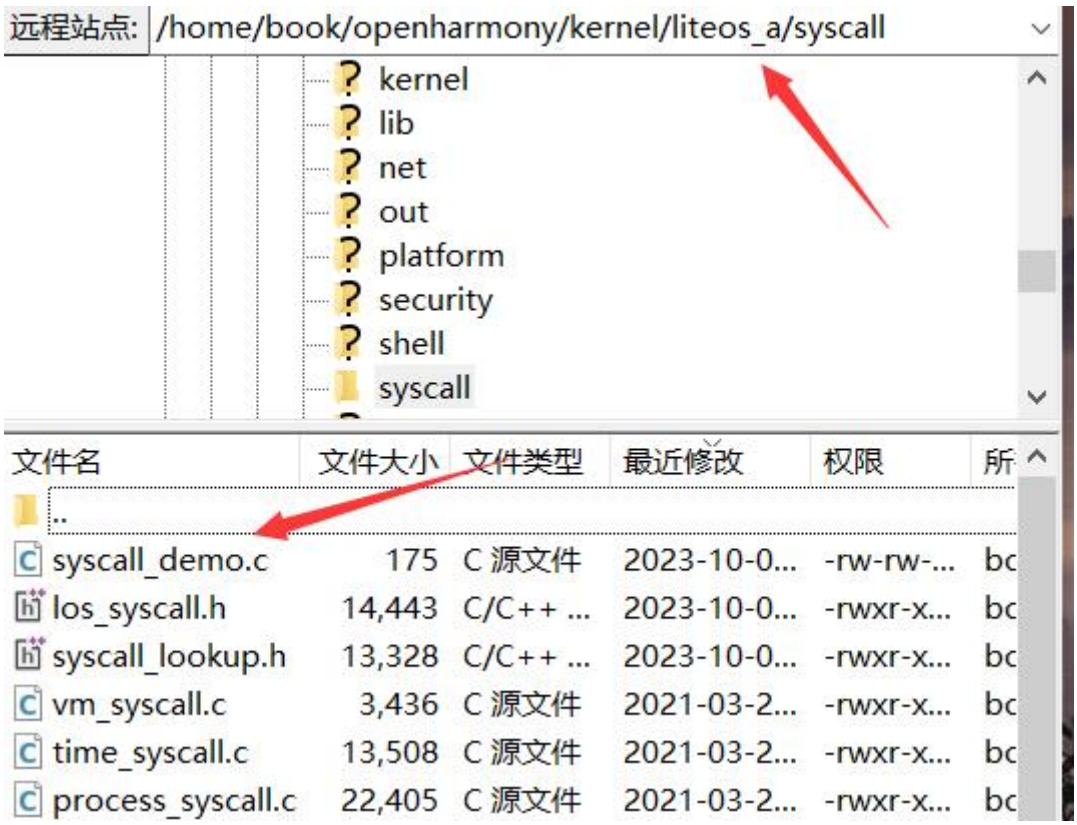
```
SYSCALL_HAND_DEF(__NR_chown32, SysChown, int, ARG_NUM_3)
#ifdef LOSCFG_SECURITY_CAPABILITY
SYSCALL_HAND_DEF(__NR_ohoscapget, SysCapGet, UINT32, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_ohoscapset, SysCapSet, UINT32, ARG_NUM_1)
#endif

SYSCALL_HAND_DEF(__NR_new_syscall_sampSYS,SysNewSyscallSample,void,ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_mmap2, SysMmap, void*, ARG_NUM_6)
SYSCALL_HAND_DEF(__NR_getuid32, SysGetUserID, int, ARG_NUM_0)
SYSCALL_HAND_DEF(__NR_getgid32, SysGetGroupID, unsigned int, ARG_NUM_0)
SYSCALL_HAND_DEF(__NR_geteuid32, SysGetEffUserID, int, ARG_NUM_0)
SYSCALL_HAND_DEF(__NR_getegid32, SysGetEffGID, unsigned int, ARG_NUM_0)
SYSCALL_HAND_DEF(__NR_getresuid32, SysGetRealEffSaveUserID, int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_getresgid32, SysGetRealEffSaveGroupID, int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_setresuid32, SysSetRealEffSaveUserID, int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_setresgid32, SysSetRealEffSaveGroupID, int, ARG_NUM_3)
SYSCALL_HAND_DEF(__NR_setreuid32, SysSetRealEffUserID, int, ARG_NUM_2)
SYSCALL_HAND_DEF(__NR_setregid32, SysSetRealEffGroupID, int, ARG_NUM_2)
SYSCALL_HAND_DEF(__NR_setgroups32, SysSetGroups, int, ARG_NUM_2)
SYSCALL_HAND_DEF(__NR_getgroups32, SysGetGroups, int, ARG_NUM_2)
SYSCALL_HAND_DEF(__NR_setuid32, SysSetUserID, int, ARG_NUM_1)
SYSCALL_HAND_DEF(__NR_setgid32, SysSetGroupID, int, ARG_NUM_1)
```

4. **在内核中新增系统调用对应的处理函数：**

需要在内核中新增系统调用函数声明及函数实现，并加入编译构建文件。首先，如下所示，修改/openharmony/kernel/liteos_a/syscall/los_syscall.h 文件，SysNewSyscallSample 为新增系统调用的内核处理函数声明。

```
extern ssize_t preadv(int __fd, const struct iovec *__iov, int __count, off_t __offset);
extern ssize_t pwritev(int __fd, const struct iovec *__iov, int __count, off_t __offset);
extern int chattr(const char *pathname, struct IATTR *attr);

extern void SysNewSyscallSample(int num);

extern int SysClose(int fd);
extern ssize_t SysRead(int fd, void *buf, size_t nbytes);
extern ssize_t SysWrite(int fd, const void *buf, size_t nbytes);
extern int SysOpen(const char *path, int oflags, ...);
```

然后在/openharmony/kernel/liteos_a/syscall 目录下新建源文件 syscall_demo.c，新增系统调用的内核处理函数实现。




```c
#include "los_printf.h"

void SysNewSyscallSample(int num)
{
    PRINTK("kernel mode: num = %d\n", num);
    return;
}
```

最后，在文件 kernel/liteos_a/syscall/BUILD.gn 中增加对 syscall_demo.c 源文件的编译管理。

BUILD.gn

C: > Users > ayu > Desktop > BUILD.gn

```
import("//kernel/liteos_a/liteos.gni")

module_switch = defined(LOSCFG_KERNEL_SYSCALL)
module_name = get_path_info(rebase_path("."), "name")
kernel_module(module_name) {
  sources = [
    "fs_syscall.c",
    "los_syscall.c",
        "net_syscall.c",
    "syscall_demo.c",
   "time_syscall.c",
  "ipc_syscall.c",
  "misc_syscall.c",
   "process_syscall.c",
  "vm_syscall.c",
    "syscall_demo.c"
  ]
}
```

5. 编译内核：

使用 make -j 8 命令重新编译内核，使新增加的系统调用生效。

```
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ make  -j  8
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a”
/home/book/openharmony/kernel/liteos_a/tools/menuconfig/conf --silentoldconfig /home/book/openharmony/kernel/liteos_a/Kconfig
mv -f /home/book/openharmony/kernel/liteos_a/include/generated/autoconf.h /home/book/openharmony/kernel/liteos_a/platform/include/menuconfig.h
make[1]: 离开目录“/home/book/openharmony/kernel/liteos_a”
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a/arch/arm/arm”
src/startup/reset_vector_up.S:138:2: warning: deprecated since v7, use 'dsb'
 mcr p15, 0, r0, c7, c10, 4 @ DSB
 ^
src/startup/reset_vector_up.S:139:2: warning: deprecated since v7, use 'isb'
 mcr p15, 0, r0, c7, c5, 4 @ ISB
 ^
make[1]: 离开目录“/home/book/openharmony/kernel/liteos_a/arch/arm/arm”
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a/platform”
make[1]: 离开目录“/home/book/openharmony/kernel/liteos_a/platform”
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a/kernel/common”
```

```
t/imx6ull/liteos.bin
/home/book/llvm/bin/../bin/llvm-objdump -t /home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos |sort >/home/book/openharmony/kernel/liteos_a/out/imx6ul
l/liteos.sym.sorted
/home/book/llvm/bin/../bin/llvm-objdump -d /home/book/openharmony/kernel/liteos_a/out/imx6ull/liteos >/home/book/openharmony/kernel/liteos_a/out/imx6ull/lite
os.asm
make[1]: 进入目录“/home/book/openharmony/kernel/liteos_a/apps”
make[2]: 进入目录“/home/book/openharmony/kernel/liteos_a/apps/shell”
make[2]: 离开目录“/home/book/openharmony/kernel/liteos_a/apps/shell”
make[2]: 进入目录“/home/book/openharmony/kernel/liteos_a/apps/init”
make[2]: 离开目录“/home/book/openharmony/kernel/liteos_a/apps/init”
make[1]: 离开目录“/home/book/openharmony/kernel/liteos_a/apps”
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ cd  /home/book/doc_and_source_for_openharmony/apps/hello
```

6. 调用并验证：

对实验二中的 hello 程序进行修改，使其调用新增的系统调用，如图：

```c
#include <stdio.h>
#include <syscall.h>

void newSyscallSample(int num)
{
        printf("user mode: num = %d\n",num);
        syscall(SYS_new_syscall_sample,num);
         return;
}



int main(void)
{
         printf("\nHello, harmony!\n\n");
         newSyscallSample(10);
         return 0;
}
```

编译 hello 程序，并将其加载到开发板中：

```
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cd  /home/book/doc_and_source_for_openharmony/apps/hello
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ clang -target arm-liteos   --sysroot=/home/book/openharmony/prebuilts/lite/sysroot/
o  hello   hello.c
hello.c:7:8: warning: implicit declaration of function 'syscall' is invalid in C99 [-Wimplicit-function-declaration]
       syscall(SYS_new_syscall_sample,num);
       ^

1 warning generated.
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ ls
hello  hello.c  Makefile
```

```
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ cp  out/imx6ull/rootfs.img out/imx6ull/rootfs.jffs2
book@ry-virtual-machine:~/openharmony/kernel/liteos_a$ cd  /home/book/doc_and_source_for_openharmony/apps/hello
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cp  hello  /home/book/openharmony/kernel/liteos_a/out/imx6ull/rootfs/bin
book@ry-virtual-machine:~/doc_and_source_for_openharmony/apps/hello$ cd  /home/book/openharmony/kernel/liteos_a/out/imx6ull/
book@ry-virtual-machine:~/openharmony/kernel/liteos_a/out/imx6ull$ mkfs.jffs2  -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
```

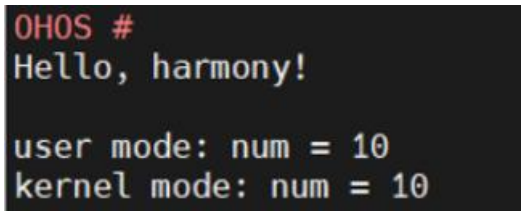
在开发板中运行 hello 程序，可见程序成功执行系统调用，实验成功：

```
OHOS # ./bin/hello
OHOS #
Hello, harmony!

user mode: num = 10
kernel mode: num = 10

OHOS #
```

四、 实验结果

本实验旨在向鸿蒙 Liteos 中新增一个系统调用。通过对鸿蒙系统的研究，我确定了在其内核中插入新调用的合适位置。新加入的系统调用的功能是打印一个数字。实验的主要步骤包括设计、编码和测试该调用。在测试环节，新的系统调用成功实现了预期

的功能，如下图所示：

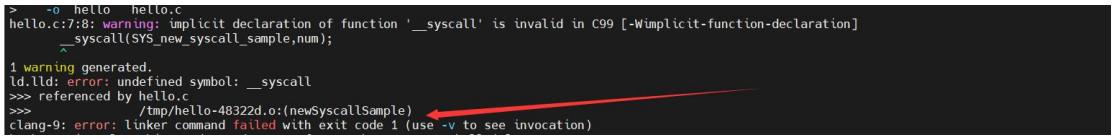


五、 实验分析

在添加新的系统调用之前，应先了解系统调用， LiteOS_A 内核实现用户态与内核态的区分隔离，用户态程序不能直接访问内核资源，而系统调用则为用户态程序提供了一种访问内核资源、与内核进行交互的通道。

系统调用过程为：调用触发—>上下文切换—>参数传递—>系统调用分发—>执行系统调用—>返回结果—>返回用户空间。因此，需要添加新增的系统调用号，并新增用户态的函数接口声明及实现。接着在内核系统调用头文件中确定并添加新增的系统调用号及对应内核处理函数的声明，并在内核中新增该系统调用对应的内核处理函数。完成以上步骤后，在程序中调用该新增的系统调用，并验证实验是否成功。

在实验过程中，遇到一个问题，即如果在修改内核后不重新编译内核而是直接编译程序，那么就会报错：



解决方法就是先编译内核，再编译程序。

六、 实验总结

本次实验目标是在鸿蒙操作系统中添加一个新的系统调用。

首先，需要在鸿蒙的源代码中定位合适的位置来插入新的系统调用。这一过程需要对 Liteos_a 的内部结构有深入的了解，以确保新的系统调用不会影响到其他部分的功能。在代码实现之后，又进行了测试，确保新的系统调用能够正常工作。

总的来说，这次实验增强了我对操作系统工作原理的理解，特别是如何在一个现有的系统中添加新功能。Liteos 的灵活性和模块化设计为我学习操作系统提供了宝贵的实践经验。

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

八、 附录

1. syscall_demo.c ：

```
#include "los_printf.h"
void SysNewSyscallSample(int num)
{
    PRINTK("kernel mode: num = %d\n", num);
    return;
 }
```

2. BUILD.gn ：

```
import("//kernel/liteos_a/liteos.gni")
module_switch = defined(LOSCFG_KERNEL_SYSCALL)
 module_name = get_path_info(rebase_path("."), "name")
       kernel_module(module_name) {
         sources = [
          "fs_syscall.c",
          "los_syscall.c",
          "net_syscall.c",
          "syscall_demo.c",
         "time_syscall.c",
         "ipc_syscall.c",
```

```
        "misc_syscall.c",
        "process_syscall.c",
        "vm_syscall.c",
        "syscall_demo.c"
        ]
}
```

3. hello.c ：

```
#include <stdio.h>
#include <syscall.h>

void newSyscallSample(int num)
{
        printf("user mode: num = %d\n",num);
        syscall(SYS_new_syscall_sample,num);
         return;
 }


int main(void)
{
         printf("\nHello, harmony!\n\n");
         newSyscallSample(10);
         return 0;
}
```