LAB 4 系统调用入门

📚 实验要求

- 实现功能为读取字符串并打印的系统调用echo(char *str,int len)，指定系统调用号为666
- 在用户态编写测试程序test_echo，实现输出字符串“Test String"

🔥 实验步骤

1.在系统调用表注册系统调用号、调用名称、实现函数

>> vim ./entry/syscall.tbl

```
########### Note ##############
# add syscall here
666     echo        sys_echo    //here
# (`__sys_dummy` means unimplemented)
# id    name        impl_name
############Proc###############
  1     fork        sys_fork
  3     wait        sys_wait
 93     exit        sys_exit
```

2.定义系统调用具体实现

> *一些提示(引用自题目描述)*

> 1. *使用`printf`来打印输出，如果遇到函数提示报红，未定义等问题，可以尝试`include "common.h"`来解决。*
> 2. *通常情况下，我们返回0表示系统调用成功，否则失败。*
> 3. *可以使用`argXXX`系列函数来帮助你获取系统调用的参数，它们被定义在`include/kernel/syscall.h`中。*
> 4. *我们无法直接访问用户态传入的指针所指向的内存位置，可以使用`copy_from_user`函数来，参见`include/mm/vm.h`*

🔥**实验中发现printf不在common.h中，此处应使用#include "printf.h"**

```
[root@ecs-e959:/labsyscall/src# cd ./include
[root@ecs-e959:/labsyscall/src/include# ls
common.h  debug.h  gcc_attr.h  printf.h  sbi.h  utils.h
[root@ecs-e959:/labsyscall/src/include# cat common.h
#ifndef _H_COMMON_
#define _H_COMMON_

#include <stdarg.h>
#include "param.h"
#include "memlayout.h"
#include "sys/error.h"
#include "types.h"
#include "utils.h"
#include "str.h"

extern volatile int panicked;

#endif
```

>> vim ./src/sys_echo.c

```c
#include "printf.h"
#include "mm/vm.h"
#include "kernel/syscall.h"
#include "kernel/waitqueue.h"
//实现读取用户态传入的字符串并打印
uint64_t sys_echo() {
    uint64_t user_addr; //用户空间传入的字符串地址。
    int len;
```

```
    if (argaddr(0, &user_addr) < 0) { //获取系统调用参数1（字符串地址），失败则返回-1。
        return -1;
    }
    if (argint(1, &len) < 0) { //获取参数2（字符串长度），同样做错误检查
        return -1;
    }
    if (len <= 0 || len > 255) { //检查长度在合法范围
        return -1;
    }

    char kbuf[256] = {0};
    if (copy_from_user(kbuf, user_addr, len) != 0) { //完成从用户态到内核态的内存拷贝
        return -1;
    }
    kbuf[len] = '\0'; //手动给字符串末尾加上 \0以防越界
    printf("%s\n", kbuf);  //也可以使用kprintf等

    return 0;
}
```

```
//./src/mm/copy.c
//内存拷贝函数的实现
int copy_from_user(void *to, uint64_t from, size_t n) {
    //省略一段安全性检查
    char *s = (char *)from;  //强制转换成字节
    char *d = (char *)to;
    if (s < d && s + n > d) { //判断源地址 s 和目标地址 d 是否存在重叠且目标地址在原地址之后
        s += n;                 //将指针移到末尾并从后往前拷贝,防止源字符串提前被覆盖
        d += n;
        while (n-- > 0)
            *--d = *--s;
    } else {
        while (n-- > 0)         //安全情况，正常从前往后拷贝
```

```
            *d++ = *s++;
    }

    return 0;
}
```

3.编辑Makefile文件，添加编译链接

>> vim ./src/Makefile

```
obj-y += sys_echo.o
```

4.编写用户态程序 测试系统调用

>> vim ./user/src/test_echo.c

```
#include "string.h"
#include "syscall.h"

// 你的系统调用号
#define SYS_ECHO 666

int main(int argc, char **argv) {
    char *str = "Test String";
    int len = strlen(str);
    syscall(SYS_ECHO, str, len);
    return 0;
}
```

5.运行内核，执行测试程序

>> chmod +x run-qemu.sh

>> ./run-qemu.sh

>> ls

>> ./test_echo ✅

```
[rustsbi] pmp12: 0xc000000 ..= 0xc3fffff (rw-)
[rustsbi] enter supervisor 0x80200000

.bss 0x8022e3f0-0x802f2908 (0xc4518) cleared

tataka os
hart 0 start
Lab 4 - Syscall Introduction

Kernel starts successfully!
Edit user/src/init0.c to run your program.
Enjoy it!
child welcome exited with 0
/$ ls
arg              brk              chdir            clone            close

dup              dup2             execve           fork             getfree

getpid           gettimeofday     init0            ls               mkdir_

mmap             munmap           open             pipe             sh

sleep            stack_overflow   test_echo        thread           times

wait             waitpid          welcome          write            yield


/$ ./test_echo
Test String
```

💡 思考

📌 系统调用参数具体是如何获取的?

:利用特定寄存器

>>cat ./user/lib/arch/riscv/syscall_arch

```
//用户态存入
#define __asm_syscall(...)                  \
    __asm__ __volatile__("ecall\n\t"    \   //触发系统调用
                         : "=r"(a0)     \   //返回值存放在 a0 寄存器中
                         : __VA_ARGS__  \   //传入的寄存器输入参数,
a1,a2,a3...
                         : "memory");   \   //表示这段汇编可能影响内存，防
止编译器优化导致错误
    return a0;
```

```
//带两个参数的syscall  (usr_addr,len)
static inline long __syscall2(long n, long a, long b)
{
    register long a7 __asm__("a7") = n;
    register long a0 __asm__("a0") = a;
    register long a1 __asm__("a1") = b;
    __asm_syscall("r"(a7), "0"(a0), "r"(a1)) //"0"表示复用输出寄存器
a0 作为输入
}
```

>>cat ./src/kernel/syscall.c

```
//内核态读取
 32 static uint64_t argraw(int n) {
 33     struct proc *p = myproc();
 34     switch (n) {
 35     case 0:
 36         return proc_get_tf(p)->a0;
 37     case 1:
 38         return proc_get_tf(p)->a1;
```

```
39      case 2:
40          return proc_get_tf(p)->a2;
41      case 3:
42          return proc_get_tf(p)->a3;
43      case 4:
44          return proc_get_tf(p)->a4;
45      case 5:
46          return proc_get_tf(p)->a5;
47      }
48      panic("argraw");
49      return -1;
50  }
51
52  int argint(int n, int *ip) {
53      *ip = argraw(n);
54      return 0;
55  }
56
57  int argaddr(int n, uint64_t *ip) {
58      *ip = argraw(n);
```

📌 在`syscall.tbl`中添加表项生成系统调用声明的原理

1.根据syscall.tbl生成syscall_gen.h
2.组合使用#include和宏，映射系统调用号与功能、名称.....

>> cat ./include/generated/syscall_gen.h

>> cat ./src/kernel/syscall.c

```
// Syscall Declaration 生成函数声明便于外部调用(extern uinnt64_4
sys_echo(void))
#define __SYS_CALL(NUM, NAME, FUNC) extern uint64_t FUNC(void);
```

```
#include "generated/syscall_gen.h"
#undef __SYS_CALL

// Syscall Table
// 生成系统调用跳转表syscalls[]，实现根据系统调用号执行功能 syscalls[666]--执行sys_echo()
#define __SYS_CALL(NUM, NAME, FUNC) [NUM] FUNC,
static uint64_t(*syscalls[])(void) = {
    #include "generated/syscall_gen.h"
};
#undef __SYS_CALL

// Syscall Name Map  生成syscall_names[]名称映射表
#define __SYS_CALL(NUM, NAME, FUNC) [NUM] #NAME, //#转成字符串“echo”
static char *__syscall_names[] = {
    #include "generated/syscall_gen.h"
};
#undef __SYS_CALL
```

📌 在RISC-V指令集架构中，如何调用系统调用

用户态：C语言中调用 syscall 函数,如syscall(666, ptr, len),将参数放入寄存器中并触发ecall。（使用汇编调用同理，li a*，ecall）

内核处理流程

>> cat ./src/kernel/trap.c

```
 32 /**
 33  * @brief handle an interrupt, exception, or system call from user space.
 34  */
 35 void usertrap(void) {
    ///省略安全性检查等
```

```
65      switch (scause) {
66          when_syscall {
67              // sepc points to the ecall instruction,
68              // but we want to return to the next instruction.
69              p->trapframe->epc += 4;  //跳过ecall指令
70              p->stub_time = ticks;
71              intr_on(); //打开中断响应
72              syscall(); //系统调用
73              p->s_time += ticks - p->stub_time;
74              break;
75          }
```

>> cat ./src/kernel/syscall.c

```
void syscall(void) {
    struct proc *p = current; //获取当前运行的进程 current
    int num = proc_get_tf(p)->a7; //从该进程中断帧的寄存器a7中获取系统调
用号num
    uint64_t epc = read_csr(sepc); //读取当前异常发生的 PC，用于后续恢复

    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        uint64_t ret = syscalls[num](); //执行对应系统调用函数
        proc_get_tf(p)->a0 = ret;  //将系统调用的返回值写回 a0 寄存器中，
供用户态使用。
    } else {
        kprintf("PID %d %s: "rd("unknown sys call %d")" sepc
%lx\n",
            p->pid, p->name, num, epc);
        proc_get_tf(p)->a0 = -ENOSYS;
    }
}
```