

# 实验一 system call

PB20050987 梁兆懿

在实验0中，我们初步学习了系统调用。实验一要求我们添加两个新的系统调用功能：系统跟踪调用以及正在运行系统的信息打印。

实验需要切换到系统调用分支：

```
$ git fetch
$ git checkout syscall
$ make clean
```

## System call tracing

在该实验中，我们需要实现系统跟踪功能，修改 xv6 内核以在每个系统调用即将返回时打印出一行。该行应包含进程 ID、系统调用名称和返回值;无需打印系统调用参数。

要求实现的输出为：

```
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 966
3: syscall read -> 70
3: syscall read -> 0
$
$ trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 966
4: syscall read -> 70
4: syscall read -> 0
4: syscall close -> 0
$
$ grep hello README
$
$ trace 2 usertests forkforkfork
usertests starting
test forkforkfork: 407: syscall fork -> 408
408: syscall fork -> 409
409: syscall fork -> 410
410: syscall fork -> 411
409: syscall fork -> 412
410: syscall fork -> 413
409: syscall fork -> 414
411: syscall fork -> 415
...
$
```

在第一个实验中，提示较多，我们可以根据提示一步一步进行。

首先在Makefile中增加\$U/\_trace，从而实现trace.c的调用。

```
$U/_trace\
```

随后如实验0，在系统调用相关文件完成`user/user.h`、`user/usys.pl`、`kernel/syscall.h`、`kernel/syscall.c`中新增关于`trace`的声明。

```
//user.h
int trace(int); // added by me.
//usys.pl
entry("trace"); #added by me
//syscall.h
#define SYS_trace 22
//syscall.c
extern uint64 sys_trace(void);

[SYS_trace]    sys_trace,
```

查阅得知，`usys.S`可以定义`SYSCALL(name)`的汇编代码，`syscall.h`可以定义system call numbers，`syscall.c`可以通过`fetchaddr`获取当前进程的`uint64`地址。

然后，我们在`kernel/sysproc.c`中定义系统调用`uint64 sys_trace(void)`，并在结构体`proc`中增加变量来记录`mask`

```
uint64 sys_trace(void)
{
    int n;
    if(argint(0, &n) < 0)
        return -1;
    myproc()->tracemask = n;
    return 0;
}
//proc
int tracemask;
```

我们通过修改`fork`，保证父进程`mask`的继承

```
// np->trace_mask = p->trace_mask;
```

最后在`syscall.c`中增加打印信息的函数以及数组存储系统调用名。（没看到提示用数组存储系统调用名，搞了很久才发现~）

```
static char syscall_names[23][16] = {"fork", "exit", "wait", "pipe", "read",
    "kill", "exec", "fstat", "chdir", "dup", "getpid", "sbrk", "sleep", "uptime",
    "open", "write", "mknod", "unlink", "link", "mkdir", "close", "trace",
    "sysinfo"}; //后面加上sysinfo

void syscall(void) {
    int num;
    struct proc *p = myproc();
    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        p->trapframe->a0 = syscalls[num]();
        if(p->tracemask > 0 && (p->tracemask & (1<<num)))
```

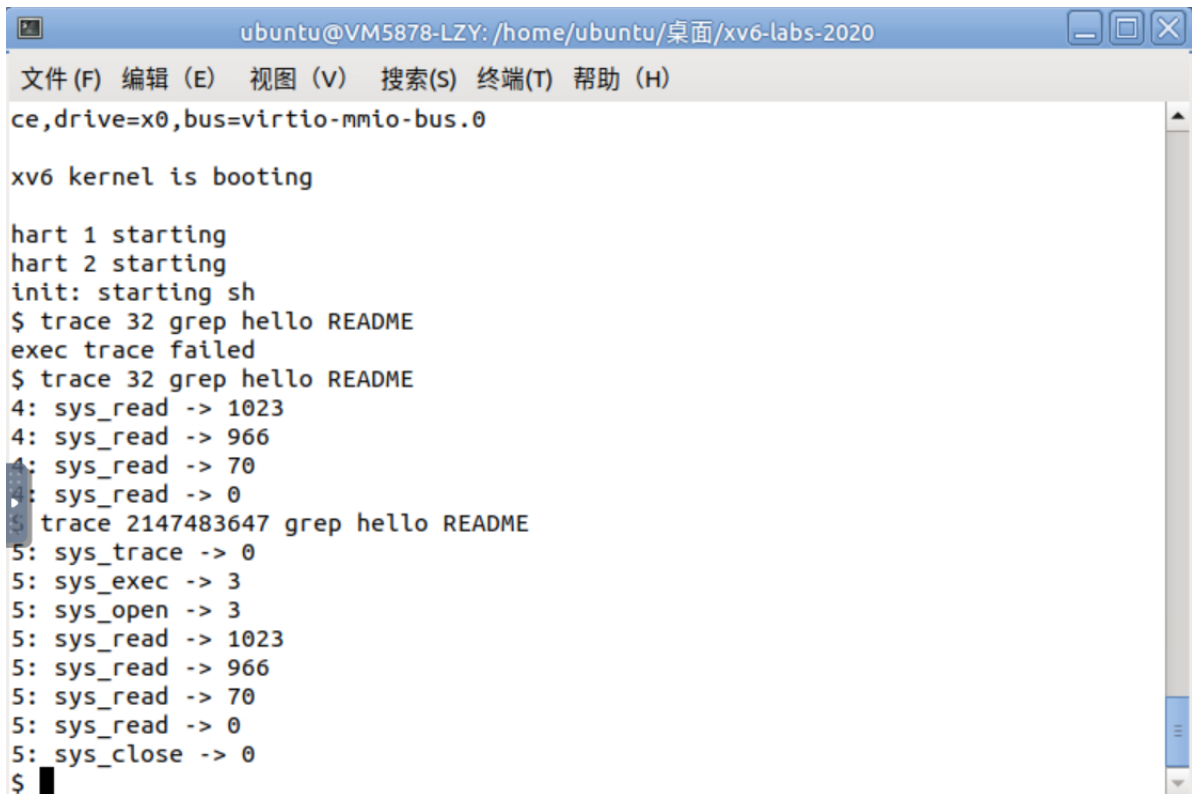
```

    {
        printf("%d: sys_%s -> %d\n", p->pid, syscall_names[num-1], p->trapframe->a0); // -1 对应上名字
    }
} else {
    printf("%d %s: unknown sys call %d\n",
        p->pid, p->name, num);
    p->trapframe->a0 = -1;
}
}
}

```

## 实验结果

最后，实验结果如下：



```

ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ trace 32 grep hello README
exec trace failed
$ trace 32 grep hello README
4: sys_read -> 1023
4: sys_read -> 966
4: sys_read -> 70
4: sys_read -> 0
$ trace 2147483647 grep hello README
5: sys_trace -> 0
5: sys_exec -> 3
5: sys_open -> 3
5: sys_read -> 1023
5: sys_read -> 966
5: sys_read -> 70
5: sys_read -> 0
5: sys_close -> 0
$

```

```
ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
9: sys_fork -> 49
8: sys_fork -> 50
9: sys_fork -> 51
8: sys_fork -> 52
9: sys_fork -> 53
8: sys_fork -> 54
8: sys_fork -> 55
9: sys_fork -> 56
8: sys_fork -> 57
9: sys_fork -> 58
8: sys_fork -> 59
9: sys_fork -> 60
9: sys_fork -> 61
8: sys_fork -> 62
9: sys_fork -> 63
9: sys_fork -> 64
8: sys_fork -> 65
9: sys_fork -> 66
8: sys_fork -> 67
9: sys_fork -> -1
OK
5: sys_fork -> 68
ALL TESTS PASSED
$ █
```

## 实验感受

经历了实验0的洗礼，实验一的第一问可以说——还是写不动，做了一个晚上。在实验中学到了许多头文件的含义以及功能，并且对自己的粗心大意（不看提示）付出了两个小时的代价。不过完成之后回过头来，感觉难度也不是原本以为那么大，还是由于自己编程功能比较薄弱，以后需要多加练习。

## Sysinfo (moderate)

该实验要求添加系统调用sysinfo，收集正在运行的系统的信息。需要设置一个参数：指向结构sysinfo的指针。内核应填写此结构的字段：freemem字段设置为可用内存的字节数，nproc字段设置为状态为UNUSED的进程数。

## 实验代码

首先，如同上一个题目，我们声明测试函数sysinfotest，并且对sysinfo进行相应的声明：

```
//Makefile
$U/_sysinfotest\
//user.h
struct sysinfo;
int sysinfo(struct sysinfo*);
//usys.pl
entry("sysinfo");
//syscall.h
#define SYS_sysinfo 23
//syscall.c
extern uint64 sys_sysinfo(void);

[SYS_sysinfo] sys_sysinfo,
```

借助提示，我们参考第四章kernel/sysfile.c/sys\_fstat 和 kernel/file.c/filestat，将内核数据传输到用户态中。在函数syscall.c输入：

```

uint64 sys_sysinfo(void)
{
    uint64 addr;
    if(argaddr(0, &addr) < 0)
        return -1;

    struct proc *p = myproc();
    struct sysinfo info;
    info.freemem = freemem();
    info.nproc = nproc();
    if(copyout(p->pagetable, addr, (char*)&info, sizeof(info)) < 0)
        return -1;
    return 0;
}

```

然后在 *kernel/kalloc.c*、*proc.c* 中实现收集数据的函数

```

//kalloc.c
uint64 freemem(void){
    uint64 freemem = 0;
    struct run *r;
    acquire(&kmem.lock);
    for(r = kmem.freelist; r ; r = r->next){
        freemem +=PGSIZE;
    }
    release(&kmem.lock);
    return freemem;
}

//proc.c
uint64 nproc(void){
    struct proc *p;
    uint64 nproc = 0;
    for(p = proc ; p < &proc[NPROC]; p++){
        if(p->state != UNUSED){
            nproc++;
        }
    }
    return nproc;
}

```

并且需要在def.h中声明相关函数：

```

uint64      freemem(void);
uint64      nproc(void);

```

此外，还需要声明结构体，在user/user.h：

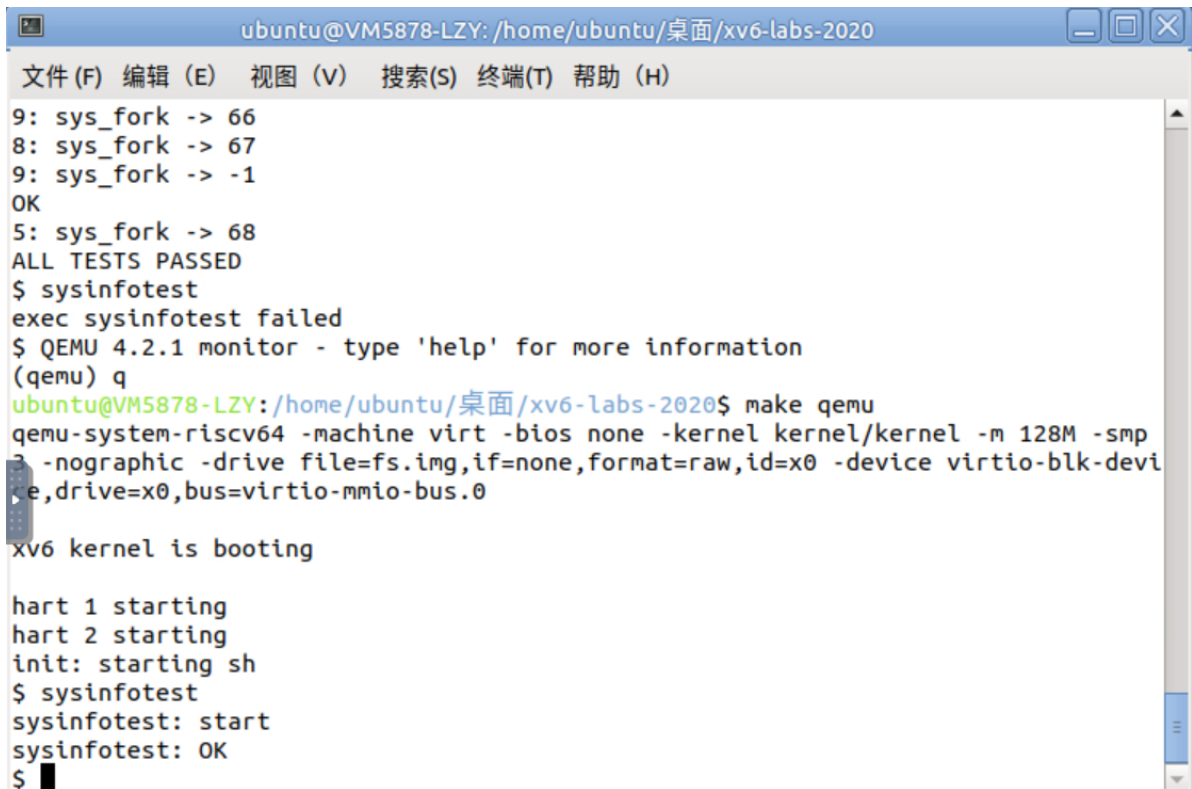
```

struct sysinfo;
int sysinfo(struct sysinfo *);

```

(第一次因为忘记声明折腾了半天，后面隔了很久写实验报告忘记声明结构体相关的头文件了/(ToT)/~~~以后一定做完实验马上写实验报告！)

## 运行结果



```
ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020
文件(F) 编辑(E) 视图(V) 搜索(S) 终端(T) 帮助(H)
9: sys_fork -> 66
8: sys_fork -> 67
9: sys_fork -> -1
OK
5: sys_fork -> 68
ALL TESTS PASSED
$ sysinfotest
exec sysinfotest failed
$ QEMU 4.2.1 monitor - type 'help' for more information
(qemu) q
ubuntu@VM5878-LZY: /home/ubuntu/桌面/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
sysinfotest: OK
$ █
```

## 实验感受

实验二主要难度在于提示更少了，以及要自己理解编写的代码更多了。需要仔细阅读第四章，在 kernel/sysfile.c/sys\_fstat 和 kernel/file.c/filestat 函数中找到方法。在实验中由于忘记声明结构体，花费了许多时间，报错解决后，剩下的步骤一步步修改，写起来比实验一顺利多了。