

- 一、实验内容
 - 1.1 trace
 - 1.1.1 实验步骤
 - 1.1.2 实验测试结果
 - 1.2. sysinfo
 - 1.2.1 实验步骤
 - 1.2.2 实验测试结果
- 二、问题解答
 - 2.1
 - 2.2
 - 2.3
 - 2.4

一、实验内容

1.1 trace

1.1.1 实验步骤

a.

- 首先在user/user中加入函数定义。

```
int trace(int);
```

- 在user/usys.pl中加入用户系统调用名称。

```
entry("trace");
```

- 在kernel/syscall.h中加入系统调用号。

```
#define SYS_trace 23
```

b. 在kernel/proc.h中的proc结构体中添加uint64 mask, 并在kernel/proc.c的freeproc函数中添加

```
p->mask = 0;
```

防止对下次创建的进程造成影响。 c. 在kernel/sysproc.c中添加sys_trace实现, 首先根据argint接受传入的第一个参数, 若无参数则返回-1, 否则将当前进程的mask赋值为第一个参数。

```
uint64 sys_trace(void){
    int mask;
    if(argint(0, &mask) < 0){
        return -1;
    }
    struct proc* p = myproc();
    p->mask = mask;
    return 0;
}
```

d.

- 在kernel/syscall.c中的syscalls中添加"[SYS_trace] sys_trace", 并添加syscall_names字符串数组方便后续打印

```
static char* syscall_names[] = {"", "fork", "exit", "wait", "pipe",
                                "read", "kill", "exec", "fstat",
                                "chdir", "dup", "getpid", "sbrk",
                                "sleep", "uptime", "open", "write",
                                "mknod", "unlink", "link", "mkdir",
                                "close", "rename", "trace"};
```

- 在syscall中调用系统调用前先利用argint获取第一个参数的int形式。
- 在调用系统调用后, 若mask>0则说明该进程调用了trace系统调用, 则检查当前系统调用相应的位是否为1, 若为1则进行打印。

```
int arg0;
argint(0, &arg0);
p->trapframe->a0 = syscalls[num]();
if (p->mask > 0 && (p->mask & 1 << num)){
    printf("%d: sys_%s(%d) -> %d\n", p->pid, syscall_names[num], arg0, p->trapframe->a0);
}
```

e. 在fork系统调用的实现中添加如下语句以便子进程继承父进程的mask。

```
safestrcpy((char*)&np->mask, (char*)&p->mask, sizeof(p->mask));
```

1.1.2 实验测试结果

1. trace 32 grep hello README

```
$ trace 32 grep hello README
3: sys_read(3) -> 1023
3: sys_read(3) -> 966
3: sys_read(3) -> 70
3: sys_read(3) -> 0
```

2. trace 2147483647 grep hello README

```
$ trace 2147483647 grep hello README
4: sys_trace(2147483647) -> 0
4: sys_exec(12240) -> 3
4: sys_open(12240) -> 3
4: sys_read(3) -> 1023
4: sys_read(3) -> 966
4: sys_read(3) -> 70
4: sys_read(3) -> 0
4: sys_close(3) -> 0
```

3. grep hello README

```
$ grep hello README
$
```

4. trace 2 usertests forkforkfork

```
10: sys_fork(-1) -> 64
9: sys_fork(-1) -> 65
11: sys_fork(-1) -> 66
9: sys_fork(-1) -> 67
10: sys_fork(-1) -> 68
9: sys_fork(-1) -> -1
10: sys_fork(-1) -> -1
11: sys_fork(-1) -> -1
OK
6: sys_fork(0) -> 69
ALL TESTS PASSED
```

5. ./grade-lab-syscall

```
piacesy@piacesy:~/Code/xv6-oslab24$ ./grade-lab-syscall
make: 'kernel/kernel' is up to date.
== Test trace 32 grep == trace 32 grep: OK (0.8s)
== Test trace all grep == trace all grep: OK (1.1s)
== Test trace nothing == trace nothing: OK (1.0s)
== Test trace children == trace children: OK (9.7s)
== Test sysinfotest == sysinfotest: OK (1.6s)
Score: 34/34
```

1.2. sysinfo

1.2.1 实验步骤

a.

- 首先在user/user中加入函数定义。

```
int sysinfo(struct sysinfo*);
```

- 在user/usys.pl中加入用户系统调用名称。

```
entry("sysinfo");
```

- 在kernel/syscall.h中加入系统调用号。

```
#define SYS_sysinfo 24
```

b. 在kalloc.c中添加kcount_freemem函数用于计算剩余的内存字节数。

- 首先获取锁，确保遍历空闲内存列表时不会发生并发修改。
- 而后遍历所有的空闲内存块并进行计数。
- 释放锁后返回空闲内存块乘以每个页面的大小即为剩余的内存字节数。

```
uint64 kcount_freemem(void){
    struct run* r;
    uint64 count = 0;

    acquire(&kmem.lock);
    r = kmem.freelist;
    while(r){
        count += 1;
        r = r->next;
    }
    release(&kmem.lock);
    return count * PGSIZE;
}
```

c. 在kernel/proc.c中添加count_proc函数计算当前状态为UNUSED的进程数。

- 遍历进程表中的所有进程。
- 获取遍历到的进程的锁，确保不发生并发修改
- 检查其状态，若为UNUSED，则将count加1。
- 检查完毕后释放锁。
- 返回UNUSED状态的进程数量。

```
uint64 count_proc(void){
    struct proc* p;
    uint64 count = 0;
    for(int i=0; i< NPROC; i++){
        p = &proc[i];
        acquire(&p->lock);
        if(p->state == UNUSED){
            count += 1;
        }
        release(&p->lock);
    }
    return count;
}
```

d. 在kernel/defs.h中添加kcount_freemem和count_proc的声明。 e. 在kernel/sysproc.c中先添加sysinfo.h头文件，而后添加sys_sysinfo实现。

- 首先声明一个sysinfo结构体，而后通过kcount_freemem和count_proc进行赋值。
- 利用argaddr获取用户态中需要赋值的sysinfo结构体的地址。
- 而后利用copyout函数将内核态中的sysinfo结构体复制到内核态中的地址中。

```
uint64 sys_sysinfo(void){
    struct sysinfo info;
    uint64 addr;
    info.freemem = kcount_freemem();
    info.nproc = count_proc();
    if(argaddr(0, &addr) < 0){
        return -1;
    }

    if(copyout(myproc()->pagetable, addr, (char*)&info, sizeof(info)) < 0){
        return -1;
    }else{
        return 0;
    }
}
```

e. 在kernel/syscall.c的syscalls和syscall_names数组中分别添加"[SYS_sysinfo] sys_sysinfo"和"sysinfo"以便调用和trace打印。

1.2.2 实验测试结果

- sysinfotest

```
$ sysinfotest
sysinfotest: start
sysinfotest: OK
```

二、问题解答

2.1

- 首先在trace.c的main函数中对mask参数进行检查，若无问题则调用trace函数。
- 调用trace函数后陷入内核态，进入syscall函数，而后调用sys_trace函数。
- sys_trace函数中将对当前进程的mask进程设置。
- 之后回到用户态，使用exec函数执行mask参数后的命令。
- 之后每次调用系统调用时，在syscall函数中都会对该系统调用在mask中对应的位进行检查，若为1则进行打印。

2.2

syscall.h文件中定义了每个系统调用对应的编号。在usys.pl中的以下语句中

```
print " li a7, SYS_${name}\n";
```

syscall.h用于将对应的系统调用编号加载到寄存器a7中。在syscall.c中，syscall.h中的系统调用编号用于设置syscalls数组中对应索引的系统调用函数，便于后续根据系统调用编号进行调用。

2.3

在syscall.c中的syscalls数组中，根据syscall.h中的系统调用编号设置了对应索引的系统调用函数。syscall函数中根据

```
p->trapframe->a7
```

获取系统调用编号，而后在syscalls数组中获取对应的系统调用函数，再进行调用。而后syscall将系统调用的返回值存放在

```
p->trapframe->a0
```

2.4

- argint函数用于从第n个参数获取一个int值。
- argaddr函数用于从第n个参数获取一个uint64值作为地址。
- argstr函数利用fetchstr函数从第n个参数中获取一个字符串。
 - 其中fetchstr用于从当前进程的地址空间中将指定地址的字符串复制到内核中。
 - 类似的fetchaddr用于从当前进程的地址空间中将指定地址的uint64值复制到内核中。
- syscall.c中的argraw函数用于从当前进程的trapframe中提取系统调用的第n个参数。