

实验报告

Lab 2

System calls

姓名：张明瑞

班级：2020 级信息安全

学号：20307130247

一、 实验：

Part A: 创建一个新的系统调用 trace

(一) 实验步骤

1.在 Makefile 的 UPROGS 变量中添加\$U/_trace

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_trace\
```

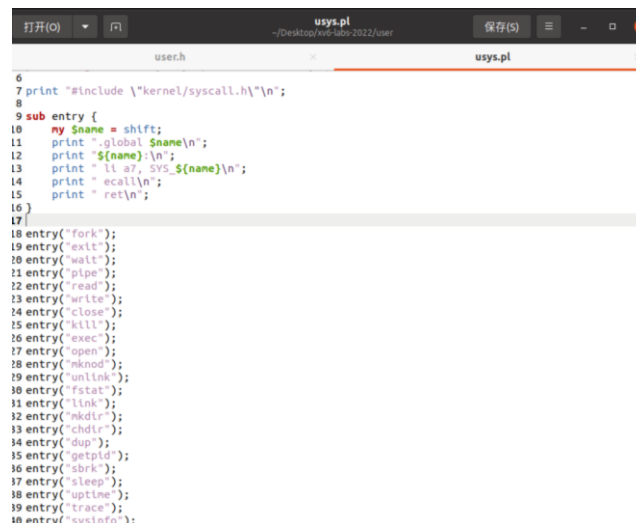
2.在 user/user.h 中添加 syscall 函数 trace 的声明

```
int sleep(int);
int uptime(void);
int trace(int);
```

3.在 kernel/syscall.h 中加入 trace 的系统调用号

```
20 #define SYS_link      19
21 #define SYS_mkdir     20
22 #define SYS_close     21
23 #define SYS_trace     22
```

4.在 user/usys.pl 中添加入口“trace”



```
6
7 print "#include \"kernel/syscall.h\"\\n";
8
9 sub entry {
10     my $name = shift;
11     print ".global $name\\n";
12     print "$($name):\\n";
13     print "    li a7, SYS_$($name)\\n";
14     print "    ecall\\n";
15     print "    ret\\n";
16 }
17
18 entry("fork");
19 entry("exit");
20 entry("wait");
21 entry("pipe");
22 entry("read");
23 entry("write");
24 entry("close");
25 entry("kill");
26 entry("exec");
27 entry("open");
28 entry("mknod");
29 entry("unlink");
30 entry("fstot");
31 entry("link");
32 entry("mkdir");
33 entry("chdir");
34 entry("dup");
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("trace");
40 entry("sysinfo");
```

添加入口“trace”后, 执行 make 指令时 Makefile 会引用该文件以生成 user/usys.S 文件。usys.S 文件是用户态系统调用的接口, 由 RISC-V 指令编写而成。

```
108 .global trace
109 trace:
110     li a7, SYS_trace
111     ecall
112     ret
```

5.在 kernel/sysproc.c 中添加 sys_trace 函数，使用 proc 结构中的变量 mask 存储自身参数，以实现新的系统调用

```
96 uint64
97 sys_trace(void)
98 {
99     int traceMask;
100
101     argint(0, &traceMask);
102     myproc()->mask = traceMask;
103     return 0;
104 }
```

(sysproc.c 中的 sys_trace 函数)

使用 argint()保存函数 sys_trace 的参数，然后将该参数值赋给结构 myproc()的变量 mask。

```
// these are private to the process, so p->lock need not be held.
uint64 kstack;           // Virtual address of kernel stack
uint64 sz;                // Size of process memory (bytes)
int mask;                 // The syscall number to be traced
```

(在 proc.h 中新建的 int 型变量 mask)

6.在 kernel/syscall.c 的函数指针数组中增加 trace()函数的对应内容

```
101 extern uint64 sys_link(void);
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_trace(void);
```

```
128 [SYS_link]      sys_link,
129 [SYS_mkdir]     sys_mkdir,
130 [SYS_close]     sys_close,
131 [SYS_trace]     sys_trace,
```

7.修改 kernel/proc.c 中的 fork()函数，让子进程继承父进程的要跟踪的系统调用号 mask

```
280 int
281 fork(void)
282 {
283     int i, pid;
284     struct proc *np;
285     struct proc *p = myproc();
286
287     // Allocate process.
288     if((np = allocproc()) == 0){
289         return -1;
290     }
291
292     // Copy user memory from parent to child.
293     if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
294         freeproc(np);
295         release(&np->lock);
296         return -1;
297     }
298     np->sz = p->sz;
299
300     // copy mask from parent process to child process.
301     np->mask = p->mask;
```

只需要一行代码：np->mask = p->mask。

其中 np 和 p 都是 fork 原有的、为了复制父进程而创建的变量，分别指代新进程、当前进程。

8.在 kernel/syscall.c 中创建一个包含各系统调用函数的名称的字符数组，修改函数 syscall()以输出追踪过程

```

135 void
136 syscall(void)
137 {
138     int num;
139     char syscall_names[24][10] = {"", "fork", "exit", "wait", "pipe", "read", "kill", "exec",
    "fstat", "chdir", "dup", "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod",
    "unlink", "link", "mkdir", "close", "trace", "sysinfo");
140     struct proc *p = myproc();
141
142     num = p->trapframe->a7;
143     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
144         // Use num to lookup the system call function for num, call it,
145         // and store its return value in p->trapframe->a0
146         p->trapframe->a0 = syscalls[num]();
147         int i = p->mask;
148         for (int j = 0; j < num; j++)
149             i = i / 2;
150         if (i % 2 == 1)
151         {
152             printf("%d: syscall %s -> %d\n", p->pid, syscall_names[num], p->trapframe->a0);
153         }
154     } else {
155         printf("%d %s: unknown sys call %d\n",
156             p->pid, p->name, num);
157         p->trapframe->a0 = -1;
158     }
159 }

```

创建一个整型变量 *i*，初始化为 *mask* 的值，*i* 的二进制位数为 32，当其对应位为 1 时即代表当前进程为想要追踪的进程；于是需要判断将 *i* 右移 *num* 位后其最右位是 0 还是 1，若是 0 则不会输出语句，若是 1 则输出结果。

(二) 运行结果

分别输入四条指令：

trace 32 grep hello README

trace 2147483647 grep hello README

grep hello README

trace 2 usertests forkforkfork

```

xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
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 -> 961
4: syscall read -> 321
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
10: syscall fork -> 12
11: syscall fork -> 13
10: syscall fork -> 14
11: syscall fork -> 15
10: syscall fork -> 16
11: syscall fork -> 17
10: syscall fork -> 18
11: syscall fork -> 19
9: syscall fork -> 20
11: syscall fork -> 21
10: syscall fork -> 22
11: syscall fork -> 23
9: syscall fork -> 24
11: syscall fork -> 25
10: syscall fork -> 26
11: syscall fork -> 27
10: syscall fork -> 28
10: syscall fork -> 29
12: syscall fork -> 30
9: syscall fork -> 31
12: syscall fork -> 32
9: syscall fork -> 33
12: syscall fork -> 35
9: syscall fork -> 36
12: syscall fork -> 37
10: syscall fork -> 34
9: syscall fork -> 38
12: syscall fork -> 39
11: syscall fork -> 40
12: syscall fork -> 41
24: syscall fork -> 42
29: syscall fork -> 43
12: syscall fork -> 44
25: syscall fork -> 45
12: syscall fork -> 46
9: syscall fork -> 47
12: syscall fork -> 48
29: syscall fork -> 49
12: syscall fork -> 50
18: syscall fork -> 51
29: syscall fork -> 52
18: syscall fork -> 53
18: syscall fork -> 54
13: syscall fork -> 55
18: syscall fork -> 56
13: syscall fork -> 57
36: syscall fork -> 58
13: syscall fork -> 59
9: syscall fork -> 60
13: syscall fork -> 61
9: syscall fork -> 62
13: syscall fork -> 63
39: syscall fork -> 64
9: syscall fork -> 65
13: syscall fork -> 66
45: syscall fork -> 67
9: syscall fork -> 68
10: syscall fork -> -1
9: syscall fork -> -1
13: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
$

```

其中 32 对应的二进制为 0000 0000 0000 0000 0000 0000 0010 0000，则 trace 会追踪对应系统调用号为 5 的函数，即 read。

Part B: 创建一个新的系统调用 sysinfo

(一) 实验步骤：

1. 在 Makefile 的 UPROGS 中添加“\$U/_sysinfotest\”

```

$U/_trace\
$U/_sysinfotest\

```

2.在 user/user.h 中添加结构 sysinfo 和函数 sysinfo 的声明

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

3.在 kernel/syscall.h 中添加 sysinfo 的系统调用号

```
22 #define SYS_close 21  
23 #define SYS_trace 22  
24 #define SYS_sysinfo 23
```

4.在 user/usys.pl 中添加入口“sysinfo”,

```
37 entry("sleep");  
38 entry("uptime");  
39 entry("trace");  
40 entry("sysinfo");
```

与 Part A 相似, 在 make 指令执行时 Makefile 会在 usys.S 中生成 RISC-V 指令编写的 sysinfo 内容。

```
113 .global sysinfo  
114 sysinfo:  
115 li a7, SYS_sysinfo  
116 ecall  
117 ret
```

5.在 kernel/syscall.c 的函数指针数组中添加 sysinfo 函数的相关内容

```
101 extern uint64 sys_link(void);  
102 extern uint64 sys_mkdir(void);  
103 extern uint64 sys_close(void);  
104 extern uint64 sys_trace(void);  
105 extern uint64 sys_sysinfo(void);  
106  
107 // An array mapping syscall numbers from syscall.h  
108 // to the function that handles the system call.  
109 static uint64 (*syscalls[])(void) = {  
110 [SYS_fork] sys_fork,  
111 [SYS_exit] sys_exit,  
112 [SYS_wait] sys_wait,  
113 [SYS_pipe] sys_pipe,  
114 [SYS_read] sys_read,  
115 [SYS_kill] sys_kill,  
116 [SYS_exec] sys_exec,  
117 [SYS_fstat] sys_fstat,  
118 [SYS_chdir] sys_chdir,  
119 [SYS_dup] sys_dup,  
120 [SYS_getpid] sys_getpid,  
121 [SYS_sbrk] sys_sbrk,  
122 [SYS_sleep] sys_sleep,  
123 [SYS_uptime] sys_uptime,  
124 [SYS_open] sys_open,  
125 [SYS_write] sys_write,  
126 [SYS_mknod] sys_mknod,  
127 [SYS_unlink] sys_unlink,  
128 [SYS_link] sys_link,  
129 [SYS_mkdir] sys_mkdir,  
130 [SYS_close] sys_close,  
131 [SYS_trace] sys_trace,  
132 [SYS_sysinfo] sys_sysinfo,  
133 };
```

6.在 kernel/sysinfo.h 中添加结构 studentNum 来保存自己的学号

```
6 struct studentNum{  
7 char stuNum[50];  
8};
```

7.在 kernel/kalloc.c 中添加函数 getFreemem 以获取空闲内存

```

85 int
86 getFreemem(void)
87 {
88     struct run *r;
89     int freeMemory = 0, num = 0;
90     acquire(&kmem.lock);
91     r = kmem.freelist;
92     while(r != NULL)
93     {
94         num++;
95         r = r->next;
96     }
97     release(&kmem.lock);
98     freeMemory = PGSIZE * num;
99     return freeMemory;
100 }

```

通过阅读 kalloc.c 中 kalloc 和 kfree 两个函数可以看出 kmem.freelist 是一个空闲内存块的链表，通过读取其长度，再乘以每一内存块的长度 PGSIZE，即可得到剩余内存空间的大小。

8.在 kernel/proc.c 中添加函数 getNproc()以获取当前在运行的进程数

```

689 int
690 getNproc(void)
691 {
692     int num = 0;
693     struct proc *p;
694     for(p = proc; p < &proc[NPROC]; p++){
695         acquire(&p->lock);
696         if(p->state != UNUSED)
697             num++;
698         release(&p->lock);
699     }
700     return num;
701 }

```

通过阅读前面的源码可以发现 proc 存储的是当前在运行的进程，于是首先让我们申请的指针 p 指向 proc 的首地址，然后获取锁，检测该进程是否为 UNUSED，若结果为否，则计数加 1 并释放锁，若结果为是则直接释放锁。最后返回计数 num。

9.在 kernel/sysproc.c 中添加 sys_sysinfo 函数

```

8 #include "sysinfo.h"

106 uint64
107 sys_sysinfo(void)
108 {
109     int pointer;
110     struct sysinfo sysInfo;
111     struct proc *p = myproc();
112     struct studentNum stNm;
113     char ID[] = "My student number is 20307130247\n";
114     for(int i = 0; i < sizeof(stNm.stuNum) && i < sizeof(ID); i++)
115         stNm.stuNum[i] = ID[i];
116     argint(0, &pointer);
117     sysInfo.freemem = getFreemem();
118     sysInfo.nproc = getNproc();
119     printf(stNm.stuNum);
120     if(copyout(p->pagetable, pointer, (char *)&sysInfo, sizeof(sysInfo)) < 0)
121         return -1;
122     return 0;
123 }

```

创建 int 型变量 pointer，使用 argint()读取 sysinfo 的参数值并保存到 pointer 中。

创建一个 studentNum 类型的结构，这个结构是我在 sysinfo.h 中新建的，其内容为一个字符数组，在后文中会借之打印学号相关内容。使用 for 循环将学号信息赋给结构中的字符数组。调用函数 getFreemem 和 getNproc 获取空闲内存字节数和当前进程数，并将之保存至 sysinfo 结构的内含变量中。

(二) 运行结果：

使用指令 sysinfotest，运行测试程序，其输出截图如下，可以发现代码顺利运行，得到“sysinfotest: OK”正确反馈。


```
merry@ubuntu: ~/Desktop/xv6-labs-2022
ballocc: first 812 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sysinfotest
sysinfotest: start
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
My student number is 20307130247
sysinfotest: OK
$
```

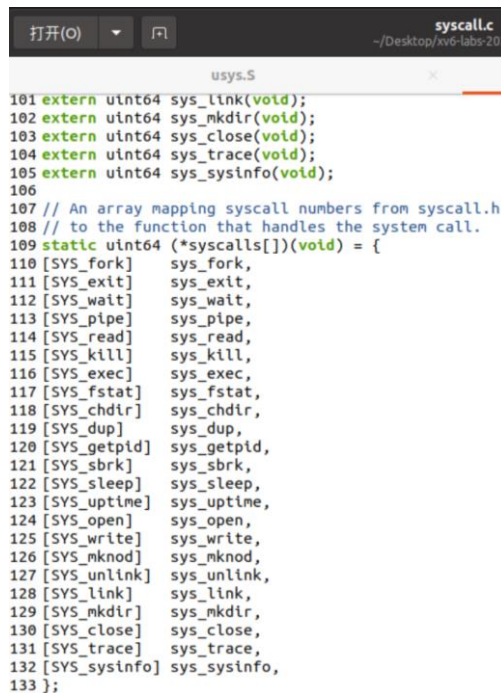
二、 回答问题

1.简述 trace 全流程:

我们输入 trace 指令后执行的是用户态下的函数 trace.c, 在 trace.c 中有一个调用 trace()函数的语句。我们已在 user.h 中声明了该函数, 但现在需要具体调用其内容而非声明语句, 这时先前在 usys.pl 中设置的 entry("trace")就会起到作用, 在执行 make qemu 指令时 Makefile 会执行 usys.pl 文件, 然后根据其指令在 usys.S 中写入如下语句:

```
108 .global trace
109 trace:
110 li a7, SYS_trace
111 ecall
112 ret
```

这个汇编指令表示当其他代码调用 trace 函数时, 将系统调用号 SYS_trace 通过 load imm 指令存入 a7 寄存器, 同时使用 ecall 指令进入内核态。usys.S 文件即是用户态与内核态的接口。在内核态中, 我们在 syscall.h 中定义了 SYS_trace 的系统调用号, 并且在 syscall.c 中有一个函数指针数组, 经过我们添加后, SYS_trace 号对应的就是内核中的 sys_trace 函数。



```
101 extern uint64 sys_link(void);
102 extern uint64 sys_mkdir(void);
103 extern uint64 sys_close(void);
104 extern uint64 sys_trace(void);
105 extern uint64 sys_sysinfo(void);
106
107 // An array mapping syscall numbers from syscall.h
108 // to the function that handles the system call.
109 static uint64 (*syscalls[])(void) = {
110 [SYS_fork] sys_fork,
111 [SYS_exit] sys_exit,
112 [SYS_wait] sys_wait,
113 [SYS_pipe] sys_pipe,
114 [SYS_read] sys_read,
115 [SYS_kill] sys_kill,
116 [SYS_exec] sys_exec,
117 [SYS_fstat] sys_fstat,
118 [SYS_chdir] sys_chdir,
119 [SYS_dup] sys_dup,
120 [SYS_getpid] sys_getpid,
121 [SYS_sbrk] sys_sbrk,
122 [SYS_sleep] sys_sleep,
123 [SYS_uptime] sys_uptime,
124 [SYS_open] sys_open,
125 [SYS_write] sys_write,
126 [SYS_mknod] sys_mknod,
127 [SYS_unlink] sys_unlink,
128 [SYS_lnk] sys_lnk,
129 [SYS_mkdir] sys_mkdir,
130 [SYS_close] sys_close,
131 [SYS_trace] sys_trace,
132 [SYS_sysinfo] sys_sysinfo,
133 };
```

在 syscall.c 的 syscall 函数中，源码通过“p->trapframe->a7”获取当前的系统调用号，然后使用“p->trapframe->a0 = syscalls[num];”语句，通过该系统调用号查找对应函数并调用该 syscall 函数。

而我们就在此处查找成功后检查返回的函数是否为需要追踪的函数，创建一个整型变量 i，初始化为 mask 的值，i 的二进制位数为 32，当其对应位为 1 时即代表当前进程为想要追踪的进程；于是需要判断将 i 右移 num 位后其最右位是 0 还是 1，若是 0 则不会输出语句，若是 1 则输出结果。

另外在代码使用 fork() 申请子进程的时候子进程也要也要继承父进程的 mask。

2.kernel/syscall.h 是干什么的，如何起作用的？

syscall.h 是系统调用函数的头文件，存储、定义了各系统调用函数的系统调用号。正如问题 1 中所回答，syscall.c 中的 syscall() 函数使用该系统调用号来查找、引用对应函数。

3.命令“trace 32 grep hello README”中的 trace 字段是用户态下的还是实现的系统调用函数 trace？

该指令中的第一个词 trace 字段是用户态下的，系统调用函数 trace 会在内核态下被调用，这之前需要在 usys.S 中由用户态转到内核态。

三、 实验中碰到的问题

1.阅读源码

对于一些函数，如果不去调用它的地方查看其使用方法，我觉得自己很难成功使用它实现相应功能，事实上很多地方我是依照其他函数的调用它时的语句，先模仿着写，再尝试理解的。比如这句话：

```
for(p = proc; p < &proc[NPROC]; p++){
```

我一开始是直接 from 其他函数搬到检测进程数的函数 getNproc() 里面来的，后面才发现它的判断条件并不是数的大小而是地址大小，一个数组的地址是连续的，当 p 超出数组末时即到达

proc[NPROC]。

还有这句话：

```
if(copyout(p->pagetable, pointer, (char *)&sysInfo, sizeof(sysInfo)) < 0)
```

我也是依照其他函数的使用方式直接复制过来，再修改相应参数，sysInfo 本来是个结构，但这里将其地址强制转换为一个字符指针，大概就是一个指向 sysInfo 的字符型指针，其作用相当于把 sysInfo 作为一个字符变量使用。

2.找到应当修改的地方，然后 debug。

这次实验我自己写的总代码量不大，但需要阅读的量偏大、范围有些广，总体给人一种零散感，时常不知道该到哪里去修改一两行语句，万幸报错时会指出哪里有问题，这足以找到大部分 bug 的来源了。

3.C 语言常见函数的使用

在复制字符串的时候我本来想使用 string.h 和 strcpy(), 但我想起 lab0 时，我引用 C 语言原本的库经常跟 xv6 的一些重写覆盖过的函数冲突，于是我直接采取 for 循环数组挨个赋值的形式来完成字符串复制了。

四、 实验感想

我阅读源码的能力还是欠佳，相信在接下来的实验中多加锻炼后，应该能有所提升，至少学会怎么调用一些函数的功能。另外在此次实验中我发现备注真的非常重要，xv6 源码的备注帮助我理解了很多代码段。