



# Lab2

## 实验报告

姓名 李晓畅

学号 20307130261

班级 计算机科学技术

# 实现思路

## PART A

It should take one argument, an integer "mask", whose bits specify which system calls to trace.

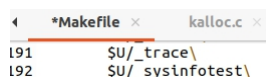
For example, to trace the `fork` system call, a program calls `trace(1 << SYS_fork)`, where `SYS_fork` is a syscall number from `kernel/syscall.h`. You have to modify the xv6 kernel to print out a line when each system call is about to return, if the system call's number is set in the mask. The line should contain the process id, the name of the system call and the return value; you don't need to print the system call arguments.

The trace system call should enable tracing for the process that calls it and any children that it subsequently forks, but should not affect other processes.

按照提示进行：

- Add `SU/_trace` to UPROGS in Makefile

因为我们需要编译这些代码，所以我们把它们添加到 **Makefile** 中



```
*Makefile x kalloc.c x
L91     SU/_trace\
L92     SU/_sysinfotest\
```

- Run `make qemu` and you will see that the compiler cannot compile `user/trace.c`, because the user-space stubs for the system call don't exist yet: add a prototype for the system call to `user/user.h`, a stub to `user/usys.pl`, and a syscall number to `kernel/syscall.h`. The Makefile invokes the perl script `user/usys.pl`, which produces `user/usys.S`, the actual system call stubs, which use the RISC-V `ecall` instruction to transition to the kernel. Once you fix the compilation issues, run `trace 32 grep hello README`; it will fail because you haven't implemented the system call in the kernel yet.

1. 需要在 **user.h** 中添加函数声明；  
`int trace(int);`  
`int sysinfo(struct sysinfo*);`

2. 在 **usys.pl** 中添加，  
`39 entry("trace");`  
`40 entry("sysinfo");` 这是一个 Perl 文件，在 make 时生成 **usys.S** 文件，这个文件作用在于系统调用号通过 **li(load imm)** 存入 a7 寄存器，之后使用 **ecall** 进入内核态，最后返回；

3. 在 **syscall.h** 中添加系统调用号；  
`23 #define SYS_trace 22`  
`24 #define SYS_sysinfo 23`

- Add a `sys_trace()` function in `kernel/sysproc.c` that implements the new system call by remembering its argument in a new variable in the proc structure (see `kernel/proc.h`). The functions to retrieve system call arguments from user space are in `kernel/syscall.c`, and you can see examples of their use in `kernel/sysproc.c`.

下面就可以实现函数的接口了，具体来说：

```
uint64
sys_trace(void){
    int n;
    argint(0,&n); // a0->n => arg[1]->n
    myproc()->mask = n;
    return 0;
}
```

按照其他函数的方式调用 *argint* 读取参数；

*argint* 利用用户空间的 *%esp* 寄存器定位第 *n* 个参数：*%esp* 指向系统调用结束后的返回地址。参数就恰好在 *%esp* 之上 (*%esp+4*)。因此第 *n* 个参数就在 *%esp+4+4\*n*；

换句话说这里读取了第一个参数，也就是 *mask*；

根据题目描述，应该只追踪进程和它的子进程的对应系统调用，所以应该更新对应进程的 *mask* 信息；

因而对应的结构体也需要修改：

```
// these are private to the process, so p->lock need not be held.
int mask;
```

*Mask* 应该是私有信息，所以之后也不需要获得 *lock*

- Modify *fork()* (see *kernel/proc.c*) to copy the trace mask from the parent to the child process.

需要对 *fork* 函数进行一些修改 *np->mask=p->mask*；，使得子进程可以继承父进程的 *mask*

- Modify the *syscall()* function in *kernel/syscall.c* to print the trace output. You will need to add an array of *syscall* names to index into.

```
135 static char* sys_names[] = {
136     [SYS_fork]    "fork",
137     [SYS_exit]    "exit",
138     [SYS_wait]    "wait",
139     [SYS_pipe]    "pipe",
140     [SYS_read]    "read",
141     [SYS_kill]    "kill",
142     [SYS_exec]    "exec",
143     [SYS_fstat]   "fstat",
144     [SYS_chdir]   "chdir",
145     [SYS_dup]     "dup",
146     [SYS_getpid]  "getpid",
147     [SYS_sbrk]    "sbrk",
148     [SYS_sleep]   "sleep",
149     [SYS_uptime]  "uptime",
150     [SYS_open]    "open",
151     [SYS_write]   "write",
152     [SYS_mknod]   "mknod",
153     [SYS_unlink]  "unlink",
154     [SYS_link]    "link",
155     [SYS_mkdir]   "mkdir",
156     [SYS_close]   "close",
157     [SYS_trace]   "trace",
158     [SYS_sysinfo] "sysinfo",
159 };
160 //map sys call id -> sys call name
```

因为需要打印调用的名称，所以需要一张调用号与名字间的映射表。参照上方不难写出这样的映射。

```

void
syscall(void)
{
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // Use num to lookup the system call function for num, call it,
        // and store its return value in p->trapframe->a0
        p->trapframe->a0 = syscalls[num]();

        if((p->mask)&(1<<num)){
            printf("%d: syscall %s -> %d\n", p->pid, sys_names[num], p->trapframe->a0);
        }
    } else {
        printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
        p->trapframe->a0 = -1;
    }
}

```

如果系统调用成功，就根据 mask 检查是否需要追踪对应的系统调用，若需要则做对应输出。

测试结果截图：

```

$ 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
9: syscall fork -> 13
10: syscall fork -> 14
11: syscall fork -> 15
11: syscall fork -> 16
9: syscall fork -> 17
10: syscall fork -> 18
10: syscall fork -> 19
11: syscall fork -> 20
12: syscall fork -> 21
13: syscall fork -> 22
12: syscall fork -> 23
9: syscall fork -> 24
24: syscall fork -> 25
9: syscall fork -> 26
14: syscall fork -> 27
17: syscall fork -> 28
14: syscall fork -> 29
9: syscall fork -> 30
25: syscall fork -> 31
31: syscall fork -> 32
26: syscall fork -> 33
29: syscall fork -> 34
9: syscall fork -> 35
27: syscall fork -> 36
9: syscall fork -> 37
10: syscall fork -> 38
9: syscall fork -> 39
26: syscall fork -> 40
9: syscall fork -> 41
10: syscall fork -> 42
9: syscall fork -> 43
10: syscall fork -> 44
11: syscall fork -> 45
9: syscall fork -> 46
10: syscall fork -> 47
9: syscall fork -> 48

```

```

11: syscall fork -> 50
27: syscall fork -> 51
20: syscall fork -> 52
9: syscall fork -> 53
11: syscall fork -> 54
13: syscall fork -> 55
11: syscall fork -> 56
13: syscall fork -> 57
28: syscall fork -> 58
9: syscall fork -> 59
38: syscall fork -> 60
42: syscall fork -> 61
42: syscall fork -> 62
42: syscall fork -> 63
42: syscall fork -> 64
19: syscall fork -> 65
13: syscall fork -> 66
50: syscall fork -> 67
19: syscall fork -> 68
11: syscall fork -> -1
50: syscall fork -> -1
54: syscall fork -> -1
48: syscall fork -> -1
52: syscall fork -> -1
63: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED

```

运行成功

## PART B

The system call takes one argument: a pointer to a struct **sysinfo** (see *kernel/sysinfo.h*).

The kernel should fill out the fields of this struct: the freemem field should be set to the number of bytes of free memory, and the nproc field should be set to the number of processes whose state is not **UNUSED**.

In addition, you need to add a structure containing a string type field in the *kernel/sysinfo.h*. This field can be named arbitrarily to indicate your student number. You need also **print your student ID** when calling the sysinfo function.

We provide a test program sysinfotest. You pass this assignment if it prints "**sysinfotest: OK**".

按照提示进行：

- Add `$U/_sysinfotest` to UPROGS in Makefile.
- Run **make qemu**; *user/sysinfotest.c* will fail to compile. Add the system call sysinfo, following the same steps as in the previous assignment. To declare the prototype for sysinfo() in *user/user.h* you need predeclare the existence of struct sysinfo:

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

Once you fix the compilation issues, run **sysinfotest**; it will fail because you haven't implemented the system

- call in the kernel yet.

类似于之前修改文件，修改可参 PART A 中图片；

因为包含新的结构体，所以在 *user.h* 中除函数声明 `int sysinfo(struct sysinfo*);`

外还要添加结构体声明 `struct sysinfo;`；

另外，*sysinfo.h* 是这样的：

```
1 struct sysinfo {  
2   uint64 freemem; // amount of free memory (bytes)  
3   uint64 nproc;  // number of process  
4   char sid[12];  
5 };
```

- sysinfo needs to copy a struct sysinfo back to user space;  
• see sys\_fstat() (*kernel/sysfile.c*) and filestat() (*kernel/file.c*) for examples of how to do that using copyout().

提示中指出我们实现 syscall 中的代码使用 copyout 实现：



```

105 uint64
106 sys_sysinfo(void){
107     struct sysinfo info;
108     info.nproc=get_nproc();
109     info.freemem=get_freemem();
110     info.sid[0]='2';info.sid[1]='0';info.sid[2]='3';
111     info.sid[3]='0';info.sid[4]='7';info.sid[5]='1';
112     info.sid[6]='3';info.sid[7]='0';info.sid[8]='2';
113     info.sid[9]='6';info.sid[10]='1';info.sid[11]='\0';
114     printf("my student number is ");
115     printf(info.sid);
116     printf("\n");
117
118     struct proc* p=myproc();
119     int n;
120     argint(0,&n);
121
122     if (copyout(p->pagetable, n, (char*)&info, sizeof(info)) < 0)
123         return -1;
124     return 0;
125 }
126 }
127

```

调用时先计算进程数和空间剩余，并对应初始化一个 sysinfo 以用于拷贝；

类似可以获取参数；

根据提示参考 *copyout()*用法，见 *filestat()*：

```

97     if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)

```

```

133 // addr is a user virtual address.

```

参照即可调用 *copyout()*实现复制

- To collect the amount of free memory, add a function to *kernel/kalloc.c*.

```

84 int
85 get_freemem()
86 {
87     struct run* r;
88     acquire(&kmem.lock);
89     r=kmem.freelist;
90     release(&kmem.lock);
91
92     int freemem=0;
93     while(r){
94         freemem++;
95         r=r->next;
96     }
97     return freemem*PGSIZE;
98 }

```

先获取空闲存储页链表，遍历，计数即可得到总共的空闲页；再乘以页大小即可获得空闲内存。

- To collect the number of processes, add a function to *kernel/proc.c*.

```

588 int get_nproc(){
589     int nproc=0;
590     for(int i=0;i<NPROC;i++){
591         if(proc[i].state!=UNUSED) nproc++;
592     }
593     return nproc;
594 }

```

直接统计进程表中非 UNUSED 的部分就是对应结果了。

测试结果截图：

```
$ sysinfotest
sysinfotest: start
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
my student number is 20307130261
sysinfotest: OK
$
```

运行成功

## 问题回答

### TRACE 全流程

- 在用户态通过系统调用 `trace` 为当前进程的 `proc` 设置 `mask` 表示要求追踪的系统调用；
- 用户进行系统调用，由用户态切换为内核态，用户进程经由 `syscall()` 进行系统调用时进行检查，如果 `mask` 指示该进程需要追踪，那么在调用结束后就会打印对应信息；
- 另外，如果进程创建了子进程，那么子进程也会有相同的 `mask` 信息，这意味着在子进程进行系统调用时也会进行检查和输出；
- 系统调用结束后会回到用户态，继续执行用户代码。

### KERNEL/SYSCALL.H

正如注释 `1 // System call numbers` 所说，定义了系统调用号，这在 `syscall.c` 中我们就使用过。

每个系统调用被赋予一个系统调用号。这样，通过独一无二的号就可以关联系统调用。当用户空间的进程执行一个系统调用的时候，这个系统调用号就用来指明到底是要执行哪个系统调用；进程不会提及系统调用的名称。

系统调用号相当重要，一旦分配就不能再有任何变更，否则编译好的应用程序就会崩溃。此外，如果一个系统调用被删除，它所占用的系统调用号

也不允许被回收利用，否则，以前编译过的代码会调用这个系统调用，但事实上却调用的是另外一个系统调用。

## TRACE 字段

用户态下的。

正如在前面流程分析中表明的那样，这里的 trace 意味着发起了一个系统调用，具体实现，也就是 `sys_trace()` 将在内核态中执行。

## 问题和解决

### 调试和检查

在 PART B 中，代码可以运行，但是一直跑不出结果。

利用添加输出语句、GDB 调试方法并加以分析，定位问题是获取空余存储死循环。修改加入 `r=r->next;` 改变循环变量即可正确运行。

### 语法细节

在修改学号时发现不管怎么写都不对，不论是：`info.sid="20307130261";`

还是：`info.sid={'2','0','3','0','7','1','3','0','2','6','1','\0'};`

似乎都不行，最后采取的办法是逐个赋值。

参考 `string.h` 似乎并不支持 C++ 那样赋值？



## 实验感想

### 面向切面

PART A 中的需求实现似乎就是一种希望基于切面编程的思想。

我们希望为调用前后添加一些公共的操作，但并不希望改变调用的过程，如果可以尽可能少的改变结构，那就再好不过了。

我们在实现上直接修改了代码，实现了这一需求。但是直接修改代码或许并不一定是最好的方法，如果提示没有说明的话，准确更改代码其实应该是不容易的一件事情。

由此我们看到面向切面编程（Aspect Oriented Programming, AOP）是有实际意义的，它可以通过预编译方式和运行期动态代理的方式实现不修改源代码的情况下给程序动态统一添加功能。

### 类比

Xv6 的函数不像很多库那样有非常完备的注释，因而理解它的作用或许需要观察它在其他函数中的使用来实现。参照其他地方编写映射、调用函数，这样同样可以实现目的。换句话说，我们理解代码的方式其实是多种多样的。