Ostep 第二次实验报告

张志豪 20307130085

〇、实验结果

```
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
$
```

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sysinfotest
sysinfotest: start
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
my student number is 20307130085
sysinfotest: OK
$
```

一、实验思路

1、System call tracing

根据预期的结果，我们实验的目的是追踪一个命令的所有系统调用。

首先根据提示，在 makefile 中添加 trace，用来注册用户命令

```
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\
```

此时 make qemu 会报错，说明找不到 trace 函数，接下来需要实现 trace 函数

```
icit-function-declaration]
   17 |   if (trace(atoi(argv[1])) < 0) {
      |       ^~~~~
cc1: all warnings being treated as errors
make: *** [<builtin>: user/trace.o] Error 1
zzh@ubuntu:~/Desktop/oslab/ostep_lab2/xv6-labs-2022$
```

我们在 user.h 中加入 trace 函数的声明：

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(const char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int trace(int);
```

同样，根据提示我们在 usys.pl 中对 trace 存根

```
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("fstat");
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");
```

接下来添加 systemcall number：

trace.c

```
1 // System call numbers
2 #define SYS_fork    1
3 #define SYS_exit    2
4 #define SYS_wait    3
5 #define SYS_pipe    4
6 #define SYS_read    5
7 #define SYS_kill    6
8 #define SYS_exec    7
9 #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup    10
12 #define SYS_getpid 11
13 #define SYS_sbrk   12
14 #define SYS_sleep  13
15 #define SYS_uptime 14
16 #define SYS_open   15
17 #define SYS_write  16
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 #define SYS_trace  22
```

修改以上内容之后，makefile 会通过 usys.pl 脚本生成一个汇编文件 usys.s，用汇编调用 kernel 中的 call。此时重新 make qemu 发现已经可以启动（当然不会有输出）

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

接下来根据提示，在 sysproc 文件中加入函数 sys_trace

```
uint64
sys_trace(void)
{
  printf("hello");
}
```

我们先“象征性”地加入一个函数，接下来观察调用它的文件 syscall.c：

```
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]();
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

将 syscall[num]的返回值赋值到结构体 p 的一个元素上。
作为一个表驱动文件，我们应该在调用表中加上 trace 的函数指针：

```
129 [SYS_trace]   sys_trace
```

```
104 extern uint64 sys_trace(void);
```

现在运行 qemu 已经可以看到“hello”了。
接下来我们对 syscall 函数进行操作，将每一次的系统调用 print 出来。

```
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", "trace","sysinfo"};
```

我们将所有可能的系统调用名称存放在一个数组中，注意数组下标为 0 的时候用‘’占位。而 mask 存在的意义就是让系统知道应该把哪些调用打印出来，着就意味着 syscall 和 sys_trace 都要获取这个 mask，我们在 proc.h 的结构体中存放这个 mask。

```
// Per-process state
struct proc {
  struct spinlock lock;
  int mask;
```

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

If((1<<num)&p->mask)用来检测进程号是否命中 mask。如果命中就打印。其实现在我们就已经可以实现打印结果了，但是根据提示我们还要将父进程的 mask 给子进程，所以在 proc.c 中添加一句话即可：

```
  acquire(&np->lock);
  np->state = RUNNABLE;
  np->mask = p->mask;
  release(&np->lock);

  return pid;
```

2、Sysinfo

先看一下 sysinfo.h，发现其只有两个元素，用来表示空余内存和进程数

```
struct sysinfo {
  uint64 freemem;   // amount of free memory (bytes)
  uint64 nproc;     // number of process
};
```

与 syscall trace 相似，我们先加入 sysinfo 的函数声明和表注册

```
uint64
sys_sysinfo(void)
{
  uint64 addr;
  struct sysinfo info;
  struct proc *p = myproc();
  argaddr(0, &addr);
  if (addr < 0)
          return -1;
  info.freemem = free_mem();
  info.nproc = nproc();

  if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0)
    return -1;

  return 0;
}
```

参考 sys_file 中的代码，用 argaddr 接收参数，用 if (copyout(p->pagetable, addr, (char *)&info, sizeof(info)) < 0) return -1;把运算结果塞进 addr 指针指向的地方。

接下来我们要实现获取 freememory 和 nprocess 两个参数：

在 kalloc.c 中，空闲内存以链表的形式存在，并且单位长度为 pagesize

```
// Return the number of bytes of free memory
uint64
free_mem(void)
{
  struct run *r;
  uint64 num = 0;
  acquire(&kmem.lock);
  r = kmem.freelist;
  while (r)
  {
    num++;
    r = r->next;
  release(&kmem.lock);
  return num * PGSIZE;
}
```

我们用 num*pgsize 得到总 的空余内存

在 proc.c 中统计没有被 unuse 的进程，并打印学号：

```
// Return the number of processes whose state is not UNUSED
uint64
nproc(void)
{
  struct proc *p;
  uint64 num = 0;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->state != UNUSED)
    {
      num++;
      printf("my student number is 20307130085\n");
    }
    release(&p->lock);
  }
  return num;
}
```

二、问题回答

1、System calls Part A 部分，简述一下 trace 全流程.

```
zzh@ubuntu:~/Desktop/test/Lab2_source/xv6-labs-2022 (copy)/kernel$ grep "trace"
*
Binary file kernel matches
kernel.asm:000000008000232e <sys_trace>:
kernel.asm:sys_trace(void)
kernel.asm:    8000234a:        0007ca63               bltz    a5,8000235e <sys
_trace+0x30>
kernel.sym:000000008000232e sys_trace
syscall.c:  "mkdir", "close", "trace","sysinfo"};
syscall.c:extern uint64 sys_trace(void);
syscall.c:[SYS_trace]   sys_trace,
syscall.h:#define SYS_trace  22
Binary file syscall.o matches
sysproc.c:sys_trace(void)
Binary file sysproc.o matches
zzh@ubuntu:~/Desktop/test/Lab2_source/xv6-labs-2022 (copy)/kernel$
```

trace.c 通过 entry 入口进入内核态后，根据系统调用数字映射到相应系统调用函数名，以及相应函数位置执行函数，执行函数的过程中需要考虑到 mask 的值。

2、kernel/syscall.h 是干什么的，如何起作用的？

定义了一些内核空间的系统调用号，根据指定的这些参数和所有系统调用的汇编语言接口来确定需要用哪个系统调用。

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？

这里的 trace 字段应该时用户态的，这个 trace 调用了 user 中的功能函数 trace 然后在该功能函数当中传递信息到内核态实现系统调用函数 trace。

三、实验遇到的问题

在 proc.c 中有 freeproc 函数，在进程结束时会重置各种值为 0，我们也要将 mask 归零，否则会报错。

```
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  if(p->pagetable)
    proc_freepagetable(p->pagetable, p->sz);
  p->pagetable = 0;
  p->sz = 0;
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->chan = 0;
  p->killed = 0;
  p->xstate = 0;
  p->state = UNUSED;
  p->mask=0;
}
```

四、实验感想

本实验实现了从用户态到内核态的调用，需要修改很多文件中函数的细节，跟随 hint 走下来也挺难的，但是 trace 的过程还是很有意思的。

五、参考

1、https://blog.csdn.net/weixin_48283247/article/details/121217307?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-121217307-blog-121263520.pc_relevant_default&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-121217307-blog-121263520.pc_relevant_default&utm_relevant_index=1#t8

2、https://www.bilibili.com/video/av379678940/?vd_source=dc1aadfe42e11e42b72abe04869d1bd1