实验报告

Lab 2

姓名：田鑫

班级：信息安全

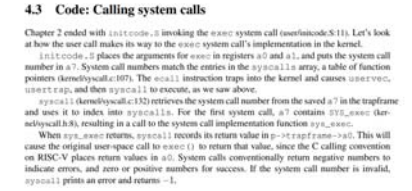
学号：19307110353

PartA: System call tracing

实验思路及测试结果：

1、首先根据 hint 在 makefile、user.h 文件当中添加响应的生命，使编译可以通过。这之后运行 **`trace 32 grep hello README`** 会失败。

2、找到应该实现的位置，查书可知，系统调用会经过 syscall.c，由 syscall 发起



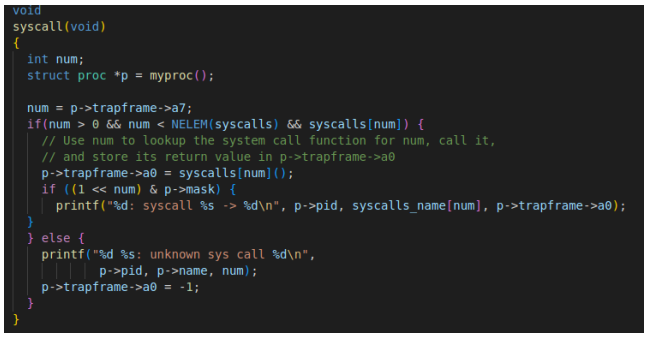
4.3 Code: Calling system calls

Chapter 2 ended with `initcode.S` invoking the `exec` system call (user/initcode.S:11). Let's look at how the user call makes its way to the `exec` system call's implementation in the kernel.

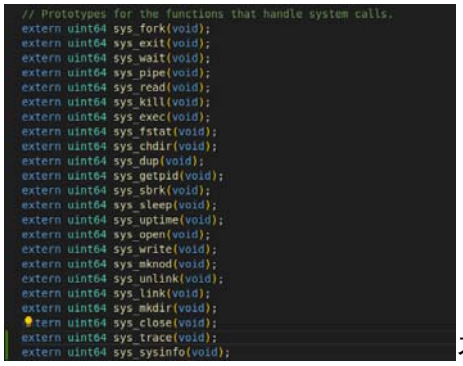
`initcode.S` places the arguments for `exec` in registers `a0` and `a1`, and puts the system call number in `a7`. System call numbers match the entries in the `syscalls` array, a table of function pointers (kernel/syscall.c:107). The `ecall` instruction traps into the kernel and causes `uservec`, `usertrap`, and then `syscall` to execute, as we saw above.

`syscall` (kernel/syscall.c:132) retrieves the system call number from the saved `a7` in the trapframe and uses it to index into `syscalls`. For the first system call, `a7` contains `SYS_exec` (kernel/syscall.h:8), resulting in a call to the system call implementation function `sys_exec`.

When `sys_exec` returns, `syscall` records its return value in `p->trapframe->a0`. This will cause the original user-space call to `exec()` to return that value, since the C calling convention on RISC-V places return values in `a0`. System calls conventionally return negative numbers to indicate errors, and zero or positive numbers for success. If the system call number is invalid, `syscall` prints an error and returns −1.



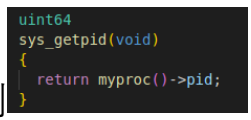
```
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

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



```
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
extern uint64 sys_fstat(void);
extern uint64 sys_chdir(void);
extern uint64 sys_dup(void);
extern uint64 sys_getpid(void);
extern uint64 sys_sbrk(void);
extern uint64 sys_sleep(void);
extern uint64 sys_uptime(void);
extern uint64 sys_open(void);
extern uint64 sys_write(void);
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_trace(void);
extern uint64 sys_sysinfo(void);
```

在 syscall.c 文件中有众多系统调用的实现，跳转到其声明就可以知道要在哪写代码了，至于格式则照搬。

3、sys_trace 的实现：依据提示应该用 mask 来标识。先看看一个实例



```
uint64
sys_getpid(void)
{
  return myproc()->pid;
}
```

，知道进程的信息应该保存在 proc 结构当中

跳转至其定义
```
// Per-process state
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;        // Process state
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  int xstate;                  // Exit status to be returned to parent's wait
  int pid;                     // Process ID

  // wait_lock must be held when using this:
  struct proc *parent;         // Parent process

  // 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)
  pagetable_t pagetable;       // User page table
  struct trapframe *trapframe; // data page for trampoline.S
  struct context context;      // swtch() here to run process
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int mask;                    // to trace (not certitude)
};
```
，应该是进程特有的。添加一个 mask 作标识。

现在解决在哪输出的问题，有了 mask 表示后，进程的哪些系统调用应该被显示已经知道，那么由谁输出？答案是 syscall，因为所有的系统调用都经过这里。

观察 syscall
```
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]();
```
，num 这个数字决定调用哪个，所以添加这样的代码

```
    if ((1 << num) & p->mask) {
      printf("%d: syscall %s -> %d\n", p->pid, syscalls_name[num], p->trapframe->a0);
    }
```
来输出。

其中 syscalls_name 数组是新建的，用于保存系统调用的名字，与上面的做好对仗。查阅 p 是什么东西时，发现这个结构中 pid，所以直接用。返回结果则由上面的 `p->trapframe->a0 = syscalls[num]();` 猜测。

```
// to the function that handles the
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_trace]   sys_trace,
[SYS_sysinfo] sys_sysinfo,
};

//Build below from above
char* syscalls_name[] = {
[SYS_fork]    "fork",
[SYS_exit]    "exit",
```

4、修改 fork，最后的问题是修改 fork，使子进程也能被 trace。我们

先找到 fork 的实现

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;//can kao

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  //you shang mian na tiao, caice zhejie xiugai zijinceng de mask
  np->mask = p->mask;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);
```

主要由 no->sz = p->sz 这条作参考，添加 np->mask=p->mask。

5、测试结果：

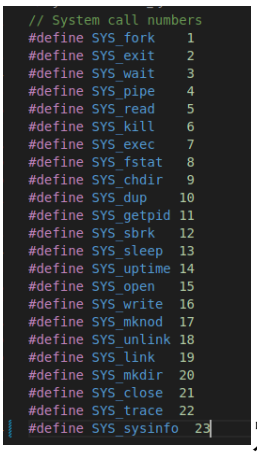
```
$ trace 2 usertests forkforkfork
usertests starting
3: syscall fork -> 4
test forkforkfork: 3: syscall fork -> 5
5: syscall fork -> 6
6: syscall fork -> 7
6: syscall fork -> 8
7: syscall fork -> 9
6: syscall fork -> 10
7: syscall fork -> 11
9: syscall fork -> 12
8: syscall fork -> 13
7: syscall fork -> 14
9: syscall fork -> 15
6: syscall fork -> 16
7: syscall fork -> 17
8: syscall fork -> 18
10: syscall fork -> 19
6: syscall fork -> 20
7: syscall fork -> 21
11: syscall fork -> 22
8: syscall fork -> 23
6: syscall fork -> 24
7: syscall fork -> 25
```

```
$ trace 2147483647 grep hello README
67: syscall trace -> 0
67: syscall exec -> 3
67: syscall open -> 3
67: syscall read -> 1023
67: syscall read -> 961
67: syscall read -> 321
67: syscall read -> 0
67: syscall close -> 0
$
```

符合要求。

6、trace 全流程简述（参考 xv6book）：首先在用户态发起 trace（），然后操作系统判断是否为系统调用，是则进入内核态，由 syscall.c 处理。Syscall.c 中 syscall 获取用户态中的参数，并进入相应的函数(trace，调用 sysproc.c 中的代码)。Trace 首先将保存于 trapframe 中的参数用 argint 提取出来，然后修改进程对应的 mask 变量。之后如果再有系

统调用 syscall 函数在调用结束后，会检查 mask 与调用函数，符合则输出。

7、syscall.h 的作用： 
```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_trace  22
#define SYS_sysinfo 23
```
定义 syscall 对应的编号。

8、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？：

为用户态下，实际是调用这个 

中的程序。

Part 2：Sysinfo

1、首先按提示，和上面一样操作，保证编译可以通过。

2、和 trace 一样，在相应的地方添加标识（主要是 syscall.c/h），并建立一个空函数。

3、按提示建立一个 sysinfo 的结构。

4、实现具体内容。首先要解决从哪弄数据，以及放到哪。

由提示，找到 sys_fstat()；可以法相其调用了这样一个函数

```
// Get metadata about file f.
// addr is a user virtual address, pointing to a struct stat.
int
filestat(struct file *f, uint64 addr)
{
  struct proc *p = myproc();
  struct stat st;

  if(f->type == FD_INODE || f->type == FD_DEVICE){
    ilock(f->ip);
    stati(f->ip, &st);
    iunlock(f->ip);
    if(copyout(p->pagetable, addr, (char *)&st, sizeof(st)) < 0)
      return -1;
    return 0;
  }
  return -1;
}
```

其中有个函数是 copyout，能够完成从内核到用户的复制。

```
// Copy from kernel to user.
// Copy len bytes from src to virtual address dstva in a given page table.
// Return 0 on success, -1 on error.
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;

  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
    if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)), src, n);

    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

所以我们用这样一句话完成复制

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

然后是参数怎么来的?

第一个参数 p->pagetable 实际就是当前进程的 pagetable，filestat 中也是这样操作的。

```
struct proc *p = myproc();
```

第二个参数是要复制去的地址。这个怎么来了？实际上和 trace 一样，sysinfo 的参数是一个指针，我们获取这个指针就好

```
uint64 addr;
struct sysinfo info;
struct proc *p = myproc();

argaddr(0, &addr);
```

最后两个参数很简单。

```
info.freemem = free_mem();
info.nproc = proc_num();
```

然后解决 info 中的信息从哪来的问题。

首先依照提示找到 kalloc.c,大致了解其作用

```
// Physical memory allocator, for user processes,
// kernel stacks, page-table pages,
// and pipe buffers. Allocates whole 4096-byte pages.
```

然后发现

```
struct {
  struct spinlock lock;
  struct run *freelist;//xianyi
} kmem;
```

其中的 freelist，猜测其为空闲内存，之后测试的时候发现还需要乘以 pagesize 才是实际大小。

```
void *
kalloc(void)
{
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
  return (void*)r;
}
```

在这里发现的。

所以我们直接在 kalloc.c 当中写一个函数实现功能即可

```
//new
uint64 free_mem(void) {
  struct run *free = kmem.freelist;
  uint64 n = 0;
  while(free) { //jisuan you duoshao kongxian ye
    n++;
    free = free->next;
  }
  return n * PGSIZE;
}
//new
```

然后找到 proc.c，大概看看代码，找到这个，确定从哪获取进程是否

UNUSED.

```
void
procinit(void)
{
  struct proc *p;

  initlock(&pid_lock, "nextpid");
  initlock(&wait_lock, "wait_lock");
  for(p = proc; p < &proc[NPROC]; p++) {
      initlock(&p->lock, "proc");
      p->state = UNUSED;
      p->kstack = KSTACK((int) (p - proc));
  }
}
```

然后写个函数实现想要的功能（依照其他函数仿写）

```
//fang xie de daima
uint64 proc_num(void) {
  uint64 n = 0;
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) n++;
    release(&p->lock);
  }
  return n;
}
```

5、测试结果

```
$ sysinfotest
sysinfotest: start
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
my student number is 19307110353
sysinfotest: OK
$
```

可以通过。

【遇到的问题与感想】

1、一开始做 trace 时，不知道在哪添加代码，无从下手，最后是翻阅参考手册，在对 xv6 的系统调用流程有一定的了解后，参考其他系统调用慢慢模仿着写出来的。之后测试的时候还发现忘了改写 fork，调试了很久，还改过 usertests 中的代码。写代码时也比较糊涂，主要是对许多内核调用的东西不太清楚，比如 trapframe 是什么，都是依靠看别的代码来学习以及查资料完成的。

2、做 sysinfo 的时候前面要简单许多，因为已经知道了该如何创建一个系统调用。比较难的问题是如何从内核到用户态复制数据，但是在

有了提示与参考的代码之后还是可以解决，虽然对例如 pagetable 之类的还不是很熟悉。获取具体的信息倒也不难，提示已经给出了地址，只需要阅读代码，找到想要的信息在哪即可。

3、总的来说，第一个实验更有挑战，需要知道系统调用的流程，如何创建系统调用，在哪输出信息。做完这个实验之后，对于系统调用的理解上升了不少。