Lab 1: Xv6 and Unix utilities

1950787 杨鑫

实验目的：

- 配置xv6的系统运行环境，了解qemu和相应的工具链
- 熟悉xv6环境以及一些基本的系统调用，并使用这些系统调用编写一些简单的用户程序
- 通过编程实现，对操作系统的一些基本核心概念（如进程，管道，文件等）加深认知和了解

xv6的系统调用：

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

该图来自xv6的官方文档，列出了基本的系统调用以及相应解释，在接下来的编程中会使用到它们。

实验步骤：

a. Boot xv6

首先根据实验说明安装虚拟机qemu和编译工具链RISC-V： QEMU 5.1, GDB 8.3, GCC, and Binutils.

本次实验以及后面所有实验的环境均是 ubuntu 20.04，因此执行以下命令完成配置：

```
sudo apt-get install git build-essential gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu

sudo apt-get remove qemu-system-misc
sudo apt-get install qemu-system-misc=1:4.2-3ubuntu6
```

然后需要拉取所需要的xv6资源：

```
git clone git://g.csail.mit.edu/xv6-labs-2020
```

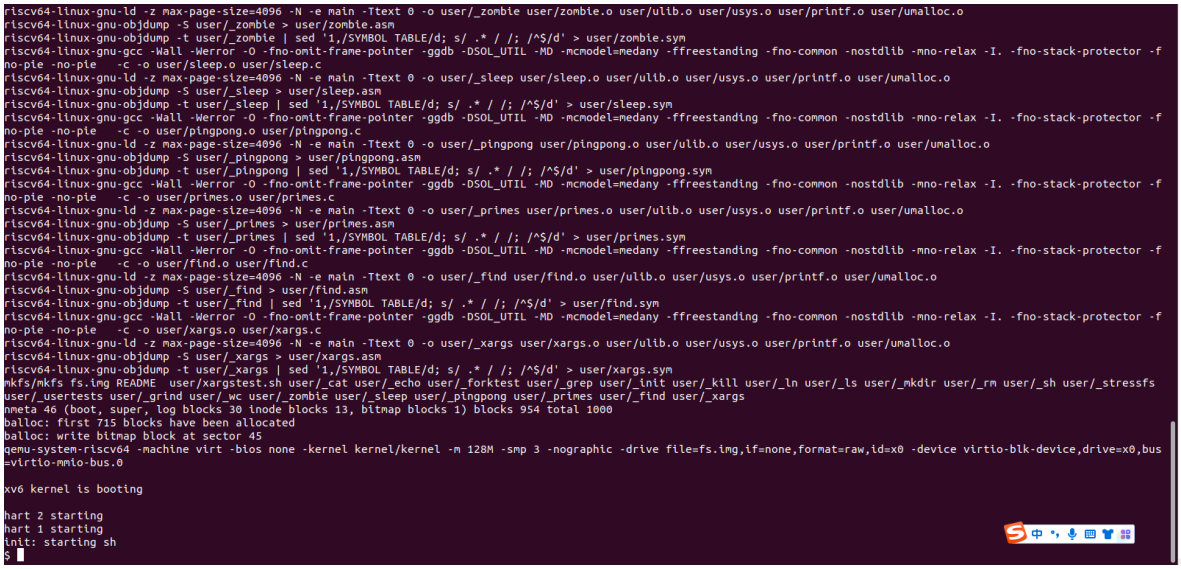
然后切换到第一个实验 util 分支：

```
cd xv6-labs-2020
git checkout util
```

届时，所有的代码资源已经准备好，接下来尝试构建和运行xv6，在 xv6-labs-2020 文件夹下执行命令：

```
make qemu
```

执行结果：



最下方出现

```
xv6 kernel is booting

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

即表示运行成功，表明实验环境正常，可以继续接下来的一系列实验。

b. sleep

实验目的：

实现一个用户级的 sleep 程序，它将使用系统内核提供的 sleep 函数来完成该功能，同时还需要检测参数个数等是否合法。

实验分析：

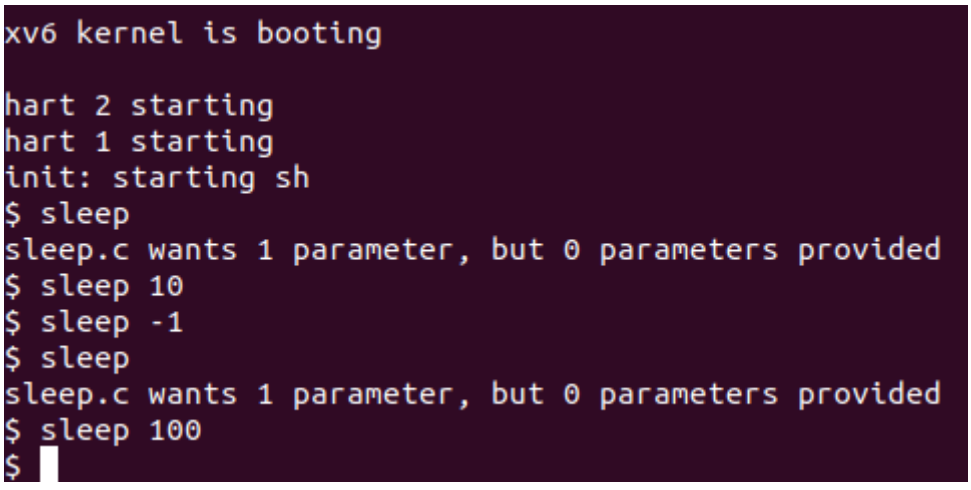
本实验很简单，只需要通过系统调用完成该功能即可，不过需要加上一些判断以确保输入合法。

实验核心代码：

新建 sleep.c 文件并输入：

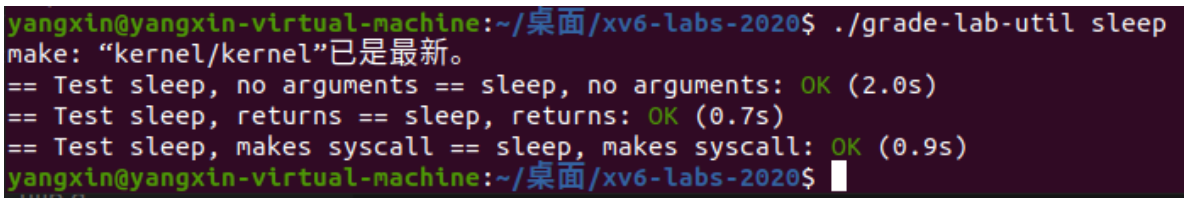
```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if(argc != 2){
    fprintf(2, "sleep.c wants 1 parameter, but %d parameters provided\n", argc -
1);
    exit(1);
  }
  int time = atoi(argv[1]);
  if (time <= 0) exit(1);
  sleep(time);
  exit(0);
}
```

测试：



可以看到，在缺省参数时会提示错误，参数合法时可以正常执行

评分：



可以通过所有测试

c. pingpong

实验目的：

实现一个用户级的 pingpong 程序，它使用系统调用完成如下功能：父进程向子进程通过管道发送一个字节，子进程收到后，输出：<pid> : received ping，pid 为其进程标识符，然后再通过管道向父进程写一个字节；父进程收到后，输出：<pid> : received pong，然后退出。

实验分析：

注意到这里需要考虑到进程的同步和通信机制。使用管道以完成通信功能，由于管道的p[0]为读端，p[1]为写端，而本实验要求子进程同时需要读和写，父进程也是需要读和写，故必须使用两个管道，分别完成父子进程的读写协作；此外，还需要考虑父子进程的关系，为了产生一个子进程，可以使用系统调用 fork()，使用后产生一个子进程，父进程里返回子进程 pid，而子进程里返回0，这样可以用于区分父子进程；最后是同步，为了保证在读写时有一定逻辑次序，必须要保证一个同步关系，这里是使用 write 和 read 的系统调用完成该关系，read 在管道为空时阻塞而write 在管道为满时阻塞，从而实现同步功能。

实验核心代码：

pingpong.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int
main(int argc, char *argv[])
{
  if (argc != 1) {
    fprintf(2, "pingpong.c wants 0 parameter, but received %d parameters\n",
argc - 1);
    exit(1);
  }

  int p1[2];
  int p2[2];

  pipe(p1);
  pipe(p2);

  if (fork() == 0) {
    close(p1[1]);
    close(p2[0]);

    char buffer[16];
    if (read(p1[0], buffer, 1) == 1) {
      printf("%d: received ping\n", getpid());
      write(p2[1], buffer, 1);
      exit(0);
    }
  } else {
    close(p1[0]);
    close(p2[1]);

    char buffer[16];
    write(p1[1], "p", 1);
```

```
    read(p2[0], buffer, 1);
    printf("%d: received pong\n", getpid());
  }

  exit(0);
}
```

测试：

```
$ pingpong
4: received ping
3: received pong
$
```

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-util pingpon
g
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (0.8s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

d. primes

实验目的：

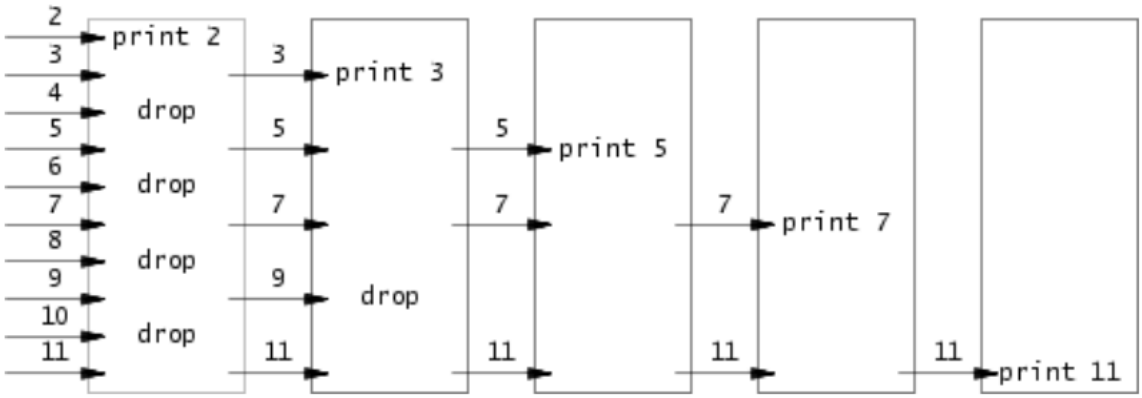
使用管道写一个并发版本的质数筛选器，本实验只需要35以内即可。

实验分析：

通过管道技术来筛选质数，即将一系列数字作为输入，筛选掉部分元素后继续将剩余的输入其子进程。需要用到父子进程和管道通信机制。具体算法如下：

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
```

具体来说，就是一个进程产生2,3,4...这些数字到最左边的管道，然后第一个管道将第一个数字（2）打印出来（一定为质数），然后检测其余数字，若是能被2整除则直接丢弃（不是质数），剩余的数字输入给第二个进程，然后重复与第一个进程一样的操作直至没有数字剩余，形象的图解如下：



注意到由于本实验环境的内存较小，所以无法同时使用太多的文件描述符和管道，因此要注意不使用的文件描述符要及时关闭，管道和进程也应当按需创建，否则可能无法支持程序正常运行。

实验核心代码：

prime.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

void child(int p[]) {
    close(p[1]);
    int mark;
    if (read(p[0], &mark, sizeof(mark)) == 0) return;
    printf("prime %d\n", mark);

    int t;
    int p1[2];
    pipe(p1);
    if (fork() == 0) {
        child(p1);
    } else {
        close(p1[0]);
        while(read(p[0], &t, sizeof(t)) > 0) {
            if (t % mark != 0) {
                write(p1[1], &t, sizeof(t));
            }
        }
        close(p1[1]);
        close(p[0]);
        wait(0);
    }

    exit(0);
}

int
main(int argc, char *argv[])
{
  if(argc != 1){
    fprintf(2, "primes.c wants 0 parameter, but %d parameters provided\n", argc
- 1);
```

```
    exit(1);
  }

  int p[2];
  pipe(p);

  if (fork() == 0) {
    child(p);
  } else {
    close(p[0]);
    int i;
    for (i = 2; i <= 35; i++) {
        write(p[1], &i, sizeof(i));
    }
    close(p[1]);
    wait(0);
  }

  exit(0);
}
```

测试：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (1.7s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所以测试

e. find

实验目的：

编写一个简单的用户级程序：在指定的文件路径下找到所有指定文件名的文件，并打印其相对路径

实验分析：

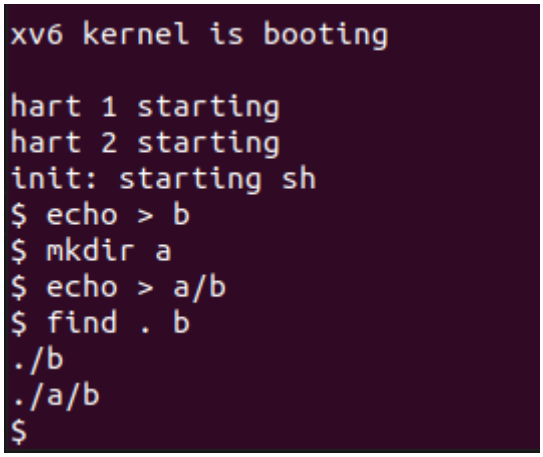
首先需要打开文件路径下的文件夹，然后依次读取其目录下所有子文件，若为文件则比较名字，相同则打印；若为文件夹则递归继续查询该文件夹下内容。要注意读取子文件时要跳过"." 和 ".."，分别代表当前目录和父目录，如果不跳过会陷入死循环并且打印不在指定路径下的文件。

实验核心代码：

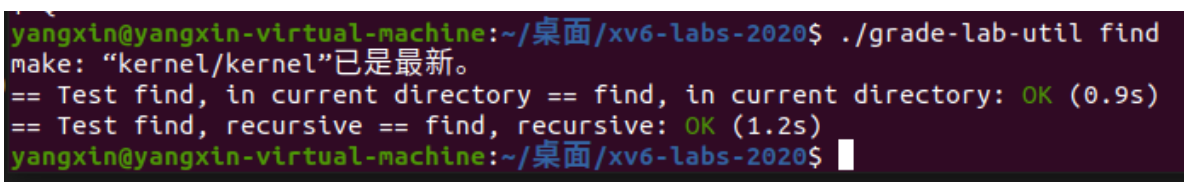
```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/fs.h"

// return the filename without parent directory.
char*
fmtname(char *path)
{
  char *p;
  for(p=path+strlen(path); p >= path && *p != '/'; p--)
    ;
  p++;

  return p;
}

// find all files in the path.
void find(char* path, char* name) {
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;

  if((fd = open(path, 0)) < 0){
    return;
  }

  if(fstat(fd, &st) < 0){
    close(fd);
    return;
  }

  switch(st.type){
  case T_FILE:
    if (strcmp(fmtname(path), name) == 0) {
        printf("%s\n", path);
    }
    break;

  case T_DIR:
    strcpy(buf, path);

    // add '/' to the end of the path.
    p = buf+strlen(buf);
    *p++ = '/';
    while(read(fd, &de, sizeof(de)) == sizeof(de)){
```

```
      if(de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") ==
0) // not recurse into . and ..
        continue;
      memmove(p, de.name, DIRSIZ);
      p[DIRSIZ] = 0;
      if(stat(buf, &st) < 0){
        continue;
      }

      find(buf, name);
    }
    break;
  }
  close(fd);
}

int
main(int argc, char *argv[])
{
  if(argc != 3){
    fprintf(2, "find.c wants 2 parameters, but %d parameters provided.\n", argc
- 1);
    exit(1);
  }

  find(argv[1], argv[2]);
  exit(0);
}
```

测试：



评分：



可以通过所有测试

f. xargs

实验目的：

编写一个简单的用户程序：从标准输入中读取内容然后执行命令，并且将读取的内容作为参数给该命令。

实验分析：

首先记录下xargs命令原有的参数，然后逐字符地从标准输入（文件描述符为0）读入，直到遇到 '\n' 或 ' '；当遇到 '\n' 时表示这一行结束，利用exec系统调用执行相应程序，其参数为xargs命令原有的参数加上该行输入的参数；当遇到 ' ' 时表示得到一个参数，将其附加到xargs命令的末尾，并继续读入剩余的参数。

实验核心代码：

```
#include "kernel/types.h"
#include "user/user.h"
#include "kernel/param.h"

int main(int argc, char *argv[]) {
  char xargs[MAXARG], buf[MAXARG];
  char *cmd[MAXARG];
  char *p = buf;
  int count = 0, cur = 0;

  for (int i = 1; i < argc; i++) cmd[count++] = argv[i];
  int k;
  while ((k = read(0, xargs, sizeof xargs)) > 0) {
    for (int i = 0; i < k; i++) {
        if (xargs[i] == '\n') {
            buf[cur] = 0;
            cmd[count++] = p;
            cmd[count] = 0;
            count = argc - 1;
            cur = 0;
            if (fork() == 0) exec(argv[1], cmd);
            wait(0);
        } else if (xargs[i] == ' ') {
            buf[cur++] = 0;
            cmd[count++] = p;
            p = &buf[cur];
        } else {
            buf[cur++] = xargs[i];
        }
    }
  }

  exit(0);
}
```

测试：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ echo operating system | xargs echo hello
hello operating system
$
```

评分：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-util xargs
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (1.6s)
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

可以通过所有测试

实验评分：

按照实验要求对本次实验的所有小实验进行评分，结果如图：

```
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$ ./grade-lab-util
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (1.3s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (0.9s)
== Test pingpong == pingpong: OK (1.1s)
== Test primes == primes: OK (0.9s)
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (1.2s)
== Test time ==
time: OK
Score: 100/100
yangxin@yangxin-virtual-machine:~/桌面/xv6-labs-2020$
```

通过了所有测试

问题以及解决办法：

a. 如何在用户程序里使用系统内核函数

在最开始写 sleep 用户程序的时候，需要用到系统调用 sleep，这就需要在用户空间的 sleep.c 使用内部的系统调用，我想查看内部系统调用的实现方法，于是开始溯源，首先在 user.h 中发现了系统调用 sleep 的声明：

```
4   // system calls
5   int fork(void);
6   int exit(int) __attribute__((noreturn));
7   int wait(int*);
8   int pipe(int*);
9   int write(int, const void*, int);
10  int read(int, void*, int);
11  int close(int);
12  int kill(int);
13  int exec(char*, char**);
14  int open(const char*, int);
15  int mknod(const char*, short, short);
16  int unlink(const char*);
17  int fstat(int fd, struct stat*);
18  int link(const char*, const char*);
19  int mkdir(const char*);
20  int chdir(const char*);
21  int dup(int);
22  int getpid(void);
23  char* sbrk(int);
24  int sleep(int);
```

但是却找不到其实现，于是我很疑惑，没有实现它为什么可以正常执行呢？然后我查询了网上的资源以及和同学进行探讨，发现在usys.pl中有一系列系统调用的入口：

```
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("fstat");
31  entry("link");
32  entry("mkdir");
33  entry("chdir");
34  entry("dup");
35  entry("getpid");
36  entry("sbrk");
37  entry("sleep");
```

它产生一个usys.S文件，里面部分内容为：

```
.global sleep
sleep:
 li a7, SYS_sleep
 ecall
 ret
```

即在遇到 sleep 的时候，将 SYS_sleep 装入 a7 寄存器（用于存放系统调用种类的寄存器），然后执行 ecall （即切换至内核态执行函数）。继续追踪至内核代码，有一个 syscall.h 文件，标识了系统调用编号（如上面的 SYS_sleep）

```
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
```

然后用户态切换至内核态时需要执行 trampoline.S，首先会保存用户寄存器状态：

```
        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        sd t2, 88(a0)
        sd s0, 96(a0)
        sd s1, 104(a0)
        sd a1, 120(a0)
        sd a2, 128(a0)
        sd a3, 136(a0)
        sd a4, 144(a0)
        sd a5, 152(a0)
        sd a6, 160(a0)
```

注意，用户的 a7 寄存器用来存放系统调用编号，a0,a1...用来存放参数

然后跳转至usertrap()

```
    # jump to usertrap(), which does not return
    jr t0
```

usertrap() 大概是检测一个专用的寄存器，判断是否 == 8，若是则说明是系统调用，否则则可能是中断或者其他异常，然后如果判断成功，就可以执行 syscall() 函数了

```
53    if(r_scause() == 8){
54      // system call
55
56      if(p->killed)
57        exit(-1);
58
59      // sepc points to the ecall instruction,
60      // but we want to return to the next instruction.
61      p->trapframe->epc += 4;
62
63      // an interrupt will change sstatus &c registers,
64      // so don't enable until done with those registers.
65      intr_on();
66
67      syscall();
```

syscall() 首先取出 a7 寄存器的值（即存放系统调用编号的），然后执行函数指针数组所对应的这个系统调用即可，返回至放入 a0 寄存器

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

函数指针数组为：

```
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,
};
```

这样就可以根据编号找到真正的系统调用函数了，我们这里跟进看一下 sys_sleep：

```
55  uint64
56  sys_sleep(void)
57  {
58    int n;
59    uint ticks0;
60
61    if(argint(0, &n) < 0)
62      return -1;
63    acquire(&tickslock);
64    ticks0 = ticks;
65    while(ticks - ticks0 < n){
66      if(myproc()->killed){
67        release(&tickslock);
68        return -1;
69      }
70      sleep(&ticks, &tickslock);
71    }
72    release(&tickslock);
73    return 0;
74  }
```

发现它居然不需要参数，而我们在用户区使用sleep系统调用时需要参数，其实它通过 argint() 取出了所需的参数并存到 n 上了，这就是为什么之前需要把函数参数放入一个寄存器，就是在这里将其读出来，至此，用户程序执行系统调用的完整过程就已经阐释清楚了。

b. 为什么内核函数（如sys_sleep）参数为void

在上面 **a** 已经解释清楚了，因为执行系统调用时，先把用户程序的参数全部放入寄存器中，在内核函数执行时直接从寄存器中读取即可。

c. find 中的一个小问题

我最开始写 find 的时候，发现写完了总是得不到结果，似乎是陷入了死循环，但是查看代码发现没有循环或者递归陷入死循环，因此十分困惑。

后来查阅资料时才发现文件夹下有两个特殊的子文件，即 "." 和 ".."，分别表示当前目录和父目录，如果不排除它们则肯定会陷入死循环中，因此需要跳过这两个节点。

实验心得：

1. 通过该实验成功构建了实验环境，为后序实验打下基础
2. 对于最基本的系统调用，如fork()，wait()，write()，read()等有了比较熟练的使用和了解，对基本概念，文件，管道等也有了更深的认知和了解，并能熟练使用
3. 管道一个读出端，一个写入端，可以通过write和read完成同步操作

4. fork函数在父子进程有不同的返回值，这样可以区分父子进程，并且子进程是从fork后开始继续执行
5. 同步操作还可以通过wait等实现，它使得父进程等待子进程执行完毕再执行
6. 对于文件结构有了熟悉和了解，如何递归搜索等
7. 本实验最重要的感悟是系统调用的实现和调用过程，以及用户态和内核态的转变（在5.a 中有详细叙述），结合理论知识，更加体会到了操作系统的重要性和设计的精妙性