

Lab1: Xv6 and Unix utilities

2351882 王小萌

Tongji University, 2025 Summer

代码仓库链接: <https://github.com/wmxmw/Tongji-University-xv6-labs-2021.git>

1.Boot xv6

1.1 实验目的

1.2 实验步骤

1.3 实验中遇到的问题和解决办法

1.4 实验心得

2.sleep

2.1 实验目的

2.2 实验步骤

2.3 实验中遇到的问题和解决办法

2.4 实验心得

3 pingpong

3.1 实验目的

3.2 实验步骤

3.3 实验中遇到的问题和解决办法

3.4 实验心得

4 primes

4.1 实验目的

4.2 实验步骤

4.3 实验中遇到的问题和解决办法

4.4 实验心得

5 find

5.1 实验目的

5.2 实验步骤

5.3 实验中遇到的问题和解决办法

5.4 实验心得

6 xargs

6.1 实验目的

6.2 实验步骤

6.3 实验中遇到的问题和解决办法

6.4 实验心得

7 实验检验得分

8 lab1 代码提交

1 Boot xv6

1.1 实验目的

启动 xv6,熟悉 xv6 及部分重要的系统调用。

1.2 实验步骤

1. 获取实验室的 xv6 源代码并检出 util 分支:

```
vale@Puppyyoo:~$ git clone git://g.csail.mit.edu/xv6-labs-2021
Cloning into 'xv6-labs-2021'...
remote: Enumerating objects: 7051, done.
remote: Counting objects: 100% (7051/7051), done.
remote: Compressing objects: 100% (3423/3423), done.
remote: Total 7051 (delta 3701), reused 6830 (delta 3600), pack-reused 0
Receiving objects: 100% (7051/7051), 17.20 MiB | 2.39 MiB/s, done.
Resolving deltas: 100% (3701/3701), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.
```

```
vale@Puppyyoo:~$ cd xv6-labs-2021
vale@Puppyyoo:~/xv6-labs-2021$
vale@Puppyyoo:~/xv6-labs-2021$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
vale@Puppyyoo:~/xv6-labs-2021$
```

2. 利用 make qemu 指令运行 xv6:

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ _
```

3. 输入 ls 指令能看到内容输出, 以下是使用 ls 指令列出的根目录下的文件:

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 23896
echo       2 5 22728
forktest   2 6 13080
grep        2 7 27248
init        2 8 23824
kill        2 9 22696
ln          2 10 22648
ls          2 11 26128
mkdir       2 12 22792
rm          2 13 22784
sh          2 14 41664
stressfs    2 15 23800
usertests   2 16 156008
grind       2 17 37968
wc          2 18 25040
zombie      2 19 22192
console     3 20 0
$ _
```

4. 在 xv6 中按 Ctrl + p 会显示当前系统的进程信息:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$
1 sleep  init
2 sleep  sh
```

5. 在 xv6 中按 Ctrl + a , 然后按 x 即可退出 xv6 系统。

```
$ QEMU: Terminated
vale@Puppyyoo:~/xv6-labs-2021$ _
```

1.3 实验中遇到的问题和解决办法

问题：在获取源代码时出现网络问题。

解决办法：更换网络环境，尝试使用更稳定的网络连接。

1.4 实验心得

本次实验让我初步了解了 xv6 操作系统的启动流程以及基本的系统调用机制。通过使用 `make qemu` 启动 xv6，并在其中运行各种命令，我对操作系统内核的基本功能有了更直观的认识。尤其是通过查看进程信息、退出系统等操作，熟悉了 xv6 的交互方式。

此外，在实验过程中也遇到了一些问题，例如如何正确运行用户程序。通过查阅资料和反复实践，我逐步掌握了 xv6 的编译与调试方法，提升了对操作系统底层机制的理解能力。这次实验为后续更深入的操作系统开发打下了坚实的基础。

2 sleep

2.1 实验目的

1. 为 xv6 实现 UNIX 程序 `sleep`。
2. 实现的 `sleep` 应当按用户指定的 ticks 数暂停，其中 tick 是 xv6 内核定义的时间概念，即定时器芯片两次中断之间的时间。解决方案应该在文件 `user/sleep.c` 中。

2.2 实验步骤

1. 参阅 `kernel/sysproc.c` 以获取实现 `sleep` 系统调用的 xv6 内核代码。

```

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 }

```

可以看到 user/user.h 提供了 sleep 的声明以便其他程序调用。

```

user > C user.h
23     int sleep(int);
24     int uptime(void);
25
26

```

用汇编程序编写的 user/usys.S 可以帮助 sleep 从用户区跳转到内核区；

```

97     ret
98     .global sleep
99     sleep:
100     li a7, SYS_sleep
101     ecall
102     ret

```

2. 仿照 user/echo.c 的写法，编写 user/sleep.c 文件，确保 main 函数调用 exit() 以退出程序。

```

C sleep.c  X  Makefile  D sleep.d
user > C sleep.c
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int main(int argc, char *argv[])
6  {
7      if (argc < 2)
8      {
9          fprintf(2, "need a param\n");
10         exit(1);
11     }
12     int i = atoi(argv[1]);
13     sleep(i);
14     exit(0);
15 }

```

3. 将 sleep 程序添加到 Makefile 中的 UPROGS 中即可运行;

```
179 UPROGS=\
180     $U/_cat\
181     $U/_echo\
182     $U/_forktest\
183     $U/_grep\
184     $U/_init\
185     $U/_kill\
186     $U/_ln\
187     $U/_ls\
188     $U/_mkdir\
189     $U/_rm\
190     $U/_sh\
191     $U/_stressfs\
192     $U/_usertests\
193     $U/_grind\
194     $U/_wc\
195     $U/_zombie\
196     $U/_sleep\
```

4. 里执行 make qemu 编译运行 xv6。再运行 sleep 命令结果:

```
vale@Puppyyoo:~$ cd xv6-labs-2021
vale@Puppyyoo:~/xv6-labs-2021$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL -DLAB_UTIL -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o user/sleep.o user/sleep.c
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_sleep user/sleep.o user/ulib.o user/usys.o user/printf.o user/unalloc.o
riscv64-linux-gnu-objdump -S user/_sleep > user/sleep.asm
riscv64-linux-gnu-objdump -t user/_sleep | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/sleep.sym
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie user/_sleep
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000
ballocc: first 623 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 -drive file=fs.img,if=none,format=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
$ sleep 10
$ sleep
need a param
```

未输入参数会有输入错误提示:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ sleep
need a param
```

5. 实验单项评分

```

sleep          2 20 22640
console        3 21 0
$ QEMU: Terminated
vale@Puppyoo:~/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.4s)
== Test sleep, returns == sleep, returns: OK (0.8s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
vale@Puppyoo:~/xv6-labs-2021$

```

2.3 实验中遇到的问题和解决办法

问题：程序在输入了不正确的参数时崩溃。

解决办法：在 main 函数中，判断参数数量是否等于 2，如果不是,则表示输入参数数量不正确，返回 Usage: need a param。

```

if (argc < 2)
{
    fprintf(2, "need a param\n");
    exit(1);
}

```

2.4 实验心得

实验时需要明白程序的功能，并且阅读该程序相关的依赖文件，理清参数传递和头文件依赖关系等，避免参数传递出错或缺少头文件等。在编译并运行 sleep 程序之前，我们除了需要正确配置 xv6 环境之外，还需要及时让系统支持并正确实现 sleep 系统调用，否则程序将无法被系统调用并运行测试。

可以使用循环和条件语句进行参数的检查和处理,避免参数出错。提高程序的鲁棒性

3 PingPong

3.1 实验目的

编写一个使用 UNIX 系统调用的程序，在两个进程之间“ping-pong”一个字节，使用两个管道，每个方向一个。父进程应该向子进程发送一个字节;子进程应该打印“: received ping”，其中是进程 ID，并在管道中写入字节发送给父进程，然后退出;父级应该从读取从子进程而来的字节，打印“: received pong”，然后退出。解决方案应该在文件 user/pingpong.c 中。

3.2 实验步骤

1. 编写 pingpong.c 的代码程序;

使用的系统调用:

使用 pipe 来创建管道; 使用 fork 创建子进程; 使用 read 从管道中读取数据, 并且使用 write 向管道中写入数据; 使用 getpid 获取调用进程的 pid。

```
C sleep.c  C pingpong.c x  M Makefile  D sleep.d
user > C pingpong.c
1  #include "kernel/types.h"
2  #include "user/user.h"
3  #include "kernel/fcntl.h"
4  int main(){
5
6      int parent_fd[2];
7      int child_fd[2];
8      pipe(parent_fd[2]);
9      pipe(child_fd[2]);
10     char buf[8];
11     if (fork() 0)
12     {
13         read(parent_fd[0], buf, sizeof buf);
14         int id = getpid(); //Get the ID of the current process and print it
15         printf("%d: received %s\n", id, buf);
16         write(child_fd[1], "pong", 4); //Write string "pong" to child_fd[1] pipe
17         //Close the read side and write side of child_fd.
18         close(child_fd[0]);
19         close(child_fd[1]);
20     }
21     else
22     {
23         int id = getpid();
24         write(parent_fd[1], "ping", 4); //Write the string "ping" to the parent_fd[1] pipe
25         close(parent_fd[1]);
26         int status;
27         wait(&status);
28         read(child_fd[0], buf, sizeof buf);
29         printf("%d: received %s\n", id, buf);
30         close(child_fd[0]);
31     }
32 }
```

2. 将 pingpong 程序添加到 Makefile 中的 UPROGS 中即可运行;

```
95     $U/_zombie\
96     $U/_sleep\
97     $U/_pingpong\
98
```

3. 保存后在终端里执行 make qemu 编译运行 xv6。然后在命令行中输入 pingpong , 运行 pingpong 程序, 结果:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

4. 单项评分

```
vale@Puppyyoo:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (0.7s)
```

3.3 实验中遇到的问题和解决办法

1. 问题：程序陷入死锁或出现死循环。

解决办法：检查是否有未正确关闭的文件描述符，在合适的时机关闭读写端，避免导致进程阻塞或死锁的情况

3.4 实验心得

通过这个实验，我了解了管道的概念和使用方法，以及父子进程间的通信机制：使用 `pipe()` 函数创建管道，使用 `fork()` 函数创建子进程，使用文件描述符来进行进程间的读写操作。在父子进程间通信时，必须确保管道的正确打开和关闭，避免造成进程阻塞或死锁的情况。进程通信要实现正确的进程同步。通过适当的管道读写操作和进程等待机制（如使用 `wait()` 函数）实现了父进程和子进程的同步，确保了数据的正确交换和打印顺序。

对父子进程关系的理解：子进程是由父进程派生出来的，它们共享某些资源，并在不同的代码路径中执行。子进程的建立与 `fork` 系统调用有关，会将父进程的寄存器、内存空间和进程控制块复制一份，生成子进程，并且将子进程的进程控制块中的父进程指针指向其父进程；此时父子进程几乎完全一致，为了方便程序判断自己是父还是子，`fork` 会给两个进程不同的返回值，父进程得到的是子进程的 `pid`，而子进程的返回值则为 0。

4 Primes

4.1 实验目的

使用管道编写一个基本筛选器的并发版本，将 2 至 35 中的素数筛选出来。想法来自于 Unix 管道的发明者 Doug McIlroy。学习使用 `pipe` 和 `fork` 来设置管道。第一个进程将数字 2 到 35 输入管道。对于每个素数创建一个进程，该进程通过一个管道从左边的邻居读取数据，并通过另一个管道向右边的邻居写入数据。由于 xv6 的文件描述符和进程数量有限，第一个进程可以在 35 处停止。解决方案位于 `user/primes.c` 中。

4.2 实验步骤

1. 编写 `primes.c` 的代码程序；

在程序中，创建父进程，父进程将数字 2 到 35 输入管道，在需要时创建管道中的进程。对于 2-35 中的每个素数创建一个进程，进程之间需要进行数据传递：该进程通过一个管道从左边的父进程读取数据，并通过另一个管道向右边子进程写入

数据。对于每一个生成的进程而言，当前进程最顶部的数即为素数；对每个进程中剩下的数进行检查，如果是素数则保留并写入下一进程，如果不是素数则跳过。完成数据传递或更新时，需要及时关闭一个进程不需要的文件描述符（防止程序在父进程到达 35 之前耗尽 xv6 的资源）。当管道的写入端关闭时，read 函数返回 0。

```
C sleep.c C pingpong.c C primes.c X C mkdir.c M Makefile
user > C primes.c
1 #include "kernel/types.h"
2 #include "user/user.h"
3 void filter(int readfd) {
4     int prime;
5     if (read(readfd, &prime, sizeof(prime)) == 0) { // 读取管道中的第一个数
6         close(readfd); // 如果没有数可读，关闭读端并返回
7         return;
8     }
9     printf("prime %d\n", prime);
10    int p[2];
11    pipe(p); // 创建新的管道
12    if (fork() == 0) { // 创建子进程
13        close(p[1]); // 关闭写端
14        filter(p[0]); // 子进程递归调用filter函数继续筛选素数
15        close(p[0]);
16    }
17    else {
18        close(p[0]); // 关闭读端
19        int num;
20        while (read(readfd, &num, sizeof(num)) > 0) { // 读取父管道的数
21            if (num % prime != 0) { // 如果不能被当前素数整除
22                write(p[1], &num, sizeof(num)); // 写入新管道
23            }
24        }
25        close(p[1]); // 关闭写端
26        close(readfd); // 关闭读端
27        wait(0); // 等待子进程结束
28    }
29 }
30 int main(int argc, char *argv[]) {
31     int p[2];
32     pipe(p); // 创建初始管道
33     if (fork() == 0) { // 创建子进程
34         close(p[1]); // 关闭写端
35         filter(p[0]); // 子进程调用filter函数开始筛选
36         close(p[0]);
37     }
38     else {
39         close(p[0]); // 关闭读端
40         for (int i = 2; i <= 35; i++) {
41             write(p[1], &i, sizeof(i)); // 将2到35的数写入管道
42         }
43         close(p[1]); // 关闭写端
44         wait(0); // 等待子进程结束
45     }
46     exit(0);
47 }
```

2. 将 primes 程序添加到 Makefile 中的 UPROGS 中即可运行；

```
96 $U/_sleep\
97 $U/_pingpong\
98 $U/_primes\
99
```

3. 保存后在终端里执行 make qemu 编译运行 xv6；并在命令行中输入 primes，出现：

```

vale@Puppyyoo: /xv6-labs-2021$ make qemu
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -g
-fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie
riscv64-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o
f.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_primes > user/primes.asm
riscv64-linux-gnu-objdump -t user/_primes | sed '1,/SYMBOL TABLE/
mkfs/mkfs fs.img README user/xargstest.sh user/_cat user/_echo u
ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests u
er/_primes
mmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap bloc
ballocc: first 672 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kerne
raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.

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
$

```

5. 在终端里运行 `./grade-lab-util primes` 可进行评分：

```

$ QEMU: Terminated
vale@Puppyyoo: ~/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.5s)

```

4.3 实验中遇到的问题和解决办法

问题：如果在父进程中不使用 `wait()` 函数,会不会出现问题。

解决办法：本次的测试结果与不使用 `wait()` 函数的结果相一致，但父进程可能会在子进程执行完 成之前继续执行自己的代码。这可能会导致父进程在子进程还没有完成时就退出，从而使子进程成 为没有父进程的进程。此外，没有正确等待子进程完成的父进程可能无法获取子进程的退出状态， 也无法做进一步的处理。 `wait()` 函数确保了父进程在子进程完成之后继续执行

4.4 实验心得

实现过程中，需要维护读端和写端的管道，不断读取上一个进程写入管道的内容，并在 合适的条 件下生成子进程并将其它数字写入管道。 加深了对 `fork` 系统调用的理解:子进程和父进程在 `fork()` 调用点之后的代码是独立执行的，并且 拥有各自独立的地址空间。因此，父进程和子进程可以在 `fork()` 后继续执行不同的逻辑，实现 并行

或分支的程序控制流程。因此数据如果要想实现传递，则可以在 进行数据“交换”，将子变为下一级的父，从而实现了数据传递。

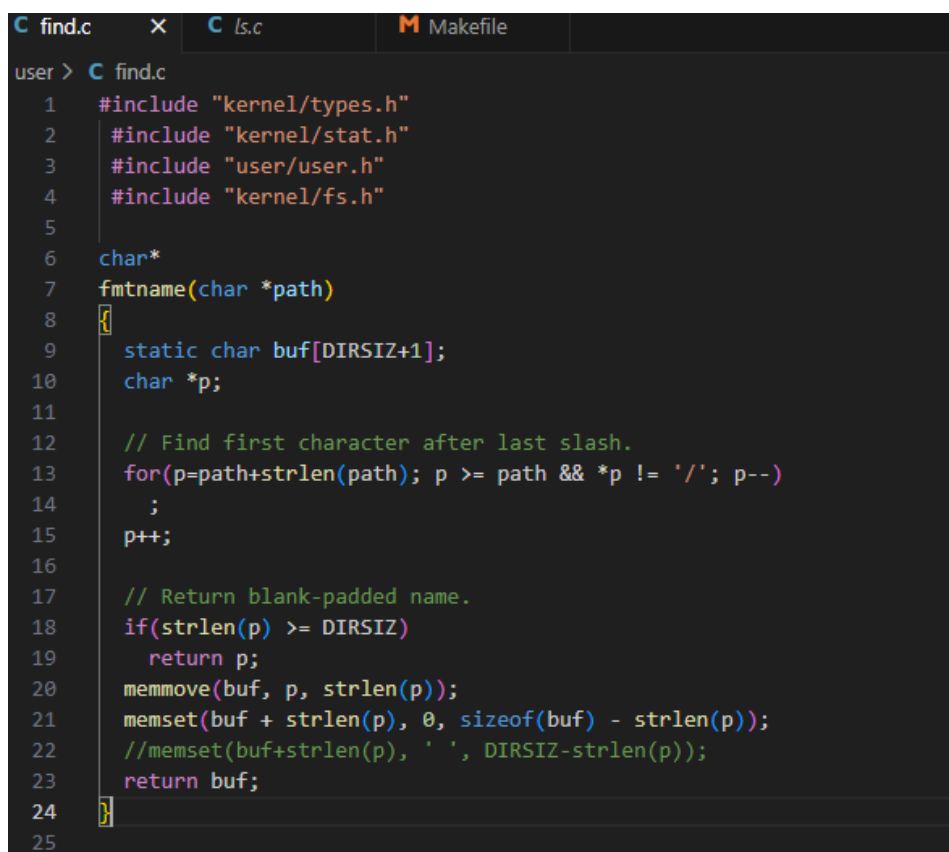
5 Find

5.1 实验目的

1. 理解文件系统中目录和文件的基本概念和组织结构。
2. 熟悉在 xv6 操作系统中使用系统调用和文件系统接口进行文件查找操作。
3. 应用递归算法实现在目录树中查找特定文件。

5.2 实验步骤

1. 查看 user/lis.c 以了解如何读取目录。 user/lis.c 中包含一个 fmtname 函数，用于格式化文件 的名称。它通过查找路径中最后一个 '/' 后的第一个字符来获取文件的名称部分。



```
user > C find.c
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4  #include "kernel/fs.h"
5
6  char*
7  fmtname(char *path)
8  {
9      static char buf[DIRSIZ+1];
10     char *p;
11
12     // Find first character after last slash.
13     for(p=path+strlen(path); p >= path && *p != '/'; p--)
14         ;
15     p++;
16
17     // Return blank-padded name.
18     if(strlen(p) >= DIRSIZ)
19         return p;
20     memmove(buf, p, strlen(p));
21     memset(buf + strlen(p), 0, sizeof(buf) - strlen(p));
22     //memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
23     return buf;
24 }
25
```

2. 编写 find.c 的代码程序。其中的 fmtname 函数与 ls.c 相同。

```

user > C find.c
27 void find(char *path, char *target) {
28     char buf[512], *p;
29     int fd;
30     struct dirent de;
31     struct stat st;
32     // 打开路径
33     if ((fd = open(path, 0)) < 0) {
34         fprintf(2, "find: cannot open %s\n", path);
35         return;
36     }
37     // 获取文件状态
38     if (fstat(fd, &st) < 0) {
39         fprintf(2, "find: cannot stat %s\n", path);
40         close(fd);
41         return;
42     }
43     switch (st.type) {
44         case T_FILE:
45             if (strcmp(dirname(path), target) == 0) {
46                 printf("%s\n", path);
47             }
48             break;
49         case T_DIR:
50             if (strlen(path) + 1 + DIRSIZ + 1 > sizeof(buf)) {
51                 printf("find: path too long\n");
52                 break;
53             }
54             strcpy(buf, path);
55             p = buf + strlen(buf);
56             *p++ = '/';
57             while (read(fd, &de, sizeof(de)) == sizeof(de)) {
58                 if (de.inum == 0)
59                     continue;
60                 memmove(p, de.name, DIRSIZ);
61                 p[DIRSIZ] = 0;
62                 if (strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)
63                     continue;
64                 find(buf, target);
65             }
66             break;
67     }
68     close(fd);
69 }

```

```

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(2, "Usage: find <path> <filename>\n");
        exit(1);
    }
    find(argv[1], argv[2]);
    exit(0);
}

```

3. 将 find 程序添加到 Makefile 中的 UPROGS 中即可运行;

```

37     $U/_pingpong\
38     $U/_primes\
39     $U/_find\
40

```

4. 保存后在终端里执行 make qemu 编译运行 xv6; 然后在命令行中输入以下命令：
mkdir 为创建文件； echo 为直接将数据写入文件，若文件存在则直接写入，若不存在的话新建并写入。

```
echo > b
mkdir a
echo > a/b
find . b
find a b
```

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ echo > b
$ mkdir a
$ echo > a/b
$ find . b
./b
./a/b
$ find a b
a/b
$ QEMU: Terminated
vale@Puppyoo:~/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.0s)
vale@Puppyoo:~/xv6-labs-2021$
```

5. 在终端里运行 `./grade-lab-util find` 可进行评分：

```
vale@Puppyoo:~/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (0.9s)
== Test find, recursive == find, recursive: OK (1.1s)
```

5.3 实验中遇到的问题和解决办法

问题：无法继续向下递归地查找子目录中的文件。

解决办法：`ls.c` 程序只能提供基本的文件和目录信息，需要对 `find` 函数进行递归遍历。

5.4 实验心得

通过这个实验，深入理解了文件系统中目录和文件的关系，以及如何通过系统调用和文件系统接口 来访问和操作文件。还学会了使用递归算法实现对目录树的深度遍历，以便能够在整个目录结构中查找符合条件的文件。

6 Xargs

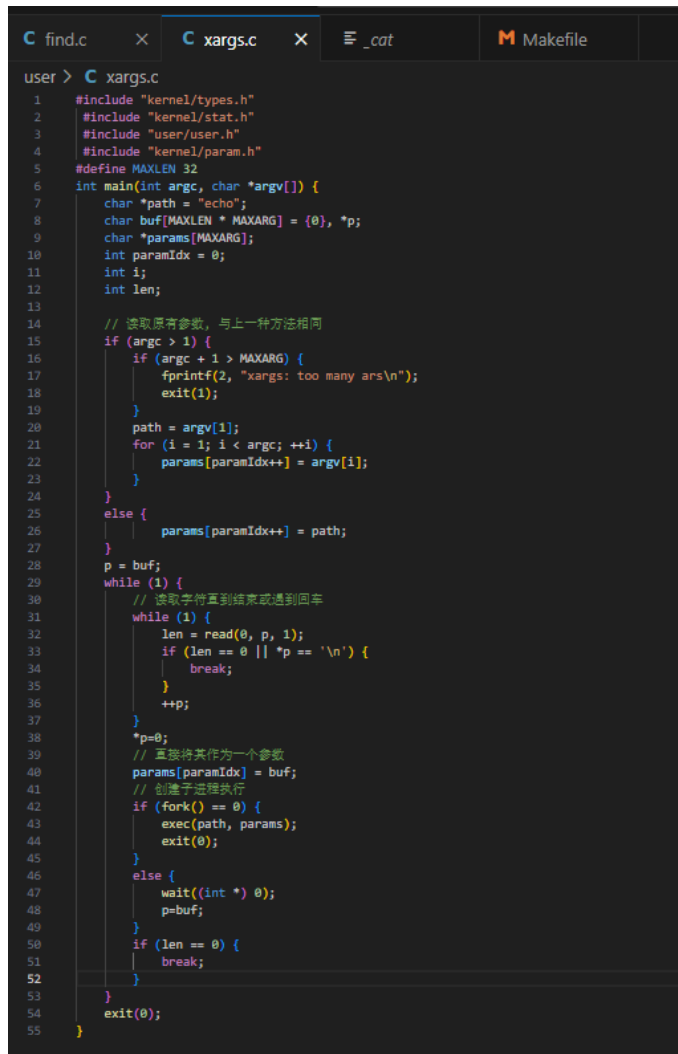
6.1 实验目的

编写一个 UNIX `xargs` 程序的简单版本：从标准输入中读取行，并为每一行运行一个命令，将行作为参数 提供给命令。解决方案位于文件 `user/xargs.c` 中。

6.2 实验步骤

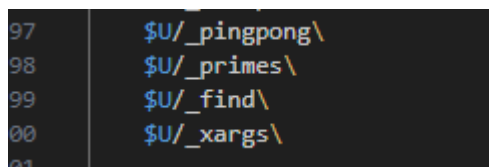
1. 编写 xargs.c 的代码程序;

使用 fork 和 exec 对每行输入调用命令，在父进程中使用 wait 等待子进程完成命令。要读取单个输入行，一次读取一个字符，直到出现换行符（'\n'）



```
1 #include "kernel/types.h"
2 #include "kernel/stat.h"
3 #include "user/user.h"
4 #include "kernel/param.h"
5 #define MAXLEN 32
6 int main(int argc, char *argv[]) {
7     char *path = "echo";
8     char buf[MAXLEN * MAXARG] = {0}, *p;
9     char *params[MAXARG];
10    int paramIdx = 0;
11    int i;
12    int len;
13
14    // 读取原有参数，与上一种方法相同
15    if (argc > 1) {
16        if (argc + 1 > MAXARG) {
17            fprintf(2, "xargs: too many args\n");
18            exit(1);
19        }
20        path = argv[1];
21        for (i = 1; i < argc; ++i) {
22            params[paramIdx++] = argv[i];
23        }
24    }
25    else {
26        params[paramIdx++] = path;
27    }
28    p = buf;
29    while (1) {
30        // 读取字符直到结束或遇到回车
31        while (1) {
32            len = read(0, p, 1);
33            if (len == 0 || *p == '\n') {
34                break;
35            }
36            ++p;
37        }
38        *p = 0;
39        // 直接将其作为一个参数
40        params[paramIdx] = buf;
41        // 创建子进程执行
42        if (fork() == 0) {
43            exec(path, params);
44            exit(0);
45        }
46        else {
47            wait((int *) 0);
48            p = buf;
49        }
50        if (len == 0) {
51            break;
52        }
53    }
54    exit(0);
55 }
```

2. 将 xargs 程序添加到 Makefile 中的 UPROGS 中即可运行;



```
97 $U/_pingpong\
98 $U/_primes\
99 $U/_find\
00 $U/_xargs\
01
```

3. 保存后在终端里执行 make qemu 编译运行 xv6 然后在命令行中输入以下命令 : sh < xargstest.sh

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ hello
hello
hello
$ $ QEMU: Terminated
v@Puppyoo:~/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.4s)
v@Puppyoo:~/xv6-labs-2021$

```

4. 在终端里运行 `./grade-lab-util xargs` 可进行评分：

```

v@Puppyoo:~/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (0.6s)

```

6.3 实验中遇到的问题和解决办法

问题：子进程执行命令时没有对执行结果进行处理。


解决办法：当前代码在创建子进程后，调用了 `exec` 函数执行命令，但没有对命令执行结果进行处理。可以使用 `wait` 函数等待子进程执行完毕，并检查执行结果。

6.4 实验心得

通过这个实验，我熟悉了命令行参数的获取和处理，包括选项解析和参数拆分。学习了外部命令的执行，调用 `exec` 函数来执行外部命令，理解执行外部程序的基本原理。并且运行中需要注意，对文件系统的修改会在运行 `qemu` 时持续存在；要获得一个干净的文件系统，运行 `make clean`，然后再运行 `make qemu`。

7 实验检验得分

1. 在实验目录下创建 `time.txt`，填写完成实验时间数。
2. 创建 `answers-lab-Util` 文件将程序运行结果填入。

 answers-lab-Util.txt	2025/7/13 0:52	文本文档	1 KB
----------------------------------------------------------------------------------------------------------	----------------	------	------

3. 在终端中执行 `make grade`,得到 lab1 总分：

```

== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (2.0s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (1.1s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.1s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (0.9s)
== Test primes ==
$ make qemu-gdb
primes: OK (1.2s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.0s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (0.9s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.1s)
== Test time ==
time: OK
Score: 100/100

```

8 Lab1 代码提交

配置用户邮箱与名称:

```

vale@Puppyyoo:~/xv6-labs-2021$ git config --global user.email 2109477059@qq.com
vale@Puppyyoo:~/xv6-labs-2021$ git config --global user.name WangXiaomeng

```

保存并提交到本地分支:

```

vale@Puppyyoo:~/xv6-labs-2021$ git commit -m "util finish"
[util 6d13c36] util finish
12 files changed, 277 insertions(+)
create mode 100644 answers-lab-Util.txt
create mode 100644 time.txt
create mode 100644 user/find.c
create mode 100644 user/find.c:Zone.Identifier
create mode 100644 user/pingpong.c
create mode 100644 user/pingpong.c:Zone.Identifier
create mode 100644 user/primes.c
create mode 100644 user/primes.c:Zone.Identifier
create mode 100644 user/sleep.c
create mode 100644 user/xargs.c
create mode 100644 user/xargs.c:Zone.Identifier

```