

同济大学操作系统课程设计——Lab1: Xv6 and Unix utilities

2151422武芷朵 Tongji University, 2024 Summer

同济大学操作系统课程设计——Lab1: Xv6 and Unix utilities

2151422武芷朵 Tongji University, 2024 Summer

1.Boot xv6 (easy)

- 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 (easy)

- 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 (easy)

- 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 (moderate)/(hard)

- 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 (moderate)

- 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 (moderate)

- 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

Lab1: Xv6 and Unix utilities 实用工具实验

项目地址: [wzd232604/TJOS-xv6-2024-labs](https://github.com/wzd232604/TJOS-xv6-2024-labs): 同济大学操作系统课程设计-xv6实验([github.com](https://github.com/wzd232604/TJOS-xv6-2024-labs))

1.Boot xv6 (easy)

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:

```
$ git clone git://g.csail.mit.edu/xv6-labs-2021

Cloning into 'xv6-labs-2021'...

...

$ cd xv6-labs-2021

$ git checkout util

Branch 'util' set up to track remote branch 'util' from 'origin'.

Switched to a new branch 'util'
```

2. 利用make qemu指令运行xv6:

```
wzd@ubuntu:~/Desktop$ 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 3702), reused 6830 (delta 3600), pack-reused 0
Receiving objects: 100% (7051/7051), 17.20 MiB | 3.02 MiB/s, done.
Resolving deltas: 100% (3702/3702), done.
warning: remote HEAD refers to nonexistent ref, unable to checkout.

wzd@ubuntu:~/Desktop$ cd xv6-labs-2021cd xv6-labs-2021
bash: cd: too many arguments
wzd@ubuntu:~/Desktop$ cd xv6-labs-2021
wzd@ubuntu:~/Desktop/xv6-labs-2021$ git checkout util
Branch 'util' set up to track remote branch 'util' from 'origin'.
Switched to a new branch 'util'
wzd@ubuntu:~/Desktop/xv6-labs-2021$ make qemu
riscv64-linux-gnu-gcc -c -o kernel/entry.o kernel/entry.S
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 kernel/kalloc.o kernel/kalloc.c
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -DSOL_UTIL
```

3. 输入ls指令能看到内容输出, 这些是使用ls指令列出根目录下的文件。

在 xv6 中按 Ctrl + p 会显示当前系统的进程信息。

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

```
wzd@ubuntu: ~/Desktop/xv6-labs-2021
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
xargstest.sh 2 3 93
cat        2 4 23904
echo       2 5 22736
forktest   2 6 13088
grep        2 7 27256
init        2 8 23832
kill        2 9 22704
ln          2 10 22656
ls          2 11 26136
mkdir       2 12 22800
rm          2 13 22792
sh          2 14 41672
stressfs    2 15 23808
usertests   2 16 156016
grind       2 17 37976
wc          2 18 25040
zombie      2 19 22200
console     3 20 0
$
```

```
console 3 20 0
$
1 sleep init
2 sleep sh
QEMU: Terminated
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

1.3 实验中遇到的问题和解决办法

在安装环境时遇到较多困难，通过网上查阅资料，和同学讨论等方法正确安装虚拟机等。

1.4 实验心得

- 通过本实验，我初步了解了 xv6 这一操作系统内核，同时也了解了 qemu 模拟器的使用方法：可以直接使用 make qemu 编译并在 qemu 中运行 xv6。若一切正常，make 将会执行一系列的编译和链接操作，输出大量的log，并且用 qemu 启动 xv6 系统。
- xv6 启动后，init 进程会启动一个 shell 等待用户的命令。在这个简易的 shell 中，可以使用 ls 指令列出根目录下的文件。
- 若要结束运行 xv6 并终止 qemu，需在键盘上同时按下 Ctrl+A 键，然后按下 X 键，即可终止 qemu 的运行。

2.sleep (easy)

2.1 实验目的

实现xv6的UNIX程序sleep：sleep应该使当前进程暂停用户指定的时钟周期数，其中tick计时数是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

使用系统调用sleep：实现的源码放置在 `user/sleep.c`。

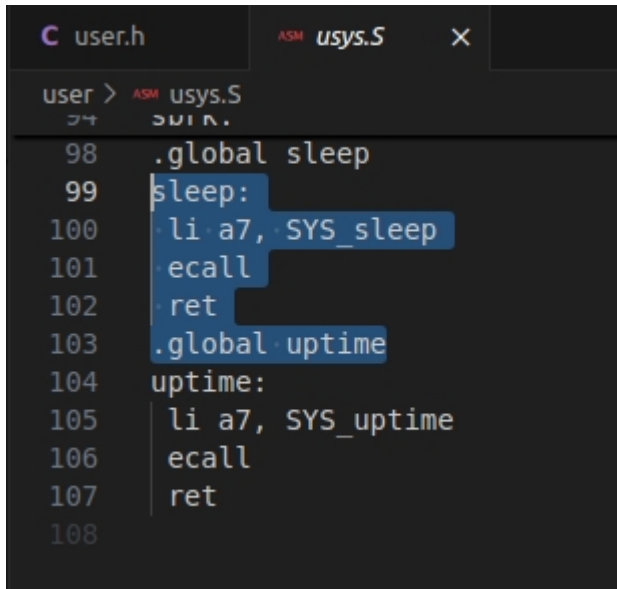
1. 参阅 `kernel/sysproc.c` 以获取实现sleep系统调用的xv6内核代码：

```
sysproc.c  X
kernel > C sysproc.c > sys_sleep(void)
54
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 }
```

user/user.h 提供了sleep的声明以便其他程序调用:

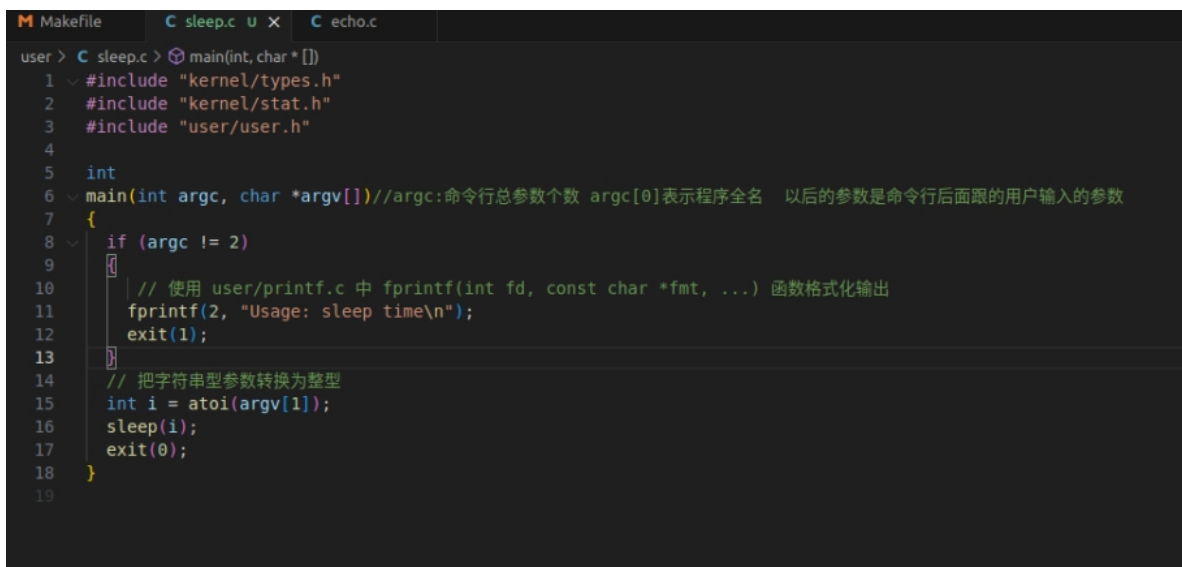
```
user.h  X
user > C user.h > sleep(int)
1 struct stat;
2 struct rtcdate;
3
4 // system calls
5 int fork(void);
6 int exit(int) __attribute__((noreturn));
7 int wait(int*);
8 int pipe(int*);
9 int write(int, const void*, int);
10 int read(int, void*, int);
11 int close(int);
12 int kill(int);
13 int 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);
25 int uptime(void);
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；



```
C user.h  ASM usys.S X
user > ASM usys.S
98 .global sleep
99 sleep:
100     li a7, SYS_sleep
101     ecall
102     ret
103 .global uptime
104 uptime:
105     li a7, SYS_uptime
106     ecall
107     ret
108
```

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



```
M Makefile  C sleep.c U X  C echo.c
user > C sleep.c > main(int, char *[])
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char *argv[]) //argc:命令行总参数个数  argv[0]表示程序全名  以后的参数是命令行后面跟的用户输入的参数
7  {
8      if (argc != 2)
9      {
10         // 使用 user/printf.c 中 fprintf(int fd, const char *fmt, ...) 函数格式化输出
11         fprintf(2, "Usage: sleep time\n");
12         exit(1);
13     }
14     // 把字符串型参数转换为整型
15     int i = atoi(argv[1]);
16     sleep(i);
17     exit(0);
18 }
19
```

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

```
M Makefile M X C sleep.c U C echo.c
M Makefile
177 .PRECIOUS: %.o
178
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\
197
198
```

4. 保存后在终端里执行make qemu编译运行xv6;
5. 在命令行中输入 sleep + 参数(例如10), 则系统会在10个时钟周期后重新出现命令行;

```
hart 2 starting
hart 1 starting
init: starting sh
$ sleep 10
$
```

如果命令行参数不等于2个, 则打印错误信息:

```
hart 2 starting
hart 1 starting
init: starting sh
$ sleep 1 1
Usage: sleep time
$ sleep
Usage: sleep time
$
```

6. 在终端里运行 ./grade-lab-util sleep 可进行评分:

```
QEMU: terminated
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-util sleep
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (1.6s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

2.3 实验中遇到的问题和解决办法

1. 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: sleep time。

2. 问题：运行命令 `./grade-lab-util sleep` 时报错为 `exec ./grade-lab-util failed`。

- 解决办法：需要同时按下 Ctrl+A 键，然后按下 X 键，终止 qemu 的运行,在终端执行命令。

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。
- 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。

3 pingpong (easy)

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

使用系统调用：

- 使用pipe来创造管道;
- 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。

实现的源码放置在 `user/pingpong.c`。

1. 编写 `pingpong.c` 的代码程序;


```
Makefile M C pingpong.c U X
user > C pingpong.c > main(int, char *[])
1  #include "kernel/types.h"
2  #include "user/user.h"
3
4  int
5  main(int argc, char *argv[])
6  {
7      int parent_fd[2];
8      int child_fd[2];
9      pipe(parent_fd);
10     pipe(child_fd);
11     char buf[8]; // 存储传输的数据
12     // 使用 fork() 创建子进程，并通过返回值判断当前进程是否为子进程 (pid为0)
13     if (fork() == 0) {
14         // child process
15         read(parent_fd[0], buf, 4);
16         printf("%d: received %s\n", getpid(), buf);
17         write(child_fd[1], "pong", strlen("pong"));
18     }
19     else {
20         // parent process
21         write(parent_fd[1], "ping", strlen("ping"));
22         wait(0);
23         read(child_fd[0], buf, 4);
24         printf("%d: received %s\n", getpid(), buf);
25     }
26     exit(0);
27 }
28
```

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


```
Makefile M X C pingpong.c U
Makefile
177 .PRECIOUS: %.o
178
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\
197     $U/_pingpong\
198
```

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

```
hart 1 starting
hart 2 starting
init: starting sh
$ pingpong
4: received ping
3: received pong
$
```

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

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.4s)
```

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。
2. 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。

3.4 实验心得

- 通过这个实验，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。
- 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 (moderate)/(hard)

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

实现的源码放置在 user/primes.c。

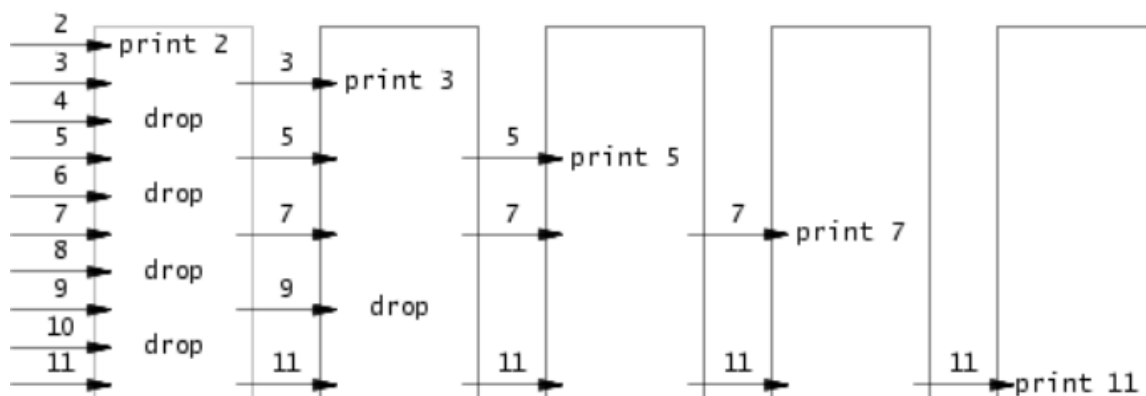
创建父进程，父进程将数字2到35输入管道，在需要时创建管道中的进程。

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。

在数据传递的过程中，父进程需要等待子进程的结束，并回收共享的资源和数据等，即一旦第一个进程到达35，它应该等待直到整个管道终止。因此，主primes进程应该在所有输出都打印完毕，并且所有其他primes进程都退出后才退出。



1. 编写 `primes.c` 的代码程序;

```
Makefile M C primes.c U X
user > C primes.c > main(int, char const *[])
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  void prime(int input_fd);
6
7  int main(int argc, char const *argv[])
8  {
9      // 定义描述符
10     int parent_fd[2];
11     // 创建管道
12     pipe(parent_fd);
13     // 创建进程
14     if (fork())
15     {
16         close(parent_fd[0]);
17         int i;
18         // 将数字 2 到 35 写入管道
19         for (i = 2; i < 36; i++)
20         {
21             write(parent_fd[1], &i, sizeof(int));
22         }
23         close(parent_fd[1]);
24     }
25     else
26     {
27         close(parent_fd[1]);
28         // 子进程调用 prime 函数处理输入
29         prime(parent_fd[0]);
30     }
31     wait(0);
32     exit(0);
33 }
```

```

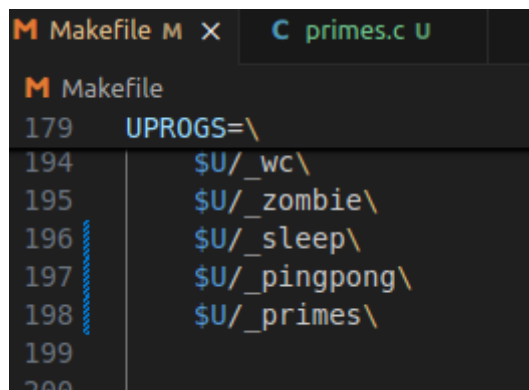
void prime(int input_fd)
{
    int base;
    //如果从管道读取的数据为空，说明已经没有数字可处理，退出函数
    if (read(input_fd, &base, sizeof(int)) == 0)
    {
        exit(0);
    }
    printf("prime %d\n", base);

    //如果还有数字可处理，创建新的子进程
    int p[2];
    pipe(p);
    if (fork() == 0)
    {
        close(p[1]);
        prime(p[0]);
    }
    else
    {
        close(p[0]);
        int n;
        int eof;
        do
        {
            eof = read(input_fd, &n, sizeof(int));
            // 如果 n 不能被 base 整除，则将 n 写入管道
            if (n % base != 0)
            {
                write(p[1], &n, sizeof(int));
            }
        } while (eof);

        close(p[1]);
    }
    wait(0);
    exit(0);
}

```

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



The screenshot shows a terminal window with two tabs: 'Makefile' and 'primes.c'. The 'Makefile' tab is active, displaying the following content:

```

179  UPROGS=\
194      $U/_wc\
195      $U/_zombie\
196      $U/_sleep\
197      $U/_pingpong\
198      $U/_primes\
199
200

```

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

```
hart 2 starting
hart 1 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：

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-util primes
make: 'kernel/kernel' is up to date.
== Test primes == primes: OK (1.7s)
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

4.3 实验中遇到的问题和解决办法

1. 问题：无法正确读取管道中的数据。
 - 解决办法：检查管道的读取端和写入端是否正确关闭，以确保数据能够正确传递。
2. 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

- 通过这个实验，实现了一个简单的质数筛选器。通过使用管道和递归调用，每个子进程将负责筛选出下一个质数，并将剩余的数字传递给下一个子进程。
- 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。因此数据如果要实现传递，则可以在 `fork()` 判定为子进程的分支上进行数据“交换”，将子变为下一级的父，从而实现了数据传递。

5 find (moderate)

5.1 实验目的

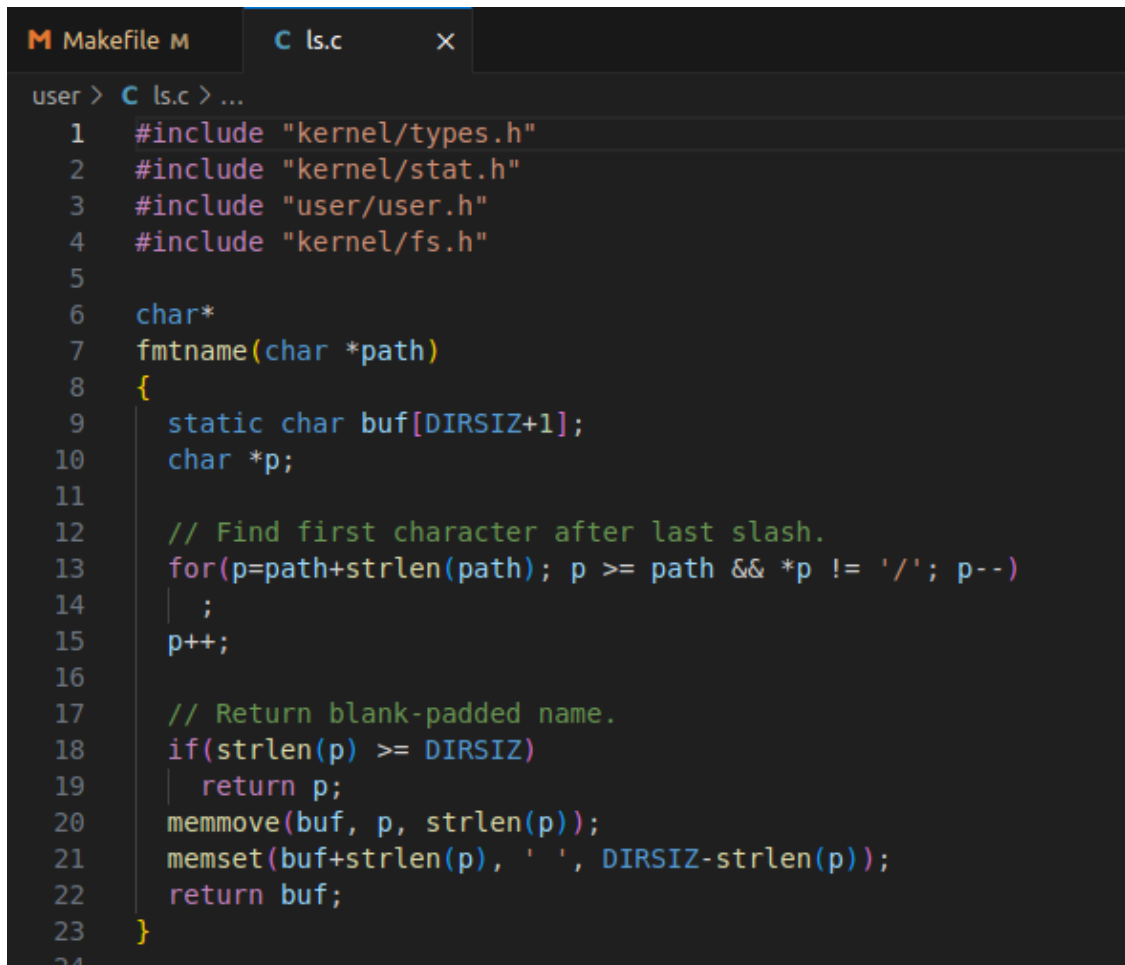
学习并编写一个简单版本的UNIX查找程序：程序应当实现查找目录树中带有特定名称的所有文件。解决方案位于文件 `user/find.c` 中。

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

实现的源码放置在 `user/find.c`。

1. 查看 `user/ls.c` 以了解如何读取目录。`user/ls.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。如果名称的长度大于等于 `DIRSIZ`，则直接返回名称。否则，将名称拷贝到一个静态字符数组 `buf` 中，并用空格填充剩余的空间，保证输出的名称长度为 `DIRSIZ`。



```
user > C ls.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), ' ', DIRSIZ-strlen(p));
22     return buf;
23 }
```

2. 编写 `find.c` 的代码程序;

```
void find(char *dir, char *file)
{
    char buf[512], *p;
    int fd;
    struct dirent de;
    struct stat st;
    // open() 函数打开路径，返回一个文件描述符，如果错误返回 -1
    if ((fd = open(dir, 0)) < 0)
    {
        // 报错，提示无法打开此路径
        fprintf(2, "find: cannot open %s\n", dir);
        return;
    }
    // int fstat(int fd, struct stat *); 系统调用 fstat 与 stat 类似，但它以文件描述符作为参数
```

```

// int stat(char *, struct stat *); stat 系统调用, 可以获得一个已存在文件的模式, 并
将此模式赋值给它的副本; 以文件名作为参数, 返回文件的 i 结点中的所有信息; 如果出错, 则返回 -1
if (fstat(fd, &st) < 0)
{
    // 出错则报错
    fprintf(2, "find: cannot stat %s\n", dir);
    // 关闭文件描述符 fd
    close(fd);
    return;
}
if (st.type != T_DIR)// 如果不是目录类型
{
    fprintf(2, "find: %s is not a directory\n", dir);
    // 关闭文件描述符 fd
    close(fd);
    return;
}
if(strlen(dir) + 1 + DIRSIZ + 1 > sizeof buf)// 如果路径过长放不入缓冲区, 则报错提
示
{
    fprintf(2, "find: directory too long\n");
    // 关闭文件描述符 fd
    close(fd);
    return;
}
strcpy(buf, dir);// 将 dir 指向的字符串即绝对路径复制到 buf
p = buf + strlen(buf);// buf 是一个绝对路径, p 是一个文件名, 通过加 "/" 前缀拼接在
buf 的后面
*p++ = '/';
while (read(fd, &de, sizeof(de)) == sizeof(de))// 读取 fd , 如果 read 返回字节
数与 de 长度相等则循环
{
    if(de.inum == 0)
        continue;
    // strcmp(s, t);根据 s 指向的字符串小于 (s<t)、等于 (s==t) 或大于 (s>t) t 指向
的字符串的不同情况分别返回负整数、0或正整数,不要递归 "." 和 ".."
    if (!strcmp(de.name, ".") || !strcmp(de.name, ".."))
        continue;
    // memmove, 把 de.name 信息复制 p, 其中 de.name 代表文件名
    memmove(p, de.name, DIRSIZ);
    if(stat(buf, &st) < 0)// 设置文件名结束符
    {
        fprintf(2, "find: cannot stat %s\n", buf);
        continue;
    }
    if (st.type == T_DIR)// 如果是目录类型, 递归查找
    {
        find(buf, file);
    }
    else if (st.type == T_FILE && !strcmp(de.name, file))// 如果是文件类型 并且
名称与要查找的文件名相同
    {
        printf("%s\n", buf);
    }
}
}

```

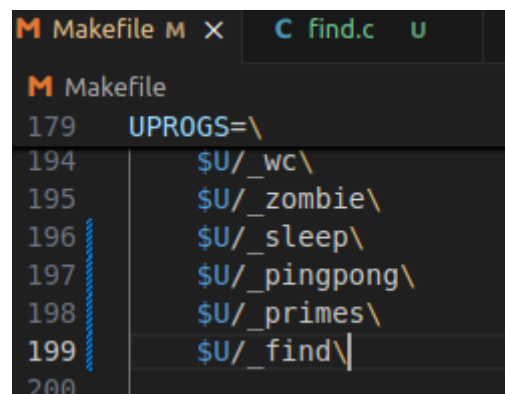


```

1
2 int
3 main(int argc, char *argv[])
4 {
5     // 如果参数个数不为 3 则报错
6     if (argc != 3)
7     {
8         // 输出提示
9         fprintf(2, "usage: find dirName fileName\n");
10        // 异常退出
11        exit(1);
12    }
13    // 调用 find 函数查找指定目录下的文件
14    find(argv[1], argv[2]);
15    // 正常退出
16    exit(0);
17 }

```

2. 将find程序添加到 Makefile 中的UPROGS中即可运行;



```

M Makefile M X C find.c U
M Makefile
179 UPROGS=\
194     $U/_wc\
195     $U/_zombie\
196     $U/_sleep\
197     $U/_pingpong\
198     $U/_primes\
199     $U/_find\
200

```

3. 保存后在终端里执行make qemu编译运行xv6;

4. 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

```

出现:

```
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
$
```

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

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.6s)
== Test find, recursive == find, recursive: OK (1.6s)
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

5.3 实验中遇到的问题和解决办法

1. 问题：打开指定路径或获取文件信息失败，出现 "cannot open" 或 "cannot stat" 错误。
解决办法：检查路径,文件名是否正确，确认存在，并确保程序有足够的权限来打开该路径。
2. 问题：路径过长，出现 "directory too long" 错误。
解决办法：可能需要缩短路径或增加缓冲区的大小。
3. 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。
- 通过这个实验，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 (moderate)

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

实现的源码放置在 `user/xargs.c`。

1. 通过示例理解xarg的工作原理：
对字符串进行处理，| 之前的结果会在缓冲流中。

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ echo hello too | xargs echo bye
bye hello too
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

2. 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'）

```
M Makefile M    C xargs.c U X
user > C xargs.c > main(int, char * [])
1  #include "kernel/types.h"
2  #include "user/user.h"
3
4  int main(int argc, char *argv[]) {
5      char inputBuf[32]; // 记录上一个命令的输入
6      char charBuf[320]; // 存储所有标记字符的缓冲区
7      char* charBufPointer = charBuf;
8      int charBufSize = 0;
9
10     char *commandToken[32]; // 使用空格(' ')分隔输入后记录的标记
11     int tokenSize = argc - 1; // 记录标记数量（初始值为argc - 1，因为xargs不会被执行）
12     int inputSize = -1;
13
14     // 首先将初始argv参数复制到commandToken
15     for(int tokenIdx=0; tokenIdx<tokenSize; tokenIdx++)
16         commandToken[tokenIdx] = argv[tokenIdx+1];
17
18
19     while((inputSize = read(0, inputBuf, sizeof(inputBuf))) > 0) {
20         for(int i = 0; i < inputSize; i++) {
21             char curChar = inputBuf[i];
22             if(curChar == '\n') { // 如果读取到'\n'，执行命令
23                 charBuf[charBufSize] = 0; // 在标记的末尾设置'\0'
24                 commandToken[tokenSize++] = charBufPointer;
25                 commandToken[tokenSize] = 0; // 在数组末尾设置空指针
26
27                 if(fork() == 0) { // 创建子进程执行命令
28                     exec(argv[1], commandToken);
29                 }
30                 wait(0);
31                 tokenSize = argc - 1; // 初始化
32                 charBufSize = 0;
33                 charBufPointer = charBuf;
34             }
35             else if(curChar == ' ') {
36                 charBuf[charBufSize++] = 0; // 标记字符串的结尾
37                 commandToken[tokenSize++] = charBufPointer;
38                 charBufPointer = charBuf + charBufSize; // 切换到新字符串的起始位置
39             }
40             else {
41                 charBuf[charBufSize++] = curChar;
42             }
43         }
44     }
45     exit(0);
46 }
```

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

```
M Makefile M X C xargs.c U
M Makefile
179 UPROGS=\
194 $U/_wc\
195 $U/_zombie\
196 $U/_sleep\
197 $U/_pingpong\
198 $U/_primes\
199 $U/_find\
200 $U/_xargs\
201
```

3. 保存后在终端里执行make qemu编译运行xv6;

4. 在命令行中输入以下命令 : xargstest.sh

```
hart 2 starting
hart 1 starting
init: starting sh
$ sh < xargstest.sh
$ $ $ $ $ $ hello
hello
hello
$ $
```

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

```
wzd@ubuntu:~/Desktop/xv6-labs-2021$ ./grade-lab-util xargs
make: 'kernel/kernel' is up to date.
== Test xargs == xargs: OK (1.4s)
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```

6.3 实验中遇到的问题和解决办法

1. 问题: make qemu编译运行xv6输入 xargstest.sh 结果不符。

解决办法: 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。

2. 问题: 输入缓冲区大小限制。

解决办法: 当前代码使用了一个固定大小的输入缓冲区 inputBuf[32]。如果输入超过32个字符, 可

能会导致数据截断。可以考虑增加缓冲区大小或者动态分配内存。

3. 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, 熟悉命令行参数的获取和处理, 包括选项解析和参数拆分。
- 学习外部命令的执行, 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。
- 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. 在终端中执行 `make grade`

```
$ make qemu-gdb
sleep, no arguments: OK (4.6s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.9s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (0.8s)
== Test pingpong ==
$ make qemu-gdb
pingpong: OK (0.6s)
== Test primes ==
$ make qemu-gdb
primes: OK (0.8s)
== Test find, in current directory ==
$ make qemu-gdb
find, in current directory: OK (1.2s)
== Test find, recursive ==
$ make qemu-gdb
find, recursive: OK (1.2s)
== Test xargs ==
$ make qemu-gdb
xargs: OK (1.4s)
== Test time ==
time: OK
Score: 100/100
wzd@ubuntu:~/Desktop/xv6-labs-2021$
```