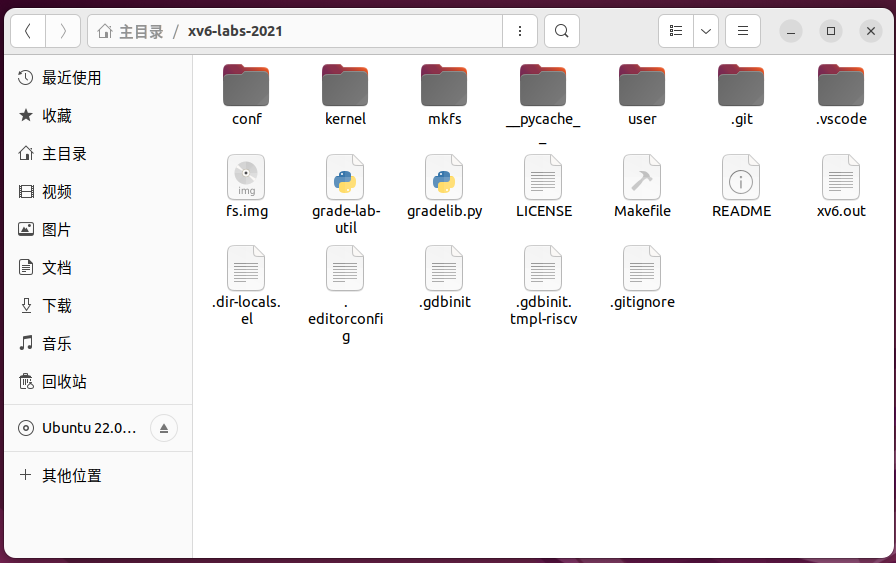
仓库链接：https://github.com/zhaoyihui233/mit-6.S081-lab-2021/tree/main/lab1%20Utilities

# 1 Boot xv6

## 1.1实验内容

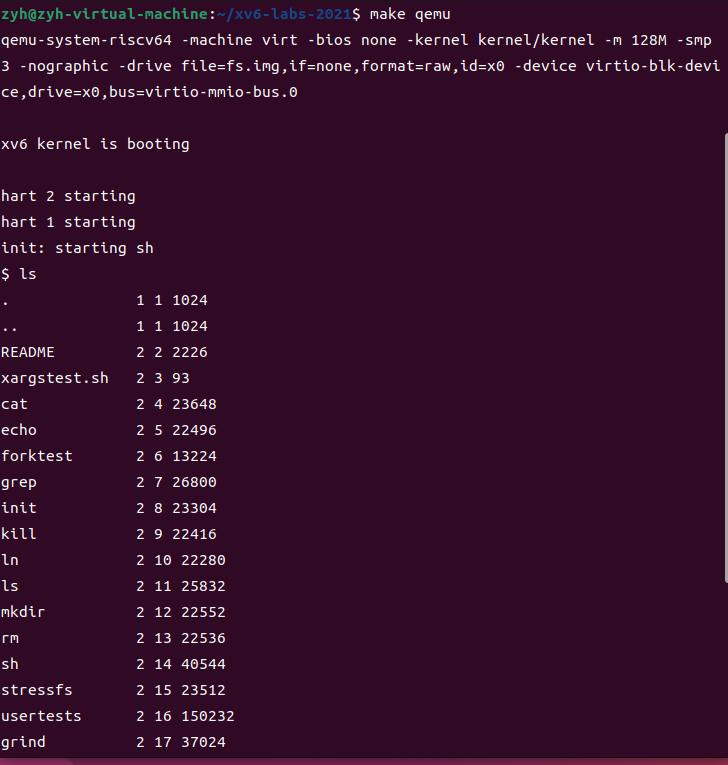
建立好实验环境，启动xv6。

## 1.2代码位置与截图



## 1.3实验结果

启动xv6，并执行ls指令。

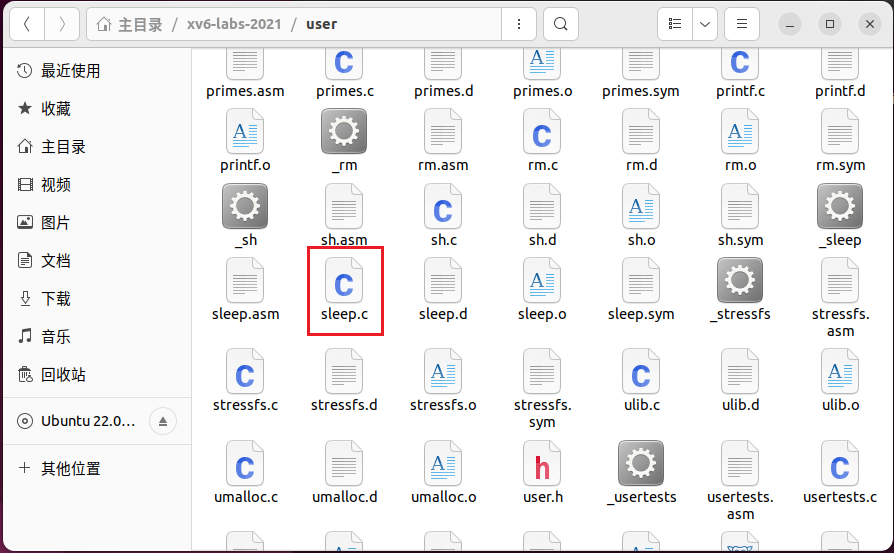


# 2 sleep

## 2.1实验内容

编写程序，实现等待用户指定个数的tick。可以通过使用sleep系统调用实现。

## 2.2代码位置与截图



#include "kernel/types.h"

#include "user/user.h"

int main(int argc, char \*argv[])

{

if (argc != 2)

{

// 把错误信息输出到标准错误流

fprintf(2, "Error: Must 1 argument for sleep\n");

exit(1);

}

int sleepNum = atoi(argv[1]);

printf("Sleep for a little while...\n");

sleep(sleepNum);

exit(0);

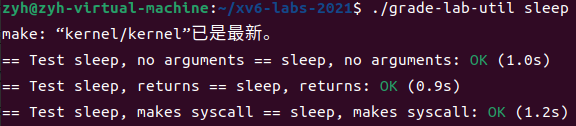
}

## 2.3实验结果

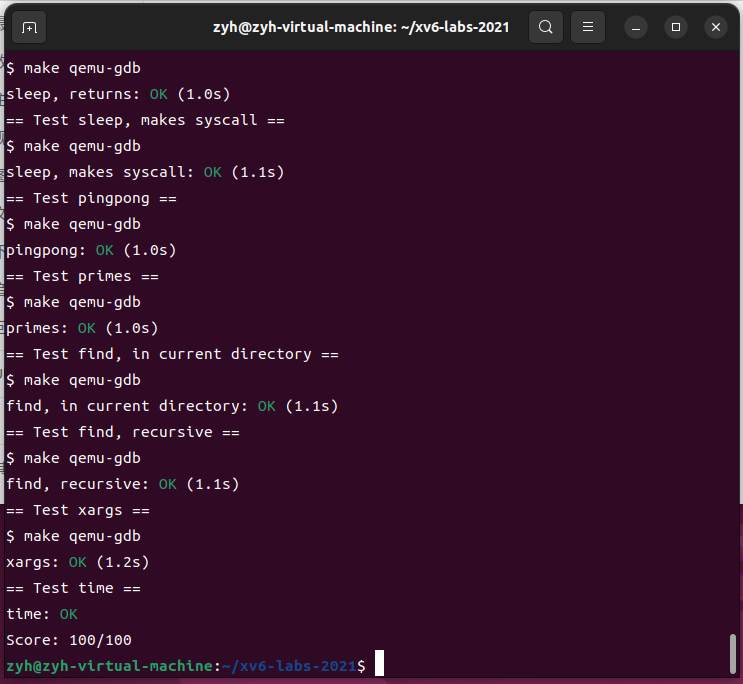
下面是运行截图：



测试结果：



make grade:



## 2.4代码&结果分析

### 2.4.1系统调用sleep

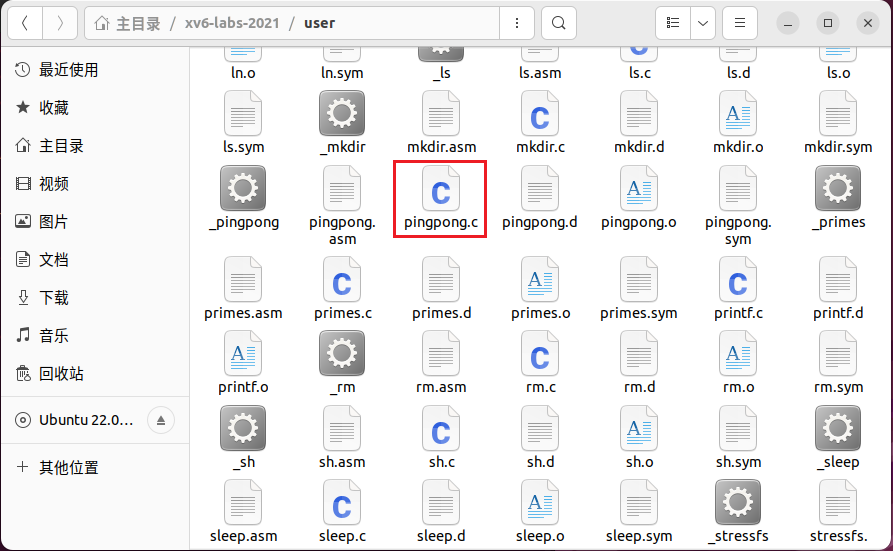
接受一个参数，程序会等待对应的时钟周期数。

# 3 pingpong

## 3.1实验内容

写一个程序，在两个进程间“ping-pong”一个字节，使用一对pipe，一个pipe对应一个数据传输的方向，另外一个pipe对应另外一个方向。

## 3.2代码位置与截图



#include <kernel/types.h>

#include <user/user.h>

int main()

{

// pipe1(p1)：写端父进程，读端子进程

// pipe2(p2)；写端子进程，读端父进程

int p1[2], p2[2];

// 来回传输的字符数组：一个字节

char buffer[] = {'X'};

// 传输字符数组的长度

long length = sizeof(buffer);

// 父进程写，子进程读的pipe

pipe(p1);

// 子进程写，父进程读的pipe

pipe(p2);

// 下面是进程分支

// 子进程

if (fork() == 0)

{

// 关掉不用的p1[1]、p2[0]

close(p1[1]);

close(p2[0]);

// 子进程从pipe1的读端，读取字符数组

if (read(p1[0], buffer, length) != length)

{

printf("Error: Child process read error!");

exit(1);

}

// 打印读取到的字符数组

printf("%d: received ping\n", getpid());

// 子进程向pipe2的写端，写入字符数组

if (write(p2[1], buffer, length) != length)

{

printf("Error: Child process write error!");

exit(1);

}

exit(0);

}

// 父进程

// 关掉不用的p1[0]、p2[1]

close(p1[0]);

close(p2[1]);

// 父进程向pipe1的写端，写入字符数组

if (write(p1[1], buffer, length) != length)

{

printf("Error: Parent process write error!");

exit(1);

}

// 父进程从pipe2的读端，读取字符数组

if (read(p2[0], buffer, length) != length)

{

printf("Error: Parent process read error!");

exit(1);

}

// 打印读取的字符数组

printf("%d: received pong\n", getpid());

// 等待进程子退出

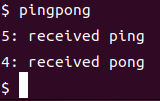
wait(0);

exit(0);

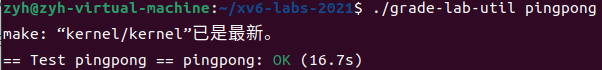
}

## 3.3实验结果

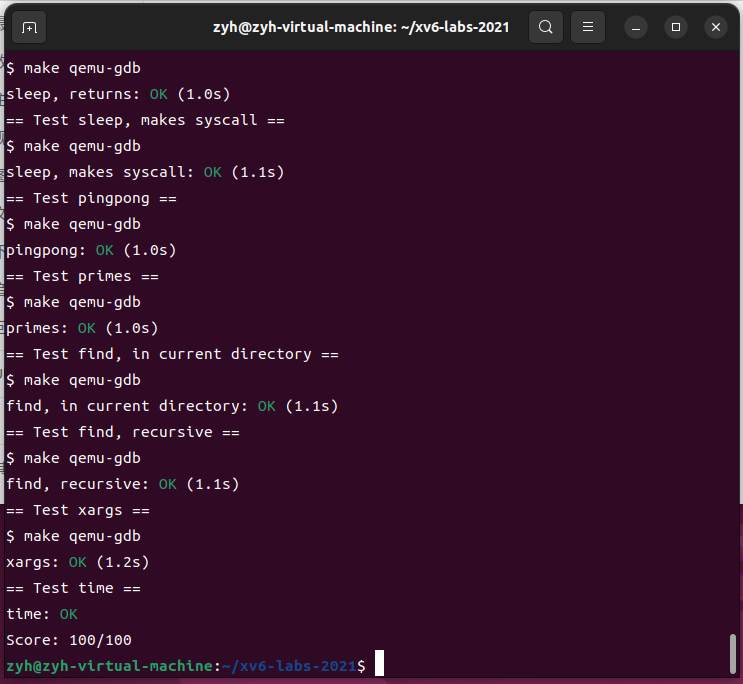
下面是运行截图：



测试结果：



make grade:



## 3.4代码&结果分析

### 3.4.1系统调用pipe

在xv6中，管道是一种用于进程间通信的机制，它可以在不同的进程之间传递数据。通过调用pipe函数，可以创建一个管道，并返回两个文件描述符（file descriptor）：一个用于读取管道数据，另一个用于写入管道数据。

在调用pipe(p1)创建管道后，p1[0]代表读取端（管道的输入），p1[1]代表写入端（管道的输出）。通常情况下，一个进程会负责向管道写入数据，而另一个进程会负责从管道读取数据。

如果一个进程仅需要从管道中读取数据，那么它应该关闭写入端的文件描述符p1[1]，即执行close(p1[1])。同样地，如果一个进程仅需要向管道中写入数据，那么它应该关闭读取端的文件描述符p1[0]，即执行close(p1[0])。

关闭不需要的文件描述符是为了确保进程在操作管道时不会产生不必要的副作用。例如，如果一个进程关闭了读取端的文件描述符p1[0]，它将无法从管道读取数据，而且对应的缓冲区也会被释放，这样就避免了不必要的资源消耗和竞争条件。

### 3.4.2系统调用read和write

管道通常会与read和write系统调用一起使用。read系统调用用于从文件描述符中读取数据，而write系统调用用于向文件描述符中写入数据。在管道的情况下，一个进程可以使用write将数据写入管道，而另一个进程可以使用read从管道中读取这些数据。

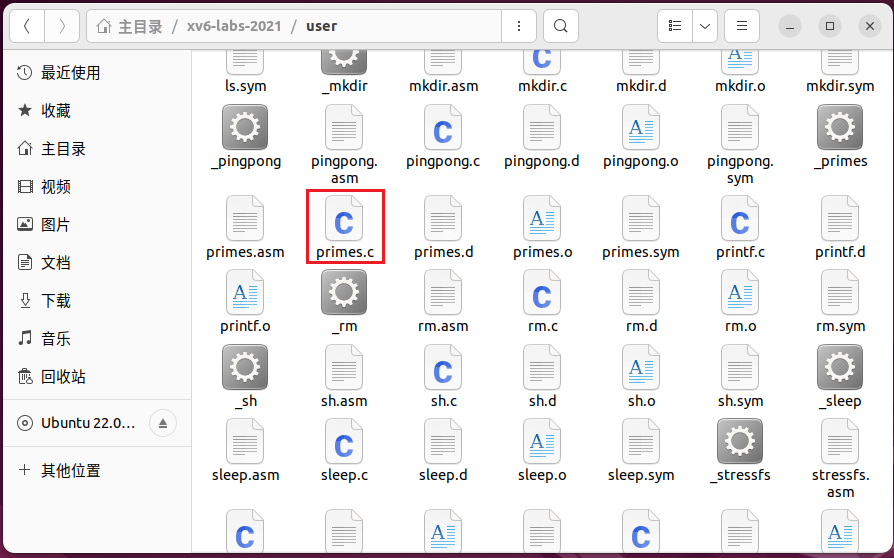
具体步骤是：在创建管道后，一个进程会拥有管道的写入端文件描述符，而另一个进程会拥有管道的读取端文件描述符。进程 A 可以使用 write 将数据写入管道的写入端文件描述符。进程 B 可以使用 read 从管道的读取端文件描述符中读取进程 A 写入的数据。通过这种方式，进程 A 和进程 B 可以通过管道进行通信，实现数据的传输。、

# 4 primes

## 4.1实验内容

使用管道（pipe）和fork来建立管道。第一个进程将数字2到35输入到管道中。对于每个质数，你需要创建一个进程，它从父进程通过一个管道读取数据，并通过另一个管道将数据写入新的子进程。

## 4.2代码位置与截图



#include "kernel/types.h"

#include "user/user.h"

// input是剩余要判断的数字的数组

// num是剩余要判断的数字的个数

void findPrimes(int \*input, int num)

{

if (num == 1)

{

// num为1说明递归到只剩一个数字了,且该数字一定是质数,输出它并停止递归

printf("prime %d\n", \*input);

return;

}

// p作为管道通信工具

int p[2];

// input一开始指向的数字一定是质数(第一次调用函数时指向2,在后续递归调用函数时也都是指向质数)

int prime = \*input;

int temp;

printf("prime %d\n", prime);

// 建立管道

pipe(p);

// 下面将建立两个子进程,一个向管道输入待筛选的数字,即input指向之后的数字;另一个进程接收数字,并进行筛选

// 第一个进程,向管道输入待筛选的数字

if (fork() == 0)

{

for (i = 0; i < num; i++)

{

temp = \*(input + i);

// 向管道输入待筛选的数字

write(p[1], (char \*)(&temp), 4);

}

exit(0);

}

close(p[1]);

// 第二个进程,接收第一个进程传送的数据,并筛选,筛去\*input(即prime)的倍数,它们肯定不是质数.

// 没被筛去的数字中,最小的肯定是质数

if (fork() == 0)

{

int counter = 0;

char buffer[4];

while (read(p[0], buffer, 4) != 0)

{

temp = \*((int \*)buffer);

if (temp % prime != 0)

{

\*input = temp;

input += 1;

counter++;

}

}

// input - counter意为让函数第一个参数指向没被筛去的最小的数字,它肯定是整数

findPrimes(input - counter, counter);

exit(0);

}

wait(0);

wait(0);

}

int main()

{

// input是存放2-35，共34个数字的数组

int input[34];

int i = 0;

for (; i < 34; i++)

{

input[i] = i + 2;

}

// 调用寻找质数的函数

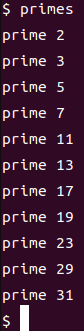
findPrimes(input, 34);

exit(0);

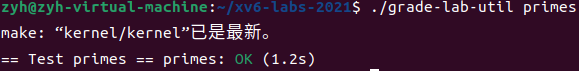
}

## 4.3实验结果

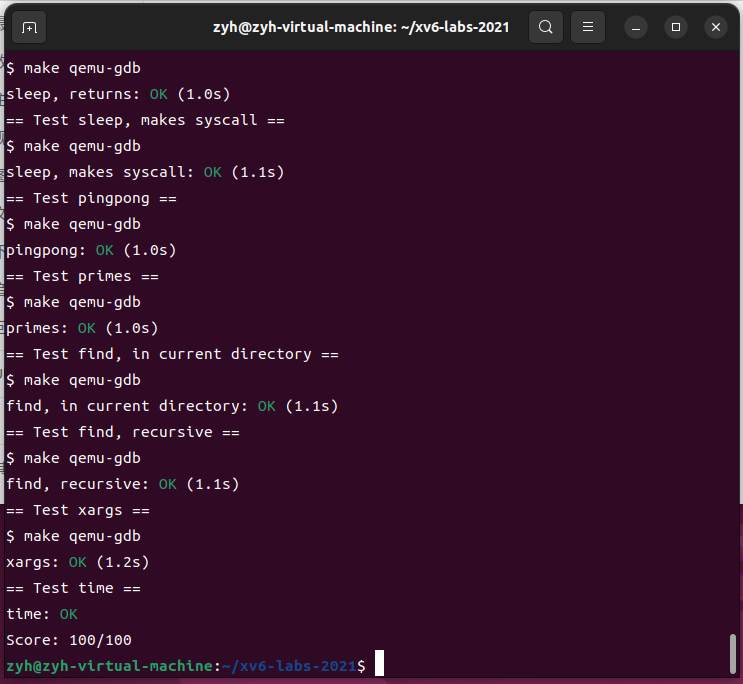
下面是运行截图：



测试结果：



make grade:



## 4.4代码&结果分析

### 4.4.1代码思路

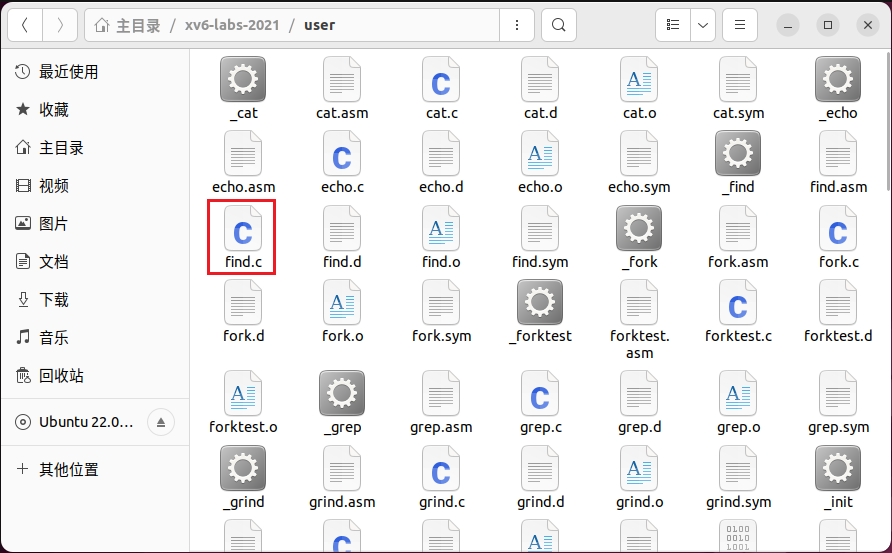
在一个递归函数中，每一次递归都创建两个子进程，一个子进程向管道里传递剩余要筛选的数字，另一个进程通过管道接收这些数字，并筛选数字，筛选方式就是使用挨个检查这些数字是否能被其中最小的数字整除，若能说明该数字一定不是素数；若不能，筛选后的数字也不一定是素数，还需要用筛选后的数字递归调用函数。

# 5 find

## 5.1实验内容

编写程序，实现在指定目录的文件树下查找指定名字的文件。

## 5.2代码位置与截图



#include "kernel/types.h"

#include "kernel/fcntl.h"

#include "kernel/stat.h"

#include "kernel/fs.h"

#include "user/user.h"

/\*

将路径格式化为文件名,即将路径中最后一个\号后的字符串提取出来

\*/

char \*pathToFilename(char \*path)

{

static char buf[DIRSIZ + 1];

char \*p;

// 将路径中最后一个\号后的字符串提取出来

for (p = path + strlen(path); p >= path && \*p != '/'; p--)

;

p++;

memmove(buf, p, strlen(p) + 1);

return buf;

}

/\*

在某路径中查找某文件,主干函数

\*/

void find(char \*path, char \*findName)

{

int fd;

struct stat st;

// open指定的文件(夹),同时检查指定路径是否存在

if ((fd = open(path, O\_RDONLY)) < 0)

{

fprintf(2, "Error: cannot open %s\n", path);

return;

}

// 系统调用fstat用于获取文件(夹)信息,并存入stat结构体

if (fstat(fd, &st) < 0) // 返回-1说明获取信息失败

{

fprintf(2, "Error: cannot stat %s\n", path);

close(fd);

return;

}

// 根据path打开的文件的类型,执行不同的策略

char buf[512], \*p;

struct dirent de;

switch (st.type)

{

case T\_FILE: // 文件类型是文件

if (strcmp(pathToFilename(path), findName) == 0)

{

printf("%s\n", path);

}

break;

case T\_DIR: // 文件类型是文件夹

if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf)

{

printf("Error: path too long\n");

break;

}

strcpy(buf, path);

p = buf + strlen(buf);

\*p++ = '/';

while (read(fd, &de, sizeof(de)) == sizeof(de))

{

// printf("de.name:%s, de.inum:%d\n", de.name, de.inum);

if (de.inum == 0 || de.inum == 1 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0)

continue;

memmove(p, de.name, strlen(de.name));

p[strlen(de.name)] = 0;

// 由于遍历到的是文件夹,所以其内要调用find函数遍历

find(buf, findName);

}

break;

}

close(fd);

}

int main(int argc, char \*argv[])

{

// 判断输入是否正确

if (argc < 3)

{

printf("Error: Must 3 argements! Input format: find <path> <fileName>\n");

exit(0);

}

// 调用find函数

find(argv[1], argv[2]);

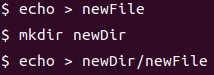
exit(0);

}

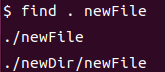
## 5.3实验结果

下面是运行截图：

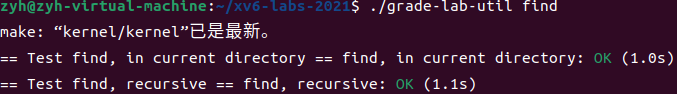
先在根目录下创建一个名叫newFile的文件和一个名叫newDir的文件夹，再在newDir里创建一个名叫newFile的文件。



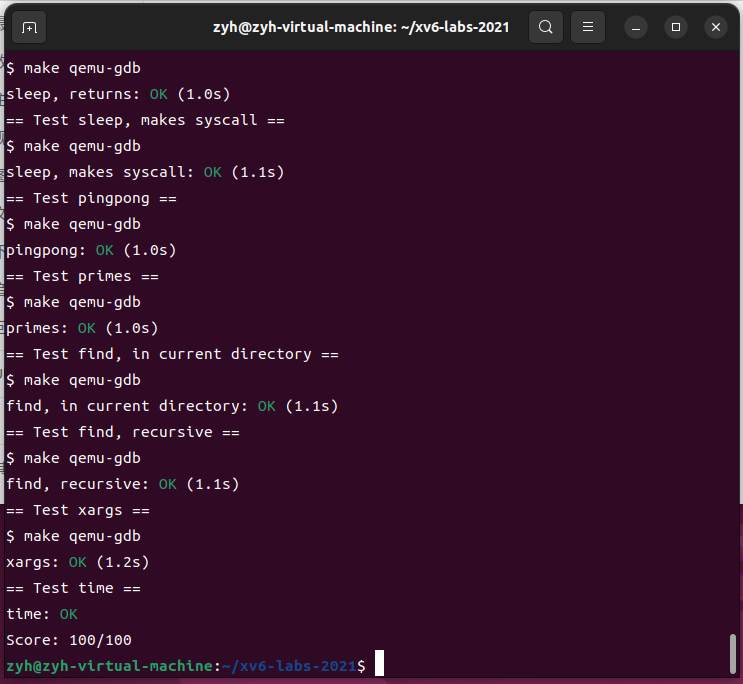
然后再使用find从根目录查找newFile：



测试结果：



make grade:



## 5.4代码&结果分析

### 5.4.1结构体stat

stat是用于存储文件或文件夹信息的结构体。在代码中，通过调用 fstat函数将文件的信息存储在stat结构体变量中。

### 5.4.2结构体dirent

使用与目录项相关的结构体dirent来读取目录内容。在代码中的 read(fd, &de, sizeof(de))中，de就是一个 struct dirent 结构体变量，用于存储读取到的目录项数据。

### 5.4.3代码思路

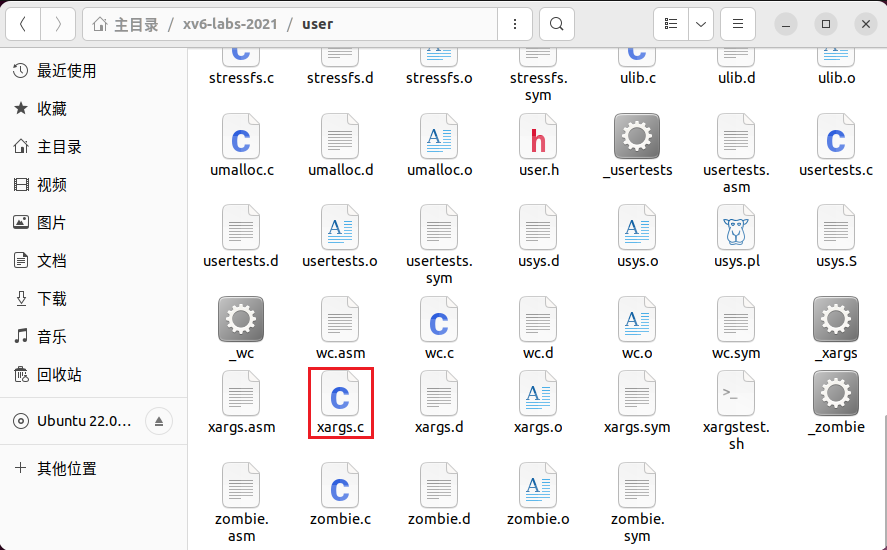
要在指定目录及其目录树中查找某个名字的文件，可以借助递归的思想解决。对于初始目录，先判断该目录是否存在，若不存在说明用户输入错误。若存在，则根据目录对应得文件的类型分别做出动作：如果是一个文件，那么比较文件名与查找名，若相同则输出目录；如果是一个文件夹，则对于文件夹里每一个文件，都需要调用递归函数。当遍历完初始目录树的所有文件后，算法结束。

# 6 xargs

## 6.1实验内容

实现类似unix xargs类似功能，比如echo hello too | xargs echo bye，要输出bye hello too。即等价于echo bye hello too，将上个命令输出的每行作为参数，拼接到xargs后面的指令后面。echo hello too输出为hello too，将其拼接到echo bye后面，就是echo bye hello too。

## 6.2代码位置与截图



#include "kernel/types.h"

#include "user/user.h"

#include "kernel/param.h"

int main(int argc, char \*argv[])

{

int i = 0;

char \*lineSplit[MAXARG]; // lineSplit作为给exec执行argv[1]的参数

for (int j = 1; j < argc; j++) // 从j=1开始,因为argv[0]=xargs,argv[1]才是要执行的命令

{

lineSplit[i++] = argv[j];

}

// 下面将标准输入处理后接到lineSplit

int m = 0, k;

char block[32]; // 标准输入读取到block里

char buf[32]; // 将block的内容先存入buf

char \*p = buf; // 在给lineSplit赋值时，p总指向一个完整的参数

while ((k = read(0, block, sizeof(block))) > 0) // 循环处理能处理长输入而正常运行,只不过输出可能会出乎意料

{

for (int l = 0; l < k; l++)

{

// 读取到回车,说明标准输入已全部读取

if (block[l] == '\n')

{

buf[m] = 0;

m = 0;

lineSplit[i++] = p;

p = buf;

lineSplit[i] = 0; // 最后要加上\0

i = argc - 1;

if (fork() == 0) // 子进程调用exec执行argv[1]命令

{

exec(argv[1], lineSplit);

}

wait(0);

}

else if (block[l] == ' ')

{

buf[m++] = 0; // buf[m]设为0,对应的block[m]为空格

lineSplit[i++] = p;

p = &buf[m];

}

else

{

buf[m++] = block[l]; // 将block赋值到buf里

}

}

}

exit(0);

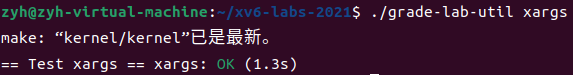
}

## 6.3实验结果

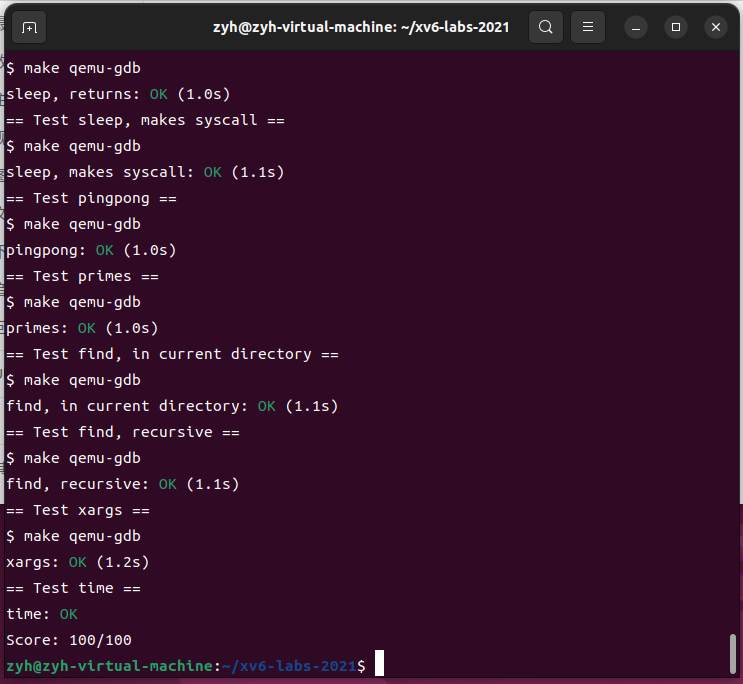
下面是运行截图：



测试结果：



make grade:



## 6.4代码&结果分析

### 6.4.1代码思路

该问题有两个要点：怎么获取完整输入，怎么执行后一个命令。

对于第一个问题：例如，对于echo hello | xargs echo bye，在xargs.c的代码中，argc（参数个数）为3，argv[0]=xargs，xargs[1]=echo，xargs[2]=bye，这样可以获得部分输入。那么怎么获取前面的hello呢？根据管道操作符的意义，可以通过read文件描述符0来获取hello。

对于第二个问题，可以通过exec系统调用实现。为了更符合命令行输入，我做了如下转换：假如命令为echo hello too | xargs echo bye，那么从文件描述符0获取到的是“hello too”，给exec执行的字符串数组要转化成{“echo”,“bye”,“hello”,“too”,“\0”}而不是直接使用{“echo”,“bye”,“hello too” ,“\0”}。