华东师范大学数据学院实验报告

课程名称： 操作系统	年级：2022级	上机实践成绩：
指导教师：翁楚良	姓名：刘蔚璁	
上机实践名称：	学号：10225501443	上机实践日期：2024.3.9-2024.3.10

一、实验目的

熟悉如何在xv6环境下写代码

二、实验内容

完成sleep、pingpong、primes、find、xargs五个作业

三、实验环境

VSCode

四、实验过程及结果

1.sleep

1.1.实验目的

为 xv6实现 UNIX 程序 sleep， sleep 应该暂停用户指定的滴答数。

1.2.实验过程

1.2.1.查看user/中的其他程序，了解如何获取传递给程序的命令行参数

以已有的echo.c为例，echo是一个常见的命令，用于在终端输出用户输入的内容。可以通过此函数来了解命令行传参。

```
int
main(int argc, char *argv[])
{
  int i;

  for(i = 1; i < argc; i++){
    write(1, argv[i], strlen(argv[i]));
    if(i + 1 < argc){
      write(1, " ", 1);
    } else {
      write(1, "\n", 1);
    }
  }
  exit(0);
}
```

可知argc代表传入参数的数量，argv[]是指针数组，用于存储每个命令行参数的字符串。

1.2.2.了解sleep命令及其实现流程

sleep 是一个命令行工具，也是一个系统调用，用于在一段时间内使程序休眠或等待。
首先在user/user.h找到定义接口名为sleep的系统调用。

```
int sleep(int);
```

接着到user/usys.s查看相关汇编代码，可知从用户态跳转到内核态实现sleep系统调用的函数是sys_sleep()。

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

最后到kernel/sysproc.c查看sleep的实现代码。

```
uint64
sys_sleep(void)
{
int n;
uint ticks0;

    //从用户空间获取第一个整数参数
    if (argint(0, &n) < 0)
        return -1;

    //获取全局锁，用于保护系统时钟 ticks 变量的访问
    acquire(&tickslock);
    ticks0 = ticks;

    // 在系统时钟 ticks 增加到 n 之前，循环等待
    while (ticks - ticks0 < n) {
        // 如果当前进程被杀死，释放锁并返回错误
        if (myproc()->killed) {
            release(&tickslock);
            return -1;
        }
        //注意！这里的sleep不是指int sleep(int),是同名函数
        // 调用 sleep 函数，将当前进程置为睡眠状态
        sleep(&ticks, &tickslock);
    }

    // 循环结束后，释放全局锁
    release(&tickslock);

    // 返回 0 表示 sleep 调用成功
    return 0;
}
```

通过参阅以上代码，我对xv6的系统调用实现流程有了一定了解，可以开始尝试代码编写了。

1.2.3.编辑sleep.c

首先参考echo.c，引入三个头文件。

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

sleep.c要求实现：

(1)若用户忘记传递参数，打印一条错误信息；

(2)命令行参数以字符串形式传递，使用atoi()将其转换为整数；

(3)使系统调用sleep()；

(4)确保main调用exit()以退出程序。

因为sleep.c的要求较少，只需编写main函数就可实现，具体内容见代码及注释。

```
int main(int argc,char *argv[])
{
    /*如果命令行参数小于2，打印错误信息*/
    if(argc<2){
        fprintf(2, "Usage: sleep number\n");
        exit(1);
    }

    /*把传入的第二个参数通过atoi函数转化为整数*/
    int number = atoi(argv[1]);

    /*调用sleep函数，使程序休眠number秒*/
    sleep(number);
    exit(0);
}
```

1.2.4.编译并测试

按照文档提示修改Makefile文档，并输入`make qemu`编译程序，然后输入`./grade-lab-util sleep`进行测试，如图，测试通过。

```
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util sleep
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (4.8s)
== Test sleep, returns == sleep, returns: OK (5.4s)
== Test sleep, makes syscall == sleep, makes syscall: OK (4.7s)
```

2.pingpong

2.1.实验目的

使用 UNIX 系统调用通过一对 pipe （每个方向一个）在两个进程之间“pingpong”一个字节。

2.2.实验过程

2.2.1.了解pingpong及其实现流程

在操作系统中，“pingpong”通常是指两个或多个进程之间的交互模式，其中一个进程发送消息（“ping”），而另一个进程接收并回应该消息（“pong”）。这种交互模式经常用于说明并发编程或进程间通信的概念。

实现pingpong需要以下几种系统调用：

- fork：用于创建一个子进程作为调用进程的副本,子进程的PID为0。
- pipe：用于创建管道。
  - 管道是一种在Unix/Linux系统中用于进程间通信的机制，它允许一个进程的输出直接成为另一个进程的输入。管道通常用于在两个相关的进程之间传递数据。`管道分为 读出端 和 写入端 两个部分`，进程可以向写端写入数据，也可以从读端读出数据。使用 pipe 系统调用时，内核会创建一个管道，并返回两个文件描述符，一个用于读取端（p[0]），一个用于写入端(p[1])。
- read：用于从文件描述符（包括管道）中读取数据。
  - `read(fd, buffer, sizeof(buffer))`是read系统调用的常见形式，fd 是文件描述符，指定要读取的文件、管道、套接字等；buf 是用于存储读取数据的缓冲区的指针；count 是要读取的字节数。write同理。
- write：用于向文件描述符（包括管道）中写入数据。
- getpid：用于获取当前进程的进程ID，从而在输出中区分父子进程。

另外还需要对文件描述符有一定了解。文件描述符是在Unix和类Unix操作系统中用于标识已打开文件或I/O通道的整数。0、1、2分别用于标准输入、标准输出和标准错误。

- `文件描述符 0（stdin）`是标准输入的文件描述符。它通常与键盘输入相关联，用于从用户获取输入。
  eg.read(0, buffer, sizeof(buffer)); 从标准输入读取数据。
- `文件描述符 1（stdout）`是标准输出的文件描述符。它通常与屏幕输出相关联，用于向用户输出信息。
  eg.write(1, "Hello, World!\n", 13); 向标准输出写入数据。
- `文件描述符 2（stderr）`是标准错误输出的文件描述符。它通常与屏幕输出相关联，用于向用户输出错误信息。
  eg.fprintf(stderr, "Error: Something went wrong.\n"); 向标准错误输出写入错误信息。

对这些信息有了一定的了解后，就可以开始编写pingpong.c了。

2.2.2.编辑pingpong.c

pingpong.c要求：

(1)父进程应向子进程发送一个字节；子进程应打印“ ：收到 ping”，其中 是其进程 ID，将管道上的字节写入到父进程，然后退出；

(2)父进程应该从子进程读取字节，打印“ ：收到 pong”，然后退出。

这里有两种实现思路，分别是单管道实现和双管道实现。

- 单管道实现
  - 只设置一个管道，父进程实现“ping”的写入和“pong”的读取，子进程实现“ping”的读取和“pong”的写入。
  - 因为两个字符串共用一个管道，可能会出现冲突，所以应该调用函数`wait()`,这个函数可以是父进程暂停，直到子进程终止后再继续父进程。

```c
int main(int argc,char* argv[]){
  //创建一个管道，实现ping、pong的读写
  //创建成功后，p[0]为管道读取端，p[1]为管道写入端
  int p[2];
  pipe(p);

  char readtext[10];//存储父进程和子进程的读出内容

  //创建子进程
  //在父进程中folk返回子进程的pid，在子进程中folk返回0
  int pid = fork();

  if(pid==0){
      read(p[0],readtext,10);
      printf("%d: received %s\n",getpid(),readtext);
      write(p[1],"pong",10);
      exit(0);//子进程一定要退出
  }

  else{
      write(p[1],"ping",10);
      //父进程阻塞，等待子进程读取
      wait(0);
      read(p[0],readtext,10);
      printf("%d: received %s\n",getpid(),readtext);
  }

  exit(0);
}
```

    - 在实验的时候我有一个疑问：父子进程是同时运行的，若子进程执行read的时候父进程的write还没有结束，如何保证子进程能从管道中读取到内容呢?
      查阅资料发现，read 操作在管道中没有数据可读时，会阻塞直到满足以下条件之一：
        - 管道中有数据可读
        - 管道的写入端被关闭
- 双管道实现
    - 设置两个管道，分别实现两个字符串的读取和输入

```c
int main(int argc, char *argv[]) {
  // 定义两个整型数组，用于存放管道的文件描述符
  int p1[2];  //ping
  int p2[2];  //pong

  // 创建两个管道
  pipe(p1);
  pipe(p2);

  // 创建子进程
  int pid = fork();

  if (pid == 0) {
      // 关闭不需要的文件描述符
      close(p1[1]);   //ping的写入端
      close(p2[0]);   //pong的读取段

      // 存放子进程接收到的数据
      char son[3];
      read(p1[0], son, 4);
      close(p1[0]);

      // 打印子进程接收到的信息
      printf("%d: received %s\n",getpid(),son);

      write(p2[1], "pong", 4);
      close(p2[1]);
  }

  else if (pid > 0) {

      close(p1[0]);   //ping的读取端
      close(p2[1]);   //pong的写入端

      write(p1[1], "ping", 4);
      close(p1[1]);

      // 存放父进程接收到的数据
      char father[3];
      read(p2[0], father, 4);
      close(p2[0]);

      // 打印父进程接收到的信息
      printf("%d: received %s\n",getpid(),father);
  }
```

- 使用`close()`及时关闭无用的管道端口可以避免阻塞和资源泄露。在管道中，一旦一个端口（读取端或写入端）被关闭，就无法再次显式地打开它，因为相关的资源已经被释放。

2.2.3.编译并测试

我对以上两份代码进行了测试，结果均相同，如图。

```
$ pingpong
4: received ping
3: received pong
$ QEMU: Terminated
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util pingpong
make: “kernel/kernel”已是最新。
== Test pingpong == pingpong: OK (2.4s)
```

3.primes

3.1.实验目的

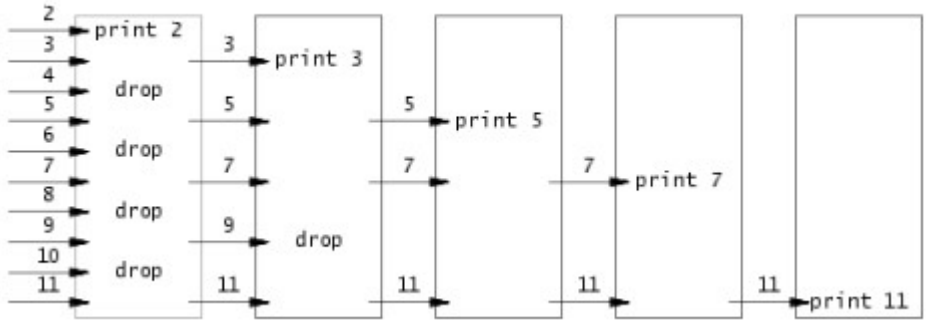
使用 pipe 编写素数筛的并发版本。

3.2.实验过程

3.2.1.了解primes及其实现流程

primes是一个质数筛选程序它的功能是从标准输入读取一系列数字，然后通过筛选算法找出其中的质数，并将这些质数输出到标准输出，在本实验中只需要实现筛选打印出2-35之间的素数。

实验思路



上图是实验思路，以数字2-11为例。

- 读取管道中的第一个数，这个数一定是质数，直接打印至标准输出。
- 继续读取管道中剩余的数字，对每个数字进行判断：
  - 如果当前数字能被当前的质数（取出的第一个数）整除，则说明它不是质数，直接忽略。
  - 如果当前数字不能被当前的质数整除，则将它保留到下一个管道；
- 这个过程持续直到管道内没有数据。

因为要编写并发版本，而父子进程是独立的，所以要实现父子进程之间的通信和数据传输就需要重定向，通过函数dup来实现，查看`user.c`发现xv6有提供dup函数。

- `dup()`
    - dup函数是一个系统调用，用于复制文件描述符。它创建了一个新的文件描述符，指向与原始文件描述符相同的文件。这样，就可以通过两个文件描述符来访问相同的文件，实现文件描述符的复制。
    - 如果没有指定具体的文件描述符，dup函数会将指定的文件描述符复制到系统中当前可用的最小文件描述符上（标准输入、标准输出或标准错误）。

另外，要完成本题还需要补充一点管道的相关知识。

- 在Unix-like系统中，管道是一个先进先出（FIFO）的数据结构，一旦数据被读取，它就会从管道中移除，不会保留在管道中。读取操作会导致管道中的数据被消费掉，所以读取过的数据不会再次出现在后续的读取操作中。因此，一旦数据被读取，它就不会再保留在管道中了。

3.2.2.编辑primes.c

通过观察实现过程可以发现在每个管道中筛选质数是比较固定的流程，所以可以用递归来实现。

- 函数mapping
  编写一个重定向函数，这个函数的功能是把新创建的管道的读端或写端重定向到标准输入和标准输出，使父子进程能共享同一个管道中的数据。

```
  void mapping(int n, int fd[])
{
    //标准输出：文件描述符为1；标准输入：文件描述符为0
    close(n);
    //dup函数会将指定的文件描述符复制到系统中当前可用的最小文件描述符上
    dup(fd[n]);
    close(fd[0]);
    close(fd[1]);
}
```

- 函数primes
  这是一个递归函数，对于每个管道，因为管道的第一个数一定是质数，所以直接打印，而剩下的数则进行判断，若不能被第一个数整除则加入新的管道留给下次递归判断，若可以被整除则直接忽略。

```
void primes()
{
    int previousP,nextP;
    //如果可以从管道里读到数据
    if(read(0,&previousP,sizeof(int)))
    {
        //管道里的第一个数字一定是素数，直接打印出来
        printf("prime %d\n",previousP);

        //创建新的管道
        int fd[2];
        pipe(fd);

        //创建子进程
        if(fork() == 0)
        {
            //把管道的写入端重定向到标准输出（输出到终端->写入管道）
            mapping(1,fd);
            //如果可以从管道中读到数据
            while (read(0,&nextP,sizeof(int)))
            {
                //且读到的数据不能被第一个读出的数整除
                if(nextP % previousP != 0)
                {
                    //把这个数字写入新创建的管道里
                    //注意！虽然读数据写数据的文件标识符是0和1，但是两个不同的管道
                    write(1,&nextP,sizeof(int));
                }
            }

        }
        else
        {
            //子进程结束后执行
            wait(0);
            //把管道的读取端重定向到标准输入（从终端读取->从管道读取）
            mapping(0,fd);
            //递归
            primes();
        }
    }
}
```

- 主函数
  主函数要做的就是把2-35写入管道，然后调用递归函数，具体如下。

```c
int main(int argc,int *argv[])
{
    //创建管道
    int fd[2];
    pipe(fd);

    //创建子进程
    if(fork() == 0)
    {
        //把管道的写入端重定向到标准输出（输出到终端->写入管道）
        mapping(1,fd);
        for(int i = 2; i <= 35; i++)
        {
            //把数字2-35写入管道中
            write(1,&i,sizeof(int));
        }
    }
    else
    {
        //子进程结束后再执行父进程
        wait(0);
        //把管道的读取端重定向到标准输入（从终端读取->从管道读取）
        mapping(0,fd);
        //调用递归函数
        primes();
    }
    exit(0);
}
```

需要注意的是，因为父进程和子进程的功能需要有前后之分，所以应该使用wait()函数保证父进程在子进程运行完之后运行。

3.2.3.编译并测试

`make qemu` + `./grade-lab-util primes`

```
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util primes
make: “kernel/kernel”已是最新。
== Test primes == primes: OK (3.5s)
```

4.find

4.1.实验目的

编写一个简单版本的 UNIX 查找程序：查找目录树中具有特定名称的所有文件。

4.2.实验过程

4.2.1.了解find及其实现流程

find是一个用于在操作系统中查找文件或目录的命令行工具或命令。它通常用于在文件系统中执行搜索操作，以找到匹配指定条件的文件或目录。在这里我们要实现的是指定的目录及其子目录中搜索符合条件的文件或目录，并打印出它们的路径。
终端输入的格式为 `find [路径] [目标文件名/目录名]` ，可能有以下几种情况：

- 路径的最后一项即是所搜索的内容
- 目标内容在所给路径下
- 目标内容在所给路径的子目录下

ls指令是输出目录下的内容，所以根据提示先阅读 `ls.c` 的代码，收获如下：

- `ls.c` 设置了函数 `fmtname(char *path)` ，该函数的作用是从路径中提取文件名（即从字符串 "a/b/c/d" 中获取字符串 "d"），该函数可以直接在find中复用
- 存放目录底下的文件名或者子目录信息的结构体（在 `kernel/fs.h` 中查看）

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

- 存放文件的元数据信息（文件的基本属性，包括文件所在文件系统、文件的类型、大小等信息）的结构体（在 `kernel/stat.h` 中查看）

```
struct stat {
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short type;  // Type of file
  short nlink; // Number of links to file
  uint64 size; // Size of file in bytes
};
```

- 可以用 `fd = open(path, 0)` 打开指定路径的文件，并返回一个文件描述符
- 可以用 `fstat(fd, &st)` 获取一个已打开文件的状态信息，并将其存储stat中
- 可以用switch判断已打开文件的类型并分别操作

```
switch(st.type){
//该文件为文件时
case T_FILE:
  [具体操作]
//该文件为文件夹时
case T_DIR:
  [具体操作]
}
```

根据 `ls.c` 中收获的工具，就可以梳理find的流程了。
传入的三个参数分别是函数名（find）、路径、目标文件。首先打开该路径的文件并获取其状态信息，

- 如果这个路径的终点就是一个文件，那么判断这个文件和目标文件（第三个参数）是否相同，如果相同则直接打印该路径，查询结束。
- 如果这个路径的终点是一个目录（或称文件夹），那就循环读取这个目录下的每个文件/子目录，把每一个文件分别加入路径中，然后递归调用find，这样就能实现层层深入查找子目录中的文件。

4.2.2.编辑find.c

首先可以对ls.c中从路径提取文件名的函数进行复用。

```
//从字符串"a/b/c/d"中获取字符串"d",基本原理为从后往前找到第一个'/'
char* fmtname(char *path)
{
  //DIRSIZ:文件名的最大长度
  static char buf[DIRSIZ+1];
  char *p;

  //从最后一个字符往前循环，直到遇到'/'
  for(p=path+strlen(path); p >= path && *p != '/'; p--);
  p++;

  //疑问：为什么要分情况返回p或者缓冲区指针？（经测试直接返回p也可以）
  // Return blank-padded name.
  if(strlen(p) >= DIRSIZ)
    return p;
  //将字符串从指针p指向的位置复制到缓冲区buf中
  memmove(buf, p, strlen(p));
  buf[strlen(p)] = '\0';

  return buf;
}
```

接着是给定一个路径和一个文件名，比较路径终点的文件名和所给文件名是否相同的函数。

```
void cmp(char *a, char *b)
{
    //strcmp 函数会逐个比较两个字符串的字符，
    //如果在某个位置上两个字符不相同，则返回它们的ASCII差值；
    //如果两个字符串完全相同，则返回0。
    if(!strcmp(fmtname(a), b))
        printf("%s\n", a);
}
```

接着就是主体find函数，具体操作见代码与注释。

```
void
find(char *path, char *target)
{
  //缓冲区buf用来储存路径
  char buf[512], *p;
  //文件描述符
  int fd;
  //存的是目录底下的文件名或者子目录信息
  struct dirent de;
  //文件的元数据信息（文件的基本属性，包括文件所在文件系统、文件的类型、大小等信息）
  struct stat st;

  //打开指定路径终点的文件，并返回一个文件描述符。0：只读
  if((fd = open(path, 0)) < 0){
    fprintf(2, "find: cannot open %s\n", path);
    return;
  }

  //获取一个已打开文件的状态信息，并将其存储stat中
  if(fstat(fd, &st) < 0){
    fprintf(2, "find: cannot stat %s\n", path);
    close(fd);
    return;
  }

  switch(st.type){
    //该文件为文件时，若匹配成功，结束
  case T_FILE:
    cmp(path, target);
    break;

    //该文件为文件夹时
  case T_DIR:

    //如果路径过长
    if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
      printf("find: path too long\n");
      break;
    }

    //把当前路径写到缓冲区中
    strcpy(buf, path);
    //指针指向路径末尾
    p = buf+strlen(buf);
    *p++ = '/';
```

```
    //通过循环从已经打开的目录文件中读取目录条目
    //de存的是目录底下的文件名或者子目录信息
    while(read(fd, &de, sizeof(de)) == sizeof(de)){

      //若文件夹下文件数量为0，1或者该文件夹名字为"."或".."则不进入，防止套娃
      /*de.inum 是目录项的 inode 号。
      在 Unix 系统中，inode 号为 0 表示这个目录项未被使用，inode 号为 1 表示这个目录的父目录。*/
      /*de.name 是目录项的名称。
      .（点）代表当前目录，..（点点）代表父目录。*/
      if(de.inum == 0 || de.inum == 1 || strcmp(de.name, ".")==0 || strcmp(de.name, "..")==0)
        continue;

      //将文件名追加到路径，递归find
      memmove(p, de.name, strlen(de.name));
      p[strlen(de.name)] = '\0';
      find(buf, target);
    }
    break;
  }
  close(fd);
```

而主函数要实现的则是简单地调用find函数。

4.2.3.编译并测试

```
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util find
make: “kernel/kernel”已是最新。
== Test find, in current directory == find, in current directory: OK (4.3s)
== Test find, recursive == find, recursive: OK (4.2s)
```

5.xargs

5.1.实验目的

编写一个简单版本的 UNIX xargs 程序：从标准输入读取行并为每行运行一个命令，将该行作为参数提供给命令。

5.2.实验过程

5.2.1.了解xargs及其实现流程

xargs是一个用于构建和执行命令行的实用程序，通常用于处理通过管道传递的输入。xargs从标准输入或者其他命令的输出中读取数据，并将其作为参数传递给指定的命令。
以下是基本语法：

```
command | xargs [options] [command]
```

- command： 要执行的命令。
- [options]： xargs 的选项，用于控制其行为。
- [command]： 可选，如果省略，则默认使用 echo 命令。
- 注意题干说的“从标准输入读取行并为每行运行一个命令”，这里的标准输入即0，和管道pipe的0是同一个引用。即将基本语法中的管道的读入直接使用标准输入的0替代即可
  - 管道的读端文件描述符通常是0。在UNIX系统中，标准输入通常使用文件描述符0，因此当数据通过管道传递到另一个进程时，它会被发送到读端的文件描述符0。
- 这个命令的具体执行流程如下
  - command命令生成一些输出，可能是文件名、目录名或其他数据，这些数据将作为xargs的输入。
  - xargs接收command的输出作为标准输入，并根据给定的选项对其进行处理，通常是将输入分割为单个参数，然后将这些参数传递给[command]。
  - [command]命令接收xargs处理后的参数作为输入，并执行相应的操作。

要完成实验，还需要了解 `exec()` 系统调用，exec() 函数接受两个参数：

- argv[1]：这是一个字符串，表示要执行的程序的路径或者命令名。
- pi_buf：这是一个指向字符串指针数组的指针，表示传递给新程序的参数列表。pi_buf 中的每个字符串都是一个参数。最后一个参数必须为 NULL，以便告知新程序参数列表的结束。

5.2.2.编辑xargs.c

具体的操作见代码注释。

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

int main(int argc, char *argv[])
{
    int i;
    int j = 0;
    char *pi_buf[32];       //储存所有参数的字符串指针数组
    char total_buf[256];    //储存所有输入的缓存区
    char *buf = total_buf;  //指向当前处理到的缓存区位置
    char *pi;               //指向当前处理字符串的开头
    pi = buf;

    //首先载入 xargs 后指令自带的参数
    for(i = 1; i < argc; ++i)
    {
        pi_buf[j++] = argv[i];
    }

    int len;
    int sum = 0;

    //这里的标准输出0指的是管道前的命令生成的输出
    //循环读入标准输入0
    while((len = read(0, buf, 32)) > 0)
    {
        sum += len; //当标准输入长度大于256时退出
        if(sum > 256)
        {
            printf("the args is too long!\n");
            exit(0);
        }

        //处理该轮读入的数据
        //参数之间可以由空格分隔，也可以通过换行符分隔。
        for(i = 0; i < len; ++i)
        {
            //读到了一个字符串的结尾
            if(buf[i] == ' ' )
            {
                buf[i] = 0; //手动将字符串结尾置为'\0'
                pi_buf[j++] = pi;   //在参数数组中加入新的参数字符串
                pi = &buf[i+1]; //pi指向下一个字符串的开头（如果有的话）
            }
            //读到了一行的结尾
            else if(buf[i] == '\n')
```

```
            {
                buf[i] = 0;
                pi_buf[j++] = pi;
                pi = &buf[i + 1];
                //上同读到字符串结尾的操作

                //将参数列表尾置0， exec要求参数列表第一个为指令本身，列表尾为0，即{cmd,arg1,arg2,
                pi_buf[j] = 0;

                //启动子进程执行命令
                if(fork() == 0)
                {
                    exec(argv[1], pi_buf);
                    exit(0);
                }

                else
                {
                    //处理完标准输出的一行，将参数列表置回初始状态
                    //{cmd, arg1, arg2, extern_arg1, extern_arg1,.. , 0} -> {cmd, arg1, arg2, }
                    j = argc - 1;
                    wait(0);
                }
            }
        }
        //当前缓冲区处理位置更新
        buf = buf + len;
    }

    exit(0);
    return 0;
```

5.2.3.编译并测试

```
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util xargs
make: “kernel/kernel”已是最新。
== Test xargs == xargs: OK (6.2s)
```

6.总运行结果

```
weicong@weicong-virtual-machine:~/xv6-labs-2021$ ./grade-lab-util
make: “kernel/kernel”已是最新。
== Test sleep, no arguments == sleep, no arguments: OK (2.9s)
== Test sleep, returns == sleep, returns: OK (2.5s)
== Test sleep, makes syscall == sleep, makes syscall: OK (2.3s)
== Test pingpong == pingpong: OK (2.5s)
== Test primes == primes: OK (2.7s)
== Test find, in current directory == find, in current directory: OK (2.9s)
== Test find, recursive == find, recursive: OK (4.5s)
== Test xargs == xargs: OK (5.2s)
== Test time ==
time: OK
Score: 100/100
```

五、实验总结

一开始我对系统调用感到很陌生，但经过这次实验，我不仅更为深入地了解了sleep,pingpong,primes,find,xargs的实现原理，还对ls，echo等系统调用有了更深的理解。同时，我对文件描述符、管道、标准输入标准输出有了一定的认识，总之这次实验让我学会了很多新的知识（具体内容在实验过程中呈现）！