哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2022 年秋季

课程名称：操作系统

实验名称：XV6 与 UNIX 实用程序

实验性质：课内实验

实验时间：9 月 19 日 地点：T2507

学生班级：5 班

学生学号：200110513

学生姓名：宗晴

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2022 年 9 月

一、 回答问题

1. 阅读 sleep.c，回答下列问题

（1） 当用户在 xv6 的 shell 中，输入了命令“sleep hello world\n"，请问在 sleep 程序里面，argc 的值是多少，argv 数组大小是多少。

此时，在 sleep 程序里，argc 的值为 3，argv 数组的大小也为 3。

（2） 请描述上述第一道题 sleep 程序的 main 函数参数 argv 中的指针指向了哪些字符串，它们的含义是什么。

argv 数组中的指针分别指向“sleep”，“hello”和“world”这三个字符串。argv 是一个指针，指向提供给主函数参数的字符串指针数组。其中，第一个字符串“sleep”为当前调用的程序命令的名称，第二个字符串“hello”为 sleep 命令的第一个输入参数，第三个字符串“world”为 sleep 命令的第二个输入参数。一般来说，argv[0]指向当前调用的程序命令名称的字符串，argv[1]到 argv[argc-1]均为该命令的输入参数。

（3） 哪些代码调用了系统调用为程序 sleep 提供了服务？

```
1  #include "kernel/types.h"
2  #include "user.h"
3
4  int main(int argc,char* argv[]){
5      if(argc != 2){
6          printf("Sleep needs one argument!\n"); //检查参数数量是否正确
7          exit(-1);
8      }
9      int ticks = atoi(argv[1]); //将字符串参数转为整数
10     sleep(ticks);              //使用系统调用sleep
11     printf("(nothing happens for a little while)\n");
12     exit(0); //确保进程退出
13 }
```

如上图所示，第 7 行的 exit(-1)与第 12 行的 exit(0)调用了 exit()系统调用函数；第 10 行的 sleep(ticks)调用了 sleep()系统调用函数。

2. 了解管道模型，回答下列问题

（1） 简要说明你是怎么创建管道的，又是怎么使用管道传输数据的。

① 创建管道

```
int p[2];
int ret;
ret = pipe(p);
```

如上图，首先创建一个长度为 2 的 int 型数组，然后通过 pipe(p)系统调用创建管道，用 int 型变量 ret 接收返回值，若成功创建，则返回 0，否则返回-1。管道成功创建后，pipe 会给 p 数组的两个位置赋上两个文件描述符来代表读写端，其中 p[1]作为写入端，p[0]作为读出端。文件描述符即 xv6 分配给每个文件的整数 ID，从 0 开始计数，0、1、2 分别代表标准输入文件、标准输出文件和标准错误输出文件。

② 传输数据

```
int w_len = write(p[1], buffer, n);
int r_len = read(p[0], buffer, n);
```

如上图，使用 write 和 read 系统调用传输数据。具体来说，write 从 buffer 中写 n 个字节到写入端 p[1]的文件描述符所指向的文件中，若成功写入则返回 n，否则返回-1。read 从读出端 p[0]的文件描述符所指向的文件中读出 n 个字节到 buffer 中，若成功读出则返回读出的字节数，否则返回-1。管道可视作一个固定大小的的缓冲区，若管道已经被写满，则写进程将会被阻塞，直到读进程读走数据。同样若管道已经被读空，则读进程将会被阻塞，直到写进程写入数据。并且管道通信是互斥的，即一个进程在对管道进行读/写操作时，另一个进程必须等待。

（2） fork 之后，我们怎么用管道在父子进程传输数据？

```
int p[2];

if ((pid = fork()) < 0) printf("Fork Error!\n");

if (pid == 0)
{   // 子进程
    ...
    close(p[1]); // 关闭写端
    read(p[0], buffer, sizeof(buffer));
    close(p[0]); // 读取完成，关闭读端
    ...
    exit(0);
} else
{  // 父进程
    ...
    close(p[0]); // 关闭读端
    write(p[1], buffer, sizeof(buffer));
    close(p[1]); // 写入完成，关闭写端
    ...
    exit(0);
}
```

如上图所示，若 fork 的返回值为负数，则表示创建子进程失败；若返回 0，则表示当前进程为子进程；若返回值大于零，则表示当前进程为父进程，且该返回值为子进程 pid。

以父进程写入数据、子进程读出数据为例。由于单个进程通常只持有某个管道的读出端或者写入端，因此使用的时候需要将另一端关闭。父进程在向管道写入数据之前，需要关闭自己的读端，然后待写入完成并且无需再次写入后，再关闭自己的写端。然后由子进程从管道中读出数据，在读数据之前，子进程同样需要先关闭自己的写端，然后待读取完数据并且无需再次读取后再关闭自己的读端。

需注意，从管道读取数据是一次性操作，数据一旦被读取就会从管道中被抛弃。同时，管道只能采用半双工通信，即某一时刻只能单向传输，要实现双方互动通信，需要定义两个管道。

（3） 试解释，为什么要提前关闭管道中不使用的一端？（提示：结合管道的阻塞机制）

因为管道的读写都是阻塞的，若不提前关闭读进程的写端，则当写进程正常结束后（关闭了其读写端），对于读进程来说，写端文件描述符的引用计数大于 0（包括读进程的写端），则此时若读进程已经将管道读空并且继续读取时，读进程将会被阻塞，但读进程的写端并不会向管道内写入数据，所以读进程会一直处于阻塞状态。

而提前关闭读进程的写端则不会出现这种情况，因为此时若读进程想要继续读取,但又由于此时写端文件描述符的引用计数等于0(读写进程均关闭了写端)，所以读进程在再次读取时会返回 0，表示再无数据可读，读进程因此可以顺利结束。

若不提前关闭写进程的读端，则同样会导致写进程在将管道写满之后一直处于阻塞状态。

二、 实验详细设计

注意不要照搬实验指导书上的内容，请根据你自己的设计方案来填写

1、sleep

对于 sleep 程序，argv 需要接受两个参数，包括“sleep”本身以及 sleep 的时间。因此程序首先需判断 argv 接受参数的个数是否为 2，若不是则会输出 "Sleep needs one argument!"，然后异常退出。

接着，程序将 sleep 接收到的字符串转换为整数，只将字符串中遇到的第一段数字字符串转换为整型，而忽略该数字前后的其它非数字字符。调用系统调用 sleep，将程序阻塞相应的时间后，输出"(nothing happens for a little while)"，然后正常退出。

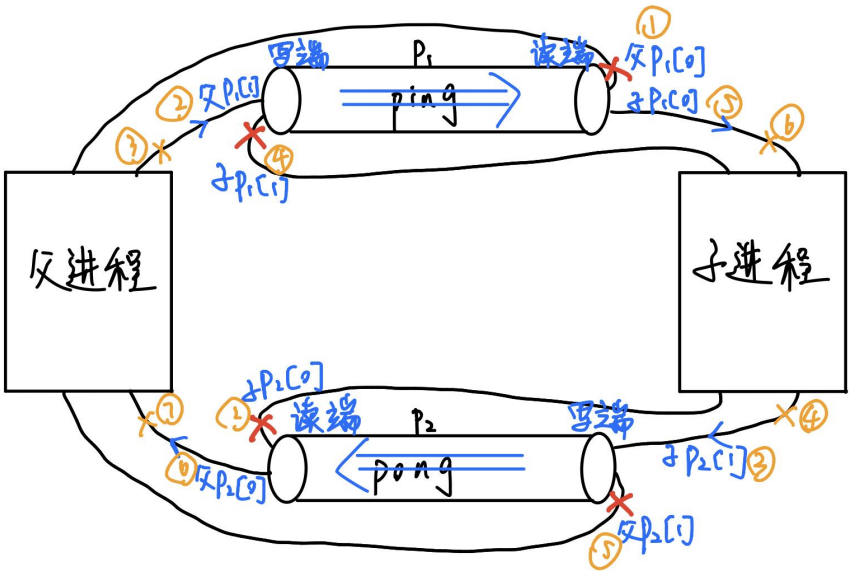
2、pingpong

该程序需要两个进程在管道两侧来回通信，由父进程将“ping”写入管道，

然后由子进程将其从管道中读出并打印，接着子进程将“pong”写入另一个管道，然后由父进程将其从管道中读出并打印。

为此，我们需要创建两个管道，一个用于父进程写“ping”后子进程读，另一个用于子进程写“pong”后父进程读。首先创建两个管道，若创建失败即返回值小于 0，则输出"Pipe Error!"并异常退出。然后调用 fork 系统调用创建子进程，若创建失败即返回的 pid 小于 0，则输出"Fork Error!"并异常退出。若 pid 等于 0，则表示当前在子进程，若大于 0，则表示当前在父进程。

父子两进程通信流程如下图所示：



通信顺序如上图中序号所示，首先通过 close 系统调用，关闭父进程在管道 1 的读端，然后通过 write 系统调用将“ping”写入管道，最后关闭父进程在管道 1 的写端。如下图：

```
close(p1[0]); // 关闭读端
write(p1[1], ping, sizeof(ping));
close(p1[1]); // 写入完成，关闭写端
```

接着子进程通过 close 系统调用，关闭其在管道 1 的写端，然后通过 read 系统调用将“ping”读出到 buffer 中，最后关闭子进程在管道 1 的读端，并输出自身的 pid 和读到的内容。如下图：

```
int pid_child = getpid();
close(p1[1]); // 关闭写端
read(p1[0], buffer, sizeof(buffer));
close(p1[0]); // 读取完成，关闭读端
printf("%d: received %s\n", pid_child, buffer);
```

然后子进程通过 close 系统调用，关闭其在管道 2 的读端，然后通过 write 系统调用将“pong”写入管道，最后关闭子进程在管道 2 的写端并正常退出。如下图：

```
close(p2[0]); // 关闭读端
write(p2[1], pong, sizeof(pong));
close(p2[1]); // 写入完成，关闭写端
```

接着父进程通过 close 系统调用，关闭其在管道 2 的写端，然后通过 read

系统调用将“pong”读出到 buffer 中，最后关闭父进程在管道 2 的读端，并输出自身的 pid 和读到的内容，同时正常退出。如下图：

```
close(p2[1]); // 关闭写端
read(p2[0], buffer, sizeof(buffer));
close(p2[0]); // 读取完成，关闭读端
printf("%d: received %s\n", pid_parent, buffer);
```

3、primes

该程序要求输出 2-35 之间的的所有质数。我们利用管道实现筛选，首先利用系统调用 pipe 创建管道，利用系统调用 fork 创建子进程，若创建失败则输出错误信息并异常退出。若 fork 返回值大于零，则表示当前在父进程，那么关闭父进程的读端，然后将 2-35 写入管道，接着关闭父进程的写端，等待子进程退出。代码如下：

```
} else{
    // 父进程
    close(p[0]);  // 关闭读端
    for(int i = 2; i <= last_num; i++){
        write(p[1], &i, sizeof(int));
    }
    close(p[1]);  // 关闭写端
    wait(0);
```

若 fork 返回值等于零，则表示当前在子进程，进行素数筛选。代码如下：

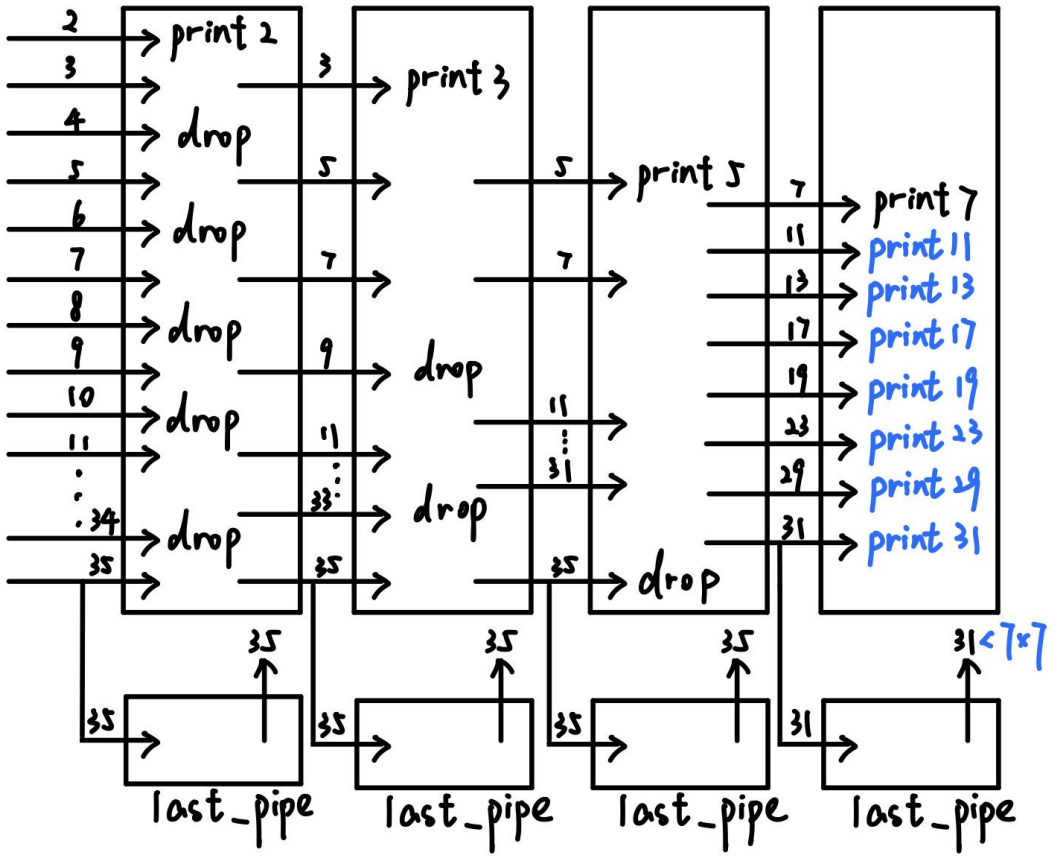
```
if (pid == 0){
    // 子进程
    filter(p, last_num);
```

用于素数筛选的 filter 函数输入参数为管道的读写端以及上一轮筛选后的最后一个数：

```
void filter(int p[2], int last_num){
```

该函数的设计如下图。我们输出每次读出的第一个数字，然后排除掉该数的倍数，将其它数字写入下一个管道进行下一轮筛选。所以每次读出的第一个数字一定是素数。同时，若当前轮的最后一个数小于第一个数的平方，那么这一轮不会再筛掉任何一个数，因为若其中任何一个数存在小于第一个数的因数，那它应该已经在之前轮被筛去。并且，之后每轮的最后一个数都会小于第一个数的平方，所以该轮的所有数均为素数。因此这时筛选便可停止，输出该轮剩下的所有数即可。该判断可以节省大量不必要的筛选时间。

由于管道是按顺序读取，所以我们另外创建了一个 last_pipe 管道用于传输每轮的最后一个数字。



首先，关闭写端，从管道中读取第一个数，由于前面没有能整除它的数，所以该数一定为素数，输出。代码如下：

```
int prime, next_num;
close(p[1]);  // 关闭写端
read(p[0], &prime, sizeof(int));  // 读取第一个数，一定为素数
printf("prime %d\n", prime);
```

若管道中还存在其他数，那么判断该轮的最后一个数是否小于第一个数的平方，若是，则直接输出剩余所有数：

```
int not_end = read(p[0], &next_num, sizeof(int));  // not_end表示是否读完
if(not_end){
    if(last_num < prime * prime){
// 若最后一个数字小于第一个素数的平方，则表示这一轮不会再筛掉任何合数，即剩下的均为素数
        do{
            printf("prime %d\n", next_num);  // 输出剩余的所有数
        } while (read(p[0], &next_num, sizeof(int)));
        exit(0);
```

否则还需进行下一轮的筛选，调用 pipe 系统调用创建新的管道 new_p 和用于传输最后一个数字的管道 last_p，以及 fork 系统调用创建新的子进程：

```
else{  // 否则需进行下一轮的筛选
  int new_p[2], last_p[2];  // last_p管道用来传输上一轮写入的最后一个数字
  if (pipe(new_p) < 0 || pipe(last_p) < 0){
      printf("Pipe Error!\n");
      exit(-1);
  }
  int pid;
  if ((pid = fork()) < 0){
      printf("Fork Error!\n");
      exit(-1);
  }
```

在父进程中排除掉第一个数的所有倍数，并且把其它数写入新的管道 new_p 中。此外，还需要把最后一个数写入 last_p 管道中。

```
} else{
    // 父进程
    close(new_p[0]);  // 关闭新管道的读端
    if(next_num % prime){
        write(new_p[1], &next_num, sizeof(int));
    }
    while (read(p[0], &next_num, sizeof(int)))
    {
        if(next_num % prime){  // 若不是第一个数的倍数，则继续写入下一轮
            write(new_p[1], &next_num, sizeof(int));
        }
    }
    close(p[0]);  // 关闭原管道的读端
    close(new_p[1]);  // 关闭新管道的写端
    close(last_p[0]);  // 关闭用于传输最后一个数的管道的读端
    write(last_p[1], &next_num, sizeof(int));  // 传输该轮的最后一个数
    close(last_p[1]);  // 关闭用于传输最后一个数的管道的写端
    wait(0);
}
```

在子进程中读取 last_p 管道中上一轮的最后一个数，并将管道读写端即该数当做参数，输出 filter 函数进行下一轮筛选：

```
if (pid == 0){
    // 子进程
    close(last_p[1]);  // 关闭写端
    read(last_p[0], &last_num, sizeof(int));  // 读取上一轮的最后一个数
    close(last_p[0]);  // 关闭读端
    filter(new_p, last_num);  // 进行下一轮的筛选
```

4、find

该程序要求在目录树中查找名称与字符串匹配的所有文件，并输出文件的相对路径。输入参数为待查找的目录，以及待匹配的文件名。

借鉴 ls 程序的代码，首先判断用户输入的包括 find 在内的参数个数是否为 3，若不是则输出错误信息并异常退出；若是则进入 find 函数进行搜索：

```
if(argc < 3){
  printf("find: need 3 params!\n");
  exit(-1);
}
find(argv[1], argv[2]);  // 第一个参数为路径，第二个参数为待查找文件名
exit(0);
```

`find(char *path, char *file)` 表示在 path 路径下查找名字为 file 的文件。

在 find 函数中，首先通过 open 系统调用打开 path 路径对应的文件。如下图，open(path, 0)中的 0 表示打开方式为只读，函数返回文件描述符给 fd。（若失败则输出错误信息并返回。）然后通过 fstat 系统调用将文件描述符 fd 所对应的文件信息存储到 st 中。（若失败则输出错误信息，关闭文件并返回。）

```
if((fd = open(path, 0)) < 0){  // 打开路径
  fprintf(2, "find: cannot open %s\n", path);
  return;
}

if(fstat(fd, &st) < 0){  // 存储文件信息
  fprintf(2, "find: cannot stat %s\n", path);
  close(fd);
  return;
}
```

然后根据 st.type 判断当前打开的是文件还是文件夹，若是文件则直接返回：

```
switch(st.type){
case T_FILE:  // 表示第一个参数不是文件夹名
  printf("find: the first param should be dirname\n");
  return;
```

若是文件夹，则首先判断路径是否过长：

```
case T_DIR:
  if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
    printf("find: path too long\n");
    break;
```

若未超出 buf 所能存储的字符串长度，则继续进行后续判断。将路径 path 拷贝到 buf 中，将指针 p 指向当前的最后一位，加上“/”用于分隔当前路径和后续的文件名：

```
strcpy(buf, path);
p = buf+strlen(buf);
*p++ = '/';
```

然后遍历该文件夹中的所有文件，需要注意避免进入“.”和“..”而造成无限递归。其中文件名等信息存储在 de 中：

```
while(read(fd, &de, sizeof(de)) == sizeof(de)){
  if(de.inum == 0 || !strcmp(de.name, ".") || !strcmp(de.name, ".."))  // 避免递归进入"."和".."
    continue;
```

将当前文件名拷贝到 buf 数组当前存储的内容之后，其中 p 指针一直指向上层文件夹末尾的“/”后一个地址。然后在最后加上字符串的结束符：

```
memmove(p, de.name, DIRSIZ);
p[DIRSIZ] = 0;  // 加上结束符
```

然后存储当前文件的信息，若失败则输出错误信息并返回：

```
if(stat(buf, &st) < 0){
  printf("find: cannot stat %s\n", buf);
  continue;
```

判断当前所指向的是文件还是文件夹。若是文件，则判断该文件名是否与待查找的文件名一致，若一直则输出 buf 中存储的包括当前文件名在内的完整路径。若是文件夹，则将当前 buf 中存储的文件夹路径与待查找的文件名一同输入到 find 函数中进行下一层的查找：

```
switch(st.type){
  case T_FILE:
      if(!strcmp(de.name, file)) printf("%s\n", buf);  // 成功找到文件
      break;
  case T_DIR:
      find(buf, file);  // 递归查找当前文件夹
      break;
}
```

5、xargs

该程序要求从标准输入中读取一行，并为每行运行一次指定的命令，且将该行作为命令的参数提供。

该程序至少接受到包括 xargs 在内的两个参数，同时参数个数也不能超过上限 MAXARG。若参数个数不满足要求，则输出对应的错误信息并异常退出：

```
if(argc < 2){
  printf("xargs: need at least 2 params!\n");
  exit(-1);
}
if(argc > MAXARG + 1){
  printf("xargs: params are too much!\n");
  exit(-1);
}
```

创建 params 指针数组用于存储命令及每次待运行的全部参数，将包括命令在内的第二个及以后的参数地址复制到该数组中：

```
char *params[MAXARG];
for(int i = 1; i < argc; i++){
  params[i-1] = argv[i];  // 将包括命令在内的第二个及以后的参数地址复制到参数列表中
```

创建 buf 数组存储接下来每行读到的内容。调用现有的 gets 函数读取每行的内容，该函数在读到 ctrl+D 的时候，返回空，由此判断用户的输入是否已经结束。对于读到的每行内容，需要去掉最末位读到的“\n”，然后将该行字符串按照空格切分成单独的参数，对于每个单独的参数，均将其首地址存储到 params 指针数组中。

```
char buf[512] = "\0";
int pid;
while (strcmp(gets(buf, 512), "\0"))
{
  buf[strlen(buf)-1] = '\0';  // 去掉最末位读入的"\n"
  paramSplit(buf, params, argc);  // 将参数按空格切分
```

其中切分函数 paramSplit 函数如下，参数为每行读入的内容，参数指针数组以及原参数个数：

```
void paramSplit(char *buf, char *params[], int argc){
```

该函数首先将所有空格替换为字符串结束符，然后将 params 指针数组中的指针指向切分后的参数，同时记录参数个数，确保参数个数不会超过上限。最后把 params 指针数组中的最后一个位置置零，用于后续 exec 系统调用的实现：

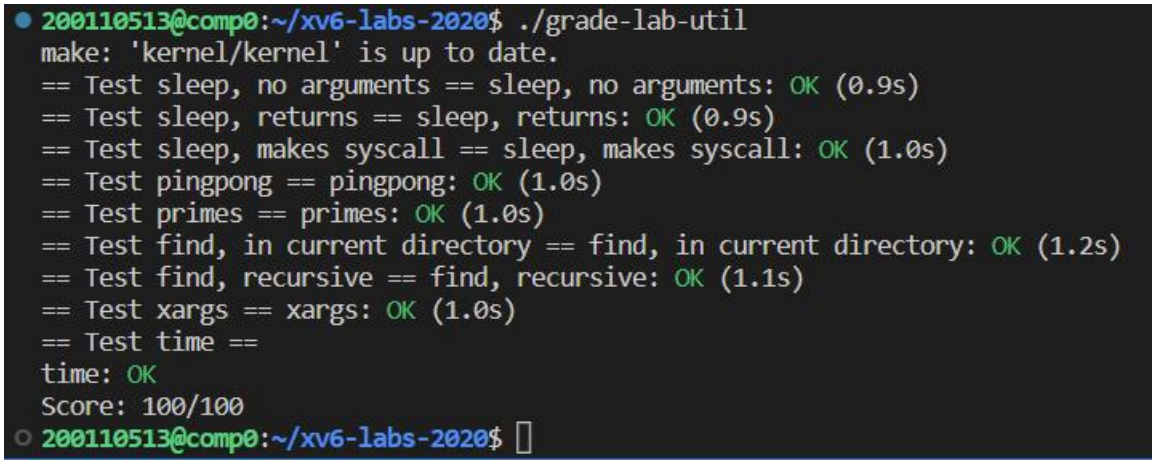
```
while(i < len){
      while(i < len && buf[i] != ' ') i++;  // 按空格切分
      while(i < len && buf[i] == ' ') buf[i++] = '\0'; // 将空格换成字符串结束符
      if(argc - 1 + pnum >= MAXARG){
        printf("xargs: params are too much!\n");
        exit(-1);
      }
      params[argc - 1 + pnum] = buf + begin;  // params指针数组指向切分后的参数
      begin = i;
      pnum++;  // num记录参数个数
  }
if(argc - 1 + pnum >= MAXARG){
  printf("xargs: params are too much!\n");
  exit(-1);
}
params[argc - 1 + pnum] = 0;
```

参数切分完成后，通过 fork 系统调用创建子进程用于执行输入的命令。创建失败则输出错误信息并异常退出。返回值为 0 表示在子进程，调用 exec 系统调用，传入待执行命令及参数指针数组，即可执行该命令。若执行失败则输出错误信息并异常退出。若 fork 返回值大于零，表示在父进程，则等待子进程退出即可：

```
  if ((pid = fork()) < 0) {
    printf("Fork Error!\n");
    exit(-1);
  }
  if(pid == 0){  // 子进程
    if(exec(argv[1], params) == -1) {
      printf("xargs: exec failed!");
      exit(-1);
    }
    exit(0);
  } else wait(0);  // 父进程
}
```

三、 实验结果截图

请填写


200110513@comp0:~/xv6-labs-2020$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.9s)
== Test sleep, returns == sleep, returns: OK (0.9s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test primes == primes: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.2s)
== Test find, recursive == find, recursive: OK (1.1s)
== Test xargs == xargs: OK (1.0s)
== Test time ==
time: OK
Score: 100/100
200110513@comp0:~/xv6-labs-2020$