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

实验设计报告

开课学期：2023 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：2023.9.21 地点：T2507

学生班级：8

学生学号：210110812

学生姓名：李春阳

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2023 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。

argc=3；
argv 数组大小为 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 参数是一个指向字符串数组的指针，它存储了命令行参数的值。对于命令“sleep hello world\n"，argv[0]指向"sleep"，表示当前调用的程序命令的名称；argv[1]指向" hello "，表示命令的第一个输入参数；argv[2]指向"world"，表示命令的第二个输入参数；argv 的最后一个元素为空指针（即数值 0），用以标志参数的结束。

（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 }
```

第 7 行代码中，调用系统调用接口 exit()，在参数数量不为 2 时错误的退出程序；
第 10 行代码中，调用系统调用接口 sleep()，使当前进程睡眠 ticks 时间；
第 12 行代码中，调用系统调用接口 exit()，显式的正常退出程序。

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 管道的创建：
首先创建两个大小为 2 的 int 型数组，然后通过 pipe()系统调用创建管道。pipefd 参数返回两个文件描述符，pipefd [0]指向管道的读出端，pipefd [1]指向管道的写入端。

如系统调用成功，则返回值为 0，否则则返回-1。当管道创建出现错误时，控制台将打印出“Pipe Error”的提示。

```
int pipefd1[2];   //管道1传递ping:父进程->子进程
int pipefd2[2];   //管道2传递pong:子进程->父进程

char buffer[MAXSIZE];

//判断管道是否出现错误
if (pipe(pipefd1) < 0 || pipe(pipefd2) < 0) {
    printf("Pipe Error!\n");
    exit(-1);
}
```

② 传输数据：

我们以子进程读管道，父进程写管道为例：

read(pipefd1[0], buffer, MAXSIZE);

write(pipefd1[1], "ping", MAXSIZE);

在管道中的传输数据通过 read 和 write 系统调用实现，write 将字符串“ping”写到写入端 pipefd1[1]的文件描述符所指向的文件中，若成功则返回写入的字节，否则返回-1，并设置 errno。read 从读出端 pipefd1[0]的文件描述符所指向的文件中读 MAXSIZE 个字节到缓冲区 buffer 中，若成功读出则返回读出的字节，否则返回-1，并设置 errno。管道实际上是内核开辟的一段长度有限的缓冲区，管道的读写就是对缓冲区的读写。

（2） fork 之后，我们怎么用管道在父子进程传输数据？

父进程通过 fork()系统调用创建子进程，并设置一个缓冲区 buffer[MAXSIZE]；

① 子进程读管道，父进程写管道

由于进程通常只持有某个管道的读出端或者写入端，因此需要将父进程的读管道关闭，子进程的写管道关闭；然后父进程向管道中写入数据，子进程将管道中的数据读出；一次数据传输完成后，父进程关闭写入端，子进程关闭读出端。

② 子进程写管道，父进程读管道

为了实现父子双方互相通信，因此需要定义两个单向管道。其传输过程与①描述大致相同。

```
19    /* 正常创建后，p[1]为管道写入端，p[0]为管道读出端*/
20    if (fork() == 0) {
21        /* 子进程 */
22        /* 子进程读管道，父进程写管道 */
23        close(pipefd1[1]); // 关闭写端
24        read(pipefd1[0], buffer, MAXSIZE);
25        printf("%d: received %s\n", getpid(), buffer);
26        close(pipefd1[0]); // 读取完成，关闭读端
27
28        /* 子进程写管道，父进程读管道 */
29        close(pipefd2[0]); // 关闭读端
30        write(pipefd2[1], "pong", MAXSIZE);
31        close(pipefd2[1]); // 读取完成，关闭写端
32
33    } else if (fork()>0) {
34        /* 父进程 */
35        /* 子进程读管道，父进程写管道 */
36        close(pipefd1[0]); // 关闭读端
37        write(pipefd1[1], "ping", MAXSIZE);
38        close(pipefd1[1]); // 写入完成，关闭写端
39
40        /* 子进程写管道，父进程读管道 */
41        close(pipefd2[1]); // 关闭写端
42        read(pipefd2[0], buffer, MAXSIZE);
43        printf("%d: received %s\n", getpid(), buffer);
44        close(pipefd2[0]); // 读取完成，关闭读端
45    }
```

（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）

关闭管道中不使用的一端是为了避免出现阻塞的情况。

在管道通信中，数据通过管道的读取端传输到写入端。当管道中没有数据可读时，读取端会阻塞（即暂停执行），直到有数据可读为止。类似地，当管道的写入端已满时，写入端会阻塞，直到有空间可写入数据为止。

如果在使用管道进行进程间通信时，某一端不再需要读取或写入数据，但仍保持打开状态，那么这一端可能会一直阻塞等待数据的到来或空间的释放，从而导致程序无法继续执行。

通过提前关闭管道中不使用的一端，可以避免阻塞情况的发生。关闭不使用的一端后，如果另一端尝试读取或写入数据，会立即得到一个特殊的信号（文件描述符返回值为 0），表明管道已经关闭。这样就可以在程序中及时处理这种情况，避免阻塞和无法继续执行的问题。

二、 实验详细设计

1. sleep 设计方案：已给出源代码

2. pingpong 设计方案：

该程序需要实现两个进程在管道两侧来回通信。父进程将“ping”写入管道，子进程从管道将其读出并打印<pid>: received ping。子进程从父进程收到字符串后，将“pong”写入另一个管道，然后由父进程从该管道读取并打印<pid>: received pong。

根据题目含义，我们创建两个大小为 2 的 int 型数组，并通过 pipe()系统调用创建管道。pipefd 参数返回两个文件描述符，pipefd [0]指向管道的读出端，pipefd [1]指向管道的写入端。如系统调用成功，则返回值为 0，否则则返回-1。当管道创建出现错误时，控制台将打印出“Pipe Error”的提示。

```
int pipefd1[2];
int pipefd2[2];

if (pipe(pipefd1) < 0 || pipe(pipefd2) < 0) {
    printf("Pipe Error!\n");
    exit(-1);
}
```

管道正确创建后，通过 fork()系统调用创建子进程。当子进程读管道、父进程写管道时，先关闭管道不使用的一端（子进程关闭写端、父进程关闭读端），然后通过 write 系统调用将字符串“ping”写入缓冲区 buffer 中，通过 read 系统调用将缓冲区中的数据读出并打印。当此次数据传输完成后，父进程关闭写入端，子进程关闭读出端。

```
if (fork() == 0) {                  /*子进程 */
    close(pipefd1[1]);
    read(pipefd1[0], buffer, MAXSIZE);
    printf("%d: received %s\n", getpid(), buffer);
```

```
        close(pipefd1[0]);
    } else if (fork()>0) {              /*父进程 */
        close(pipefd1[0]);
        write(pipefd1[1], "ping", MAXSIZE);
        close(pipefd1[1]);
    }
```

子进程写管道、父进程读管道时同理。

```
    if (fork() == 0) {                  /*子进程 */
        close(pipefd2[0]);
        write(pipefd2[1], "pong", MAXSIZE);
        close(pipefd2[1]);
    } else if (fork()>0) {              /*父进程 */
        close(pipefd2[1]);
        read(pipefd2[0], buffer, MAXSIZE);
        printf("%d: received %s\n", getpid(), buffer);
        close(pipefd2[0]);
    }
```

3. pingpong 设计方案：

该程序需要实现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。

基于 ls.c 代码，我们修改 fmtname 函数。去掉 ls 代码中格式化的部分，仅保留该函数“将文件路径中的目录部分去除，只返回文件名”的作用，如下所示：

```
char *fmtname(char *path) {
    static char buf[DIRSIZ + 1];
    char *p;

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

    memmove(buf, p, strlen(p)+1);
    return buf;
}
```

不同于 ls 函数，find 函数共需输入两个参数：find(char *path, char *filename)。在该函数中，首先通过 open 系统调用打开文件，并判断是否能正确打开文件并存储文件信息。

```
  if ((fd = open(path, 0)) < 0) {
      fprintf(2, "find: cannot open %s\n", path);
      return;
  }
  if (fstat(fd, &st) < 0) {
      fprintf(2, "find: cannot stat %s\n", path);
      close(fd);
      return;
```

```
}
```

然后根据 st.type 进行分类处理：如果类型是文件，则通过 strcmp()判断该文件是否为待查找的文件，若是则输出对应的路径。

```
case T_FILE:
    if(strcmp(fmtname(path), filename) == 0){
        printf("%s\n", path);
    }
    break;
```

如果类型为文件夹，则在检查缓存是否溢出后，使用 find 递归进入子目录寻找待查找的文件（跳过“.”和“..”的递归）

```
strcpy(buf, path);
p = buf + strlen(buf);
*p++ = '/';
while (read(fd, &de, sizeof(de)) == sizeof(de)) {
    if (de.inum == 0) continue;
    memmove(p, de.name, DIRSIZ);
    p[DIRSIZ] = 0;
    if(strcmp(de.name,".") == 0 || strcmp(de.name,"..") == 0){
        continue;
    }
    find(buf, filename);
}
break;
```

对于 main 函数，需要判断参数的数量，如果参数的数量不为 3，则打印“Wrong number of parameters”，否则将第二个和第三个参数传递给 find。

```
int main(int argc, char *argv[]) {
    if (argc == 3) {
        find(argv[1], argv[2]);
        exit(0);
    } else {
        printf("Wrong number of parameters!\n");
        exit(-1);
    }
}
```

三、 实验结果截图

sleep.c

```
$ sleep 10
(nothing happens for a little while)
```

```
210110812@comp7:~/xv6-oslab23-hitsz$ ./grade-lab-util sleep
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)
```

pingpong.c

```
$ pingpong
4: received ping
3: received pong
```

```
210110812@comp7:~/xv6-oslab23-hitsz$ ./grade-lab-util pingpong
make: 'kernel/kernel' is up to date.
== Test pingpong == pingpong: OK (1.3s)
```

find.c

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

```
210110812@comp0:~/xv6-oslab23-hitsz$ ./grade-lab-util find
make: 'kernel/kernel' is up to date.
== Test find, in current directory == find, in current directory: OK (1.0s)
== Test find, recursive == find, recursive: OK (1.1s)
```

```
210110812@comp0:~/xv6-oslab23-hitsz$ ./grade-lab-util
make: 'kernel/kernel' is up to date.
== Test sleep, no arguments == sleep, no arguments: OK (0.8s)
== Test sleep, returns == sleep, returns: OK (1.0s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
== Test pingpong == pingpong: OK (1.0s)
== Test find, in current directory == find, in current directory: OK (1.1s)
== Test find, recursive == find, recursive: OK (1.2s)
Score: 60/60
```