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

实验设计报告

开课学期：2024 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 29 地点：T2507

学生班级：22 级计算机 4 班

学生学号：220110430

学生姓名：吴梓滔

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2024 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 的值是 3，argv 数组大小为 4。

（2） 请描述上述第一道小题 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[0]=”sleep”, argv[1]=”hello”,argv[2]=”world”。argv[0]是命令名，后续的两个参数是传给命令执行的参数。argv[3]=NULL,表示结束。

（3） 程序 sleep 使用了哪些系统调用？

printf 用于向标准输出打印指定的信息。
exit 用于退出程序。
sleep 用于进程暂停指定的时间。

2. 了解管道模型，回答下列问题

（1） 简要说明在 pingpong 实验中，你是怎么创建管道的？结合 fork 系统调用说明你是怎么使用管道在父子进程之间传输数据的。

```
int c2f[2];     // Pipe for child to father
int f2c[2];     // Pipe for father to child
pipe(c2f);
pipe(f2c);
```

先声明长度为 2 的数组，用于接收管道两端的文件描述符；在调用系统调用 pipe，传入对应的数组作为参数。上述代码分别创建了供子进程向父进程传输信息的管道和父进程向子进程传输信息的管道。

```
int pid = fork();
```

```
if (pid < 0) {
    // fork() 返回值如果是负数就是错误的
    exit(1);
} else if (pid == 0) {
    // child
    close(f2c[1]);
    close(c2f[0]);
    ……
} else {
    // father
    close(f2c[0]);
    close(c2f[1]);
    ……
}
```

在通过 fork 调用创建子进程后，此时父子进程是独立地拥有 c2f、f2c 两个管道的一共 4 个端口，因此父子进程各自关闭掉该进程不操作的管道端口。子进程关闭 f2c 的写端和 c2f 的读端，父进程关闭 f2c 的读端和 c2f 的写端。这样就完成了管道的创建。

在传输数据时，直接使用 write 和 read 系统调用，首个参数传入对应的管道文件描述符，第二个参数传入要传输的信息的地址，第三个参数传入传输的字节数，就可以完成读写。例如父进程传向子进程的信息，在本设计中是传输进程 pid，父进程中有 write(f2c[1], &father_id, 4)用于向 f2c 管道写入，子进程中有 read(f2c[0], &recieved_father_id, sizeof(int))用于向 c2f 管道中读取。由于管道的阻塞性质，在子进程创建后，两个进程同步运行，一定会由父进程先执行 write 一句，再由子进程执行 read 一句。子进程向父进程的传输同理。

（2） 再次阅读实验原理-管道这一节中给出的关于 wc 命令的示例程序，假设子进程没有关闭管道写端，运行该程序后，尝试描述会发生什么？为什么在 Linux 环境下编译运行再执行 ps 指令会有如下结果？

```c
#include <unistd.h>

int main() {
    int p[2];         /* 存储管道的两个文件描述符 */
    char *argv[2];
    argv[0] = "wc";   /* 设置要执行的命令为"wc" */
    argv[1] = 0;      /* 参数数组以NULL结尾，表示没有更多参数 */
    pipe(p);          /* 创建管道，p[0]是管道的读端，p[1]是管道的写端 */

    if (fork() == 0) {
        /* 子进程 */
        close(0);     /* 关闭标准输入（文件描述符0） */
        dup(p[0]);    /* 复制管道的读端p[0]，让文件描述符0指向管道的读端 */
        close(p[0]); /* 关闭不再需要的管道读端 */
        // close(p[1]); /* 关闭不再需要的管道写端 */
        execv("/bin/wc", argv); /* 执行"wc"程序 */
    } else {
        /* 父进程 */
        close(p[0]);                        /* 关闭管道的读端 */
        write(p[1], "hello world\n", 12); /* 向管道写入"hello world\n" */
        close(p[1]);                        /* 关闭管道的写端，表示写入完成 */
    }
    return 0;
}
```

```
~/os/xv6-oslab24-hitsz | on util ?4  gcc test.c -o test
~/os/xv6-oslab24-hitsz | on util ?4  ./test
~/os/xv6-oslab24-hitsz | on util ?4  ps
    PID TTY          TIME CMD
  44896 pts/6    00:00:01 zsh
  44938 pts/6    00:00:00 gitstatusd-linu
  46667 pts/6    00:00:00 wc
  46724 pts/6    00:00:00 ps
~/os/xv6-oslab24-hitsz | on util ?4
```

运行该程序后，由于子进程里有管道的写端未关闭，子进程执行 wc 后成为 wc 进程通过管道读取输入会一直处于阻塞状态，会等待新数据而永远不会结束。当管道中没有数据可读时，read 会阻塞等待，直到有数据被写入，或者直到所有指向管道写端的文件描述符都被关闭。如果所有写端的文件描述符都被关闭，read 操作会返回 0，表示读到文件的末尾。由于子进程里的管道写端未关闭，read 保持阻塞状态，使 wc 不会结束。

在 Linux 环境下运行，由于 Linux 中的管道有类似的阻塞等待行为，由于子进程管道写端没有关闭，当子进程执行了 excev 后成为 wc 进程，wc 进程通过 read 尝试从管道读取数据，将一直处于阻塞状态，所以运行 ps 命令会看到有一个 wc 进程没有结束。

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） pingpong 设计

该程序需要在父子进程之间互相通信，因而需要创建两个管道，分别用于父进程向子进程、子进程向父进程传输。由于父进程和子进程需要互相输出对方的 pid，而子进程收到消息固定打印“ping”，父进程收到消息固定打印“pong”，输出消息中己方 pid 可以直接用 getpid() 获得，考虑将进程 pid 作为通过管道传输的信息。再考虑固定是父进程先向子进程发送，再由子进程向父进程发送，因此可以先后完成两次管道传输的编程。只需父进程先向子进程发送父进程的 pid，子进程收到并打印消息后，再由子进程向父进程发送子进程的 pid，父进程收到并打印消息，即可推出程序。

先创建两个管道：

```
int c2f[2];     // Pipe for child to father
int f2c[2];     // Pipe for father to child
pipe(c2f);
pipe(f2c);
```

管道的设计：

c2f：子进程向父进程发送消息使用，子进程持有写端口 c2f[1],父进程持有读端口 c2f[0]。

f2c：父进程向子进程发送消息使用，父进程持有写端口 f2c[1],子进程持有读端口 f2c[0]。

通过 fork()创建子进程，在进入父进程或子进程后，立即关闭不需要的管道文件描述符。

```
int pid = fork();
if (pid < 0) {
    // fork() 返回值如果是负数就是错误的
    exit(1);
} else if (pid == 0) {
    // child
    close(f2c[1]);
    close(c2f[0]);
    int recieved_father_id;
    int child_id = getpid();
    read(f2c[0], &recieved_father_id, sizeof(int));
    close(f2c[0]);
   printf("%d: received ping from pid %d\n", child_id, recieved_father_id);
    write(c2f[1], &child_id, 4);
    close(c2f[1]);
    exit(0);
} else {
    // father
    close(f2c[0]);
```

```
            close(c2f[1]);
            int father_id = getpid();
            int received_child_id;
            write(f2c[1], &father_id, 4);
            close(f2c[1]);
            read(c2f[0], &received_child_id, sizeof(int));
            close(c2f[0]);
             printf("%d: received pong from pid %d\n", father_id,
received_child_id);
            exit(0);
        }
```

随后利用管道读取能够阻塞等待的行为，先由父进程写入并关闭写端，子进程读后立即关闭读端并输出信息（标红色代码所示）。随后子进程写入并关闭写端，父进程读后立即关闭读端并输出消息（标蓝色代码所示）。之后即可退出程序。

父进程和子进程通过 write 调用向管道写入的信息，实际上不是字符串，而是直接将本进程的 pid 通过 getpid()获取后，是一个 int 型变量，通过地址将该 int 型变量直接通过管道传出。接收时的 read 调用传入的第二个参数也是用一个 int 型的地址去接收。这样管道实际上是直接传输了一个 int 型的进程 pid 值。子进程收到父进程的消息后输出“ping”的文本，父进程收到子进程的消息后输出“pong”的文本。

（2） find 设计

find 命令在使用时，是通过 find <path> <filename>进行调用，因此仿造 ls.c 的设计方式，结构分为 void find() 和 int main()两个部分，将主要功能在 find()内实现，在 main()里面可以进行命令读取的参数是否正确的判断，并在参数正确时调用 find()。

具体 find()的参数设计：find 读入两个参数，即用户输入的 path 和 filename。path 是一个目录，而 filename 是目标要寻找的文件或目录名。find(path, filename)将在标准输出打印出 path 目录下所有的名为 filename 的文件或者目录的名称和路径。

首先仿造 ls.c 对 path 进行打开和读取 stat。

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

将目录 path 打开并读取 stat 信息到 st 中后，由于我们认为 path 必须是一个目录，所以进行一次类型的判断，判断 path 的类型必须是目录，才可以继续执行。

```
 if (st.type != T_DIR) {
     fprintf(2, "find: path should be directory\n");
     close(fd);
     return;
```

```
 }
```

此后可以模仿 ls.c 的设计方式，依次读取该目录下每个文件进行处理，我们希望获取文件名和 filename 进行比较，并且对于目录要递归调用 find()，这需要判断每个文件的类型，因此需要获得这些文件的 stat。ls.c 中已经实现了这个功能。读取该目录下的每一个文件，每次读取的文件为 de，将它从 path 开始的路径存储为字符串 buf，并读取 stat 信息到 st 中，此后可以对 st 进行类型的判断和名称的比较。

```
strcpy(buf, path);               // copy path
p = buf + strlen(buf);
*p++ = '/';                     // end of path append '/'
while (read(fd, &de, sizeof(de)) == sizeof(de)) {
   if (de.inum == 0) continue;
   memmove(p, de.name, DIRSIZ);
   p[DIRSIZ] = 0;
   if (stat(buf, &st) < 0) {
       printf("cannot stat %s\n", buf);
       continue;
     }
   …… // 对 st 的文件进行类型判断和处理
 }
```

上面一段代码在 ls.c 中以及实现，至此已经完成了对文件 stat 信息的读取，储存在结构体 st 中。通过 st 判断类型，如果是文件，则比较这个文件的名称是否与目标文件或目录名 filename 一致，若一致则可以打印；如果是目录，则先判断该目录名称是否与目标名 filename 一致，若一致则输出；此后要对该目录递归使用 find 命令。上面省略部分的代码如下：

```
switch (st.type) {
  case T_FILE:
    if (strcmp(filename, de.name) == 0) {
      printf("%s\n", buf);
    }
    break;

 case T_DIR:
   if (strcmp(filename, de.name) == 0) {
       printf("%s\n", buf);
    }
   if (strcmp(de.name, ".") != 0 && strcmp(de.name, "..") != 0) {
       find(buf, filename);
   }
   break;
}
```

通过 st.type 进行类型判断，如果是文件类型，则只需比较文件名是否与目标一致，并输出从路径。对于目录类型，为了实现对目录的查找，先进行文件名的比较，再开始递归查找。由于在此前的文件路径构建中，buf 是从目录 path 开始的文件路径，因此直接输出 buf 即可。在实现以上功能后，即可关闭文件描述符，find 的实现就完成了。

在 main()函数中，先对参数进行检查，argc 应该为 3，否则就报错退出。在参数正确时

直接调用 find()函数即可。

```
int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("find need 2 parameters!");
        exit(1);
    }
    find(argv[1], argv[2]);
    exit(0);
}
```

（3） xv6 启动流程分析

给出 commands.gdb 文件中的 GDB 命令序列解释，要求能打印出"initcode"和"init"程序的名字

```
1   b trap.c:55
2   c
3   p cpus[$tp]->proc->name
4   n
5   p cpus[$tp]->proc->name
6   da
```

命令 1 将断点设置在 trap.c 文件第 55 行，即“syscall();”的位置。在 xv6 启动过程中，这一句进行系统调用，在这一句系统调用后，cpu 当前的进程就从 initcode 切换为 init 程序。

命令 2 从开始时开始执行，会一直执行到命令 1 所设置的断点处停止。此时 initcode 已经在运行。

命令 3 打印出进程，执行后会按要求输出 initcode。

命令 4 单步步过，执行当前一步的代码，会执行完第 55 行的“syscall();”，直接到下一句。

命令 5 打印出进程，此时进程已经是 init，会按要求输出 init。

命令 6 刷新 gdb 面板，此后 history 一栏按要求显示历史输入。

三、 实验结果截图

请给出用户程序实验结果截图（包括 sleep、pingpong、find 运行截图，以及 make grade 命令结果截图）和 xv6 启动流程实验输出截图

sleep 命令

```
$ sleep 20
(nothing happens for a little while)
```

pingpong 命令

```
$ pingpong
5: received ping from pid 4
4: received pong from pid 5
$ pingpong
7: received ping from pid 6
6: received pong from pid 7
```

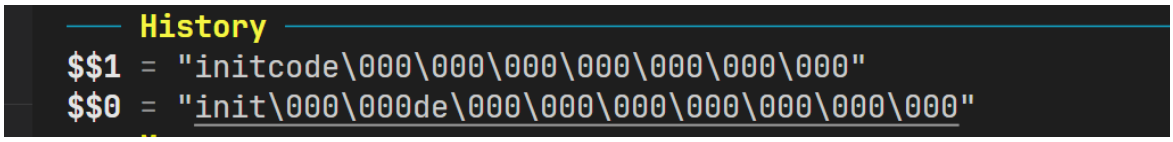
find 命令（创建了三个名为 target 的文件或目录）

```
$ mkdir wztnb
$ mkdir wztnb/target
$ mkdir wztnb/abc
$ echo > wztnb/abc/target
$ echo > target
$ find . target
./wztnb/target
./wztnb/abc/target
./target
```

make grade 执行结果

```
make[1]: Leaving directory '/home/students/220110430/xv6-oslab24-hitsz'
== Test sleep, no arguments ==
$ make qemu-gdb
sleep, no arguments: OK (4.4s)
== Test sleep, returns ==
$ make qemu-gdb
sleep, returns: OK (0.7s)
== Test sleep, makes syscall ==
$ make qemu-gdb
sleep, makes syscall: OK (1.0s)
== Test pingpong lenient testing ==
$ make qemu-gdb
pingpong lenient testing: OK (1.0s)
== Test pingpong strict testing with changing pids ==
$ make qemu-gdb
pingpong strict testing with changing pids: OK (1.0s)
== Test find, in current directory and create a file ==
$ make qemu-gdb
find, in current directory and create a file: OK (1.0s)
== Test find, in current directory and create a dir ==
$ make qemu-gdb
find, in current directory and create a dir: OK (1.1s)
== Test find, find file recursive ==
$ make qemu-gdb
find, find file recursive: OK (1.1s)
== Test find, find dir recursive with no duplicates ==
$ make qemu-gdb
find, find dir recursive with no duplicates: OK (1.2s)
Score: 60/60
```

xv6 启动实验输出


History
$$1 = "initcode\000\000\000\000\000\000\000"
$$0 = "init\000\000de\000\000\000\000\000\000\000"