## Operating System Lab 3 : 文件系统的用户界面

# 实 验 报 告

刘子涵　518021910690

## 【1】实验题目

1. 编写一个**文件复制**的 C 语言程序：
    - 基于文件的系统调用：`read(fd, buf, nbytes)` 和 `write(fd, buf, nbytes)`
    - 基于文件的库函数：`fread(buf, size, nitems, fp)` 和 `fwrite(buf, size, nitems, fp)`
        - 当上述函数中的 `nbytes`，`size`，`nitems` 取值为 1 （一次读写一个字节），比较以上两种程序的执行效率
        - 当上述函数中的 `nbytes` 和 `size` 取值为 1024，`nitems` 取值为 1（一次读写 1024 个字节），比较以上两种程序的执行效率
    - 基于 `fscanf` 和 `fprintf`
    - 基于 `fgetc` 和 `fputc`
    - 基于 `fgets` 和 `fputs`（仅限于行结构的文本文件）
2. 编写一个父子进程用**无名管道**进行数据传送的 C 语言程序
    - 父进程逐一读出一个文件的内容，并通过管道发送给子进程
    - 子进程从管道中读出信息，再将其写入一个新的文件
    - 程序结束后，对原文件和新文件的内容进行比较
3. 在两个用户的独立程序之间，使用**有名管道**编写一个 C 语言程序，重新实现上述功能。
4. 使用父子进程间的管道通信，实现 `who | wc -l` 命令，查询当前登录用户数量。

## 【2】实验目的

1. 理解有名管道和无名管道的原理和区别；
2. 提高用 C 语言编制程序的能力，熟悉标准库函数 API 接口；
3. 进一步理解、使用和掌握文件的系统调用、文件的标准子例程，能利用和选择这些基本的文件操作完成复杂的文件处理工作。

## 【3】实验环境

- 操作系统：Ubuntu 20.04（Linux）
- Linux 内核版本：5.4.0
- GCC 版本：9.3.0

## 【4】实验原理

### 1. 基本文件操作 API

- **打开文件** `open()`

```
int open(const char *pathname, int flags, mode_t mode);
```

其中，参数 pathname 指向欲打开的文件路径字符串，flag 为打开文件模式，具体包括：O_RDONLY 以只读方式打开文件，O_WRONLY 以只写方式打开文件, O_RDWR 以可读写方式打开文件。上述三种模式互斥，不能同时使用。但上述模式可与以下一些标志使用或运算一起使用。O_CREAT 若打开的文件不存在则自动建立该文件，O_APPEND 表示所写入的数据会以附加的方式加入到文件后面等。

- **关闭文件** `close()`

```
int close(int fd);
```

其中，fd 表示需要关闭文件的文件描述符，调用成功返回 0 ，错误的返回 -1。

- **读文件** `read()`

```
ssize_t read(int fd, void *buf, size_t count);
```

该函数会把参数 fd 所指的文件传送 count 个字节到 buf 指针所指的内存中。若参数 count 为 0，则 read() 不会有作用并返回 0。返回值为实际读取到的字节数，如果返回 0，表示已到达文件尾或是无可读取的数据，文件读写位置会随读取到的字节移动。

- **写文件** `write()`

```
ssize_t write (int fd, const void *buf, size_t count);
```

会把参数 buf 所指的内存写入 count 个字节到参数 fd 所指的文件内。文件读写位置也会随之移动。如果顺利write() 会返回实际写入的字节数。当有错误发生时则返回 -1，错误代码存入 errno 中。

- **打开文件流** `fopen()`

```
FILE *fopen(const char *path, const char *mode);
```

其中，path 字符串包含欲打开的文件路径及文件名，参数 mode 字符串则代表着流形态。mode 有下列几种形态字符串： ”r” 或 ”rb” 表示以只读方式打开文件，该文件必须存在； ”w” 或 ”wb” 表示以写方式打开文件，并把文件长度截短为零。”a” 或 ”ab” 表示以写方式打开文件，新内容追加在文件尾。文件顺利打开后，指向该流的文件指针就会被返回。如果文件打开失败则返回 NULL，并把错误代码存在 errno 中。

- **关闭文件流** `fclose()`

```
int fclose(FILE *stream);
```

其中，stream 为文件流指针。若关文件动作成功则返回 0，有错误发生时则返回 EOF，并把错误代码存到 errno。

- **读文件流** `fread()`

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

其中，buffer 表示用于接收数据的内存地址；size 表示要读写的字节数，单位是字节；count 表示进行读写多少个 size 字节的数据项，每个元素是 size 字节；stream 表示输入流。返回实际读取的元素个数。如果返回值与 count 不相同，则可能文件结尾或发生错误，从 ferror 和 feof 获取错误信息或检测是否到达文件结尾。

- **写文件流** `fwrite()`

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
```

参数意义与 fread() 函数相同，返回实际写入的数据块数目。

- **格式化文件读** `fscanf()`

```
int fscanf(FILE *stream, const char *format, ...);
```

其中，stream 为指向 FILE 对象的指针，format 为格式化说明符，类似 scanf() 函数。

- **格式化文件写** `fprintf()`

```
int fprintf(FILE *stream, const char *format, ...);
```

参数意义与 fscanf() 相同。

- **读字符** `fgetc()`

```
int fgetc(FILE *stream);
```

从文件流中读入一个字符，stream 为指向 FILE 对象的指针。

- **写字符** `fputc()`

```
int fputc(const char *s, FILE *stream);
```

向文件流写入一个字符 s，stream 为指向 FILE 对象的指针。成功返回一个非负整数，出错返回 EOF。

- **读行** `fgets()`

```
char *fgets(char *s, int size, FILE *stream);
```

读取少于 size 长度的字符到字符数组 s，直到新的一行开始或是文件结束。

- **写行** `fputs()`

```
int fputs(const char *s, FILE *stream);
```

**2. 管道通信**

管道是一种进程间通信方式，具有特点：① 管道是单向的、先进先出（FIFO）的，它把一个进程的输出和另一个进程的输入连接在一起；② 一个进程（写进程）在管道的尾部写入数据，另一个进程（读进程）从管道的头部读出数据；③ 数据被一个进程读出后，将被从管道中删除，其它读进程将不能再读到这些数据；④ 管道提供了简单的流控制机制，进程试图读空管道时，进程将阻塞。同样，管道已经满时，进程再试图向管道写入数据，进程将阻塞；⑤ 管道包括**无名管道**和**有名管道**两种，前者用于父进程和子进程间的通信，后者可用于运行于同一系统中的任意两个进程间的通信。

- **无名管道通信 API**

无名管道是半双工的，就是对于一个管道来讲，只能读，或者写。另外，无名管道只能在相关的，有共同祖先的进程间使用（一般用于父子进程）。通过 fork 或者 execve 调用创建的子进程继承了父进程的文件描述符。打开/关闭管道使用 `int pipe(int filedes[2]);` 。如果成功建立了管道，则会打开两个文件描述符， 并把他们的值保存在一个整数数组中。第一个文件描述符用于读取数据，第二个文件描述符用于写入数据。管道的两个文件描述符相当于管道的两端，一端只负责读数据，一端只负责写数据。如果出错返回 -1，同时设置 errno 。读写管道与读写普通文件方式一样，调用 write 与 read 函数即可。注意无名管道是半双工的，不能对一个管道的某一端同时进行读写操作。

- **有名管道通信 API**

不同于无名管道，有名管道创建设备文件，以 FIFO 的形式存在于文件系统中。这样，即使与 FIFO 的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过 FIFO 相互通信。因此，通过 FIFO 不相关的进程也能交换数据。创建有名管道使用 `int mkfifo(const char *pathname, mode_t mode);` 。其中，pathname 表示管道文件名，mode 表示管道的读写模式。可以使用 fopen, fclose, fwrite, fread 等一般文件的操作来读写管道文件。值得注意的是，有名管道也是半双工通信，但通过两个有名管道可以实现全双工通信。

- **管道命令的运行机制**

首先介绍系统调用 `int dup(int fd)` 。它复制一个已存在的文件描述字，返回一个新的文件描述字（当前可用的、最小的文件描述字），这**两个文件描述字共享一个文件的读写指针**。利用 dup 系统调用使用最小的、可用的文件描述字的规则，可以使对任何文件或管道的读写与所希望的文件描述字联系起来。比如，使用 fd 为 0 的标准输入与管道读相联系，fd 为 1 的标准输出与管道写相联系，这就是管道命令的运行机制。如果将文件的输入/输出与标准输入/输出相联系，这就是 I/O 重定向的运行机制。

为了将前一个命令的标准输出与一个管道的写端相连接，后一个命令的标准输入与同一个管道的读端相连接，可以先关闭文件描述字 0 且用 dup 复制读管道的文件描述字，由于 0 是最小的文件描述字，且又刚刚使它处于 “空闲” 可用的状态，故 dup 将返回描述字为 0 的管道的读端。 同样也可使描述字 1 成为管道的写端。 接下来如果再关闭管道文件的原先两个文件描述字， 就构成描述字为 0 和 1 的管道的读写端。

## 【5】实验设计、实现与测试

**1. `read()` / `write()` 实现文件复制（ `copy()` ）**

```
/* file copy based on `read()` and `write()` system call */
void copy(char *indir, char *outdir) {
    int in, out, len;
    char buf[BUFSIZE];

    if ((in = open(indir, O_RDONLY)) == -1) {
        fprintf(stderr, "copy: cannot open source file\n");
        exit(1);
    }
```

```
    if ((out = open(outdir, O_WRONLY|O_CREAT)) == -1) {
        fprintf(stderr, "copy: cannot open target file\n");
        exit(1);
    }

    while ((len = read(in, buf, BUFSIZE)) > 0)
        write(out, buf, len);

    // while ((len = read(in, buf, 1)) > 0)
    //     write(out, buf, 1);

    close(in);
    close(out);
}
```

### 2. `fread()` / `fwrite()` 实现文件复制（`fcopy()`）

```
/* file copy based on `fread()` and `fwrite()` api */
void fcopy(char *indir, char *outdir) {
    FILE *in, *out;
    int len;
    char buf[BUFSIZE];

    if (!(in = fopen(indir, "r"))) {
        fprintf(stderr, "fcopy: cannot open source file\n");
        exit(1);
    }
    if (!(out = fopen(outdir, "w"))) {
        fprintf(stderr, "fcopy: cannot open target file\n");
        exit(1);
    }

    while ((len = fread(buf, sizeof(char), BUFSIZE, in)) > 0)
        fwrite(buf, sizeof(char), len, out);

    // while ((len = fread(buf, 1, 1, in)) > 0)
    //     fwrite(buf, 1, 1, out);

    fclose(in);
    fclose(out);
}
```

### 3. `fscanf()` / `fprintf()` 实现文件复制（`fcopyf()`）

```
/* file copy based on `fscanf()` and `fprintf()` api */
void fcopyf(char *indir, char *outdir) {
    FILE *in, *out;
    char data;

    if (!(in = fopen(indir, "r"))) {
        fprintf(stderr, "fcopyf: cannot open source file\n");
        exit(1);
    }
```

```
    if (!(out = fopen(outdir, "w"))) {
        fprintf(stderr, "fcopyf: cannot open target file\n");
        exit(1);
    }

    while (fscanf(in, "%c", &data) != EOF)
        fprintf(out, "%c", data);

    fclose(in);
    fclose(out);
}
```

### 4. `fgetc()` / `fputc()` 实现文件复制（`fcopyc()`）

```
/* file copy based on `fgetc()` and `fputc()` api */
void fcopyc(char *indir, char *outdir) {
    FILE *in, *out;
    char data;

    if (!(in = fopen(indir, "r"))) {
        fprintf(stderr, "fcopyc: cannot open source file\n");
        exit(1);
    }
    if (!(out = fopen(outdir, "w"))) {
        fprintf(stderr, "fcopyc: cannot open target file\n");
        exit(1);
    }

    while ((data = fgetc(in)) != EOF)
        fputc(data, out);

    fclose(in);
    fclose(out);
}
```

### 5. `fgets()` / `fputs()` 实现文件复制（`fcopys()`）

```
/* file copy based on `fgets()` and `fputs()` api */
void fcopys(char *indir, char *outdir) {
    FILE *in, *out;
    char buf[BUFSIZE];

    if (!(in = fopen(indir, "r"))) {
        fprintf(stderr, "fcopys: cannot open source file\n");
        exit(1);
    }
    if (!(out = fopen(outdir, "w"))) {
        fprintf(stderr, "fcopys: cannot open target file\n");
        exit(1);
    }

    while (fgets(buf, BUFSIZE, in))
        fputs(buf, out);
```

```
    fclose(in);
    fclose(out);
}
```

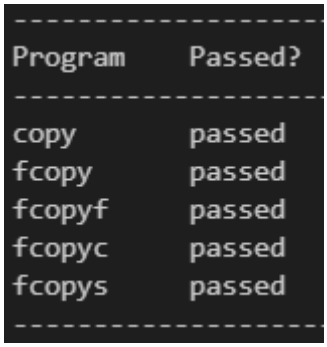
### 6. 测试前五个函数：功能 & 性能

我编写了一个 shell 脚本用于测试以上 5 个复制函数的正确性（功能）和时间开销（性能）。具体来说，待复制的源文件为同一文件 `test.in`，大小约为 10.00 MB。

- **功能测试：**使用 `diff` 命令验证源文件和目标文件是否相同，从而判断正确性。
- **性能测试：**在运行相应复制函数前后，记录时间戳，运行结束后计算差值，重复 20 次该过程取平均值，得到运行时间。
- 以下为测试脚本代码：

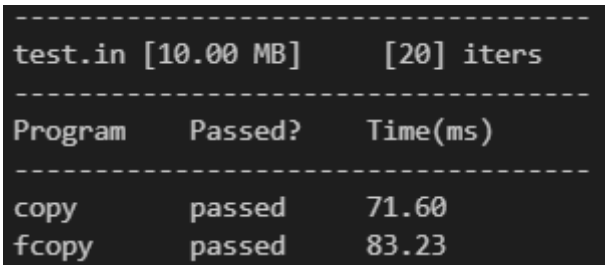
```
#!/bin/bash

name=(copy fcopy fcopyf fcopyc fcopys)
times=20
totalcost=0

make
echo "-------------------------------------"
echo "test.in [10.00 MB]     [$times] iters"
echo "-------------------------------------"
echo "Program    Passed?    Time(ms)"
echo "-------------------------------------"
for (( i=0;i<5;i++ )) do
    # test for correctness
    ./copyfile $i
    diff test/test.in test/test.${name[i]}.out > /dev/null
    if [ $? -eq 0 ]; then
        res=passed
    else
        res=NOTpassed
    fi
    # test for time cost
    totalcost=0
    for (( j=0;j<$times;j++ )) do
        start=$(date +%s%N)
        ./copyfile $i
        end=$(date +%s%N)
        cost=`echo "scale=2; ($end-$start)/1000000" | bc`
        totalcost=`echo "$cost+$totalcost" | bc`
    done;
    totalcost=`echo "scale=2; $totalcost/$times" | bc`
    printf "%-10s %-10s %.2f\n" ${name[i]} $res $totalcost
done;
echo "-------------------------------------"
```

- 功能测试截图与分析

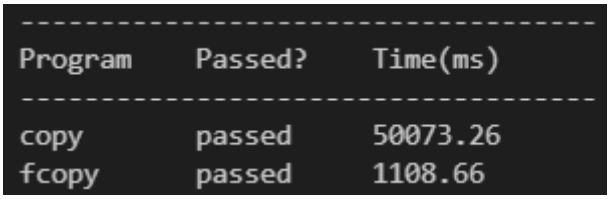



使用 `diff` 命令比较输入文件和输出文件，相同则为 passed ，可见五个函数都成功实现了复制功能。

- 性能测试截图与分析
  - `read()/write()` 参数 `nbytes=1024`，`fread()/fwrite()` 参数 `size=1024, nitems=1`




  - `read()/write()` 参数 `nbytes=1`，`fread()/fwrite()` 参数 `size=1, nitems=1`




- 从以上测试结果可以看出，前者 `read/write` 和 `fread/fwrite` 时间接近，后者 `read/write` 时间远高于 `fread/fwrite` 时间，原因可能是 `fread/fwrite` 会自动分配缓存，在一次读写 1024 字节的情况下缓存带来的效益并不高，甚至因为 `read/write` 是更底层的系统调用运行开销可能更小；而在一次读写 1 字节的情况下缓存带来的效益就比较显著了。
- 该程序的运行时间还受**文件大小**的影响，可以选取不同大小的文件（x 轴）、不同种类的 API 参数（y 轴），多次测量运行时间取平均值（z 轴），在三维空间绘制多对曲线，进行进一步的时间性能分析。还可以从内存开销方面进一步实验。

#### 7. 利用无名管道实现文件复制 & 功能测试

`unnamed-pipe/copyfile.c` 实现：

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

void main() {
    /*
     * pipe file descriptor:
     *   fd_pipe[0]: read
     *   fd_pipe[1]: write
     */
```

```
    int fd_pipe[2];
    pid_t pid;
    char buf[BUFSIZE];
    int len;
    FILE *in, *out;

    if (!(in = fopen("test.in", "r"))) {
        fprintf(stderr, "cannot open source file\n");
        exit(1);
    }
    if (!(out = fopen("test.upipe.out", "w"))) {
        fprintf(stderr, "cannot open target file\n");
        exit(1);
    }

    /* create a one-way communication pipe */
    if (pipe(fd_pipe) == -1) {
        fprintf(stderr, "cannot create pipe\n");
    }

    if (pid = fork()) {
        /*
         * parent process:
         *   read from file and write to pipe
         */
        close(fd_pipe[0]);
        while ((len = fread(buf, sizeof(char), BUFSIZE, in)) > 0)
            write(fd_pipe[1], buf, len);
        close(fd_pipe[1]);
    } else {
        /*
         * child process:
         *   read from pipe and write to file
         */
        close(fd_pipe[1]);
        while ((len = read(fd_pipe[0], buf, BUFSIZE)) > 0)
            fwrite(buf, sizeof(char), len, out);
        close(fd_pipe[0]);
    }
}
```

【功能测试】运行编译好的 `copyfile`，使用 `diff` 命令进行测试，源文件和目标文件相同。

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/unnamed-pipe# diff test.in test.upipe.out
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/unnamed-pipe#
```

#### 8. 利用有名管道实现文件复制 & 功能测试

`named-pipe/proc1.c` 实现（从管道文件读，写入文本文件）：

```
/*
 * read from pipe and write to file
 */
```

```c
#include "copyfile.h"

void main() {
    int fd, len;
    char buf[BUFSIZE];
    FILE *out;

    if (!(out = fopen("test.pipe.out", "w"))) {
        fprintf(stderr, "cannot open target file\n");
        exit(1);
    }

    /* `access`: test for access to named pipe */
    if (access(FIFO, F_OK) == -1) {
        /* inaccessible, and `mkfifo`: create a new pipe */
        if (mkfifo(FIFO, 0666) < 0 && errno != EEXIST) {
            fprintf(stderr, "cannot create pipe\n");
            exit(1);
        }
    }

    // open named pipe (read only)
    if ((fd = open(FIFO, O_RDONLY)) == -1) {
        fprintf(stderr, "cannot open pipe file\n");
        exit(1);
    }

    while((len = read(fd, buf, BUFSIZE)) > 0)
        fwrite(buf, sizeof(char), len, out);

    close(fd);
    unlink(FIFO);
}
```

`named-pipe/proc2.c` 实现（从文本文件读，写入管道文件）：

```c
/*
 * read from file and write to pipe
 */

#include "copyfile.h"

void main() {
    int fd, len;
    char buf[BUFSIZE];
    FILE *in;

    if (!(in = fopen("test.in", "r"))) {
        fprintf(stderr, "cannot open source file\n");
        exit(1);
    }

    // open named pipe (write only)
    if ((fd = open(FIFO, O_WRONLY)) == -1) {
        fprintf(stderr, "cannot open pipe file\n");
```

```
        exit(1);
    }

    while((len = fread(buf, sizeof(char), BUFSIZE, in)) > 0)
        write(fd, buf, len);

    close(fd);
}
```

【功能测试】编译 `proc1.c` 和 `proc2.c`，得到 `proc1` 和 `proc2`。先运行 `proc1`（管道文件为空，所以该进程会阻塞，等待 `proc2` 往管道文件写入数据），再运行 `proc2`，之后两个进程结束。使用 `diff` 命令进行测试，源文件和目标文件相同。

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/named-pipe# diff test.in test.pipe.out
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/named-pipe#
```

**9. 利用管道通信实现命令：`who | wc -l`**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <sys/wait.h>

void main() {
    int pipedes[2];
    pid_t pid;

    // create pipe
    if (pipe(pipedes) == -1) {
        perror("cannot create pipe\n");
        exit(1);
    }

    // create child process 1
    while ((pid = fork()) == -1);
    if (pid == 0) {
        /* child process 1 */
        // close file descriptor 1 (stdout)
        if (close(1) == -1) {
            perror("cannot close stdout\n");
            exit(1);
        }
        // duplicate write pipe descriptor and associate with stdout
        if (dup(pipedes[1]) != 1) {
            perror("cannot duplicate write pipe descriptor\n");
            exit(1);
        }
        // close the original pipe descriptors inherited from parent
        if (close(pipedes[0]) == -1 || close(pipedes[1]) == -1) {
```

```
            perror("cannot close original pipe descriptors\n");
            exit(1);
        }
        execlp("who", "who", NULL);
    }

    // create child process 2
    while ((pid = fork()) == -1);
    if (pid == 0) {
        /* child process 2 */
        // close file descriptor 0 (stdin)
        if (close(0) == -1) {
            perror("cannot close stdin\n");
            exit(1);
        }
        // duplicate read pipe descriptor and associate with stdin
        if (dup(pipedes[0]) != 0) {
            perror("cannot duplicate read pipe descriptor\n");
            exit(1);
        }
        // close the original pipe descriptors inherited from parent
        if (close(pipedes[0]) == -1 || close(pipedes[1]) == -1) {
            perror("cannot close original pipe descriptor\n");
            exit(1);
        }
        execlp("wc", "wc", "-l", NULL);
    }

    if (close(pipedes[0]) == -1 || close(pipedes[1]) == -1) {
        perror("cannot close pipe descriptors\n");
        exit(1);
    }
    while (wait(NULL) != -1);
}
```

【功能测试】编译运行程序 `whowc.c` ，再运行 `who | wc -l` ，两者结果一致。

```
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/who-wc# who | wc -l
1
root@iZuf63xs8u1971bor8zpc1Z:~/os/lab3/who-wc# ./whowc
1
```

## 【6】实验心得

本次实验总体上难度较低，但在代码编写过程中，会存在对 API 不熟悉而导致使用错误的情况，花费较多时间才得以调试成功。经过本次实验的实践，我对文件系统的操作 API 有了较为全面的了解，并通过性能对比，对多种读写机制的内在原理有了深入理解。另外，我还了解了管道通信的基本原理和相关 API 的使用。感谢老师上课的指导和帮助，让我能够顺利完成本次实验，受益良多！

源代码 GitHub 地址：https://github.com/zhliuworks/OS-lab/tree/master/Lab3