

# 操作系统专题实验报告

班级： 计算机 001

学号： 2203613040

姓名： 曾锦程

2022 年 12 月 08 日

## 目录

实验一 OpenEuler 系统环境实验	6
1.1 实验目的	6
1.2 实验内容	6
1.3 实验思想	6
1.4 实验步骤	7
1.4.1 进程相关编程实验	7
第一步（基本要求）	7
第二步（扩展部分）	9
1.4.2 线程相关编程实验	11
函数介绍	11
实验过程	12
1.5 测试数据设计	16
1.6 程序运行初值及运行结果分析	16
1.7 实验总结	16
1.7.1 实验中的问题及解决过程	16
1.7.2 实验收获	17
1.7.3 意见与建议	17
1.8 附件	17
1.8.1 附件 1 程序	17
1.8.2 附件 2 readme	17
实验二 Linux 进程通信与内存管理实验	17
2.1 实验目的	17
2.2 实验内容	18
2.3 实验思想	18
2.4 实验步骤	21
2.4.1 进程的软中断通信	21

2.4.2 进程的管道通信	32
2.4.3 页面置换	39
2.5 测试数据设计	42
2.6 程序运行初值及运行结果分析	42
2.7 页面置换算法复杂度分析	42
2.8 回答问题	43
2.8.1 软中断通信	43
2.8.2 管道通信	44
2.9 实验总结	44
2.9.1 实验中的问题及解决过程	44
2.9.2 实验收获	45
2.9.3 意见与建议	45
2.10 附件	45
2.10.1 附件 1 程序	45
2.10.2 附件 2 Readme	45
实验三 类 EXT2 文件系统设计	45
3.1 实验目的	45
3.2 实验内容	45
3.3 实验思想	45
3.4 实验步骤	57
3.4.1 EXT2 文件系统结构	57
3.4.2 文件系统设计介绍	59
3.5 程序运行初值及运行结果分析	64
3.6 实验总结	64
3.6.1 实验中的问题及解决过程	64
3.6.2 实验收获	64
3.6.3 意见与建议	65
3.7 附件	65

3.7.1 附件 1 程序.....	65
3.7.1 附件 2 Readme.....	65

# 实验一 OpenEuler 系统环境实验

## 1.1 实验目的

熟悉 Linux 操作系统运行环境，进行进程线程相关编程实验

## 1.2 实验内容

### 一、进程相关编程实验

(1) 熟悉操作命令、编辑、编译、运行程序。完成操作系统原理课程教材 P103 作业 3.7（采用图 3-32 所示的程序）的运行验证，多运行程序几次观察结果；去除 wait 后再观察结果并进行理论分析。

(2) 扩展课本上的程序：a) 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释，同时输出两种变量的地址观察并分析；b) 在 return 前增加对全局变量的操作并输出结果，观察并解释；c) 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数执行自己写的一段程序，在此程序中输出进程 PID 进行比较并说明原因；

### 二、线程相关编程实验

1、在进程中给一变量赋初值并创建两个线程；

2、在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果；

3、多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；

4、将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

## 1.3 实验思想

### 一、进程相关编程实验

主要是通过 fork() 函数来生成子进程，然后通过各种输出，全局变量操作等等，来对父子进程关系观察，其中用到 wait()、getpid()、getppid()、system() 等函数

## 二、线程相关编程实验

主要是引入线程库，在 linux 中调用线程，使用 `pthread_create()`、`pthread_join()` 来完成的线程的创建和结束操作，其中也是调用上面进程类似函数对进程中多个线程进行观察

### 1.4 实验步骤

#### 1.4.1 进程相关编程实验

实验过程：

##### 第一步（基本要求）

将课本上图 3-32 程序代码原封不动地写在程序 `work_1.c` 中，通过 linux 中 `gcc` 命令进行编译并运行：

```
gcc work_1.c -o work_1
./work_1
```

`fork()` 函数会给父进程返回子进程号，而给子进程返回 0，进程号输出如下几张图。最开始我是使用课本原本的代码运行，结果如下：

```
[root@VM-8-12-centos operating_system_project]# ./work_1
child: pid = 0child: pid = 26164parent: pid = 26164parent: pid1 = 26163[root@VM-8-12-centos operating_syste
[root@VM-8-12-centos operating_system_project]# ./work_1
child: pid = 0child: pid = 26229parent: pid = 26229parent: pid1 = 26228[root@VM-8-12-centos operating_system_project]# ./work_1
child: pid = 0child: pid = 26234parent: pid = 26234parent: pid1 = 26233[root@VM-8-12-centos operating_system_project]# ./work_1
child: pid = 0child: pid = 26240parent: pid = 26240parent: pid1 = 26239[root@VM-8-12-centos operating_system_project]#
```

可以看出多次都是 `child` 进程先输出而后父进程输出 由于一开始有点看不习惯就把代码输出了换行符 `\n`，但是发现结果如下：

```
parent: pid = 27542
parent: pid1 = 27541
child: pid = 0
child: pid = 27542
```

加上换行符后，结果居然变成了父进程先输出，后经过上网搜索发现，`printf` 函数中如果没有加入结束符 `\n` 则不会直接输出，而是等进程结束后再输出，后仔细查看代码后发现，书上的代码相当与父进程使用 `wait` 函数等待子进程结束之后再结束进程，所以有了第一次的运行结果，而第二张图是由于父进程本身先运行导致父进程先执行 `printf` 而先输出！

于是之后使用 `work_2.c` 进行验证，`work_2.c` 中把 `work_1.c` 中代码删除 `wait()` 函数，运行结果为：

```

parent: pid = 29255parent: pid1 = 29254child: pid = 0child: pid = 29255[root@VM-8-12-centos opeating_system_project]# ./work_2
parent: pid = 29260parent: pid1 = 29259[root@VM-8-12-centos opeating_system_project]# child: pid = 0child: pid = 29260./work_2
parent: pid = 29265parent: pid1 = 29264child: pid = 0child: pid = 29265[root@VM-8-12-centos opeating_system_project]# ./work_2
parent: pid = 29275parent: pid1 = 29274child: pid = 0child: pid = 29275[root@VM-8-12-centos opeating_syste

```

结果发现是父进程先结束，与上述想法一样，父进程先运行先结束，实际上此时的子进程就变成了所谓的孤儿进程？（父进程先结束而子进程还没有结束）于是尝试使用 work\_2\_2.c 验证父进程是否先结束，即 return 0 前面加一行父进程号的输出，结果如下：

```

[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 4609parent: pid1 = 4608ppid = 18563child: pid = 0child: pid = 4609ppid = 4608[root@VM-8-12-centos opeating_system_project]#
./work_2_2
parent: pid = 4614parent: pid1 = 4613ppid = 18563child: pid = 0child: pid = 4614ppid = 4613[root@VM-8-12-centos opeating_system_project]#
./work_2_2
parent: pid = 4624parent: pid1 = 4623ppid = 18563child: pid = 0child: pid = 4624ppid = 4623[root@VM-8-12-centos opeating_system_project]#

```

但是发现子进程的 ppid 依然是父进程，说明父进程并没有结束，子进程并不是孤儿进程，考虑是不是进程加载速度的原因？随后再对 work\_2\_2.c 进行修改，把 wait(NULL) 加在子进程尾部，发现输出如下：

```

parent: pid = 7050parent: pid1 = 7050ppid = 7050child: pid = 0child: pid = 7050ppid = 1[root@VM-8-12-centos
[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8068parent: pid1 = 8067ppid = 7603child: pid = 0child: pid = 8068ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8073parent: pid1 = 8072ppid = 7603child: pid = 0child: pid = 8073ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8078parent: pid1 = 8077ppid = 7603child: pid = 0child: pid = 8078ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8088parent: pid1 = 8087ppid = 7603child: pid = 0child: pid = 8088ppid = 8087[root@VM-8-12-centos opeating_system_project]#

```

此时是父进程已经结束，而子进程变成孤儿进程，但是被挂在 1 号进程上，由于 1 号进程回收子进程资源，但是后面发现子进程中调用 wait 函数实际上是让子进程等待父进程，这样验证不正确。于是改成再子进程中加循环，让子进程更慢结束，发现子进程 ppid 有时候是父进程号，有时候是 1 号进程，这里就不太懂了，不知道父进程 return 后到底有没有结束，无法理解。个人猜想是父进程 return 后并没有马上结束，而是等待一段时间再结束，如果没有等到，则自行结束

```

parent: pid = 7050parent: pid1 = 7050ppid = 7050child: pid = 0child: pid = 7050ppid = 1[root@VM-8-12-centos
[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8068parent: pid1 = 8067ppid = 7603child: pid = 0child: pid = 8068ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8073parent: pid1 = 8072ppid = 7603child: pid = 0child: pid = 8073ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8078parent: pid1 = 8077ppid = 7603child: pid = 0child: pid = 8078ppid = 1[root@VM-8-12-centos opeating_system_project]# ./work_2_2
parent: pid = 8088parent: pid1 = 8087ppid = 7603child: pid = 0child: pid = 8088ppid = 8087[root@VM-8-12-centos opeating_system_project]#

```

查找期间发现两个概念为僵尸进程和孤儿进程：

- 孤儿进程即父进程先结束了子进程处于死循环中等等没有结束，导致父进程先结束，正如上面测试同理，然而这样问题比较小，因为 linux 系统会将子进程挂在 1 号初始进程中，让 1 进程回收（1 进程永远不会结束）

- 僵尸进程即子进程结束但是父进程处于运行状态或者睡眠状态(两种 S 和 D)等等，导致父进程无法回收子进程，则导致子进程无法释放，这样问题很大，子进程内容会一直存在

## 第二步（扩展部分）

根据实验要求(a)添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释，同时输出两种变量的地址观察并分析；

于是在 work\_3.c 中加入全局变量 `int i = 0`，同时为了输出结果方便阅读，将程序中 `printf` 中均加了换行符，在父进程操作后加了 `wait()` 函数保证子进程一定比父进程先结束（安全），输出结果如下：

```
[root@VM-8-12-centos opeating_system_project]# gcc work_3.c -o work_3
[root@VM-8-12-centos opeating_system_project]# ./work_3
操作前i = 0
i = -1
i的虚拟地址是5AB4
parent: pid = 21434
parent: pid1 = 21433
操作前i = 0
i = 1
i的虚拟地址是5AB4
child: pid = 0
child: pid = 21434
```

结果分析如下：可以看出 `fork()` 之后子进程继承了父进程的各种资源，比如全局变量 `i`，可以明显通过父子进程操作(`i--`和 `i++`)和输出看出父子进程中 `i` 是两个变量，相当于是父进程子进程各对自己进程的 `i` 变量进行操作，互不干扰 而对于地址来说，两个 `i` 的虚拟地址相同，但从输出结果来看两个 `i` 并不是存储在同一个位置的 `i`，查阅后发现，是 `os` 的内存管理机制，让虚拟地址虽然相同但却映射到不同的物理地址上

根据实验要求(b)，在 `return` 前增加对全局变量的操作并输出结果,观察并解释;

同时 `return` 前加了对 `i` 的操作 (`i--`，为了父子进程区分)和输出, 输出结果如下:



```

● [root@MM-8-12-centos opeating_system_project]# gcc work_4.c -o work_4
● [root@MM-8-12-centos opeating_system_project]# ./work_4
i = -1
i的虚拟地址是F534
parent: pid = 26902
parent: pid1 = 26901
i = 1
i的虚拟地址是F534
child: pid = 0
child: pid1 = 26902
return前的i = 0
return前的i = -2

```

结果分析如下: return 前操作为  $i--$ , 父进程在 else 代码块中等待子进程结束的信号, 所以子进程先运行到 return, 所以先输出子进程的  $i=0$  (子进程中  $i=1$ ), 后父进程收到信号后继续运行, 所以后输出父进程的  $i=-2$  (父进程中  $i=-1$ )

根据实验要求(c), 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数执行自己写的一段程序, 在此程序中输出进程 PID 进行比较并说明原因;

自己写的程序 helloworld 如下:

```

#include<stdio.h>
#include <unistd.h>
int main(){
    printf("hello world!\n");
    printf("helloworld 中 pid = %d\n", getpid());
    printf("helloworld 的 ppid = %d\n", getppid());
    return 0;
}

```

(1)使用 system 函数 即 work\_5.c, 其中使用 system('./helloworld')来调用 helloworld 程序, 写在子进程 else 代码块中, 输出结果如下:

```

● [root@VM-8-12-centos opreating_system_project]# ./work_5
parent: pid = 32292
parent: pid1 = 32291
child: pid = 0
child: pid1 = 32292
hello world!
helloworld的pid = 32293
helloworld的ppid = 32292
○ 是否运行? [root@VM-8-12-centos opreating_system_project]#

```

结果分析如下：

可以看出 system 调用后，依然是继续执行子进程，而不是覆盖子进程运行(子进程后面输出了"是否运行? ")，实际上 system 函数相当于 fork()+exec()，并不会覆盖子进程，而是创建新的子进程(输出中被调用进程的 ppid 为原进程 id)并用 exec 覆盖运行，运行结束回到原进程继续运行

(2)使用 exec 族函数 即 work\_6.c, 其中使用 execl 函数执行已经编译好的 helloworld 代码 输出结果如下:

```

● [root@VM-8-12-centos opreating_system_project]# ./work_6
parent: pid = 32549
parent: pid1 = 32548
child: pid = 0
child: pid1 = 32549
hello world!
helloworld的pid = 32549
helloworld的ppid = 32548

```

结果分析如下： 可以看出调用 helloworld 程序后，helloworld 的进程号为子进程进程号，父进程号为原本的父进程进程号，即 exec 族函数是将调用的程序覆盖子进程运行(work\_6.c 在调用后又有一行输出，但是并没有输出)

## 1.4.2 线程相关编程实验

预备知识

对于含有线程的程序，要编译需使用如下命令：

```
gcc work_7.c -lpthread -o work_7
./work_7
```

要使用线程需要:

```
#include<pthread.h>
```

## 函数介绍

```
pthread_create(pthread_t* idpoint, NULL, (void*)function, input)
///课本上介绍了软件实现线程主要是以调用函数的方式进行, 所以该函数几个项分别是
///第一项是线程 id 地址, 第二项为线程属性(一般为 NULL),第三项为函数地址
///第四项为函数输入值
pthread_join(pthread_t id, NULL)
///主要是用来结束线程,第一项为线程 id 号,第二项为储存线程结束状态, 即储存线程函数
///return 值
pthread_mutex_t mutex; //定义互斥信号量的方法
pthread_mutex_init(pthread_mutex_t* mutexpoint, NULL)//对互斥信号量初始化
///第一个参数为信号量的地址, 第二参数为属性, 一般为 NULL
pthread_mutex_destroy(pthread_mutex_t* mutexpoint);//摧毁互斥信号量
///init 实际上是一个类似 malloc 的过程, 需要 destroy 函数来进行摧毁, 进程结束并不会释放
pthread_mutex_lock(&mtx);
//获取锁
pthread_mutex_unlock(&mtx);
//释放锁
```

## 实验过程

- 1、在进程中给一变量赋初值并创建两个线程；
- 2、在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果；
- 3、多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；

于是编写程序 work\_7.c,输出结果如下:

```
● [root@VM-8-12-centos opreating_system_project]# gcc work_7.c -lpthread -o work_7
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -5000
i = 0
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -5000
i = 0
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -5000
i = 0
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -5000
i = 0
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -2534
i = 129
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -5000
i = 0
● [root@VM-8-12-centos opreating_system_project]# ./work_7
i = -4238
i = 31
○ [root@VM-8-12-centos opreating_system_project]# ./work_7
```

结果分析：

创建线程后两个线程同时对 i 变量进行操作（线程之间共享数据区），由于同时操作，就无法确定同一时刻是哪个线程在运行操作，就会出现第五个和第七个的输出结果

会这样的原因是此时 i 是互斥资源,应该对 i 做互斥，于是程序修改如 work\_8.c,输出结果如下：

```
• [root@VM-8-12-centos opeating_system_project]# gcc work_8.c -lpthread -o work_8
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
• [root@VM-8-12-centos opeating_system_project]# ./work_8
i = 5000
i = 0
```

结果分析：

实际上是通过改变函数 `pthread_join` 的位置来实现的，是通过确认两个线程的同步关系，来保证 `i` 只有一个线程在操作。通过输出关系来看实现成功

同时使用互斥锁尝试，修改代码为 `work_9.c`，输出结果如下：

```

● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
● [root@VM-8-12-centos opeating_system_project]# ./work_9
i = -5000
i = 0
○ [root@VM-8-12-centos opeating_system_project]# ./work_9

```

结果分析：

使用了互斥锁，让对 i 的操作和输出变为临界区资源，让其只能同一时刻只有一个线程操作 5000 次，也实现了实验目标

4、将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

于是编写代码 work\_10.c，使用 system 函数调用 helloworld，输出结果如下：

```

● [root@VM-8-12-centos opeating_system_project]# gcc work_10.c -lpthread -o work_10
● [root@VM-8-12-centos opeating_system_project]# ./work_10
hello world!
helloworld的pid = 21937
helloworld的ppid = 21935
threadone
pid = 21936
ppid = 9993
tid = 933345024
Main thread
pid = 21935
ppid = 9993
tid = 941721408

```

结果分析：

在子线程中调用函数，结果 helloworld 进程 ppid 为原进程的 pid，这个合理，但是 threadone 的 pid 竟然和原本进程不一样，ppid 为两个线程共同的父进程号，经过查阅资料发现，linux 中的线程是通过进程模拟而来，所以新建线程实际上是新建了一个进程，对资源共享等等是模拟而来。tid 为正常情况

编写代码 work\_11.c，使用 execv 函数调用 helloworld，输出结果如下：

```
● [root@VM-8-12-centos opeating_system_project]# ./work_11
threadone
pid = 19241
ppid = 9993
tid = 1411905280
hello world!
helloworld的pid = 19240
helloworld的ppid = 9993
```

结果分析：

- (1) 可以看出 ppid 仍为 9993，可能 9993 的进程专门用来线程模拟（多次运行父进程均为 9993)
- (2) 分析下面的代码可以看出，调用 execv 后，对下面的线程也覆盖，让整个进程被 helloworld 覆盖运行，符合线程原理，多线程处于同一个进程中。但实际上 threadone 的进程号应该和原进程不一样，所以覆盖运行也是模拟而来。（也许是通过返回原进程进程号而实现;可以从 helloworld 的父进程变为 9993 看出是模拟而来)
- (3) tid 为正常情况

## 1.5 测试数据设计

第一部分中全局变量的操作设计成父进程为 i--,子进程为 i++,让两个进程的值区分开，同时 return 前进行 i--操作，让 return 前 i 值不同，根据输出顺序可以发现父子进程的运行顺序，更好的了解父子进程原理。并且在 system()后面和 exec 族函数的后面再写了一行输出，来判断是否存在覆盖操作

第二部分中线程做了 5000 次 i++操作，主线程做了 5000 次 i--操作，这样让不做同步互斥，

和做了同步互斥有着明显区别，而且这里只创建了一个线程，因为本身进程的过程也是一个线程，所以没有再创建线程。并且在 `system()` 后面和 `exec` 族函数的后面再写了一行输出，来判断是否存在覆盖操作

其中还让被调用执行的程序输出 `pid` 与 `ppid`，让进程之间的关系更加清晰

## 1.6 程序运行初值及运行结果分析

运行与结果分析为了便于理解，均附于 1.4 中

## 1.7 实验总结

### 1.7.1 实验中的问题及解决过程

实验中问题主要是机制和函数的使用问题。

进程实验中缓冲区的问题以及 `system()` 函数的调用，`execv()` 函数的调用等等，最后均通过 ppt 中附带的网址，自行查找 csdn 来理解

线程实验中线程创建与结束，互斥锁的实现，线程编译所需要专门 link 的库和 linux 系统通过进程模拟线程等等，也是通过自行查找 csdn 来了解知识（readme 中自己写了预备知识）

### 1.7.2 实验收获

书上的原理是一套，但是实际上操作系统怎么实现而各有区别。

原理和实践结合，让我对操作系统有了更深的理解

### 1.7.3 意见与建议

希望可以专门上课讲一些 linux 的基本操作命令等等，感觉自行查找相对于系统讲解的效率要差很多



## 1.8 附件

### 1.8.1 附件 1 程序

### 1.8.2 附件 2 readme

## 实验二 Linux 进程通信与内存管理实验

### 2.1 实验目的

本实验在用户态下，根据教材所学习的操作系统原理，完成 Linux 下进程通信与内存管理算法的实现，通过实验，进一步理解所学理论知识。

#### (1) 进程的软中断通信

编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在 POSIX 规范中系统调用的功能和使用。

#### (2) 管道通信

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制。

#### (3) 页面的替换

模拟实现 FIFO 算法，LRU 算法

### 2.2 实验内容

1. 进程软中断与管道通信（必做）

2. 页面置换（选做）

### 2.3 实验思想

1. UNIX 相关系统调用介绍

## fork(): 创建一个子进程

(1) 创建的子进程是 fork 调用者进程(即父进程)的复制品,即进程映像.除了进程标识数以及与进程特性有关的一些参数外,其他都与父进程相同,与父进程共享文本段和打开的文件,并都受进程调度程序的调度.

(2) 如果创建进程失败,则 fork()返回值为-1,若创建成功,则从父进程返回值是子进程号,从子进程返回的值是 0

## exec(): 装入并执行相应文件

(1) 因为 FORK 会将调用进程的所有内容原封不动地拷贝到新创建的子进程中去,而如果之后马上调用 exec,这些拷贝的东西又会马上抹掉,非常不划算.于是设计了一种叫作“写时拷贝”的技术,使得 fork 结束后并不马上复制父进程的内容,而是到了真正要用的时候才复制

wait():父进程处于阻塞状态,等待子进程终止,其返回值为所等待子进程的进程号

exit():进程自我终止,释放所占资源,通知父进程可以删除自己,此时它的状态变为 P\_state= SZOMB,即僵死状态.如果调用进程在执行 exit 时其父进程正在等待它的中止,则父进程可立即得到该子进程的 ID 号

## getpid():获得进程号

lockf(files,function,size):用于锁定文件的某些段或整个文件。本函数适用的头文件为: #include<unistd.h>

参数定义: int lockf(files,function,size);

int files,function;

long size;

files 是文件描述符, function 表示锁状态, 1 表示锁定, 0 表示解锁; size 是锁定或解锁的字节数, 若为 0 则表示从文件的当前位置到文件尾。

**kill(pid,sig): 一个进程向同一用户的其他进程 pid 发送一中断信号**

**signal(sig,function): 捕捉中断信号 sig 后执行 function 规定的操作**

头文件为: #include <signal.h>

参数定义: signal(sig,function)

int sig;

void (\*func) ();

其中 sig 共有 19 个值

值	名字	说明
01	SIGHUP	挂起
02	SIGINT	中断, 当用户从键盘键入“del”键时
03	SIGQUIT	退出, 当用户从键盘键入“quit”键时
04	SIGILL	非法指令
05	SIGTRAP	断点或跟踪指令
06	SIGIOT	IOT指令
07	SIGEMT	EMT指令
08	SIGFPE	浮点运算溢出
09	SIGKILL	要求终止进程
10	SIGBUS	总线错误
11	SIGSEGV	段违例, 即进程试图去访问其地址空间以外的地址
12	SIGSYS	系统调用错
13	SIGPIPE	向无读者的管道中写数据
14	SIGALARM	闹钟
15	SIGTERM	软件终止
16	SIGUSR1	用户自定义信号
17	SIGUSR2	用户自定义信号
18	SIGCLD	子进程死
19	SIGPWR	电源故障

## pipe(fd)

```
int fd[2];
```

其中 fd[1]是写端，向管道中写入，fd[0]是读端，从管道中读出。

## 暂停一段时间 sleep;

调用 sleep 将在指定的时间 seconds 内挂起本进程。

其调用格式为：“unsigned sleep(unsigned seconds);”；返回值为实际的挂起时间。

## 暂停并等待信号 pause;

调用 pause 挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。

其调用格式为“int pause(void);”

## 2.4 实验步骤

### 2.4.1 进程的软中断通信

编制实现软中断通信的程序

使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号（即按 delete 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 kill()向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

```
Child process 1 is killed by parent !!  
Child process 2 is killed by parent !!
```

父进程调用 wait()函数等待两个子进程终止后，输入以下信息，结束进程执行：

```
Parent process is killed!!
```

多运行几次编写的程序，简略分析出现不同结果的原因。

1、根据流程图编写程序，猜想一下这个程序的运行结果，然后多次运行，观察 Delete/quit 键

```

Parent process is killed !!
⊗ [root@VM-8-12-centos operating_system_2]# ./work_1_1

Child process 2 is killed by parent !!

Child process 1 is killed by parent !!
^C
● [root@VM-8-12-centos operating_system_2]# ./work_1_1

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!
^\\
Parent process is killed !!
● [root@VM-8-12-centos operating_system_2]# ./work_1_1

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

Parent process is killed !!

```

前后，会出现什么结果？分析原因。

根据实验辅导书的参考代码，写成 work\_1\_1.c，多次运行结果如下：

结果分析：

第一次是通过 ctrl + c 来输入 delete 信号来让整个父进程直接退出，不输出父进程被杀死信号。第二次是通过输入 ctrl + \来输入 quit 信号来让父进程从 sleep 状态中唤醒，因为代码中接收 3 信号，从而父进程输出（子进程不是因为该信号杀死是因为课本中代码根本不是通过信号量杀死，而是自行结束，与父进程无关，且早于父进程），第三次输出是父进程自己从 sleep 状态中唤醒，从而输出与第二次输出一样的效果。并且从第一次和第二三次输出子进程顺序不一，可以看出两者顺序并不固定。

为了验证上述结果分析中括号中内容，写代码 work\_1\_2.c 验证，把 kill 直接删除，输出结果如下：

```

● [root@VM-8-12-centos operating_system_2]# ./work_1_2

Child process 2 is killed by parent !!

Child process 1 is killed by parent !!

Parent process is killed !!

```

结果分析： 结果仍相同，成功验证上述内容

2、如果程序运行界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”，怎样修改程序使得只有接收到相应的中断信号后再发生跳转，执行输出？

对 work\_1\_1.c 中代码进行修改(work\_1\_3.c)，对子进程均加入该语句：

```
while(wait_flag);
```

让进程中只有运行 stop 函数，即 wait\_flag = 0，才能运行下一步，使用忙等实现

输出结果如下：

```

● [root@VM-8-12-centos operating_system_2]# ./work_1_3
^\\
Child process 2 is killed by parent !!

Child process 1 is killed by parent !!

Parent process is killed !!
● [root@VM-8-12-centos operating_system_2]# ./work_1_3
^\\
Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

Parent process is killed !!

```

```

• [root@MM-8-12-centos operating_system_2]# ./work_1_3

Child process 2 is killed by parent !!

Child process 1 is killed by parent !!

Parent process is killed !!

```

结果分析：可以看出修改后子进程要通过接收信号才能结束，并输出。但是这里有一个问题，由于 `signal(3, stop)` 语句为父子进程共享，也就是说实际上输入 quit 信号后，子进程本身就会结束，而不是父进程的 kill 信号导致结束！然而如果是等待 5s 后的中断，此时为父进程发出的 kill 导致子进程结束。

为了验证上述分析，写代码 `work_1_4.c`，将代码中 `stop` 修改如下：

```

void stop(int signum){
    wait_flag = 0;
    printf("\n %dstop \n",signum);
}

```

输出结果如下：

```

• [root@MM-8-12-centos operating_system_2]# ./work_1_4
^\\

3 stop
3 stop

17 stop

Child process 2 is killed by parent !!

16 stop

3 stop

Child process 1 is killed by parent !!

Parent process is killed !!

```

```

● [root@VM-8-12-centos operating_system_2]# ./work_1_4
^\\
3 stop

3 stop
^\\
3 stop

Child process 2 is killed by parent !!

3 stop

Parent process is killed !!

```

```

● [root@VM-8-12-centos operating_system_2]# ./work_1_4
^\\
3 stop

17 stop

3 stop

Child process 2 is killed by parent !!

16 stop

3 stop

Child process 1 is killed by parent !!

Parent process is killed !!

```

```

● [root@VM-8-12-centos operating_system_2]# ./work_1_4

16 stop

Child process 1 is killed by parent !!

17 stop

Child process 2 is killed by parent !!

Parent process is killed !!

```



结果分析：

可以发现程序运行有三种结果，第一种是子进程一接收 quit 信号终止，第二种是子进程一二均是 kill 发出的信号终止，第三种是都收到 quit 信号终止。说明如果不做信号屏蔽子进程会收到 quit 信号，无法确定是否是 kill 发出的信号才发生跳转。而 5s 后的中断可以发现只能是 kill 发出的中断信号，因为此时没有 quit 信号。

为了实现真正的只有收到相应的信号才发生跳转，且子进程结束有固定顺序，写 work\_1\_5.c，代码如下：

```

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

int wait_flag;
void stop(int signal);

int main(){
    int pid1, pid2;
    signal(3, stop);
    while((pid1 = fork()) == -1);
    if(pid1 > 0){
        while((pid2 = fork()) == -1);
        if(pid2 > 0){
            wait_flag = 1;
            sleep(5);
            printf("\nkill 1!\n");
            kill(pid1, 16);
            wait(0);
            printf("\nkill 2!\n");
            kill(pid2, 17);
            wait(0);
            printf("\n Parent process is killed !!\n");
            exit(0);
        }
        else {
            signal(3, SIG_IGN);
            wait_flag = 1;
            signal(17, stop);
            while(wait_flag);
            printf("\n Child process 2 is killed by parent !!\n");
            exit(0);
        }
    }
    else {
        signal(3, SIG_IGN);
        wait_flag = 1;
        signal(16, stop);
        while(wait_flag);
        printf("\n Child process 1 is killed by parent !!\n");
        exit(0);
    }
    return 0;
}

void stop(int signal){
    wait_flag = 0;
    printf("\n %d stop \n", signal);
}

```

为了验证上述分析，写代码 work\_1\_4.c，将代码中 stop 修改如下：

```

void stop(int signal){
    // 定义两个进程号变量
    // 或者 signal(14, stop);
    wait_flag = 0;
    printf("\n %d stop \n", signal);
}

```

输出结果如下：

结果分析：

可以发现程序运行有三种结果，第一种是子进程一接收 quit 信号终止，进程一二均是 kill 发送的信号，进程二结束的信号都收到 quit 信号终止，而 5s 后的中断可以发现只能是 kill 发送的结束信号，因为此时没有

为了实现真正的只有收到相应的信号才发生跳转，且子进程结束有

work\_1\_5.c，代码如下：

输出结果如下：

进程的管道通信

输出结果如下：

```
● [root@VM-8-12-centos operating_system_2]# ./work_1_5
^\\
3 stop

kill 1!

16 stop

Child process 1 is killed by parent !!

kill 2!

17 stop

Child process 2 is killed by parent !!

Parent process is killed !!
```

```
● [root@VM-8-12-centos operating_system_2]# ./work_1_5

kill 1!

16 stop

Child process 1 is killed by parent !!

kill 2!

17 stop

Child process 2 is killed by parent !!

Parent process is killed !!
```

如图可以看出程序运行顺序为，父进程先接收到信号或者等待 5s 后中断，然后发出 kill 信号，之后子进程收到后才结束，最后父进程 wait 函数等到子进程结束再结束，且子进程两个信号区分开，用 wait(0) 改变来实现顺序结束，完成实验要求。

3、将本实验中通信产生的中断通过 14 号信号值进行闹钟中断，将 signal(3,stop) 当中数字信号变为 2，体会不同中断的执行样式，从而对软中断机制有一个更好的理解。

编写代码 work\_1\_6.c, 代码如下:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
int wait_flag;
void stop(int signum);
int main(){
    int pid1, pid2;
    signal(2,stop);
    signal(14,stop);
    while((pid1 = fork( )) == -1);
    if(pid1 > 0) {
        while((pid2 = fork( )) == -1);
        if(pid2 > 0) {
            wait_flag = 1;
            alarm(5);
            while(wait_flag);
            kill(pid1,14);
            kill(pid2,14);
            printf("\nkill!\n");
            wait(0);
            wait(0);
            printf("\n Parent process is killed !!\n");
            exit(0);
        }
        else {
            signal(2,SIG_IGN);
            wait_flag = 1;
            while(wait_flag);
            printf("\n Child process 2 is killed by parent !!\n");
            exit(0);
        }
    }
    else {
        signal(2,SIG_IGN);
        wait_flag = 1;
        while(wait_flag);
        printf("\n Child process 1 is killed by parent !!\n");
        exit(0);
    }
}
void stop(int signum){
    wait_flag = 0;
    printf("\n %d stop \n",signum);
}
```

输出结果如下：

```
● [root@VM-8-12-centos operating_system_2]# ./work_1_6
^C
2 stop

kill!

14 stop

Child process 2 is killed by parent !!

14 stop

Child process 1 is killed by parent !!

Parent process is killed !!
```

```
● [root@VM-8-12-centos operating_system_2]# ./work_1_6

14 stop

kill!

14 stop

Child process 1 is killed by parent !!

14 stop

Child process 2 is killed by parent !!

Parent process is killed !!
```

结果分析：完成实验 3 内容，说明 kill 可以通过发送任意信号给子进程接受，从而实现通信，signal 中 14 原本是需要 alarm 信号才能完成，但是 kill 也能实现，同时父进程可以通过 ctrl+c 接收也可以通过 alarm 接受信号来完成下一步，由于 fork（）只进行 fork（）后面的代码，所以可以确定子进程不是由 alarm 的信号来导致结束。

如果将 kill 删除，并且把 alarm(5)置于最外层或者父进程内，即 test.c, 发现输出结果如下：

```
⊗ [root@VM-8-12-centos operating_system_2]# ./test
^C
  2 stop

kill!

  14 stop
^\\Quit
⊗ [root@VM-8-12-centos operating_system_2]# ./test

  14 stop

kill!
^\\Quit
```

结果分析：

即说明 alarm 信号只由本程序所拥有，与 signal 函数性质不同，并且验证上述内容。

编写 test\_1.c，将 alarm 写在 fork()后，输出结果如下：

```
● [root@VM-8-12-centos operating_system_2]# ./test_1
^C
2 stop

kill!

14 stop

14 stop

Child process 1 is killed by parent !!

14 stop

Child process 2 is killed by parent !!

Parent process is killed !!
```

```
● [root@VM-8-12-centos operating_system_2]# ./test_1

14 stop

kill!

14 stop

Child process 2 is killed by parent !!

14 stop

Child process 1 is killed by parent !!

Parent process is killed !!
```

结果分析：

即成功接收到时钟信号，也同时证明了 alarm 函数只有本程序会拥有。

实际上该课本给出的代码怎么修改，都会导致忙等，这是不好的，所以对程序修改，最终自己修改的 work\_1\_7.c，代码如下：

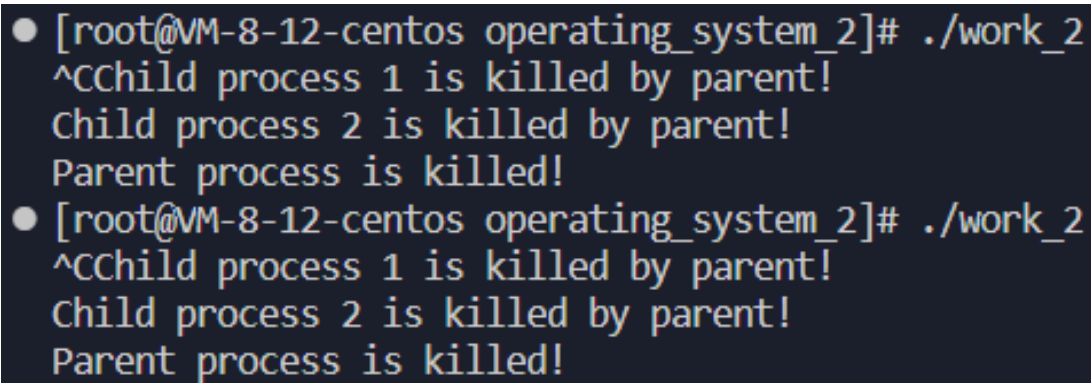
```
#include<sys/types.h>
#include<signal.h>
#include<unistd.h>
#include<stdlib.h>
#include<wait.h>

int pid1,pid2;
void kill_process(){
    kill(pid1, 16);
    kill(pid2, 17);
}
void killed(int signum){
    if(signum == 16){
        printf("Child process 1 is killed by parent!\n");
    }
    else if(signum == 17){
        printf("Child process 2 is killed by parent!\n");
    }
}

int main()
{
    while((pid1 = fork()) == -1);
    if(pid1>0)
    {
        while((pid2 = fork()) == -1);
        if(pid2 > 0)
        {
            signal(2, kill_process);
            signal(14, kill_process);
            alarm(5);
            wait(NULL);
            wait(NULL);
            printf("Parent process is killed!\n");
            exit(0);
        }
        else
        {
            signal(2,SIG_IGN);
            signal(17, killed);
            pause();
            exit(0);
        }
    }
    else
    {
        signal(2,SIG_IGN); //子进程中屏蔽ctrl+c的信号
        signal(16, killed);
        pause();
        exit(0);
    }
    return 0;
}
```



输出结果如下：



```
● [root@VM-8-12-centos operating_system_2]# ./work_2
^Cchild process 1 is killed by parent!
Child process 2 is killed by parent!
Parent process is killed!
● [root@VM-8-12-centos operating_system_2]# ./work_2
^Cchild process 1 is killed by parent!
Child process 2 is killed by parent!
Parent process is killed!
```

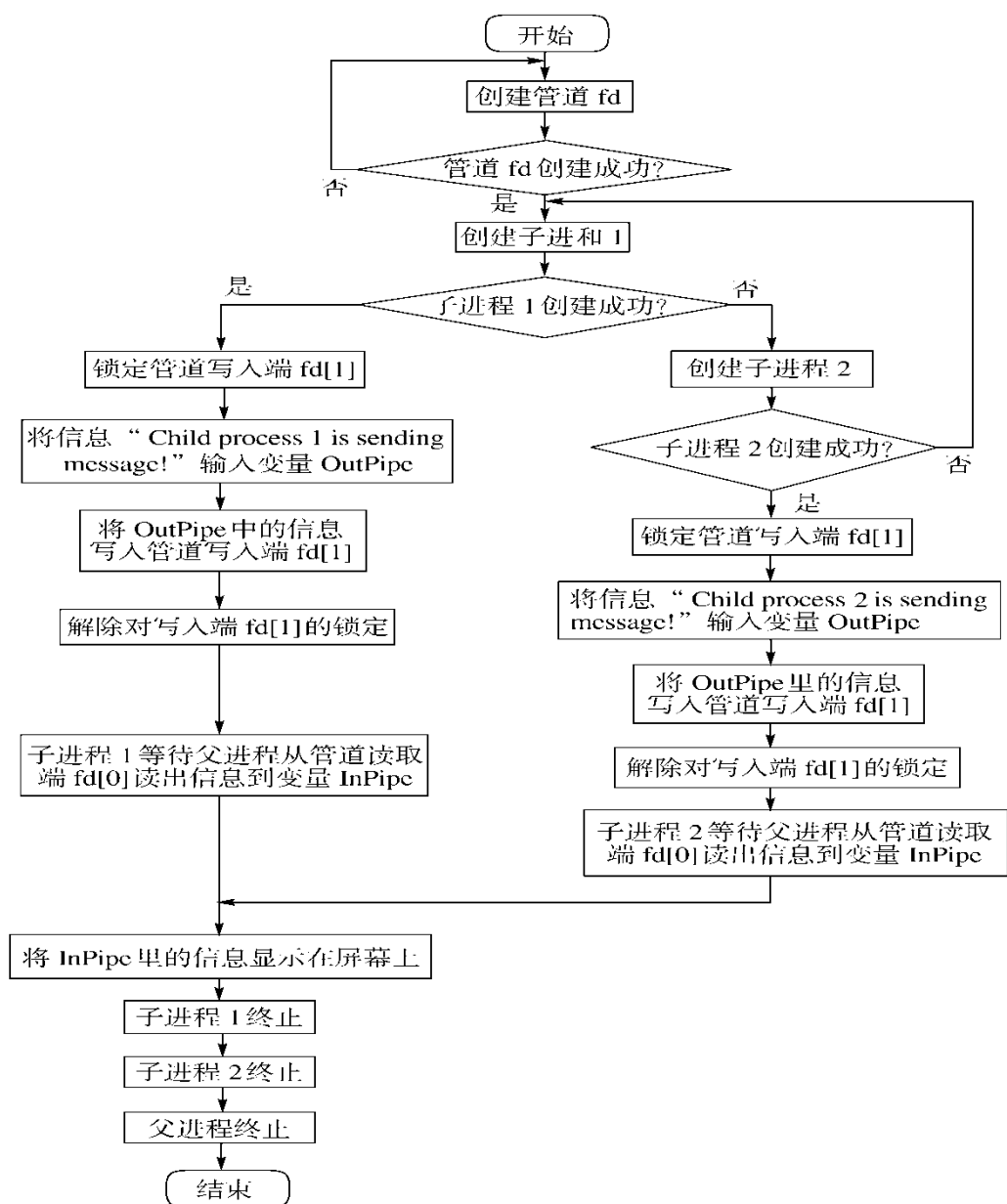
结果分析：

该程序使用了 `pause` 来等待信号，`pause` 函数是让进程被挂起，等待传递的信号之后才会运行，与 `sleep` 类似，这样可以避免忙等，让内存充分利用，是一种更好的实现方式。

## 2.4.2 进程的管道通信

- (1) 先猜想一下这个程序的运行结果。分析管道通信是怎样实现同步与互斥的；
- (2) 然后按照注释里的要求把代码补充完整，运行程序；
- (3) 修改程序并运行，体会互斥锁的作用，比较有锁和无锁程序的运行结果，并解释之。

流程图：



于是将代码补全，写成 work\_2\_1.c, 代码如下：

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
int pid1,pid2;      // 定义两个进程变量
int main(){
    int fd[2];
    int i = 0;
    char InPipe[10000];      // 定义读缓冲区
    char c1='1', c2='2';
    pipe(fd);                // 创建管道
    while((pid1 = fork( )) == -1);    // 如果进程1创建不成功,则空循环
    if(pid1 == 0) {           // 如果子进程1创建成功,pid1为进程号
        lockf(fd[1],1,0);    // 锁定管道
        for(i=0;i<2000;i++){
            write(fd[1],&c1,1);    // 分2000次每次向管道写入字符'1'
        }
        sleep(5);            // 等待读进程读出数据
        lockf(fd[1],0,0);    // 解除管道的锁定
        exit(0);             // 结束进程1
    }
    else{
        while((pid2 = fork()) == -1);    // 若进程2创建不成
        if(pid2 == 0) {
            lockf(fd[1],1,0);
            for(i=0;i<2000;i++){
                write(fd[1],&c2,1);    // 分2000次每次向管道写入字符'2'
            }
            sleep(5);
            lockf(fd[1],0,0);
            exit(0);
        }
        else{
            wait(0);          // 等待子进程1 结束
            wait(0);          // 等待子进程2 结束
            lockf(fd[0],1,0);  // 从管道中读出4000个字
            read(fd[0], InPipe, 4000);
            InPipe[4000] = '\0';
            lockf(fd[0],0,0);
            printf("%s\n",InPipe);    // 显示读出的数据
            exit(0);              // 父进程结束
        }
    }
}
```

输出结果为：

[illegible]

### 结果分析:

可以看出很好的实现了读写同步与写进程互斥的问题，但是子进程 1 和 2 的写入顺序还是不确定，有时候 1 在前，有时候 2 在前。

在 work 2 2.c 中取消锁，输出结果为：

[illegible]

### 结果分析:

可以看出 12 的写入顺序不确定，12 会交替输出等等，竞争使用管道 2:

在 work 2 3.c 中取消 wait ()，输出结果为：

# 操作系统专题实验报告

[illegible]

### 结果分析:

没有做同步之后，读进程在没有写完之后就已经读出，导致 2 输一部分就已经结束，或者只读取 1 的内容

猜想：能不能通过信号量来实现进程 1 和进程 2 的同步？ 通过查询资料有名信号量来实现进程同步通信

<https://dontla.blog.csdn.net/article/details/126441626> (linux 有名信号量)

代码如下：

```

#include<unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int pid1,pid2;      // 定义两个进程变量
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
int main(){
    int fd[2];
    int i = 0;
    union semun sem_union;
    sem_union.val = 0;
    int semid=semget(0x5000,1,0640|IPC_CREAT);
    semctl(semid, 0, SETVAL, sem_union);
    char InPipe[10000];      // 定义读缓冲区
    char c1='1', c2='2';
    pipe(fd);               // 创建管道
    while((pid1 = fork( )) == -1);      // 如果进程1创建不成功,则空循环
    if(pid1 == 0) {          // 如果子进程1创建成功,pid1为进程号
        lockf(fd[1],1,0);    // 锁定管道
        for(i=0;i<2000;i++){
            write(fd[1],&c1,1);    // 分2000次每次向管道写入字符'1'
        }
        sleep(5);           // 等待write读出数据
        lockf(fd[1],0,0);    // 解除管道的锁定
        struct sembuf sem_a;
        sem_a.sem_num = 0;
        sem_a.sem_op = 1;
        sem_a.sem_flg = SEM_UNDO;
        semop(semid, &sem_a, 1);
        exit(0);             // 结束进程1
    }
    else{
        while((pid2 = fork()) == -1);      // 若进程2创建不成
        if(pid2 == 0) {
            struct sembuf sem_b;
            sem_b.sem_num = 0;
            sem_b.sem_op = -1;
            sem_b.sem_flg = SEM_UNDO;
            semop(semid, &sem_b, 1);
            lockf(fd[1],1,0);
            for(i=0;i<2000;i++){
                write(fd[1],&c2,1);    // 分2000次每次向管道写入字符'2'
            }
        }
    }
}

```

```

else{
    while((pid2 = fork()) == -1);           // 若进程2创建不成
    if(pid2 == 0) {
        struct sembuf sem_b;
        sem_b.sem_num = 0;
        sem_b.sem_op = -1;
        sem_b.sem_flg = SEM_UNDO;
        semop(semid, &sem_b, 1);
        lockf(fd[1], 1, 0);
        for(i=0; i<2000; i++){
            write(fd[1], &c2, 1);           // 分2000次每次向管道写入字符'2'
        }
        sleep(5); //等待write
        lockf(fd[1], 0, 0);
        exit(0);
    }
    else{
        wait(0);                           // 等待子进程1 结束
        wait(0);                           // 等待子进程2 结束
        lockf(fd[0], 1, 0);                 // 从管道中读出4000个字
        read(fd[0], InPipe, 4000);
        InPipe[4000] = '\0';
        lockf(fd[0], 0, 0);
        printf("%s\n", InPipe);             // 显示读出的数据
        exit(0);                           // 父进程结束
    }
}
}

```

tence

输出结果如下:

成功实现进程 1 和 2 之前的同步关系!

## FIFO 算法

```
[root@MM-8-12-centos operating_system_2]# ./FIFO
请输入allocation page:13
请输入process page:13
请输入总的序列数:100
随机序列为: 0 9 11 4 1 1 12 3 1 7 2 3 0 4 12 1 9 10 10 11 7 8 3 3 3 5 2 3 10 4 9 12 1 9 5 4 10 5 9 1
1 1 11 3 1 4 5 0 4 2 1 11 0 4 2 3 11 6 6 10 10 2 10 11 1 4 4 11 11 0 12 2 0 2 3 5 9 10 7 1 2 10 12
2 1 3 7 0 9 0 10 7 4 9 7 5 3 12 6 1
命中率: 87%
```

```

● [root@VM-8-12-centos operating_system_2]# ./FIFO
请输入allocation page:3
请输入process page:5
请输入总的序列数:6
随机序列为: 3 1 2 0 3 0
命中率: 16.6667%

```

40



```

● [root@VM-8-12-centos operating_system_2]# ./test_FIFO
请输入allocation page:3
请输入process page: 5
请输入总的序列数: 12
随机序列为:1 2 3 4 1 2 5 1 2 3 4 5
命中率: 25%
● [root@VM-8-12-centos operating_system_2]# ./test_FIFO
请输入allocation page:4
请输入process page: 5
请输入总的序列数: 12
随机序列为:1 2 3 4 1 2 5 1 2 3 4 5
命中率: 16.6667%

```

可以看出分配的页面数增多，反而命中率下降。

## LRU 算法

最初为了先实现功能，编写代码 LRU.cpp,实现了交互的功能，输出结果如下：

```

● [root@VM-8-12-centos operating_system_2]# ./LRU
请输入allocation page:2
请输入process page:3
请输入总的序列数:4
随机序列为: 1 1 0 1
命中率: 50%

```

```

● [root@VM-8-12-centos operating_system_2]# ./LRU
请输入allocation page:3
请输入process page:5
请输入总的序列数:6
随机序列为: 3 1 2 0 3 0
命中率: 16.6667%

```

```

● [root@VM-8-12-centos operating_system_2]# ./FIFO
请输入allocation page:3
请输入process page:6
请输入总的序列数:8
随机序列为: 1 4 3 1 5 1 4 0
命中率: 12.5%
● [root@VM-8-12-centos operating_system_2]# ./LRU
请输入allocation page:3
请输入process page:6
请输入总的序列数:8
随机序列为: 1 4 3 1 5 1 4 0
命中率: 25%

```

可以简单验证，基本符合功能要求

## 代码重构

新建 page\_change 文件夹，对代码进行重构：

FIFO.cpp

LRU.cpp

struct.cpp

pagechange.h

(代码见附件)

编写 test.cpp 进行调用，代码如下：

```
#include "pagechange.h"
int main(){
    int ap,pp,total;
    cout<<"请输入 allocation page:";
    cin>>ap;
    cout<<"请输入 process page:";
    cin>>pp;
    cout<<"请输入总的序列数:";
    cin>>total;
    cout<<"随机序列为： ";
    queue<int> main;
    for(int i=0;i<total;i++){
        main.push(rand()%pp);
        cout<<main.back()<<" ";
    }
    cout<<endl;
    int diseffect1 = FIFO(main,ap,pp);
    int diseffect2 = LRU(main,ap,pp);
```

```

float minzhong1,minzhong2;
minzhong1 = (1.00-(1.00*diseffect1)/total)*100;           //计算命中率
minzhong2 = (1.00-(1.00*diseffect2)/total)*100;
cout<<"FIFO 命中率:"<<minzhong1<<"%"<<endl;
cout<<"LRU 命中率:"<<minzhong2<<"%"<<endl;
return 0;
}

```

编写 test2.cpp 实现可输入序列，代码如下：

```

#include "pagechange.h"
int main(){
    int ap,pp,total;
    cout<<"请输入 allocation page:";
    cin>>ap;
    cout<<"请输入 process page:";
    cin>>pp;
    cout<<"请输入总的序列数:";
    cin>>total;
    cout<<"请输入序列:";
    int main[total];
    for(int i=0;i<total;i++){
        cin>>main[i];
    }
    int diseffect1 = FIFO(main,ap,pp,total);
    int diseffect2 = LRU(main,ap,pp,total);
    float minzhong1,minzhong2;
    minzhong1 = (1.00-(1.00*diseffect1)/total)*100;           //计算命中率
    minzhong2 = (1.00-(1.00*diseffect2)/total)*100;
    cout<<"FIFO 命中率:"<<minzhong1<<"%"<<endl;
    cout<<"LRU 命中率:"<<minzhong2<<"%"<<endl;
    return 0;
}

```

输出结果为：

```
[root@MM-8-12-centos page_change]# ./main
请输入allocation page:3
请输入process page:6
请输入总的序列数:8
随机序列为: 1 4 3 1 5 1 4 0
FIFO命中率:12.5%
LRU命中率:25%
```

结果较好，对两个函数封装较完整。

## 2.5 测试数据设计

观察 Bledy 现象时，从 csdn 中查找到特定能观察到 bledy 现象的序列，因为光从随机函数生成序列观察 bledy 现象十分困难

## 2.6 程序运行初值及运行结果分析

运行与结果分析为了便于理解，均附于 2.4 中

## 2.7 页面置换算法复杂度分析

通过对代码分析可知

FIFO 中时间主要用于遍历整个随机序列 total，而且对于每个序列需要遍历已分配内存页，最坏情况为 ap，故最坏时间复杂度为  $O(ap * total)$ ，空间复杂度则是两个结构体数组和内存分配页数组空间相加，为  $O(total + ap + pp)$

LRU 中时间主要用于遍历整个随机序列 total，而且对于每个序列需要遍历现在分配的内存页面，最坏情况为 ap，故最坏时间复杂度为  $O(ap * total)$ ，空间复杂度则与 FIFO 相同，为  $O(total + ap + pp)$

## 2.8 回答问题

### 2.8.1 软中断通信

- 你最初认为运行结果会怎么样？写出你猜测的结果；

我认为应该是等待 5s 后，父进程发出信号使子进程结束，子进程结束后立即输出，父进程然后输出。

- 实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将 5 秒内中断和 5 秒后中断的运行结果截图，试对产生该现象的原因进行分析。

实际结果是子进程先输出，等待 5s 后父进程才输出，产生该输出结果的原因是，该程序连等待信号都没有，而父进程在等待 5s 后，发现子进程已经结束，于是输出并结束。

- 针对实验过程 2，怎样修改的程序？修改前后程序的运行结果是什么？请截图说明。

如上述可知，修改程序将子进程做信号屏蔽，让子进程只有收到 kill 发出的信号才有用，修改后，成功实现目标。

- 针对实验过程 3，程序运行的结果是什么样子？时钟中断有什么不同？

如上图所示，时钟中断既可以通过定时触发，也可以 kill 触发。

- kill 命令在程序中使用了幾次？每次的作用是什么？执行后的现象是什么？

2 次，每次的作用是发送信号给子进程，执行后子进程接收到父进程信号结束

- 使用 kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

进程调用 return 函数和 exit 函数都可以主动退出，而 kill 是强制退出。主动退出比较好，如果在某个子进程退出前父进程被 kill 强制退出，则子进程会被 init 进程接管；如果用 kill 命令杀死某个子进程而其父进程没有调用 wait 函数等待，则该子进程为处于僵死状态占用资源。

## 2.8.2 管道通信

- 你最初认为运行结果会怎么样？

屏幕会输出 2000 个 1 和 2000 个 2。

- 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

多次实验发现有时先输出 2000 个 1，有时输出 2000 个 2，输出顺序不固定，这是因为子进程 1 和子进程 2 执行的先后顺序是不一定的，如果进程 1 先执行则先输入 1；如果进程 2 先执行则先输入 2。但是并没有交错输出 1 或 2，这是因为子进程在写入前线上锁，保证只有一个进程在向管道输入数据。

- 实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果。

通过父进程 wait 子进程来实现读写同步，通过互斥锁来控制互斥，不控制的话，后果如 2.4.1 中输出结果所示。

## 2.9 实验总结

### 2.9.1 实验中的问题及解决过程

实验中遇到的问题一开始主要是进程软中断中通信实验要求有点看不懂，后面发现指导书上的代码，发现原来是指导书上代码错误，所以实验要求才那样写

然后为了实现管道写进程之间的同步关系，查阅 linux 有名信号量，最终实现

页面置换问题在于一开始写的代码不够规范，于是查阅多文件联合编译，将函数封装，提高代码质量

### 2.9.2 实验收获

学会了多文件编译与多种系统调用，linux 中进程之间通信的方法，并且在实验过程中发现 signal 16 的系统默认处理实际上时杀死进程，而不是课本上写的无操作。书上的原理是一套，但是实际上操作系统怎么实现而各有区别。原理和实践结合，让我对操作系统有了更深的理解

## 2.9.3 意见与建议

感觉页面置换那部分实验要求并不是很明确，而且进程软中断通信的实验要求 1、2 很怪，并不是正确程序的要求，貌似是从指导书代码而来

## 2.10 附件

### 2.10.1 附件 1 程序

### 2.10.2 附件 2 Readme

## 实验三 类 EXT2 文件系统设计

### 3.1 实验目的

通过一个简单文件系统的设计，加深理解文件系统的内部实现原理

### 3.2 实验内容

模拟 EXT2 文件系统原理设计实现一个类 EXT2 文件系统

### 3.3 实验思想

为了进行简单的模拟，基于 Ext2 的思想和算法，设计一个类 Ext2 的文件系统，实现 Ext2 文件系统的功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。

## 设计文件系统应该考虑的几个层次

- 介质的物理结构
- 物理操作——设备驱动程序完成
- 文件系统的组织结构（逻辑组织结构）
- 对组织结构其上的操作

- 为用户使用文件系统提供的接口

## 文件系统的基本实现

- 数据结构及其磁盘布局

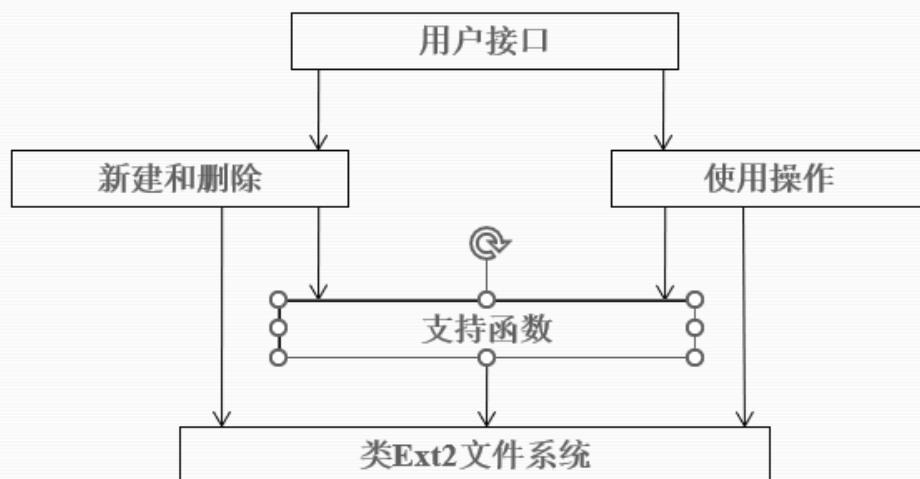
文件的存储：如何帮助用户找到磁盘中存储的文件？如何防止一个用户非法访问另一个用户的文件？如何知道磁盘被使用的情况？

整体布局、超级块、文件的索引、目录的存储、空闲块管理、块组描述符

- 文件的基本操作

接口和用户界面设计、节点创建删除设计、节点操作设计

### 接口和用户界面设计、节点创建删除设计、节点操作设计



## 类 ext2 文件系统的数据结构

- 块的定义



为简单起见，逻辑块大小与物理块大小均定义为 512 字节。由于位图只占用一个块，因此，每个组的数据块个数以及索引结点的个数均确定为  $512 \times 8 = 4096$ 。进一步，每组的数据容量确定为  $4096 \times 512\text{B} = 2\text{MB}$ 。另外，模拟系统中，假设只有一个用户，故可以省略去文件的所有者 ID 的域。

### ● 组描述符

为简单起见，只定义一个组。因此，组描述符只占用一个块。同时，superblock 块省略，其功能由组描述符块代替，即组描述符块中需要增加文件系统大小，索引结点的大小，卷名等原属于 superblock 的域。由此可得组描述符的数据结构如下：

```
struct ext2_group_desc
```

```
{
```

类型	bytes	域	释意
char[ ]	16	bg_volume_name[16];	卷名
__u16	2	bg_block_bitmap;	保存块位图的块号
__u16	2	bg_inode_bitmap;	保存索引结点位图的块号
__u16	2	bg_inode_table;	索引结点表的起始块号
__u16	2	bg_free_blocks_count;	本组空闲块的个数
__u16	2	bg_free_inodes_count;	本组空闲索引结点的个数
__u16	2	bg_used_dirs_count;	本组目录的个数
char[ ]	4	bg_pad[4];	填充(0xff)

```
};
```

合计 32 个字节，由于只有一个组，且占用一个块，故需要填充剩下的  $512 - 32 = 480$  字节。

## 索引结点数据结构

- 由于容量已经确定，文件最大即为 2MB。需要 4096 个数据块。索引结点的数据结

构中，仍然采用多级索引机制。由于文件系统总块数必然小于  $4096 \times 2$ ，所以只需要 13 个二进制位即可对块进行全局计数，实际实现用 unsigned int 16 位变量，即 2 字节表示 1 个块号。

- 在一级子索引中，如果一个数据块都用来存放块号，则可以存放  $512/2 = 256$  个。因此，只使用一级子索引可以容纳最大的文件为  $256 \times 512 = 128\text{KB}$ 。需要使用二级子索引。只使用二级子索引时，索引结点中的一个指针可以指向  $256 \times 256$  个块，即  $256 \times 256 \times 512 = 8\text{MB}$ ，已经可以满足要求了。为了尽量“像”ext2，也为了简单起见，索引结点的直接索引定义 6 个，一级子索引定义 1 个，二级子索引定义 1 个。总计 8 个指针。

### ● 索引结点的数据结构定义

```
struct ext2_inode {
```

类型	字节长度	域	释意
__u16	2	i_mode;	文件类型及访问权限
__u16 2		i_blocks;	文件的数据块个数
__u32 4		i_size;	大小(字节)
__u64	4	i_atime;	访问时间
__u64	4	i_ctime;	创建时间
__u64	4	i_mtime;	修改时间
__u64	4	i_dtime;	删除时间
__u16[8]	$2 \times 8 = 16$	i_block [8]	指向数据块的指针
char[8]	8	i_pad	填充 1(0xff)
}			

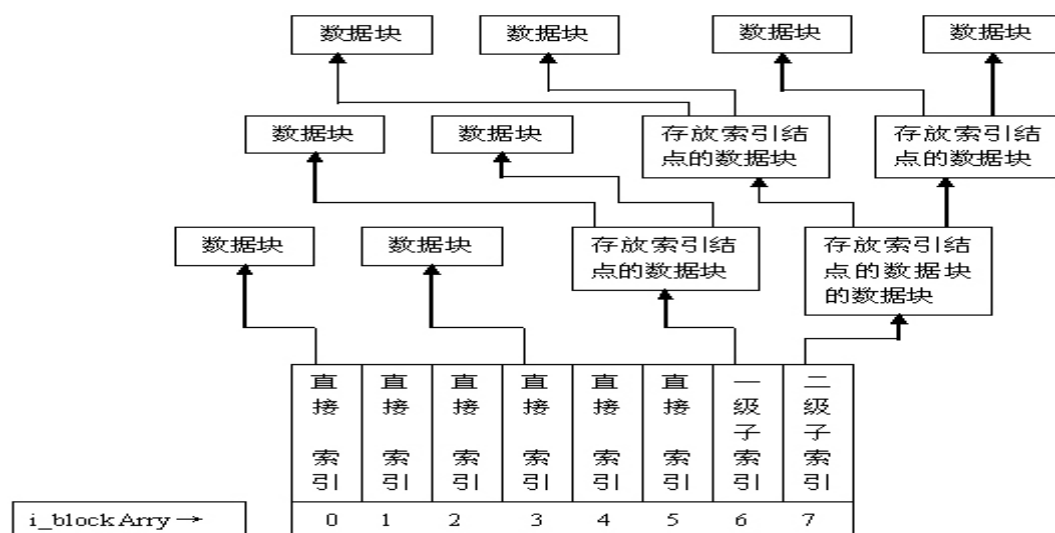
即，每个索引结点的长度为 64 字节。

### ● 索引节点各字段说明

- `i_mode` 域构成一个文件访问类型及访问权限描述。即 linux 中的 `drwxrwxrwx` 描述。d 为目录，r 为读控制，w 为写控制，x 为可执行标志。并且，3 个 `rwX` 分别是所有者(owner)，组(group)，全局(universe)这三个对象的权限。为了简单的起见，模拟系统中，`i_mode` 的 16 位如下分配。高 8 位(`high_i_mode`)，是目录项中文件类型码的一个拷贝。低 8 位(`low_i_mode`)中的最低 3 位分别用来标识 `rwX` 3 个属性。高 5 位不用，用 0 填充。在显示文件访问权限时，3 个对象均使用低 3 位的标识。这样，由这 16 位，即可生成一个文件的完整的 `drwxrwxrwx` 描述。

- 特别的，在 Unix 中，不带扩展名的文件定性为可执行文件。在模拟系统中，凡是扩展名为 `.exe`, `.bin`, `.com` 及不带扩展名的，都被加上 x 标识。

### ● `i_block` 域与文件大小以及数据块的关系



- (1) 当文件长度小于等于  $512 \times 6 = 3072$  字节 (3KB) 时，只用到直接索引
- (2) 当文件长度大于 3072 字节，并且小于等于  $3KB + 128KB$  即 131KB 时，除使用直接索引处，还将使用一级子索引
- (3) 当文件长度大于 131KB，并且小于 2MB (文件系统最大值) 时，将开始使用二级

## 文件系统的数据容量

- 假设仅保存一个大文件，现在来计算这个文件最多能有多大。由于索引结点表本身要占用数据块，文件最大长度并没有 2MB。文件系统初始化时，根目录用去一个数据块。当文件大于 131KB 时，一级子索引本身还要占用一个数据块 512 字节。即，已经用去  $131\text{KB} + 0.5\text{KB}(\text{根目录}) + 0.5\text{KB}(\text{索引数据块}) = 132\text{KB}$  的空间。
- 此外，由于文件系统的限制，二级子索引并不会全部使用。这里定义二级子索引的使用顺序是深度优先。即，必须先使用完第 1 个二级子索引  $512 * 256 = 128\text{KB}$  后，再使用第 2 个二级子索引。这样，每使用一个 2 级索引，多占用 1 个数据块，512 字节。假设使用了  $n (< 256)$  个二级子索引，占用的空间大小为  $(1+n) * 512 = (1+n)/2 \text{ KB}$ 。加上已经使用的 132KB 的空间，总共使用的容量为  $131.5 + 128 * n + (1+n)/2 \text{ KB}$ 。令其等于文件系统的容量，2MB，解方程:  $132 + 128 * n + (1+n)/2 = 2048$  得  $n = 14.91$ 。即最多只能使用满 14 个二级子索引，最多可以使用到 15 个二级子索引。
- 由上可得，对一个文件，实际中子索引系统所占用的数据块最多为  $1 + 1 + 15 = 17$  个。加上根目录，总共用了 18 个数据块来存放文件系统的信息。当 14 个二级子索引全部使用满时，用来存储文件数据的数据块最为  $6 + 256 + 14 * 256 = 3846$  个。当使用第 15 个二级子索引时，又用 1 个数据块来存放索引结点。文件系统总计有 4096 个数据块。那么留下给第 15 个二级子索引使用的数据块为  $4096 - 18 - 3846 = 232$ 。这是第 15 个二级子索引的索引结点数的上限。文件系统的实际容量为  $(4096 - 18) * 512 = 2087936 \text{ 字节} = 2039 \text{ KB} = 1.99\text{MB}$

以上容量是在仅有一个文件的情况下计算出来的。实际系统中，文件的数目很多，大小也不尽相同。现在估算文件系统容量如下：

文件大小	占用的数据块	使用的索引结点块
$\leq 3\text{KB}$	$\leq 6$	0
$\leq 131\text{KB}$	$\leq 6 + 256 = 262$	1
$\leq 2039\text{KB}$	$\leq 6 + 256 + 14 * 256 + 232 = 4078$	17

假设每个文件大小平均分布，并把目录当成普通文件则平均占用数据块

$4078/2=2039$  个,平均使用索引结点块为:

$$\frac{3 * 0 + (131 - 3) * 1 + (2039 - 131) * 17}{2039} = 16$$

则硬盘空间利用效率为:

$$\frac{2039}{2039 + 16} = 99.22\%$$

则文件系统容量大约为 $(4096 - 1) * 512 * 99.22\% = 2080214$  字节 =

2031.4KB=1.98MB。

## 索引结点表

- 由于每个索引结点大小为 64 个字节，最多有  $512 * 8 = 4096$  个索引结点。故，索引结点表的大小为  $64 * 4096 = 256\text{KB}$ ，512 个块。为了和 ext2 保持一致，索引结点从 1 开始计数，0 表示 NULL。数据块则从 0 开始计数。

## 模拟文件系统的“硬盘”数据结构

- 基于以上若干定义，得到模拟文件系统的“硬盘”数据结构：

组描述符	数据块位图	索引结点位图	索引结点表	数据块
1 block	1 block	1 block	512 blocks	4096 blocks
512 Bytes	512 Bytes	512 Bytes	256 KB	2 MB

- 整个模拟文件系统所需要的“硬盘”空间为  $1+1+1+512+4096=4611$  个块。共计  $4611 * 512\text{bytes} = 2,360,832$  字节 = 2305.5KB = 2.25MB。其中，数据容量为

1.99MB。最多可容纳的文件数目为  $4096-17=4079$  个。每个文件占用的数据空间最小为 512 字节，即一个块大小

## 目录与文件

- 与 ext2 相同，目录作为特殊的文件来处理。将第 1 个索引结点指向根目录。根目录的索引结点中直接索引域指向数据块 0。
- 目录体的数据结构与 ext2 基本相同，唯一的区别在于索引节点号用 16 位来表示：

```
struct ext2_dir_entry {
```

Type	Bytes	Field	释意
------	-------	-------	----

__u16	2	inode;	索引节点号
-------	---	--------	-------

__u16	2	rec_len;	目录项长度
-------	---	----------	-------

__u8	1	name_len;	文件名长度
------	---	-----------	-------

__u8	1	file_type;	文件类型(1:普通文件, 2:目录...)
------	---	------------	-----------------------

char[ ]	8*_LEN	name[EXT2_NAME_LEN];	文件名
---------	--------	----------------------	-----

```
};
```

其中，文件名最大长度为 255 字符（节）。因此，目录项的长度范围是 7 至 261 字节。

- 当文件系统在初始化时，根目录的数据块（即数据块 1）将被初始化。其所包含的所有索引节点号以及目录项长度域将被置 0。当文件被删除时，其所在目录项长度不变，索引节点号将被置 0。
- 当新建一个文件时，程序将从目录的数据块查找索引节点号为 0 的目录项，并检查其长度是否够用。是，则分配给该文件，否则继续查找，直到找到长度够用，或者是长度为 0（即未被使用过）的地址，为文件建立目录项。
- 当建立的是一个目录时，将其所分配到的索引结点所指向的数据块清空。并且自动写入两个特殊的目录项。一个是当前目录“.”,其索引结点即指向本身的数据块。另一个是上一级目录“..”，其索引结点指向上一级目录的数据块。例如，/root 目录。其

索引结点号为 1。并且，第 1 个数据块存放着该目录的目录项。`/root` 目录在文件系统初始化时自动生成。同时，在目录项中自动生成以下两项：

inode	rec_len	name_len	file_type	name		
1	8	1	2	.	\0	
1	9	2	2	.	.	\0

## 文件类型

- 文件类型项与 ext2 完全一样：

文件类型号	描述
0	未知
1	普通文件
2	目录
3	字符设备
4	块设备
5	管道(Pipe)
6	套接字
7	符号指针

## 模拟文件系统的操作

		操 作 对 象	
操作(命令)	功能	索引结点	数据块
<u>dir</u>	列当前目录	✓	
<u>mkdir</u>	建立目录	✓	✓
<u>rmdir</u>	删除目录	✓	✓
<u>create</u>	建立文件	✓	
<u>delete</u>	删除文件	✓	✓
<u>cd</u>	进入目录	✓	
<u>attrib</u>	更改文件保护码	✓	
<u>open</u>	打开文件	✓	
<u>close</u>	关闭文件	✓	
<u>read</u>	读文件	✓	✓
<u>write</u>	写文件	✓	✓

- 为了实现这些操作，内存中也必须有相应的数据结构。首先，内存中应当定义一个“当前目录”的数据结构，用来存放“当前目录”的索引结点号。此外，内存中还应当有一个“文件打开表”的数据结构。包括，打开文件 ID，索引结点号。两个域。文件打开表应当是一个数组，数组的元组即允许同时打开的文件个数

## “存储空间”的管理

- 这里涉及到模拟文件系统的 5 个底层操作：索引结点的分配与释放、数据块的分配与释放以及数据块的寻址。
- 这些操作将采用 ext2 基本相同的方法实现。区别在于：Ext2 中对 superblock 的操作将变成对组描述符的操作。此外，数据块在分配时不采取预先分配策略。查找空闲块的方法可采用从某个起始点开始线性查找。

## 程序结构

- 初始化模拟文件系统
  - 在已有的文件系统的基础上建立一个大小为 FS\_SPACE=2,360,832 字节(即 2.25MB)的文件 FS.txt，这个文件即用来模拟硬盘。以后，文件系统的所有操作，均通过读写这个文件实现。并且，完全模拟硬盘读写方式，一次读取 1 个



块，即 512 字节。即使只有 1 个字节的修改，也通过读写一个数据块来实现。

- 常驻内存的数据结构也被初始化。

- 文件系统级（底层）函数及其子函数

- 这些函数完成了所有文件系统底层的操作封装。并为上层即命令层提供服务。该层实现了所有对文件系统“硬盘”的块操作功能。例如：分配和回收索引结点与数据块，索引结点的读取与写入，数据块的读取与写入，索引结点及数据块位图的设置，组描述符的修改，多级索引的实现等。

- 命令层函数

- 文件系统所支持的命令及其功能在这一层实现。一共实现 11 个命令：  
dir,mkdir,rmdir,create,delete,cd,attrib, open,close,read,write。为了实现这些命令，本层使用底层所提供的服务。

- 用户接口层

- 主要功能是接收及识别用户命令，词法分析，提取命令及参数。组织调用命令层对应的命令实现相应功能。本层实际上是一个基于命令层基础上的 shell。
- 为了完善接口的功能，shell 程序中增加了一些附加命令。这些命令无需调用文件系统级函数。这些附加命令有：quite 命令退出程序，format 命令重新建立文件系统。

## 各层函数列表

- 初始化文件系统

- initialize\_disk() /\*建立文件系统\*/
- initialize\_memory() /\*初始化文件系统的内存数据\*/

- 底层

- update\_group\_desc() /\*将内存中的组描述符更新到"硬盘"\*/
- reload\_group\_desc() /\*载入可能已更新的组描述符\*/

- `load_inode_entry()` /\*载入特定的索引结点\*/
- `update_inode_entry()` /\*更新特定的索引结点\*/
- `load_block_entry()` /\*载入特定的数据块\*/
- `update_block_entry()` /\*更新特定的数据块\*/
- `update_inode_i_block()` /\*根据多级索引机制更新索引结点的数据块信息域\*/
- `ext2_new_inode()` /\*分配一个新的索引结点\*/
- `ext2_alloc_block()` /\*分配一个新的数据块\*/
- `ext2_free_inode()` /\*释放特定的索引结点\*/
- `ext2_free_block_bitmap()` /\*释放特定块号的数据块位图\*/
- `ext2_free_blocks()` /\*释放特定文件的所有数据块\*/
- `search_filename()` /\*在当前目录中查找文件\*/
- `test_fd()` /\*检测文件打开 ID(fd)是否有效\*/

## ● 命令层

- `dir()` /\*无参数\*/
- `mkdir()` /\*filename\*/
- `rmdir()` /\*filename\*/
- `create()` /\*filename\*/
- `delete_()` /\*filename\*/
- `cd()` /\*filename\*/
- `attrib()` /\*filename, rw\*/
- `open()` /\*filename\*/
- `close()` /\*fd\*/

- 
- read()                                /\*fd\*/
  - write()                                /\*fd, source\*/
  - 用户接口层及附加命令
    - shell()                                /\*启动用户接口\*/
    - format()                               /\*重新建立文件系统,无参数\*/
    - quit()                                 /\*退出 shell(),无参数\*/
  - 常驻内存的数据结构释意
    - unsigned short fopen\_table[16] ; /\*文件打开表, 最多可以同时打开 16 个文件\*/
    - unsigned short last\_alloc\_inode; /\*上次分配的索引结点号\*/
    - unsigned short last\_alloc\_block; /\*上次分配的数据块号\*/
    - unsigned short current\_dir ; /\*当前目录(索引结点) \*/
    - char current\_path[256];                /\*当前路径(字符串) \*/
    - struct ext2\_group\_desc;                /\*组描述符\*/

## 3.4 实验步骤

以下是自己制作文件系统的具体实现：

### 3.4.1 EXT2 文件系统结构

组描述符：

```
struct ext2_group_desc //组描述符 80 字节
{
    char bg_volume_name[16]; //卷名
    int bg_block_bitmap;     //保存块位图的块号
    int bg_inode_bitmap;     //保存索引结点位图的块号
```

```

int bg_inode_table;    //索引结点表的起始块号
int bg_free_blocks_count; //本组空闲块的个数
int bg_free_inodes_count; //本组空闲索引结点的个数
int bg_used_dirs_count; //本组目录的个数
char psw[16];
char bg_pad[24]; //填充(0xff)
};

```

索引节点:

```

struct ext2_inode //索引节点 64 字节
{
    int i_mode;    //文件类型及访问权限          4
    int i_blocks; //文件的数据块个数
    int i_size;    //大小(字节)
    int fd;
    time_t i_atime; //访问时间          8
    time_t i_ctime; //创建时间
    time_t i_mtime; //修改时间
    time_t i_dtime; //删除时间
    short i_block[8]; //指向数据块的指针
};

```

目录体:

```

struct ext2_dir_entry //目录体 32 字节
{
    int inode;    //索引节点号
    int rec_len;    //目录项长度
    int name_len;    //文件名长度
    int file_type;    //文件类型(1:普通文件, 2:目录...)
    char name[EXT2_NAME_LEN]; //文件名
    char authority;    //权限 0 代表读写 1 代表只读 2 代表只写
};

```

文件系统维护的全局变量:

```
extern ext2_group_desc group_desc; //组描述符
extern ext2_inode inode;
extern ext2_dir_entry dir;          //目录体
extern FILE *f;                     /*文件指针*/
extern unsigned int last_allco_inode; //上次分配的索引节点号
extern unsigned int last_allco_block; //上次分配的数据块号
extern unsigned int fopen_table[16]; //文件打开表
```

文件系统所依赖的所有函数:

```
int getch();
int format(ext2_inode *cu);
int init_open_table();
int dir_entry_position(int dir_entry_begin, short i_block[8]);
int Open(ext2_inode *current, char *name);
int Close(ext2_inode *current);
int Read(ext2_inode *current, char *name);
int FindInode();
int FindBlock();
void Dellnode(int len);
void DelBlock(int len);
int attrib(ext2_inode *current, char *name, char authority);
void add_block(ext2_inode *current, int i, int j);
int FindEntry(ext2_inode *current);
int Create(int type, ext2_inode *current, char *name);
int Write(ext2_inode *current, char *name);
void lsdir(ext2_inode *current, int mode);
int initialize(ext2_inode *cu);
int openfile(ext2_inode *current, char *name);
int closefile(ext2_inode *current, char *name);
int Password();
int login();
```

```

void exitdisplay();
int initfs(ext2_inode *cu);
void getstring(char *cs, ext2_inode node);
int Delet(int type, ext2_inode *current, char *name);
void shellloop(ext2_inode currentdir);

```

文件系统设置的大小、路径等等：

```

#define blocks 4611 // 1+1+1+512+4096,总块数
#define blocksiz 512 //每块字节数
#define data_begin_block 515 //数据开始块
#define dirsiz 32 //目录体长度
#define EXT2_NAME_LEN 15 //文件名长度
#define PATH "vdisk" //文件系统

```

### 3.4.2 文件系统设计介绍

#### 文件类型

设计的文件类型有两种，一种是目录，一种是文件，分别编号为 1 和 2

Directory	..	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022
File	p	Sat Dec 3 12:05:35 2022	Sat Dec 3 12:08:02 2022	Sat Dec 3 12:05:35 2022

#### 文件系统功能

(1) dir

```

[root@ext2]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sat Dec 3 12:05:27 2022      Sat Dec 3 12:05:27 2022      Sat Dec 3 12:05:27 2022
Directory ..           Sat Dec 3 12:05:27 2022      Sat Dec 3 12:05:27 2022      Sat Dec 3 12:05:27 2022
File      p              Sat Dec 3 12:05:35 2022      Sat Dec 3 12:08:02 2022      Sat Dec 3 12:05:35 2022
File      p1             Sat Dec 3 12:05:45 2022      Sat Dec 3 12:07:15 2022      Sat Dec 3 12:05:45 2022
File      p2             Sat Dec 3 12:05:49 2022      Sat Dec 3 12:07:20 2022      Sat Dec 3 12:05:49 2022
File      p3             Sat Dec 3 12:05:54 2022      Sat Dec 3 12:07:22 2022      Sat Dec 3 12:05:54 2022
File      p4             Sat Dec 3 12:05:56 2022      Sat Dec 3 12:07:23 2022      Sat Dec 3 12:05:56 2022
File      p5             Sat Dec 3 12:05:57 2022      Sat Dec 3 12:07:25 2022      Sat Dec 3 12:05:57 2022
File      p6             Sat Dec 3 12:05:59 2022      Sat Dec 3 12:07:26 2022      Sat Dec 3 12:05:59 2022
File      p7             Sat Dec 3 12:06:01 2022      Sat Dec 3 12:07:28 2022      Sat Dec 3 12:06:01 2022
File      p8             Sat Dec 3 12:06:03 2022      Sat Dec 3 12:07:29 2022      Sat Dec 3 12:06:03 2022
File      p9             Sat Dec 3 12:06:05 2022      Sat Dec 3 12:07:31 2022      Sat Dec 3 12:06:05 2022
File      p10            Sat Dec 3 12:06:07 2022      Sat Dec 3 12:07:33 2022      Sat Dec 3 12:06:07 2022
File      p11            Sat Dec 3 12:06:10 2022      Sat Dec 3 12:07:34 2022      Sat Dec 3 12:06:10 2022
File      p12            Sat Dec 3 12:06:12 2022      Sat Dec 3 12:07:37 2022      Sat Dec 3 12:06:12 2022
File      p13            Sat Dec 3 12:06:15 2022      Sat Dec 3 12:07:39 2022      Sat Dec 3 12:06:15 2022
File      p14            Sat Dec 3 12:06:17 2022      Sat Dec 3 12:07:40 2022      Sat Dec 3 12:06:17 2022
File      p15            Sat Dec 3 12:06:19 2022      Sat Dec 3 12:07:41 2022      Sat Dec 3 12:06:19 2022
File      p16            Sat Dec 3 12:06:21 2022      Sat Dec 3 12:07:43 2022      Sat Dec 3 12:06:21 2022

```

## 操作系统专题实验报告

```
[root@ext2]# dir -m
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime	Authority
Directory	.	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	NULL
Directory	..	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	NULL
File	p	Sat Dec 3 12:05:35 2022	Sat Dec 3 12:08:02 2022	Sat Dec 3 12:05:35 2022	rw
File	p1	Sat Dec 3 12:05:45 2022	Sat Dec 3 12:07:15 2022	Sat Dec 3 12:05:45 2022	rw
File	p2	Sat Dec 3 12:05:49 2022	Sat Dec 3 12:07:20 2022	Sat Dec 3 12:05:49 2022	rw
File	p3	Sat Dec 3 12:05:54 2022	Sat Dec 3 12:07:22 2022	Sat Dec 3 12:05:54 2022	rw
File	p4	Sat Dec 3 12:05:56 2022	Sat Dec 3 12:07:23 2022	Sat Dec 3 12:05:56 2022	rw
File	p5	Sat Dec 3 12:05:57 2022	Sat Dec 3 12:07:25 2022	Sat Dec 3 12:05:57 2022	rw
File	p6	Sat Dec 3 12:05:59 2022	Sat Dec 3 12:07:26 2022	Sat Dec 3 12:05:59 2022	rw
File	p7	Sat Dec 3 12:06:01 2022	Sat Dec 3 12:07:28 2022	Sat Dec 3 12:06:01 2022	rw
File	p8	Sat Dec 3 12:06:03 2022	Sat Dec 3 12:07:29 2022	Sat Dec 3 12:06:03 2022	rw
File	p9	Sat Dec 3 12:06:05 2022	Sat Dec 3 12:07:31 2022	Sat Dec 3 12:06:05 2022	rw
File	p10	Sat Dec 3 12:06:07 2022	Sat Dec 3 12:07:33 2022	Sat Dec 3 12:06:07 2022	rw
File	p11	Sat Dec 3 12:06:10 2022	Sat Dec 3 12:07:34 2022	Sat Dec 3 12:06:10 2022	rw
File	p12	Sat Dec 3 12:06:12 2022	Sat Dec 3 12:07:37 2022	Sat Dec 3 12:06:12 2022	rw
File	p13	Sat Dec 3 12:06:15 2022	Sat Dec 3 12:07:39 2022	Sat Dec 3 12:06:15 2022	rw
File	p14	Sat Dec 3 12:06:17 2022	Sat Dec 3 12:07:40 2022	Sat Dec 3 12:06:17 2022	rw
File	p15	Sat Dec 3 12:06:19 2022	Sat Dec 3 12:07:41 2022	Sat Dec 3 12:06:19 2022	rw
File	p16	Sat Dec 3 12:06:21 2022	Sat Dec 3 12:07:43 2022	Sat Dec 3 12:06:21 2022	rw
Directory	show	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:18:47 2022	NULL

### (2) mkdir

```
[root@ext2 show]# mkdir dir
Congratulations! dir is created
```

```
[root@ext2 show]# dir
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
Directory	.	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:14:13 2022
Directory	..	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022
Directory	dir	Sun Dec 4 10:14:24 2022	Sun Dec 4 10:14:24 2022	Sun Dec 4 10:14:24 2022

### (3) rmdir

```
[root@ext2 show]# rmdir dir
Congratulations! dir is deleted!
```

```
[root@ext2 show]# dir
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
Directory	.	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:15:44 2022
Directory	..	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022

### (4) create

```
[root@ext2 show]# create file
Congratulations! file is created
```

```
[root@ext2 show]# dir
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
Directory	.	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:14:13 2022	Sun Dec 4 10:15:44 2022
Directory	..	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022	Sat Dec 3 12:05:27 2022
File	file	Sun Dec 4 10:17:15 2022	Sun Dec 4 10:17:15 2022	Sun Dec 4 10:17:15 2022

## (5) delete

```
[root@ext2 show]# delete file
Congratulations! file is deleted!

[root@ext2 show]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sun Dec  4 10:14:13 2022    Sun Dec  4 10:14:13 2022    Sun Dec  4 10:18:47 2022
Directory ..       Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022
```

## (6) cd

```
[root@ext2]# cd show

[root@ext2 show]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sun Dec  4 10:14:13 2022    Sun Dec  4 10:14:13 2022    Sun Dec  4 10:18:47 2022
Directory ..       Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022
[root@ext2 show]# cd ../

[root@ext2]#
```

## (7) attrib

```
[root@ext2]# attrib p1 r
Modify successfully!

[root@ext2]# dir -m
Type      FileName      CreateTime      LastAccessTime      ModifyTime      Authority
Directory .              Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    NULL
Directory ..       Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    Sat Dec  3 12:05:27 2022    NULL
File      p              Sat Dec  3 12:05:35 2022    Sat Dec  3 12:08:02 2022    Sat Dec  3 12:05:35 2022    r
File      p1             Sat Dec  3 12:05:45 2022    Sat Dec  3 12:07:15 2022    Sat Dec  3 12:05:45 2022    r
File      p2             Sat Dec  3 12:05:49 2022    Sat Dec  3 12:07:20 2022    Sat Dec  3 12:05:49 2022    rw
File      p3             Sat Dec  3 12:05:54 2022    Sat Dec  3 12:07:22 2022    Sat Dec  3 12:05:54 2022    rw
File      p4             Sat Dec  3 12:05:56 2022    Sat Dec  3 12:07:23 2022    Sat Dec  3 12:05:56 2022    rw
File      p5             Sat Dec  3 12:05:57 2022    Sat Dec  3 12:07:25 2022    Sat Dec  3 12:05:57 2022    rw
File      p6             Sat Dec  3 12:05:59 2022    Sat Dec  3 12:07:26 2022    Sat Dec  3 12:05:59 2022    rw
File      p7             Sat Dec  3 12:06:01 2022    Sat Dec  3 12:07:28 2022    Sat Dec  3 12:06:01 2022    rw
File      p8             Sat Dec  3 12:06:03 2022    Sat Dec  3 12:07:29 2022    Sat Dec  3 12:06:03 2022    rw
File      p9             Sat Dec  3 12:06:05 2022    Sat Dec  3 12:07:31 2022    Sat Dec  3 12:06:05 2022    rw
File      p10            Sat Dec  3 12:06:07 2022    Sat Dec  3 12:07:33 2022    Sat Dec  3 12:06:07 2022    rw
File      p11            Sat Dec  3 12:06:10 2022    Sat Dec  3 12:07:34 2022    Sat Dec  3 12:06:10 2022    rw
File      p12            Sat Dec  3 12:06:12 2022    Sat Dec  3 12:07:37 2022    Sat Dec  3 12:06:12 2022    rw
File      p13            Sat Dec  3 12:06:15 2022    Sat Dec  3 12:07:39 2022    Sat Dec  3 12:06:15 2022    rw
File      p14            Sat Dec  3 12:06:17 2022    Sat Dec  3 12:07:40 2022    Sat Dec  3 12:06:17 2022    rw
File      p15            Sat Dec  3 12:06:19 2022    Sat Dec  3 12:07:41 2022    Sat Dec  3 12:06:19 2022    rw
File      p16            Sat Dec  3 12:06:21 2022    Sat Dec  3 12:07:43 2022    Sat Dec  3 12:06:21 2022    rw
Directory show      Sun Dec  4 10:14:13 2022    Sun Dec  4 10:14:13 2022    Sun Dec  4 10:18:47 2022    NULL
```

## (8) open



```
[root@ext2]# open p  
Open successfully!
```

```
[root@ext2]# open p16  
Failed! The file_open_table is full
```

(9) close

```
[root@ext2]# close p1  
Close successfully!
```

(10) read

```
[root@ext2]# read p1  
  
asdsadsadasdsadasd
```

读限制

```
[root@ext2]# attrib f1 w  
Modify successfully!  
  
[root@ext2]# read f1  
Failed! The file can't be read
```

(11) write

```
[root@ext2]# write p1  
  
asdsadsadasdsadasd
```

写限制

```
[root@ext2]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sun Dec  4 11:12:46 2022      Sun Dec  4 11:12:46 2022      Sun Dec  4 11:12:46 2022
Directory ..             Sun Dec  4 11:12:46 2022      Sun Dec  4 11:12:46 2022      Sun Dec  4 11:12:46 2022
[root@ext2]# create f1
Congratulations! f1 is created

[root@ext2]# create f2
Congratulations! f2 is created

[root@ext2]# attrib f1 r
Modify successfully!

[root@ext2]# write f1
Failed! The file can't be written
```

ps: 在读写命令下达时, open 函数默认执行, 也就是说读写进程不能超过 16 个

(12) login

```
Do you want to logout from filesystem?[Y/N]y
[no user]# login
please input the password(init:123):123

[root@ext2]#
```

(13) logout

```
[root@ext2]# logout
Do you want to logout from filesystem?[Y/N]y
[no user]#
```

(14) exit

```
[root@ext2]# exit
Do you want to exit from filesystem?[Y/N]y
Thank you for using!
```

(15) format

```
[root@ext2]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sun Dec 4 11:02:29 2022 Sun Dec 4 11:02:29 2022 Sun Dec 4 11:02:29 2022
Directory ..             Sun Dec 4 11:02:29 2022 Sun Dec 4 11:02:29 2022 Sun Dec 4 11:02:29 2022
File      p1              Sun Dec 4 11:02:37 2022 Sun Dec 4 11:06:05 2022 Sun Dec 4 11:05:59 2022
File      p2              Sun Dec 4 11:02:40 2022 Sun Dec 4 11:04:21 2022 Sun Dec 4 11:02:40 2022
File      p              Sun Dec 4 11:02:44 2022 Sun Dec 4 11:04:18 2022 Sun Dec 4 11:02:44 2022
File      p3              Sun Dec 4 11:02:49 2022 Sun Dec 4 11:04:23 2022 Sun Dec 4 11:02:49 2022
File      p4              Sun Dec 4 11:02:52 2022 Sun Dec 4 11:04:25 2022 Sun Dec 4 11:02:52 2022
File      p5              Sun Dec 4 11:02:54 2022 Sun Dec 4 11:04:26 2022 Sun Dec 4 11:02:54 2022
File      p6              Sun Dec 4 11:02:55 2022 Sun Dec 4 11:04:27 2022 Sun Dec 4 11:02:55 2022
File      p7              Sun Dec 4 11:02:56 2022 Sun Dec 4 11:04:29 2022 Sun Dec 4 11:02:56 2022
File      p8              Sun Dec 4 11:02:57 2022 Sun Dec 4 11:04:31 2022 Sun Dec 4 11:02:57 2022
File      p9              Sun Dec 4 11:02:58 2022 Sun Dec 4 11:04:33 2022 Sun Dec 4 11:02:58 2022
File      p10             Sun Dec 4 11:03:01 2022 Sun Dec 4 11:04:34 2022 Sun Dec 4 11:03:01 2022
File      p11             Sun Dec 4 11:03:03 2022 Sun Dec 4 11:04:37 2022 Sun Dec 4 11:03:03 2022
File      p12             Sun Dec 4 11:03:04 2022 Sun Dec 4 11:04:38 2022 Sun Dec 4 11:03:04 2022
File      p13             Sun Dec 4 11:03:05 2022 Sun Dec 4 11:04:39 2022 Sun Dec 4 11:03:05 2022
File      p14             Sun Dec 4 11:03:07 2022 Sun Dec 4 11:04:41 2022 Sun Dec 4 11:03:07 2022
File      p15             Sun Dec 4 11:03:09 2022 Sun Dec 4 11:04:42 2022 Sun Dec 4 11:03:09 2022
File      p16             Sun Dec 4 11:03:10 2022 Sun Dec 4 11:03:10 2022 Sun Dec 4 11:03:10 2022
File      p17             Sun Dec 4 11:03:12 2022 Sun Dec 4 11:03:12 2022 Sun Dec 4 11:03:12 2022
File      p18             Sun Dec 4 11:03:14 2022 Sun Dec 4 11:03:14 2022 Sun Dec 4 11:03:14 2022

[root@ext2]# format
Do you want to format the filesystem?
It will be dangerous to your data.
[Y/N]y

[root@ext2]# dir
Type      FileName      CreateTime      LastAccessTime      ModifyTime
Directory .              Sun Dec 4 11:12:46 2022 Sun Dec 4 11:12:46 2022 Sun Dec 4 11:12:46 2022
Directory ..             Sun Dec 4 11:12:46 2022 Sun Dec 4 11:12:46 2022 Sun Dec 4 11:12:46 2022
[root@ext2]#
```

(16) password

```
[root@ext2]# password
Please input the old password
123
Please input the new password:jc
Modify the password?[Y/N]y

[root@ext2]# ewxit
Failed! Command not available

[root@ext2]# exit
Do you want to exit from filesystem?[Y/N]y
Thank you for using!
○ [root@VM-8-12-centos operating_system_3]# ./main
Hello! Welcome to Ext2_like file system!
please input the password(init:123):jc

[root@ext2]#
```

## 3.5 程序运行初值及运行结果分析

本实验没有该项

## 3.6 实验总结

### 3.6.1 实验中的问题及解决过程

本实验主要遇到的问题是修改指导书中的错误，指导书中错误的地方很多，看完整个代码修改正确后，才开始进行功能扩充工作。

并且再次用到联合编译，让文件系统代码比较清晰，便于封装

对于 getch () 函数中输入输出模式感到疑惑，使用了<termios.h>库来实现，通过查找 csdn 了解

包括其他对文件操作 fseek, fread, fwrite 等函数也是通过查找了解

### 3.6.2 实验收获

学会了看千行代码的能力，学会把代码拆开封装，便于修改和阅读，对文件的操作也更加熟练

更好地学习了文件系统知识，了解了 EXT2 文件系统的工作机制

很大地加强了代码能力

### 3.6.3 意见与建议

实验要求可以增加一个多用户的实现，在我的代码中，只要把 inode 稍作修改就可以实现多用户的文件系统，同时多个用户进入，各自拥有独立的文件打开表，而且用户结构体也比较好定义，只需要账户名和密码就可以进入

## 3.7 附件

### 3.7.1 附件 1 程序

### 3.7.1 附件 2 Readme