西安交通大学

操作系统专题实验报告

班级：计算机 2101

学号：2215015058

姓名：陈实

2023 年 12 月 02 日

目录

1 openEuler 系统环境实验

1.1 进程相关编程实验

1.1.1 实验目的

1. 熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看系统基本信息以了解系统；
2. 编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管理等机制在操作系统实际运行中的作用。

1.1.2 实验内容

1. 熟悉操作命令、编辑、编译、运行程序。完成图 1 程序的运行验证，多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。图 1-1 教材中所给代码

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

int main()
{
pid_t pid, pid1;

   /* fork a child process */
   pid = fork();

   if (pid < 0) { /* error occurred */
     fprintf(stderr, "Fork Failed");
     return 1;
   }
   else if (pid == 0) { /* child process */
     pid1 = getpid();
     printf("child: pid = %d",pid); /* A */
     printf("child: pid1 = %d",pid1); /* B */
   }
   else { /* parent process */
     pid1 = getpid();
     printf("parent: pid = %d",pid); /* C */
     printf("parent: pid1 = %d",pid1); /* D */
     wait(NULL);
   }

   return 0;
}
```

图 1: 教材中所给代码（p103 作业 3.7）

2. 扩展图 1 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；
   (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；

1.1.3 实验思想

本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，进一步加入 wait() 函数分析其作用。

1.1.4 实验步骤

1. 编写并多次运行图 1-1 中代码
2. 删去图 1-1 代码中的 wait() 函数并多次运行程序，分析运行结果。
3. 修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作（自行设计），观察并解释所做操作和输出结果
4. 在步骤三基础上，在 return 前增加对全局变量的操作（自行设计）并输出结果，观察并解释所做操作和输出结果。
5. 修改图 1-1 程序，在子进程中调用 `system` 与 `exec` 族函数。编写 `system_call.c` 文件输出进程号 PID，编译后生成 `system_call` 可执行文件。在子进程中调用 `system_call`，观察输出结果并分析总结。

1.1.5 测试数据设计

无测试数据

1.1.6 程序运行初值及运行结果分析

1. 编译并运行图 1-1 中代码，运行结果如下：
   观察输出，总是先输出子进程的 pid，因为有 wait(NULL), 父进程会等带子进程结束。以第一排输出为例在父进程中:
   pid: 存放 fork() 的返回值, 也就是子进程的 PID。
   pid1: 存放 getpid() 的返回值, 也就是父进程自身的 PID。
   在子进程中:
   pid: 存放 fork() 的返回值 0, 表示当前是子进程。
   pid1: 存放 getpid() 的返回值, 也就是子进程自身的 PID。

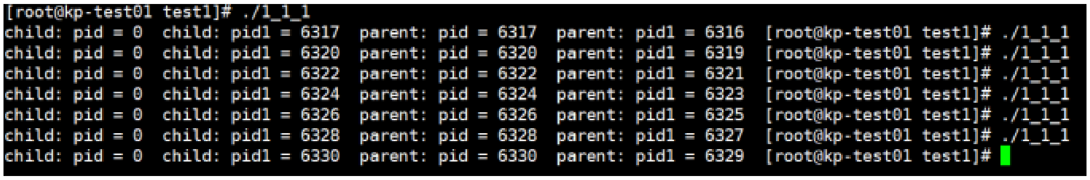


图 2: 编译并运行图 1-1 中代码

2. 删去图 1-1 代码中的 wait() 函数并多次运行程序

   删除 wait 函数，观察父进程和子进程的输出顺序。输出顺序变化的原因是父进程和子进程的执行顺序不确定，取决于系统的调度算法。

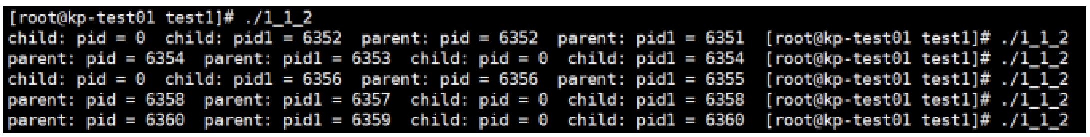


图 3: 删去图 1-1 代码中的 wait() 函数并多次运行程序

3. 修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作

   设置了一个全局变量 $value = 0$，父进程 $value- = 100$，子进程 $value+ = 100$，都输出 value 的值和地址。观察父进程和子进程的输出结果。父进程和子进程的 value 值不同，说明父进程和子进程的数据是独立的，但 value 的地址相同，说明父进程和子进程的地址空间是共享的。

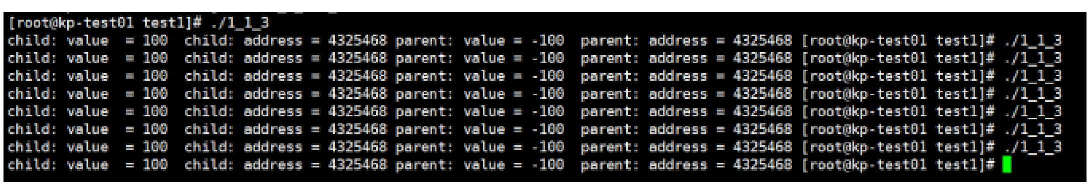


图 4: 修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作

4. 在步骤三基础上，在 return 前增加对全局变量的操作（自行设计）并输出结果：

   父进程和子进程的 value 值不同，说明父进程和子进程的数据是独立的，但 value 的地址相同，说明父进程和子进程的地址空间是共享的。

```
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
before return : value = -200  before return : address = 4325460
child: value = 100  child: address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]#
```

图 5: 在步骤三基础上，在 return 前增加对全局变量的操作

5. 修改图 1-1 程序，在子进程中调用 `system` 与 `exec` 族函数。`system()` 函数：用于在程序中执行另一个可执行文件，与当前进程并行执行

   `exec()` 族函数：用于在程序中执行另一个可执行文件，并将当前进程替换为新的进程 以第一次

```
[root@kp-test01 test1]# ./1_1_5
parent: pid =2596
child: pid= 0
parent: pid1 =2595
child: pid1 = 2596
process: pid = 2596
[root@kp-test01 test1]# ./1_1_5
parent: pid =2598
child: pid= 0
parent: pid1 =2597
child: pid1 = 2598
process: pid = 2598
[root@kp-test01 test1]# ./1_1_5
parent: pid =2600
child: pid= 0
parent: pid1 =2599
child: pid1 = 2600
process: pid = 2600
[root@kp-test01 test1]# ./1_1_5
parent: pid =2602
child: pid= 0
parent: pid1 =2601
child: pid1 = 2602
process: pid = 2602
```

图 6: 使用 execl 函数

输出为例，子进程的 pid 为 2596，父进程的 pid 为 2595，`system_call` 的 pid 为 2596，与子进程的 pid 相同。因为 `execl()` 函数会用 `system_call` 可执行文件替换当前进程，所以 `system_call` 的 pid 与子进程的 pid 相同。以第一次运行为例，parent 的 pid 为 3128，child 的 pid 为 3129，`system_call` 的 pid 为 3130，因为 `system()` 函数会新建一个进程，所以 `system_call` 的 pid 与 child 的 pid 不同。

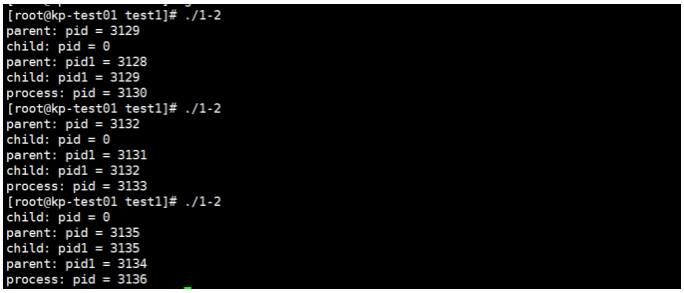


图 7: 使用 system 函数

1.2 线程相关编程实验

1.2.1 实验目的

1. 探究多线程编程中的线程共享进程信息。在计算机编程中，多线程是一种常见的并发编程方式，允许程序在同一进程内创建多个线程，从而实现并发执行。由于这些线程共享同一进程的资源，包括内存空间和全局变量，因此可能会出现线程共享进程信息的现象。本实验旨在通过创建多个线程并使其共享进程信息，以便深入了解线程共享资源时可能出现的问题。

1.2.2 实验内容

1. 在进程中给一变量赋初值并成功创建两个线程；
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（自行设计）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。

1.2.3 实验思想

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况下，引入互斥和同步机制来确保线程间的正确协同操作。

1.2.4 实验步骤

1. 设计程序，创建两个子线程，两线程分别对同一个共享变量多次操作，观察输出结果。
2. 修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。运行程序，观察最终共享变量的值。

3. 在第一部分实验了解了 system() 与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

1.2.5 测试数据设计

定义共享变量初始值为 0，两个线程分别对其进行 100000 次 +/- 100 操作，最终在主进程中输出处理后的变量值需要操作次数大才能体现出明显区别

1.2.6 程序运行初值及运行结果分析

1. 设计程序，创建两个子线程，两线程分别对同一个共享变量多次操作，观察输出结果
   由于两个线程没有处理共享变量的同步问题，由于两个线程对 sharedVariable 的操作是并发的，会同时去操作共享变量，所以最后的结果不确定。

```
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1198031539
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -190265009
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -444022726
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1658028857
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1888172259
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -1087440094
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 337864100
[root@kp-test01 test1]#
```

图 8: 编译并运行图 1-1 中代码

2. 修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。运行程序，观察最终共享变量的值。
   相较于步骤 1，添加了信号量 semaphore。sem_wait() 函数会对信号量 semaphore 进行 P 操作，sem_post() 函数会对信号量 semaphore 进行 V 操作。由于 sem_wait() 和 sem_post() 函数是原子操作，所以最后的结果为 0。由于 sem_wait() 和 sem_post() 函数是原子操作，每次只有一个线程能够对共享变量进行操作，两个线程对操作是相反的，所以最后的结果为 0。

```
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
Final sharedVariable: 0
[root@kp-test01 test1]#
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_2
Thread 1 created successfully.
Thread 2 created successfully.
```

图 9: 编译并运行图 1-1 中代码

3. 在第一部分实验了解了 system() 与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

   线程 1 和线程 2 以及他们调用 system_call2 的 4 个 TID 都不同，说明线程的 TID 是独立的。两个线程的 PID 相同，说明两个线程是同一个进程的两个线程。它们调用的 system_call2 的 PID 互不相同且与线程的 PID 不同，说明调用 system() 函数会新建一个进程。

   可以发现只调用成功了一次 system_call2，新的进程的 PID 与线程的 PID 相同，说明调用

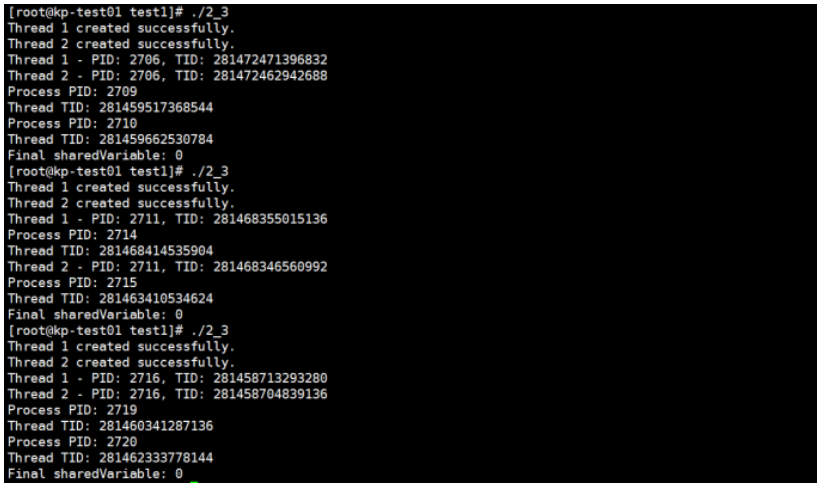


图 10: system 函数

execl() 函数会取代当前进程。同时，只输出了 system_call2 的 PID，因为线程属于同一个进程，当一个线程调用 execl() 函数时，会取代整个进程，所以另一个线程的代码不会执行。

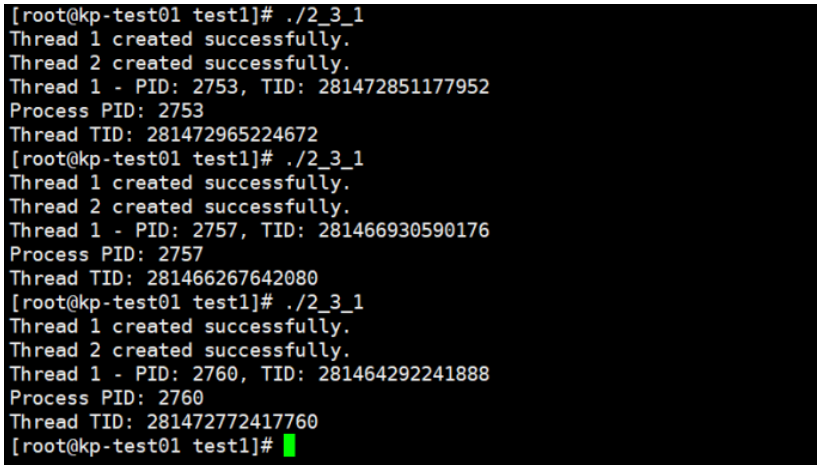


图 11: exec 函数

1.3 自旋锁相关编程实验

1.3.1 实验目的

1. 了解自旋锁的基本概念：通过研究自旋锁的工作原理和特点，深入理解自旋锁相对于其他锁机制的优势和局限性；
2. 实验自旋锁的应用：在一个多线程的实验环境中，设计一个竞争资源的场景，让多个线程同时竞争对该资源的访问；
3. 实现自旋锁的同步：使用自旋锁来保护竞争资源的访问，确保同一时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

1.3.2 实验内容

1. 在进程中给一变量赋初值并成功创建两个线程；
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（自行设计）并输出结果；
3. 使用自旋锁实现互斥和同步；

1.3.3 实验思想

本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程间的同步，

1.3.4 实验步骤

1. 根据实验内容要求，编写模拟自旋锁程序代码 spinlockc
2. 补充完成代码后，编译并运行程序，分析运行结果

1.3.5 测试数据设计

无测试数据

1.3.6 程序运行初值及运行结果分析

由于自旋锁的存在，每次只能有一个线程对共享变量进行操作，所以最后的结果为 10000。
在代码实现中是使用了互斥锁，当一个线程对共享变量进行操作时，会对互斥锁进行加锁，其他线程对共享变量进行操作时，会对互斥锁进行加锁，但是加锁失败，所以会一直循环等待，直到加锁成功。

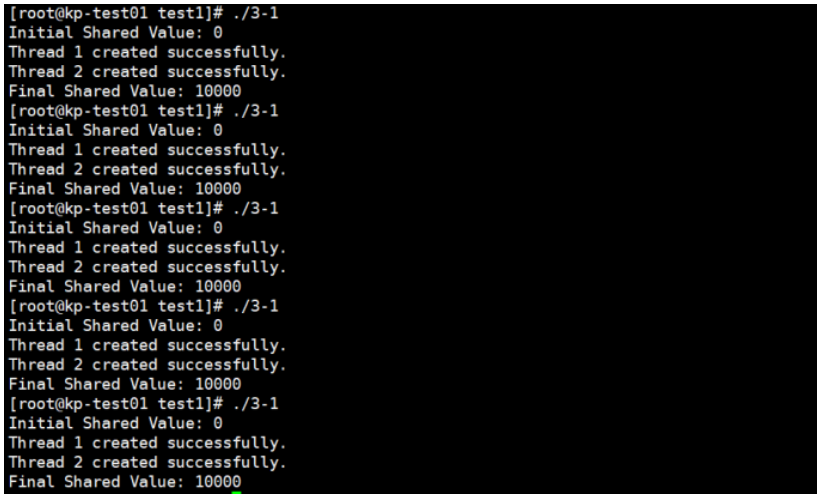


图 12: 自旋锁

1.4 实验总结

1.4.1 实验中的问题与解决过程

1. Linux 系统中的输出缓存区的输出与 Windows 系统的不同，通过查询资料，在 Linux 系统中，printf 并不会马上将数据直接输出到屏幕，而是将输出结果暂存在缓冲中，当缓冲区满、刷新缓冲区或程序结束时，缓冲区的内容才会进行输出。

1.4.2 实验收获

1. 对进程和线程的理解更加的深刻
2. 了解了如何在程序中调用其他可执行程序，通过 system 函数以及 execl 函数

1.4.3 意见与建议

将常见问题总结添加到实验指导书中

1.5 附件

1.5.1 附件 1 实验 1.1 代码

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

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d", pid);
        printf("child: pid1 = %d", pid1);
    }
    else{
        pid1 = getpid();
        printf("parent: pid = %d", pid);
        printf("parent: pid1 = %d/n", pid1);
        wait(NULL);
    }
    return 0;
}
```

1.5.2 附件 2 实验 1.2 代码

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

#define NUM_THREADS 2
```

```
5  #define NUM_ITERATIONS 100000
6
7  int sharedVariable = 0;
8
9  void *threadFunction1(void *arg) {
10     printf("thread1 created success!");
11     for (int i = 0; i < NUM_ITERATIONS; i++) {
12         sharedVariable+=i;
13     }
14     pthread_exit(NULL);
15 }
16
17 void *threadFunction2(void *arg) {
18     printf("thread2 created success!");
19     for (int i = 0; i < NUM_ITERATIONS; i++) {
20         sharedVariable-=i;
21     }
22     pthread_exit(NULL);
23 }
24
25 int main() {
26     pthread_t threads[NUM_THREADS];
27
28     if (pthread_create(&threads[0], NULL,
29                 threadFunction1, NULL) != 0) {
30         perror("pthread_create");
31         return 1;
32     }
33
34     if (pthread_create(&threads[1], NULL,
35                 threadFunction2, NULL) != 0) {
36         perror("pthread_create");
37         return 1;
38     }
```

```
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Final sharedVariable: %d\n", sharedVariable);

    return 0;
}
```

1.5.3 附件 3 实验 1.3 代码

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

typedef struct {
    int flag;
} spinlock_t;

// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}

// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
        // 自旋等待
    }
}

// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) {
    __sync_lock_release(&lock->flag);
}

// 共享变量
```

```
26 int shared_value = 0;
27
28 // 线程函数
29 void *thread_function(void *arg) {
30     spinlock_t *lock = (spinlock_t *)arg;
31     for (int i = 0; i < 5000; ++i) {
32         spinlock_lock(lock);
33         shared_value++;
34         spinlock_unlock(lock);
35     }
36     return NULL;
37 }
38
39 int main() {
40     pthread_t thread1, thread2;
41     spinlock_t lock;
42     // 初始化自旋锁
43     spinlock_init(&lock);
44     // 创建两个线程
45     pthread_create(&thread1, NULL, thread_function, &lock);
46     pthread_create(&thread2, NULL, thread_function, &lock);
47     // 等待线程结束
48     pthread_join(thread1, NULL);
49     pthread_join(thread2, NULL);
50     // 输出共享变量的值
51     printf("Shared␣Value:␣%d\n", shared_value);
52
53     return 0;
54 }
```

1.5.4 附件 4 README

README.md 2023-10-25

Operating System Lab

1. 进程相关编程实验

步骤1

创建一个子进程，子进程输出自己的pid和父进程的pid，父进程输出自己的pid和子进程的pid。当pid<0时，说明创建子进程失败，当pid=0时，说明是子进程，当pid>0时，说明是父进程。

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

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }
    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
        wait(NULL);
    }
    return 0;
}
```

观察输出，总是先输出子进程的pid，因为有`wait(NULL)`,父进程会等带子进程结束。以第一排输出为例

```
在父进程中：

pid：存放fork()的返回值，也就是子进程的PID。

pid1：存放getpid()的返回值，也就是父进程自身的PID。

在子进程中：

pid：存放fork()的返回值0，表示当前是子进程。
```

pid1: 存放getpid()的返回值,也就是子进程自身的PID。

```
[root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6317  parent: pid = 6317  parent: pid1 = 6316  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6320  parent: pid = 6320  parent: pid1 = 6319  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6322  parent: pid = 6322  parent: pid1 = 6321  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6324  parent: pid = 6324  parent: pid1 = 6323  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6326  parent: pid = 6326  parent: pid1 = 6325  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6328  parent: pid = 6328  parent: pid1 = 6327  [root@kp-test01 test1]# ./1_1_1
child: pid = 0  child: pid1 = 6330  parent: pid = 6330  parent: pid1 = 6329  [root@kp-test01 test1]#
```

步骤2

删除wait函数，观察父进程和子进程的输出顺序。输出顺序变化的原因是父进程和子进程的执行顺序不确定，取决于系统的调度算法。

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

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
    }
    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
    }
    return 0;
}
```

删除`wait(NULL)`后，child和parent的输出顺序不确定。原因是父进程和子进程的执行顺序不确定，取决于系统的调度算法。

```
[root@kp-test01 test1]# ./1_1_2
child: pid = 0  child: pid1 = 6352  parent: pid = 6352  parent: pid1 = 6351  [root@kp-test01 test1]# ./1_1_2
parent: pid = 6354  parent: pid1 = 6353  child: pid = 0  child: pid1 = 6354  [root@kp-test01 test1]# ./1_1_2
child: pid = 0  child: pid1 = 6356  parent: pid = 6356  parent: pid1 = 6355  [root@kp-test01 test1]# ./1_1_2
parent: pid = 6358  parent: pid1 = 6357  child: pid = 0  child: pid1 = 6358  [root@kp-test01 test1]# ./1_1_2
parent: pid = 6360  parent: pid1 = 6359  child: pid = 0  child: pid1 = 6360  [root@kp-test01 test1]# ./1_1_2
```

步骤3

设置了一个全局变量value=0，父进程value-=100，子进程value+=100，都输出value的值和地址。

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

int value=0;

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        value += 100;
        pid1 = getpid();
        printf("child: value = %d  ", value);
        printf("child: address = %d  ", &value);
    }
    else{
        value -= 100;
        pid1 = getpid();
        printf("parent: value = %d  ", value);
        printf("parent: address = %d  ", &value);
    }
    return 0;
}
```

观察父进程和子进程的输出结果。父进程和子进程的value值不同，说明父进程和子进程的数据是独立的，但value的地址相同，说明父进程和子进程的地址空间是共享的。

```
[root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]# ./1_1_3
child: value  = 100  child: address = 4325468 parent: value = -100  parent: address = 4325468 [root@kp-test01 test1]#
```

步骤4

在步骤三的基础上，在return前增加了value的输出，观察父进程和子进程的输出结果。

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

README.md 2023-10-25

```
int value=0;

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        value += 100;
        pid1 = getpid();
        printf("child: value = %d  ", value);
        printf("child: address = %d  \n", &value);
    }
    else{
        value -= 100;
        pid1 = getpid();
        printf("parent: value = %d  ", value);
        printf("parent: address = %d  \n", &value);
    }
    value*=2;
    printf("before return : value = %d  ", value);
    printf("before return : address = %d  \n", &value);
    return 0;
}
```

父进程和子进程的value值不同，说明父进程和子进程的数据是独立的，但value的地址相同，说明父进程和子进程的地址空间是共享的。

```
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
before return : value = -200  before return : address = 4325460
child: value = 100  child: address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]# ./1_1_4
parent: value = -100  parent: address = 4325460
child: value = 100  child: address = 4325460
before return : value = -200  before return : address = 4325460
before return : value = 200  before return : address = 4325460
[root@kp-test01 test1]#
```

步骤5

1. 用`touch system_call.c`生成一个system_call.c文件，用`gcc system_call.c -o system_call`编译生成一个system_call可执行文件。在步骤三的基础上，用execl函数调用system_call可执行文件。execl() 函数可以用于在程序中执行另一个可执行文件，并将当前进程替换为新的进程。

system_call.c

```c
#include<stdio.h>
#include<unistd.h>

int main(){
        pid_t pid=getpid();
        printf("process: pid = %d\n",pid);
        return 0;
}
```

execl函数

README.md 2023-10-25

1. execl()参数列表如下： `int execl(const char *path, const char *arg0, ... /* (char *)0 */);`
    1. 第一个参数 const char *path：指定要执行的可执行文件的路径。在这里，"./system_call" 表示当前目录下的 system_call 可执行文件。
    2. 第二个参数 const char *arg0, ...：可选的参数列表，以 NULL 结束。在这里，我们将 "system_call" 作为第二个参数传递给可执行文件。
    3. 最后一个参数必须是 NULL，用于标记参数列表的结束。

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

int main()
{
  pid_t pid, pid1;

  pid = fork();

  if(pid<0)
     {
       fprintf(stderr,"Fork Failed");
       return 1;
     }
  else if (pid ==0)
  {
     pid1 = getpid();
     printf("child: pid= %d\n",pid);
     printf("child: pid1 = %d\n",pid1);
     execl("./system_call","",NULL);
  }
  else
  {
    pid1 = getpid();
    printf("parent: pid =%d\n",pid);
    printf("parent: pid1 =%d\n",pid1);
    wait(NULL);
    }
   return 0;
  }
```

以第一次输出为例，子进程的pid为2596，父进程的pid为2595，system_call的pid为2596，与子进程的pid相同。因为execl()函数会用system_call可执行文件替换当前进程，所以system_call的pid与子进程的pid相同。

```
[root@kp-test01 test1]# ./1_1_5
parent: pid =2596
child: pid= 0
parent: pid1 =2595
child: pid1 = 2596
process: pid = 2596
[root@kp-test01 test1]# ./1_1_5
parent: pid =2598
child: pid= 0
parent: pid1 =2597
child: pid1 = 2598
process: pid = 2598
[root@kp-test01 test1]# ./1_1_5
parent: pid =2600
child: pid= 0
parent: pid1 =2599
child: pid1 = 2600
process: pid = 2600
[root@kp-test01 test1]# ./1_1_5
parent: pid =2602
child: pid= 0
parent: pid1 =2601
child: pid1 = 2602
process: pid = 2602
```

system函数

加上`#include<stdlib.h>`,使用system()函数调用system_call可执行文件。system() 函数可以用于在程序中执行另一个可执行文件，并将当前进程替换为新的进程。 传递`system("./system_call")`给system()函数，表示在当前目录下执行system_call可执行文件。

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

int main(){
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0){
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0){
        pid1 = getpid();
        printf("child: pid = %d  ", pid);
        printf("child: pid1 = %d  ", pid1);
        system("./system_call");
    }
    else{
        pid1 = getpid();
        printf("parent: pid = %d  ", pid);
        printf("parent: pid1 = %d  ", pid1);
```

```
            wait(NULL);
        }
        return 0;
    }
```

以第一次运行为例，parent的pid为3128，child的pid为3129，systemcall的pid为3130，因为system()函数会新建一个进程，所以systemcall的pid与child的pid不同。

```
[root@kp-test01 test1]# ./1-2
parent: pid = 3129
child: pid = 0
parent: pid1 = 3128
child: pid1 = 3129
process: pid = 3130
[root@kp-test01 test1]# ./1-2
parent: pid = 3132
child: pid = 0
parent: pid1 = 3131
child: pid1 = 3132
process: pid = 3133
[root@kp-test01 test1]# ./1-2
child: pid = 0
parent: pid = 3135
child: pid1 = 3135
parent: pid1 = 3134
process: pid = 3136
```

==所以对比`execl()`和`system()`,`execl()`会取代当前进程，其后面的程序不会执行，而`system()`会新建一个进程，与原来的进程并行执行。==

2. 线程相关编程实验

步骤1

创建了两个线程，一个全局共享变量sharedVariable，一个线程对sharedVariable进行加100操作，一个线程对sharedVariable进行减100操作，分别执行100000次。最后输出sharedVariable的值。

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

#define NUM_THREADS 2
#define NUM_ITERATIONS 100000

int sharedVariable = 0;

void *threadFunction1(void *arg) {
    printf("thread1 created success!");
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sharedVariable+=i;
    }
    pthread_exit(NULL);
}
```

```c
void *threadFunction2(void *arg) {
    printf("thread2 created success!");
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sharedVariable-=i;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    if (pthread_create(&threads[0], NULL, threadFunction1, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }

    if (pthread_create(&threads[1], NULL, threadFunction2, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Final sharedVariable: %d\n", sharedVariable);

    return 0;
}
```

由于两个线程没有处理共享变量的同步问题，由于两个线程对sharedVariable的操作是并发的，会同时去操作共享变量，所以最后的结果不确定。

```
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1198031539
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -190265009
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -444022726
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1658028857
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 1888172259
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: -1087440094
[root@kp-test01 test1]# ./2_1
thread1 created success!thread2 created success!Final sharedVariable: 337864100
[root@kp-test01 test1]#
```

步骤2

相较于步骤1，添加了信号量semaphore。sem_wait()函数会对信号量semaphore进行P操作，sem_post()函数会对信号量semaphore进行V操作。由于sem_wait()和sem_post()函数是原子操作，所以最后的结果为0。

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM_THREADS 2
#define NUM_ITERATIONS 100000

int sharedVariable = 0;
sem_t semaphore;

void *threadFunction1(void *arg) {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable += 100;
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

void *threadFunction2(void *arg) {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable -= 100;
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    if (sem_init(&semaphore, 0, 1) != 0) {
        perror("sem_init");
        return 1;
    }

    if (pthread_create(&threads[0], NULL, threadFunction1, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 1 created successfully.\n");
    }

    if (pthread_create(&threads[1], NULL, threadFunction2, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 2 created successfully.\n");
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
```

```
    }

    printf("Final sharedVariable: %d\n", sharedVariable);

    sem_destroy(&semaphore);

    return 0;
}
```

由于sem_wait()和sem_post()函数是原子操作，每次只有一个线程能够对共享变量进行操作，两个线程对操作是相反的，所以最后的结果为0。

步骤3

创建了一个system_call2.c文件，用于输出进程的PID和线程的TID。

system_call2.c

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

int main() {
    pid_t pid = getpid(); // 获取进程的PID
    pthread_t tid = pthread_self(); // 获取线程的TID

    printf("Process PID: %d\n", pid);
    printf("Thread TID: %lu\n", tid);

    return 0;
}
```

system()函数

README.md 2023-10-25

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <unistd.h>

#define NUM_THREADS 2
#define NUM_ITERATIONS 100000

int sharedVariable = 0;
sem_t semaphore;

void *threadFunction1(void *arg) {
    pid_t thread_pid = getpid();
    pthread_t thread_tid = pthread_self();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable += 100;
        sem_post(&semaphore);
    }
    printf("Thread 1 - PID: %d, TID: %lu\n", thread_pid, thread_tid);
    system("./system_call2");
    pthread_exit(NULL);
}

void *threadFunction2(void *arg) {
    pid_t thread_pid = getpid();
    pthread_t thread_tid = pthread_self();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable -= 100;
        sem_post(&semaphore);
    }
    printf("Thread 1 - PID: %d, TID: %lu\n", thread_pid, thread_tid);
    system("./system_call2");
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    if (sem_init(&semaphore, 0, 1) != 0) {
        perror("sem_init");
        return 1;
        }

    if (pthread_create(&threads[0], NULL, threadFunction1, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 1 created successfully.\n");
    }
```

```
    if (pthread_create(&threads[1], NULL, threadFunction2, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 2 created successfully.\n");
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final sharedVariable: %d\n", sharedVariable);

    sem_destroy(&semaphore);

    return 0;
}
```

线程1和线程2以及他们调用system_call2的4个TID都不同，说明线程的TID是独立的。两个线程的PID相同，说明两个线程是同一个进程的两个线程。它们调用的system_call2的PID互不相同且与线程的PID不同，说明调用system()函数会新建一个进程。

```
[root@kp-test01 test1]# ./2_3
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2706, TID: 281472471396832
Thread 2 - PID: 2706, TID: 281472462942688
Process PID: 2709
Thread TID: 281459517368544
Process PID: 2710
Thread TID: 281459662530784
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_3
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2711, TID: 281468355015136
Process PID: 2714
Thread TID: 281468414535904
Thread 2 - PID: 2711, TID: 281468346560992
Process PID: 2715
Thread TID: 281463410534624
Final sharedVariable: 0
[root@kp-test01 test1]# ./2_3
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2716, TID: 281458713293280
Thread 2 - PID: 2716, TID: 281458704839136
Process PID: 2719
Thread TID: 281460341287136
Process PID: 2720
Thread TID: 281462333778144
Final sharedVariable: 0
```

execl()函数

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
```

```c
#include <unistd.h>

#define NUM_THREADS 2
#define NUM_ITERATIONS 100000

int sharedVariable = 0;
sem_t semaphore;

void *threadFunction1(void *arg) {
    pid_t thread_pid = getpid();
    pthread_t thread_tid = pthread_self();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable += 100;
        sem_post(&semaphore);
    }
    printf("Thread 1 - PID: %d, TID: %lu\n", thread_pid, thread_tid);
    execl("./system_call2","",NULL);
    pthread_exit(NULL);
}

void *threadFunction2(void *arg) {
    pid_t thread_pid = getpid();
    pthread_t thread_tid = pthread_self();
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        sem_wait(&semaphore);
        sharedVariable -= 100;
        sem_post(&semaphore);
    }
    printf("Thread 1 - PID: %d, TID: %lu\n", thread_pid, thread_tid);
    execl("./system_call2","",NULL);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];

    if (sem_init(&semaphore, 0, 1) != 0) {
        perror("sem_init");
        return 1;
        }

    if (pthread_create(&threads[0], NULL, threadFunction1, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 1 created successfully.\n");
    }

    if (pthread_create(&threads[1], NULL, threadFunction2, NULL) != 0) {
        perror("pthread_create");
        return 1;
    } else {
        printf("Thread 2 created successfully.\n");
```

```
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Final sharedVariable: %d\n", sharedVariable);

    sem_destroy(&semaphore);

    return 0;
}
```

可以发现只调用成功了一次system_call2，新的进程的PID与线程的PID相同，说明调用execl()函数会取代当前进程。同时，只输出了system_call2的PID，因为线程属于同一个进程，当一个线程调用execl()函数时，会取代整个进程，所以另一个线程的代码不会执行。

```
[root@kp-test01 test1]# ./2_3_1
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2753, TID: 281472851177952
Process PID: 2753
Thread TID: 281472965224672
[root@kp-test01 test1]# ./2_3_1
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2757, TID: 281466930590176
Process PID: 2757
Thread TID: 281466267642080
[root@kp-test01 test1]# ./2_3_1
Thread 1 created successfully.
Thread 2 created successfully.
Thread 1 - PID: 2760, TID: 281464292241888
Process PID: 2760
Thread TID: 281472772417760
[root@kp-test01 test1]#
```

3. 自旋锁相关编程实验

步骤1

```
/**
 * spinlock.c
 * in xjtu
 * 2023.8
 */

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

```c
typedef struct {
    int flag;
} spinlock_t;

// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}

// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
        // 自旋等待
    }
}

// 释放自旋锁
void spinlock_unlock(spinlock_t *lock) {
    __sync_lock_release(&lock->flag);
}

// 共享变量
int shared_value = 0;

// 线程函数
void *thread_function(void *arg) {
    spinlock_t *lock = (spinlock_t *)arg;
    for (int i = 0; i < 5000; ++i) {
        spinlock_lock(lock);
        shared_value++;
        spinlock_unlock(lock);
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    spinlock_t lock;

    // 初始化自旋锁
    spinlock_init(&lock);

    // 创建两个线程
    pthread_create(&thread1, NULL, thread_function, &lock);
    pthread_create(&thread2, NULL, thread_function, &lock);

    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // 输出共享变量的值
    printf("Shared Value: %d\n", shared_value);

    return 0;
```

```
    }
```

由于自旋锁的存在，每次只能有一个线程对共享变量进行操作，所以最后的结果为10000。

```
[root@kp-test01 test1]# ./3-1
Initial Shared Value: 0
Thread 1 created successfully.
Thread 2 created successfully.
Final Shared Value: 10000
[root@kp-test01 test1]# ./3-1
Initial Shared Value: 0
Thread 1 created successfully.
Thread 2 created successfully.
Final Shared Value: 10000
[root@kp-test01 test1]# ./3-1
Initial Shared Value: 0
Thread 1 created successfully.
Thread 2 created successfully.
Final Shared Value: 10000
[root@kp-test01 test1]# ./3-1
Initial Shared Value: 0
Thread 1 created successfully.
Thread 2 created successfully.
Final Shared Value: 10000
[root@kp-test01 test1]# ./3-1
Initial Shared Value: 0
Thread 1 created successfully.
Thread 2 created successfully.
Final Shared Value: 10000
```

2 进程通信与内存管理

2.1 进程的软中断通信

2.1.1 实验目的

1. 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。

2.1.2 实验内容

1. 使用 man 命令查看 fork 、kill 、signal、sleep、exit 系统调用的帮助手册。
2. 根据流程图编制实现软中断通信的程序
3. 多次运行所写程序，比较 5s 内按下 Ctrl+\ 或 Ctrl+Delete 发送中断，或 5s 内不进行任何操作发送中断，分别会出现什么结果？分析原因。
4. 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，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。

2.1.3 实验思想

父进程会从键盘上接收到中断信号，然后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，执行对应的自定义操作。

2.1.4 实验步骤

运行三次程序，分别按下 Ctrl+\ 或 Ctrl+Delete 发送中断，或 5s 内不进行任何操作发送中断，观察运行结果。

2.1.5 测试数据设计

无测试数据

2.1.6 程序运行初值及运行结果分析

分别测试了 5s 内按下 Ctrl+\ 或 Ctrl+Delete 发送中断，或 5s 内不进行任何操作发送中断的情况。首先会输出收到的中断信号的值，然后输出子进程收到的中断信号的值，子进程获得对应软中断信号，执行对应的自定义操作，printf 输出对应提示信息，然后终止子进程。父进程调用 wait() 函数等待两个子进程终止后，输出提示信息，结束进程执行。

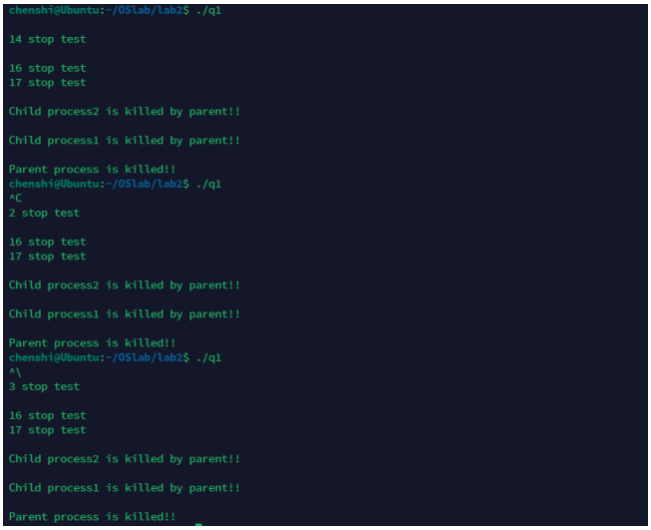


图 13: 父进程收到 3 种不同的中断信号

2.1.7 问题回答

1. 你最初认为运行结果会怎么样？写出你猜测的结果
   父进程收到中断信号后，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，执行对应的自定义操作，printf 输出对应提示信息。

2. 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。
   实际结果与预期结果一致。两个子进程输出的顺序不确定。接收不同的中断前没有区别，只有 printf 的中断值不同。接收不同中断后，子进程收到的中断值不同，执行的操作不同。

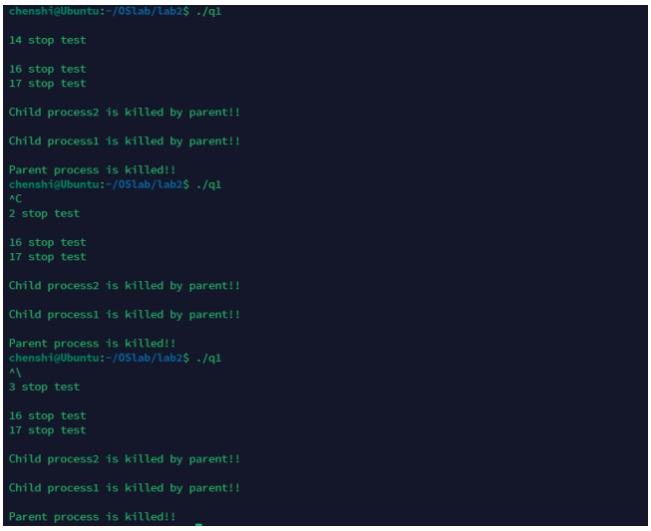


图 14: 5s 内按下 Ctrl+\ 或 Ctrl+Delete 发送中断

产生原因，设置了闹钟中断，5s 后会产生闹钟中断。

3. 改为闹钟中断后，程序运行的结果是什么样子？与之前有什么不同？

   没有不同，因为 5s 后会产生闹钟中断，所以不会有不同。闹钟中断和软中断的执行函数一样。

4. kill 命令在程序中使用了几次？每次的作用是什么？执行后的现象是什么？

   2 次，分别 kill 两个子进程。执行后，子进程终止。

5. 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？

   使用 exit() 函数可以在进程的内部杀死进程。使用 exit() 函数更好，因为可以在进程内部进行一些操作。

2.2 进程的管道通信

2.2.1 实验目的

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。

2.2.2 实验内容

1. 学习 man 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读参考资料。

2. 根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

2.2.3 实验思想

父进程创建俩个子进程，一个父子管道，其中两个进程各写入 2000 个字符，分有锁和无锁的情况。分别观察有锁和无锁情况下的写入情况。

2.2.4 实验步骤

1. 根据流程图和所给管道通信程序，补全程序，要求使用互斥锁实现管道通信的同步与互斥。

2. 删除互斥锁，运行程序，观察运行结果。

2.2.5 测试数据设计

进程 1 向管道中写入 2000 个字符 1，进程 2 向管道中写入 2000 个字符 2。

2.2.6 程序运行初值及运行结果分析

1. 有锁的情况下，在这次测试中，进程 1 先获取到管道的写入权，将管道上锁连续写入了 2000 个字符 1，写完后解锁管道，然后进程 2 获取到管道的写入权，将管道上锁连续写入了 2000 个字符 2，

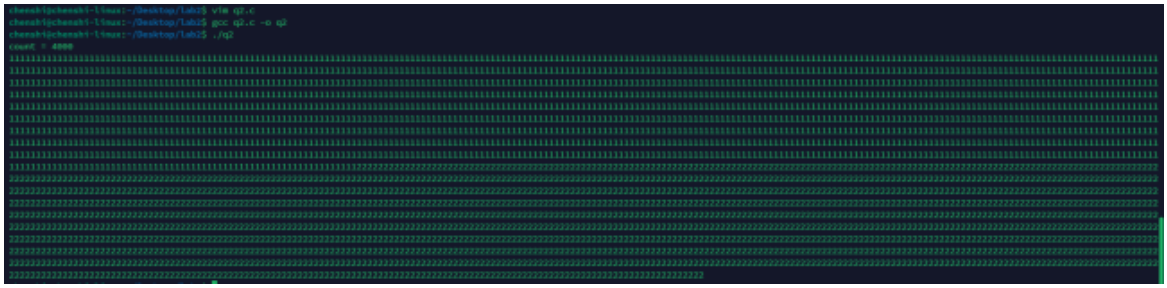


图 15: 有锁的情况下

2. 无锁的情况下，进程 1 和进程 2 争夺管道写入，各写入 2000 个字符后，进程 1 和进程 2 结束。所以在输出中

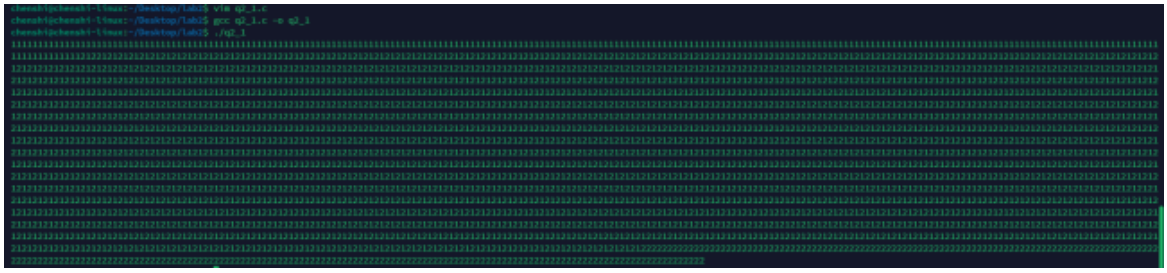


图 16: 无锁的情况下

2.2.7 问题回答

1. 你最初认为运行结果会怎么样？
   有锁的情况下，输出 4000 个字符，前 2000 个为同一个字符，后 2000 个为另一个字符，具体哪个字符在前取决于哪个进程先获取到管道的写入权。
   无锁的情况下，4000 个字符中字符 1 和字符 2 交替混杂出现。

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。
   实际情况与预测吻合，有锁的情况下，一个进程写入完毕后，另一个进程才能写入。无锁的情况下，两个进程同时写入.

3. 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？
   通过对写入段上锁，不上锁输出不可预测

2.3 内存的分配与回收

2.3.1 实验目的

通过设计实现内存分配管理的三种算法（FF，BF，WF），理解内存分配及回收的过程及实现思路，理解如何提高内存的分配效率和利用率。

2.3.2 实验内容

1. 理解内存分配 FF，BF，WF 策略及实现的思路。
2. 参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。每种算法要实现内存分配、回收、空闲块排序以及合并、紧缩等功能。

2.3.3 实验思想

1. 首次适应 (FF)：从内存的起始地址开始查找，找到第一个满足大小要求的空闲分区进行分配。
2. 最佳适应 (BF)：从整个空闲分区链表中找到一个最小的满足大小要求的空闲分区进行分配。
3. 最坏适应 (WF)：从整个空闲分区链表中找到一个最大的满足大小要求的空闲分区进行分配。

2.3.4 实验步骤

1. 根据实验内容要求，编写模拟内存分配管理程序代码 memoryċ
2. 修改程序，实现内存分配管理的三种算法（FF，BF，WF）
3. 实现内存紧缩
4. 编译并运行程序，分析运行结果
5. 比较三种算法的优缺点

2.3.5 测试数据设计

1. 初始化内存空间，大小为 1024 字节
2. 分别按 FF，BF，WF 算法分配内存空间，大小为 100，200，300。
3. 释放内存空间，大小为 200。
4. 分配内存空间位 600，测试内存紧缩

2.3.6 程序运行初值及运行结果分析

1. 设置内存空间为 1024，设置内存分配为 Best Fit 算法

图 17: 设置内存空间为 1024，设置内存分配为 Best Fit 算法

2. 设置了五个进程，每个进程分配空间为 64

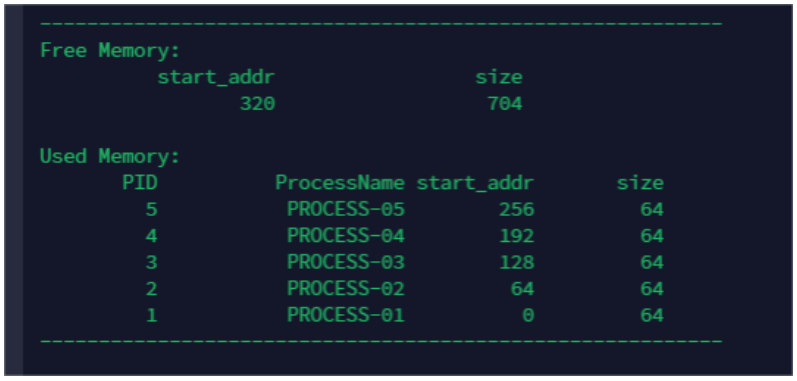


图 18: 设置了五个进程，每个进程分配空间为 64

3. 删除进程 2 和进程 3，展示全部进程的内存分配情况，验证了内存回收的正确性

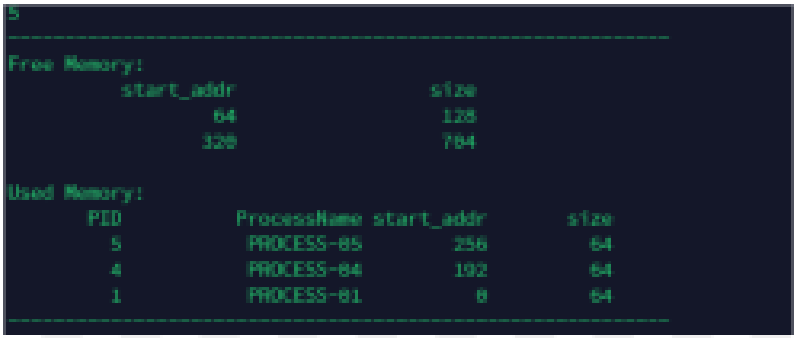


图 19: 删除进程 2 和进程 3，展示全部进程的内存分配情况

4. 采用 worst fit 算法，分配一块空间为 64 的内存，如果实现正确，起始地址应该为 320，经验证，实现正确

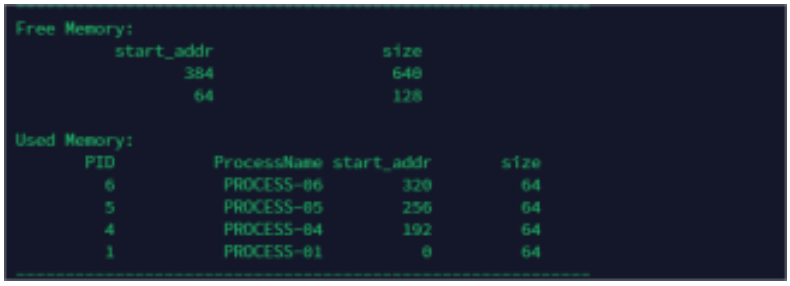


图 20: 采用 worst fit 算法，分配一块空间为 64 的内存

5. 采用 Best Fit 算法，分配一块空间为 64 的内存，如果实现正确，起始地址应该为 64，经验证，实现正确

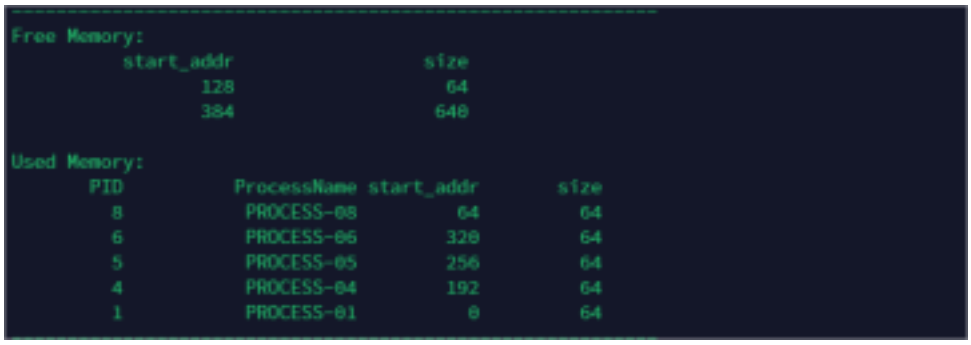


图 21: 采用 Best Fit 算法，分配一块空间为 64 的内存

6. 采用 First Fit 算法，分配一块空间为 64 的内存，如果实现正确，起始地址应该为 128，经验证，实现正确

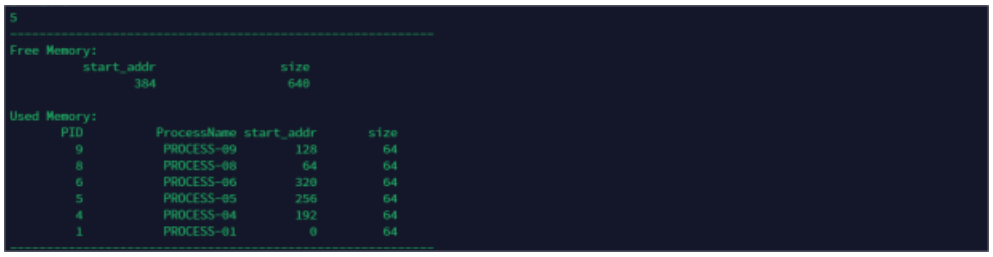


图 22: 采用 First Fit 算法，分配一块空间为 64 的内存

7. 验证内存紧缩的正确性，设置内存为 1000，分配五个进程各占用 200，释放进程 1，3

```
5
------------------------------------------------------
Free Memory:
         start_addr               size
                  0                200
                400                200

Used Memory:
       PID         ProcessName start_addr       size
         5          PROCESS-05        800        200
         4          PROCESS-04        600        200
         2          PROCESS-02        200        200
------------------------------------------------------
```

图 23: 验证内存紧缩的正确性

8. 再新建进程 6，分配 300 内存，会触发内存紧缩，释放所有空闲块，重新分配内存

```
------------------------------------------------------------
Free Memory:
          start_addr               size
                 900                100

Used Memory:
       PID         ProcessName start_addr       size
         6          PROCESS-06        600        300
         2          PROCESS-02          0        200
         4          PROCESS-04        200        200
         5          PROCESS-05        400        200
------------------------------------------------------------
```

图 24: 再新建进程 6，分配 300 内存，会触发内存紧缩

2.3.7 问题回答

1. 对涉及的 3 个算法进行比较，包括算法思想、算法的优缺点、在实现上如何提高算法的查找性能。

   (a) 算法思想

      i. First Fit 算法：从内存的起始地址开始查找，找到第一个满足大小要求的空闲分区进行分配。

      ii. Best Fit 算法：从整个空闲分区链表中找到一个最小的满足大小要求的空闲分区进行分配。

      iii. Worst Fit 算法：从整个空闲分区链表中找到一个最大的满足大小要求的空闲分区进行分配。

(b) 算法的优缺点

i. First Fit 算法：分配速度快，但是会产生大量的内存碎片。

ii. Best Fit 算法：分配速度慢，但是会产生最小的内存碎片。

iii. Worst Fit 算法：分配速度慢，但是会产生最大的内存碎片。

(c) 在实现上如何提高算法的查找性能

i. First Fit 算法: 将空闲块按照地址排序，每次分配时从头开始查找，找到第一个满足大小要求的空闲分区进行分配。

ii. Best Fit 算法: 将空闲块按照从小到大排序，每次分配时从头开始查找，找到第一个满足大小要求的空闲分区进行分配。

iii. Worst Fit 算法: 将空闲块按照从大到小排序，每次分配时从头开始查找，找到第一个满足大小要求的空闲分区进行分配。

2. 3 种算法的空闲块排序分别是如何实现的。

对于三种算法，都是将空闲块按照某种顺序排序。所以可全部采用同一种排序算法，我使用的是插入排序算法。对于三个算法，分别按照地址，块大小，块大小排序。

3. 结合实验，举例说明什么是内碎片、外碎片，紧缩功能解决的是什么碎片。

(a) 内碎片：分配给进程的内存块比进程所需的内存块大，但是进程只用所需的内存块，多余的内存块不能被其他进程使用，这部分内存块就是内碎片。

(b) 外碎片：由于内存块之间的空闲块太小，不能被其他进程使用，这部分内存块就是外碎片。

(c) 紧缩功能解决的是外碎片。

4. 在回收内存时，空闲块合并是如何实现的？

在回收内存时，会将回收的内存块与空闲块链表中的空闲块进行合并，合并的方法是将空闲块链表中的空闲块按照地址排序，然后遍历空闲块链表，如果相邻的两个空闲块地址连续，就将两个空闲块合并。然后按照原本的排序方式进行排序。

2.4 实验总结

2.4.1 实验中的问题与解决过程

1. 不了解键盘信号的相关操作，一个信号会发给所有的进程。

2. 对链表的操作不熟悉，在实现内存紧缩的时候，对链表的操作出现了问题。解决方法：将所有的已占用块移到内存的起始位置，然后将所有的空闲块移到已占用块的后面。

2.4.2 实验收获

1. 对进程的创建和进程间的通信有了更深的理解。
2. 对内存的分配和回收有了更深的理解。
3. 对链表的操作有了更深的理解。
4. 对信号的相关操作有了更深的理解。

2.4.3 意见与建议

优化内存的分配与回收部分提供的代码片段，首先调整一下代码的格式，将函数定义和实现放在了主函数前，同时发现了两个应当补全的函数 find_pid 和 do_exit，但是在代码中没有找到这两个函数的实现，所以在实验中自己实现了这两个函数。

2.5 附件

2.5.1 附件 1 实验 2.1 代码

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

int flag = 0;
pid_t pid1 = -1, pid2 = -1;

void inter_handler(int signal) {
        flag++;
        printf("\n%d stop test\n\n",signal);
        if (flag==1&& printf("16 stop test\n")&&
                    printf("17 stop test\n"))
        {
            kill(pid1, 16);
            kill(pid2, 17);
        }
    }
```

```
20
21 void vv(){};
22
23 void waiting() {
24     alarm(5);
25 }
26
27 int main() {
28     while (pid1 == -1) pid1 = fork();
29     if (pid1 > 0) {
30         while (pid2 == -1) pid2 = fork();
31         if (pid2 > 0) {
32             signal(SIGINT, inter_handler);
33             signal(SIGQUIT, inter_handler);
34             signal(SIGALRM, inter_handler);
35             waiting();
36             wait(NULL);
37             wait(NULL);
38             printf("\nParent process is killed!!\n");
39         } else {
40             signal(SIGQUIT, SIG_IGN);
41             signal(SIGINT, SIG_IGN);
42             signal(SIGALRM, SIG_IGN);
43             signal(17,vv);
44             pause();
45             printf("\nChild process2 is killed by parent!!\n");
46             return 0;
47         }
48     } else {
49         signal(SIGQUIT, SIG_IGN);
50         signal(SIGINT, SIG_IGN);
51         signal(SIGALRM, SIG_IGN);
52         signal(16,vv);
53         pause();
```

```
54          printf("\nChild process1 is killed by parent!!\n");
55          return 0;
56      }
57      return 0;
58  }
```

2.5.2 附件 2 实验 2.2 代码

```
1  #include <unistd.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/wait.h>
6
7  int pid1, pid2;
8  int main()
9  {
10     int fd[2];
11     char InPipe[8000];
12     char c1 = '1', c2 = '2';
13     pipe(fd);
14
15     while ((pid1 = fork()) == -1);
16
17     if (pid1 == 0)
18     {
19         close(fd[0]); // 关闭读管道，子进程 1 不需要读
20         lockf(fd[1], 1, 0);
21         for (int i = 0; i < 2000; i++) {
22             write(fd[1], &c1, 1); // 向管道写入字符 '1'
23         }
24         sleep(5);
25         lockf(fd[1], 0, 0);
26         exit(0);
27     }
```

```
28      else
29      {
30          while ((pid2 = fork()) == -1);
31
32          if (pid2 == 0)
33          {
34              close(fd[0]); // 关闭读管道，子进程 2 不需要读
35              lockf(fd[1], 1, 0);
36              for (int i = 0; i < 2000; i++) {
37                  write(fd[1], &c2, 1); // 向管道写入字符 '2'
38              }
39              sleep(5);
40              lockf(fd[1], 0, 0);
41              exit(0);
42          }
43          else
44          {
45              close(fd[1]); // 关闭写管道，父进程不需要写
46
47              waitpid(pid1, NULL, 0); // 等待子进程 1 结束
48              waitpid(pid2, NULL, 0); // 等待子进程 2 结束
49
50              int count = 0;
51              while (read(fd[0], &InPipe[count], 1) > 0) {
52                  count++;
53              }
54              InPipe[count] = '\0'; // 加字符串结束符
55              printf("count␣=␣%d\n", count);
56              printf("%s\n", InPipe);
57              exit(0);
58          }
59      }
60  }
```

2.5.3 附件 3 实验 2.3 代码

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PROCESS_NAME_LEN 32    /*进程名长度 */
#define MIN_SLICE 10           /* 最小碎片的大小 */
#define DEFAULT_MEM_SIZE 1024 /* 内存大小 */
#define DEFAULT_MEM_START 0    /* 起始位置 */
/* 内存分配算法 */
#define MA_FF 1
#define MA_BF 2
#define MA_WF 3
int mem_size = DEFAULT_MEM_SIZE; /* 内存大小 */
int ma_algorithm = MA_FF;        /* 当前分配算法 */
static int pid = 0;              /* 初始 pid */
int flag = 0;                    /* 设置内存大小标志 */
/* (1) 主要功能 */
/*
1 - Set memory size (default = 1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
*/

/*(2) 主要数据结构 */
/* 1） 内存空闲分区的描述 */
/* 描述每一个空闲块的数据结构*/
struct free_block_type
{
    int size;
    int start_addr;
    struct free_block_type *next;
```

```
34  };
35  /* 指向内存中空闲块链表的首指针 */
36  struct free_block_type*free_block;
37
38  /*2) 描述已分配的内存块 */
39  /* 每个进程分配到的内存块的描述*/
40  struct allocated_block
41  {
42      int pid;
43      int size;
44      int start_addr;
45      char process_name[PROCESS_NAME_LEN];
46      struct allocated_block *next;
47  };
48  /* 进程分配内存块链表的首指针 */
49  struct allocated_block*allocated_block_head = NULL;
50
51  /*初始化空闲块，默认为一块，可以指定大小及起始地址*/
52  struct free_block_type *init_free_block(int mem_size)
53  {
54      struct free_block_type*fb;
55      fb = (struct free_block_type *)malloc
56                          (sizeof(struct free_block_type));
57      if (fb == NULL)
58      {
59          printf("No mem\n");
60          return NULL;
61      }
62      fb->size = mem_size;
63      fb->start_addr = DEFAULT_MEM_START;
64      fb->next = NULL;
65      return fb;
66  }
67
```

```
68  /*显示菜单*/
69  void display_menu()
70  {
71      printf("\n");
72      printf("1␣-␣Set␣memory␣size␣(default=%d)\n",
73                                  DEFAULT_MEM_SIZE);
74      printf("2␣-␣Select␣memory␣allocation␣algorithm\n");
75      printf("3␣-␣New␣process\n");
76      printf("4␣-␣Terminate␣a␣process\n");
77      printf("5␣-␣Display␣memory␣usage\n");
78      printf("0␣-␣Exit\n");
79  }
80
81  /*设置内存的大小*/
82  int set_mem_size()
83  {
84      int size;
85      if (flag != 0)
86      { // 防止重复设置
87          printf("Cannot␣set␣memory␣size␣again\n");
88          return 0;
89      }
90      printf("Total␣memory␣size␣=");
91      scanf("%d", &size);
92      char input;
93      while ((input = getchar()) != '\n' && input != EOF)
94      {
95          // 读入一个字符后清空缓存区，避免\n被读入
96          //或者一次输入多个字符
97      }
98      if (size > 0)
99      {
100         mem_size = size;
101         free_block->size = mem_size;
```

```
102     }
103     flag = 1;
104     return 1;
105 }
106
107 // 所有的排序算法都采用插入排序，只需要修改比较的条件即可
108
109 /*按 FF 算法重新整理内存空闲块链表*/
110 void rearrange_FF()
111 {
112     if (free_block == NULL)
113         return;
114     struct free_block_type *current,*next, *temp;
115     current = free_block;
116     struct free_block_type*new_head = (struct
117     free_block_type *)malloc(sizeof(struct free_block_type));
118     // 头结点dummy
119     new_head->next = NULL;
120     new_head->size = 0;
121     new_head->start_addr = -1;
122     while (current != NULL)
123     {
124         next = current->next;
125         temp = new_head;
126         while (temp->next != NULL)
127         {
128             if (temp->next->start_addr > current->start_addr)
129             {
130                 break;
131             }
132             temp = temp->next;
133         }
134         current->next = temp->next;
135         temp->next = current;
```

```
136             current = next;
137         }
138         free_block = new_head->next;
139         free(new_head);
140     }
141
142     /*按 BF 算法重新整理内存空闲块链表*/
143     void rearrange_BF()
144     {
145         if (free_block == NULL)
146             return;
147         struct free_block_type *current,*next, *temp;
148         current = free_block;
149         struct free_block_type*new_head =
150         (struct free_block_type *)malloc
151         (sizeof(struct free_block_type));
152         new_head->next = NULL;
153         new_head->size = 0;
154         new_head->start_addr = -1;
155         while (current != NULL)
156         {
157             next = current->next;
158             temp = new_head;
159             while (temp->next != NULL)
160             {
161                 if (temp->next->size > current->size)
162                 {
163                     break;
164                 }
165                 temp = temp->next;
166             }
167             current->next = temp->next;
168             temp->next = current;
169             current = next;
```

```
170         }
171         free_block = new_head->next;
172         free(new_head);
173     }
174
175     /*按 WF 算法重新整理内存空闲块链表*/
176     void rearrange_WF()
177     {
178         if (free_block == NULL)
179             return;
180         struct free_block_type *current,*next, *temp;
181         current = free_block;
182         struct free_block_type*new_head =
183         (struct free_block_type *)malloc
184         (sizeof(struct free_block_type));
185         new_head->next = NULL;
186         new_head->size = 10000000;
187         new_head->start_addr = -1;
188         while (current != NULL)
189         {
190             next = current->next;
191             temp = new_head;
192             while (temp->next != NULL)
193             {
194                 if (temp->next->size < current->size)
195                 {
196                     break;
197                 }
198                 temp = temp->next;
199             }
200             current->next = temp->next;
201             temp->next = current;
202             current = next;
203         }
```

```
204         free_block = new_head->next;
205         free(new_head);
206     }
207
208     /*按指定算法重新整理内存空闲块链表*/
209     void rearrange(int algorithm)
210     {
211         switch (algorithm)
212         {
213         case MA_FF:
214             rearrange_FF();
215             break;
216         case MA_BF:
217             rearrange_BF();
218             break;
219         case MA_WF:
220             rearrange_WF();
221             break;
222         }
223     }
224
225     /*设置当前的分配算法 */
226     void set_algorithm()
227     {
228         int algorithm;
229         printf("\t1␣-␣First␣Fit\n");
230         printf("\t2␣-␣Best␣Fit\n");
231         printf("\t3␣-␣Worst␣Fit\n");
232         scanf("%d", &algorithm);
233         char input;
234         while ((input = getchar()) != '\n' && input != EOF)
235         {
236             // 读入一个字符后清空缓存区，避免\n被读入
237             //或者一次输入多个字符
```

```
238         }
239         if (algorithm >= 1 && algorithm <= 3)
240             ma_algorithm = algorithm;
241         /* 按指定算法重新排列空闲区链表*/
242         rearrange(ma_algorithm);
243     }
244
245     //已分配内存空间按照起始地址排序
246     void sort_allocated_block(){
247         struct allocated_block *current,*next, *temp;
248         current = allocated_block_head;
249         struct allocated_block*new_head =
250         (struct allocated_block *)malloc
251         (sizeof(struct allocated_block));
252         new_head->next = NULL;
253         new_head->size = 0;
254         new_head->start_addr = -1;
255         while (current != NULL)
256         {
257             next = current->next;
258             temp = new_head;
259             while (temp->next != NULL)
260             {
261                 if (temp->next->start_addr > current->start_addr)
262                 {
263                     break;
264                 }
265                 temp = temp->next;
266             }
267             current->next = temp->next;
268             temp->next = current;
269             current = next;
270         }
271         allocated_block_head = new_head->next;
```

```
272     free(new_head);
273 }
274
275 /*分配内存模块*/
276 int  allocate_mem(struct allocated_block *ab)
277 {
278     struct free_block_type*fbt, *pre;
279     int request_size = ab->size;
280     fbt = pre = free_block;
281     if (free_block == NULL)
282     {
283         return -1;
284     }
285
286     // 根据当前算法在空闲分区链表中搜索合适的空闲分区进行分配
287     // 分配时需要考虑多种情况，如分割、合并、内存紧缩等
288     // 1. 找到可满足空闲分区且分配后剩余空间足够大，则分割
289     // 2. 找到可满足空闲分区且但分配后剩余空间比较小，则一起分配
290     // 3. 找不可满足需要的空闲分区但空闲分区之和能满足需要，
291     //则采用内存紧缩技术，
292     // 进行空闲分区的合并，然后再分配
293     // 4. 在成功分配内存后，应保持空闲分区按照相应算法有序
294     // 5. 分配成功则返回 1，否则返回 -1
295     // 请自行补充实现...
296
297     // 找到第一个满足要求的空闲分区
298     while (fbt != NULL)
299     {
300         if (fbt->size >= request_size)
301         {
302             break;
303         }
304         pre = fbt;
305         fbt = fbt->next;
```

```
306         }
307         // 没找到，则合并空闲分区后再找
308         if (fbt == NULL)
309         {
310             int rest = 0;
311             fbt = free_block;
312             while (fbt != NULL) {
313                 rest += fbt->size;
314                 fbt = fbt->next;
315 
316             }
317             if(rest < request_size) {
318                 return -1;
319             }else {
320                 sort_allocated_block();
321                 struct allocated_block* abt=allocated_block_head;
322                 int prev=0;
323                 while(abt!=NULL){
324                     abt->start_addr=prev;
325                     prev=prev+abt->size;
326                     abt=abt->next;
327                 }
328                 //释放所有空闲分区
329                 fbt = free_block->next;
330                 while (fbt != NULL)
331                 {
332                     pre = fbt;
333                     fbt = fbt->next;
334                     free(pre);
335                 }
336                 free_block->size = mem_size-prev;
337                 free_block->start_addr = prev;
338                 free_block->next = NULL;
339 
```

```
340             }
341             fbt = free_block;
342         }
343         // 找到了，分配
344         if (fbt->size - request_size > MIN_SLICE)
345         {
346             ab->start_addr = fbt->start_addr;
347             ab->size = request_size;
348             fbt->start_addr += request_size;
349             fbt->size -= request_size;
350         }
351         else
352         {
353             ab->start_addr = fbt->start_addr;
354             ab->size = fbt->size;
355             if (fbt == free_block)
356             {
357                 free_block = fbt->next;
358             }
359             else
360             {
361                 pre->next = fbt->next;
362             }
363             free(fbt);
364         }
365         return 1;
366 }
367
368 /*创建新的进程，主要是获取内存的申请数量*/
369 int new_process()
370 {
371     struct allocated_block *ab;
372     int size;
373     int ret;
```

```
374     ab = (struct allocated_block *)
375     malloc(sizeof(struct allocated_block));
376     if (!ab)
377         exit(-5);
378     ab->next = NULL;
379     pid++;
380     sprintf(ab->process_name, "PROCESS-%02d", pid);
381     ab->pid = pid;
382     printf("Memory for %s:", ab->process_name);
383     scanf("%d", &size);
384     char input;
385     while ((input = getchar()) != '\n' && input != EOF)
386     {
387         // 读入一个字符后清空缓存区，避免\n被读入
388         //或者一次输入多个字符
389     }
390     if (size > 0)
391         ab->size = size;
392     ret = allocate_mem(ab); /* 从空闲区分配内存，
393      ret==1 表示分配 ok*/
394
395     /* 如果此时 allocated_block_head 尚未赋值，则赋值 */
396     if ((ret == 1) && (allocated_block_head == NULL))
397     {
398         allocated_block_head = ab;
399         return 1;
400     }
401     /* 分配成功，将该已分配块的描述插入已分配链表 */
402     else if (ret == 1)
403     {
404         ab->next = allocated_block_head;
405         allocated_block_head = ab;
406         return 2;
407     }
```

```
408     else if (ret == -1)
409     {
410         /* 分配不成功 */
411         printf("Allocation fail\n");
412         free(ab);
413         return -1;
414     }
415     return 3;
416 }
417
418 /*根据pid查找内存块*/
419 struct allocated_block *find_process(int pid)
420 {
421     struct allocated_block*ab = allocated_block_head;
422     while (ab != NULL)
423     {
424         if (ab->pid == pid)
425             break;
426         ab = ab->next;
427     }
428     return ab;
429 }
430
431 /*将 ab 所表示的已分配区归还，并进行可能的合并*/
432 int free_mem(struct allocated_block *ab)
433 {
434     int algorithm = ma_algorithm;
435     struct free_block_type*fbt, *pre,*work;
436
437     // 进行可能的合并，基本策略如下
438     // 1. 将新释放的结点插入到空闲分区队列末尾
439     // 2. 对空闲链表按照地址有序排列
440     // 3. 检查并合并相邻的空闲分区
441     // 4. 将空闲链表重新按照当前算法排序
```

```
442         // 请自行补充...
443
444         //  头插法
445         work = (struct free_block_type *)
446         malloc(sizeof(struct free_block_type));
447         if (work == NULL)
448         {
449             return -1;
450         }
451
452         work->start_addr = ab->start_addr;
453         work->size = ab->size;
454         work->next = free_block;
455         free_block = work;
456
457         rearrange_FF(free_block);
458         fbt = free_block;
459         while (fbt->next != NULL)
460         {
461             if (fbt->start_addr + fbt->size == fbt->next->start_addr)
462             {
463                 struct free_block_type *temp = fbt->next;
464                 fbt->size += fbt->next->size;
465                 fbt->next = fbt->next->next;
466                 free(temp);
467             }
468             else
469             {
470                 fbt = fbt->next;
471             }
472         }
473         rearrange(ma_algorithm);
474
475         return 1;
```

```
476 }
477
478 /*释放 ab 数据结构节点*/
479 int dispose(struct allocated_block *free_ab)
480 {
481     struct allocated_block*pre, *ab;
482     if (free_ab == allocated_block_head)
483     { /* 如果要释放第一个节点 */
484         allocated_block_head = allocated_block_head->next;
485         free(free_ab);
486         return 1;
487     }
488     pre = allocated_block_head;
489     ab = allocated_block_head->next;
490     while (ab != free_ab)
491     {
492         pre = ab;
493         ab = ab->next;
494     }
495     pre->next = ab->next;
496     free(ab);
497     return 2;
498 }
499
500 /*删除进程, 归还分配的存储空间, 并删除描述该进程内存分配的节点*/
501 void kill_process()
502 {
503     struct allocated_block *ab;
504     int pid;
505     printf("Kill␣Process,␣pid=");
506     scanf("%d", &pid);
507     char input;
508     while ((input = getchar()) != '\n' && input != EOF)
509     {
```

```
510             // 读入一个字符后清空缓存区，避免\n被读入
511             //或者一次输入多个字符
512         }
513         ab = find_process(pid);
514         if (ab != NULL)
515         {
516             free_mem(ab); /* 释放 ab 所表示的分配区 */
517             dispose(ab);  /* 释放 ab 数据结构节点 */
518         }
519     }
520
521     /*显示当前内存的使用情况，包括空闲区的情况和已经分配的情况*/
522     int display_mem_usage()
523     {
524         struct free_block_type *fbt = free_block;
525         struct allocated_block*ab = allocated_block_head;
526
527         printf("------------------------------
528     ␣␣␣␣------------------------------\n");
529         /* 显示空闲区 */
530         printf("Free␣Memory:\n");
531         printf("%20s␣%20s\n", "start_addr", "size");
532         while (fbt != NULL)
533         {
534             printf("%20d␣%20d\n", fbt->start_addr, fbt->size);
535             fbt = fbt->next;
536         }
537
538         /* 显示已分配区 */
539         printf("\nUsed␣Memory:\n");
540         printf("%10s␣%20s␣%10s␣%10s\n", "PID",
541         "ProcessName", "start_addr", "size");
542         while (ab != NULL)
543         {
```

```
544             printf("%10d␣%20s␣%10d␣%10d\n",
545             ab->pid, ab->process_name, ab->start_addr, ab->size);
546             ab = ab->next;
547         }
548         printf("-----------------------------------
549 ␣␣␣␣-----------------------------------\n");
550 }
551
552 /*退出,释放所有链表*/
553
554 void do_exit()
555 {
556     struct free_block_type *fbt = free_block;
557     struct allocated_block*ab = allocated_block_head;
558     while (fbt != NULL)
559     {
560         free_block = fbt->next;
561         free(fbt);
562         fbt = free_block;
563     }
564     while (ab != NULL)
565     {
566         allocated_block_head = ab->next;
567         free(ab);
568         ab = allocated_block_head;
569     }
570 }
571
572 void tab(char choice)
573 {
574 }
575
576 int main()
577 {
```

```
578     char choice, input;
579     pid = 0;
580     free_block = init_free_block(mem_size); // 初始化空闲区
581     while (1)
582     {
583         display_menu(); // 显示菜单
584         choice = getchar();
585         while ((input = getchar()) != '\n' && input != EOF)
586         {
587             // 读入一个字符后清空缓存区，避免\n被读入
588             //或者一次输入多个字符
589         }
590         switch (choice)
591         {
592         case '1':
593             set_mem_size();
594             break; // 设置内存大小
595         case '2':
596             set_algorithm();
597             flag = 1;
598             break; // 设置算法
599         case '3':
600             new_process();
601             flag = 1;
602             break; // 创建新进程
603         case '4':
604             kill_process();
605             flag = 1;
606             break; // 删除进程
607         case '5':
608             display_mem_usage();
609             flag = 1;
610             break; // 显示内存使用
611         case '0':
```

```
612                 do_exit();
613                 exit(0);
614                 /*释放链表并退出*/
615             default:
616                 break;
617             }
618         }
619         return 0;
620     }
```

2.5.4 附件 4 README

实验二 进程通信与内存管理

2.1 进程的软中断通信

1. **实验代码**

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

int flag = 0;
pid_t pid1 = -1, pid2 = -1;

void inter_handler(int signal) {
        flag++;
        printf("\n%d stop test\n\n",signal);
        if (flag==1&& printf("16 stop test\n")&& printf("17 stop test\n"))
        {
            kill(pid1, 16);
            kill(pid2, 17);
        }
    }

void vv(){};

void waiting() {
    alarm(5);
}

int main() {
    while (pid1 == -1) pid1 = fork();
    if (pid1 > 0) {
        while (pid2 == -1) pid2 = fork();
        if (pid2 > 0) {
            signal(SIGINT, inter_handler);
            signal(SIGQUIT, inter_handler);
            signal(SIGALRM, inter_handler);
            waiting();
            wait(NULL);
            wait(NULL);
            printf("\nParent process is killed!!\n");
```

```
        } else {
            signal(SIGQUIT, SIG_IGN);
            signal(SIGINT, SIG_IGN);
            signal(SIGALRM, SIG_IGN);
            signal(17,vv);
            pause();
            printf("\nChild process2 is killed by parent!!\n");
            return 0;
        }
    } else {
        signal(SIGQUIT, SIG_IGN);
        signal(SIGINT, SIG_IGN);
        signal(SIGALRM, SIG_IGN);
        signal(16,vv);
        pause();
        printf("\nChild process1 is killed by parent!!\n");
        return 0;
    }
    return 0;
}
```

代码解析：

1. 函数 `inter_handler`：当接收到 `SIGINT` 或 `SIGQUIT` 或 `SIGALRM` 信号时，分别向子进程1和子进程2发送16和17信号。flag用于判断是否是第一次接收到信号，如果是则向子进程发送信号，否则不发送信号。
2. 函数 `vv`：空函数，用于处理子进程接收到的信号。
3. 函数 `waiting`：设置5秒的定时器。5秒后向父进程发送 `SIGALRM` 信号。
4. 主函数：创建两个子进程，父进程调用 `waiting` 函数设置定时器。为父进程接收到的信号设置处理函数 `inter_handler`。为子进程接收到的信号设置处理函数 `vv`。父进程等待两个子进程结束后，输出提示信息。子进程在没接到信号前通过 `pause` 函数挂起，接到信号后输出提示信息。

2. **实验结果**

```
chenshi@Ubuntu:~/OSlab/lab2$ ./q1

14 stop test

16 stop test
17 stop test

Child process2 is killed by parent!!

Child process1 is killed by parent!!

Parent process is killed!!
chenshi@Ubuntu:~/OSlab/lab2$ ./q1
^C
2 stop test

16 stop test
17 stop test

Child process2 is killed by parent!!

Child process1 is killed by parent!!

Parent process is killed!!
chenshi@Ubuntu:~/OSlab/lab2$ ./q1
^\
3 stop test

16 stop test
17 stop test

Child process2 is killed by parent!!

Child process1 is killed by parent!!

Parent process is killed!!
```

分别验证了等待5秒和输入 `Ctrl+C` 和 `Ctrl+\` 的情况，都实现了预期的效果。

2.2 进程的管道通信

有锁情况

1. **实验代码**

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

int pid1, pid2;
int main()
{
    int fd[2];
    char InPipe[8000];
```

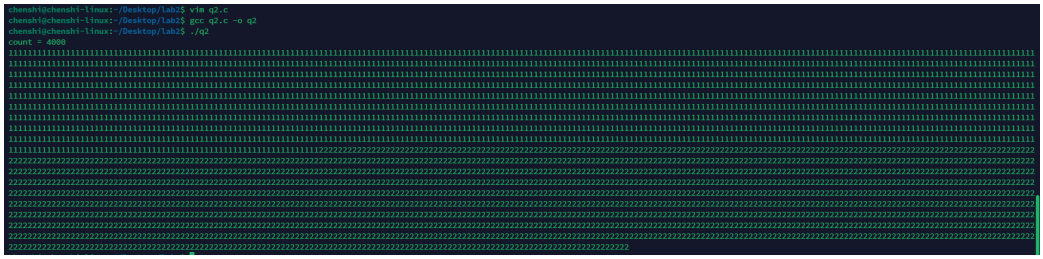
```
    char c1 = '1', c2 = '2';
    pipe(fd);

    while ((pid1 = fork()) == -1);

    if (pid1 == 0)
    {
        close(fd[0]); // 关闭读管道，子进程 1 不需要读
        lockf(fd[1], 1, 0);
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1); // 向管道写入字符 '1'
        }
        sleep(5);
        lockf(fd[1], 0, 0);
        exit(0);
    }
    else
    {
        while ((pid2 = fork()) == -1);

        if (pid2 == 0)
        {
            close(fd[0]); // 关闭读管道，子进程 2 不需要读
            lockf(fd[1], 1, 0);
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1); // 向管道写入字符 '2'
            }
            sleep(5);
            lockf(fd[1], 0, 0);
            exit(0);
        }
        else
        {
            close(fd[1]); // 关闭写管道，父进程不需要写

            waitpid(pid1, NULL, 0); // 等待子进程 1 结束
            waitpid(pid2, NULL, 0); // 等待子进程 2 结束

            int count = 0;
            while (read(fd[0], &InPipe[count], 1) > 0) {
                count++;
            }
            InPipe[count] = '\0'; // 加字符串结束符
            printf("count = %d\n", count);
            printf("%s\n", InPipe);
            exit(0);
```

```
        }
    }
}
```

代码解析：

1. 在main函数开始创建了一个管道，用于父子进程之间的通信。
2. 创建两个子进程，子进程1向管道中写入2000个字符'1'，子进程2向管道中写入2000个字符'2'。
3. 子进程在写入前会使用 `lockf(fd[1], 1, 0);` 函数对管道进行加锁，写入后通过 `lockf(fd[1], 0, 0)` 解锁。这样可以保证两个子进程写入的数据不会交叉。
4. 父进程通过 `waitpid` 函数等待两个子进程结束后，从管道中读取数据并输出。

2. **实验结果**



可以看到输出的 `count` 为4000，说明两个子进程都向管道中写入了2000个字符。输出的字符串中，字符'1'和字符'2'是分开的，说明两个子进程写入的数据没有交叉。

无锁情况

1. **实验代码**

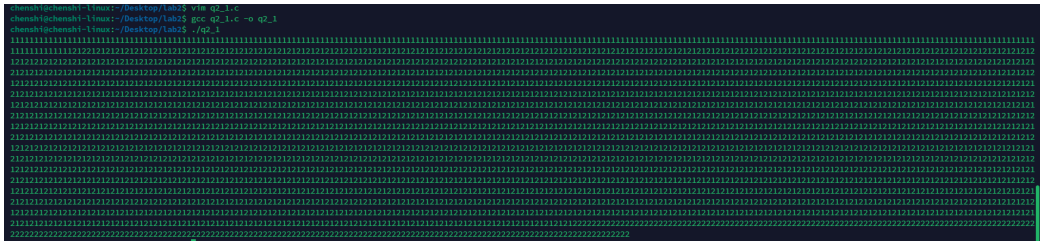
```c
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int pid1, pid2;

int main() {
    int fd[2];
    char InPipe[1000];
    char c1 = '1', c2 = '2';
    pipe(fd);
```

```
    while ((pid1 = fork()) == -1);

    if (pid1 == 0) {
        close(fd[0]);
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1);
        }
        sleep(5);
        exit(0);
    }
    else {
        while ((pid2 = fork()) == -1);
        if (pid2 == 0) {
            close(fd[0]);
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1);
            }
            sleep(5);
            exit(0);
        }
        else {
            close(fd[1]);
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
            int count = 0;
            while (read(fd[0], &InPipe[count], 1) > 0) {
                count++;
            }
            InPipe[count] = '\0';
            printf("%s\n", InPipe);
            exit(0);
        }
    }
}
```

代码解析：

1. 与有锁情况相比，只是去掉了加锁和解锁的代码。

2. **实验结果**



可以看到输出的 count 为4000，说明两个子进程都向管道中写入了2000个字符，没有字符写入被吞。输出的字符串中，字符'1'和字符'2'是交叉的，说明两个子进程写入的数据交叉了。

2.3 内存的分配与回收

1. **实验代码**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PROCESS_NAME_LEN 32   /*进程名长度 */
#define MIN_SLICE 10          /* 最小碎片的大小 */
#define DEFAULT_MEM_SIZE 1024 /* 内存大小 */
#define DEFAULT_MEM_START 0   /* 起始位置 */
/* 内存分配算法 */
#define MA_FF 1
#define MA_BF 2
#define MA_WF 3
int mem_size = DEFAULT_MEM_SIZE; /* 内存大小 */
int ma_algorithm = MA_FF;        /* 当前分配算法 */
static int pid = 0;              /* 初始 pid */
int flag = 0;                    /* 设置内存大小标志 */
/* (1) 主要功能 */
/*
1 - Set memory size (default = 1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
*/

/*(2) 主要数据结构 */
/* 1） 内存空闲分区的描述 */
/* 描述每一个空闲块的数据结构*/
struct free_block_type
{
```

```
    int size;
    int start_addr;
    struct free_block_type *next;
};
/* 指向内存中空闲块链表的首指针 */
struct free_block_type*free_block;

/*2） 描述已分配的内存块 */
/* 每个进程分配到的内存块的描述*/
struct allocated_block
{
    int pid;
    int size;
    int start_addr;
    char process_name[PROCESS_NAME_LEN];
    struct allocated_block *next;
};
/* 进程分配内存块链表的首指针 */
struct allocated_block*allocated_block_head = NULL;

/*初始化空闲块，默认为一块，可以指定大小及起始地址*/
struct free_block_type *init_free_block(int mem_size)
{
    struct free_block_type*fb;
    fb = (struct free_block_type *)malloc(sizeof(struct free_block_type));
    if (fb == NULL)
    {
        printf("No mem\n");
        return NULL;
    }
    fb->size = mem_size;
    fb->start_addr = DEFAULT_MEM_START;
    fb->next = NULL;
    return fb;
}

/*显示菜单*/
void display_menu()
{
    printf("\n");
    printf("1 - Set memory size (default=%d)\n", DEFAULT_MEM_SIZE);
    printf("2 - Select memory allocation algorithm\n");
    printf("3 - New process\n");
    printf("4 - Terminate a process\n");
    printf("5 - Display memory usage\n");
    printf("0 - Exit\n");
```

```
}

/*设置内存的大小*/
int set_mem_size()
{
    int size;
    if (flag != 0)
    { // 防止重复设置
        printf("Cannot set memory size again\n");
        return 0;
    }
    printf("Total memory size =");
    scanf("%d", &size);
    char input;
    while ((input = getchar()) != '\n' && input != EOF)
    {
        // 读入一个字符后清空缓存区，避免\n被读入或者一次输入多个字符
    }
    if (size > 0)
    {
        mem_size = size;
        free_block->size = mem_size;
    }
    flag = 1;
    return 1;
}

// 所有的排序算法都采用插入排序，只需要修改比较的条件即可

/*按 FF 算法重新整理内存空闲块链表*/
void rearrange_FF()
{
    if (free_block == NULL)
        return;
    struct free_block_type *current,*next, *temp;
    current = free_block;
    struct free_block_type*new_head = (struct free_block_type
*)malloc(sizeof(struct free_block_type)); // 头结点dummy
    new_head->next = NULL;
    new_head->size = 0;
    new_head->start_addr = -1;
    while (current != NULL)
    {
        next = current->next;
        temp = new_head;
        while (temp->next != NULL)
```

```
        {
            if (temp->next->start_addr > current->start_addr)
            {
                break;
            }
            temp = temp->next;
        }
        current->next = temp->next;
        temp->next = current;
        current = next;
    }
    free_block = new_head->next;
    free(new_head);
}

/*按 BF 算法重新整理内存空闲块链表*/
void rearrange_BF()
{
    if (free_block == NULL)
        return;
    struct free_block_type *current,*next, *temp;
    current = free_block;
    struct free_block_type*new_head = (struct free_block_type
*)malloc(sizeof(struct free_block_type));
    new_head->next = NULL;
    new_head->size = 0;
    new_head->start_addr = -1;
    while (current != NULL)
    {
        next = current->next;
        temp = new_head;
        while (temp->next != NULL)
        {
            if (temp->next->size > current->size)
            {
                break;
            }
            temp = temp->next;
        }
        current->next = temp->next;
        temp->next = current;
        current = next;
    }
    free_block = new_head->next;
    free(new_head);
}
```

```
/*按 WF 算法重新整理内存空闲块链表*/
void rearrange_WF()
{
    if (free_block == NULL)
        return;
    struct free_block_type *current,*next, *temp;
    current = free_block;
    struct free_block_type*new_head = (struct free_block_type
*)malloc(sizeof(struct free_block_type));
    new_head->next = NULL;
    new_head->size = 10000000;
    new_head->start_addr = -1;
    while (current != NULL)
    {
        next = current->next;
        temp = new_head;
        while (temp->next != NULL)
        {
            if (temp->next->size < current->size)
            {
                break;
            }
            temp = temp->next;
        }
        current->next = temp->next;
        temp->next = current;
        current = next;
    }
    free_block = new_head->next;
    free(new_head);
}

/*按指定算法重新整理内存空闲块链表*/
void rearrange(int algorithm)
{
    switch (algorithm)
    {
    case MA_FF:
        rearrange_FF();
        break;
    case MA_BF:
        rearrange_BF();
        break;
    case MA_WF:
        rearrange_WF();
```

```
        break;
    }
}

/*设置当前的分配算法 */
void set_algorithm()
{
    int algorithm;
    printf("\t1 - First Fit\n");
    printf("\t2 - Best Fit\n");
    printf("\t3 - Worst Fit\n");
    scanf("%d", &algorithm);
    char input;
    while ((input = getchar()) != '\n' && input != EOF)
    {
        // 读入一个字符后清空缓存区，避免\n被读入或者一次输入多个字符
    }
    if (algorithm >= 1 && algorithm <= 3)
        ma_algorithm = algorithm;
    /* 按指定算法重新排列空闲区链表*/
    rearrange(ma_algorithm);
}

//已分配内存空间按照起始地址排序
void sort_allocated_block(){
    struct allocated_block *current,*next, *temp;
    current = allocated_block_head;
    struct allocated_block*new_head = (struct allocated_block
*)malloc(sizeof(struct allocated_block));
    new_head->next = NULL;
    new_head->size = 0;
    new_head->start_addr = -1;
    while (current != NULL)
    {
        next = current->next;
        temp = new_head;
        while (temp->next != NULL)
        {
            if (temp->next->start_addr > current->start_addr)
            {
                break;
            }
            temp = temp->next;
        }
        current->next = temp->next;
        temp->next = current;
```

```
        current = next;
    }
    allocated_block_head = new_head->next;
    free(new_head);
}

/*分配内存模块*/
int  allocate_mem(struct allocated_block *ab)
{
    struct free_block_type*fbt, *pre;
    int request_size = ab->size;
    fbt = pre = free_block;
    if (free_block == NULL)
    {
        return -1;
    }

    // 根据当前算法在空闲分区链表中搜索合适的空闲分区进行分配
    // 分配时需要考虑多种情况，如分割、合并、内存紧缩等
    // 1. 找到可满足空闲分区且分配后剩余空间足够大，则分割
    // 2. 找到可满足空闲分区且但分配后剩余空间比较小，则一起分配
    // 3. 找不可满足需要的空闲分区但空闲分区之和能满足需要，则采用内存紧缩技术，
    // 进行空闲分区的合并，然后再分配
    // 4. 在成功分配内存后，应保持空闲分区按照相应算法有序
    // 5. 分配成功则返回 1，否则返回 -1
    // 请自行补充实现...

    // 找到第一个满足要求的空闲分区
    while (fbt != NULL)
    {
        if (fbt->size >= request_size)
        {
            break;
        }
        pre = fbt;
        fbt = fbt->next;
    }
    // 没找到，则合并空闲分区后再找
    if (fbt == NULL)
    {
        int rest = 0;
        fbt = free_block;
        while (fbt != NULL) {
            rest += fbt->size;
            fbt = fbt->next;
```

```
        }
        if(rest < request_size) {
            return -1;
        }else {
            sort_allocated_block();
            struct allocated_block* abt=allocated_block_head;
            int prev=0;
            while(abt!=NULL){
                abt->start_addr=prev;
                prev=prev+abt->size;
                abt=abt->next;
            }
            //释放所有空闲分区
            fbt = free_block->next;
            while (fbt != NULL)
            {
                pre = fbt;
                fbt = fbt->next;
                free(pre);
            }
            free_block->size = mem_size-prev;
            free_block->start_addr = prev;
            free_block->next = NULL;

        }
        fbt = free_block;
    }
    // 找到了，分配
    if (fbt->size - request_size > MIN_SLICE)
    {
        ab->start_addr = fbt->start_addr;
        ab->size = request_size;
        fbt->start_addr += request_size;
        fbt->size -= request_size;
    }
    else
    {
        ab->start_addr = fbt->start_addr;
        ab->size = fbt->size;
        if (fbt == free_block)
        {
            free_block = fbt->next;
        }
        else
        {
            pre->next = fbt->next;
```

```
        }
        free(fbt);
    }
    return 1;
}

/*创建新的进程，主要是获取内存的申请数量*/
int new_process()
{
    struct allocated_block *ab;
    int size;
    int ret;
    ab = (struct allocated_block *)malloc(sizeof(struct allocated_block));
    if (!ab)
        exit(-5);
    ab->next = NULL;
    pid++;
    sprintf(ab->process_name, "PROCESS-%02d", pid);
    ab->pid = pid;
    printf("Memory for %s:", ab->process_name);
    scanf("%d", &size);
    char input;
    while ((input = getchar()) != '\n' && input != EOF)
    {
        // 读入一个字符后清空缓存区，避免\n被读入或者一次输入多个字符
    }
    if (size > 0)
        ab->size = size;
    ret = allocate_mem(ab); /* 从空闲区分配内存， ret==1 表示分配 ok*/

    /* 如果此时 allocated_block_head 尚未赋值，则赋值 */
    if ((ret == 1) && (allocated_block_head == NULL))
    {
        allocated_block_head = ab;
        return 1;
    }
    /* 分配成功，将该已分配块的描述插入已分配链表 */
    else if (ret == 1)
    {
        ab->next = allocated_block_head;
        allocated_block_head = ab;
        return 2;
    }
    else if (ret == -1)
    {
        /* 分配不成功 */
```

```
        printf("Allocation fail\n");
        free(ab);
        return -1;
    }
    return 3;
}

/*根据pid查找内存块*/
struct allocated_block *find_process(int pid)
{
    struct allocated_block*ab = allocated_block_head;
    while (ab != NULL)
    {
        if (ab->pid == pid)
            break;
        ab = ab->next;
    }
    return ab;
}

/*将 ab 所表示的已分配区归还，并进行可能的合并*/
int free_mem(struct allocated_block *ab)
{
    int algorithm = ma_algorithm;
    struct free_block_type*fbt, *pre,*work;

    // 进行可能的合并，基本策略如下
    // 1. 将新释放的结点插入到空闲分区队列末尾
    // 2. 对空闲链表按照地址有序排列
    // 3. 检查并合并相邻的空闲分区
    // 4. 将空闲链表重新按照当前算法排序
    // 请自行补充...

    //  头插法
    work = (struct free_block_type *)malloc(sizeof(struct free_block_type));
    if (work == NULL)
    {
        return -1;
    }

    work->start_addr = ab->start_addr;
    work->size = ab->size;
    work->next = free_block;
    free_block = work;

    rearrange_FF(free_block);
```

```
    fbt = free_block;
    while (fbt->next != NULL)
    {
        if (fbt->start_addr + fbt->size == fbt->next->start_addr)
        {
            struct free_block_type *temp = fbt->next;
            fbt->size += fbt->next->size;
            fbt->next = fbt->next->next;
            free(temp);
        }
        else
        {
            fbt = fbt->next;
        }
    }
    rearrange(ma_algorithm);

    return 1;
}

/*释放 ab 数据结构节点*/
int dispose(struct allocated_block *free_ab)
{
    struct allocated_block*pre, *ab;
    if (free_ab == allocated_block_head)
    { /* 如果要释放第一个节点 */
        allocated_block_head = allocated_block_head->next;
        free(free_ab);
        return 1;
    }
    pre = allocated_block_head;
    ab = allocated_block_head->next;
    while (ab != free_ab)
    {
        pre = ab;
        ab = ab->next;
    }
    pre->next = ab->next;
    free(ab);
    return 2;
}

/*删除进程，归还分配的存储空间，并删除描述该进程内存分配的节点*/
void kill_process()
{
    struct allocated_block *ab;
```

```
    int pid;
    printf("Kill Process, pid=");
    scanf("%d", &pid);
    char input;
    while ((input = getchar()) != '\n' && input != EOF)
    {
        // 读入一个字符后清空缓存区，避免\n被读入或者一次输入多个字符
    }
    ab = find_process(pid);
    if (ab != NULL)
    {
        free_mem(ab); /* 释放 ab 所表示的分配区 */
        dispose(ab);  /* 释放 ab 数据结构节点 */
    }
}

/*显示当前内存的使用情况，包括空闲区的情况和已经分配的情况*/
int display_mem_usage()
{
    struct free_block_type *fbt = free_block;
    struct allocated_block*ab = allocated_block_head;

    printf("----------------------------------------------------------\n");
    /* 显示空闲区 */
    printf("Free Memory:\n");
    printf("%20s %20s\n", "start_addr", "size");
    while (fbt != NULL)
    {
        printf("%20d %20d\n", fbt->start_addr, fbt->size);
        fbt = fbt->next;
    }

    /* 显示已分配区 */
    printf("\nUsed Memory:\n");
    printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start_addr",
"size");
    while (ab != NULL)
    {
        printf("%10d %20s %10d %10d\n", ab->pid, ab->process_name, ab-
>start_addr, ab->size);
        ab = ab->next;
    }
    printf("----------------------------------------------------------\n");
}

/*退出,释放所有链表*/
```

```
void do_exit()
{
    struct free_block_type *fbt = free_block;
    struct allocated_block*ab = allocated_block_head;
    while (fbt != NULL)
    {
        free_block = fbt->next;
        free(fbt);
        fbt = free_block;
    }
    while (ab != NULL)
    {
        allocated_block_head = ab->next;
        free(ab);
        ab = allocated_block_head;
    }
}

void tab(char choice)
{
}

int main()
{
    char choice, input;
    pid = 0;
    free_block = init_free_block(mem_size); // 初始化空闲区
    while (1)
    {
        display_menu(); // 显示菜单
        choice = getchar();
        while ((input = getchar()) != '\n' && input != EOF)
        {
            // 读入一个字符后清空缓存区，避免\n被读入或者一次输入多个字符
        }
        switch (choice)
        {
        case '1':
            set_mem_size();
            break; // 设置内存大小
        case '2':
            set_algorithm();
            flag = 1;
            break; // 设置算法
        case '3':
```

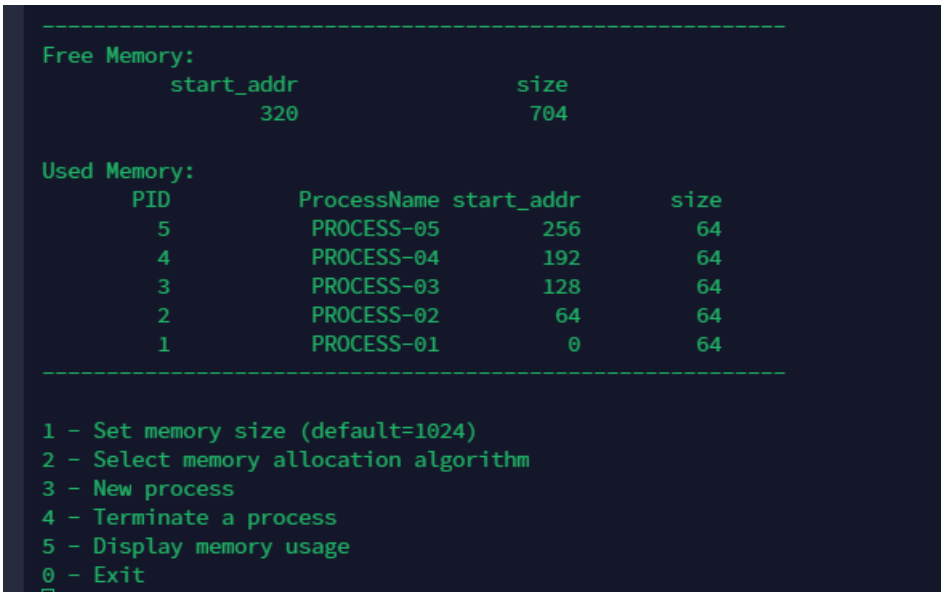
```
            new_process();
            flag = 1;
            break; // 创建新进程
        case '4':
            kill_process();
            flag = 1;
            break; // 删除进程
        case '5':
            display_mem_usage();
            flag = 1;
            break; // 显示内存使用
        case '0':
            do_exit();
            exit(0);
            /*释放链表并退出*/
        default:
            break;
        }
    }
    return 0;
}
```

1. 首先调整了一下代码的格式，将函数定义和实现放在了主函数前，同时发现了两个没有定义的函数`find_pid`和`do_exit`，于是将这两个函数的定义和实现补充了
2. FF和BF和WF算法都是使用的插入排序，只需要修改比较的条件即可，分别比较的是起始地址、空闲块大小和空闲块大小
3. 同时在所有的读入后加入了清空缓存区的代码，避免了回车键导致的一些重复显示问题
4. 代码实质上和数据结构课的链表一致，维护了两个链表，分别表示空闲块和已分配块，每次分配和回收都会对空闲块链表进行重新排序，保证了空闲块链表的有序性

2. **实验结果**
    1. 设置内存空间为1024，采用Best Fit算法

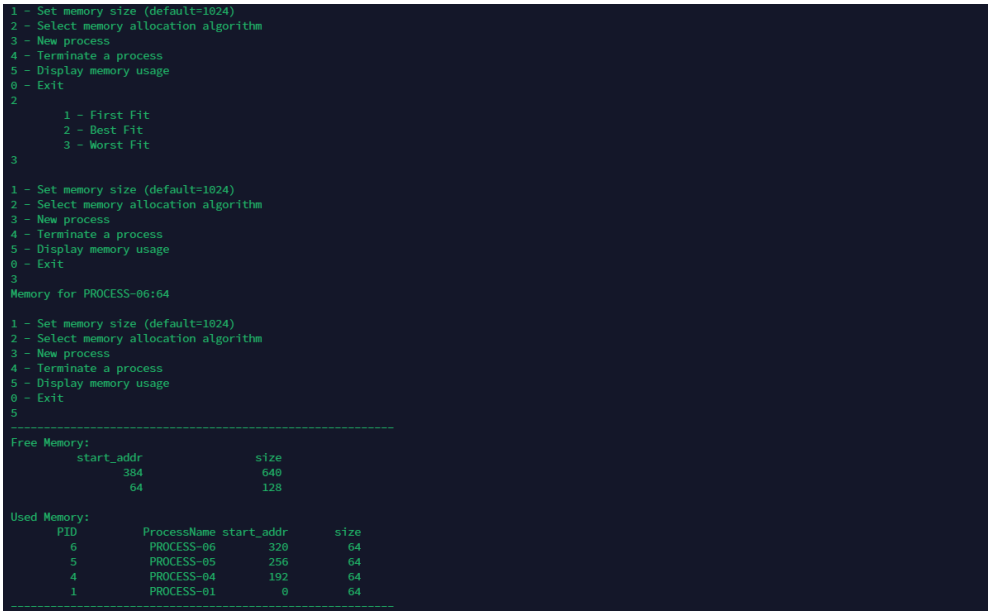
```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
1
Total memory size =1024

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
2
        1 - First Fit
        2 - Best Fit
        3 - Worst Fit
2
```

2. 设置了五个进程，每个进程分配空间为64

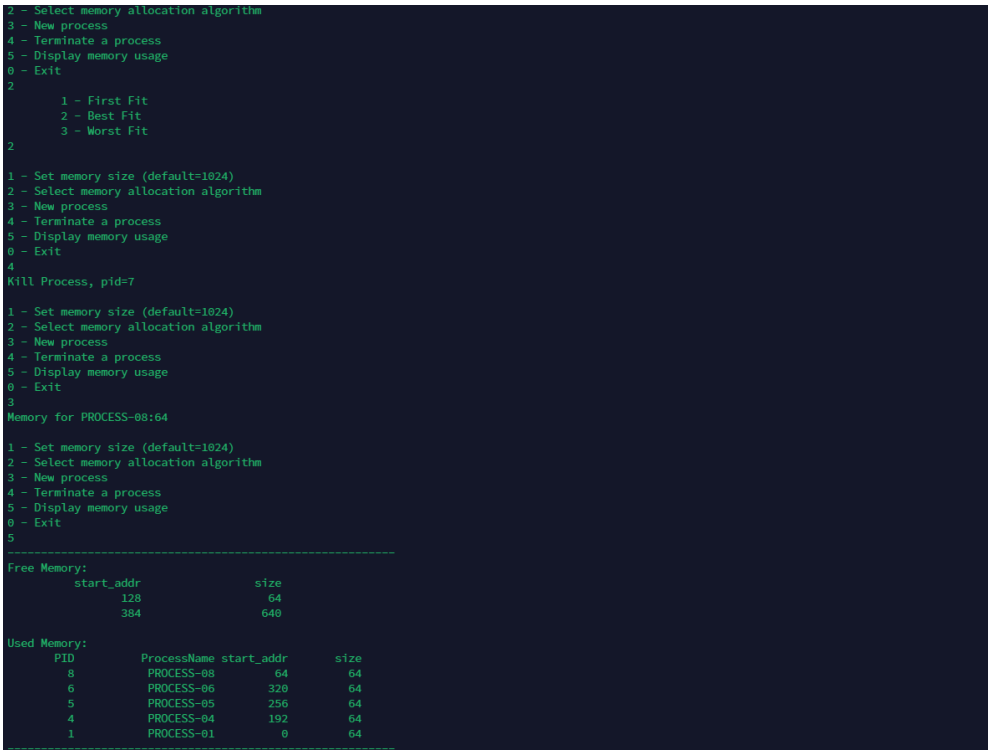


3. 删除进程2和进程3，展示全部进程的内存分配情况，验证了内存回收的正确性

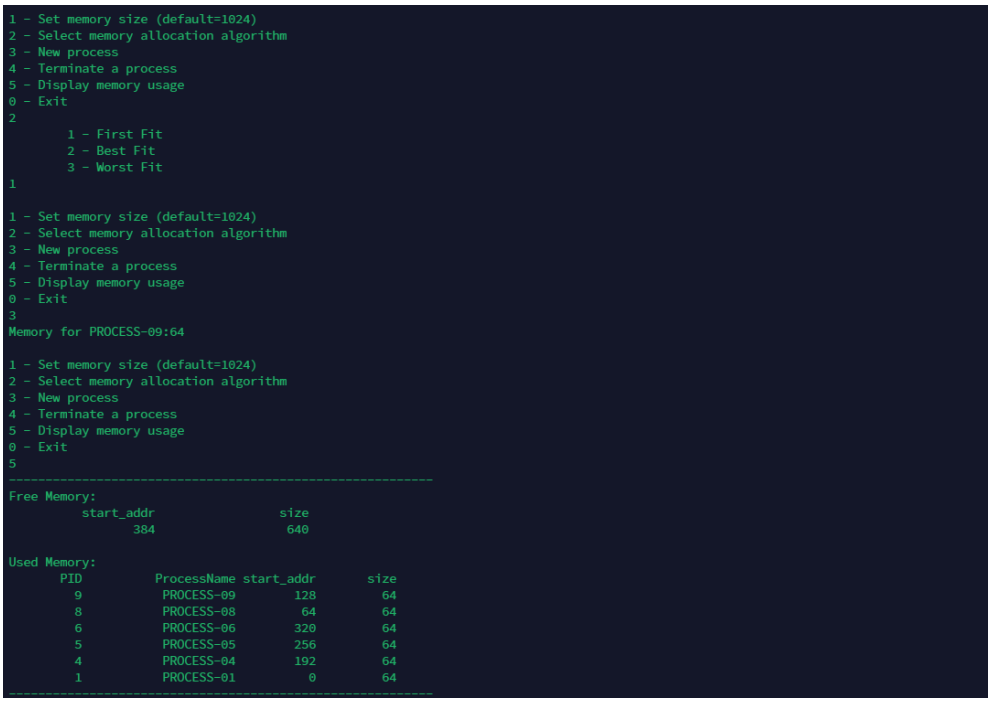
4. 采用worst fit算法，分配一块空间为64的内存，如果实现正确，起始地址应该为320，经验证，实现正确



5. 采用Best Fit算法，分配一块空间为64的内存，如果实现正确，起始地址应该为64，经验证，实现正确

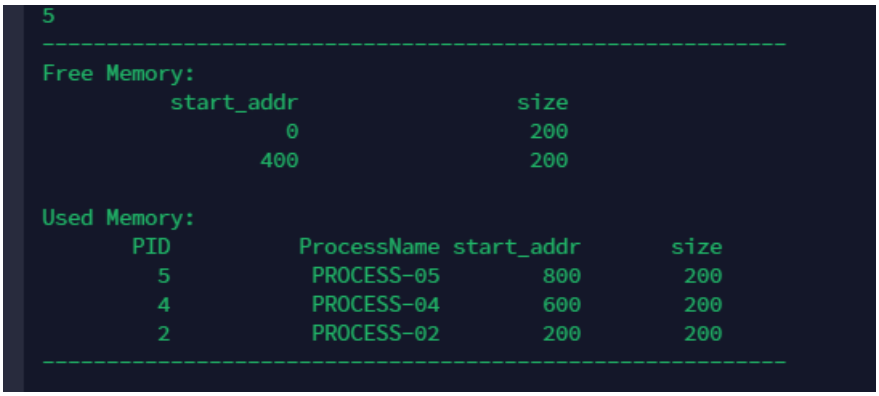


6. 采用First Fit算法，分配一块空间为64的内存，如果实现正确，起始地址应该为128，经验证，实现正确



```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
2
        1 - First Fit
        2 - Best Fit
        3 - Worst Fit
1

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
3
Memory for PROCESS-09:64

1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
         start_addr               size
                384                640

Used Memory:
      PID         ProcessName start_addr       size
        9          PROCESS-09        128         64
        8          PROCESS-08         64         64
        6          PROCESS-06        320         64
        5          PROCESS-05        256         64
        4          PROCESS-04        192         64
        1          PROCESS-01          0         64
------------------------------------------------------------
```

7. 验证内存紧缩的正确性，设置内存为1000，分配五个进程各占用200，释放进程1，3



```
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                   0                 200
                 400                 200

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        800        200
         4           PROCESS-04        600        200
         2           PROCESS-02        200        200
------------------------------------------------------------
```

再新建进程6，分配300内存，会触发内存紧缩，释放所有空闲块，重新分配内存

```
1 - Set memory size (default=1024)
2 - Select memory allocation algorithm
3 - New process
4 - Terminate a process
5 - Display memory usage
0 - Exit
5
------------------------------------------------------------
Free Memory:
          start_addr                size
                 900                 100

Used Memory:
       PID          ProcessName start_addr       size
         6           PROCESS-06        600        300
         2           PROCESS-02          0        200
         4           PROCESS-04        200        200
         5           PROCESS-05        400        200
------------------------------------------------------------
```

3 文件系统

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

用一个文件模拟磁盘，用 fwrite 模仿磁盘的读写操作，用 fseek 模仿磁盘的寻道操作，用 fread 模仿磁盘的读操作。

3.4 实验步骤

1. 定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项。
2. 实现底层函数，包括分配数据块等操作。
3. 实现命令层函数，包括 dir 等操作。
4. 完成 shell 的设计。
5. 测试整个文件系统的功能。

3.5 测试数据设计

1. 创建文件系统，格式化磁盘
2. 在根目录下创建文件夹 1
3. 在文件夹 1 下创建文件夹 2
4. 在文件夹 2 下创建文件 a
5. 打开文件 a，写入数据
6. 打开文件 a，读出数据
7. 修改文件 a 的权限为不可读不可写
8. 打开文件 a，写入数据

9. 打开文件 a，读出数据
10. 回到根目录，删除文件夹 1

3.6 代码简介（具体代码见附件）

1. 定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项。
   (a) define 部分：定义了一些常量，如磁盘大小、块大小、磁盘起始地址等。
   (b) super_block：超级块，用于存储文件系统的信息，如文件系统名、磁盘总大小、块大小，用户名和密码。
   (c) group_desc：块组描述符，用于存储块位图的起始块号、索引结点位图的起始块号、索引结点表的起始块号、本组空闲块的个数、本组空闲索引结点的个数、组中分配给目录的结点数。
   (d) inode：索引结点，用于存储文件的信息，如文件类型及访问权限、文件所占的数据块个数、文件或目录大小、访问时间、创建时间、修改时间、删除时间、直接索引方式指向数据块号。
   (e) dir_entry：目录项，用于存储目录的信息，如索引节点号、目录项长度、文件名长度、文件类型、文件名。
   (f) 定义了一批全局变量，如最近分配的节点号、最近分配的数据块号、当前目录的节点号、当前路径长度、文件打开表、超级块缓冲区、组描述符缓冲区、inode 缓冲区、位图缓冲区、目录项缓冲区、针对数据块的缓冲区、文件写入缓冲区、虚拟磁盘指针、当前路径名。
   (g) 为了方便从文件中读取数据，定义了很多缓冲区，如超级块缓冲区、组描述符缓冲区、inode 缓冲区、位图缓冲区、目录项缓冲区、针对数据块的缓冲区、文件写入缓冲区。
2. 实现底层函数，包括分配数据块等操作。
   (a) 文件系统各部分的读写
   以超级块为例，通过 fopen 打开文件，然后通过 fseek 定位到超级块的位置，再通过 fwrite 写入超级块缓冲区的内容，最后通过 fflush 立刻将缓冲区的内容输出，保证磁盘内存数据的一致性。
   (b) 分配数据块与 inode 节点
   实现思路：记录最近分配的数据块号，然后从这个数据块号开始，从位图循环中寻找空闲的数据块，找到后将该数据块置为 1，然后更新位图，最后更新组描述符中的空闲块个数
   (c) 释放数据块与 inode 节点
   实现思路：通过位图找到要释放的数据块，然后将该数据块置为 0，最后更新组描述符中的空闲块个数

(d) 初始化磁盘和文件系统

初始化磁盘，在当前目录下创建一个名为 Ext2 的文件，然后将其大小设置为 2MB，将各部分内容设置为初始化状态，即格式化。

2. 初始化文件系统：读取 Ext2 文件，然后将各部分内容读取到缓冲区中。

3. 命令行函数

(a) cd 命令：切换当前目录，实现思路：通过 reserch_file 函数查找目录，然后将当前目录的节点号设置为该目录的节点号，最后将当前路径名设置为该目录的路径名。

(b) ls 命令：显示当前目录下的文件和目录，实现思路：通过 reserch_file 函数查找目录，然后通过 reload_dir 函数将目录项读取到目录项缓冲区中，最后通过遍历目录项缓冲区，将目录项的文件名输出

(c) mkdir 命令：创建目录，实现思路：通过 reserch_file 函数查找目录，然后通过 alloc_block 函数分配一个数据块，通过 get_inode 函数得到一个 inode 节点，然后将该目录的信息写入到该 inode 节点中，最后将该目录的信息写入到该数据块中。

(d) touch 命令：创建文件，实现思路：通过 reserch_file 函数查找目录，然后通过 alloc_block 函数分配一个数据块，通过 get_inode 函数得到一个 inode 节点，然后将该文件的信息写入到该 inode 节点中。

(e) rm 命令：删除文件或目录，实现思路：通过 reserch_file 函数查找文件或目录，然后通过 remove_block 函数释放该文件或目录的数据块，通过 remove_inode 函数释放该文件或目录的 inode 节点。

(f) write 命令：写文件，实现思路：先通过循环使用 getchar 函数从键盘读取字符，然后将读取到的字符写入到文件写入缓冲区中，知道读取到 # 为止，然后根据文件大小，判断是否需要使用一级索引或二级索引，最后将文件写入缓冲区中的内容写入到文件中。

为了实现读写保护，在调用 write 函数时，会先读取该文件对应的 inode 节点，取 i_mode 的倒数第二位，如果为 1，则说明该文件可写，否则不可写。

(g) read 命令：读文件，实现思路：先通过 reserch_file 函数查找文件，然后根据文件大小，判断是否需要使用一级索引或二级索引，然后按数据块按字符输出。

为了实行读取保护，在调用读函数后，会先取该文件的 inode 节点，取 i_mode 的倒数第三位，只有在为 1 的时候读出

(h) open 命令：打开文件，实现思路：先通过 reserch_file 函数查找文件，然后将该文件的 inode 节点号写入到文件打开表中

(i) close 命令：关闭文件，实现思路：先通过 reserch_file 函数查找文件，然后将该文件的 inode 节点号从文件打开表中删除

(j) chmod 命令：修改文件权限，实现思路：通过 i_mode & 0x0106 | 0babc 的方式修改文件权限.

(k) help 命令：显示帮助信息，实现思路：printf 输出帮助信息

(l) exit 命令：退出文件系统，实现思路：通过 exit(0) 退出文件系统

(m) format 命令：格式化文件系统，实现思路：通过 initialize_disk 函数初始化磁盘，然后将各部分内容设置为初始化状态，即格式化

4. 完成 shell 的设计

(a) 模仿 Ubuntu 的 shell，实现思路：通过循环，不断读取用户输入的命令，然后通过 strcmp 函数判断用户输入的命令，然后调用相应的函数。

3.7 程序运行初值及运行结果分析

1. 初始化磁盘

初始化磁盘后，设置了 Username 和 Password，然后将其写入到超级块中，然后将各部分内容写入到磁盘中。

```
chenshi@Ubuntu:~/OSlab/lab3$ ./test
The File system does not exist!
Creating the ext2 file system
Please wait ...
Please set your username and password (less than 10 characters)
username: chen
password: chen
The ext2 file system has been installed!
volume name       : EXT2FS
disk size         : 4612(blocks)
blocks per group  : 4612(blocks)
ext2 file size    : 2306(kb)
block size        : 512(kb)
```

图 25: 初始化

2. 登录

使用正确的用户名和密码登录，登录成功。使用 ls 命令查看当前目录下的文件和目录，当前目录下有两个目录，分别是. 和.. ，. 表示当前目录，.. 表示上一级目录。默认权限是 r__w___ , 即可读可写。其中可读指可以 cd 进入该目录，可写指可以在该目录下创建文件或目录。

```
Welcome to ext2 file system!
Username:chen
Password:chen
Login successfully!
chen@Ubuntu:~$ ls
items          type          mode          size              Access time           Creation ctime          Modification mtime
.              <DIR>         r_w__         ----        2023.11.28 16:39:06     2023.11.28 16:39:06     2023.11.28 16:39:06
..             <DIR>         r_w__         ----        2023.11.28 16:39:06     2023.11.28 16:39:06     2023.11.28 16:39:06
chen@Ubuntu:~$
```

图 26: 登录

3. 创建目录和文件，写文件

   由于 a 是无后缀的文件，所以默认为有可读可写可执行权限，写入后。使用 ls 命令查看当前目录下的文件和目录，可以看到 a 文件的大小为 30Byte，权限为 r_w_x ，Access Time 为 Open 的时间，Modify Time 为 Write 的时间，Create Time 为 Create 的时间

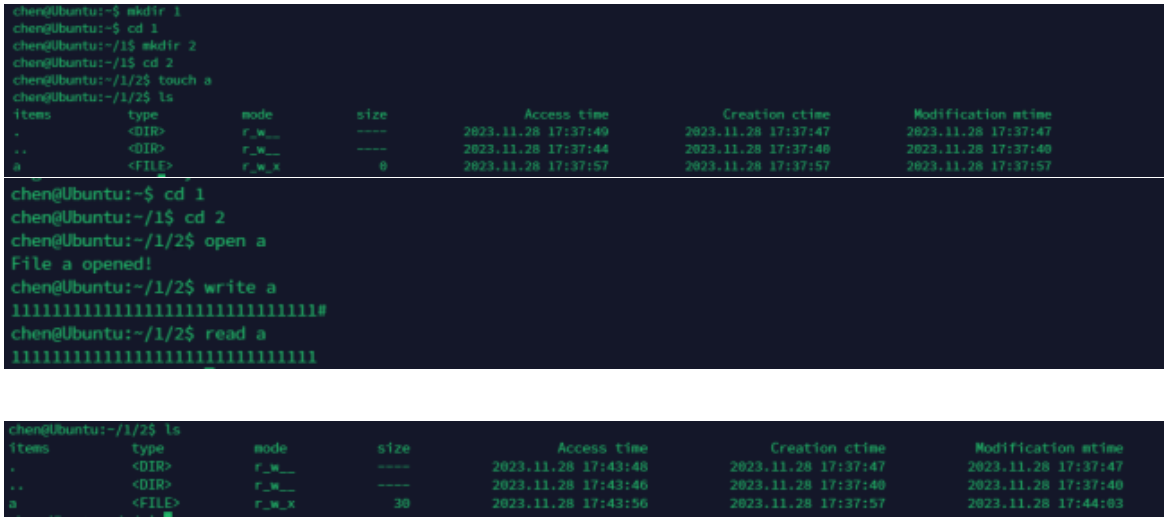


图 27: 创建目录和文件，写文件

4. 验证读写保护

   通过 chmod 命令修改 a 文件的权限为 _____x ，即不可读不可写可执行，然后使用 read 命令读取 a 文件，可以看到读取失败，因为没有读的权限。使用 write 命令写入 a 文件，可以看到写入失败

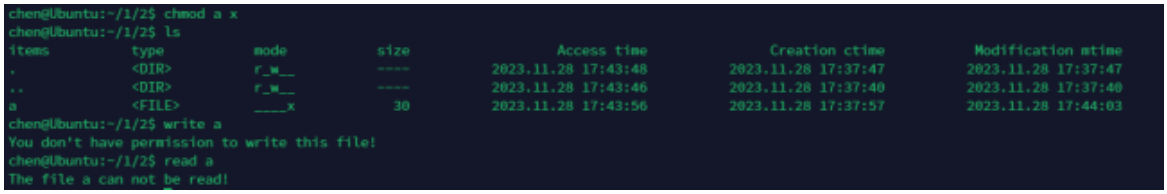


图 28: 验证读写保护

5. 验证删除多级目录

   文件夹 1 下有文件夹为 2，文件夹 2 下有文件 a，然后使用 rm 命令删除文件夹 1，可以看到删除成功，验证了删除多级目录的功能。

   使用 ckdisk 命令检查磁盘，可以看到磁盘信息与初始化磁盘时的信息一致，验证了删除多级目录后，磁盘信息得到了正确的更新，对应的 inode 节点和数据块也被释放。

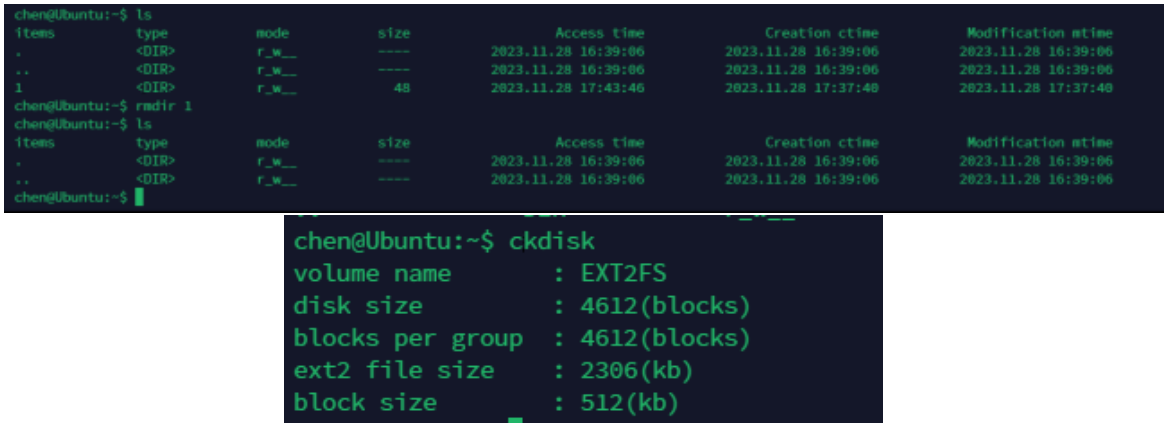


图 29: 验证删除多级目录

3.8 问题回答

1. 在设计文件系统时数据结构遇到的挑战问题

   在设计文件系统时，遇到的挑战问题是如何将数据结构写入到磁盘中，使用 fseek 函数定位到对应位置的时候会比较复杂，因为需要考虑到各个数据结构的大小，所以需要仔细计算每个数据结构的大小。

2. 实现底层函数时遇到的挑战问题

   (a) 实现一级和二级索引的时候，read，write，remove 函数都需要适配一级和二级索引。

(b) 频繁的修改要频繁将缓存区写入文件中，偶尔会忘记调用 load 函数

3. 调试代码时遇到的挑战问题

   不清楚哪里出错，所以需要仔细检查每个函数的实现，然后通过 printf 输出调试信息。

3.9 实验总结

3.9.1 实验中的问题与解决过程

1. 不清楚哪里出错，所以需要仔细检查每个函数的实现，然后通过 printf 输出调试信息。
2. 在设计文件系统时，遇到的挑战问题是如何将数据结构写入到磁盘中，使用 fseek 函数定位到对应位置的时候会比较复杂，因为需要考虑到各个数据结构的大小，所以需要仔细计算每个数据结构的大小。
3. 实现一级和二级索引的时候，read，write，remove 函数都需要适配一级和二级索引。
4. 频繁的修改要频繁将缓存区写入文件中，偶尔会忘记调用 load 函数

3.9.2 实验收获

1. 对文件系统的实现有了更深的理解。
2. 对于 c 的文件操作更加的熟悉。
3. 对于大型项目的实现有了更深的理解。

3.9.3 意见与建议

延长实验时间，让我们有更多的时间去实现更多的功能。两周的时间实在是太短了。

3.10 附件

3.10.1 附件 1 ext2_func.h

```
#ifndef _EXT2_FUNC_H
#define _EXT2_FUNC_H

#define VOLUME_NAME     "EXT2FS"   // 卷名
#define BLOCK_SIZE 512          // 块大小
#define DISK_SIZE       4612       //磁盘总块数

#define DISK_START 0            // 磁盘开始地址
#define SB_SIZE 128                //超级块大小是128

#define GD_SIZE 32                 // 块组描述符大小是32B
#define GDT_START       (0+512)    // 块组描述符起始地址

#define BLOCK_BITMAP (512+512) // 块位图起始地址
#define INODE_BITMAP (1024+512)// inode 位图起始地址

#define INODE_TABLE (1536+512) // 索引节点表起始地址 4*512
#define INODE_SIZE 64           // 每个inode的大小是64B
#define INODE_TABLE_COUNTS      4096 // inode entry 数

#define DATA_BLOCK (263680+512) // 数据块起始地址 4*512+4096*64
#define DATA_BLOCK_COUNTS       4096    // 数据块数

#define BLOCKS_PER_GROUP        4612 // 每组中的块数

#define USER_NAME "root" //用户名
#define PASSWORD "123456" //密码

struct super_block // 32 B
{
        char sb_volume_name[16]; //文件系统名
        unsigned short sb_disk_size; //磁盘总大小
        unsigned short sb_blocks_per_group; // 每组中的块数
        unsigned short sb_size_per_block;       // 块大小
    char username[10]; //用户名
        char password[10]; //密码
};

struct group_desc // 32 B
{
    char bg_volume_name[16]; //文件系统名
    unsigned short bg_block_bitmap; //块位图的起始块号
```

```
    unsigned short bg_inode_bitmap; //索引结点位图的起始块号
    unsigned short bg_inode_table; //索引结点表的起始块号
    unsigned short bg_free_blocks_count; //本组空闲块的个数
    unsigned short bg_free_inodes_count; //本组空闲索引结点的个数
    unsigned short bg_used_dirs_count; //组中分配给目录的结点数
    char bg_pad[4]; //填充(0xff)
};
struct inode // 64 B
{
    unsigned short i_mode;   //文件类型及访问权限
    unsigned short i_blocks; //文件所占的数据块个数
    unsigned int i_size;    // 文件或目录大小(单位 byte)
    unsigned long i_atime;   //访问时间
    unsigned long i_ctime;   //创建时间
    unsigned long i_mtime;   //修改时间
    unsigned long i_dtime;   //删除时间
    unsigned short i_block[8]; //直接索引方式 指向数据块号
    char i_pad[8];           //填充(0xff)
};
struct dir_entry //16B
{
    unsigned short inode; //索引节点号
    unsigned short rec_len; //目录项长度
    unsigned short name_len; //文件名长度
    char file_type; //文件类型(1 普通文件 2 目录.. )
    char name[9]; //文件名
};


static unsigned short last_alloc_inode; // 最近分配的节点号 */
static unsigned short last_alloc_block; // 最近分配的数据块号 */
static unsigned short current_dir;   // 当前目录的节点号 */

static unsigned short current_dirlen; // 当前路径长度 */

static short fopen_table[16]; // 文件打开表 */

static struct super_block sb_block[1];  // 超级块缓冲区
static struct group_desc gdt[1];        // 组描述符缓冲区
static struct inode inode_area[1];  // inode缓冲区
static unsigned char bitbuf[512]={0}; // 位图缓冲区
static unsigned char ibuf[512]={0};
static struct dir_entry dir[32];   // 目录项缓冲区 32*16=512
```

```
static char Buffer[512];  // 针对数据块的缓冲区
static char tempbuf[2*1024*1024];       // 文件写入缓冲区
static FILE *fp;        // 虚拟磁盘指针


extern char current_path[256];    // 当前路径名 */

static unsigned long getCurrentTime();//得到当前时间
static void update_super_block(void);   // 更新超级块内容
static void reload_super_block(void);   //加载超级块内容
static void update_group_desc(void);    //更新组描述符内容
static void reload_group_desc(void);    //加载组描述符内容
static void update_inode_entry(unsigned short i); //更新indoe表
static void reload_inode_entry(unsigned short i); //加载inode表
static void update_block_bitmap(void);  //更新块位图
static void reload_block_bitmap(void);  //加载块位图
static void update_inode_bitmap(void);  //更新inode位图
static void reload_inode_bitmap(void);  //加载inode位图
static void update_dir(unsigned short i);//更新目录
static void reload_dir(unsigned short i);//加载目录
static void update_block(unsigned short i);//更新数据块
static void reload_block(unsigned short i);//加载数据库
static int alloc_block(void);//分配数据块
static int get_inode(void); //得到inode节点
static unsigned short reserch_file(char tmp[9],int file_type,unsigned short *inode_num,unsigned
static void dir_prepare(unsigned short tmp,unsigned short len,int type);
static void remove_block(unsigned short del_num);//删除数据块
static void remove_inode(unsigned short del_num);//删除inode节点
static unsigned short search_file(unsigned short Ino);//在打开文件表中查找是否已打开文件
static void sleep(int k);
static void initialize_disk(void);//初始化磁盘


#endif
```

3.10.2 附件 2 ext2_func.c

```c
#include <stdio.h>
#include <string.h>
#include "ext2_func.h"
#include "shell.h"
#include <time.h>
#include <stdlib.h>

char current_path[256]; //当前路径名
char path_head[256]; //路径名头

static unsigned long getCurrentTime() {
    // 获取当前时间
    time_t rawtime;
    time(&rawtime);

    // 将 time_t 转换为 unsigned long
    unsigned long currentTimeStamp = (unsigned long)rawtime;

    return currentTimeStamp;
}

char* convertTimeStampToString(unsigned long timestamp) {
    // 将 unsigned long 转换为 time_t
    time_t rawtime = (time_t)timestamp;

    // 将 time_t 转换为 struct tm
    struct tm *timeinfo = localtime(&rawtime);

    // 格式化时间
    static char buffer[20];  // 用于存储格式化后的时间字符串
    strftime(buffer, sizeof(buffer), "%Y.%m.%d %H:%M:%S", timeinfo);

    return buffer;
}

static void update_super_block(void) //写超级块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DISK_START,SEEK_SET);
    fwrite(sb_block,SB_SIZE,1,fp);
    fflush(fp); //立刻将缓冲区的内容输出，保证磁盘内存数据的一致性
}
```

```
static void reload_super_block(void) //读超级块
{
    fseek(fp,DISK_START,SEEK_SET);
    fread(sb_block,SB_SIZE,1,fp);//读取内容到超级块缓冲区中
}

static void update_group_desc(void) //写组描述符
{
    fp=fopen("./Ext2","r+");
    fseek(fp,GDT_START,SEEK_SET);
    fwrite(gdt,GD_SIZE,1,fp);
    fflush(fp);
}

static void reload_group_desc(void) // 读组描述符
{

    fseek(fp,GDT_START,SEEK_SET);
    fread(gdt,GD_SIZE,1,fp);
}

static void update_inode_entry(unsigned short i) // 写第i个inode
{
    fp=fopen("./Ext2","r+");
    fseek(fp,INODE_TABLE+(i-1)*INODE_SIZE,SEEK_SET);
    fwrite(inode_area,INODE_SIZE,1,fp);
    fflush(fp);
}

static void reload_inode_entry(unsigned short i) // 读第i个inode
{
    fseek(fp,INODE_TABLE+(i-1)*INODE_SIZE,SEEK_SET);
    fread(inode_area,INODE_SIZE,1,fp);
}

static void update_dir(unsigned short i) //   写第i个 数据块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fwrite(dir,BLOCK_SIZE,1,fp);
    fflush(fp);
}
```

```
static void reload_dir(unsigned short i) // 读第i个 数据块
{
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fread(dir,BLOCK_SIZE,1,fp);
    //fclose(fp);
}

static void update_block_bitmap(void) //写block位图
{
    fp=fopen("./Ext2","r+");
    fseek(fp,BLOCK_BITMAP,SEEK_SET);
    fwrite(bitbuf,BLOCK_SIZE,1,fp);
    fflush(fp);
}

static void reload_block_bitmap(void) //读block位图
{
    fseek(fp,BLOCK_BITMAP,SEEK_SET);
    fread(bitbuf,BLOCK_SIZE,1,fp);
}

static void update_inode_bitmap(void) //写inode位图
{
    fp=fopen("./Ext2","r+");
    fseek(fp,INODE_BITMAP,SEEK_SET);
    fwrite(ibuf,BLOCK_SIZE,1,fp);
    fflush(fp);
}

static void reload_inode_bitmap(void) // 读inode位图
{
    fseek(fp,INODE_BITMAP,SEEK_SET);
    fread(ibuf,BLOCK_SIZE,1,fp);
}

static void update_block(unsigned short i) // 写第i个数据块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fwrite(Buffer,BLOCK_SIZE,1,fp);
    fflush(fp);
}
```

```
static void reload_block(unsigned short i) // 读第i个数据块
{
    fseek(fp,DATA_BLOCK+i*BLOCK_SIZE,SEEK_SET);
    fread(Buffer,BLOCK_SIZE,1,fp);
}


static int alloc_block(void) // 分配一个数据块,返回数据块号
{

    //bitbuf共有512个字节，表示4096个数据块。根据last_alloc_block/8计算它在bitbuf的哪一个字节

    unsigned short cur=last_alloc_block;
    //printf("cur: %d\n",cur);
    unsigned char con=128; // 1000 0000b
    int flag=0;
    if(gdt[0].bg_free_blocks_count==0)
    {
        printf("There is no block to be alloced!\n");
        return(0);
    }
    reload_block_bitmap();
    cur/=8;
    while(bitbuf[cur]==255)//该字节的8个bit都已有数据
    {
        if(cur==511)cur=0; //最后一个字节也已经满，从头开始寻找
        else cur++;
    }
    while(bitbuf[cur]&con) //在一个字节中找具体的某一个bit
    {
        con=con/2;
        flag++;
    }
    bitbuf[cur]=bitbuf[cur]+con;
    last_alloc_block=cur*8+flag;

    update_block_bitmap();
    gdt[0].bg_free_blocks_count--;
    update_group_desc();
    return last_alloc_block;
}
```

```
static int get_inode(void) // 分配一个inode
{
    unsigned short cur=last_alloc_inode;
    unsigned char con=128;
    int flag=0;
    if(gdt[0].bg_free_inodes_count==0)
    {
        printf("There is no Inode to be alloced!\n");
        return 0;
    }
    reload_inode_bitmap();

    cur=(cur-1)/8;   //第一个标号是1，但是存储是从0开始的
    //printf("%s",)
    while(ibuf[cur]==255) //先看该字节的8个位是否已经填满
    {
        if(cur==511)cur=0;
        else cur++;
    }
    while(ibuf[cur]&con)  //再看某个字节的具体哪一位没有被占用
    {
        con=con/2;
        flag++;
    }
    ibuf[cur]=ibuf[cur]+con;
    last_alloc_inode=cur*8+flag+1;
    update_inode_bitmap();
    gdt[0].bg_free_inodes_count--;
    update_group_desc();
    return last_alloc_inode;
}

//当前目录中查找文件或目录为tmp，并得到该文件的 inode 号，它在上级目录中的数据块号以及数据块中目录的项号

static unsigned short reserch_file(char tmp[9],int file_type,unsigned short *inode_num,

{
    unsigned short j,k;
    reload_inode_entry(current_dir); //进入当前目录
    j=0;
    while(j<inode_area[0].i_blocks)
    {
        reload_dir(inode_area[0].i_block[j]);
```

```
        k=0;
        while(k<32)
        {
            if(!dir[k].inode||dir[k].file_type!=file_type||strcmp(dir[k].name,tmp))
            {
                k++;
            }
            else
            {
                *inode_num=dir[k].inode;
                *block_num=j;
                *dir_num=k;
                return 1;
            }
        }
        j++;
    }
    return 0;
}

/*为新增目录或文件分配 dir_entry
对于新增文件，只需分配一个inode号
对于新增目录，除了inode号外，还需要分配数据区存储.和..两个目录项*/

static void dir_prepare(unsigned short tmp,unsigned short len,int type)
{
    reload_inode_entry(tmp);

    if(type==2) // 目录
    {
        inode_area[0].i_size=32;
        inode_area[0].i_blocks=1;
        inode_area[0].i_block[0]=alloc_block();
        unsigned long temp_time=getCurrentTime();
        inode_area[0].i_atime=temp_time;
        inode_area[0].i_ctime=temp_time;
        inode_area[0].i_mtime=temp_time;
        inode_area[0].i_dtime=0;
        dir[0].inode=tmp;
        dir[1].inode=current_dir;
        dir[0].name_len=len;
        dir[1].name_len=current_dirlen;
        dir[0].file_type=dir[1].file_type=2;
```

```
        for(type=2;type<32;type++)
            dir[type].inode=0;
        strcpy(dir[0].name,".");
        strcpy(dir[1].name,"..");
        update_dir(inode_area[0].i_block[0]);

        inode_area[0].i_mode=0b0000001000000110;
    }
    else
    {
        inode_area[0].i_size=0;
        inode_area[0].i_blocks=0;
        inode_area[0].i_mode=0b0000000100000110;
        unsigned long temp_time=getCurrentTime();
        inode_area[0].i_atime=temp_time;
        inode_area[0].i_ctime=temp_time;
        inode_area[0].i_mtime=temp_time;
        inode_area[0].i_dtime=0;

    }
    update_inode_entry(tmp);
}


//删除一个块号

static void remove_block(unsigned short del_num)
{
    unsigned short tmp;
    tmp=del_num/8;
    reload_block_bitmap();
    switch(del_num%8) // 更新block位图 将具体的位置为0
    {
        case 0:bitbuf[tmp]=bitbuf[tmp]&127;break; // bitbuf[tmp] & 0111 1111b
        case 1:bitbuf[tmp]=bitbuf[tmp]&191;break; //bitbuf[tmp]  & 1011 1111b
        case 2:bitbuf[tmp]=bitbuf[tmp]&223;break; //bitbuf[tmp]  & 1101 1111b
        case 3:bitbuf[tmp]=bitbuf[tmp]&239;break; //bitbbuf[tmp] & 1110 1111b
        case 4:bitbuf[tmp]=bitbuf[tmp]&247;break; //bitbuf[tmp]  & 1111 0111b
        case 5:bitbuf[tmp]=bitbuf[tmp]&251;break; //bitbuf[tmp]  & 1111 1011b
        case 6:bitbuf[tmp]=bitbuf[tmp]&253;break; //bitbuf[tmp]  & 1111 1101b
        case 7:bitbuf[tmp]=bitbuf[tmp]&254;break; // bitbuf[tmp] & 1111 1110b
    }
```

```
    update_block_bitmap();
    gdt[0].bg_free_blocks_count++;
    update_group_desc();
}


//删除一个inode 号

static void remove_inode(unsigned short del_num)
{
    unsigned short tmp;
    tmp=(del_num-1)/8;
    reload_inode_bitmap();
    switch((del_num-1)%8)//更改block位图
    {
        case 0:bitbuf[tmp]=bitbuf[tmp]&127;break;
        case 1:bitbuf[tmp]=bitbuf[tmp]&191;break;
        case 2:bitbuf[tmp]=bitbuf[tmp]&223;break;
        case 3:bitbuf[tmp]=bitbuf[tmp]&239;break;
        case 4:bitbuf[tmp]=bitbuf[tmp]&247;break;
        case 5:bitbuf[tmp]=bitbuf[tmp]&251;break;
        case 6:bitbuf[tmp]=bitbuf[tmp]&253;break;
        case 7:bitbuf[tmp]=bitbuf[tmp]&254;break;
    }
    update_inode_bitmap();
    gdt[0].bg_free_inodes_count++;
    update_group_desc();
}

//在打开文件表中查找是否已打开文件
static unsigned short search_file(unsigned short Inode)
{
    unsigned short fopen_table_point=0;
    while(fopen_table_point<16&&fopen_table[fopen_table_point++]!=Inode);
    if(fopen_table_point==16)
    {
        return 0;
    }
    return 1;
}

void initialize_disk(void)  //初始化磁盘
{
```

```
int i=0;
printf("Creating the ext2 file system\n");
printf("Please wait ");
while(i<1)
{
    printf("... ");
    ++i;
}
printf("\n");
last_alloc_inode=1;
last_alloc_block=0;
for(i=0;i<16;i++)
{
    fopen_table[i]=0; //清空缓冲表
}
for(i=0;i<BLOCK_SIZE;i++)
{
    Buffer[i]=0; // 清空缓冲区
}
if(fp!=NULL)
{
    fclose(fp);
}
fp=fopen("./Ext2","w+"); //此文件大小是4612*512=2361344B，用此文件来模拟文件系统
fseek(fp,DISK_START,SEEK_SET);//将文件指针从0开始
for(i=0;i<DISK_SIZE;i++)
{
    fwrite(Buffer,BLOCK_SIZE,1,fp); // 清空文件，即清空磁盘全部用0填充 Buffer为缓冲区起始地址，B
}
// 初始化超级块内容
char username[10]; //用户名
    char password[10]; //密码
printf("Please set your username and password (less than 10 characters)\n");
printf("username: ");
scanf("%s",username);
printf("password: ");
scanf("%s",password);
reload_super_block();
strcpy(sb_block[0].sb_volume_name,VOLUME_NAME);
strcpy(sb_block[0].username,username);
strcpy(sb_block[0].password,password);
sb_block[0].sb_disk_size=DISK_SIZE;
sb_block[0].sb_blocks_per_group=BLOCKS_PER_GROUP;
```

```
sb_block[0].sb_size_per_block=BLOCK_SIZE;
update_super_block();
// 根目录的inode号为1
reload_inode_entry(1);

reload_dir(0);
char temp1[256] = "";
strcpy(temp1,sb_block[0].username);
strcat(temp1,"@Ubuntu:~");
strcpy(path_head, temp1);
strcpy(current_path, path_head); // 修改路径名为根目录
// 初始化组描述符内容
reload_group_desc();

gdt[0].bg_block_bitmap=BLOCK_BITMAP; //第一个块位图的起始地址
gdt[0].bg_inode_bitmap=INODE_BITMAP; //第一个inode位图的起始地址
gdt[0].bg_inode_table=INODE_TABLE;   //inode表的起始地址
gdt[0].bg_free_blocks_count=DATA_BLOCK_COUNTS; //空闲数据块数
gdt[0].bg_free_inodes_count=INODE_TABLE_COUNTS; //空闲inode数
gdt[0].bg_used_dirs_count=0; // 初始分配给目录的节点数是0
update_group_desc(); // 更新组描述符内容

reload_block_bitmap();
reload_inode_bitmap();

unsigned long temp_time=getCurrentTime();
inode_area[0].i_mode=518;
inode_area[0].i_blocks=0;
inode_area[0].i_size=32;
inode_area[0].i_atime=temp_time;
inode_area[0].i_ctime=temp_time;
inode_area[0].i_mtime=temp_time;
inode_area[0].i_dtime=0;
inode_area[0].i_block[0]=alloc_block(); //分配数据块
//printf("%d f\n",inode_area[0].i_block[0]);
inode_area[0].i_blocks++;
current_dir=get_inode();
update_inode_entry(current_dir);

//初始化根目录的目录项
dir[0].inode=dir[1].inode=current_dir;
dir[0].name_len=0;
dir[1].name_len=0;
```

```
    dir[0].file_type=dir[1].file_type=2;
    strcpy(dir[0].name,".");
    strcpy(dir[1].name,"..");
    update_dir(inode_area[0].i_block[0]);
    printf("The ext2 file system has been installed!\n");
    check_disk();
    fclose(fp);
}

//初始化内存
void initialize_memory(void)
{
    int i=0;
    last_alloc_inode=1;
    last_alloc_block=0;
    for(i=0;i<16;i++)
    {
        fopen_table[i]=0;
    }
    current_dir=1;
    fp=fopen("./Ext2","r+");
    if(fp==NULL)
    {
        printf("The File system does not exist!\n");
        initialize_disk();
        exit(0);
        return ;
    }
    reload_super_block();
    char temp1[256] = "";
    strcpy(temp1,sb_block[0].username);
    strcat(temp1,"@Ubuntu:~");
    strcpy(path_head, temp1);
    strcpy(current_path, path_head); // 修改路径名为根目录
    if(strcmp(sb_block[0].sb_volume_name,VOLUME_NAME))
    {
        printf("The File system [%s] is not suppoted yet!\n", sb_block[0].sb_volume_name);
        printf("The File system loaded error!\n");
        initialize_disk();
        return ;
    }
    reload_group_desc();
}
```

```
//格式化
void format(void)
{
    initialize_disk();
    initialize_memory();
}

//进入某个目录，实际上是改变当前路径
void cd(char tmp[9])
{
    //返回根目录
    if(!strcmp(tmp,"~"))
    {
        strcpy(current_path,path_head);
        current_dir=1;
        current_dirlen=0;
        return;
    }
    unsigned short i,j,k,flag;
    flag=reserch_file(tmp,2,&i,&j,&k);
    if(flag)
    {
        reload_inode_entry(i);
        if (inode_area[0].i_mode&1 != 1)
        {
            printf("You don't have permission to enter this directory!\n");
            return;
        }
        inode_area[0].i_atime=getCurrentTime();
        update_inode_entry(i);

        current_dir=i;
        if(!strcmp(tmp,"..")&&dir[k-1].name_len) /* 到上一级目录且不是..目录 */
        {
            current_path[strlen(current_path)-dir[k-1].name_len-1]='\0';
            current_dirlen=dir[k].name_len;
        }
        else if(!strcmp(tmp,"."))
        {
                return ;
        }
        else if(strcmp(tmp,".")) // cd 到子目录
```

```
        {
            strcat(current_path,"/");
            current_dirlen=strlen(tmp);
            strcat(current_path,tmp);
        }
    }
    else
    {
        printf("The directory %s not exists!\n",tmp);
    }
}

// 创建目录
void mkdir(char tmp[9],int type)
{
    //printf("%s %d\n",tmp,type);
    unsigned short tmpno,i,j,k,flag;

    // 当前目录下新增目录或文件
    reload_inode_entry(current_dir);
    if(!reserch_file(tmp,type,&i,&j,&k)) // 未找到同名文件
    {
        if(inode_area[0].i_size==4096) // 目录项已满
        {
            printf("Directory has no room to be alloced!\n");
            return;
        }
        flag=1;
        if(inode_area[0].i_size!=inode_area[0].i_blocks*512) // 目录中有某些块中32个 dir_entry 未
        {
            i=0;
            while(flag&&i<inode_area[0].i_blocks)
            {
                reload_dir(inode_area[0].i_block[i]);
                j=0;
                while(j<32)
                {
                    if(dir[j].inode==0)
                    {
                        flag=0; //找到某个未装满目录项的块
                        break;
                    }
                    j++;
```

```
                }
                i++;
            }
            tmpno=dir[j].inode=get_inode();

            dir[j].name_len=strlen(tmp);
            dir[j].file_type=type;
            strcpy(dir[j].name,tmp);
            update_dir(inode_area[0].i_block[i-1]);
        }
        else // 全满 新增加块
        {
            inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
            inode_area[0].i_blocks++;
            reload_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
            tmpno=dir[0].inode=get_inode();
            dir[0].name_len=strlen(tmp);
            dir[0].file_type=type;
            strcpy(dir[0].name,tmp);
            // 初始化新块的其余目录项
            for(flag=1;flag<32;flag++)
            {
                dir[flag].inode=0;
            }
            update_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        }
        inode_area[0].i_size+=16;

        update_inode_entry(current_dir);

        // 为新增目录分配 dir_entry
        dir_prepare(tmpno,strlen(tmp),type);
    }
    else  // 已经存在同名文件或目录
    {
        printf("Directory has already existed!\n");
    }

}
//创建文件
void cat(char tmp[9],int type)
{
    unsigned short tmpno,i,j,k,flag;
```

```
reload_inode_entry(current_dir);
if(!reserch_file(tmp,type,&i,&j,&k))
{
    if(inode_area[0].i_size==4096)
    {
        printf("Directory has no room to be alloced!\n");
        return;
    }
    flag=1;
    if(inode_area[0].i_size!=inode_area[0].i_blocks*512)
    {
        i=0;
        while(flag&&i<inode_area[0].i_blocks)
        {
            reload_dir(inode_area[0].i_block[i]);
            j=0;
            while(j<32)
            {
                if(dir[j].inode==0)//找到了未分配的目录项
                {
                    flag=0;
                    break;
                }
                j++;
            }
            i++;
        }
        tmpno=dir[j].inode=get_inode();//分配一个新的inode项
        dir[j].name_len=strlen(tmp);
        dir[j].file_type=type;
        strcpy(dir[j].name,tmp);
        update_dir(inode_area[0].i_block[i-1]);
    }
    else //分配一个新的数据块
    {
        inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
        inode_area[0].i_blocks++;
        reload_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        tmpno=dir[0].inode=get_inode();
        dir[0].name_len=strlen(tmp);
        dir[0].file_type=type;
        strcpy(dir[0].name,tmp);
        //初始化新快其他项目为0
```

```
            for(flag=1;flag<32;flag++)
            {
                dir[flag].inode=0;
            }
            update_dir(inode_area[0].i_block[inode_area[0].i_blocks-1]);
        }
        inode_area[0].i_size+=16;
        update_inode_entry(current_dir);
        //将新增文件的inode节点初始化
        dir_prepare(tmpno,strlen(tmp),type);

        //当文件无后缀或者后缀为.exe时，将文件的权限设置为可执行
        //判断是否为.exe文件
        int condition1 = strcmp(tmp + strlen(tmp) - 4, ".exe");
        //判断是否为无后缀文件，查找文件名中是否有.
        int condition2 = 1;
        for (int i = 0; i < strlen(tmp); i++) {
            if (tmp[i] == '.') {
                condition2 = 0;
                break;
            }
        }
        if (condition1 == 0 || condition2 == 1) {
            reload_inode_entry(tmpno);
            inode_area[0].i_mode = 0b0000001000000111;
            update_inode_entry(tmpno);
        }


    }
    else
    {
        printf("File has already existed!\n");
    }
}

//删除一个空目录
void rmdir(char tmp[9])
{
    unsigned short i,j,k,flag;
    unsigned short m,n;
    unsigned short temp = current_dir;
    if (!strcmp(tmp, "..") || !strcmp(tmp, "."))
```

```
{
    printf("The directory can not be deleted!\n");
    return;
}
flag=reserch_file(tmp,2,&i,&j,&k);
if (flag)
{
    reload_inode_entry(dir[k].inode); // 加载要删除的节点
    if(inode_area[0].i_size==32)  // 只有.and ..
    {
        inode_area[0].i_size=0;
        inode_area[0].i_blocks=0;

        remove_block(inode_area[0].i_block[0]);
        // 更新 tmp 所在父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_area[0].i_block[j]);
        remove_inode(dir[k].inode);
        dir[k].inode=0;
        update_dir(inode_area[0].i_block[j]);
        inode_area[0].i_size-=16;
        flag=0;

        m=1;
        while(flag<32&&m<inode_area[0].i_blocks)
        {
            flag=n=0;
            reload_dir(inode_area[0].i_block[m]);
            while(n<32)
            {
                if(!dir[n].inode)
                {
                    flag++;
                }
                n++;
            }
            //如果删除过后，整个数据块的目录项全都为空。类似于在数组中删除某一个位置
            if(flag==32)
            {
                remove_block(inode_area[0].i_block[m]);
                inode_area[0].i_blocks--;
                while(m<inode_area[0].i_blocks)
                {
```

```
                    inode_area[0].i_block[m]=inode_area[0].i_block[m+1];
                    ++m;
                }
            }
        }
        update_inode_entry(current_dir);
        return;
    }
    else
    {
        for(int l=0;l<inode_area[0].i_blocks;l++)
        {
            reload_dir(inode_area[0].i_block[l]);
            for(int m=0;m<32;m++)
            {
                if(!strcmp(dir[m].name,".")||!strcmp(dir[m].name,"..")||dir[m].inode==0)
                    continue;
                if(dir[m].file_type==2)
                {
                    strcpy(current_path,tmp);
                    current_dir = i;
                    rmdir(dir[m].name);
                    current_dir = temp;
                }
                else if(dir[m].file_type==1)
                {
                    current_dir = i;
                    del(dir[m].name);
                    current_dir = temp;
                }
            }
            if(inode_area[0].i_size==32)
            {
                strcpy(current_path,path_head);
                current_dir=temp;
                rmdir(tmp);
            }
        }
        return;
            printf("Directory is not null!\n");
    }
}
else
```

```
    {
        printf("Directory to be deleted not exists!\n");
    }
}

//删除文件
void del(char tmp[9])
{
    unsigned short i,j,k,m,n,flag;
    m=0;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag)
    {
        //删除文件需要父目录的写权限
        reload_inode_entry(current_dir);
        if (inode_area[0].i_mode&2 != 2)
        {
            printf("You don't have permission to delete this file!\n");
            return;
        }
        flag = 0;
        // 若文件 tmp 已打开，则将对应的 fopen_table 项清0
        while(fopen_table[flag]!=dir[k].inode&&flag<16)
        {
                flag++;
        }
        if(flag<16)
        {
                fopen_table[flag]=0;
        }
        reload_inode_entry(i); // 加载删除文件 inode
        //删除文件对应的数据块
        while(m<inode_area[0].i_blocks)
        {
                remove_block(inode_area[0].i_block[m++]);
        }
        inode_area[0].i_blocks=0;
        inode_area[0].i_size=0;
        remove_inode(i);
        // 更新父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_area[0].i_block[j]);
        dir[k].inode=0; //删除inode节点
```

```
        update_dir(inode_area[0].i_block[j]);
        inode_area[0].i_size-=16;
        m=1;
        //删除一项后整个数据块为空，则将该数据块删除
        while(m<inode_area[i].i_blocks)
        {
            flag=n=0;
            reload_dir(inode_area[0].i_block[m]);
            while(n<32)
            {
                if(!dir[n].inode)
                {
                        flag++;
                }
                n++;
            }
            if(flag==32)
            {
                remove_block(inode_area[i].i_block[m]);
                inode_area[i].i_blocks--;
                while(m<inode_area[i].i_blocks)
                {
                        inode_area[i].i_block[m]=inode_area[i].i_block[m+1];
                        ++m;
                }
            }
        }
        update_inode_entry(current_dir);
    }
    else
    {
        printf("The file %s not exists!\n",tmp);
    }
}

//打开文件
void open_file(char tmp[9])
{
    unsigned short flag,i,j,k;
    flag=reserch_file(tmp,1,&i,&j,&k);
    reload_inode_entry(i);
    inode_area[0].i_atime=getCurrentTime();
```

```
    update_inode_entry(i);
    if(flag)
    {
        if(search_file(dir[k].inode))
        {
                printf("The file %s has opened!\n",tmp);
        }
        else
        {
            flag=0;
            while(fopen_table[flag])
            {
                flag++;
            }
            fopen_table[flag]=dir[k].inode;
            printf("File %s opened!\n",tmp);
        }
    }
    else printf("The file %s does not exist!\n",tmp);
}

//关闭文件
void close_file(char tmp[9])
{
    unsigned short flag,i,j,k;
    flag=reserch_file(tmp,1,&i,&j,&k);

    if(flag)
    {
        if(search_file(dir[k].inode))
        {
            flag=0;
            while(fopen_table[flag]!=dir[k].inode)
            {
                flag++;
            }
            fopen_table[flag]=0;
            printf("File %s closed!\n",tmp);
        }
        else
        {
                printf("The file %s has not been opened!\n",tmp);
        }
```

```
    }
    else
    {
        printf("The file %s does not exist!\n",tmp);
    }
}

// 读文件
void read_file(char tmp[9])
{
    unsigned short flag,i,j,k,t;
    unsigned short b1,b2,b3;
    b1=b2=b3=0;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag)
    {
        if(search_file(dir[k].inode)) //读文件的前提是该文件已经打开
        {
            reload_inode_entry(dir[k].inode);
            //判断是否有读的权限
            if(!(inode_area[0].i_mode&4)) // i_mode:111b:读,写,执行
            {
                printf("The file %s can not be read!\n",tmp);
                return;
            }
            //输出直接索引的内容
            if (inode_area[0].i_blocks<=6){
                b1=inode_area[0].i_blocks;
            }else if (inode_area[0].i_blocks>6){
                b1=6;
            }
            if (b1>0){
                for(flag=0;flag<b1;flag++)
                {
                    reload_block(inode_area[0].i_block[flag]);
                    for(t=0;t<inode_area[0].i_size-flag*512;++t)
                    {
                        printf("%c",Buffer[t]);
                    }
                }
            }

            //输出一级索引的内容
```

```
if(inode_area[0].i_blocks>6&&inode_area[0].i_blocks<=262)
{
    b2=inode_area[0].i_blocks-6;
}else if (inode_area[0].i_blocks>262){
    b2=256;
}
if (b2>0){
    reload_block(inode_area[0].i_block[6]);
    char index_1[512];
    memcpy(index_1,Buffer,512);
    for (flag = 0; flag < b2; flag++)
    {
        unsigned short block_num_1;
        memcpy(&block_num_1, &index_1[flag * 2], sizeof(unsigned short));
        reload_block(block_num_1);
        for (t = 0; t < inode_area[0].i_size - (flag + 6) * 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
}

//输出二级索引的内容
if(inode_area[0].i_blocks>262&&inode_area[0].i_blocks<=65818)
{
    b3=inode_area[0].i_blocks-262;
}

if (b3>0){
    reload_block(inode_area[0].i_block[7]);
    char index_2[512];
    memcpy(index_2,Buffer,512);
    for (flag = 0; flag < b3; flag++)
    {
        unsigned short block_num_2;
        memcpy(&block_num_2, &index_2[flag * 2], sizeof(unsigned short));
        reload_block(block_num_2);
        char index_3[512];
        memcpy(index_3,Buffer,512);
        for (t = 0; t < 256; t++)
        {
            unsigned short block_num_3;
            memcpy(&block_num_3, &index_3[t * 2], sizeof(unsigned short));
```

```
                    reload_block(block_num_3);
                    for (int m = 0; m < inode_area[0].i_size - (flag + 262) * 512; ++m)
                    {
                        printf("%c", Buffer[m]);
                    }
                }
            }
        }


        if(inode_area[0].i_blocks==0)
        {
            printf("The file %s is empty!\n",tmp);
        }
        else
        {
            printf("\n");
        }
    }
    else
    {
            printf("The file %s has not been opened!\n",tmp);
    }
}
else printf("The file %s not exists!\n",tmp);
}

void write_file(char tmp[9]) // 写文件
{
    unsigned short flag,i,j,k,size=0,need_blocks,length;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag){
        reload_inode_entry(i);
        if ((inode_area[0].i_mode & 2) == 0)
        {
            printf("You don't have permission to write this file!\n");
            return;
        }else{
            write_file_111(tmp);
        }
    }
}
```

```
//文件以覆盖方式写入
void write_file_(char tmp[9]) // 写文件
{
    unsigned short flag,i,j,k,size=0,need_blocks,length;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if (flag)
    {
        reload_inode_entry(i);
        inode_area[0].i_mtime=getCurrentTime();
        update_inode_entry(i);
        if(search_file(dir[k].inode))
        {
            reload_inode_entry(dir[k].inode);
            while(1)
            {
                tempbuf[size]=getchar();
                if(tempbuf[size]=='#')
                {
                    tempbuf[size]='\0';
                    break;
                }
                if(size>=4095)
                {
                    printf("Sorry,the max size of a file is 4KB!\n");
                    break;
                }
                size++;
            }
            if(size>=4095)
            {
                length=4096;
            }
            else
            {
                length=strlen(tempbuf);
            }
            //计算需要的数据块数目
            need_blocks=length/512;
            if(length%512)
            {
                need_blocks++;
            }
            if(need_blocks<9) // 文件最大 8 个 blocks(512 bytes)
```

```
{
    // 分配文件所需块数目
    //因为以覆盖写的方式写，要先判断原有的数据块数目
    if(inode_area[0].i_blocks<=need_blocks)
    {
        while(inode_area[0].i_blocks<need_blocks)
        {
            inode_area[0].i_block[inode_area[0].i_blocks]=alloc_block();
            inode_area[0].i_blocks++;
        }
    }
    else
    {
        while(inode_area[0].i_blocks>need_blocks)
        {
            remove_block(inode_area[0].i_block[inode_area[0].i_blocks - 1]);
            inode_area[0].i_blocks--;
        }
    }
    j=0;
    while(j<need_blocks)
    {
        if(j!=need_blocks-1)
        {
            reload_block(inode_area[0].i_block[j]);
            memcpy(Buffer,tempbuf+j*BLOCK_SIZE,BLOCK_SIZE);
            update_block(inode_area[0].i_block[j]);
        }
        else
        {
            reload_block(inode_area[0].i_block[j]);
            memcpy(Buffer,tempbuf+j*BLOCK_SIZE,length-j*BLOCK_SIZE);
            inode_area[0].i_size=length;
            update_block(inode_area[0].i_block[j]);
        }
        j++;
    }
    update_inode_entry(dir[k].inode);
}
else
{
    printf("Sorry,the max size of a file is 4KB!\n");
}
```

```
        }
        else
        {
                printf("The file %s has not opened!\n",tmp);
        }
    }
    else
    {
        printf("The file %s does not exist!\n",tmp);
    }
}

void write_file_111(char tem[9])
{
    fflush(stdin);
    unsigned short flag,i,j,k,size=0,need_blocks;
    int length1,length2,length3;//1级写入，2级写入，3级写入
    length1=length2=length3=0;
    flag=reserch_file(tem,1,&i,&j,&k);
    if (flag){
        reload_inode_entry(i);
        inode_area[0].i_mtime=getCurrentTime();
        update_inode_entry(i);
        reload_group_desc();
        if(search_file(dir[k].inode)){
            reload_inode_entry(dir[k].inode);
            while(1)
            {
                tempbuf[size]=getchar();
                if(tempbuf[size]=='#')
                {
                    tempbuf[size]='\0';
                    break;
                }
                if(size>=gdt->bg_free_blocks_count*512)//判断文件大小是否超过最大值，超过全部不写入
                {
                    printf("Sorry,the max size of a file is %dKB!\n",gdt->bg_free_blocks_count/
                    printf("Write failed!\n");
                    return;
                }
                size++;
            }
            if (size<=6*512){
```

```
    length1 = strlen(tempbuf);
}else if (size<=12*512){
    length1 = 6*512;
}
if(length1>0){
    need_blocks=length1/512;
    if(length1%512)
    {
        need_blocks++;
    }
    int x = 0;
    while (x <= need_blocks)
    {
        inode_area[0].i_block[x]=alloc_block();
        reload_block(inode_area[0].i_block[x]);
        memcpy(Buffer,tempbuf+x*BLOCK_SIZE,BLOCK_SIZE);
        update_block(inode_area[0].i_block[x]);
        x++;
    }
    if(inode_area[0].i_blocks>need_blocks)//清空剩下的块
    {
        while (x <= 6)
        {
            x++;
            remove_block(inode_area[0].i_block[x]);
        }
    }
}
if (size > 6*512 && size <= 262*512){
    length2 = size - 6*512;
}else if (size > 262*512){
    length2 = 256*512;
}
if (length2){//采用一级索引
    need_blocks=length2/512;
    if(length2%512)
    {
        need_blocks++;
    }
    inode_area[0].i_block[6]=alloc_block();
    reload_block(inode_area[0].i_block[6]);
    char index_1[512];          //一级索引
    memcpy(index_1,Buffer,512);
```

```
    int x;
    for (x = 0; x < need_blocks; x++) // x代表写入的块数
    {
        unsigned short block_index_temp = alloc_block();
        unsigned short* target = (unsigned short*)(index_1 + sizeof(unsigned short)
        memcpy(target, &block_index_temp, sizeof(unsigned short));
        reload_block(block_index_temp);
        memcpy(Buffer,tempbuf+(x+6)*BLOCK_SIZE,BLOCK_SIZE);
        update_block(block_index_temp);
    }
    if(inode_area[0].i_blocks-6>need_blocks)//清空剩下的块
    {
        while(x<=255){
            unsigned short temp;
            memcpy(&temp, &index_1[x * 2], sizeof(unsigned short));
            remove_block(temp);
            x++;
        }
    }
    memcpy(Buffer,index_1,512);
    update_block(inode_area[0].i_block[6]);//更新一级索引
}
if (size > 262*512 ){
    length3 = size - 262*512;
}
if (length3 >0){
    need_blocks=length3/512;
    if(length3%512)
    {
        need_blocks++;
    }
    inode_area[0].i_block[7]=alloc_block();
    unsigned short num_1 = need_blocks/256;
    unsigned short num_2 = need_blocks%256;
    reload_block(inode_area[0].i_block[7]);
    char index_2[512];          //二级索引
    memcpy(index_2,Buffer,512);
    int x;
    for (x = 0; x < num_1; x++) // x代表写入的块数
    {
        unsigned short block_index_temp = alloc_block();
        unsigned short* target = (unsigned short*)(index_2 + sizeof(unsigned short)
        memcpy(target,&block_index_temp,sizeof(unsigned short));
```

```
                    reload_block(block_index_temp);
                    char index_1[512];          //一级索引
                    memcpy(index_1,Buffer,512);
                    int y;
                    for (y = 0; y < 256; y++) // y代表写入的块数
                    {
                        unsigned short block_index_temp = alloc_block();
                        unsigned short* target = (unsigned short*)(index_1 + sizeof(unsigned sh
                        memcpy(target,&block_index_temp,sizeof(unsigned short));
                        reload_block(block_index_temp);
                        memcpy(Buffer,tempbuf+(x*256+y+262)*BLOCK_SIZE,BLOCK_SIZE);
                        update_block(block_index_temp);
                    }
                    memcpy(Buffer,index_1,512);
                    update_block(block_index_temp);
                }
                memcpy(Buffer,index_2,512);
                update_block(inode_area[0].i_block[7]);//更新二级索引
            }
            inode_area[0].i_size=size;
            inode_area[0].i_blocks= size%512==0?size/512:size/512+1;
            update_inode_entry(dir[k].inode);
        }else {
            printf("The file %s has not opened!\n",tem);
        }
    }else{
        printf("The file %s does not exist!\n",tem);
    }
}



//查看目录下的内容
void ls(void)
{
    printf("items          type           mode           size                  Access "
        "time               Creation ctime           Modification mtime\n");
    unsigned short i,j,k,flag;
    i=0;
    reload_inode_entry(current_dir);
    while(i<inode_area[0].i_blocks)
    {
```

```
k=0;
reload_dir(inode_area[0].i_block[i]);
while(k<32)
{
    if(dir[k].inode)
    {
        printf("%s",dir[k].name);
        if(dir[k].file_type==2)
        {
            j=0;
            reload_inode_entry(dir[k].inode);
            if(!strcmp(dir[k].name,".."))
            {
                while(j++<13)
                {
                        printf(" ");
                }
                flag=1;
            }
            else if(!strcmp(dir[k].name,"."))
            {
                while(j++<14)
                {
                        printf(" ");
                }
                flag=0;
            }
            else
            {
                while(j++<15-dir[k].name_len)
                {
                        printf(" ");
                }
                flag=2;
            }
            printf("<DIR>          ");
            switch(inode_area[0].i_mode&7)
            {
                case 1:printf("____x");break;
                case 2:printf("__w__");break;
                case 3:printf("__w_x");break;
                case 4:printf("r____");break;
                case 5:printf("r___x");break;
```

```
            case 6:printf("r_w__");break;
            case 7:printf("r_w_x");break;
        }
        if(flag!=2)
        {
            printf("          ----");
        }
        else
        {
            printf("          ");
            printf("%4d",inode_area[0].i_size);
        }
        printf("%29s",convertTimeStampToString(inode_area[0].i_atime));
        printf("%29s",convertTimeStampToString(inode_area[0].i_ctime));
        printf("%29s",convertTimeStampToString(inode_area[0].i_mtime));

    }
    else if(dir[k].file_type==1)
    {
        j=0;
        reload_inode_entry(dir[k].inode);
        while(j++<15-dir[k].name_len)printf(" ");
        printf("<FILE>         ");
        switch(inode_area[0].i_mode&7)
        {
            case 1:printf("____x");break;
            case 2:printf("__w__");break;
            case 3:printf("__w_x");break;
            case 4:printf("r____");break;
            case 5:printf("r___x");break;
            case 6:printf("r_w__");break;
            case 7:printf("r_w_x");break;
        }
        printf("          ");
        printf("%4d",inode_area[0].i_size);
        printf("%29s",convertTimeStampToString(inode_area[0].i_atime));
        printf("%29s",convertTimeStampToString(inode_area[0].i_ctime));
        printf("%29s",convertTimeStampToString(inode_area[0].i_mtime));
    }
    printf("\n");
}
k++;
reload_inode_entry(current_dir);
```

```
        }
        i++;
    }
}

//检查磁盘状态
void check_disk(void)
{
        reload_super_block();
        printf("volume name       : %s\n", sb_block[0].sb_volume_name);
        printf("disk size         : %d(blocks)\n", sb_block[0].sb_disk_size);
        printf("blocks per group  : %d(blocks)\n", sb_block[0].sb_blocks_per_group);
        printf("ext2 file size    : %d(kb)\n", sb_block[0].sb_disk_size*sb_block[0].sb_size_per_
        printf("block size        : %d(kb)\n", sb_block[0].sb_size_per_block);
}


int ext2_load()
{
    int times=5;
    printf("Welcome to ext2 file system!\n");
    while (1)
    {
        char temp_username[10],temp_password[10];
        printf("Username:");
        fflush(stdout);
        scanf("%s",temp_username);
        printf("Password:");
        fflush(stdout);
        scanf("%s",temp_password);
        if(strcmp(temp_username,sb_block[0].username)==0&&strcmp(temp_password,sb_block[0].passw
            printf("Login successfully!\n");
            if (strcmp(temp_username,USER_NAME)==0&&strcmp(temp_password,PASSWORD)==0){
                printf("You are using the default username and password, please reset your user
                reset_password();
            }
            break;
        }
        else{
            printf("Username or password is wrong!\n");
            printf("You have %d times to try!\n",times);
            times--;
            if(times==0){
```

```
                printf("You have tried too many times!\n");
                return 0;
            }
        }
    }
    return 1;
}

int reset_password(void)
{
    printf("Please input the new username:");
    fflush(stdout);
    scanf("%s",sb_block[0].username);
    printf("Please input the new password:");
    fflush(stdout);
    scanf("%s",sb_block[0].password);
    printf("Reset password successfully!\n");
    update_super_block();
    exit(0);
    return 1;
}

void help(){
    printf("============================================\n");
    printf("%-8s: %s\n", "format", "format the disk");
    printf("%-8s: %s\n", "mkdir", "create a directory");
    printf("%-8s: %s\n", "rmdir", "remove a directory");
    printf("%-8s: %s\n", "cd", "change the current directory");
    printf("%-8s: %s\n", "ls", "list the files in the current directory");
    printf("%-8s: %s\n", "touch", "create a file");
    printf("%-8s: %s\n", "rm", "delete a file");
    printf("%-8s: %s\n", "open", "open a file");
    printf("%-8s: %s\n", "close", "close a file");
    printf("%-8s: %s\n", "read", "read a file");
    printf("%-8s: %s\n", "write", "write a file");
    printf("%-8s: %s\n", "chmod", "change the mode of a file");
    printf("%-8s: %s\n", "help", "show the help information");
    printf("%-8s: %s\n", "quit", "exit the file system");
    printf("============================================\n");
}


//修改文件的权限
```

```
void chmod(char tmp[9],char mod[4])
{
    unsigned short flag,i,j,k,t;
    flag=reserch_file(tmp,1,&i,&j,&k);
    reload_inode_entry(i);
    if(flag){
        unsigned short tt = 0b1111111100000000;
        if (!strcmp(mod, "r")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 4;
        } else if (!strcmp(mod, "w")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 2;
        } else if (!strcmp(mod, "x")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 1;
        } else if (!strcmp(mod, "rw")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 6;
        } else if (!strcmp(mod, "rx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 5;
        } else if (!strcmp(mod, "wx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 3;
        } else if (!strcmp(mod, "rwx")) {
            inode_area[0].i_mode = (inode_area[0].i_mode & tt) | 7;
        } else {
            printf("The mode is wrong!\n");
            return;
        }
        update_inode_entry(i);
    }
}
```

3.10.3 附件 3 shell.h

```
#ifndef _SHELL_H
#define _SHELL_H


extern void help(void);
extern void initialize_memory(void);
extern void format(void);
extern void cd(char tmp[9]);
extern void mkdir(char tmp[9],int type);
extern void cat(char tmp[9],int type);
extern void rmdir(char tmp[9]);
extern void del(char tmp[9]);
extern void open_file(char tmp[9]);
extern void close_file(char tmp[9]);
extern void read_file(char tmp[9]);
extern void write_file(char tmp[9]);
extern void write_file_(char tmp[9]);
extern void write_file_111(char tmp[9]);
extern void ls(void);
extern void check_disk(void);
extern int ext2_load(void);
extern int reset_password(void);
extern void help(void);
extern void chmod(char tmp[9],char mode[4]);

#endif
```

图 30: shell.h

3.10.4 附件 4 shell.c

```c
#include <stdio.h>
#include <string.h>
#include "ext2_func.h"
#include "shell.h"

int main(int argc,char **argv)
{
    char command[10],temp[9];
    initialize_memory();
    int flag = ext2_load();
    while(flag)
    {

        printf("%s$ ",current_path);
        scanf("%s",command);
        //printf("%s h\n",command);
        if(!strcmp(command,"cd")) //进入当前目录下
        {
            scanf("%s",temp);
            cd(temp);
        }
        else if(!strcmp(command,"mkdir"))  //创建目录
        {
            scanf("%s",temp);
            mkdir(temp,2);
        }
        else if(!strcmp(command,"touch"))    //创建文件
        {
            scanf("%s",temp);
            cat(temp,1);
        }

        else if(!strcmp(command,"rmdir"))  //删除空目录
        {
            scanf("%s",temp);
            rmdir(temp);
        }
        else if(!strcmp(command,"rm"))     //删除文件或目录，不提示
        {
            scanf("%s",temp);
            del(temp);
        }
```

```
else if(!strcmp(command,"open"))    //打开一个文件
{
    scanf("%s",temp);
    open_file(temp);
}
else if(!strcmp(command,"close"))   //关闭一个文件
{
    scanf("%s",temp);
    close_file(temp);
}
else if(!strcmp(command,"read"))    //读一个文件
{
    scanf("%s",temp);
    read_file(temp);
}
else if(!strcmp(command,"write"))   //写一个文件
{
    scanf("%s",temp);
    write_file(temp);
}
else if(!strcmp(command,"ls"))      //显示当前目录
{
        ls();
}
else if(!strcmp(command,"reset"))    //显示帮助信息
{
        reset_password();
}
else if(!strcmp(command,"help"))    //显示帮助信息
{
        help();
}
else if(!strcmp(command,"chmod")) //修改文件权限
{
    char mod[4];
        scanf("%s",temp);
    scanf("%s",mod);
        chmod(temp,mod);
}
else if(!strcmp(command,"format"))  //格式化硬盘
{
    char tempch;
    printf("Format will erase all the data in the Disk\n");
```

```
            printf("Are you sure?y/n:\n");
            fflush(stdin);
            scanf(" %c",&tempch);
            if(tempch=='Y'||tempch=='y')
            {
                format();
            }
            else
            {
                printf("Format Disk canceled\n");
            }
        }
        else if(!strcmp(command,"ckdisk"))  //检查硬盘
        {
                check_disk();
        }
        else if(!strcmp(command,"quit"))    //退出系统
        {
                break;
        }
        else {
            printf("No this Command,Please check!\n");
            printf("Type help to get more information\n");
            fflush(stdin);  //清空输入缓冲区
            continue;
        }
        getchar();
    }
    return 0;
}
```

3.10.5 附件 5 README

实验三--EXT2文件系统模拟

一、实验目的

1. 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文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。

二、实验内容

1. 定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项。

1. 实验代码

```
#define VOLUME_NAME "EXT2FS"   // 卷名
#define BLOCK_SIZE 512        // 块大小
#define DISK_SIZE 4612    //磁盘总块数

#define DISK_START 0          // 磁盘开始地址
#define SB_SIZE        128                //超级块大小是128

#define GD_SIZE        32                 // 块组描述符大小是32B
#define GDT_START      (0+512)    // 块组描述符起始地址

#define BLOCK_BITMAP (512+512) // 块位图起始地址
#define INODE_BITMAP (1024+512)// inode 位图起始地址

#define INODE_TABLE (1536+512) // 索引节点表起始地址 4*512
#define INODE_SIZE 64         // 每个inode的大小是64B
#define INODE_TABLE_COUNTS     4096 // inode entry 数

#define DATA_BLOCK (263680+512)        // 数据块起始地址 4*512+4096*64
#define DATA_BLOCK_COUNTS      4096    // 数据块数

#define BLOCKS_PER_GROUP       4612 // 每组中的块数

#define USER_NAME "root" //用户名
#define PASSWORD "123456" //密码

struct super_block // 32 B
{
    char sb_volume_name[16]; //文件系统名
```

```
    unsigned short sb_disk_size; //磁盘总大小
    unsigned short sb_blocks_per_group; // 每组中的块数
    unsigned short sb_size_per_block;  // 块大小
    char username[10]; //用户名
    char password[10]; //密码
};

struct group_desc // 32 B
{
    char bg_volume_name[16]; //文件系统名
    unsigned short bg_block_bitmap; //块位图的起始块号
    unsigned short bg_inode_bitmap; //索引结点位图的起始块号
    unsigned short bg_inode_table; //索引结点表的起始块号
    unsigned short bg_free_blocks_count; //本组空闲块的个数
    unsigned short bg_free_inodes_count; //本组空闲索引结点的个数
    unsigned short bg_used_dirs_count; //组中分配给目录的结点数
    char bg_pad[4]; //填充(0xff)
};
struct inode // 64 B
{
    unsigned short i_mode;   //文件类型及访问权限
    unsigned short i_blocks; //文件所占的数据块个数
    unsigned int i_size;    // 文件或目录大小(单位 byte)
    unsigned long i_atime;   //访问时间
    unsigned long i_ctime;   //创建时间
    unsigned long i_mtime;   //修改时间
    unsigned long i_dtime;   //删除时间
    unsigned short i_block[8]; //直接索引方式 指向数据块号
    char i_pad[8];           //填充(0xff)
};
struct dir_entry //16B
{
    unsigned short inode; //索引节点号
    unsigned short rec_len; //目录项长度
    unsigned short name_len; //文件名长度
    char file_type; //文件类型(1 普通文件 2 目录.. )
    char name[9]; //文件名
};

 static unsigned short last_alloc_inode; // 最近分配的节点号 */
static unsigned short last_alloc_block; // 最近分配的数据块号 */
static unsigned short current_dir;   // 当前目录的节点号 */

static unsigned short current_dirlen; // 当前路径长度 */

static short fopen_table[16]; // 文件打开表 */
```

```
static struct super_block sb_block[1];  // 超级块缓冲区
static struct group_desc gdt[1];        // 组描述符缓冲区
static struct inode inode_area[1];  // inode缓冲区
static unsigned char bitbuf[512]={0}; // 位图缓冲区
static unsigned char ibuf[512]={0};
static struct dir_entry dir[32];   // 目录项缓冲区 32*16=512
static char Buffer[512];  // 针对数据块的缓冲区
static char tempbuf[2*1024*1024];       // 文件写入缓冲区
static FILE *fp;        // 虚拟磁盘指针


extern char current_path[256];    // 当前路径名 */

static unsigned long getCurrentTime();//得到当前时间
static void update_super_block(void);   // 更新超级块内容
static void reload_super_block(void);   //加载超级块内容
static void update_group_desc(void);    //更新组描述符内容
static void reload_group_desc(void);    //加载组描述符内容
static void update_inode_entry(unsigned short i); //更新indoe表
static void reload_inode_entry(unsigned short i); //加载inode表
static void update_block_bitmap(void);  //更新块位图
static void reload_block_bitmap(void);  //加载块位图
static void update_inode_bitmap(void);  //更新inode位图
static void reload_inode_bitmap(void);  //加载inode位图
static void update_dir(unsigned short i);//更新目录
static void reload_dir(unsigned short i);//加载目录
static void update_block(unsigned short i);//更新数据块
static void reload_block(unsigned short i);//加载数据库
static int alloc_block(void);//分配数据块
static int get_inode(void); //得到inode节点
static unsigned short reserch_file(char tmp[9],int file_type,unsigned short
*inode_num,unsigned short *block_num,unsigned short *dir_num);//查找文件
static void dir_prepare(unsigned short tmp,unsigned short len,int type);
static void remove_block(unsigned short del_num);//删除数据块
static void remove_inode(unsigned short del_num);//删除inode节点
static unsigned short search_file(unsigned short Ino);//在打开文件表中查找是否已打
开文件
static void sleep(int k);
static void initialize_disk(void);//初始化磁盘
```

2. define部分：定义了一些常量，如磁盘大小、块大小、磁盘起始地址等。
3. super_block：超级块，用于存储文件系统的信息，如文件系统名、磁盘总大小、块大小，用户名和密码。

4. group_desc：块组描述符，用于存储块位图的起始块号、索引结点位图的起始块号、索引结点表的起始块号、本组空闲块的个数、本组空闲索引结点的个数、组中分配给目录的结点数。
5. inode：索引结点，用于存储文件的信息，如文件类型及访问权限、文件所占的数据块个数、文件或目录大小、访问时间、创建时间、修改时间、删除时间、直接索引方式 指向数据块号。
6. dir_entry：目录项，用于存储目录的信息，如索引节点号、目录项长度、文件名长度、文件类型、文件名。
7. 定义了一批全局变量，如最近分配的节点号、最近分配的数据块号、当前目录的节点号、当前路径长度、文件打开表、超级块缓冲区、组描述符缓冲区、inode缓冲区、位图缓冲区、目录项缓冲区、针对数据块的缓冲区、文件写入缓冲区、虚拟磁盘指针、当前路径名。
8. 为了方便从文件中读取数据，定义了很多缓冲区，如超级块缓冲区、组描述符缓冲区、inode缓冲区、位图缓冲区、目录项缓冲区、针对数据块的缓冲区、文件写入缓冲区。

2. 实现底层函数，包括分配数据块等 操作。

1. 文件系统各部分的读写

1. 实验代码

```
static void update_super_block(void) //写超级块
{
    fp=fopen("./Ext2","r+");
    fseek(fp,DISK_START,SEEK_SET);
    fwrite(sb_block,SB_SIZE,1,fp);
    fflush(fp); //立刻将缓冲区的内容输出，保证磁盘内存数据的一致性
}

static void reload_super_block(void) //读超级块
{
    fseek(fp,DISK_START,SEEK_SET);
    fread(sb_block,SB_SIZE,1,fp);//读取内容到超级块缓冲区中
}
```

2. 以超级块为例，通过fopen打开文件，然后通过fseek定位到超级块的位置，再通过fwrite写入超级块缓冲区的内容，最后通过fflush立刻将缓冲区的内容输出，保证磁盘内存数据的一致性。

2. 分配数据块与inode节点

1. 实验代码

```
static int alloc_block(void) // 分配一个数据块,返回数据块号
{
    unsigned short cur=last_alloc_block;
    //printf("cur: %d\n",cur);
    unsigned char con=128; // 1000 0000b
    int flag=0;
    if(gdt[0].bg_free_blocks_count==0)
    {
        printf("There is no block to be alloced!\n");
        return(0);
    }
    reload_block_bitmap();
    cur/=8;
    while(bitbuf[cur]==255)//该字节的8个bit都已有数据
    {
        if(cur==511)cur=0; //最后一个字节也已经满，从头开始寻找
        else cur++;
    }
    while(bitbuf[cur]&con) //在一个字节中找具体的某一个bit
    {
        con=con/2;
        flag++;
    }
    bitbuf[cur]=bitbuf[cur]+con;
    last_alloc_block=cur*8+flag;

    update_block_bitmap();
    gdt[0].bg_free_blocks_count--;
    update_group_desc();
    return last_alloc_block;
}
```

2. 实现思路：记录最近分配的数据块号，然后从这个数据块号开始，从位图循环中寻找空闲的数据块，找到后将该数据块置为1，然后更新位图，最后更新组描述符中的空闲块个数

3. 释放数据块与inode节点

1. 实验代码

```
static void remove_block(unsigned short del_num)
{
    unsigned short tmp;
    tmp=del_num/8;
    reload_block_bitmap();
```

```
    switch(del_num%8) // 更新block位图 将具体的位置为0
    {
        case 0:bitbuf[tmp]=bitbuf[tmp]&127;break;
        case 1:bitbuf[tmp]=bitbuf[tmp]&191;break;
        case 2:bitbuf[tmp]=bitbuf[tmp]&223;break;
        case 3:bitbuf[tmp]=bitbuf[tmp]&239;break;
        case 4:bitbuf[tmp]=bitbuf[tmp]&247;break;
        case 5:bitbuf[tmp]=bitbuf[tmp]&251;break;
        case 6:bitbuf[tmp]=bitbuf[tmp]&253;break;
        case 7:bitbuf[tmp]=bitbuf[tmp]&254;break;
    }
    update_block_bitmap();
    gdt[0].bg_free_blocks_count++;
    update_group_desc();
}
```

2. 实现思路：通过位图找到要释放的数据块，然后将该数据块置为0，最后更新组描述符中的空闲块个数

4. 初始化磁盘和文件系统

1. 初始化磁盘，在当前目录下创建一个名为Ext2的文件，然后将其大小设置为2MB，将各部分内容设置为初始化状态，即格式化。
2. 初始化文件系统：读取Ext2文件，然后将各部分内容读取到缓冲区中。

3. 命令行函数

1. cd命令：切换当前目录，实现思路：通过reserch_file函数查找目录，然后将当前目录的节点号设置为该目录的节点号，最后将当前路径名设置为该目录的路径名。
2. ls命令：显示当前目录下的文件和目录，实现思路：通过reserch_file函数查找目录，然后通过reload_dir函数将目录项读取到目录项缓冲区中，最后通过遍历目录项缓冲区，将目录项的文件名输出
3. mkdir命令：创建目录，实现思路：通过reserch_file函数查找目录，然后通过alloc_block函数分配一个数据块，通过get_inode函数得到一个inode节点，然后将该目录的信息写入到该inode节点中，最后将该目录的信息写入到该数据块中。
4. touch命令：创建文件，实现思路：通过reserch_file函数查找目录，然后通过alloc_block函数分配一个数据块，通过get_inode函数得到一个inode节点，然后将该文件的信息写入到该inode节点中。
5. rm命令：删除文件或目录，实现思路：通过reserch_file函数查找文件或目录，然后通过remove_block函数释放该文件或目录的数据块，通过remove_inode函数释放该文件或目录的inode节点。

6. write命令：写文件，实现思路：先通过循环使用getchar函数从键盘读取字符，然后将读取到的字符写入到文件写入缓冲区中，知道读取到#为止，然后根据文件大小，判断是否需要使用一级索引或二级索引，最后将文件写入缓冲区中的内容写入到文件中。
为了实现读写保护，在调用write函数时，会先读取该文件对应的inode节点，取i_mode的倒数第二位，如果为1，则说明该文件可写，否则不可写。

```
void write_file_111(char tem[9])
{
    fflush(stdin);
    unsigned short flag,i,j,k,size=0,need_blocks;
    int length1,length2,length3;//1级写入，2级写入，3级写入
    length1=length2=length3=0;
    flag=reserch_file(tem,1,&i,&j,&k);
    if (flag){
        reload_inode_entry(i);
        inode_area[0].i_mtime=getCurrentTime();
        update_inode_entry(i);
        reload_group_desc();
        if(search_file(dir[k].inode)){
            reload_inode_entry(dir[k].inode);
            while(1)
            {
                tempbuf[size]=getchar();
                if(tempbuf[size]=='#')
                {
                    tempbuf[size]='\0';
                    break;
                }
                if(size>=gdt->bg_free_blocks_count*512)//判断文件大小是否超过最大
值，超过全部不写入
                {
                    printf("Sorry,the max size of a file is %dKB!\n",gdt-
>bg_free_blocks_count/2);
                    printf("Write failed!\n");
                    return;
                }
                size++;
            }
            if (size<=6*512){
                length1 = strlen(tempbuf);
            }else if (size<=12*512){
                length1 = 6*512;
            }
            if(length1>0){
                need_blocks=length1/512;
```

```
                if(length1%512)
                {
                    need_blocks++;
                }
                int x = 0;
                while (x <= need_blocks)
                {
                    inode_area[0].i_block[x]=alloc_block();
                    reload_block(inode_area[0].i_block[x]);
                    memcpy(Buffer,tempbuf+x*BLOCK_SIZE,BLOCK_SIZE);
                    update_block(inode_area[0].i_block[x]);
                    x++;
                }
                if(inode_area[0].i_blocks>need_blocks)//清空剩下的块
                {
                    while (x <= 6)
                    {
                        x++;
                        remove_block(inode_area[0].i_block[x]);
                    }
                }
            }
            if (size > 6*512 && size <= 262*512){
                length2 = size - 6*512;
            }else if (size > 262*512){
                length2 = 256*512;
            }
            if (length2){//采用一级索引
                need_blocks=length2/512;
                if(length2%512)
                {
                    need_blocks++;
                }
                inode_area[0].i_block[6]=alloc_block();
                reload_block(inode_area[0].i_block[6]);
                char index_1[512];          //一级索引
                memcpy(index_1,Buffer,512);
                int x;
                for (x = 0; x < need_blocks; x++) // x代表写入的块数
                {
                    unsigned short block_index_temp = alloc_block();
                    unsigned short* target = (unsigned short*)(index_1 +
sizeof(unsigned short) * x);
                    memcpy(target, &block_index_temp, sizeof(unsigned short));
                    reload_block(block_index_temp);
                    memcpy(Buffer,tempbuf+(x+6)*BLOCK_SIZE,BLOCK_SIZE);
```

```
                    update_block(block_index_temp);
                }
                if(inode_area[0].i_blocks-6>need_blocks)//清空剩下的块
                {
                    while(x<=255){
                        unsigned short temp;
                        memcpy(&temp, &index_1[x * 2], sizeof(unsigned short));
                        remove_block(temp);
                        x++;
                    }
                }
                memcpy(Buffer,index_1,512);
                update_block(inode_area[0].i_block[6]);//更新一级索引
            }
            if (size > 262*512 ){
                length3 = size - 262*512;
            }
            if (length3 >0){
                need_blocks=length3/512;
                if(length3%512)
                {
                    need_blocks++;
                }
                inode_area[0].i_block[7]=alloc_block();
                unsigned short num_1 = need_blocks/256;
                unsigned short num_2 = need_blocks%256;
                reload_block(inode_area[0].i_block[7]);
                char index_2[512];          //二级索引
                memcpy(index_2,Buffer,512);
                int x;
                for (x = 0; x < num_1; x++) // x代表写入的块数
                {
                    unsigned short block_index_temp = alloc_block();
                    unsigned short* target = (unsigned short*)(index_2 +
sizeof(unsigned short) * x);
                    memcpy(target,&block_index_temp,sizeof(unsigned short));
                    reload_block(block_index_temp);
                    char index_1[512];          //一级索引
                    memcpy(index_1,Buffer,512);
                    int y;
                    for (y = 0; y < 256; y++) // y代表写入的块数
                    {
                        unsigned short block_index_temp = alloc_block();
                        unsigned short* target = (unsigned short*)(index_1 +
sizeof(unsigned short) * y);
                        memcpy(target,&block_index_temp,sizeof(unsigned
```

```
short));
                        reload_block(block_index_temp);
                        memcpy(Buffer,tempbuf+
(x*256+y+262)*BLOCK_SIZE,BLOCK_SIZE);
                        update_block(block_index_temp);
                    }
                    memcpy(Buffer,index_1,512);
                    update_block(block_index_temp);
                }
                memcpy(Buffer,index_2,512);
                update_block(inode_area[0].i_block[7]);//更新二级索引
            }
            inode_area[0].i_size=size;
            inode_area[0].i_blocks= size%512==0?size/512:size/512+1;
            update_inode_entry(dir[k].inode);
        }else {
            printf("The file %s has not opened!\n",tem);
        }
    }else{
        printf("The file %s does not exist!\n",tem);
    }
}
```

7. read命令：读文件，实现思路：先通过reserch_file函数查找文件，然后根据文件大小，判断是否需要使用一级索引或二级索引，然后按数据块按字符输出。
为了实行读取保护，在调用读函数后，会先取该文件的inode节点，取i_mode的倒数第三位，只有在为1的时候读出

```
void read_file(char tmp[9])
{
    unsigned short flag,i,j,k,t;
    unsigned short b1,b2,b3;
    b1=b2=b3=0;
    flag=reserch_file(tmp,1,&i,&j,&k);
    if(flag)
    {
        if(search_file(dir[k].inode)) //读文件的前提是该文件已经打开
        {
            reload_inode_entry(dir[k].inode);
            //判断是否有读的权限
            if(!(inode_area[0].i_mode&4)) // i_mode:111b:读,写,执行
            {
                printf("The file %s can not be read!\n",tmp);
                return;
            }
```

```
            //输出直接索引的内容
            if (inode_area[0].i_blocks<=6){
                b1=inode_area[0].i_blocks;
            }else if (inode_area[0].i_blocks>6){
                b1=6;
            }
            if (b1>0){
                for(flag=0;flag<b1;flag++)
                {
                    reload_block(inode_area[0].i_block[flag]);
                    for(t=0;t<inode_area[0].i_size-flag*512;++t)
                    {
                        printf("%c",Buffer[t]);
                    }
                }
            }

            //输出一级索引的内容
            if(inode_area[0].i_blocks>6&&inode_area[0].i_blocks<=262)
            {
                b2=inode_area[0].i_blocks-6;
            }else if (inode_area[0].i_blocks>262){
                b2=256;
            }
            if (b2>0){
                reload_block(inode_area[0].i_block[6]);
                char index_1[512];
                memcpy(index_1,Buffer,512);
                for (flag = 0; flag < b2; flag++)
                {
                    unsigned short block_num_1;
                    memcpy(&block_num_1, &index_1[flag * 2], sizeof(unsigned
short));
                    reload_block(block_num_1);
                    for (t = 0; t < inode_area[0].i_size - (flag + 6) * 512;
++t)
                    {
                        printf("%c", Buffer[t]);
                    }
                }
            }

            //输出二级索引的内容
            if(inode_area[0].i_blocks>262&&inode_area[0].i_blocks<=65818)
            {
                b3=inode_area[0].i_blocks-262;
```

```
            }

            if (b3>0){
                reload_block(inode_area[0].i_block[7]);
                char index_2[512];
                memcpy(index_2,Buffer,512);
                for (flag = 0; flag < b3; flag++)
                {
                    unsigned short block_num_2;
                    memcpy(&block_num_2, &index_2[flag * 2], sizeof(unsigned
short));
                    reload_block(block_num_2);
                    char index_3[512];
                    memcpy(index_3,Buffer,512);
                    for (t = 0; t < 256; t++)
                    {
                        unsigned short block_num_3;
                        memcpy(&block_num_3, &index_3[t * 2], sizeof(unsigned
short));
                        reload_block(block_num_3);
                        for (int m = 0; m < inode_area[0].i_size - (flag + 262)
* 512; ++m)
                        {
                            printf("%c", Buffer[m]);
                        }
                    }
                }
            }


            if(inode_area[0].i_blocks==0)
            {
                printf("The file %s is empty!\n",tmp);
            }
            else
            {
                printf("\n");
            }
        }
        else
        {
            printf("The file %s has not been opened!\n",tmp);
        }
    }
    else printf("The file %s not exists!\n",tmp);
}
```

8. open命令：打开文件，实现思路：先通过reserch_file函数查找文件，然后将该文件的inode节点号写入到文件打开表中
9. close命令：关闭文件，实现思路：先通过reserch_file函数查找文件，然后将该文件的inode节点号从文件打开表中删除
10. chmod命令：修改文件权限，实现思路：通过 `i_mode & 0x0106 | 0babc` 的方式修改文件权限.
11. help命令：显示帮助信息，实现思路：printf输出帮助信息
12. exit命令：退出文件系统，实现思路：通过exit(0)退出文件系统
13. format命令：格式化文件系统，实现思路：通过initialize_disk函数初始化磁盘，然后将各部分内容设置为初始化状态，即格式化

4. 完成 shell 的设计

1. 模仿Ubuntu的shell，实现思路：通过循环，不断读取用户输入的命令，然后通过strcmp函数判断用户输入的命令，然后调用相应的函数。

```
while(flag)
{

    printf("%s$ ",current_path);
    scanf("%s",command);
    if(!strcmp(command,"cd")) //进入当前目录下
    {
        scanf("%s",temp);
        cd(temp);
    }
    else if(!strcmp(command,"mkdir"))  //创建目录
    {
        scanf("%s",temp);
        mkdir(temp,2);
    }
    else if(!strcmp(command,"touch"))    //创建文件
    {
        scanf("%s",temp);
        cat(temp,1);
    }

    else if(!strcmp(command,"rmdir"))  //删除空目录
    {
        scanf("%s",temp);
        rmdir(temp);
    }
    else if(!strcmp(command,"rm"))     //删除文件或目录，不提示
```

```
    {
        scanf("%s",temp);
        del(temp);
    }
    else if(!strcmp(command,"open"))    //打开一个文件
    {
        scanf("%s",temp);
        open_file(temp);
    }
    else if(!strcmp(command,"close"))   //关闭一个文件
    {
        scanf("%s",temp);
        close_file(temp);
    }
    else if(!strcmp(command,"read"))    //读一个文件
    {
        scanf("%s",temp);
        read_file(temp);
    }
    else if(!strcmp(command,"write"))   //写一个文件
    {
        scanf("%s",temp);
        write_file(temp);
    }
    else if(!strcmp(command,"ls"))      //显示当前目录
    {
        ls();
    }
    else if(!strcmp(command,"reset"))    //显示帮助信息
    {
        reset_password();
    }
    else if(!strcmp(command,"help"))    //显示帮助信息
    {
        help();
    }
    else if(!strcmp(command,"chmod")) //修改文件权限
    {
        char mod[4];
        scanf("%s",temp);
        scanf("%s",mod);
        chmod(temp,mod);
    }
    else if(!strcmp(command,"format"))  //格式化硬盘
    {
        char tempch;
```

```
            printf("Format will erase all the data in the Disk\n");
            printf("Are you sure?y/n:\n");
            fflush(stdin);
            scanf(" %c",&tempch);
            if(tempch=='Y'||tempch=='y')
            {
                format();
            }
            else
            {
                    printf("Format Disk canceled\n");
            }
        }
        else if(!strcmp(command,"ckdisk"))  //检查硬盘
        {
            check_disk();
        }
        else if(!strcmp(command,"quit"))    //退出系统
        {
            break;
        }
        else {
            printf("No this Command,Please check!\n");
            printf("Type help to get more information\n");
            fflush(stdin);  //清空输入缓冲区
            continue;
        }
        getchar();
    }
```

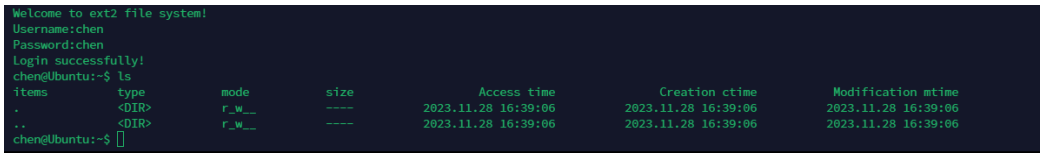
三、实验结果

1. 初始化磁盘

```
chenshi@Ubuntu:~/OSlab/lab3$ ./test
The File system does not exist!
Creating the ext2 file system
Please wait ...
Please set your username and password (less than 10 characters)
username: chen
password: chen
The ext2 file system has been installed!
volume name       : EXT2FS
disk size         : 4612(blocks)
blocks per group  : 4612(blocks)
ext2 file size    : 2306(kb)
block size        : 512(kb)
```

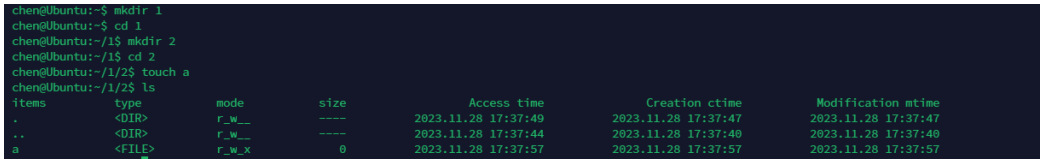
初始化磁盘后，设置了Username和Password，然后将其写入到超级块中，然后将各部分内容写入到磁盘中。

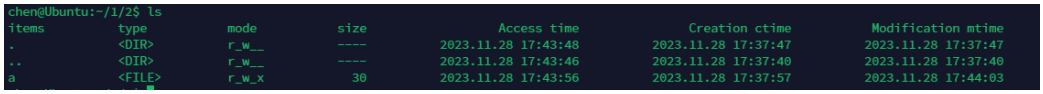
2. 登录



使用正确的用户名和密码登录，登录成功。使用ls命令查看当前目录下的文件和目录，当前目录下有两个目录，分别是`.`和`..`，`.`表示当前目录，`..`表示上一级目录。
默认权限是`r_w__`,即可读可写。其中可读指可以cd进入该目录，可写指可以在该目录下创建文件或目录。

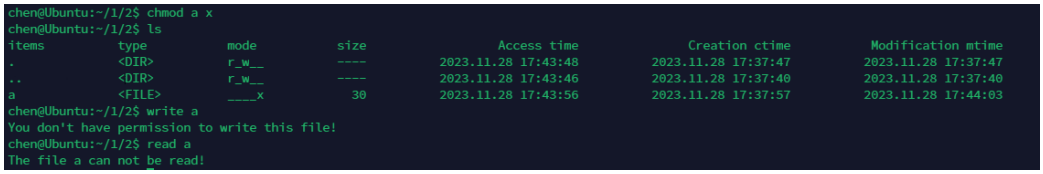
3. 创建目录和文件，写文件





由于a是无后缀的文件，所以默认为有可读可写可执行权限，写入后。使用ls命令查看当前目录下的文件和目录，可以看到a文件的大小为30Byte，权限为`rwx__`，Access Time为Open的时间，Modify Time为Write的时间，Create Time为Create的时间。

4. 验证读写保护



通过chmod命令修改a文件的权限为`____x`，即不可读不可写可执行，然后使用read命令读取a文件，可以看到读取失败，因为没有读的权限。使用write命令写入a文件，可以看到写入失败。

5. 验证删除多级目录

```
chen@Ubuntu:~$ ls
items          type          mode          size              Access time              Creation ctime           Modification mtime
.              <DIR>         r_w__         ----          2023.11.28 16:39:06      2023.11.28 16:39:06      2023.11.28 16:39:06
..             <DIR>         r_w__         ----          2023.11.28 16:39:06      2023.11.28 16:39:06      2023.11.28 16:39:06
1              <DIR>         r_w__           48          2023.11.28 17:43:46      2023.11.28 17:37:40      2023.11.28 17:37:40
chen@Ubuntu:~$ rmdir 1
chen@Ubuntu:~$ ls
items          type          mode          size              Access time              Creation ctime           Modification mtime
.              <DIR>         r_w__         ----          2023.11.28 16:39:06      2023.11.28 16:39:06      2023.11.28 16:39:06
..             <DIR>         r_w__         ----          2023.11.28 16:39:06      2023.11.28 16:39:06      2023.11.28 16:39:06
chen@Ubuntu:~$
```

文件夹1下有文件夹为2，文件夹2下有文件a，然后使用rm命令删除文件夹1，可以看到删除成功，验证了删除多级目录的功能。

```
chen@Ubuntu:~$ ckdisk
volume name        : EXT2FS
disk size          : 4612(blocks)
blocks per group   : 4612(blocks)
ext2 file size     : 2306(kb)
block size         : 512(kb)
```

使用ckdisk命令检查磁盘，可以看到磁盘信息与初始化磁盘时的信息一致，验证了删除多级目录后，磁盘信息得到了正确的更新，对应的inode节点和数据块也被释放。