

西安交通大学

操作系统专题实验报告

班级： 计算机 2101

学号： 2213311111

姓名： 邱子杰

2023 年 12 月 7 日

目 录

1 openEuler 系统环境实验.....	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验思想（或流程图）	3
1.4 实验步骤	5
1.5 程序运行初值及运行结果分析.....	6
1.6 实验总结	13
1.7 附件	14
2 进程通信与内存管理.....	32
2.1 实验目的	32
2.2 实验内容	32
2.3 实验步骤	33
2.4 程序运行初值及运行结果分析.....	36
2.5 实验总结	42
2.6 附件	43
3 模拟文件系统	66
3.1 实验目的	66
3.2 实验内容	66
3.3 实验步骤	66
3.4 运行结果与分析.....	67
3.5 遇到的问题与解决方法.....	76
3.6 实验总结	76
3.7 附件	76

1 openEuler 系统环境实验

1.1 实验目的

1.1.1 进程相关编程实验

- 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 线程相关编程实验

- 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。

1.1.2 自旋锁实验

- 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。

1.2 实验内容

进程实验

- (1) 熟悉操作命令、编辑、编译、运行程序。完成图 1-1 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。

(2) 扩展下图的程序：

```
#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;
}
```

- 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。
- 在return前增加对全局变量的操作并输出结果，观察并解释。
- 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。

线程实验

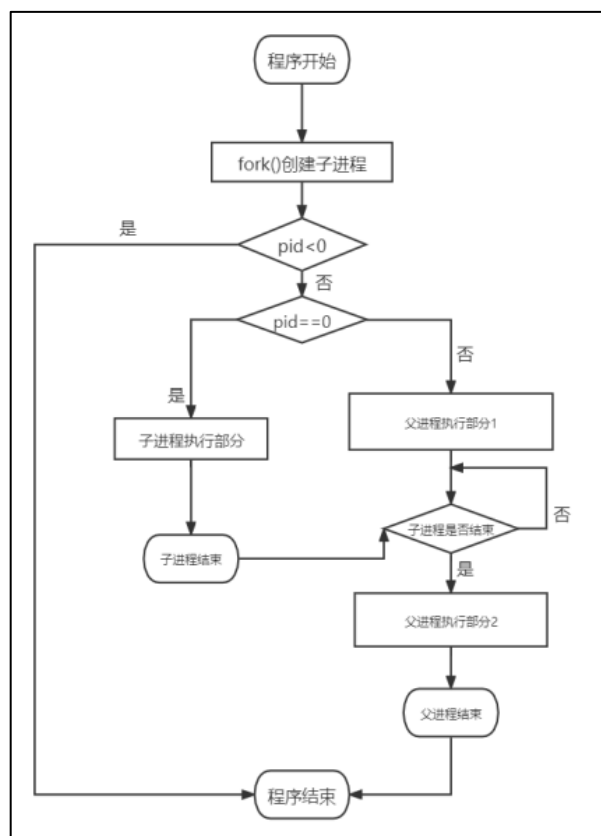
- 在进程中给一变量赋初值并成功创建两个线程。
- 在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）并输出结果。
- 多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步。
- 将任务一中第一个实验调用 `system` 函数和调用 `exec` 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

自旋锁实验

- 在进程中给一变量赋初值并成功创建两个线程。
- 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。
- 使用自旋锁实现互斥和同步。

1.3 实验思想（或流程图）

1) 进程实验



进程程序流程图

- 1) 进程：进程是计算机科学中的一个重要概念，它是操作系统中的基本执行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制。
- 2) PID：PID 是进程标识符（Process Identifier）的缩写，它是用来唯一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程。
- 3) fork() 函数：fork() 是一个在类 Unix 操作系统中常见的系统调用，用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进程，它与父进程共享很多资源，但也有一些独立的属性。fork() 被用于实现多进程编程，常见于操作系统和并发编程中。函数返回一个整数，如果返回值为负数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进程中。如果返回值大于 0，表示当前正在

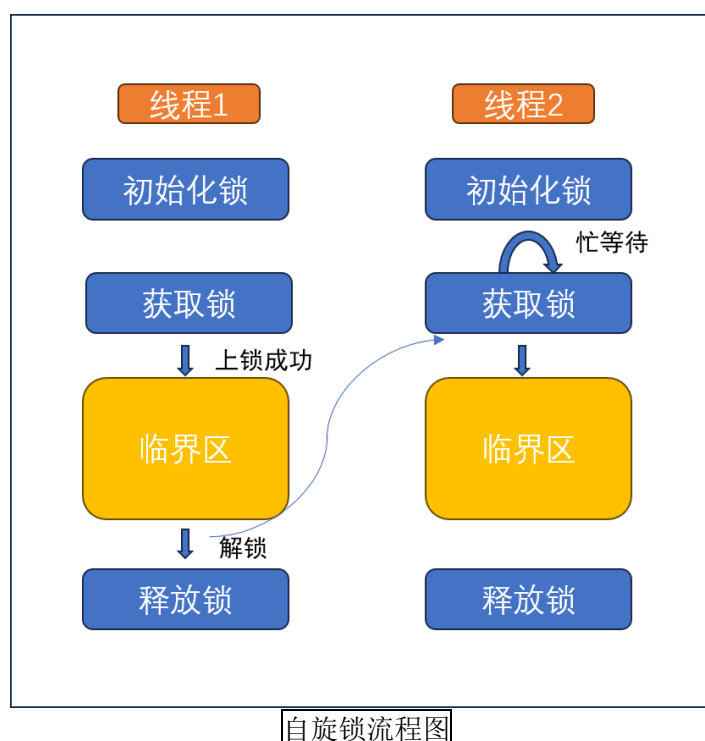
执行的代码是在父进程中，返回值是子进程的 PID 。调用 `fork()` 函数时，操作系统会创建一个新的进程，该进程是调用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

2) 线程实验

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。

互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（**Mutex**）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

3) 自旋锁实验



自旋锁是一种基于忙等待（**busy waiting**）的同步机制，用于在线程竞争共享资源时，不断尝试获取锁，而不是阻塞等待。它的工作原理可以简单地概括为以下几个步骤：

- 1) 初始化锁：自旋锁的开始是一个共享的标志变量（flag），最初为未锁定状态（0）。这个标志变量用于表示资源是否已被其他线程占用。
- 2) 获取锁：当一个线程尝试获取锁时，它会循环检查标志变量的状态。如果发现标志变量是未锁定状态（0），那么该线程将通过原子操作将标志变量设置为锁定状态（1），从而成功获取锁。如果标志变量已经是锁定状态，线程会一直在循环中等待，直到标志变量变为未锁定状态为止。
- 3) 释放锁：当持有锁的线程完成对共享资源的操作后，它会通过原子操作将标志变量设置回未锁定状态（0），从而释放锁，允许其他等待的线程尝试获取锁。

1.4 实验步骤

1.4.1 进程有关编程实验

- 1) 编写并多次运行图中代码
- 2) 删去代码中的 `wait()` 函数并多次运行程序，分析运行结果。
- 3) 修改代码，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 3.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。

1.4.2 线程有关编程实验

- 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) 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

1.4.3 自旋锁实验

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

1.5 程序运行初值及运行结果分析

■ 进程实验

1) 观察进程调度

结果分析:

- 1) 父子进程执行顺序没有关系, 但父子进程中输出 `pid` 和 `pid1` 的顺序不会颠倒, 子进程 `pid1` 为父进程 `pid1` 加 1, `Fork` 里父进程返回进程号, 子进程返回 0;
- 2) 在去掉 `wait()` 后, 同样也是可能 `parent` 先执行, 又可能 `child` 先执行。

```
[root@kp-test01 1]# ./1-1
parent: pid = 2854
child: pid = 0
parent: pid1 = 2853
child: pid1 = 2854
[root@kp-test01 1]# ./1-1
parent: pid = 2856
parent: pid1 = 2855
child: pid = 0
child: pid1 = 2856
[root@kp-test01 1]# ./1-1
```

有 `wait()` 函数执行顺序

```
[root@kp-test01 1]# ./1-1
child: pid = 0
parent: pid = 2873
child: pid1 = 2873
parent: pid1 = 2872
[root@kp-test01 1]# ./1-1
parent: pid = 2875
child: pid = 0
parent: pid1 = 2874
child: pid1 = 2875
[root@kp-test01 1]# ./1-1
parent: pid = 2877
child: pid = 0
parent: pid1 = 2876
child: pid1 = 2877
```

无 `wait()` 函数执行顺序

原因解释:

- 1) `fork` 创建子进程后, 父子进程并行执行, 两者执行顺序由 `cpu` 调度决定, 所以二者执行顺序不固定。
- 2) 对于子进程来说, `fork()` 后返回的 `pid` 为 0, `getpid` 返回当前进程 (调用这一函数的进程, 子进程的 `pid`) 所以父进程 `pid` 与子进程的 `pid1` 一样。
- 3) `wait()` 的作用是让父进程在子进程结束后继续执行, 等待挂起, 防止僵尸进程的出现。仍会出现 `parent` 先执行, 或 `child` 先执行。

2) 观察进程调度中全局变量的改变

2023 年 12 月 7 日

结果分析:

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, 在 return 前增加对全局变量的操作并输出结果: 定义全局变量 global, 初值 100. 在子进程加 2, 父进程减 2。并返回全局变量地址, 在 return 前做 global 平方操作。发现二者 global 地址一样, 但二者 global 改变是独立进行的。

```
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3139
child: pid = 0
parent: pid1 = 3138
child: pid1 = 3139
global = 10404
global = 9604
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3141
child: pid = 0
parent: pid1 = 3140
child: pid1 = 3141
global = 10404
global = 9604
```

父子进程全局变量对比

原因解释:

子进程“继承”父进程的变量, 其地址总是一样的, 因为在 fork 时整个虚拟地址空间被复制, 但是虚拟地址空间所对应的物理内存却没有复制。所以对变量的操作是独立的。

3) 调用 system 函数

结果分析:

发现调用 systemcall 后 pid 改变, 说明调用该函数创建了一个进程。

```
[root@kp-test01 1]# ./1-3
parent: pid = 3320
child: pid1 = 3320
parent: pid1 = 3319
system_call pid = 3321
[root@kp-test01 1]# ./1-3
parent: pid = 3323
child: pid1 = 3323
parent: pid1 = 3322
system_call pid = 3324
[root@kp-test01 1]# ./1-3
parent: pid = 3326
child: pid1 = 3326
parent: pid1 = 3325
system call pid = 3327
```

System 函数调用

原因解释:

可以发现 `system()` 函数只是简单地执行我们给其地指令，并没有影响程序其他部分地执行。究其原因，是因为 `system` 函数会执行参数要求的命令创建新的进程所以 `pid` 改变。

4) 调用 `exec` 族函数

结果分析:

发现调用 `systemcall` 后 `pid` 未改变，与 `child` 的 `pid` 一样。

```
[root@kp-test01 1]# ./1-4
parent: pid = 3368
child: pid1 = 3368
parent: pid1 = 3367
system_call pid = 3368
[root@kp-test01 1]# ./1-4
parent: pid = 3370
child: pid1 = 3370
parent: pid1 = 3369
system_call pid = 3370
[root@kp-test01 1]# ./1-4
parent: pid = 3372
child: pid1 = 3372
parent: pid1 = 3371
system call pid = 3372
```

exec 族函数调用

原因解释:

子进程仅仅执行了 `exec1()` 函数, 并没有执行原来的内容, 调用 `exec` 函数并不创建新进程, 所以前后进程的 ID 并没有改变, 为当一个进程调用 `exec1` 函数时, 该进程的程序会完全由新 `excel` 程序代换。

■ 线程实验

1) 在进程中创建两个线程

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。创建变量 `global` (初值为 0) 两个线程分别执行加 100 和减 100 的操作。

```
root@kp-test01 1]# gcc -o thread thread.c -lpthread
root@kp-test01 1]# ./thread
thread1 success create
thread2 success create
thread1 global = -2530200
thread2 global = -264800
global = -264800
root@kp-test01 1]# ./thread
thread1 success create
thread2 success create
thread1 global = -2375600
thread2 global = -101700
global = -101700
```

结果解释:

可以看出二者是并发执行。每次值都一样因为线程的执行并发, 不能保证执行了相同的加和减的操作。

2) 控制互斥和同步

使用 `pthread_mutex_` 函数对 `global` 变量进行互斥访问。使线程 1 先执行, 线程 2 后执行。有图 1 可知 `thread2` 创建后任在执行 `thread1` 的操作

```
thread1 success create
thread1 global = -100
thread1 global = -200
thread1 global = -300
thread1 global = -400
thread1 global = -500
thread1 global = -600
thread2 success create
thread1 global = -700
thread1 global = -800
thread1 global = -900
thread1 global = -1000
thread1 global = -1100
thread1 global = -1200
thread1 global = -1300
thread1 global = -1400
thread1 global = -1500
thread1 global = -1600
thread1 global = -1700
thread1 global = -1800
thread1 global = -1900
thread1 global = -2000
thread1 global = -2100
thread1 global = -2200
thread1 global = -2300
thread1 global = -2400
thread1 global = -2500
thread1 global = -2600
```

thread2 创建后仍在执行 thread1 的操作

```
thread2 global = -1400
thread2 global = -1300
thread2 global = -1200
thread2 global = -1100
thread2 global = -1000
thread2 global = -900
thread2 global = -800
thread2 global = -700
thread2 global = -600
thread2 global = -500
thread2 global = -400
thread2 global = -300
thread2 global = -200
thread2 global = -100
thread2 global = 0
[root@kp-test01 1]#
```

最后结果为 0

3) 调用系统函数和线程函数的比较

调用 `system` 和 `exec` 函数使用 `syscall(SYS_gettid)` 和 `pthread_self()` 输出真实 `tid` 和 `tid`, 使用 `getpid()` 输出 `pid`。

```
[root@kp-test01 1]# ./pth_sys
thread1 success create
thread2 success create
thread1 global = -1000000
thread1 getpid: 2928 , the tid=281458690224608
thread1 getpid: 2931 , the tid=281459530084096,syscall_pid=2931
thread1 return
thread2 global = -500000
thread2 getpid: 2928 , the tid=281458681770464
thread2 getpid: 2932 , the tid=281464635338496,syscall_pid=2932
thread2 return
global = -500000 [root@kp-test01 1]#
```

调用 system 函数

```
[root@kp-test01 1]# ./pth_exec
thread1 success create
thread2 success create
thread1 getpid: 2864 , the tid=281461631742432global = 5000
thread1 getpid: 2864 , the tid=281460498050816,syscall_pid=2864
thread1 return
```

调用 exec 函数

结果分析:

- 1) 线程 1、2 的 getpid 相同, 线程编号不同。调用 system 时创建全新的进程, 编号均不同。

每个进程有一个 pid (进程 ID), 获取函数: getpid(), 系统内唯一, 除了和自己的主线程一样。

- 2) 指行 exec 函数后, 原来的进程被调用的内容取代 thread2 的 syscall 不会再进行。

所以调用的 syscall 产生了输出, 此时 syscall 为主进程所以 syscall(SYS_gettid) 与 pid 一样。

原因解释:

每个线程有一个 tid (线程 ID), 获取函数: pthread_self(), 所在进程内唯一, 有可能两个进程中都有同样一个 tid。

每个线程有一个 pid, 获取函数: syscall(SYS_gettid), 系统内唯一, 除了主线程和自己的进程一样, 其他子线程都是唯一的。在 linux 下每一个进程都有一个进程 id, 类型 pid_t, 可以由 getpid() 获取。

POSIX 线程也有线程 id, 类型 pthread_t, 可以由 pthread_self() 获取, 线程 id 由线程库维护。但是各个进程独立, 所以会有不同进程中线程号相同的情况。

■ 自旋锁实验

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

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

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

void spinlock_unlock(spinlock_t *lock)
{
    __sync_lock_release(&lock->flag);
}
```

自旋锁相关代码

定义了一个 spinlock_t 结构体，用于表示自旋锁。spinlock_init 函数用于初始化自旋锁，spinlock_lock 函数用于获取自旋锁，spinlock_unlock 函数用于释放自旋锁。

在线程函数 thread_function 中，通过调用 spinlock_lock 和 spinlock_unlock 函数来保护对共享变量 shared_value 的访问。每个线程循环执行 5000 次，每次获取自旋锁后将共享变量加 1，然后释放自旋锁。

```
[root@kp-test01 1]# ./spinlock
initial: 0
thread1 success create
thread2 success create
final: 10000
[root@kp-test01 1]# █
```

自旋锁实验结果

1.6 实验总结

所有要求基本完成，在方法上有一定创新（后面部分还是借助了互联网）

遇见问题如下：

1. Wait 函数输出杂乱无章	2. System 函数使用
3. exec 函数使用	4. 无法编译 pthread 多线程

1.6.1 实验中的解决过程

1. 具体解决方法

问题一：多次实验发现规律

父子进程执行顺序没有关系，但父子进程中输出 pid 和 pidl 的顺序不会颠倒；子进程 pidl 为父进程 pidl 加 1。Fork 里父进程返回进程号，子进程返回 0；

若将 wait() 函数添加到函数首位，则父进程 wait() 函数后的部分一定是在子进程执行完后才执行，可见 wait() 函数的作用是**等待子进程执行完毕**。

问题 2 与 3 上网寻求解决办法

函数参考教程：[Linux 系统学习——exec 族函数、system 函数、popen 函数学习 exec 跟 system popen 区别-CSDN 博客](#)

问题 4 通过编译的时候添加命令 -lpthread 可以解决

函数参考教程：<https://blog.csdn.net/jiangxinyu/article/details/7778864>

1.6.2 实验收获

熟悉了云服务器的购买和应用。了解了 fork(), system(), execl() 等函数的使用，对父子进程之间的关系有了更深刻的认识。

学习了多线程库，明白各线程对进程的资源共享，也观察到了线程对进程资源的竞争以及其对程序结果正确性的影响，通过加锁来实现资源的保护。理解了“自旋”这一概念，即等待获取锁的线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。

1.7 附件

1.7.1 附件 1 程序

程序 1：创建进程

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

int main()
{
    pid_t pid, pid1; //
    pid = fork(); // 创建一个子进程
    if (pid < 0)
    { // error occurred
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    { // child process
        pid1 = getpid();
        printf("child: pid = %d", pid);
        printf("child: pid1 = %d", pid1);
    }
    else
    { // parent process
        pid1 = getpid();
        printf("parent: pid = %d", pid);
        printf("parent: pid1 = %d", pid1);
        wait(NULL);
    }
    return 0;
}
```

程序 2：全局变量

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int global = 100;
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();
    global += 2;
    printf("global = %d\n", global);
    printf("global address = %p\n", &global); // share address
    printf("child: pid = %d\n", pid);
    printf("child: pid1 = %d\n", pid1);
}
else
{ // parent process
    pid1 = getpid();
    global -= 2;
    printf("global = %d\n", global);
    printf("global address = %p\n", &global); // share address
    printf("parent: pid = %d\n", pid);
    printf("parent: pid1 = %d\n", pid1);
    wait(NULL);
}

printf("global = %d\n", global * global); // global^2
return 0;
}
```

程序 3: system 函数

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.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: pid1 = %d\n", pid1);
        system("./system_call"); // call function
    }
    else
    { // parent process
        pid1 = getpid();
        printf("parent: pid = %d\n", pid);
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }
    return 0;
}
```

程序 4: exec 函数

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.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: pid1 = %d\n", pid1);
        execl("/root/code/1/system_cal", "sytem_call", NULL); // call
function
    }
    else
    { // parent process
        pid1 = getpid();
        printf("parent: pid = %d\n", pid);
        printf("parent: pid1 = %d\n", pid1);
        wait(NULL);
    }
    return 0;
}
```

程序 5: system_call 函数

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

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

程序 6: 线程创建

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
int global = 0;
void *thread2_func()
{
    int i;
    for (i = 0; i < 100000; i++)
```

```
{
    global = global + 100;
}
printf("thread2 global = %d\n", global);
}

void *thread1_func()
{
    int i;
    for (i = 0; i < 100000; i++)
    {
        global = global - 100;
    }
    printf("thread1 global = %d\n", global);
}

int main()
{
    int status;
    pthread_t tid_one, tid_two;
    // Create Thread 1
    status = pthread_create(&tid_one, NULL, thread1_func, NULL);
    if (status != 0)
    { // error occurred
        printf("thread1 default = %d\n", status);
        return 1;
    }
    printf("thread1 success create\n");

    // Create Thread 2
    status = pthread_create(&tid_two, NULL, thread2_func, NULL);
    if (status != 0)
    { // error occurred
        printf("thread1 default = %d\n", status);
        return 1;
    }
    printf("thread2 success create\n");

    //wait
    pthread_join(tid_one, NULL);
    pthread_join(tid_two, NULL);
    printf("global = %d\n", global);
    return 0;
}
```

程序 7: 互斥控制

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
// Declare a global variable
int global = 0;
// Initialize a mutex lock
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
// Function for thread 2
```

```
void *thread2_func()
{
    // Lock the mutex to ensure mutual exclusion
    pthread_mutex_lock(&mutex);
    // Increment the global variable in a loop
    int i;
    for (i = 0; i < 5000; i++)
    {
        global = global + 100;
        printf("thread2 global = %d\n", global);
    }
    // Unlock the mutex
    pthread_mutex_unlock(&mutex);
}
// Function for thread 1
void *thread1_func()
{
    // Lock the mutex to ensure mutual exclusion
    pthread_mutex_lock(&mutex);
    // Decrement the global variable in a loop
    int i;
    for (i = 0; i < 5000; i++)
    {
        global = global - 100;
        printf("thread1 global = %d\n", global);
    }
    // Unlock the mutex
    pthread_mutex_unlock(&mutex);
}
// Main function
int main()
{
    int status;
    pthread_t tid_one, tid_two;
    // Create thread 1
    status = pthread_create(&tid_one, NULL, thread1_func, NULL);
    if (status != 0)
    {
        printf("thread1 default = %d\n", status);
        return 1;
    }
    printf("thread1 success create\n");
    // Create thread 2
    status = pthread_create(&tid_two, NULL, thread2_func, NULL);
    if (status != 0)
    {
        printf("thread1 default = %d\n", status);
        return 1;
    }
    printf("thread2 success create\n");
    // Wait for thread 1 to finish
    pthread_join(tid_one, NULL);
    // Wait for thread 2 to finish
    pthread_join(tid_two, NULL);
    // Destroy the mutex
    pthread_mutex_destroy(&mutex);
}
```

```
    return 0;
}
```

程序 8: 线程调用 system

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
int global = 0; // Declare a global variable
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Initialize a mutex lock
// Function for thread 1
void *thread1_func(void *arg)
{
    pthread_mutex_lock(&mutex); // Lock the mutex to ensure mutual exclusion
    // Decrement the global variable in a loop
    int i;
    for (i = 0; i < 5000; i++)
    {
        global = global - 200;
    }
    printf("thread1 global = %d\n", global);
    // Get thread's process id and thread id and execute a system call
    char num = '1';
    char command[100];
    printf("thread1 getpid: %d , the tid=%ld", getpid(), pthread_self());
    sprintf(command, "./pthread_system_call.out %c", num);
    system(command);
    pthread_mutex_unlock(&mutex); // Unlock the mutex
}

// Function for thread 2
void *thread2_func(void *arg)
{
    pthread_mutex_lock(&mutex); // Lock the mutex to ensure mutual exclusion
    // Increment the global variable in a loop
    int i;
    for (i = 0; i < 5000; i++)
    {
        global = global + 100;
    }
    printf("thread2 global = %d\n", global);
    // Get thread's process id and thread id and execute a system call
    char num = '2';
    char command[100];
    printf("thread2 getpid: %d , the tid=%ld", getpid(), pthread_self());
    sprintf(command, "./pthread_system_call.out %c ", num);
    system(command);
    pthread_mutex_unlock(&mutex); // Unlock the mutex
}

int main()
{
    int status;
    pthread_t tid_one, tid_two;
```

```
// Create thread 1
status = pthread_create(&tid_one, NULL, thread1_func, NULL);
if (status != 0)
{
    printf("thread1 default = %d\n", status);
    return 1;
}
printf("thread1 success create\n");
// Create thread 2
status = pthread_create(&tid_two, NULL, thread2_func, NULL);
if (status != 0)
{
    printf("thread2 default = %d\n", status);
    return 1;
}
printf("thread2 success create\n");
// Wait for thread 1 to finish
pthread_join(tid_one, NULL);
// Wait for thread 2 to finish
pthread_join(tid_two, NULL);
// Destroy the mutex
pthread_mutex_destroy(&mutex);
printf("global = %d ", global); // Print the value of global variable
return 0;
}
```

程序 9：线程调用 exec

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
int global = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread1_func(void* arg)
{
    pthread_mutex_lock(&mutex);
    int i;
    for (i = 0; i < 5000; i++)
    {
        global++;
    }
    printf("thread1 getpid: %d , the tid=%ld", getpid(), pthread_self());
    printf("global = %d\n", global);
    execl("/root/code/1/pthread_system_call", "pthread_system_call",
"1", NULL);
    pthread_mutex_unlock(&mutex);
}

void *thread2_func(void* arg)
{
    pthread_mutex_lock(&mutex);
    int i;
    for (i = 0; i < 5000; i++)
    {
```

```

        global = global -1;
    }
    printf("thread2 getpid: %d , the tid=%ld", getpid(), pthread_self());
    printf("global = %d\n", global);
    execl("/root/code/1/pthread_system_call", "pthread_system_call",
"2", NULL);
    pthread_mutex_unlock(&mutex);
}
int main()
{
    int status;
    pthread_t tid_one, tid_two;
    // thread 1
    status = pthread_create(&tid_one, NULL, thread1_func, NULL);
    if (status != 0)
    { // error occurred
        printf("thread1 default = %d\n", status);
        return 1;
    }
    printf("thread1 success create\n");

    // thread 2
    status = pthread_create(&tid_two, NULL, thread2_func, NULL);
    if (status != 0)
    { // error occurred
        printf("thread2 default = %d\n", status);
        return 1;
    }
    printf("thread2 success create\n");
    pthread_join(tid_one, NULL);
    pthread_join(tid_two, NULL);
    pthread_mutex_destroy(&mutex);
    printf("global = %d ", global);
    return 0;
}

```

程序 10: system_call 函数

```

#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
int main(int argc, char* argv[])
{
    char num = *(char*)argv[1];
    // getpid() :进程 ID   pthread_self() :线程 ID   syscall(SYS_gettid):线程
    的PID
    printf("thread%c getpid: %d , the tid=%ld,syscall_pid=%ld\n",num,
getpid(), pthread_self(), syscall(SYS_gettid));
    printf("thread%c return\n",num);
    return 0;
}

```

程序 11：自旋锁实验

```
#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;
}
void spinlock_lock(spinlock_t *lock)
{
    while (__sync_lock_test_and_set(&lock->flag, 1))
    {
        // 自旋等待
    }
}
void spinlock_unlock(spinlock_t *lock)
{
    __sync_lock_release(&lock->flag);
}
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;
    int status;
    spinlock_init(&lock);
    // 输出共享变量的值
    printf("initial: %d\n", shared_value);
    // thread 1
    status = pthread_create(&thread1, NULL, thread_function, &lock);
    if (status != 0)
    {
        printf("thread1 default = %d\n ", status);
        return 1;
    }
    printf("thread1 success create\n");
    // thread 2
    pthread_create(&thread2, NULL, thread_function, &lock);
```



```
if (status != 0)
{
    printf("threa2 default = %d\n ", status);
    return 1;
}
printf("thread2 success create\n");
// 等待线程结束
pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
// 输出共享变量的值
printf("final: %d\n", shared_value);
return 0;
}
```

1.7.2 附件 2 Readme 实验前置 华为云环境搭建

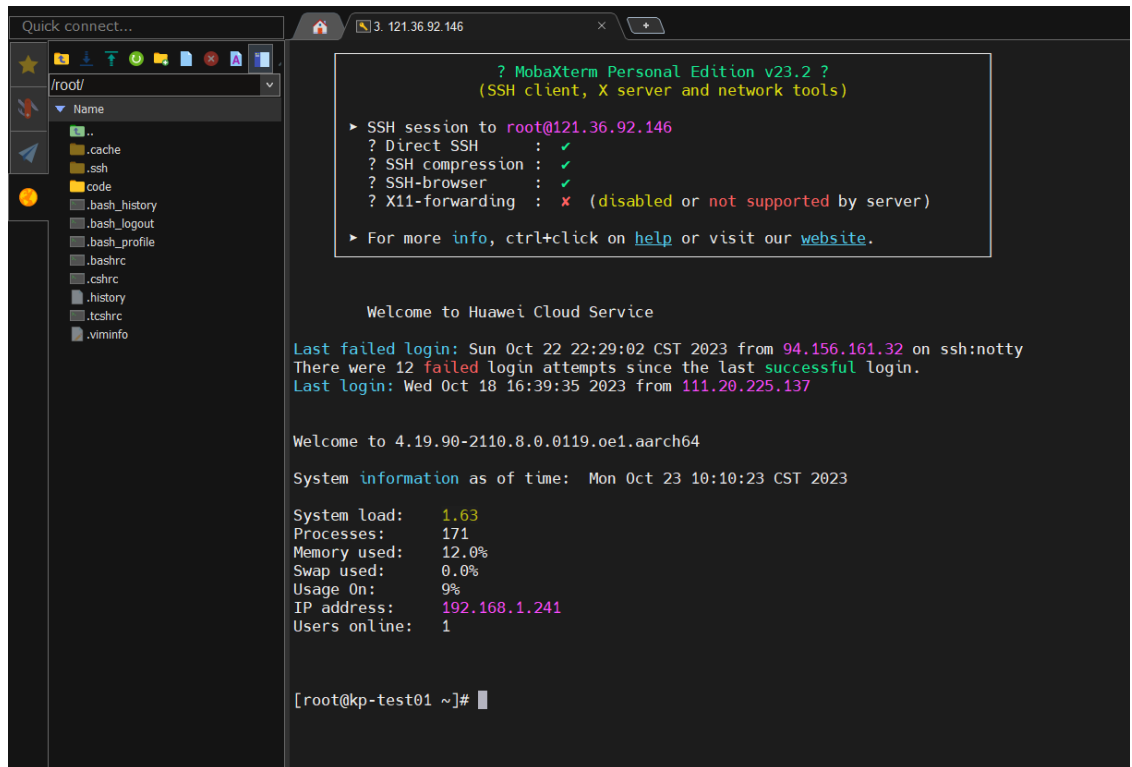
1. 在云端布置服务器



按实验指导书要求配置，服务器参数截图

2. 远程登陆服务器

使用软件 MobaXterm 远程 ssh 登陆服务器 ip :121.36.92.146 登录用户: root 密码:
()



3. 查看服务器信息

ssh 界面键入命令查看服务器的相关信息。

- 查看 gcc 版本

```
[root@kp-test01 ~]# gcc --version
gcc (GCC) 7.3.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- 查看内存信息

```
[root@kp-test01 ~]# free
              total        used        free      shared  buff/cache   available
Mem:           3047872      312128      2224832        13440        510912      2396608
Swap:              0           0           0
```

- 查看 CPU 信息

```
[root@kp-test01 ~]# lscpu
Architecture:          aarch64
CPU op-mode(s):        64-bit
Byte Order:             Little Endian
CPU(s):                 2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):              1
NUMA node(s):          1
Vendor ID:              HiSilicon
Model:                  0
Model name:             Kunpeng-920
Stepping:               0x1
CPU max MHz:            2400.0000
CPU min MHz:            2400.0000
BogoMIPS:               200.00
L1d cache:             128 KiB
L1i cache:             128 KiB
L2 cache:               1 MiB
L3 cache:               32 MiB
NUMA node0 CPU(s):     0,1
Vulnerability Itlb multihit: Not affected
Vulnerability L1tf:     Not affected
Vulnerability Mds:      Not affected
Vulnerability Meltdown: Not affected
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1: Mitigation; __user pointer sanitization
Vulnerability Spectre v2: Not affected
Vulnerability Srbds:    Not affected
Vulnerability Tsx async abort: Not affected
Flags:                  fp asimd evtstrm aes pmull sha1 sha2 crc32 atom
                        ics fphp asimdhp cpuid asimdrdm jscvt fcma dcpo
                        p asimddp asimdghm
```

os 实验 1：进程、线程相关编程实验 1.1 进程相关编程实验

1. 完成图 1.1 程序的运行

有实验截图可知父子进程执行顺序并不固定。

去除 wait 后再观察结果

```
[root@kp-test01 1]# ./1-1
child: pid = 0
parent: pid = 2873
child: pid1 = 2873
parent: pid1 = 2872
[root@kp-test01 1]# ./1-1
parent: pid = 2875
child: pid = 0
parent: pid1 = 2874
child: pid1 = 2875
[root@kp-test01 1]# ./1-1
parent: pid = 2877
child: pid = 0
parent: pid1 = 2876
child: pid1 = 2877
```

在去掉 wait()后，同样也是可能 parent 先执行，又可能 child 先执行。

理论分析：

- fork 创建子进程后，父子进程并行执行，两者执行顺序由 cpu 调度决定，所以二者执行顺序不固定。

- 对于子进程来说, `fork()` 后返回的 `pid` 为 0, `getpid` 返回当前进程 (调用这一函数的进程, 子进程的 `pid`) 所以父进程 `pid` 与子进程的 `pid1` 一样。
- `wait ()` 的作用是让父进程在子进程结束后继续执行, 等待挂起, 防止僵尸进程的出现. 仍会出现 `parent` 先执行, 或 `child` 先执行。

2. 扩展图 11 的程序:

- 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||在 `return` 前增加对全局变量的操作并输出结果:

```
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3139
child: pid = 0
parent: pid1 = 3138
child: pid1 = 3139
global = 10404
global = 9604
[root@kp-test01 1]# ./1-2
global = 98
global = 102
global address = 0x420050
global address = 0x420050
parent: pid = 3141
child: pid = 0
parent: pid1 = 3140
child: pid1 = 3141
global = 10404
global = 9604
```

定义全局变量 `global`, 初值 100. 在子进程加 2, 父进程减 2. 并返回全局变量地址, 在 `return` 前做 `global` 平方操作。发现二者 `global` 地址一样, 但二者 `global` 改变是独立进行的。

理论分析: 子进程“继承”父进程的变量, 其地址总是一样的, 因为在 `fork` 时整个虚拟地址空间被复制, 但是虚拟地址空间所对应的物理内存却没有复制。所以对变量的操作是独立的。

- 调用 `system` 函数和在子进程中调用 `exec` 族函数:

`system` 函数:

发现调用 `systemcall` 后 `pid` 改变, 说明调用该函数创建了一个进程。

```
[root@kp-test01 1]# ./1-3
parent: pid = 3320
child: pid1 = 3320
parent: pid1 = 3319
system_call pid = 3321
[root@kp-test01 1]# ./1-3
parent: pid = 3323
child: pid1 = 3323
parent: pid1 = 3322
system_call pid = 3324
[root@kp-test01 1]# ./1-3
parent: pid = 3326
child: pid1 = 3326
parent: pid1 = 3325
system_call pid = 3327
```

exec 族函数:

发现调用 systemcall 后 pid 未改变, 与 child 的 pid 一样。

```
[root@kp-test01 1]# ./1-4
parent: pid = 3368
child: pid1 = 3368
parent: pid1 = 3367
system_call pid = 3368
[root@kp-test01 1]# ./1-4
parent: pid = 3370
child: pid1 = 3370
parent: pid1 = 3369
system_call pid = 3370
[root@kp-test01 1]# ./1-4
parent: pid = 3372
child: pid1 = 3372
parent: pid1 = 3371
system_call pid = 3372
```

理论分析:

- 当进程调用 exec 函数时, 该进程被完全替换为新程序。因为调用 exec 函数并不创建新进程, 所以前后进程的 ID 并没有改变
- system 函数会执行参数要求的命令创建新的进程所以 pid 改变。

函数参考教程: [Linux 系统学习——exec 族函数、system 函数、popen 函数学习](#)
[exec 跟 system popen 区别-CSDN 博客](#) 1.2 线程相关编程实验

1. 在进程中给一变量赋初值并成功创建两个线程||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

创建变量 global (初值为 0) 两个线程分别执行加 100 和减 100 的操作。

```
root@kp-test01 1]# gcc -o thread thread.c -lpthread
root@kp-test01 1]# ./thread
thread1 success create
thread2 success create
thread1 global = -2530200
thread2 global = -264800
global = -264800
root@kp-test01 1]# ./thread
thread1 success create
thread2 success create
thread1 global = -2375600
thread2 global = -101700
global = -101700
```

可以看出二者是并发执行。每次值都一样因为线程的执行并发，不能保证执行了相同的加和减的操作。

函数教程: [Linux——线程的创建 linux 创建线程-CSDN 博客](#)

编译问题: [Linux 下 undefined reference to 'pthread_create'问题解决-CSDN 博客](#)

2. 控制互斥和同步

使用 `pthread_mutex_` 函数对 `global` 变量进行互斥访问。使线程 1 先执行，线程 2 后执行。

有图 1 可知 thread2 创建后仍在执行 thread1 的操作

```
thread1 success create
thread1 global = -100
thread1 global = -200
thread1 global = -300
thread1 global = -400
thread1 global = -500
thread1 global = -600
thread2 success create
thread1 global = -700
thread1 global = -800
thread1 global = -900
thread1 global = -1000
thread1 global = -1100
thread1 global = -1200
thread1 global = -1300
thread1 global = -1400
thread1 global = -1500
thread1 global = -1600
thread1 global = -1700
thread1 global = -1800
thread1 global = -1900
thread1 global = -2000
thread1 global = -2100
thread1 global = -2200
thread1 global = -2300
thread1 global = -2400
thread1 global = -2500
thread1 global = -2600
```

thread1 减法操作完 thread2 进行操作。

```
thread1 global = -18500  
thread1 global = -18600  
thread1 global = -18700  
thread1 global = -18800  
thread1 global = -18900  
thread1 global = -19000  
thread1 global = -19100  
thread1 global = -19200  
thread1 global = -19300  
thread1 global = -19400  
thread1 global = -19500  
thread1 global = -19600  
thread1 global = -19700  
thread1 global = -19800  
thread1 global = -19900  
thread1 global = -20000  
thread2 global = -19900  
thread2 global = -19800  
thread2 global = -19700  
thread2 global = -19600  
thread2 global = -19500  
thread2 global = -19400  
thread2 global = -19300  
thread2 global = -19200  
thread2 global = -19100  
thread2 global = -19000  
thread2 global = -18900  
thread2 global = -18800  
thread2 global = -18700
```

最后结果为 0

```
thread2 global = -1400  
thread2 global = -1300  
thread2 global = -1200  
thread2 global = -1100  
thread2 global = -1000  
thread2 global = -900  
thread2 global = -800  
thread2 global = -700  
thread2 global = -600  
thread2 global = -500  
thread2 global = -400  
thread2 global = -300  
thread2 global = -200  
thread2 global = -100  
thread2 global = 0  
[root@kp-test01 1]#
```

函数教程: [Linux | 什么是互斥锁以及如何用代码实现互斥锁 linux 实现互斥锁瘦弱的皮卡丘的博客-CSDN 博客](#)

3. 调用系统函数和线程函数的比较

– 调用 system 函数

使用 `syscall(SYS_gettid)`和 `pthread_self()`输出真实 tid 和 tid, 使用 `getpid()`输出 pid。

```
[root@kp-test01 1]# ./pth_sys
thread1 success create
thread2 success create
thread1 global = -1000000
thread1 getpid: 2928 , the tid=281458690224608
thread1 getpid: 2931 , the tid=281459530084096,syscall_pid=2931
thread1 return
thread2 global = -500000
thread2 getpid: 2928 , the tid=281458681770464
thread2 getpid: 2932 , the tid=281464635338496,syscall_pid=2932
thread2 return
global = -500000 [root@kp-test01 1]#
```

线程 1、2 的 `getpid` 相同，线程编号不同。调用 `system` 时创建全新的进程，编号均不同。

每个进程有一个 `pid`（进程 ID），获取函数：`getpid()`，系统内唯一，除了和自己的主线程一样

主线程的 `pid` 和所在进程的 `pid` 一致，可以通过这个来判断是否是主线程

每个线程有一个 `tid`（线程 ID），获取函数：`pthread_self()`，所在进程内唯一，有可能两个进程中都有同样一个 `tid`

每个线程有一个 `pid`（,获取函数：`syscall(SYS_gettid)`，系统内唯一，除了主线程和自己的进程一样，其他子线程都是唯一的。在 linux 下每一个进程都一个进程 id，类型 `pid_t`，可以由 `getpid()`获取。

POSIX 线程也有线程 id，类型 `pthread_t`，可以由 `pthread_self()`获取，线程 id 由线程库维护。

但是各个进程独立，所以会有不同进程中线程号相同节的情况。

进程 id 不可以，线程 id 又可能重复，所以这里会有一个**真实的线程 id 唯一标识，`tid`**。可以通过 linux 下的系统调用 `syscall(SYS_gettid)`来获得。

– 调用 exec 族函数

```
[root@kp-test01 1]# ./pth_exec
thread1 success create
thread2 success create
thread1 getpid: 2864 , the tid=281461631742432global = 5000
thread1 getpid: 2864 , the tid=281460498050816,syscall_pid=2864
thread1 return
thread2 return
global = 5000 [root@kp-test01 1]#
```

指行 `exec` 函数后，原来的进程被调用的内容取代 thread2 的 `systemcall` 不会再进行。

所以调用的 `systemcall` 产生了输出，此时 `systemcall` 为主进程所以 `syscall(SYS_gettid)`与 `pid` 一样。

pid 问题: [linux 中线程的 pid, 线程的 tid 和线程 pid 以及 thread-CSDN 博客](#)

[【编程基础の基础】syscall\(SYS_gettid\) sys_getpid-CSDN 博客](#) 1.3 自旋锁实验

定义了一个 `spinlock_t` 结构体, 用于表示自旋锁。`spinlock_init` 函数用于初始化自旋锁, `spinlock_lock` 函数用于获取自旋锁, `spinlock_unlock` 函数用于释放自旋锁。

在线程函数 `thread_function` 中, 通过调用 `spinlock_lock` 和 `spinlock_unlock` 函数来保护对共享变量 `shared_value` 的访问。每个线程循环执行 5000 次, 每次获取自旋锁后将共享变量加 1, 然后释放自旋锁。

```
[root@kp-test01 1]# ./spinlock
initial: 0
thread1 success create
thread2 success create
final: 10000
[root@kp-test01 1]#
```

2 进程通信与内存管理

2.1 实验目的

2.1.1 进程的软中断通信

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

2.1.2 进程的管道通信

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

2.1.2 进程的管道通信

通过模拟实现页面置换算法（FIFO、LRU），理解请求分页系统中，页面置换的实现思路，理解命中率和缺页率的概念，理解程序的局部性原理，理解虚拟存储的原理。

2.2 实验内容

2.2.1 进程的软中断通信

- 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：使用系统调用 fork() 创建两个子进程，再用系统调用 signal() 让父进程捕捉键盘上发出的中断信号（即 5s 内按下 delete 键或 quit 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 kill() 向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

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

Parent process is killed!!

- 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。

2.2.2 管道通信

- 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 页面的替换

- 1) 理解页面置换算法 FIFO、LRU 的思想及实现的思路。
- 2) 参考给出的代码思路,定义相应的数据结构,在一个程序中实现上述 2 种算法,运行时可以选择算法。算法的页面引用序列要至少能够支持随机数自动生成、手动输入两种生成方式;算法要输出页面置换的过程和最终的缺页率。
- 3) 运行所实现的算法,并通过对比,分析 2 种算法的优劣。
- 4) 设计测试数据,观察 FIFO 算法的 BELADY 现象;设计具有局部性特点的测试数据,分别运行实现的 2 种算法,比较缺页率,并进行分析。

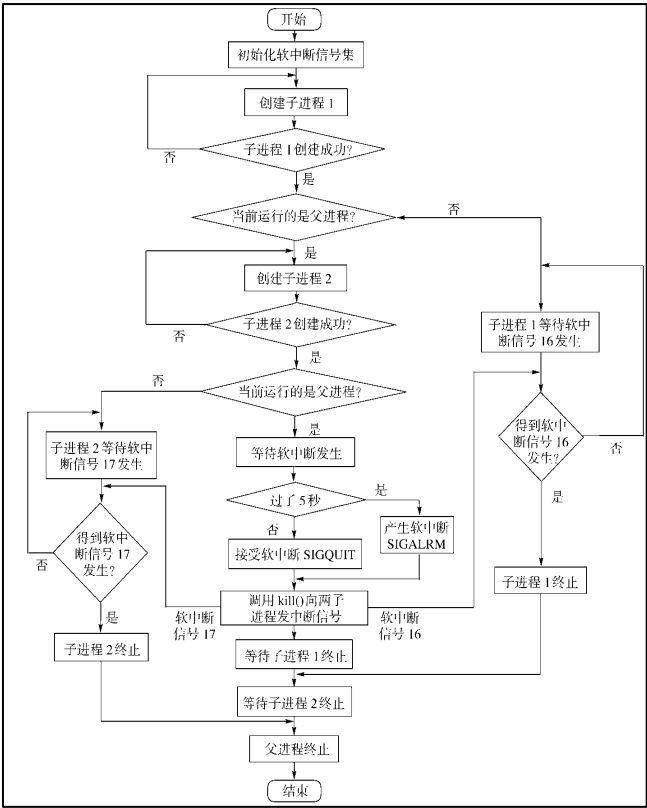
2.3 实验步骤

2.3.1 进程的软中断通信

先猜想一下这个程序的运行结果。然后按照注释里的要求把代码补充完整,运行程序。或者多次运行,并且 Delete/quit, 键后,会出现什么结果? 分析原因

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

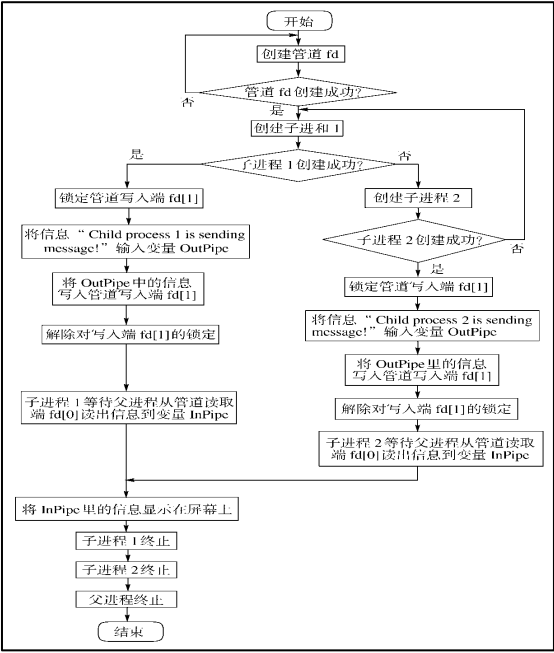
将本实验中通信产生的中断通过 14 号信号值进行闹钟中断,将 signal(3, stop) 当中数字信号变为 2,体会不同中断的执行样式,从而对软中断机制有一个更好的理解。



软中断流程图

2.3.2 进程的管道通信

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



管道通信流程图

2.3.3 页面替换

2.3.3.1 FIFO 算法

1) 准备阶段

选择一个适当的页面数量和物理内存大小，以及一个页面引用序列（即进程在执行过程中引用页面的顺序）。

2) 模拟 FIFO 算法

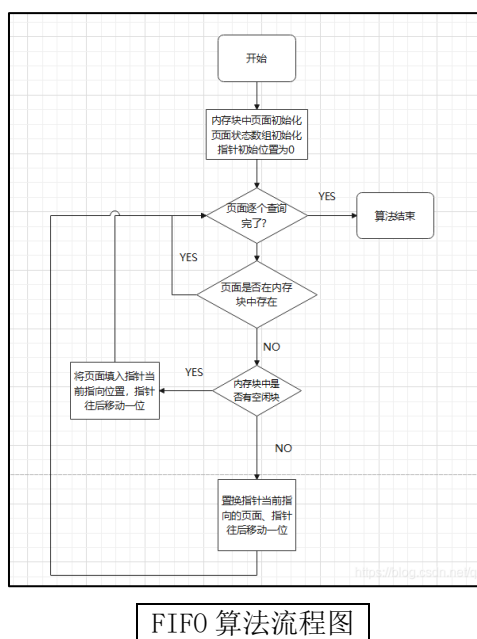
使用编程语言模拟 FIFO 算法的工作过程。可以使用队列来模拟页面的进入和离开。开始按照页面引用序列逐步模拟进程的执行。当物理内存达到限制时，进行页面置换。选择队列最前面的页面进行替换，即最早进入内存的页面。

3) 记录数据

跟踪记录每次页面置换的情况，包括被替换出的页面和进入内存的新页面。

4) 分析和讨论

分析模拟实验的结果，计算缺页率（页面不在物理内存中而需要从磁盘加载的比率）。探讨 FIFO 算法的优点和局限性，特别是其在处理页面使用频率不均匀的情况下可能出现的问题。



2.3.3.2 LRU 算法

1) 准备阶段

选择一个适当的页面数量和物理内存大小，以及一个页面引用序列（即进程在执行过程中引用页面的顺序）。

2) 模拟 LRU 算法

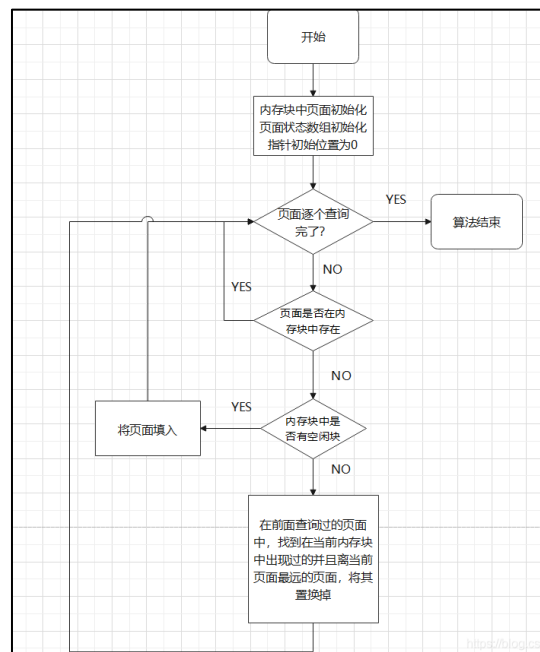
使用编程语言，模拟 LRU 算法的工作过程。可以使用队列、链表或者其他数据结构来记录页面的使用顺序。开始按照页面引用序列逐步模拟进程的执行。当物理内存达到限制时，进行页面置换。选择最近最久未使用的页面进行替换。

3) 记录数据

跟踪记录每次页面置换的情况，包括被替换出的页面和进入内存的新页面。

4) 分析和讨论

分析模拟实验的结果，计算缺页率（页面不在物理内存中而需要从磁盘加载的比率）。探讨 LRU 算法的优点和局限性，特别是其在处理页面使用频率不均匀的情况下可能出现的问题。



LRU 算法流程图

2.4 程序运行初值及运行结果分析

2.4.1 软中断

最初认为运行结果：

- 1) 子进程并行执行顺序不固定，
- 2) 5s 内接收到终止信号时父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号。子进程获得对应软中断信号输出相应信息。
- 3) 5s 后进行闹钟中断，产生 `sigalarm` 信号使父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号。

实际运行结果:

- 1) 该图表示: 5s 后执行闹钟中断(信号 14), 父进程向两个子进程分别发出整数值为 16 和 17 软中断信号, 子进程结束并输出:

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

但是子进程 1 和子进程 2 可能会交替输出, 顺序不固定。

```
[root@kp-test01 2]# ./1.2_  
  
14 stop test  
  
17 stop test  
  
Child process2 is killed by parent!!  
16 stop test  
  
Child process1 is killed by parent!!  
  
Parent process is killed!!
```

执行闹钟中断结果

- 2) 该图表示 5s 内执行 sigquit 信号(信号 2)或 sigint(信号 3), 父进程向两个子进程分别发出整数值为 16 和 17 软中断信号子进程结束。但是子进程 1 和子进程 2 可能会交替输出, 顺序不固定。

```
[root@kp-test01 2]# ./1.2_  
^C  
2 stop test  
  
16 stop test  
  
Child process1 is killed by parent!!  
  
17 stop test  
  
Child process2 is killed by parent!!  
  
Parent process is killed!!  
[root@kp-test01 2]# ./1.2_  
^\  
3 stop test  
  
17 stop test  
16 stop test  
  
Child process2 is killed by parent!!  
Child process1 is killed by parent!!  
  
Parent process is killed!!
```

执行键盘发出中断结果

2.4.2 管道通信

- 1) 你最初认为运行结果会怎么样?

输出 2000 个 1 再输出 2000 个 2。因为存在锁所以不会交替输出。

- 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。:

2023 年 12 月 7 日


```
Set input Mode
1 for rand input ----- 2 for keyboard input
0 for Exit
-----
1

block size : 4
random page number : 7 7 8 3 1 9 6 1 3 4 9 7

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1

FIFO_Mode
for 0 time block:7 -1 -1 -1
successful hit 7
for 1 time block:7 -1 -1 -1
for 2 time block:7 8 -1 -1
for 3 time block:7 8 3 -1
for 4 time block:7 8 3 1
for 5 time block:9 8 3 1
for 6 time block:9 6 3 1
successful hit 1
for 7 time block:9 6 3 1
successful hit 3
for 8 time block:9 6 3 1
for 9 time block:9 6 4 1
successful hit 9
for 10 time block:9 6 4 1
for 11 time block:9 6 4 7
miss = 8
miss rate = 66.67%
-----
```

随机生成模式

- 2) 设计测试数据，观察 FIFO 算法的 BLEADY 现象；设计具有局部性特点的测试数据，分别运行实现的 2 种算法，比较缺页率，并进行分析。

FIFO 算法的 BLEADY 现象：当使用序列 {1, 2, 3, 1, 2, 5, 1, 2, 3, 4, 5} 发现出现 BLEADY 现象：blocksize 增大，缺页率升高。（blocksize = 4 缺页率高于 blocksize = 3）

```
block size : 4
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1

FIFO_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
for 6 time block:5 2 3 4
for 7 time block:5 1 3 4
for 8 time block:5 1 2 4
for 9 time block:5 1 2 3
for 10 time block:4 1 2 3
for 11 time block:4 5 2 3
miss = 10
miss rate = 83.33%
-----
```

Block size = 4

```
block size : 3
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1
-----
FIFO_Mode
for 0 time block:1 -1 -1
for 1 time block:1 2 -1
for 2 time block:1 2 3
for 3 time block:4 2 3
for 4 time block:4 1 3
for 5 time block:4 1 2
for 6 time block:5 1 2
successful hit 1
for 7 time block:5 1 2
successful hit 2
for 8 time block:5 1 2
for 9 time block:5 3 2
for 10 time block:5 3 4
successful hit 5
for 11 time block:5 3 4
miss = 9
miss rate = 75.00%
-----
```

Block size = 3

出现原因:

FIFO 算法的置换特征与进程访问内存的动态特征是矛盾的,即被置换的页面并不是进程不会访问的。

具有局部性特点的测试数据两种算法的优劣性:

{1, 2, 3, 1, 2, 5, 1, 2, 3, 4, 5} 序列具有较好的局部性,在 block =4 情况下观察两种算法:

```
Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
2
-----
LRU_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
pos = 1
for 6 time block:1 2 5 4
successful hit 1
for 7 time block:1 2 5 4
successful hit 2
for 8 time block:1 2 5 4
pos = 0
for 9 time block:1 2 5 3
pos = 1
for 10 time block:1 2 4 3
pos = 3
for 11 time block:5 2 4 3
miss = 8
miss rate = 66.67%
-----
```

LRU

```
block size : 4
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1
-----
FIFO_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
for 6 time block:5 2 3 4
for 7 time block:5 1 3 4
for 8 time block:5 1 2 4
for 9 time block:5 1 2 3
for 10 time block:4 1 2 3
for 11 time block:4 5 2 3
miss = 10
miss rate = 83.33%
-----
```

FIFO

FIFO 的缺页率(83.33%)高于 LRU(75%)，原因：LRU 算法依据局部性原理若当前内存分配的页面数已满，则用新加入的页面直接替换掉最不常访问的页面，既对某些被频繁地访问的页面有较好的利用率；FIFO 算法则是用新加入的页面直接替换掉最先加入的页面，这种算法没有

考虑局部性（被它替换出去的页面并不一定是进程不会访问的）。所以对于局部性较好的数据（某一部分序列频率较高），LRU 算法效果好。

2.5 实验总结

2.5.1 软中断实验

1) 如何阻塞住子进程，让子进程等待父进程发来信号？

子进程通过 `waiting()` 函数等待信号到来，当 `flag` 为 1 时，一直实现阻塞；（接收到信号时 `flag` 值改变），依次确保子进程已经准备好接收信号。

2) 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？

主动退出通过调用 `exit()` 系统调用来实现。`exit()` 系统调用会终止进程的执行，并将控制权交还给操作系统。主动调用 `exit()` 系统调用更好一些。因为这种方式可以确保进程在退出之前完成清理工作，释放资源，关闭文件等操作，而使用 `kill` 命令可能会导致进程突然被终止，无法完成必要的清理工作，可能会导致资源泄漏或者数据丢失。

2.5.2 管道通信实验

1) 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？

通过控制 ``lock()`,`lockf(fd[1],1,0)`` 代表上锁，此时其他进程无法访问管道，``lockf(fd[1],0,0)`` 代表解锁，此时其他进程能访问管道。不控制同步与互斥会让子进程 1、2 同步运行，出现 1,2 交替输出的情况。

2.5.3 页面置换实验

1) 从实现和性能方面，比较分析 FIFO 和 LRU 算法。

FIFO 性能：

缺点：FIFO 算法只考虑页面进入内存的顺序，而不考虑页面的重要性和使用频率，导致性能较差。并存在 Belady 异常（无法根据页面的使用情况进行自适应的页面置换）

优点：只需要维护一个先进先出的队列，复杂度低，开销较低，可以确保每个页面都有被置换的机会，公平地对待每个页面。

LRU 性能:

优点: LRU 算法**适合具有较强时间局部性的访问序列**, 即**最近被访问的页面可能会在未来继续被访问的情况**。例如, 顺序访问或者循环访问的情况下, LRU 算法能够比较好地预测未来的访问模式, 提高缓存命中率。

缺点: 与 FIFO 算法相比, LRU 算法时间复杂度较高, 需进行大量维护栈的操作, 开销较大。
不适合的序列包括周期性访问、随机访问等无法很好地利用时间局部性的情况。

2) LRU 算法是基于程序的局部性原理而提出的算法, 你模拟实现的 LRU 算法有没有体现出该特点? 如果有, 是如何实现的?

有, 体现在维护的数据结构——栈。栈中元素位置由使用改页面的时间先后决定, 符合时间局部性原理。

2.6 附件

程序 1: 软中断

```
// 进程的软中断通信
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <signal.h>
int flag = 1;
void inter_handler(int sig)
{
    flag = 0;
    printf(" \n %d stop test \n", sig);
}
void waiting()
{
    while (flag == 1)
        ;
}
int main()
{
    // 5 秒后产生 alarm 信号
    alarm(5);
    // 产生中断信号
    signal(SIGALRM, inter_handler);
    signal(SIGINT, inter_handler);
    signal(SIGQUIT, inter_handler);
    pid_t pid1 = -1, pid2 = -1;
    while (pid1 == -1)
        pid1 = fork();
    if (pid1 > 0)
    {
```

```
// 父进程
while (pid2 == -1)
    pid2 = fork();
if (pid2 > 0)
{
    // 判断是否 5 秒内产生 SIGQUIT 信号
    // 立即杀死子进程
    // 否则等待 5 秒后杀死子进程
    waiting();
    kill(pid1, 16); // 子进程 1
    kill(pid2, 17); // 子进程 2
    wait(0); // 等待子进程 1 结束
    wait(0); // 等待子进程 2 结束
    printf("\n Parent process is killed!! \n");
}
else
{
    // 子进程 2
    long unsigned int newmask ;
    signal(17, inter_handler);
    //生成新的屏蔽字
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    sigaddset(&newmask, SIGINT);
    sigprocmask(0, &newmask, NULL);
    // 等待父进程发送信号
    waiting();

    printf("\n Child process2 is killed by parent!! \n");
}
}
else
{
    // 子进程 1
    long unsigned int newmask ;
    signal(16, inter_handler);
    //生成新的屏蔽字
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    sigaddset(&newmask, SIGINT);
    sigprocmask(0, &newmask, NULL);
    // 等待父进程发送信号
    waiting();
    printf("\n Child process1 is killed by parent!! \n");
}
return 0;
}
```

程序 2: 管道通信:

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

```

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

```

程序 3: 页面替换:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define input_number 10
#define block_num 4
#define total_input 12

int page[total_input];
int block[block_num];

struct Queue
{
    int head; // first in
    int rear; // first out
    // int num[block_num];
} queue;
//通过队列实现 FIFO 算法
struct Stack
{
    int top;
    int num[block_num];
} stack;
//通过栈实现 LRU 算法
// 寻找 block 中是否有 num 返回位置
int FIFO_Empty(int *block, int num)
{
    int empty = -1;
    for (int i = 0; i < block_num; i++)
    {
        if (block[i] == num)
        {
            return block_num + i;
        }
        if (block[i] == -1)
        {
            empty = i;
        }
    }
    return empty;
}
// 初始化变量
void page_init()
{
    for (int i = 0; i < total_input; i++)
    {
        page[i] = -1;
        block[i] = -1;
        stack.num[i] = -1;
    }
}
// 键盘输入
void input_keyboard()
{

```



```
int i;
int number;
for (i = 0; i < total_input; i++)
{
    printf("number = ");
    scanf("%d", &number);
    page[i] = number;
    if (page[i] < 0 || page[i] > total_input)
    {
        printf("error input\n");
        scanf("number : %d", &page[i]);
    }
}
printf("\nblock size : %d", block_num);
printf("\nkeyboroad page number :");
for (i = 0; i < total_input; i++)
{
    printf(" %d ", page[i]);
}
printf("\n");
}
// 随机生成输入
void input_rand()
{
    int i;
    // 设置随机数种子
    srand(time(NULL));
    for (i = 0; i < total_input; i++)
    {
        page[i] = (rand() % input_number + 1);
    }
    printf("\nblock size : %d", block_num);
    printf("\nrandom page number :");
    for (i = 0; i < total_input; i++)
    {
        printf(" %d ", page[i]);
    }
    printf("\n");
}
// 显示界面
void display1()
{
    printf("\n");
    printf(" Set input Mode \n");
    printf("1 for rand input ----- 2 for keyboard input \n");
    printf("0 for Exit\n");
    printf("-----\n");
}
// 显示界面
void display2()
{
    printf("\n");
    printf(" Set Algorithm Mode \n");
    printf("1 for FIFO ----- 2 for LRU \n");
    printf("0 for Exit\n");
    printf("-----\n");
}
```

```

}
void FIFO()
{
    int miss = 0; // miss number
    double miss_rate;
    int num;
    int empty_flag;
    queue.head = block_num-1;
    queue.rear = block_num;
    printf("-----\n");
    printf("FIFO_Mode\n");
    for (num = 0; num < total_input; num++) // input coming
    {
        empty_flag = FIFO_Empty(block, page[num]);
        // printf("empty = %d \n",empty_flag);
        if (empty_flag >= block_num) // 命中
        {
            printf("successful hit %d\n", page[num]);
        }
        else if (empty_flag == -1) // 没有命中并且满了
        {
            miss++;
            block[queue.head] = page[num];
            if (queue.rear==0) {
                queue.rear =block_num-1;
            }
            else{
                queue.rear = (queue.rear - 1) % (block_num);
            }
            if (queue.head==0) {
                queue.head =block_num-1;
            }
            else{
                queue.head = (queue.head - 1) % (block_num);
            }

            //printf("rear = %d\n",queue.rear);
            //printf("head = %d\n",queue.head);
        }

        else// 没有命中并且没满
        {
            miss++;
            block[empty_flag] = page[num];
            if (queue.rear==0) {
                queue.rear =block_num-1;
            }
            else{
                queue.rear = (queue.rear - 1) % (block_num);
            }
            //printf("rear = %d\n",queue.rear);
        }
        printf("for %d time block:", num);
        for (int i = block_num-1; i >= 0; i--)
        {

```

```

        printf("%d ", block[i]);
    }
    printf("\n");
}
miss_rate = 100 * ((double)miss / total_input);
printf("miss = %d\n", miss);
printf("miss rate = %.2f%%\n", miss_rate);
printf("-----\n");
}
void LRU()
{
    int miss = 0; // miss number
    double miss_rate;
    int num;
    int pos = 0;
    int empty_flag;
    printf("-----\n");
    printf("LRU_Mode\n");
    for (num = 0; num < total_input; num++)
    {
        empty_flag = FIFO_Empty(block, page[num]);
        if (empty_flag >= block_num) // 命中
        {
            for (int i = 0; i < stack.top; i++)
            {
                if (page[num] == stack.num[i])
                {
                    pos = i;
                    break;
                }
            }
            // 更新栈结构体
            for (int i = pos + 1; i < stack.top; i++)
            {
                stack.num[i - 1] = stack.num[i];
            }
            stack.num[stack.top - 1] = page[num];
            printf("successful hit %d\n", page[num]);
        }
        else if (empty_flag == -1) // 没有命中并且满了
        {
            miss++;
            // 找到最远使用替换
            pos = FIFO_Empty(block, stack.num[0]) - block_num;
            printf("pos = %d\n", pos);
            block[pos] = page[num];
            // 更新结构体
            for (int i = 0; i < stack.top - 1; i++)
            {
                stack.num[i] = stack.num[i + 1];
            }
            stack.num[stack.top - 1] = page[num];
        }
        else // 没有命中并且没满
        {
            miss++;

```

```
        block[empty_flag] = page[num];
        stack.num[stack.top] = page[num];
        stack.top++;
    }
    printf("for %d time block:", num);
    for (int i = block_num-1; i >= 0; i--)
    {
        printf("%d ", block[i]);
    }
    printf("\n");
}
miss_rate = 100 * ((double)miss / total_input);
printf("miss = %d\n", miss);
printf("miss rate = %.2f%%\n", miss_rate);
printf("-----\n");
}
int main()
{
    int choice;
    int algorithm;
    while (1)
    {
        page_init();
        display1();
        fflush(stdin);
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                input_rand();
                break;
            case 2:
                input_keyboard();
                break;
            case 0:
                exit(0);
            default:
                printf("error input\n");
                break;
        }
        display2();
        scanf("%d", &algorithm);
        switch (algorithm)
        {
            case 1:
                FIFO();
                break;
            case 2:
                LRU();
                break;
            case 0:
                exit(0);
                break;
            default:
                printf("error input\n");
                break;
        }
    }
}
```

```
    }  
}  
  
return 0;  
}
```

Readme

实验二 进程通信与内存管理

1.进程的软中断通信

实验内容：

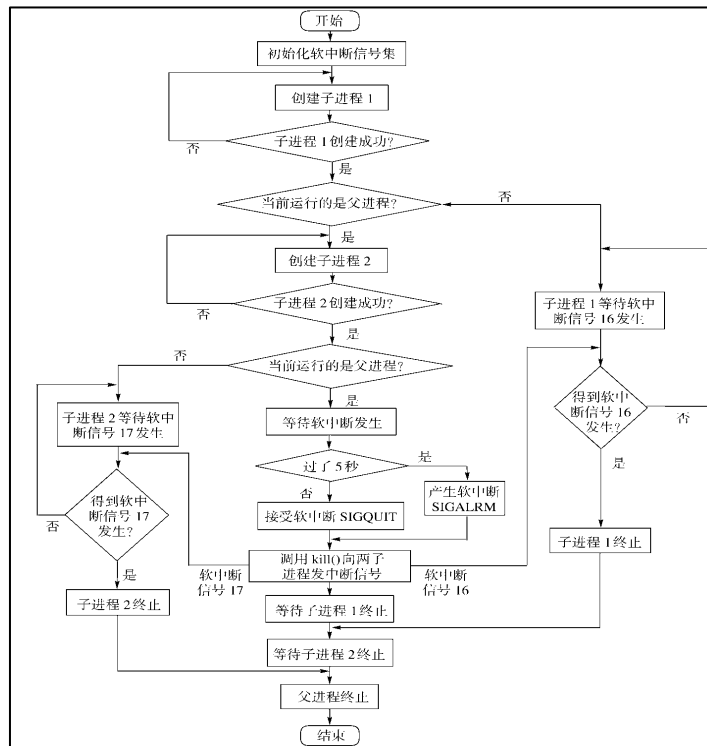
4. 根据流程图编制实现软中断通信的程序：使用系统调用 `fork()` 创建两个子进程，再用系统调用 `signal()` 让父进程捕捉键盘上发出的中断信号（即 5s 内按下 `delete` 键或 `quit` 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

Child process 1 is killed by parent !!

Child process 2 is killed by parent !!

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

Parent process is killed!



软中断流程图

- 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
- 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

编写实验代码需要考虑的问题：

- 如何阻塞住子进程，让子进程等待父进程发来信号？
- 父进程向子进程发送信号时，如何确保子进程已经准备好接收信号？

```

void inter_handler(int sig)
{
    flag = 0;
    printf("\n %d stop test \n", sig);
}

void waiting()
{
    while (flag == 1)
        ;
}

signal(17, inter_handler);
waiting();
printf("\n Child process2 is killed by parent!! \n");
  
```

子进程通过 `waiting()` 函数等待信号到来，当 `flag` 为 1 时，一直实现阻塞；（接收到信号时 `flag` 值改变），依次确保子进程已经准备好接收信号。

运行结果与分析：

9. 最初认为运行结果：

- 子进程并行执行顺序不固定，
- 5s 内接收到终止信号时父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号。子进程获得对应软中断信号输出相应信息。
- 5s 后进行闹钟中断，产生 `sigalarm` 信号使父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 软中断信号。

10. 实际运行结果：

该图表示：5s 后执行闹钟中断（信号 14），父进程向两个子进程分别发出整数值为 16 和 17 软中断信号子进程结束。

```
[root@kp-test01 2]# ./1.2_  
  
14 stop test  
  
17 stop test  
  
Child process2 is killed by parent!!  
16 stop test  
  
Child process1 is killed by parent!!  
  
Parent process is killed!!
```

执行闹钟中断

该图表示 5s 内执行 `sigquit` 信号（信号 2）或 `sigint`（信号 3），父进程向两个子进程分别发出整数值为 16 和 17 软中断信号子进程结束。

```
[root@kp-test01 2]# ./1.2_  
^C  
2 stop test  
  
16 stop test  
  
Child process1 is killed by parent!!  
  
17 stop test  
  
Child process2 is killed by parent!!  
  
Parent process is killed!!  
[root@kp-test01 2]# ./1.2_  
^\  
3 stop test  
  
17 stop test  
16 stop test  
  
Child process2 is killed by parent!!  
Child process1 is killed by parent!!  
  
Parent process is killed!!
```

为实现子进程接收父进程的软中断信号，在子进程中创建了一个新的信号屏蔽字 `newmask`，其中阻塞了 `SIGQUIT` 和 `SIGINT` 信号。接着使用 `sigprocmask` 函数将这个新的屏蔽字应用到了进程中。

```
long unsigned int newmask ;  
signal(16, inter_handler);  
sigemptyset(&newmask);  
sigaddset(&newmask, SIGQUIT);  
sigaddset(&newmask, SIGINT);  
sigprocmask(0, &newmask, NULL);
```

11. 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？

主动退出通过调用 `exit()` 系统调用来实现。`exit()` 系统调用会终止进程的执行，并将控制权交还给操作系统。主动调用 `exit()` 系统调用更好一些。因为这种方式可以确保进程在退出之前完成清理工作，释放资源，关闭文件等操作，而使用 `kill` 命令可能会导致进程突然被终止，无法完成必要的清理工作，可能会导致资源泄漏或者数据丢失。

12.

2.进程的管道通信

实验内容:

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。

完整代码

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
int pid1, pid2; // 定义两个进程变量
int main()
{
    int fd[2];
    char InPipe[4000]; // 定义读缓冲区
    char c1 = '1', c2 = '2';
    pipe(fd);
    // 创建管道
    while ((pid1 = fork()) == 1)
        ; // 如果进程1 创建不成功 则空循环
    if (pid1 == 0)
    { // 如果子进程1 创建成功, pid1 为进程号

        // 锁定管道
        lockf(fd[1], 1, 0);
        // TODO 分2000 次每次向管道写入字符'1'
        for (int i = 0; i < 2000; i++)
        {
            write(fd[1], &c1, 1);
        }
        sleep(5); // 等待读进程读出数据

        // TODO 解除管道的锁定
        lockf(fd[1], 0, 0);
```

```

        exit(0); // 结束进程1
    }
    else
    {
        while ((pid2 = fork()) == 1)
            ; // 若进程2 创建不成功 则空循环
        if (pid2 == 0)
        {
            lockf(fd[1], 1, 0);
            // 分2000 次每次向管道写入字符'2'
            for (int i = 0; i < 2000; i++)
            {
                write(fd[1], &c2, 1);
            }

            sleep(5);
            lockf(fd[1], 0, 0);
            exit(0);
        }
        else
        {
            wait(0); // 等待子进程1 结束
            wait(0); // 等待子进程2 结束
            int bytesRead = read(fd[0], InPipe, 4000); // 从管道中读出4000
个字符
            InPipe[bytesRead] = '\0'; // 加字符串结束符
            printf("%s\n", InPipe); // 显示读出的数据
            exit(0); // 父进程结束
        }
    }
}

```

运行结果与分析：

1. 你最初认为运行结果会怎么样？

输出 2000 个 1 再输出 2000 个 2。因为存在锁所以不会交替输出。

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。

实际结果：先输出 2000 个 1 再输出 2000 个 2，原因：子进程 1 先锁住占用了管道，导致子进程 2 无法访问管道，在子进程 1 完成写操作并解锁后，子进程才能使用管道。

[illegible]

实验过程中发现许运行一段时间才有输出,原因是:由于 pipe 读写的互锁产生延时, pipe 在写完 2000 个 1 和 2000 个 2 后再读的时候才产生输出。

去锁结果:

```
root@kali:~# cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 64 | xargs shuf -n 1
```

由图中可以看出子进程 1 先执行，再子进程 2 开始运行后来哦你跟着开始交替输出。

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？

通过控制 `lock()`, `lockf(fd[1], 1, 0)` 代表上锁, 此时其他进程无法访问管道, `lockf(fd[1], 0, 0)` 代表解锁, 此时其他进程能访问管道。不控制同步与互斥会让子进程 1、2 同步运行, 出现 12 交替输出的情况。

实验参考资料: <https://blog.csdn.net/studyhardi/article/details/89852839>

3.页面的置换

实验内容:

在一个程序中实现上述 2 种算法,运行时可以选择算法。算法的页面引用序列要至少能够支持随机数自动生成、手动输入两种生成方式;算法要输出页面置换的过程和最终的缺页率。

- 运行所实现的算法，并通过对比，分析 2 种算法的优劣。
 - 实现随机数自动生成、手动输入两种生成方式，并输出过程和最终缺页率

```
Set input Mode
1 for rand input ----- 2 for keyboard input
0 for Exit
-----
1

block size : 4
random page number : 7 7 8 3 1 9 6 1 3 4 9 7

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1
-----
FIFO_Mode
for 0 time block:7 -1 -1 -1
successful hit 7
for 1 time block:7 -1 -1 -1
for 2 time block:7 8 -1 -1
for 3 time block:7 8 3 -1
for 4 time block:7 8 3 1
for 5 time block:9 8 3 1
for 6 time block:9 6 3 1
successful hit 1
for 7 time block:9 6 3 1
successful hit 3
for 8 time block:9 6 3 1
for 9 time block:9 6 4 1
successful hit 9
for 10 time block:9 6 4 1
for 11 time block:9 6 4 7
miss = 8
miss rate = 66.67%
-----
```

随机生成模式

```
Set input Mode
1 for rand input ----- 2 for keyboard input
0 for Exit
-----
2

number = 1
number = 2
number = 3
number = 4
number = 1
number = 2
number = 5
number = 1
number = 2
number = 3
number = 4
number = 5

block size : 4
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
2
-----
```

手动输入模式

2. 设计测试数据,观察 FIFO 算法的 BLEADY 现象;设计具有局部性特点的测试数据,分别运行实现的 2 种算法,比较缺页率,并进行分析。
- **FIFO 算法的 BLEADY 现象:** 当使用序列 {1, 2, 3, 1, 2, 5, 1, 2, 3, 4, 5}发现出现 BLEADY 现象: blocksize 增大, 缺页率升高。

```
block size : 4
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1
-----
FIFO_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
for 6 time block:5 2 3 4
for 7 time block:5 1 3 4
for 8 time block:5 1 2 4
for 9 time block:5 1 2 3
for 10 time block:4 1 2 3
for 11 time block:4 5 2 3
miss = 10
miss rate = 83.33%
-----
```

block size = 4

```
block size : 3
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit
-----
1
-----
FIFO_Mode
for 0 time block:1 -1 -1
for 1 time block:1 2 -1
for 2 time block:1 2 3
for 3 time block:4 2 3
for 4 time block:4 1 3
for 5 time block:4 1 2
for 6 time block:5 1 2
successful hit 1
for 7 time block:5 1 2
successful hit 2
for 8 time block:5 1 2
for 9 time block:5 3 2
for 10 time block:5 3 4
successful hit 5
for 11 time block:5 3 4
miss = 9
miss rate = 75.00%
-----
```

block size = 3

出现原因：FIFO 算法的置换特征与进程访问内存的动态特征是矛盾的，即被置换的页面并不是进程不会访问的

— 具有局部性特点的测试数据两种算法的优劣性：

{1, 2, 3, 1, 2, 5, 1, 2, 3, 4, 5}序列具有较好的局部性，在 block=4 情况下观察两种算法：

```

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit

```

```

-----
2
-----

```

```

LRU_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
pos = 1
for 6 time block:1 2 5 4
successful hit 1
for 7 time block:1 2 5 4
successful hit 2
for 8 time block:1 2 5 4
pos = 0
for 9 time block:1 2 5 3
pos = 1
for 10 time block:1 2 4 3
pos = 3
for 11 time block:5 2 4 3
miss = 8
miss rate = 66.67%
-----

```

LRU

```

block size : 4
keyboroad page number : 1 2 3 4 1 2 5 1 2 3 4 5

```

```

Set Algorithm Mode
1 for FIFO ----- 2 for LRU
0 for Exit

```

```

-----
1
-----

```

```

FIFO_Mode
for 0 time block:1 -1 -1 -1
for 1 time block:1 2 -1 -1
for 2 time block:1 2 3 -1
for 3 time block:1 2 3 4
successful hit 1
for 4 time block:1 2 3 4
successful hit 2
for 5 time block:1 2 3 4
for 6 time block:5 2 3 4
for 7 time block:5 1 3 4
for 8 time block:5 1 2 4
for 9 time block:5 1 2 3
for 10 time block:4 1 2 3
for 11 time block:4 5 2 3
miss = 10
miss rate = 83.33%
-----

```

FIFO

FIFO 的缺页率(83.33%)高于 LRU(75%)，原因：LRU 算法依据局部性原理若当前内存分配的页面数已满，则用新加入的页面直接替换掉最不常访问的页面，既对某些被频繁地访问的页面有较好的利用率；FIFO 算法则是用新加入的页面直接替换掉最先加入的页面，这种算法没有考虑局部性（被它替换出去的页面并不一定是进程不会访问的）。所以对于局部性较好的数据（某一部分序列频率较高），LRU 算法效果好。

运行结果与分析：

1. 从实现和性能方面，比较分析 FIFO 和 LRU 算法。

- FIFO

算法实现：

```
void FIFO()
{
    int miss = 0; // miss number
    double miss_rate;
    int num;
    int empty_flag;
    queue.head = block_num-1;
    queue.rear = block_num;
    for (num = 0; num < total_input; num++) // input coming
    {
        empty_flag = FIFO_Empty(block, page[num]);
        if (empty_flag >= block_num) // hit
        {
            printf("successful hit %d\n", page[num]);
        }
        else if (empty_flag == -1) // full but not hit
        {
            miss++;
            block[queue.head] = page[num];
            if (queue.rear==0) {
                queue.rear =block_num-1;
            }
            else{
                queue.rear = (queue.rear - 1) % (block_num);
            }
            if (queue.head==0) {
                queue.head =block_num-1;
            }
            else{
                queue.head = (queue.head - 1) % (block_num);
            }
        }
    }
    else
```



```

    {
        miss++;
        block[empty_flag] = page[num];
        if (queue.rear==0) {
            queue.rear =block_num-1;
        }
        else{
            queue.rear = (queue.rear - 1) % (block_num);
        }
        //printf("rear = %d\n",queue.rear);
    }
    printf("for %d time block:", num);
    for (int i = block_num-1; i >= 0; i--)
    {
        printf("%d ", block[i]);
    }
    printf("\n");
}

miss_rate = 100 * ((double)miss / total_input);
}

```

通过 FIFO_Empty () 函数判断当前内存块状态（能命中，无命中有空块，无命中无空块）。维护一个先进先出的队列，记录页面进入顺序。

无命中有空块：将页面至于空块，更新队列（尾更新）。

无命中无空块：通过队列记录页面进入顺序，进行页面置换。选择队列最前面的页面进行替换，更新队列（头尾均需跟新）。

性能：

缺点：FIFO 算法只考虑页面进入内存的顺序，而不考虑页面的重要性和使用频率，导致性能较差。并存在 Belady 异常（无法根据页面的使用情况进行自适应的页面置换）

优点：只需要维护一个先进先出的队列，复杂度低，开销较低，可以确保每个页面都有被置换的机会，公平地对待每个页面。

- LRU

算法实现：

```

void LRU()
{
    int miss = 0; // miss number
    double miss_rate;
    int num;
    int pos = 0;
    int empty_flag;
    for (num = 0; num < total_input; num++)
    {

```

```

empty_flag = FIFO_Empty(block, page[num]);
if (empty_flag >= block_num) // hit
{
    for (int i = 0; i < stack.top; i++)
    {
        if (page[num] == stack.num[i])
        {
            pos = i;
            break;
        }
    }
    for (int i = pos + 1; i < stack.top; i++)
    {
        stack.num[i - 1] = stack.num[i];
    }
    stack.num[stack.top - 1] = page[num];
    printf("successful hit %d\n", page[num]);
}

else if (empty_flag == -1) // full but not hit
{
    miss++;
    // 找到最远使用替换
    pos = FIFO_Empty(block, stack.num[0]) - block_num;
    block[pos] = page[num];

    // 更新结构体
    for (int i = 0; i < stack.top - 1; i++)
    {
        stack.num[i] = stack.num[i + 1];
    }
    stack.num[stack.top - 1] = page[num];
}

else
{
    miss++;
    block[empty_flag] = page[num];
    stack.num[stack.top] = page[num];
    stack.top++;
}

miss_rate = 100 * ((double)miss / total_input);
}

```

通过 FIFO_Empty () 函数判断当前内存块状态（能命中，无命中有空块，无命中无空块）。维护一个栈，栈代表页面被使用时间的远近。

无命中有空块：将页面至于空块，更新栈（将新加入的页面至于栈顶）。

无命中无空块：找到栈底元素在 block 的位置然后将新加入的元素置换，更新栈（所有栈中元素向下平移）。

命中：找到命中的页面在栈中的位置，将其置于栈顶，其他页面向下平移。

性能:

- LRU 算法适合具有**较强时间局部性的访问序列**，即最近被访问的页面可能会在未来继续被访问的情况。例如，顺序访问或者循环访问的情况下，LRU 算法能够比较好地预测未来的访问模式，提高缓存命中率。不适合的序列包括**周期性访问、随机访问**等无法很好地利用时间局部性的情况。
 - 与 FIFO 算法相比，LRU 算法时间复杂度较高，需进行大量维护栈的操作，开销较大。
2. LRU 算法是基于程序的局部性原理而提出的算法，你模拟实现的 LRU 算法有没有体现出该特点？如果有，是如何实现的？
- 有，**体现在维护的数据结构——栈**。栈中元素位置由使用改页面的时间先后决定，符合时间局部性原理。
3. 在设计内存管理程序时 应如何提高内存利用率。
- 实现内存碎片整理：通过内存碎片整理技术，可以将内存中的碎片化空间进行整理，从而提高内存的利用率。
 - 使用内存池技术：内存池技术可以预先分配一定大小的内存块，在需要时直接从内存池中获取，避免频繁的内存分配和释放操作，从而提高内存的利用率。
 - 实施内存压缩技术：内存压缩技术可以通过对内存中的数据进行压缩，从而减少内存的使用量，提高内存的利用率。
 - 使用内存共享技术：内存共享技术可以让多个进程共享同一块内存，避免多次复制相同的数据，从而提高内存的利用率。

3 模拟文件系统

3.1 实验目的

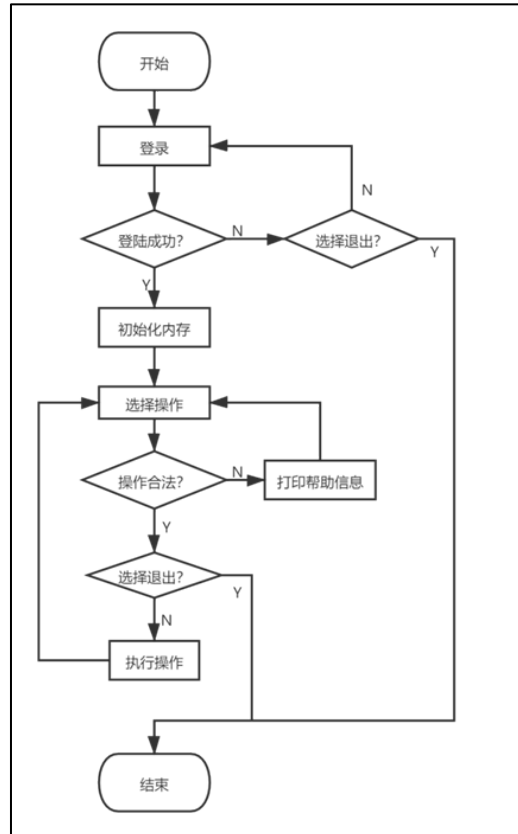
通过一个简单文件系统的设计，加深理解文件系统的内部实现原理

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) 实现包括分配数据块等底层操作
- 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



文件系统流程图

3.4 运行结果与分析

1) 定义数据结构

```
// 组描述符
struct group_desc { // 32 B
    char bg_volume_name[16]; // 文件系统的卷名
    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
    char bg_pad[4]; // 填充(0xff)
};

// 索引结点
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(0~7), 最大为7
    unsigned long 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[24]; // 填充(0xff)
};

// 目录项
struct dir_entry { // 16 B
    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
};

// 用户信息
struct user {
    char username[10];
    char password[10];
}User[USER_MAX];
```

数据结构定义

因为系统中最多有 4096 个数据块, 所以索引节点最多也是 4096。所以索引节点号只需 16 位 (unsigned short) 即可。struct inode 中为保证一个索引节点占 64 位, 需要提供 char i_pad[24], 作为填充。

用户结构体存储用户的密码和用户名

2) 实现底层操作

1. 读写缓冲区操作函数

```
// 写gdt
static void update_group_desc()
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, GDT_START, SEEK_SET);
    fwrite(&gdt, GD_SIZE, 1, fp);
    fflush(fp);
}

// 读gdt
static void reload_group_desc()
{
    fseek(fp, GDT_START, SEEK_SET);
    fread(&gdt, GD_SIZE, 1, fp);
}
```

读写组描述符

`update_group_desc`函数打开一个名为"FS.txt"的文件, 以读写模式打开 ("rb+"), 然后将文件指针移动到 GDT 的起始位置 (GDT_START), 并将内存中的 GDT 数据 (gdt) 写入文件。

最后，使用`fflush`函数刷新文件流，确保数据被写入文件。

`reload_group_desc`函数假定文件指针`fp`已经打开，然后将文件指针移动到 GDT 的起始位置（GDT_START），并从文件中读取 GDT 数据到内存中的`gdt`变量中。其他结构体缓冲区读写函数依次类推：但需注意何时用指针合适取地址

2. 分配删除块函数：标注位图

```
// 分配data_block
static int alloc_block()
{
    int flag = 0;
    if (gdt.bg_free_blocks_count == 0)
    {
        printf("There is no block to be allocated!\n");
        return 0;
    }
    reload_block_bitmap();
    for (int i = 0; i < 512; i++)
    {
        if (bitbuf[i] != 0xff)
        {
            for (int j = 0; j < 8; j++)
            {
                if ((bitbuf[i] & (1 << j)) == 0)
                {
                    bitbuf[i] |= (1 << j);
                    last_alloc_block = i * 8 + j;
                    break;
                }
            }
            break;
        }
    }

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

分配数据块

以分配块函数为例，循环遍历块位图的每个字节（共 512 个字节），然后再遍历每个字节中的每一位做与运算。如果某一位为 0，表示该数据块未被分配，则将该位设置为 1，表示分配了该数据块，并记录下该数据块的编号。最后跳出循环，返回分配的数据块编号。

3. 配置新节点及初始化目录

根据类型`type`的值来初始化`i`节点的各个字段。如果类型为 2（表示目录），则设置`i`节点的字段，并初始化目录项`dir`的内容，并将`.`和`..`目录项的名称赋值为当前目录和父目录的名称。

如果类型不为 2（表示文件），则设置`i`节点的大小、块数、创建时间、修改时间、访问时间等字段，并根据文件名的后缀(扩展名为`.exe`、`.bin`、`.com` 及不带扩展名的)来判断是否为可执行文件，如果是则设置`i`节点的执行权限。

3) 实现命令层函数

1. cd 改变路径

```
[root@ext2 /]#mkdir 1
[root@ext2 /]#cd 1
[root@ext2 /1/]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>     r_w_
..         <DIR>     r_w_
[root@ext2 /1/]#
```

cd 示意图

cd 分为三种情况：主要考虑更改 `current_dir`和`current_path`

- 1. 传送到上一级目录（..）但不在根目录上
- 2. 在根目录上执行 `cd ..`（直接 return）
- 3. 传送到子目录（考虑多级）:以/为分隔符,截取每一级的路径，搜索其索引节点及路径，不断循环。下图展示了分割路径的路径，通过遍历路径寻找 “/” 分割每一层路径。

```
[root@ext2 /]#cd 1/2/op
tmp[length] = 1
tmp2 = 1
tmp[length] = /
tmp2 = 1/
tmp2 = 1
current_dir: 2
current_path: [root@ext2 /1/
tmp[length] = 2
tmp2 = 2
tmp[length] = /
tmp2 = 2/
tmp2 = 2
current_dir: 3
current_path: [root@ext2 /1/2/
tmp[length] = 0
tmp2 = 0
tmp[length] = p
tmp2 = op
tmp2 = op
[root@ext2 /1/2/op/]#
```

cd 多级目录 示意图

2. mkdir 创建目录

```
[root@ext2 /]#mkdir op
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
op         <DIR>      r_w_      2023.11.28 20:43:43      2023.11.28 20:43:43      2023.11.28 20:43:43
```

创建目录 示意图

3. rmdir 删除目录

```
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
list       <DIR>      r_w_      2023.11.28 20:15:32      2023.11.28 20:15:32      2023.11.28 20:15:32
[root@ext2 /]#rmdir list
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
[root@ext2 /]#
```

首先检查要删除的目录是否为"."或"..", 如果是则无法删除，否则继续执行。如果目录下只有"."和".."两个项，则直接删除该目录，更新父目录信息，并释放相关的数据块和节点。

如果目录下还有其他文件或子目录，则递归调用 `rmdir` 函数，依次删除其中的文件和子目录。

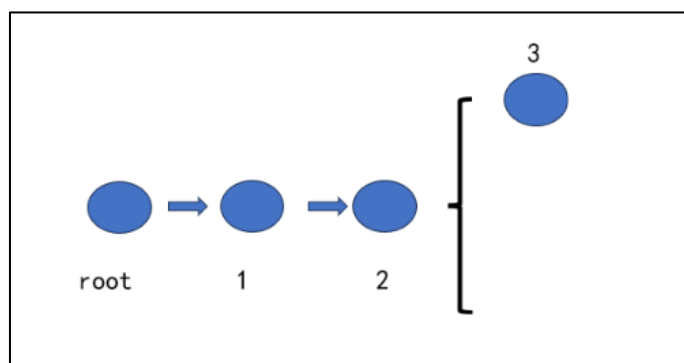
最后更新父目录信息。

在递归调用时要注意更新 `current_dir`，与当前删除的一级的目录相对应。

```
[root@ext2 /]#rmdir 1
initial tmp1: 1
2current_inode: 5
dir[m].name: 2
initial tmp1: 5
1current_inode: 6
dir[m].name: 3
the dic is empty
current_inode: 5
initial tmp1: 5
current_inode over: 5
the dic is empty
current_inode: 1
initial tmp1: 1
current_inode over: 1
```

current_dir 变化示意图

该文件系统的结构为：



有图中可知结点从根节点先层层递进到达目录 2，删除文件 3 后（判断目录是否已空）再删除目录本身。从子节点再递归删除返回。

4. open_file 打开文件

```
[root@ext2 /]#create 1
[root@ext2 /]#open 1
1 sucessfully opened!
```

成功打开文件过程

首先查找文件，如果找到了文件，则检查该文件是否已经被打开，如果已经被打开，则输出提示信息。如果文件没有被打开，则在 `fopen_table` 数组中找到一个空闲的位置，将该文件的 inode 号存入该位置。然后更新文件的访问时间。

5. read_file 读文件


```
[root@ext2 /]#create 1
[root@ext2 /]#open 1
1 sucessfully opened!
[root@ext2 /]#write 1
asdasdawdawdawdaw#

length = 20
[root@ext2 /]#read 1
asdasdawdawdawdaw
```

写小容量文件过程

创建文件需满足两个条件：

1. 该文件是可写的 W
2. 该文件已打开 open

若满足条件即可写入。写入时首先计算所需块数 `need_blocks`。根据索引个数选择索引方式：

1. (`need_blocks < 6`) 直接索引将数据块的位置信息存储在文件的 `inode` 项中的 `i_blocks` 数组中。每个数组元素对应一个数据块的位置。在代码中，通过更新 `inode` 项的 `i_blocks` 数组来分配或释放直接索引所需的数据块。然后，使用循环将数据块中的数据逐块写入磁盘。
2. (`need_blocks < 262`) 一级子索引是一个额外的索引块，其中存储的块号指向一个数据块的位置。索引块中用 `unsigned int 16 位变量`，即 2 字节表示 1 个块号。索引块都用来存放块号，可以存放 $512/2=256$ 个。因为 `Buffer` 为 `char` 型（8 位），所以每得到一个数据块需要取两个 `Buffer[]` 代表高 8 位和低 8 位。

```
Buffer[j * 2] = block_num / 256;
Buffer[j * 2 + 1] = block_num % 256;
```

3. (`need_blocks < 4072`) 写入二级索引块。代码通过循环分配一级索引块的块号，并在每个一级索引块中再次循环分配块号，将这些块的块号存储在二级索引块中的相应位置。写入数据块。代码通过循环读取 `tempbuf` 中的数据，并将其写入对应的数

数据块中。前 6 个数据块写入直接索引块，接下来的 256 个数据块写入一级索引块指向的块，剩余的数据块按照二级索引块的结构依次写入。

```
length = 3955
need_blocks = 8
inode_buff.i_block[inode_buff.i_blocks] = 1
inode_buff.i_block[inode_buff.i_blocks] = 2
inode_buff.i_block[inode_buff.i_blocks] = 3
inode_buff.i_block[inode_buff.i_blocks] = 4
inode_buff.i_block[inode_buff.i_blocks] = 5
inode_buff.i_block[inode_buff.i_blocks] = 6
inode_buff.i_block[6] = 7
Buffer[0*2] = 0
Buffer[0*2+1] = 8
Buffer[1*2] = 0
Buffer[1*2+1] = 9
Buffer[0*2] = 0
Buffer[0*2+1] = 8
block_num = 8
Buffer[1*2] = 0
Buffer[1*2+1] = 9
block_num = 9
inode_buff.i_size = 3955
```

写大容量文件过程并输出对应块号

7. ls 显示指定工作目录下之内容

```
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>      r_w_
..          <DIR>      r_w_
1           <FILE>     r_w_x      2023.11.28 22:8:19    2023.11.28 22:8:19    2023.11.28 22:8:19
list       <DIR>      r_w_      2023.11.28 22:8:26    2023.11.28 22:8:26    2023.11.28 22:8:26
```

8. chmod 更改文件权限

```
[root@ext2 /]#create file
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>      r_w_
..          <DIR>      r_w_
op          <DIR>      r_w_      2023.11.29 13:49:41    2023.11.29 13:49:41    2023.11.29 13:49:41
file       <FILE>     r_w_      2023.11.29 13:52:2     2023.11.29 13:52:2     2023.11.29 13:52:2
[root@ext2 /]#chmod file 5
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>      r_w_
..          <DIR>      r_w_
op          <DIR>      r_w_      2023.11.29 13:49:41    2023.11.29 13:49:41    2023.11.29 13:49:41
file       <FILE>     r__x      2023.11.29 13:52:2     2023.11.29 13:52:2     2023.11.29 13:52:2
```

检查权限模式是否在 0 到 7 之间，如果不在范围内则输出错误信息并返回。如果权限模式在范围内，则重新加载文件的索引节点信息，更新权限模式，并更新索引节点信息。如果文件不存在，则输出文件不存在的错误信息。

9. initialize_user 初始化用户 password_change 更改密码

```
PS C:\Users\邱子杰\Desktop\os\实验\3\code> gcc ext2.c main.c -o Fs
root
Please print password :
root
User root sign in!
[root@ext2 /]#password root
Please input your new password:
123123
[root@ext2 /]#quit
Good Bye!
PS C:\Users\邱子杰\Desktop\os\实验\3\code> gcc ext2.c main.c -o Fs
PS C:\Users\邱子杰\Desktop\os\实验\3\code> ./Fs.exe
Please print username :
root
Please print password :
root
User name or password wrong, please enter again!
If want to exit, please enter "quit" or "exit"!
Please print username :
root
Please print password :
123123
User root sign in!
[root@ext2 /]#
```

登陆界面

```
// 修改密码
void password_change(char username[10], char password[10])
{
    for (int i = 0; i < USER_MAX; i++)
    {
        if (!strcmp(User[i].username, username))
        {
            strcpy(User[i].password, password);
            fp = fopen("./password.txt", "w+");
            fwrite(User, sizeof(struct user), USER_MAX, fp);
            break;
        }
    }
    return;
}
```

修改密码代码

以写入模式重新打开`password.txt`文件（存储用户名和密码文件），并使用`fwrite`将更新后的`User`数组写入文件。



password.txt

4) 完成 shells 设计

1. shell 层进行用户登陆

```
Please print username :
root
Please print password :
123123
User root sign in!
[root@ext2 /]#
```

用户登陆界面

2. 执行命令行（包括“help”查询操作用法）

```
[root@ext2 /]#help
=====help_list=====
=====
1.cd : cd + path                2.mkdir : mkdir + dirname
3.rmdir : rmdir + dirname      4.ls : ls
5.create : create + filename   6.open : open + filename
7.close : close + filename     8.read : read + filename
9.write : write + filename     10.rm : rm + filename
11.chmod : chmod + filename + mode 12.password : password + tmp
13.format : format             14.quit : quit
=====
5. 10. 10. /]#
```

Help 查询界面

3. 退出文件系统（quit）

```
[root@ext2 /]#quit
Good Bye!
PS C:\Users\邱子杰\Desktop\os\实验\3\code>
```

退出界面

4. 格式化系统（format）

```
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>     r_w_
..          <DIR>     r_w_
[root@ext2 /]#create 1
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>     r_w_
..          <DIR>     r_w_
1          <FILE>    r_w_x     2023.11.28 22:15:46  2023.11.28 22:15:46  2023.11.28 22:15:46
[root@ext2 /]#format
Creating the ext2 file system
The ext2 file system has been installed!
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.           <DIR>     r_w_
..          <DIR>     r_w_
[root@ext2 /]#ls
```

格式化界面

3.5 遇到的问题与解决方法

1. 如何递归地实现 rmdir 删除目录

实现方法为：以 ‘/’ 为分隔符截取每一层路径，一级级深入路径的叶子节点，从叶子节点回溯删除目录的内容。

2. 如何实现多级索引机制

主要的思想是将 2 个 char 型数据块单元整合成一个 16 位索引块号。

一级子索引是一个额外的索引块，其中每个索引项指向一个数据块的位置。索引块中用 unsigned int 16 位变量，即 2 字节表示 1 个块号。索引块都用来存放块号。读取时若想获得一个块号需读取两个 Buffer[]，代表最终块号的高 8 位和低 8 位。

块号 = Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1]; 读取对应块号的内容。

读入二级索引块。通过循环读入一级索引块的块号，并在每个一级索引块中再次循环读入块号，将这些块的块号对应的数据块载入读取。

3.6 实验总结

基本实现了类 ext2 文件系统的基本功能包括底层函数的编写，命令层函数的调用和 shell 的设计。通过本次设计并实现 ext2 文件系统，我对 ext2 filesystem 的思想、结构、设计方式有了深入的了解，在实现的过程中，也对一些用到的函数掌握程度更深入，比如 fopen, fseek, fwrite, fread, fflush 等函数，总的来说，这次实验让我收获颇丰。

3.7 附件

代码 1: ext2.h

```

#ifndef EXT2_INIT_H
#define EXT2_INIT_H
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "main.h"
#define VOLUME_NAME      "EXT2FS"      //卷名
#define BLOCK_SIZE       512           //块大小
#define DISK_SIZE        4611          //磁盘总块数
#define DISK_START       0             //磁盘开始地址
#define GD_SIZE          32            //块组描述符大小是 32B
#define GDT_START        0            //块组描述符起始地址
#define BLOCK_BITMAP     512          //块位图起始地址 512
#define INODE_BITMAP     1024         //inode 位图起始地址 3*512
#define INODE_TABLE      1536         //索引节点表起始地址 3*512
#define INODE_SIZE       64           //每个 inode 的大小是 64B
#define INODE_TABLE_COUNTS 4096       //inode entry 数量
#define DATA_BLOCK      263680       //数据块起始地址 3*512+4096*64
#define DATA_BLOCK_COUNTS 4096       //数据块数
#define BLOCKS_PER_GROUP 4611         //每组中的块数
#define USER_MAX         4            //用户个数
#define FOPEN_TABLE_MAX  16           //文件打开表大小
// 组描述符
struct group_desc { // 32 B
    char bg_volume_name[16]; //文件系统的卷名
    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
    char bg_pad[4]; //填充(0xff)
};
// 索引结点
struct inode { // 64 B = 6 * 2B + 5 * 4B + 16B + 16B
    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(0~7), 最大为 7
    unsigned short i_uid; //文件所有者标识号
    unsigned short i_gid; //文件的用户组标识符
    unsigned short i_links_count; //文件的硬链接计数
    unsigned short i_flags; //打开文件的方式
    unsigned long 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[16]; //填充(0xff)
};
// 目录项
struct dir_entry { // 16 B

```



```
    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
};
// 用户信息
struct user {
    char username[10];
    char password[10];
    unsigned short u_uid; //用户标识号
    unsigned short u_gid;
}User[USER_MAX];
static unsigned short last_alloc_inode; // 最近分配的 i 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[FOPEN_TABLE_MAX]; // 文件打开表
static struct group_desc gdt; // 组描述符缓冲区
static struct inode inode_buff; // inode 缓冲区
static unsigned char bitbuf[512] = {0}; // block 位图缓冲区
static unsigned char ibuf[512] = {0}; // inode 位图缓冲区
static struct dir_entry dir[32]; // 目录项缓冲区 32*16=512
static char Buffer[BLOCK_SIZE]; // 针对数据块的缓冲区
static char tempbuf[4096*512]; // 文件写入缓冲区
static FILE *fp; // 虚拟磁盘指针
static short buffer_1[256]; // 一级索引缓冲区 256*2=512
char current_path[256]; // 当前路径
char current_user[10]; // 当前用户
static void update_group_desc(); //更新组描述符内容
static void reload_group_desc(); //加载组描述符内容
static void update_inode_entry(unsigned short i); //更新 indoe
static void reload_inode_entry(unsigned short i); //加载 inode
static void update_block_bitmap(); //更新块位
static void reload_block_bitmap(); //加载块位
static void update_inode_bitmap(); //更新 inode 位图
static void reload_inode_bitmap(); //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 update_block_1(unsigned short i); //更新数据块
static void reload_block(unsigned short i); //加载数据块
static void reload_block_1(unsigned short i); //加载数据块
static int alloc_block(); //分配数据块
static int get_inode(); //得到 inode 节点
static unsigned short research_file(char tmp[100], 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, char
name[100]);
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 initialize_disk(); //初始化磁盘
#endif //EXT2_INIT_H
```

程序 2: ext2.c

```
#include "ext2.h"
// 写 gdt
static void update_group_desc()
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, GDT_START, SEEK_SET);
    fwrite(&gdt, GD_SIZE, 1, fp);
    fflush(fp);
}
// 读 gdt
static void reload_group_desc()
{
    fseek(fp, GDT_START, SEEK_SET);
    fread(&gdt, GD_SIZE, 1, fp);
}
// 写 inode
static void update_inode_entry(unsigned short i)
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, INODE_TABLE + (i - 1) * INODE_SIZE, SEEK_SET);
    fwrite(&inode_buff, INODE_SIZE, 1, fp);
    fflush(fp);
}
// 读 inode
static void reload_inode_entry(unsigned short i)
{
    fseek(fp, INODE_TABLE + (i - 1) * INODE_SIZE, SEEK_SET);
    fread(&inode_buff, INODE_SIZE, 1, fp);
}
// 写 dir
static void update_dir(unsigned short i)
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fwrite(dir, BLOCK_SIZE, 1, fp);
    fflush(fp);
}
// 读 dir
static void reload_dir(unsigned short i)
{
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fread(dir, BLOCK_SIZE, 1, fp);
}
// 写 blockmap
static void update_block_bitmap()
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, BLOCK_BITMAP, SEEK_SET);
    fwrite(bitbuf, BLOCK_SIZE, 1, fp);
}
```

```
fflush(fp);
}
// 读 blockmap
static void reload_block_bitmap()
{
    fseek(fp, BLOCK_BITMAP, SEEK_SET);
    fread(bitbuf, BLOCK_SIZE, 1, fp);
}
// 写 inodemap
static void update_inode_bitmap()
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, INODE_BITMAP, SEEK_SET);
    fwrite(ibuf, BLOCK_SIZE, 1, fp);
    fflush(fp);
}
// 读 inodemap
static void reload_inode_bitmap()
{
    fseek(fp, INODE_BITMAP, SEEK_SET);
    fread(ibuf, BLOCK_SIZE, 1, fp);
}
// 写 data_block
static void update_block(unsigned short i)
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fwrite(Buffer, BLOCK_SIZE, 1, fp);
    fflush(fp);
}
// 写 data_block
static void update_block_1(unsigned short i)
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fwrite(buffer_1, BLOCK_SIZE, 1, fp);
    fflush(fp);
}
// 读 data_block
static void reload_block(unsigned short i)
{
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fread(Buffer, BLOCK_SIZE, 1, fp);
}
// 读 data_block
static void reload_block_1(unsigned short i)
{
    fseek(fp, DATA_BLOCK + i * BLOCK_SIZE, SEEK_SET);
    fread(buffer_1, BLOCK_SIZE, 1, fp);
}
// 分配 data_block
static int alloc_block()
{
    int flag = 0;
    if (gdt.bg_free_blocks_count == 0)
```

```
{
    printf("There is no block to be allocated!\n");
    return (0);
}
reload_block_bitmap();
for (int i = 0; i < 512; i++)
{
    if (bitbuf[i] != 0xff)
    {
        for (int j = 0; j < 8; j++)
        {
            if ((bitbuf[i] & (1 << j)) == 0)
            {
                bitbuf[i] |= (1 << j);
                last_alloc_block = i * 8 + j;
                break;
            }
        }
        break;
    }
}
update_block_bitmap();
gdt.bg_free_blocks_count--;
update_group_desc();
return last_alloc_block;
}
// 分配 inode
static int get_inode()
{
    int flag = 0;
    if (gdt.bg_free_inodes_count == 0)
    {
        printf("There is no Inode to be allocated!\n");
        return 0;
    }
    reload_inode_bitmap();
    for (int i = 0; i < 512; i++)
    {
        if (ibuf[i] != 0xff)
        {
            for (int j = 0; j < 8; j++)
            {
                if ((ibuf[i] & (1 << j)) == 0)
                {
                    ibuf[i] |= (1 << j);
                    last_alloc_inode = i * 8 + j + 1;
                    break;
                }
            }
            break;
        }
    }
}
update_inode_bitmap();
gdt.bg_free_inodes_count--;
update_group_desc();
return last_alloc_inode;
```

```
}
// 查找
static unsigned short research_file(char tmp[100], int file_type, unsigned
short *inode_num,
                                unsigned short *block_num, unsigned
short *dir_num)
{
    unsigned short j, k;
    reload_inode_entry(current_dir); // 进入当前目录
    j = 0;
    while (j < inode_buff.i_blocks)
    {
        reload_dir(inode_buff.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
static void dir_prepare(unsigned short tmp, unsigned short len, int type,
char name[100])
{
    reload_inode_entry(tmp);

    time_t Time;
    time(&Time);
    if (type == 2)
    { // dir
        inode_buff.i_size = 32;
        inode_buff.i_blocks = 1;
        inode_buff.i_block[0] = alloc_block();
        inode_buff.i_ctime = Time;
        inode_buff.i_mtime = Time;
        inode_buff.i_atime = Time;
        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_buff.i_block[0]);

    inode_buff.i_mode = 6;
}
else
{
    inode_buff.i_size = 0;
    inode_buff.i_blocks = 0;
    inode_buff.i_mode = 6;
    inode_buff.i_ctime = Time;
    inode_buff.i_mtime = Time;
    inode_buff.i_atime = Time;
    int len = strlen(name);
    if (len < 4)
    {
        inode_buff.i_mode |= 1;
    }
    else
    {
        char *lastFour = &name[len - 4];
        if (strcmp(lastFour, ".exe") == 0 || strcmp(lastFour, ".bin") ==
0 || strcmp(lastFour, ".com") == 0)
        {
            inode_buff.i_mode |= 1;
        }
        else
        {
            ;
        }
    }
}
update_inode_entry(tmp);
}
// 删除 data_block
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.bg_free_blocks_count++;
    update_group_desc();
}
// 删除 inode
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.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;
}
// 初始化磁盘
void initialize_disk()
{
    int i = 0;
    printf("Creating the ext2 file system\n");
    last_alloc_inode = 1;
    last_alloc_block = 0;
    for (i = 0; i < FOPEN_TABLE_MAX; 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("./FS.txt", "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++)
    {
        // 清空文件, 即清空磁盘全部用 0 填充, Buffer 为缓冲区起始地址,
BLOCK_SIZE 表示读取大小, 1 表示写入对象的个数
        fwrite(Buffer, BLOCK_SIZE, 1, fp);
    }
    // 根目录的 inode 号为 1
    reload_inode_entry(1);
    reload_dir(0);
    // 修改路径名为根目录
    strcpy(current_path, "");
    strcat(current_path, "[");
    strcat(current_path, current_user);
    strcat(current_path, "@FS.txt /");
    // 初始化组描述符内容
    reload_group_desc();
    gdt.bg_block_bitmap = BLOCK_BITMAP; // 第一个块位图的起始地址
    gdt.bg_inode_bitmap = INODE_BITMAP; // 第一个 inode 位图的起始
地址
    gdt.bg_inode_table = INODE_TABLE; // inode 表的起始地址
    gdt.bg_free_blocks_count = DATA_BLOCK_COUNTS; // 空闲数据块数

```

```

gdt.bg_free_inodes_count = INODE_TABLE_COUNTS; // 空闲 inode 数
gdt.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();
inode_buff.i_mode = 518;
inode_buff.i_blocks = 0;
inode_buff.i_size = 32;
inode_buff.i_atime = 0;
inode_buff.i_ctime = 0;
inode_buff.i_mtime = 0;
inode_buff.i_dtime = 0;
inode_buff.i_block[0] = alloc_block(); // 分配数据块
inode_buff.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_buff.i_block[0]);
printf("The ext2 file system has been installed!\n");
// check_disk();
// fclose(fp);
}
// 初始化用户信息
void initialize_user()
{
    // 创建 password.txt
    FILE *fp;
    fp = fopen("./password.txt", "r+");
    if (fp == NULL)
    {
        fp = fopen("./password.txt", "w+");
        // 初始化用户信息
        strcpy(User[0].username, "test");
        strcpy(User[0].password, "test");
        strcpy(User[1].username, "user");
        strcpy(User[1].password, "user");
        strcpy(User[2].username, "root");
        strcpy(User[2].password, "root");
        strcpy(User[3].username, "admin");
        strcpy(User[3].password, "admin");
        fwrite(User, sizeof(struct user), USER_MAX, fp);
        printf("The password.txt has been created!\n");
        fclose(fp);
    }
    // 读取 password.txt
    fp = fopen("./password.txt", "r+");
    fread(User, sizeof(struct user), USER_MAX, fp);
    fclose(fp);
}

```



```
        return;
    }
    // 修改密码
    void password_change(char username[10], char password[10])
    {
        for (int i = 0; i < USER_MAX; i++)
        {
            if (!strcmp(User[i].username, username))
            {
                strcpy(User[i].password, password);
                fp = fopen("./password.txt", "w+");
                fwrite(User, sizeof(struct user), USER_MAX, fp);
                break;
            }
        }
        return;
    }
    // 用户登录
    int login(char username[10], char password[10])
    {
        for (int i = 0; i < USER_MAX; i++)
        {
            if (!strcmp(User[i].username, username))
            {
                if (!strcmp(User[i].password, password))
                    return 1;
                break;
            }
        }
        return 0;
    }
    // 初始化内存
    void initialize_memory()
    {
        int i = 0;
        last_alloc_inode = 1;
        last_alloc_block = 0;
        for (i = 0; i < FOPEN_TABLE_MAX; i++)
        {
            fopen_table[i] = 0;
        }
        strcpy(current_path, "");
        strcat(current_path, "[");
        strcat(current_path, current_user);
        strcat(current_path, "@ext2 /");
        current_dir = 1;
        fp = fopen("./FS.txt", "rb+");
        if (fp == NULL)
        {
            printf("The File system does not exist!\n");
            initialize_disk();
            return;
        }
        // 如果文件全部为空，需要重新初始化
        fseek(fp, 0, 0);
        char c;
```

```
int flag = 0;
while (!feof(fp))
{
    fread(&c, 1, 1, fp);
    if (c != 0)
    {
        flag = 1;
        break;
    }
}
if (flag == 0)
{
    printf("The File system does not exist!\n");
    initialize_disk();
    return;
}
reload_group_desc();
}
// 格式化
void format()
{
    initialize_disk();
    initialize_memory();
}
// 进入某个目录，实际上是改变当前路径
void cd(char tmp[100])
{
    unsigned short i, j, k, flag;
    flag = research_file(tmp, 2, &i, &j, &k);

    if (flag)
    {
        if (!strcmp(tmp, "..") && dir[k - 1].name_len) /* 到上一级目录且不是..目录 */
        {
            current_dir = i;
            current_path[strlen(current_path) - dir[k - 1].name_len - 1] =
'\0';
            current_dirlen = dir[k].name_len;
            // 修改访问时间
            reload_inode_entry(current_dir);
            time_t t;
            time(&t);
            inode_buff.i_atime = t;
            update_inode_entry(current_dir);
            return;
        }
        else if (!strcmp(tmp, "..") && !dir[k - 1].name_len) /* 到上一级目录且是..目录 */
        {
            return;
        }
        else if (!strcmp(tmp, "."))
        {
            // 修改访问时间
```

```
        current_dir = i;
        reload_inode_entry(current_dir);
        time_t t;
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(current_dir);
        return;
    }
    else if (!strcmp(tmp, "."))
    {
        // 修改访问时间
        current_dir = i;
        reload_inode_entry(current_dir);
        time_t t;
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(current_dir);
        return;
    }
}
// 以/为分隔符, 可以多级 cd
int length = 0;
int ii = 0;
char tmp2[100];
while (tmp[length] != '\0')
{
    // 以/为分隔符
    char p[1];
    p[0] = tmp[length];
    tmp2[ii] = tmp[length];
    tmp2[ii + 1] = '\0';
    printf("tmp[length] = %c\n", tmp[length]);
    printf("tmp2 = %s\n", tmp2);
    if (!strcmp(p, "/"))
    {
        tmp2[ii] = '\0';
        unsigned short i, j, k, flag;
        printf("tmp2 = %s\n", tmp2);
        flag = research_file(tmp2, 2, &i, &j, &k);
        if (flag)
        {
            current_dir = i;
            printf("current_dir: %d\n", current_dir);
            current_dirlen += strlen(tmp2);
            strcat(current_path, tmp2);
            strcat(current_path, "/");
            printf("current_path: %s\n", current_path);
            // 修改访问时间
            reload_inode_entry(current_dir);
            time_t t;
            time(&t);
            inode_buff.i_atime = t;
            update_inode_entry(current_dir);
        }
        else
        {

```

```
        printf("The directory %s not exists!\n", tmp2);
    }
    char tmp2[100];
    ii = -1;
}
else
{
    ;
}
length++;
ii++;
}
tmp2[ii] = '\0';
printf("tmp2 = %s\n", tmp2);
flag = research_file(tmp2, 2, &i, &j, &k);
if (flag)
{
    current_dir = i;
    current_dirlen += strlen(tmp2);
    strcat(current_path, tmp2);
    strcat(current_path, "/");
    // 修改访问时间
    reload_inode_entry(current_dir);
    time_t t;
    time(&t);
    inode_buff.i_atime = t;
    update_inode_entry(current_dir);
}
else
{
    printf("The directory %s not exists!\n", tmp2);
}
}
// 创建目录
void mkdir(char tmp[100], int 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 (!research_file(tmp, type, &i, &j, &k))
    { // 未找到同名文件
        if (inode_buff.i_size == 4096)
        { // 目录项已满
            printf("Directory has no room to be allocated!\n");
            return;
        }
        flag = 1;
        if (inode_buff.i_size != inode_buff.i_blocks * 512)
        { // 目录中有某些块中 32 个 dir_entry 未用
            i = 0;
            while (flag && i < inode_buff.i_blocks)
            {
                reload_dir(inode_buff.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_buff.i_block[i - 1]);
}
else
{ // 全满 新增加块
    inode_buff.i_block[inode_buff.i_blocks] = alloc_block();
    inode_buff.i_blocks++;
    reload_dir(inode_buff.i_block[inode_buff.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_buff.i_block[inode_buff.i_blocks - 1]);
}
inode_buff.i_size += 16;
update_inode_entry(current_dir);
// 为新增目录分配 dir_entry
dir_prepare(tmpno, strlen(tmp), type, tmp);
}
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[100], int type)
{
    unsigned short tmpno, i, j, k, flag;
    reload_inode_entry(current_dir);
    if (!research_file(tmp, type, &i, &j, &k))
    {
        if (inode_buff.i_size == 4096)
        {
            printf("Directory has no room to be allocated!\n");
            return;
        }
        flag = 1;
        if (inode_buff.i_size != inode_buff.i_blocks * 512)
```

```
{
    i = 0;
    while (flag && i < inode_buff.i_blocks)
    {
        reload_dir(inode_buff.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_buff.i_block[i - 1]);
}
else
{ // 分配一个新的数据块
    inode_buff.i_block[inode_buff.i_blocks] = alloc_block();
    inode_buff.i_blocks++;
    reload_dir(inode_buff.i_block[inode_buff.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);
    // 初始化新块其他项目为 0
    for (flag = 1; flag < 32; flag++)
    {
        dir[flag].inode = 0;
    }
    update_dir(inode_buff.i_block[inode_buff.i_blocks - 1]);
}
inode_buff.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, tmp);
}
else
{
    printf("File has already existed!\n");
}
}
// 删除目录
void rmdir(char tmp[100])
{
    unsigned short i, j, k, flag;
    unsigned short m, n;
    unsigned short tmp1 = current_dir;
    printf("initial tmp1: %d\n", current_dir);
```

```

if (!strcmp(tmp, "..") || !strcmp(tmp, "."))
{
    printf("The directory can not be deleted!\n");
    return;
}
flag = research_file(tmp, 2, &i, &j, &k);
unsigned short tmp2 = i;
if (flag)
{
    reload_inode_entry(dir[k].inode); // 加载要删除的节点
    if (inode_buff.i_size == 32)
    { // 只有.and ..
        inode_buff.i_size = 0;
        inode_buff.i_blocks = 0;
        remove_block(inode_buff.i_block[0]);
        // 更新 tmp 所在父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_buff.i_block[j]);
        remove_inode(dir[k].inode);
        dir[k].inode = 0;
        update_dir(inode_buff.i_block[j]);
        inode_buff.i_size -= 16;
        flag = 0;
        m = 1;
        while (flag < 32 && m < inode_buff.i_blocks)
        {
            flag = n = 0;
            reload_dir(inode_buff.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_buff.i_block[m]);
                inode_buff.i_blocks--;
                while (m < inode_buff.i_blocks)
                {
                    inode_buff.i_block[m] = inode_buff.i_block[m + 1];
                    ++m;
                }
            }
        }
        update_inode_entry(current_dir);
        printf("current_inode over: %d\n", current_dir);
        return;
    }
    else
    {
        // printf("%d\n",inode_buff.i_size);
    }
}

```

```

        for (int l = 0; l < inode_buff.i_blocks; l++)
        {
            reload_dir(inode_buff.i_block[l]);
            for (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)
                {
                    current_dir = i;
                    printf("2current_inode: %d\n", current_dir);
                    printf("dir[m].name: %s\n", dir[m].name);
                    rmdir(dir[m].name);
                }
                else if (dir[m].file_type == 1)
                {
                    current_dir = i;
                    //printf("1current_inode: %d\n", current_dir);
                    //printf("dir[m].name: %s\n", dir[m].name);
                    del(dir[m].name);
                    current_dir = i;
                }
            }
            if (inode_buff.i_size == 32)
            {
                current_dir = tmp1;
                //printf("the dic is empty \n");
                //printf("current_inode: %d\n", current_dir);
                rmdir(tmp);
            }
        }
        return;
    }
}
else
{
    printf("Directory to be deleted not exists!\n");
}
}
// 删除文件
void del(char tmp[100])
{
    unsigned short i, j, k, m, n, flag;
    m = 0;
    flag = research_file(tmp, 1, &i, &j, &k);
    if (flag)
    {
        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_buff.i_blocks)
    {
        remove_block(inode_buff.i_block[m++]);
    }
    inode_buff.i_blocks = 0;
    inode_buff.i_size = 0;
    remove_inode(i);
    // 更新父目录
    reload_inode_entry(current_dir);
    reload_dir(inode_buff.i_block[j]);
    dir[k].inode = 0; // 删除 inode 节点
    update_dir(inode_buff.i_block[j]);
    inode_buff.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_buff.i_blocks)
    {
        flag = n = 0;
        reload_dir(inode_buff.i_block[m]);
        while (n < 32)
        {
            if (!dir[n].inode)
            {
                flag++;
            }
            n++;
        }
        if (flag == 32)
        {
            remove_block(inode_buff.i_block[m]);
            inode_buff.i_blocks--;
            while (m < inode_buff.i_blocks)
            {
                inode_buff.i_block[m] = inode_buff.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[100])
{
    unsigned short flag, i, j, k;
    flag = research_file(tmp, 1, &i, &j, &k);
    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] = (short)dir[k].inode;
            // 更新文件的访问时间
            reload_inode_entry(dir[k].inode);
            time_t t;
            time(&t);
            inode_buff.i_atime = t;
            update_inode_entry(dir[k].inode);
            printf(" %s sucessfully opened!\n", tmp);
        }
    }
    else
        printf("The file %s does not exist!\n", tmp);
}

// 关闭文件
void close_file(char tmp[100])
{
    unsigned short flag, i, j, k;
    flag = research_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(" %s sucessfully 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[100])
{
    unsigned short flag, i, j, k, t;
```

```

flag = research_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_buff.i_mode & 4)) // i_mode:111b:读,写,执行
        {
            printf("The file %s can not be read!\n", tmp);
            return;
        }
        // 读文件直接索引
        if (inode_buff.i_blocks <= 6)
        {
            for (flag = 0; flag < inode_buff.i_blocks; flag++)
            {
                reload_block(inode_buff.i_block[flag]);
                for (t = 0; t < inode_buff.i_size - flag * 512; ++t)
                {
                    printf("%c", Buffer[t]);
                }
            }
        }
        // 读文件一级索引
        else if (inode_buff.i_blocks < 262)
        {
            printf("inode_buff.i_blocks: %d\n", inode_buff.i_blocks);
            for (flag = 0; flag < 6; flag++)
            {
                reload_block(inode_buff.i_block[flag]);
                // printf("\ninode_buff.i_block[flag]: %d\n",
inode_buff.i_block[flag]);
                for (t = 0; t < 512; ++t)
                {
                    printf("%c", Buffer[t]);
                }
            }
            reload_block(inode_buff.i_block[6]);
            for (flag = 0; flag < inode_buff.i_blocks - 7; flag++)
            {
                reload_block(inode_buff.i_block[6]);
                int t = Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1];
                // printf("\nBuffer[flag * 2] * 256 + Buffer[flag * 2 +
1]: %d\n", Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1]);
                reload_block(t);
                for (t = 0; t < 512; ++t)
                {
                    printf("%c", Buffer[t]);
                }
            }
        }
        // 读文件二级索引
        else if (inode_buff.i_blocks < 4072)
        {
            for (flag = 0; flag < 6; flag++)

```

```

        {
            reload_block(inode_buff.i_block[flag]);
            for (t = 0; t < 6 * 512; ++t)
            {
                printf("%c", Buffer[t]);
            }
        }
        reload_block(inode_buff.i_block[6]);
        for (flag = 0; flag < 256; flag++)
        {
            reload_block(Buffer[flag * 2] * 256 + Buffer[flag * 2 +
1]);
            for (t = 6 * 512; t < inode_buff.i_size - 6 * 512; ++t)
            {
                printf("%c", Buffer[t]);
            }
        }
        reload_block(inode_buff.i_block[7]);
        for (flag = 0; flag < inode_buff.i_blocks - 262; flag++)
        {
            reload_block(Buffer[flag * 2] * 256 + Buffer[flag * 2 +
1]);
            for (t = 0; t < inode_buff.i_size - (flag + 262) * 512;
++t)
            {
                printf("%c", Buffer[t]);
            }
        }
    }
    if (flag == 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);
}
// 文件以覆盖方式写入
void write_file(char tmp[100])
{
    unsigned short flag, i, j, k = 0, need_blocks;
    unsigned long size = 0, length;
    flag = research_file(tmp, 1, &i, &j, &k);
    if (flag)
    {
        if (search_file(dir[k].inode))
        {

```

```
reload_inode_entry(dir[k].inode);
if (!(inode_buff.i_mode & 2)) // i_mode:111b:读,写,执行
{
    printf("The file %s can not be wried!\n", tmp);
    return;
}
// fflush(stdin);
while (1)
{
    tempbuf[size] = getchar();
    if (tempbuf[size] == '#')
    {
        tempbuf[size] = '\0';
        break;
    }
    if (size >= 4096 * 512)
    {
        printf("Sorry,the max size of a file is 2MB!\n");
        break;
    }
    size++;
}
if (size >= 4096 * 512)
{
    length = 4096 * 512;
}
else
{
    length = strlen(tempbuf);
}
printf("\nlength = %d\n", length);
// 计算需要的数据块数目
need_blocks = length / 512;
if (length % 512)
{
    need_blocks++;
}
if (need_blocks < 6) // 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_buff.i_blocks <= need_blocks)
    {
        while (inode_buff.i_blocks < need_blocks)
        {
            inode_buff.i_block[inode_buff.i_blocks] =
alloc_block();
            inode_buff.i_blocks++;
        }
    }
    else
    {
        while (inode_buff.i_blocks > need_blocks)
        {
            remove_block(inode_buff.i_block[inode_buff.i_bloc
ks - 1]);
```

```

        inode_buff.i_blocks--;
    }
}
j = 0;
while (j < need_blocks)
{
    if (j != need_blocks - 1)
    {
        reload_block(inode_buff.i_block[j]);
        memcpy(Buffer, tempbuf + j * BLOCK_SIZE, BLOCK_SIZE);
        update_block(inode_buff.i_block[j]);
    }
    else
    {
        reload_block(inode_buff.i_block[j]);
        memcpy(Buffer, tempbuf + j * BLOCK_SIZE, length - j
* BLOCK_SIZE);

        inode_buff.i_size = length;
        update_block(inode_buff.i_block[j]);
    }
    j++;
}
update_inode_entry(dir[k].inode);
}
// 一级子索引 2 字节表示 1 个块号中, 如果一个数据块都用来存放块号, 则
可以存放 256 个
// 一级索引中大于 6 的块的块号放在 inode.i_block[6]指向的数据块中
else if (need_blocks < 262)
{
    inode_buff.i_size = length;
    printf("need_blocks = %d\n", need_blocks);
    if (inode_buff.i_blocks <= 6)
    {
        while (inode_buff.i_blocks < 6)
        {
            inode_buff.i_block[inode_buff.i_blocks] =
alloc_block();
            //printf("inode_buff.i_block[inode_buff.i_blocks]
= %d\n", inode_buff.i_block[inode_buff.i_blocks]);
            inode_buff.i_blocks++;
        }
        inode_buff.i_block[6] = alloc_block();
        inode_buff.i_blocks++;
    }
    // 写一级索引 2 字节表示 1 个块号
    reload_block(inode_buff.i_block[6]);
    // printf("inode_buff.i_block[6] = %d\n",
inode_buff.i_block[6]);
    for (j = 0; j < need_blocks - 6; j++)
    {
        short block_num = alloc_block();
        inode_buff.i_blocks++;
        Buffer[j * 2] = block_num / 256;
        Buffer[j * 2 + 1] = block_num % 256;
        //printf("Buffer[%d*2] = %d\n", j, (int)Buffer[j * 2]);

```

```

        //printf("Buffer[%d*2+1] = %d\n",j, (int)Buffer[j * 2
+ 1]);
    }
    update_block(inode_buff.i_block[6]);
    // 写数据块
    j = 0;
    while (j < 6)
    {
        reload_block(inode_buff.i_block[j]);
        memcpy(Buffer, tempbuf + j * BLOCK_SIZE, BLOCK_SIZE);
        update_block(inode_buff.i_block[j]);
        j++;
    }

    for (j = 0; j < need_blocks - 6; j++)
    {
        reload_block(inode_buff.i_block[6]);
        unsigned short block_num = (int)Buffer[j * 2] * 256 +
(int)Buffer[j * 2 + 1];
        //printf("Buffer[%d*2] = %d\n", j, (int)Buffer[j * 2]);
        //printf("Buffer[%d*2+1] = %d\n",j, (int)Buffer[j * 2
+ 1]);

        // printf("block_num = %d\n", block_num);
        reload_block(block_num);
        memcpy(Buffer, tempbuf + (j + 6) * BLOCK_SIZE,
BLOCK_SIZE);
        update_block(block_num);
    }
    update_inode_entry(dir[k].inode);
    reload_inode_entry(dir[k].inode);
    //printf("inode_buff.i_size = %d\n", inode_buff.i_size);
}
// 二级索引 2 字节表示 1 个块号中, 如果一个数据块都用来存放块号, 则可
以存放 256 个
else if (need_blocks < 4072)
{
    inode_buff.i_size = length;
    if (inode_buff.i_blocks <= 6)
    {
        while (inode_buff.i_blocks < 6)
        {
            inode_buff.i_block[inode_buff.i_blocks] =
alloc_block();
            inode_buff.i_blocks++;
        }
        inode_buff.i_block[6] = alloc_block();
        inode_buff.i_blocks++;
        inode_buff.i_block[7] = alloc_block();
        inode_buff.i_blocks++;
    }
    // 写一级索引 2 字节表示 1 个块号
    reload_block(inode_buff.i_block[6]);
    for (j = 0; j < 256; j++)
    {
        reload_block(inode_buff.i_block[6]);
        int block_num = alloc_block();

```

```

        inode_buff.i_blocks++;
        Buffer[j * 2] = block_num / 256;
        Buffer[j * 2 + 1] = block_num % 256;
    }
    update_block(inode_buff.i_block[6]);
    // 写二级索引 2 字节表示 1 个块号
    reload_block(inode_buff.i_block[7]);
    for (j = 0; j < (need_blocks - 262) / 256 + 1; j++)
    {
        reload_block(inode_buff.i_block[7]);
        int block_num = alloc_block();
        inode_buff.i_blocks++;
        Buffer[j * 2] = block_num / 256;
        Buffer[j * 2 + 1] = block_num % 256;
        update_block(inode_buff.i_block[7]);
        reload_block(block_num);
        for (k = 0; k < 256; k++)
        {
            reload_block(block_num);
            int block_num1 = alloc_block();
            inode_buff.i_blocks++;
            Buffer[k * 2] = block_num1 / 256;
            Buffer[k * 2 + 1] = block_num1 % 256;
            update_block(block_num);
        }
    }
    update_block(inode_buff.i_block[7]);
    // 写数据块
    j = 0;
    while (j < 6)
    {
        reload_block(inode_buff.i_block[j]);
        memcpy(Buffer, tempbuf + j * BLOCK_SIZE, BLOCK_SIZE);
        update_block(inode_buff.i_block[j]);
        j++;
    }
    reload_block(inode_buff.i_block[6]);
    for (j = 0; j < 256; j++)
    {
        reload_block(inode_buff.i_block[6]);
        unsigned short block_num = Buffer[j * 2] * 256 + Buffer[j
* 2 + 1];

        reload_block(block_num);
        memcpy(Buffer, tempbuf + (j + 6) * BLOCK_SIZE,
BLOCK_SIZE);

        update_block(block_num);
    }
    reload_block(inode_buff.i_block[7]);
    for (j = 0; j < (need_blocks - 262) / 256 + 1; j++)
    {
        reload_block(inode_buff.i_block[7]);
        unsigned short block_num = Buffer[j * 2] * 256 + Buffer[j
* 2 + 1];

        reload_block(block_num);
        for (k = 0; k < 256; k++)
        {

```



```
        reload_block(block_num);
        unsigned short block_num1 = Buffer[k * 2] * 256 +
Buffer[k * 2 + 1];
        reload_block(block_num1);
        memcpy(Buffer, tempbuf + (j + 262 + k) * BLOCK_SIZE,
BLOCK_SIZE);
        update_block(block_num1);
    }
}

    else
    {
        printf("Sorry,the max size of a file is 2MB!\n");
    }
}
else
{
    printf("The file %s has not opened!\n", tmp);
}
}
else
{
    printf("The file %s does not exist!\n", tmp);
}
}

// 查看目录下的内容
void ls()
{
    printf("items\t\ttype\t\t\tmode\t\t\ttc_time\t\t\tta_time\t\t\ttm_time\n");
    unsigned short i, j, k, flag;
    i = 0;
    reload_inode_entry(current_dir);
    while (i < inode_buff.i_blocks)
    {
        k = 0;
        reload_dir(inode_buff.i_block[i]);
        while (k < 32)
        {
            if (dir[k].inode != 0)
            {
                printf("%s\t\t", dir[k].name);
                if (dir[k].file_type == 2)
                {
                    j = 0;
                    reload_inode_entry(dir[k].inode);
                    if (!strcmp(dir[k].name, ".."))
                    {
                        flag = 1;
                    }
                    else if (!strcmp(dir[k].name, "."))
                    {
                        flag = 0;
                    }
                }
                else
            }
        }
    }
}
```

```
{
    flag = 2;
}
printf("<DIR>\t\t\t");
switch (inode_buff.i_mode & 7)
{
case 0:
    printf("____\t");
    break;
case 1:
    printf("____x\t");
    break;
case 2:
    printf("__w__\t");
    break;
case 3:
    printf("__w_x\t");
    break;
case 4:
    printf("r____\t");
    break;
case 5:
    printf("r____x\t");
    break;
case 6:
    printf("r_w__\t");
    break;
case 7:
    printf("r_w_x\t");
    break;
}
printf("\t ");
if (k >= 2)
{
    struct tm *t;
    t = localtime(&inode_buff.i_ctime);
    printf("%d.%d.%d %d:%d:%d\t ", t->tm_year + 1900,
t->tm_mon + 1, t->tm_mday, t->tm_hour,
t->tm_min, t->tm_sec);
    t = localtime(&inode_buff.i_atime);
    printf("%d.%d.%d %d:%d:%d\t ", t->tm_year + 1900,
t->tm_mon + 1, t->tm_mday, t->tm_hour,
t->tm_min, t->tm_sec);
    t = localtime(&inode_buff.i_mtime);
    printf("%d.%d.%d %d:%d:%d", t->tm_year + 1900,
t->tm_mon + 1, t->tm_mday, t->tm_hour,
t->tm_min, t->tm_sec);
}
}
else if (dir[k].file_type == 1)
{
    j = 0;
    reload_inode_entry(dir[k].inode);
    printf("<FILE>\t\t\t");
    switch (inode_buff.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("\t\t ");
    struct tm *t;
    t = localtime(&inode_buff.i_ctime);
    printf("%d.%d.%d %d:%d:%d\t ", t->tm_year + 1900,
t->tm_mon + 1, t->tm_mday, t->tm_hour,
            t->tm_min, t->tm_sec);
    t = localtime(&inode_buff.i_atime);
    printf("%d.%d.%d %d:%d:%d\t ", t->tm_year + 1900,
t->tm_mon + 1, t->tm_mday, t->tm_hour,
            t->tm_min, t->tm_sec);
    t = localtime(&inode_buff.i_mtime);
    printf("%d.%d.%d %d:%d:%d", t->tm_year + 1900, t->tm_mon
+ 1, t->tm_mday, t->tm_hour,
            t->tm_min, t->tm_sec);
    }
    printf("\n");
}
k++;
reload_inode_entry(current_dir);
}
i++;
}
}
// 修改文件权限
void chmod(char tmp[100], unsigned short mode)
{
    unsigned short flag, i, j, k;
    flag = research_file(tmp, 1, &i, &j, &k);
    if (flag)
    {
        if (mode < 0 || mode > 7)
        {
            printf("Wrong mode!\n");
            return;
        }
    }
}

```

```

    }
    reload_inode_entry(dir[k].inode);
    inode_buff.i_mode = mode;
    update_inode_entry(dir[k].inode);
}
else
    printf("The file %s does not exist!\n", tmp);
}
// 查看指令
void help()
{
    printf("=====help_list=====
=====\\n");
    printf("=====\\n");
    printf("1.cd : cd + path\\t\\t\\t2.mkdir : mkdir + dirname\\n");
    printf("3.rmdir : rmdir + dirname\\t\\t\\t4.ls : ls\\n");
    printf("5.create : create + filename\\t\\t\\t6.open : open + filename\\n");
    printf("7.close : close + filename\\t\\t\\t8.read : read + filename\\n");
    printf("9.write : write + filename\\t\\t\\t10.rm : rm + filename\\n");
    printf("11.chmod : chmod + filename + mode\\t\\t12.password : password +
tmp\\n");
    printf("13.format : format\\t\\t\\t14.quit : quit\\n");
    printf("=====
=====\\n");
}

```

程序 3: main.h

```

#ifndef EXT2_MAIN_H
#define EXT2_MAIN_H
extern char current_path[256];
extern char current_user[10];
extern void initialize_user(); //初始化用户
extern int login(char username[10], char password[10]); //用户登录
extern void initialize_memory(); //初始化内存
extern void format(); //格式化文件系统
extern void cd(char tmp[100]); //进入某个目录, 实际上是改变当前路径
extern void mkdir(char tmp[100], int type); //创建目录
extern void cat(char tmp[100], int type); //创建文件
extern void rmdir(char tmp[100]); //删除一个空目录
extern void del(char tmp[100]); //删除文件
extern void open_file(char tmp[100]); //打开文件
extern void close_file(char tmp[100]); //关闭文件
extern void read_file(char tmp[100]); //读文件内容
extern void write_file(char tmp[100]); //文件以覆盖方式写入
extern void ls(); //查看目录下的内容
extern void help(); //查看指令
extern void chmod(char tmp[100], unsigned short mode); //修改文件权限
extern void password_change(char username[10], char password[10]); //修改密码
#endif //EXT2_MAIN_H

```

程序 4: main.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "main.h"
#include "ext2.h"
int main()
{
    char command[10], temp[100];
    char username[10], password[10];
    initialize_user();
    while (1)
    {
        printf("Please print username : \n");
        scanf("%s", username);
        if (!strcmp(username, "quit") || !strcmp(username, "exit"))
            return 0;
        printf("Please print password : \n");
        scanf("%s", password);
        if (login(username, password))
        {
            strcpy(current_user, username);
            strcpy(current_path, "[");
            strcat(current_path, current_user);
            strcat(current_path, "@ext2 /");
            printf("User %s sign in!\n", username);
            break;
        }
        else
        {
            printf("User name or password wrong, please enter again!\n");
            printf("If want to exit, please enter \"quit\" or \"exit\"!\n");
        }
    }
    initialize_memory();
    while (1)
    {
        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, "create"))
```

```
{ // 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\n", temp);
    write_file(temp);
}
else if (!strcmp(command, "ls"))
{ // 显示当前目录
    ls();
}
else if (!strcmp(command, "format"))
{ // 格式化硬盘
    format();
}
else if (!strcmp(command, "help") || !strcmp(command, "h"))
{ // 查看帮助
    help();
}
else if (!strcmp(command, "quit") || !strcmp(command, "exit"))
{ // 退出系统
    printf("Good Bye!\n");
    break;
}
else if (!strcmp(command, "chmod"))
{ // 修改权限
    scanf("%s", temp);
    unsigned short mode;
```

```
        scanf("%hd", &mode);
        chmod(temp, mode);
    }
    else if (!strcmp(command, "password"))
    {
        scanf("%s", temp);
        printf("Please input your new password:\n");
        char password1[10];
        scanf("%s", password1);
        password_change(temp, password1);
    }
    else
    {
        printf("No this Command,Please check!\n");
        help();
    }
    getchar();
}
return 0;
}
```

Readme.md

类 EXT2 文件系统的设计

[类 EXT2 文件系统的设计](#)

[实验内容](#)

[实验步骤](#)

[数据结构定义](#)

[底层函数](#)

[命令层函数](#)

[shell 的设计](#)

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

实验步骤

- 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
- 实现包括分配数据块等底层操作
- 实现命令层函数，包括 dir 等操作
- 完成 shell 的设计

- 测试整个文件系统的功能

数据结构定义

4. // 组描述符

```
struct group_desc { // 32 B
    char bg_volume_name[16]; // 文件系统的卷名
    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
    char bg_pad[4]; // 填充(0xff)
};
```

// 索引结点

```
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(0~7), 最大为7
    unsigned long 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[24]; // 填充(0xff)
};
```

// 目录项

```
struct dir_entry { // 16 B
    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
};
```

// 用户信息

```
struct user {
    char username[10];
    char password[10];
}User[USER_MAX];
```


- 因为系统中最多有 4096 个数据块，所以索引节点最多也是 4096。所以索引节点号只需 16 位（unsigned short）即可。
- struct inode 中为保证一个索引节点占 64 位，需要提供 char i_pad[24]，作为填充。

底层函数

5. 读写缓冲区类操作函数

```
// 写 gdt
static void update_group_desc()
{
    fp = fopen("./FS.txt", "rb+");
    fseek(fp, GDT_START, SEEK_SET);
    fwrite(&gdt, GD_SIZE, 1, fp);
    fflush(fp);
}

// 读 gdt
static void reload_group_desc()
{
    fseek(fp, GDT_START, SEEK_SET);
    fread(&gdt, GD_SIZE, 1, fp);
}
```

update_group_desc 函数打开一个名为"FS.txt"的文件，以读写模式打开("rb+"), 然后将文件指针移动到 GDT 的起始位置 (GDT_START)，并将内存中的 GDT 数据 (gdt) 写入文件。最后，使用 fflush 函数刷新文件流，确保数据被写入文件。

reload_group_desc 函数假定文件指针 fp 已经打开，然后将文件指针移动到 GDT 的起始位置 (GDT_START)，并从文件中读取 GDT 数据到内存中的 gdt 变量中。

其他结构体缓冲区读写函数依次类推：但需注意何时用指针合适取地址

6. 分配删除块函数：标注位图

分配函数

```
// 分配 data_block
static int alloc_block()
{
    int flag = 0;
    if (gdt.bg_free_blocks_count == 0)
    {
        printf("There is no block to be allocated!\n");
        return (0);
    }
}
```

```

    reload_block_bitmap();
    for (int i = 0; i < 512; i++)
    {
        if (bitbuf[i] != 0xff)
        {
            for (int j = 0; j < 8; j++)
            {
                if ((bitbuf[i] & (1 << j)) == 0)
                {
                    bitbuf[i] |= (1 << j);
                    last_alloc_block = i * 8 + j;
                    break;
                }
            }
            break;
        }
    }

    update_block_bitmap();
    gdt.bg_free_blocks_count--;
    update_group_desc();
    return last_alloc_block;
}

```

循环遍历块位图的每个字节（共 512 个字节），然后再遍历每个字节中的每一位做与运算。如果某一位为 0，表示该数据块未被分配，则将该位设置为 1，表示分配了该数据块，并记录下该数据块的编号。最后跳出循环，返回分配的数据块编号。

删除块函数

```

// 删除 data_block
static void remove_block(unsigned short del_num)
{
    unsigned short tmp;
    tmp = del_num / 8;
    reload_block_bitmap();
    switch (del_num % 8)
    {
        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.bg_free_blocks_count++;
    update_group_desc();
}

```

首先计算出要删除的数据块编号在块位图中的位置。接下来，根据要删除的数据块编号在字节中的位置，使用 `switch` 语句来设置对应位为 0，表示释放该数据块。

7. 配置新节点及初始化目录

```

static void dir_init(unsigned short tmp, unsigned short len, int type, char name[100])
{
    reload_inode_entry(tmp);

    time_t Time;
    time(&Time);
    if (type == 2)
    { // dir
        inode_buff.i_size = 32;
        inode_buff.i_blocks = 1;
        inode_buff.i_block[0] = alloc_block();
        inode_buff.i_ctime = Time;
        inode_buff.i_mtime = Time;
        inode_buff.i_atime = Time;
        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_buff.i_block[0]);

inode_buff.i_mode = 6;
}
else
{
    inode_buff.i_size = 0;
    inode_buff.i_blocks = 0;
    inode_buff.i_mode = 6;
    inode_buff.i_ctime = Time;
    inode_buff.i_mtime = Time;
    inode_buff.i_atime = Time;
    int len = strlen(name);
    if (len < 4)
    {
        inode_buff.i_mode |= 1;
    }
    else
    {
        char *lastFour = &name[len - 4];
        if (strcmp(lastFour, ".exe") == 0 || strcmp(lastFour, ".
bin") == 0 || strcmp(lastFour, ".com") == 0)
        {
            inode_buff.i_mode |= 1;
        }
        else
        {
            ;
        }
    }
}
update_inode_entry(tmp);
}
```

函数根据类型 `type` 的值来初始化 `i` 节点的各个字段。如果类型为 2（表示目录），则设置 `i` 节点的字段，并初始化目录项 `dir` 的内容，并将 "." 和 ".." 目录项的名称赋值为当前目录和父目录的名称。

如果类型不为 2（表示文件），则设置 i 节点的大小、块数、创建时间、修改时间、访问时间等字段，并根据文件名的后缀(扩展名为.exe,.bin,.com 及不带扩展名的)来判断是否为可执行文件，如果是则设置 i 节点的执行权限。

命令层函数

8. cd 改变路径

```
[root@ext2 /]#mkdir 1
[root@ext2 /]#cd 1
[root@ext2 /1/]#ls
items      type          mode          c_time        a_time        m_time
.          <DIR>         r_w_
..         <DIR>         r_w_
[root@ext2 /1/]#
```

// 进入某个目录，实际上是改变当前路径

```
void cd(char tmp[100])
{
    unsigned short i, j, k, flag;
    flag = research_file(tmp, 2, &i, &j, &k);
    if (flag)
    {
        if (!strcmp(tmp, "..") && dir[k - 1].name_len)
        {
            current_dir = i;
            current_path[strlen(current_path) - dir[k - 1].name_len
- 1] = '\0';
            current_dirlen = dir[k].name_len;
            // 修改访问时间
            reload_inode_entry(current_dir);
            time_t t;
            time(&t);
            inode_buff.i_atime = t;
            update_inode_entry(current_dir);
            return;
        }
        else if (!strcmp(tmp, "..") && !dir[k - 1].name_len)
        {
            return;
        }
        else if (!strcmp(tmp, "."))
        {
            // 修改访问时间
            current_dir = i;
            reload_inode_entry(current_dir);
            time_t t;
```

```
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(current_dir);
        return;
    }
    else if (!strcmp(tmp, "."))
    {
        // 修改访问时间
        current_dir = i;
        reload_inode_entry(current_dir);
        time_t t;
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(current_dir);
        return;
    }
}
// 以/为分隔符, 可以多级cd
int length = 0;
int ii = 0;
char tmp2[100];
while (tmp[length] != '\0')
{
    // 以/为分隔符
    char p[1];
    p[0] = tmp[length];
    tmp2[ii] = tmp[length];
    tmp2[ii + 1] = '\0';
    // printf("tmp[length] = %c\n", tmp[length]);
    // printf("tmp2 = %s\n", tmp2);

    if (!strcmp(p, "/"))
    {
        tmp2[ii] = '\0';
        unsigned short i, j, k, flag;
        // printf("tmp2 = %s\n", tmp2);
        flag = research_file(tmp2, 2, &i, &j, &k);
        if (flag)
        {
            current_dir = i;

            // printf("current_dir: %d\n", current_dir);
            current_dirlen += strlen(tmp2);
            strcat(current_path, tmp2);
        }
    }
}
```

```
        strcat(current_path, "/");
        // printf("current_path: %s\n", current_path);
        // 修改访问时间
        reload_inode_entry(current_dir);
        time_t t;
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(current_dir);
    }
    else
    {
        printf("The directory %s not exists!\n", tmp2);
    }
    char tmp2[100];
    ii = -1;
}
else
{
    ;
}
length++;
ii++;
}
tmp2[ii] = '\0';
// printf("tmp2 = %s\n", tmp2);
flag = research_file(tmp2, 2, &i, &j, &k);
if (flag)
{
    current_dir = i;
    current_dirlen += strlen(tmp2);
    strcat(current_path, tmp2);
    strcat(current_path, "/");
    // 修改访问时间
    reload_inode_entry(current_dir);
    time_t t;
    time(&t);
    inode_buff.i_atime = t;
    update_inode_entry(current_dir);
}
else
{
    printf("The directory %s not exists!\n", tmp2);
}
}
```

cd 分为三种情况：主要考虑更改 `current_dir` 和 `current_path`

1. 传送到上一级目录 (..) 但不在根目录上
2. 在根目录上执行 `cd ..` (直接 return)
3. 传送到子目录 (考虑多级)：以/为分隔符,截取每一级的路径，搜索其索引节点及路径，不断循环。

```
[root@ext2 /]#cd 1/2/op
tmp[length] = 1
tmp2 = 1
tmp[length] = /
tmp2 = 1/
tmp2 = 1
current_dir: 2
current_path: [root@ext2 /1/
tmp[length] = 2
tmp2 = 2
tmp[length] = /
tmp2 = 2/
tmp2 = 2
current_dir: 3
current_path: [root@ext2 /1/2/
tmp[length] = 0
tmp2 = 0
tmp[length] = p
tmp2 = op
tmp2 = op
[root@ext2 /1/2/op/]#
```

该图展示了分割路径的路径，通过遍历路径寻找 “/” 分割每一层路径。

9. mkdir 创建目录

```
[root@ext2 /]#mkdir op
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>     r_w_
..         <DIR>     r_w_
op         <DIR>     r_w_      2023.11.28 20:43:43  2023.11.28 20:43:43  2023.11.28 20:43:43
```

// 创建目录

```
void mkdir(char tmp[100], int type)
{
    unsigned short tmpno, i, j, k, flag;
    reload_inode_entry(current_dir);
    if (!research_file(tmp, type, &i, &j, &k))
    {
        if (inode_buff.i_size == 4096)
        {
            printf("Directory has no room to be allocated!\n");
            return;
        }
        flag = 1;
        if (inode_buff.i_size != inode_buff.i_blocks * 512)
        {
            i = 0;
            while (flag && i < inode_buff.i_blocks)
            {
                reload_dir(inode_buff.i_block[i]);
                j = 0;
                while (j < 32)
                {
                    if (dir[j].inode == 0)
                    {
                        flag = 0;
                        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_buff.i_block[i - 1]);
        }
        else
```

```

    { // 全满 新增加块
        inode_buff.i_block[inode_buff.i_blocks] = alloc_block();
        inode_buff.i_blocks++;
        reload_dir(inode_buff.i_block[inode_buff.i_blocks - 1]);
        tmpno = dir[0].inode = get_inode();
        dir[0].name_len = strlen(tmp);
        dir[0].file_type = type;
        for (flag = 1; flag < 32; flag++)
        {
            dir[flag].inode = 0;
        }
        update_dir(inode_buff.i_block[inode_buff.i_blocks - 1]);
    }
    inode_buff.i_size += 16;
    update_inode_entry(current_dir);
    // 为新增目录分配 dir_entry
    dir_prepare(tmpno, strlen(tmp), type, tmp);
}
else
{
    printf("Directory has already existed!\n");
}
}

```

10. rmdir 删除目录

```

[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
list       <DIR>      r_w_      2023.11.28 20:15:32  2023.11.28 20:15:32  2023.11.28 20:15:32
[root@ext2 /]#rmdir list
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_

```

// 删除目录

```

void rmdir(char tmp[100])
{
    unsigned short i, j, k, flag;
    unsigned short m, n;
    unsigned short tmp1 = current_dir;
    if (!strcmp(tmp, "..") || !strcmp(tmp, "."))
    {
        printf("The directory can not be deleted!\n");
        return;
    }
    flag = research_file(tmp, 2, &i, &j, &k);
}

```

```

unsigned short tmp2 = i;
if (flag)
{
    reload_inode_entry(dir[k].inode); // 加载要删除的节点
    if (inode_buff.i_size == 32)
    { // 只有.and ..
        inode_buff.i_size = 0;
        inode_buff.i_blocks = 0;

        remove_block(inode_buff.i_block[0]);
        // 更新 tmp 所在父目录
        reload_inode_entry(current_dir);
        reload_dir(inode_buff.i_block[j]);
        remove_inode(dir[k].inode);
        dir[k].inode = 0;
        update_dir(inode_buff.i_block[j]);
        inode_buff.i_size -= 16;
        flag = 0;
        m = 1;
        while (flag < 32 && m < inode_buff.i_blocks)
        {
            flag = n = 0;
            reload_dir(inode_buff.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_buff.i_block[m]);
                inode_buff.i_blocks--;
                while (m < inode_buff.i_blocks)
                {
                    inode_buff.i_block[m] = inode_buff.i_block[m
+ 1];

                    ++m;
                }
            }
        }
    }
}

```

```

    }
    update_inode_entry(current_dir);
    return;
}
else
{
    for (int l = 0; l < inode_buff.i_blocks; l++)
    {
        reload_dir(inode_buff.i_block[l]);
        for (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)
            {
                current_dir = i;
                rmdir(dir[m].name);
            }
            else if (dir[m].file_type == 1)
            {
                current_dir = i;
                del(dir[m].name);
                current_dir = i;
            }
        }
        if (inode_buff.i_size == 32)
        {
            current_dir = tmp1;
            rmdir(tmp);
        }
    }
    return;
}
}
else
{
    printf("Directory to be deleted not exists!\n");
}
}

```

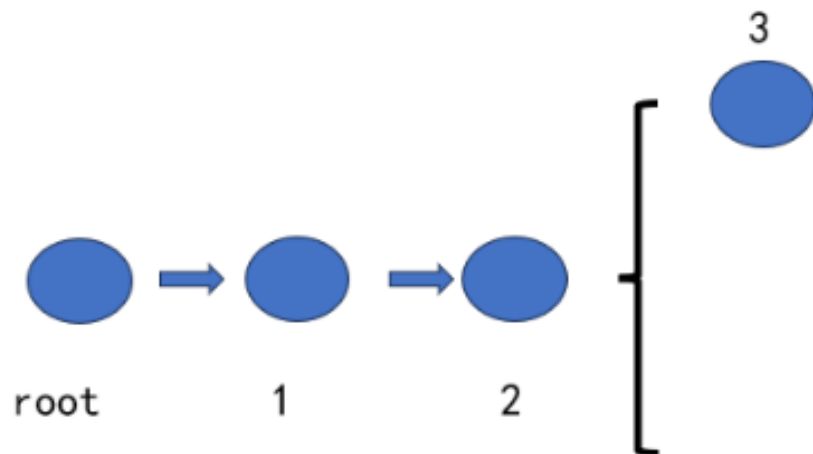
1. 首先检查要删除的目录是否为"."或"..", 如果是则无法删除, 否则继续执行。

2. 如果目录下只有"."和".."两个项，则直接删除该目录，更新父目录信息，并释放相关的数据块和节点。
3. 如果目录下还有其他文件或子目录，则递归调用 `rmdir` 函数，依次删除其中的文件和子目录。最后更新父目录信息。

在递归调用时要注意更新 `current_dir`，与当前删除的一级的目录相对应。

```
[root@ext2 /]#rmdir 1
initial tmp1: 1
2current_inode: 5
dir[m].name: 2
initial tmp1: 5
1current_inode: 6
dir[m].name: 3
the dic is empty
current_inode: 5
initial tmp1: 5
current_inode over: 5
the dic is empty
current_inode: 1
initial tmp1: 1
current_inode over: 1
```

图中表示的是 `current_dir` 的变化过程。该文件的构成为



有图中可知结点从根节点先层层递进到达目录 2, 删除文件 3 后(判断目录是否已空)再删除目录本身。从子节点再递归删除返回。

11. `open_file` 打开文件

```
[root@ext2 /]#create 1
[root@ext2 /]#open 1
1 sucessfully opened!
```

```
void open_file(char tmp[100])
{
    unsigned short flag, inode_num, block_num, dir_num;
    flag = research_file(tmp, 1, &inode_num, &block_num, &dir_num);
    if (flag)
    {
        if (search_file(dir[dir_num].inode))
        {
            printf("The file %s has opened!\n", tmp);
        }
        else
        {
            flag = 0;
            while (fopen_table[flag])
            {
                flag++;
            }
            fopen_table[flag] = (short)dir[dir_num].inode;
            // 更新文件的访问时间
            reload_inode_entry(dir[dir_num].inode);
        }
    }
}
```

```

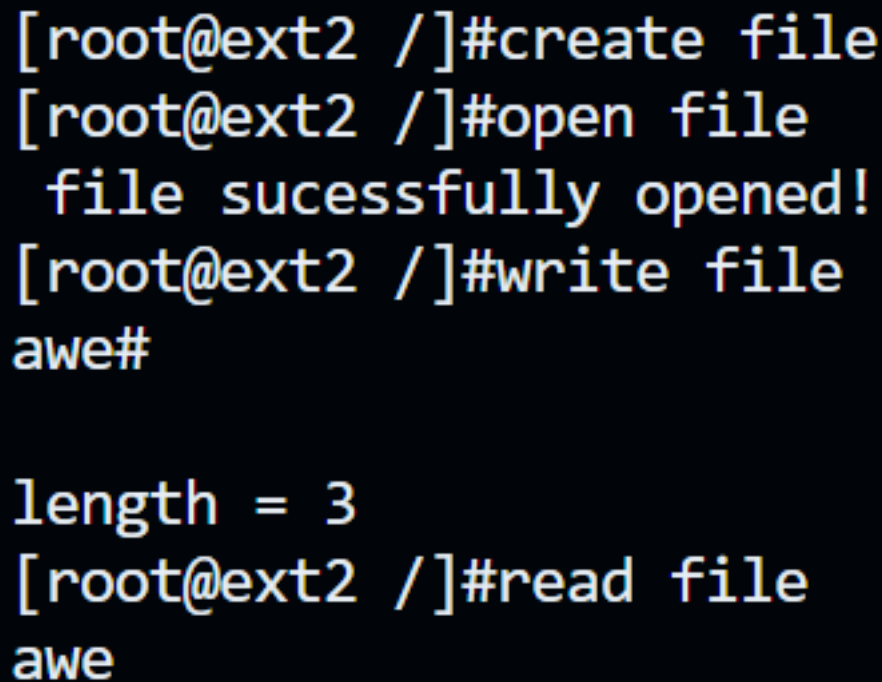
        time_t t;
        time(&t);
        inode_buff.i_atime = t;
        update_inode_entry(dir[dir_num].inode);

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

```

首先查找文件，如果找到了文件，则检查该文件是否已经被打开，如果已经被打开，则输出提示信息。如果文件没有被打开，则在 `fopen_table` 数组中找到一个空闲的位置，将该文件的 `inode` 号存入该位置。然后更新文件的访问时间。

12. read_file 读文件



```

[root@ext2 /]#create file
[root@ext2 /]#open file
file sucessfully opened!
[root@ext2 /]#write file
awe#

length = 3
[root@ext2 /]#read file
awe

```

```

void read_file(char tmp[100])
{
    unsigned short flag, i, j, k, t;
    flag = research_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_buff.i_mode & 4)) // i_mode:111b:读,写,执行
{
    printf("The file %s can not be read!\n", tmp);
    return;
}
// 读文件直接索引
if (inode_buff.i_blocks <= 6)
{
    for (flag = 0; flag < inode_buff.i_blocks; flag++)
    {
        reload_block(inode_buff.i_block[flag]);
        for (t = 0; t < inode_buff.i_size - flag * 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
}
// 读文件一级索引
else if (inode_buff.i_blocks < 262)
{
    printf("inode_buff.i_blocks: %d\n", inode_buff.i_blocks);
    for (flag = 0; flag < 6; flag++)
    {
        reload_block(inode_buff.i_block[flag]);
        // printf("\ninode_buff.i_block[flag]: %d\n", inode_buff.i_block[flag]);
        for (t = 0; t < 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
    reload_block(inode_buff.i_block[6]);
    for (flag = 0; flag < inode_buff.i_blocks - 7; flag++)
    {
        reload_block(inode_buff.i_block[6]);
        int t = Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1];
        // printf("\nBuffer[flag * 2] * 256 + Buffer[flag * 2 + 1]: %d\n", Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1]);
    }
}

```



```

        reload_block(t);
        for (t = 0; t < 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
}
// 读文件二级索引
else if (inode_buff.i_blocks < 4072)
{
    for (flag = 0; flag < 6; flag++)
    {
        reload_block(inode_buff.i_block[flag]);
        for (t = 0; t < 6 * 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
    reload_block(inode_buff.i_block[6]);
    for (flag = 0; flag < 256; flag++)
    {
        reload_block(inode_buff.i_block[6]);
        reload_block(Buffer[flag * 2] * 256 + Buffer[fla
g * 2 + 1]);
        for (t = 6 * 512; t < inode_buff.i_size - 6 * 512;
            ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
    reload_block(inode_buff.i_block[7]);
    for (flag = 0; flag < inode_buff.i_blocks - 262; fla
g++)
    {
        reload_block(inode_buff.i_block[7]);
        reload_block(Buffer[flag * 2] * 256 + Buffer[fla
g * 2 + 1]);
        for (t = 0; t < inode_buff.i_size - (flag + 262)
* 512; ++t)
        {
            printf("%c", Buffer[t]);
        }
    }
}
}

```

```

        if (flag == 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);
}

```

创建文件需满足两个条件：

1. 该文件是可读的 r
2. 该文件已打开 $open$

若满足条件即可读。读时首先读取 i_blcoks 的个数。根据数据块个数选择索引方式：

1. ($i_blcoks < 6$) 直接索引将数据块的位置信息存储在文件的 $inode$ 项中的 i_blocks 数组中。**每个数组元素对应一个数据块的位置**。直接读取 $i_blcok[]$ 数组中的块号，然后直接读取块号指向的数据块的内容。
2. ($i_blcoks < 262$) 一级子索引是一个额外的索引块，其中每个索引项指向一个数据块的位置。索引块中用 `unsigned int 16 位变量`，即 2 字节表示 1 个块号。索引块都用来存放块号。读取时若想获得一个块号需读取两个 `Buffer[]`，代表最终块号的高 8 位和低 8 位。

`int t = Buffer[flag * 2] * 256 + Buffer[flag * 2 + 1];` 读取对应块号的内容。

3. ($i_blcoks < 4072$) 读入二级索引块。代码通过循环读入一级索引块的块号，并在每个一级索引块中再次循环读入块号，将这些块的块号对应的数据块载入读取。

13. write_file 写文件

2023 年 12 月 7 日

```
tempbuf[size] = getchar();
if (tempbuf[size] == '#')
{
    tempbuf[size] = '\0';
    break;
}
if (size >= 4096 * 512)
{
    printf("Sorry,the max size of a file is 2MB!\n");
    break;
}
size++;
}
if (size >= 4096 * 512)
{
    length = 4096 * 512;
}
else
{
    length = strlen(tempbuf);
}
printf("\nlength = %d\n", length);
// 计算需要的数据块数目
need_blocks = length / 512;
if (length % 512)
{
    need_blocks++;
}
else if (need_blocks < 262)
{
    inode_buff.i_size = length;
    printf("need_blocks = %d\n", need_blocks);
    if (inode_buff.i_blocks <= 6)
    {
        while (inode_buff.i_blocks < 6)
        {
            inode_buff.i_block[inode_buff.i_blocks] = alloc_block();
            inode_buff.i_blocks++;
        }
        inode_buff.i_block[6] = alloc_block();
        inode_buff.i_blocks++;
    }
    // 写一级索引 2 字节表示 1 个块号
```

```

        reload_block(inode_buff.i_block[6]);
        printf("inode_buff.i_block[6] = %d\n", inode_buff.i_
block[6]);
        for (j = 0; j < need_blocks - 6; j++)
        {
            short block_num = alloc_block();
            inode_buff.i_blocks++;
            Buffer[j * 2] = block_num / 256;
            Buffer[j * 2 + 1] = block_num % 256;
        }
        update_block(inode_buff.i_block[6]);
        // 写数据块
        j = 0;
        while (j < 6)
        {
            reload_block(inode_buff.i_block[j]);
            memcpy(Buffer, tempbuf + j * BLOCK_SIZE, BLOCK_S
IZE);

            update_block(inode_buff.i_block[j]);
            j++;
        }

        for (j = 0; j < need_blocks - 6; j++)
        {
            reload_block(inode_buff.i_block[6]);
            unsigned short block_num = (int)Buffer[j * 2] * 2
56 + (int)Buffer[j * 2 + 1];
            reload_block(block_num);
            memcpy(Buffer, tempbuf + (j + 6) * BLOCK_SIZE, BL
OCK_SIZE);

            update_block(block_num);
        }
        update_inode_entry(dir[k].inode);
        reload_inode_entry(dir[k].inode);
    }

}

else
{
    printf("Sorry,the max size of a file is 2MB!\n");
}
}
else

```

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

创建文件需满足两个条件：

1. 该文件是可写的 *W*
2. 该文件已打开 *open*

若满足条件即可写入。写入时首先计算所需块数 *need_blocks*。根据索引个数选择索引方式：

1. (*need_blocks* < 6) 直接索引将数据块的位置信息存储在文件的 *inode* 项中的 *i_blocks* 数组中。**每个数组元素对应一个数据块的位置**。在代码中，通过更新 *inode* 项的 *i_blocks* 数组来分配或释放直接索引所需的数据块。然后，使用循环将数据块中的数据逐块写入磁盘。
2. (*need_blocks* < 262) 一级子索引是一个额外的索引块，其中存储的块号指向一个数据块的位置。索引块中用 **unsigned int 16 位变量**，即 2 字节表示 1 个块号。索引块都用来存放块号，可以存放 $512/2=256$ 个。因为 *Buffer* 为 *char* 型(8 位)，所以每得到一个数据块需要取两个 *Buffer[]* 代表高 8 位和低 8 位。

```
Buffer[j * 2] = block_num / 256;  
Buffer[j * 2 + 1] = block_num % 256;
```

3. (*need_blocks* < 4072) 写入二级索引块。代码通过循环分配一级索引块的块号，并在每个一级索引块中再次循环分配块号，将这些块的块号存储在二级索引块中的相应位置。写入数据块。代码通过循环读取 *tempbuf* 中的数据，并将其写入对应的数据块中。前 6 个数据块写入直接索引块，接下来的 256 个数据块写入一级索引块指向的块，剩余的数据块按照二级索引块的结构依次写入。

```
length = 3955  
need_blocks = 8  
inode_buff.i_block[inode_buff.i_blocks] = 1  
inode_buff.i_block[inode_buff.i_blocks] = 2  
inode_buff.i_block[inode_buff.i_blocks] = 3  
inode_buff.i_block[inode_buff.i_blocks] = 4  
inode_buff.i_block[inode_buff.i_blocks] = 5  
inode_buff.i_block[inode_buff.i_blocks] = 6  
inode_buff.i_block[6] = 7  
Buffer[0*2] = 0  
Buffer[0*2+1] = 8  
Buffer[1*2] = 0  
Buffer[1*2+1] = 9  
Buffer[0*2] = 0  
Buffer[0*2+1] = 8  
block_num = 8  
Buffer[1*2] = 0  
Buffer[1*2+1] = 9  
block_num = 9  
inode_buff.i_size = 3955
```

14. ls 显示指定工作目录下之内容

```
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>     r_w_
..         <DIR>     r_w_
1          <FILE>    r_w_x     2023.11.28 22:8:19  2023.11.28 22:8:19  2023.11.28 22:8:19
list       <DIR>     r_w       2023.11.28 22:8:26  2023.11.28 22:8:26  2023.11.28 22:8:26
```

15. chmod 更改文件权限

```
[root@ext2 /]#create file
[root@ext2 /]#ls
items      type              mode              c_time            a_time            m_time
.           <DIR>             r_w_
..          <DIR>             r_w_
op          <DIR>             r_w_              2023.11.29 13:49:41 2023.11.29 13:49:41 2023.11.29 13:49:41
file       <FILE>            r_w_              2023.11.29 13:52:2  2023.11.29 13:52:2  2023.11.29 13:52:2
[root@ext2 /]#chmod file 5
[root@ext2 /]#ls
items      type              mode              c_time            a_time            m_time
.           <DIR>             r_w_
..          <DIR>             r_w_
op          <DIR>             r_w_              2023.11.29 13:49:41 2023.11.29 13:49:41 2023.11.29 13:49:41
file       <FILE>            r_x               2023.11.29 13:52:2  2023.11.29 13:52:2  2023.11.29 13:52:2
```

```
void chmod(char tmp[100], unsigned short mode)
{
    unsigned short flag, i, j, k;
    flag = research_file(tmp, 1, &i, &j, &k);
    if (flag)
    {
        if (mode < 0 || mode > 7)
        {
            printf("Wrong mode!\n");
            return;
        }
        reload_inode_entry(dir[k].inode);
        inode_buff.i_mode = mode;
        update_inode_entry(dir[k].inode);
    }
}
```

```

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

```

检查权限模式是否在 0 到 7 之间，如果不在范围内则输出错误信息并返回。如果权限模式在范围内，则重新加载文件的索引节点信息，更新权限模式，并更新索引节点信息。如果文件不存在，则输出文件不存在的错误信息。

16. initialize_user 初始化用户 password_change 更改密码

```

PS C:\Users\邱子杰\Desktop\os\实验\3\code> gcc ext2.c main.c -o Fs
root
Please print password :
root
User root sign in!
[root@ext2 /]#password root
Please input your new password:
123123
[root@ext2 /]#quit
Good Bye!
PS C:\Users\邱子杰\Desktop\os\实验\3\code> gcc ext2.c main.c -o Fs
PS C:\Users\邱子杰\Desktop\os\实验\3\code> ./FS.exe
Please print username :
root
Please print password :
root
User name or password wrong, please enter again!
If want to exit, please enter "quit" or "exit"!
Please print username :
root
Please print password :
123123
User root sign in!
[root@ext2 /]#

```

// 初始化用户信息

```

void initialize_user()
{
    // 创建password.txt
    FILE *fp;
    fp = fopen("./password.txt", "r+");
    if (fp == NULL)
    {
        fp = fopen("./password.txt", "w+");
        // 初始化用户信息
        strcpy(User[0].username, "test");
        strcpy(User[0].password, "test");
    }
}

```



```
strcpy(User[1].username, "user");
strcpy(User[1].password, "user");

strcpy(User[2].username, "root");
strcpy(User[2].password, "root");

strcpy(User[3].username, "admin");
strcpy(User[3].password, "admin");
fwrite(User, sizeof(struct user), USER_MAX, fp);
printf("The password.txt has been created!\n");
fclose(fp);
}
// 读取password.txt
fp = fopen("./password.txt", "r+");
fread(User, sizeof(struct user), USER_MAX, fp);
fclose(fp);
return;
}

// 修改密码
void password_change(char username[10], char password[10])
{
    for (int i = 0; i < USER_MAX; i++)
    {
        if (!strcmp(User[i].username, username))
        {
            strcpy(User[i].password, password);
            fp = fopen("./password.txt", "w+");
            fwrite(User, sizeof(struct user), USER_MAX, fp);
            break;
        }
    }
    return;
}
```

这两个函数共同完成了用户信息的初始化和密码修改的功能。`initialize_user()`用于初始化用户信息并创建文件，而 `password_change()`用于修改特定用户的密码并更新文件

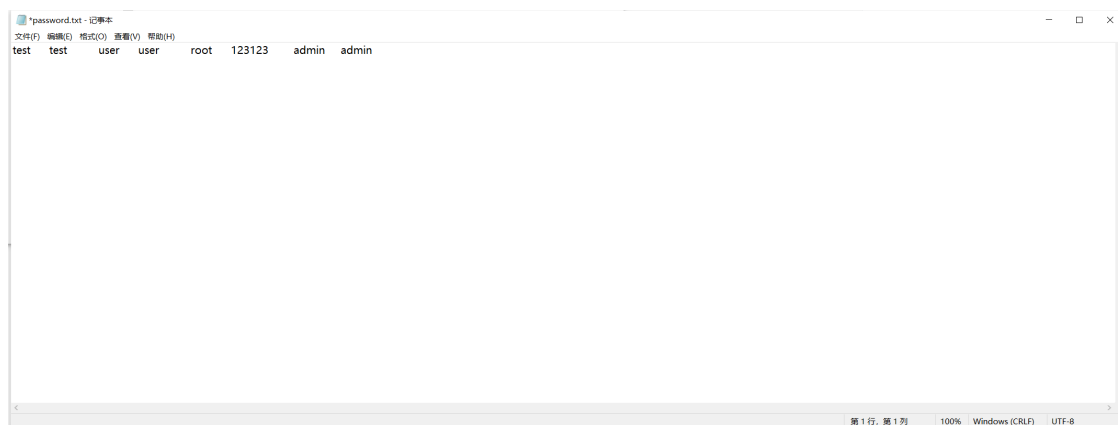
1. `initialize_user()`:

- 该函数用于初始化用户信息并创建/读取一个名为 `password.txt` 的文件。将预定义的用户信息写入 `User` 数组中。
- 接下来，使用 `fwrite` 将 `User` 数组中的用户信息写入文件中，并关闭文件。

- 最后，再次打开 `password.txt` 文件以读取其中的用户信息，并将其存储到 `User` 数组中

2. `password_change(char username[10], char password[10]):`

- 以写入模式重新打开 `password.txt` 文件，并使用 `fwrite` 将更新后的 `User` 数组写入文件。



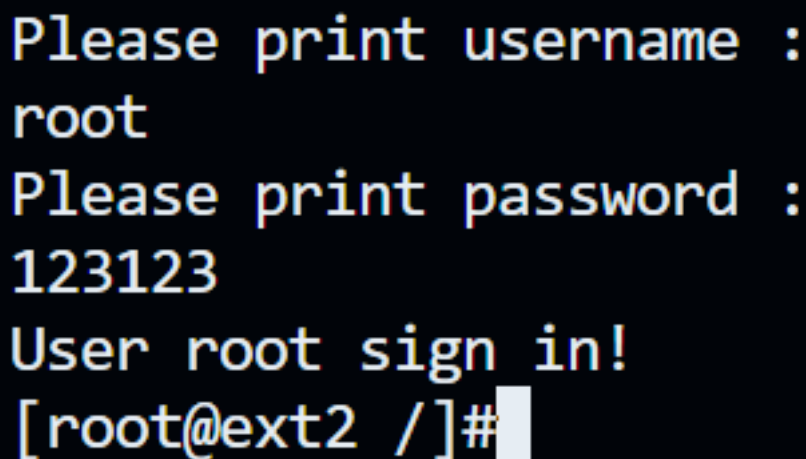
password.txt

shell 的设计

```
int main()
{
    char command[10], temp[100];
    char username[10], password[10];
    initialize_user();
    while (1)
    {
        printf("Please print username : \n");
        scanf("%s", username);
        if (!strcmp(username, "quit") || !strcmp(username, "exit"))
            return 0;
        printf("Please print password : \n");
        scanf("%s", password);
        if (login(username, password))
        {
            strcpy(current_user, username);
            strcpy(current_path, "[");
            strcat(current_path, current_user);
            strcat(current_path, "@ext2 /");
            printf("User %s sign in!\n", username);
            break;
        }
    }
}
```

```
    }  
    else  
    {  
        printf("User name or password wrong, please enter again!\n");  
        printf("If want to exit, please enter \"quit\" or \"exit\"!\n  
");  
    }  
}  
initialize_memory();  
while (1)  
{  
  
    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);  
    }  
}
```

1.shell 层进行用户登陆



A terminal window with a black background and white text. The text shows a login prompt: "Please print username :", followed by the input "root". Then another prompt: "Please print password :", followed by the input "123123". The next line shows "User root sign in!". The final line shows the shell prompt "[root@ext2 /]#" with a cursor at the end.

用户登陆界面

2.执行命令行（包括“help”查询操作用法）

```
[root@ext2 /]#help
=====help_list=====
=====
1.cd : cd + path
2.mkdir : mkdir + dirname
3.rmdir : rmdir + dirname
4.ls : ls
5.create : create + filename
6.open : open + filename
7.close : close + filename
8.read : read + filename
9.write : write + filename
10.rm : rm + filename
11.chmod : chmod + filename + mode
12.password : password + tmp
13.format : format
14.quit : quit
=====
```

3.退出文件系统

```
[root@ext2 /]#quit
Good Bye!
PS C:\Users\邱子杰\Desktop\os\实验\3\code>
```

4.格式化系统

```
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
[root@ext2 /]#create 1
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
1          <FILE>     r_w_x     2023.11.28 22:15:46  2023.11.28 22:15:46  2023.11.28 22:15:46
[root@ext2 /]#format
Creating the ext2 file system
The ext2 file system has been installed!
[root@ext2 /]#ls
items      type      mode      c_time      a_time      m_time
.          <DIR>      r_w_
..         <DIR>      r_w_
[root@ext2 /]#
```