

2023-11-30

操作系统课程设计报告

班 级: 计算机 2105 班

姓 名: 白 佳 兴

学 号: 2204311549



目 录

实验一 进程、线程相关编程实验.....	5
1.1 进程相关编程实验.....	5
1.1.1 实验目的	5
1.1.2 实验内容	5
1.1.3 实验思想	5
1.1.4 实验步骤	6
1.1.5 测试数据设计.....	6
1.1.6 程序运行初值及运行结果分析.....	6
1.1.7 实验总结	10
1.1.7.1 实验中的问题与解决过程.....	10
1.1.7.2 实验收获.....	11
1.1.7.3 意见与建议.....	11
1.2 线程相关编程实验.....	11
1.2.1 实验目的	11
1.2.2 实验内容	11
1.2.3 实验思想	12
1.2.4 实验步骤	12
1.2.5 测试数据设计.....	15
1.2.6 程序运行初值及运行结果分析.....	15
1.2.7 实验总结	16
1.2.7.1 实验中的问题与解决过程.....	16
1.2.7.2 实验收获.....	16
1.2.7.3 意见与建议.....	16
1.3 自选锁实验.....	16
1.3.1 实验目的	16
1.3.2 实验内容	16
1.3.3 实验思想	17
1.3.4 实验步骤	17

1.3.5 测试数据设计.....	20
1.3.6 程序运行初值及运行结果分析.....	20
1.3.7 实验总结	20
1.3.7.1 实验中的问题与解决过程.....	20
1.3.7.2 实验收获.....	21
1.3.7.3 意见与建议.....	22
实验二 进程通信与内存管理.....	22
2.1 进程的软中断通信.....	22
2.1.1 实验目的	22
2.1.2 实验内容	22
2.1.3 实验前准备	23
2.1.4 实验步骤	25
2.1.5 程序运行初值及运行结果分析.....	27
2.1.6 实验总结	28
2.1.6.1 实验中的问题与解决过程.....	28
2.1.6.2 实验收获.....	28
2.1.6.3 意见与建议.....	29
2.2 进程的管道通信.....	30
2.2.1 实验目的	30
2.2.2 实验内容	30
2.2.3 实验前准备	30
2.2.4 实验步骤	32
2.2.5 程序运行初值及运行结果分析.....	35
2.2.6 实验总结	36
2.2.6.1 实验中的问题与解决过程.....	36
2.2.6.2 实验收获.....	36
2.2.6.3 意见与建议.....	36
2.3 页面置换.....	37
2.3.1 实验目的	37
2.3.2 实验内容	37

2.3.3 实验前准备	37
2.3.4 实验步骤	39
2.3.5 程序运行初值及运行结果分析.....	45
2.3.6 实验总结	47
2.3.6.1 实验中的问题与解决过程.....	47
2.3.6.2 实验收获.....	47
2.3.6.3 意见与建议.....	47
 实验三 文件系统	 48
3.1 实验目的	48
3.2 实验内容	48
3.3 实验思想	48
3.4 实验步骤	48
3.5 程序运行初值及运行结果分析.....	79
3.6 实验总结	83
3.6.1 实验中的问题与解决过程.....	83
3.6.2 实验收获.....	83
3.6.3 意见与建议.....	83

实验 1.1 进程、线程相关编程实验

1.1.1 实验目的

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

1.1.2 实验内容

- (1) 熟悉操作命令、编辑、编译、运行程序。完成图 1-1 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。

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

int main()
{
    pid_t pid, pid1;

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

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

    return 0;
}
```

图 1-1 教材中所给代码（p103 作业 3.7）

- (2) 扩展图 1-1 的程序：
 - a) 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释；
 - b) 在 return 前增加对全局变量的操作并输出结果，观察并解释；
 - c) 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数

1.1.3 实验思想

- (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。

1.1.4 实验步骤

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

步骤一: 编写并多次运行图 1-1 中代码 (运行次数不限, 我们默认大家已经安装较新版本的 Linux 系统)。

```
[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33700parent: pid =33700parent: pid1 =33699[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33702parent: pid =33702parent: pid1 =33701[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33704parent: pid =33704parent: pid1 =33703[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33706parent: pid =33706parent: pid1 =33705[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33708parent: pid =33708parent: pid1 =33707[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33710parent: pid =33710parent: pid1 =33709[root@kp-test01 src]# |
```

步骤二: 删去图 1-1 代码中的 wait()函数并多次运行程序, 分析运行结果。

```
[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33894parent: pid =33894parent: pid1 =33893[root@kp-test01 src]# ./test
parent: pid =33897parent: pid1 =33896child: pid =0child: pid1 =33897[root@kp-test01 src]# ./test
parent: pid =33899parent: pid1 =33898child: pid =0child: pid1 =33899[root@kp-test01 src]# ./test
parent: pid =33901parent: pid1 =33900child: pid =0child: pid1 =33901[root@kp-test01 src]# ./test
parent: pid =33903parent: pid1 =33902child: pid =0child: pid1 =33903[root@kp-test01 src]# ./test
parent: pid =33905parent: pid1 =33904child: pid =0child: pid1 =33905[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33907parent: pid =33907parent: pid1 =33906[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33909parent: pid =33909parent: pid1 =33908[root@kp-test01 src]# ./test
parent: pid =33911parent: pid1 =33910child: pid =0child: pid1 =33911[root@kp-test01 src]# ./test
parent: pid =33913parent: pid1 =33912child: pid =0child: pid1 =33913[root@kp-test01 src]# ./test
parent: pid =33915parent: pid1 =33914child: pid =0child: pid1 =33915[root@kp-test01 src]# ./test
parent: pid =33917parent: pid1 =33916child: pid =0child: pid1 =33917[root@kp-test01 src]# ./test
parent: pid =33919parent: pid1 =33918child: pid =0child: pid1 =33919[root@kp-test01 src]# ./test
parent: pid =33921parent: pid1 =33920child: pid =0child: pid1 =33921[root@kp-test01 src]# ./test
parent: pid =33923parent: pid1 =33922child: pid =0child: pid1 =33923[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33925parent: pid =33925parent: pid1 =33924[root@kp-test01 src]# ./test
child: pid =0child: pid1 =33927parent: pid =33927parent: pid1 =33926[root@kp-test01 src]# ./test
```

分析: 父进程创建子进程后, 父子进程进入 if 语句执行各自的代码, 父进程和子进程的执行顺序没有先后之分, 完全随机, 所以在去掉 wait 后的多次运行结果中 child 和 parent 出现的先后顺序是随机的。

步骤三: 修改图 1-1 中代码, 增加一个全局变量增加全局变量 value, 在子进程中对 value 进行+1 操作, 在父进程中对 value 进行-1 操作, 观察并解释所做操作和输出结果。

```
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
[root@kp-test01 src]#
```

分析：在 if 语句的 else 部分设置 value--，在 else if(pid==0)部分设置 value++，在这两部分的 printf 中都输出 value 的值和地址，随后运行程序，child 中的 value 为 1，parent 中的 value 为 -1，且 child 中 value 的地址和 parent 中 value 的地址相同，这是因为父进程在创建子进程时子进程完全复制了一份父进程的数据，输出 value 的地址时输出的其实是 value 在进程中的虚拟地址，因为是复制的缘故，两个 value 在父进程和子进程中的虚拟地址是一样的，但是映射到内存中的物理地址是不同的，所以在一个进程中修改 value 不会影响到另一个进程中的 value

步骤四：在步骤三基础上，在 return 前增加对全局变量 value 进行 +5 操作并输出结果，观察并解释所做操作和输出结果。

```
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
  before return value=4,*value=0x420054
  before return value=6,*value=0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
  before return value=6,*value=0x420054
  before return value=4,*value=0x420054
[root@kp-test01 src]# ./test
child: value =1
parent: value =-1
child: *value =0x420054
parent: *value =0x420054
  before return value=6,*value=0x420054
  before return value=4,*value=0x420054
[root@kp-test01 src]# ./test
parent: value =-1
child: value =1
parent: *value =0x420054
child: *value =0x420054
  before return value=4,*value=0x420054
  before return value=6,*value=0x420054
```

分析：在 return 语句之前、if 语句之后添加一段对全局变量 value 的操作代码，父进程和子进程在执行完 if 语句之后都会执行这一段代码，实现对 value 的操作后打印出各自的 value 值，因为父进程和子进程的执行顺序是完全随机的，所以 child 和 parent 出现的先后顺序也无法确定，如截图中第一次执行是 child 在前，第二次执行是 parent 在前，第三次执行是 child 在前

步骤五：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。

子进程调用 system 函数：

```
[root@kp-test01 src]# ./test
parent process PID:3348
child process1 PID:3349
[root@kp-test01 src]# system_call PID:3350
child process PID:3349
```

分析：子进程调用 system 函数时执行了 system_call 文件，可以发现在执行 system_call 文件时进程的 pid 和子进程的 pid 并不一样，这是因为 system 函数是在原进程上创建了一个新的进程。system 函数执行完之后会返回原进程继续执行后续代码。

子进程调用exec函数

```
[root@kp-test01 src]# ./test
parent process PID:3359
child process1 PID:3360
[root@kp-test01 src]# system_call PID:3360
|
```

分析：子进程调用 `exec` 函数时执行了 `system_call` 文件，可以发现在执行 `system_call` 文件时进程的 `pid` 和子进程的 `pid` 是一样的，并且执行完 `exec` 函数之后并没有后续输出，说明子进程在执行 `exec` 函数时并没有创建新进程，并且执行 `exec` 函数后并没有执行后续的 `printf` 代码，这是因为 `exec` 是用新进程(命令)覆盖了原有的进程

1.1.5 测试数据设计

无需数据测试。

1.1.6 程序运行初值及运行结果分析

1.fork()不同返回值的含义

父进程调用 `fork()` 返回的是子进程的 `pid`(不为 0)，而产生的子进程返回的是 0，根据父子进程在 `fork()` 返回值上的差异可以采用 `if()` 语句进行分辨。

2.父子进程中的全局变量

父进程在创建子进程时子进程完全复制了一份父进程的数据，输出全局变量的地址时输出的其实是全局变量在进程中的虚拟地址，因为是复制的缘故，两个 `value` 在父进程和子进程中的虚拟地址是一样的，但是映射到内存中的物理地址是不同的

3.wait()的作用

等待子进程的终止：当父进程调用 `fork()` 创建子进程后，父进程可以使用 `wait()` 来等待子进程的终止。这样可以确保父进程不会在子进程之前退出，避免产生孤儿进程。

收集子进程的退出状态：`wait()` 用于收集子进程的退出状态。当子进程终止时，其退出状态会被保存，父进程通过 `wait()` 获取到这个退出状态。这包括子进程的退出码（返回给操作系统的值），以及是否正常退出等信息。

防止僵尸进程：当子进程终止后，其进程描述符会保留在系统进程表中，如果父进程不调用 `wait()` 来回收子进程的资源，子进程就会变成僵尸进程（zombie）。`wait()` 的调用就是为了回收这些已终止子进程的资源，防止产生僵尸进程。

4.孤儿进程

在操作系统中，孤儿进程是指其父进程先于它自己结束执行，导致孤儿进程成为系统的一个直接子进程。通常，当一个进程结束时，它的父进程会调用 `wait()` 或类似的系统调用来等待子进程的结束并收集其终止状态。然而，如果父进程在子进程之前结束，子进程将成为孤儿进程。

孤儿进程的生命周期不受其父进程的影响，而是由 init 进程（在 Unix 系统中通常是 PID 为 1 的进程）来接管。init 进程会周期性地调用 `wait()` 或类似的系统调用，以防止孤儿进程一直存在于系统中成为僵尸进程。

5. 并发进程的调度

先来先服务：按照进程到达的顺序进行调度。最先到达的进程先执行，直到它完成或被阻塞。FCFS 算法简单，但可能导致“等待时间过长”（Convoy Effect）的问题。

最短作业优先：选择估计运行时间最短的进程先执行。这可以减少平均等待时间，但需要预测每个进程的执行时间。

优先级调度：给每个进程分配一个优先级，优先级高的先执行。这样可以根据任务的紧急性或重要性进行调度。但可能导致低优先级的进程长时间等待。

轮转调度：将 CPU 时间切分成时间片（时间量），每个进程在一个时间片内执行。当时间片用完后，调度器将控制权交给下一个进程。这种方式确保每个进程都有机会执行，但可能导致上下文切换开销较大。

多级反馈队列调度：将就绪队列划分为多个队列，每个队列有不同的优先级。新到达的进程先进入高优先级队列，执行完一个时间片后，如果还未完成，就降低优先级并移到低优先级队列。这种方式在平衡了短作业和长作业的执行。

最高响应比优先：通过计算响应比（等待时间加服务时间的比率），选择响应比最高的进程先执行。这种算法综合考虑了等待时间和服务时间，倾向于执行长时间等待的进程。

多处理器调度：在多处理器系统中，有多个 CPU 可以同时执行不同的进程。调度算法需要考虑如何分配进程到不同的处理器上，以及如何有效地利用多个处理器。

6. system 与 exec 函数的不同

system 函数创建一个新的子进程，执行指定的 shell 命令，然后等待该子进程结束。exec 函数族并不创建新的进程，而是用新的程序替代了当前进程。原来的程序代码和数据都会被新程序替代，因此，后续代码不再执行。

1.1.7 实验总结

1.1.7.1 实验中的问题与解决过程

①在遇到 system 和 exce 函数时一头雾水，在看到 system 的介绍 `int system(const char *__command)` 时一直没弄明白这个 command 命令具体指代的是什么，后面看了具体实例后发现是需要执行文件所对应的目录

解决过程：查阅资料

[Linux 下使用 system \(\) 和 execv \(\) 实现对外部程序的调用 linux 采用 system 调用另一个进程-CSDN 博客](#)

②在完成实验一步骤三的过程中发现，父进程和子进程中全局变量 value 的地址是一

样的，可是对应的 `value` 的值却不相同，当时感到很不解，因为在以前一直都认为地址相同对应的是同一个量，后面发现 `printf("%p",&value);` 打印的只是 `value` 在进程中的虚拟地址，而相同的虚拟地址在不同的进程中会映射到不同的物理地址上面，子进程的创建是复制了父进程的虚拟地址空间的

解决过程：查阅资料

[父子进程能否共享全局变量 牛客博客 \(nowcoder.net\)](https://www.nowcoder.net)

1.1.7.2 实验收获

1. 进程管理理解深化：通过这个实验，更加深入地了解 Linux 系统中进程的创建、管理和调度。特别是通过观察 `wait()` 函数的行为，理解了父子进程间同步的重要性。
2. 编程技巧提升：这个实验让我熟悉了 linux 下的各种函数，尤其是涉及到系统级调用和进程管理的函数。对 `fork()`, `wait()`, `system()`, 和 `exec()` 等函数有了更深入的了解。
3. 系统调用与命令行工具：实验中涉及到 `system()` 和 `exec()` 系列函数，使我了解了如何在程序中执行系统命令，以及如何用 `exec()` 替换当前进程的执行内容。

1.1.7.3 意见与建议

1. 在文档中可以加入一些专门介绍某一个东西的介绍文档
2. 给出一些错误示范

实验 1.2 线程相关编程实验

1.2.1 实验目的

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

1.2.2 实验内容

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

察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

1.2.3 实验思想

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

(1) 线程创建与变量操作：首先，在一个进程内创建两个线程，并在进程内部初始化一个共享的变量。这两个线程将并发地对这个共享变量进行循环操作，执行不同的操作。

(2) 竞态条件和不稳定结果：由于线程并发执行，存在竞态条件，即两个线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。

(3) 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。

(4) 观察结果与比较：运行多次实验，观察使用互斥锁后的运行结果。应该可以发现，通过互斥锁的保护，不再出现不稳定的结果，每次运行得到的结果都是一致的。

(5) 调用系统函数和线程函数的比较：在任务一中，如果将调用系统函数和调用 exec 族函数改成在线程中实现，观察运行结果。可以发现，调用系统函数和 exec 族函数时，会输出进程的 PID（Process ID），而在线程中运行时，会输出线程的 TID（Thread ID）。这是因为线程是进程的子任务，它们共享进程的资源，但有自己的执行流程。

1.2.4 实验步骤

步骤一：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。

```
[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-283[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-1615[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:584[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:2300[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:3093[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-825[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-186[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-3334[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]#
```

分析：由于线程并发执行，存在竞态条件，即两个线程可能同时读取和修改共享变量。在没有适当的同步措施的情况下，不同线程的操作可能会交叉执行，导致结果不稳定，每次运行可能都会得到不同的结果。

比如对于一个初始化为 0 的共享变量 variable，子线程 1 对其进行+1 操作，子线程 2 对其进行-1 操作，其结果本来应该是 0，可是在实际运行的过程中，子线程 1 先将 variable（此时为 0）取出来，然后子线程 2 将 variable（此时为 0）取出来，然后子线程 1 完成+1 操作，将 1 写回 variable，最后子线程 2 完成-1 操作，将-1 写回 variable。最终 variable 的值为-1.而不是预期中的 0.

步骤二：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。

```
[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:0[root@kp-test01 src]# |
```

分析：通过设置信号量以及对信号量的 PV 操作使得任意一个时刻最多只有一个线程能够对 value 进行修改，避免了多线程之间的竞争，不管重复多少次，value 最终的值都是恒定为 0。

步骤三：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。

<pre>[root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281470756188640,pid=4518 thread2 creat success!!! thread2 tid=281470747734496,pid=4518 system_call PID:4521 system_call PID:4522 thread1 systemcall return thread2 systemcall return variable result:0[root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281473487270368,pid=4527 thread2 creat success!!! thread2 tid=281473478816224,pid=4527 system_call PID:4530 system_call PID:4531 thread1 systemcall return thread2 systemcall return variable result:0[root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281469944721888,pid=4533 thread2 creat success!!! thread2 tid=281469936267744,pid=4533 system_call PID:4536 system_call PID:4537 thread1 systemcall return thread2 systemcall return variable result:0[root@kp-test01 src]# </pre>	<pre>[root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281460938043872,pid=4547 thread2 creat success!!! thread2 tid=281460929589728,pid=4547 system_call PID:4547 [root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281470385713632,pid=4551 thread2 creat success!!! thread2 tid=281470377259488,pid=4551 system_call PID:4551 [root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281471225754080,pid=4555 thread2 creat success!!! thread2 tid=281471217299936,pid=4555 system_call PID:4555 [root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281457661243872,pid=4559 system_call PID:4559 [root@kp-test01 src]# ./test thread1 creat success!!! thread1 tid=281471947043296,pid=4563 thread2 creat success!!! thread2 tid=281471938589152,pid=4563 system_call PID:4563 [root@kp-test01 src]#</pre>
--	--

分析：左图是有 system 函数的线程运行的结果，右图是有 exec 函数的线程运行的结果，可以发

现线程执行 `system` 函数后会返回原线程并继续执行后续代码，而线程执行 `exec` 函数后直接结束，并不会继续执行后续代码，这是因为 `system` 函数是在原进程上开辟了一个新的进程，而 `exec` 函数是用新进程(命令)覆盖了原有的进程

1.2.5 测试数据设计

不需要测试数据

1.2.6 程序运行初值及运行结果分析

线程和进程的不同：

1. 定义：

进程 (Process)： 进程是程序的一次执行过程，是一个独立的运行环境。每个进程有自己的内存空间、代码、数据和系统资源。进程是系统资源分配和调度的基本单位。

线程 (Thread)： 线程是进程的一部分，是程序执行的最小单位。一个进程可以包含多个线程，它们共享相同的内存空间和系统资源，但有各自的执行流。

2. 资源占用：

进程： 进程是相对独立的，拥有自己的独立内存空间和资源。进程间通信需要特殊的机制，例如管道、消息队列等。

线程： 线程共享相同进程的地址空间和资源。它们之间的通信更容易，但也需要使用同步机制来避免竞态条件。

3. 切换开销：

进程： 进程切换开销较大，因为它涉及到上下文的切换，需要保存和恢复整个进程的状态。

线程： 线程切换开销相对较小，因为线程共享相同进程的地址空间，切换只涉及到寄存器的切换和栈的切换。

4. 创建销毁：

进程： 创建和销毁进程的开销较大，需要分配和释放独立的地址空间、资源等。

线程： 创建和销毁线程的开销较小，因为它们共享相同的地址空间和资源。

5. 同步与通信：

进程： 进程间通信相对复杂，需要使用特殊的 IPC (Inter-Process Communication) 机制。

线程： 线程之间通信更容易，但也需要使用同步机制来避免竞态条件，例如互斥锁、条件变量等。

6. 安全性：

进程： 进程是相对独立的，一个进程的崩溃不会影响其他进程。

线程： 一个线程的崩溃可能会导致整个进程崩溃，因为它们共享相同的地址空间。

7. 并发性：

进程：进程是独立运行的，各个进程之间互不干扰。

线程：线程共享相同的进程资源，因此更容易实现并发性。

1.2.7 实验总结

1.2.7.1 实验中的问题与解决过程

1. 编译的过程中遇到了 undefined reference to 'pthread_create'等错误。这个问题是因为在链接阶段没有加上-lpthread 标志，它会提醒编译器需要链接到 Pthreads 库。
2. 我分析了如何在多线程环境中使用 system()与 exec 族函数，并观察其对进程 ID 和线程 ID 的影响。注意到使用 system()调用的进程 ID 与主线程不同，这是因为 system()内部会使用 fork()创建一个新的子进程来执行命令。

1.2.7.2 实验收获

本次实验让我对多线程和多进程编程有了更深入的了解，特别是在 Linux 环境下使用 C 语言。掌握了如何创建和管理线程，以及如何使用信号量进行同步，学习 system()和 exec 族函数在多线程环境下的行为，这对于理解进程和线程的区别非常有帮助。

我还提高了问题解决能力。遇到编译错误、不一致的输出和程序终止问题时，都能够分析问题的根本原因，并找到相应的解决方案。

1.2.7.3 意见与建议

期望有更多的系统调用实例

实验 1.3 自选锁实验

1.3.1 实验目的

自旋锁作为一种并发控制机制，可以在特定情况下提高多线程程序的性能。本实验旨在通过设计一个多线程的实验环境，以及使用自旋锁来实现线程间的同步，从而实现以下目标：

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

1.3.2 实验内容

- (1) 在进程中给一变量赋初值并成功创建两个线程；

(2) 在两个线程中分别对此变量循环五千次以上做不同的操作（自行设计）并输出结果；

(3) 使用自旋锁实现互斥和同步；

1.3.3 实验思想

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。自旋锁的工作原理中关键的部分在于“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。这种方式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

1.3.4 实验步骤

步骤一：根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c，完整代码如下：

```
#include <stdio.h>
#include <pthread.h>
// 定义自旋锁结构体
typedef struct {
    int flag;
} spinlock_t;
// 初始化自旋锁
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}
// 获取自旋锁
void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
        // 自旋等待
    }
}
```

```
}

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

// 共享变量
int shared_value = 0;

// 线程函数
void *thread_function(void *arg) {
    spinlock_t *lock = (spinlock_t *)arg;

    for (int i = 0; i < 5000; ++i) {
        spinlock_lock(lock);
        shared_value++;
        spinlock_unlock(lock);
    }

    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    spinlock_t lock;
    // 输出共享变量的值
    printf("shared_value:%d\n", shared_value);
    // 初始化自旋锁
    spinlock_init(&lock);
    // 创建两个线程
    if(pthread_create(&thread1, &attr, thread_function, &lock);)
    { perror("Failed to create thread1");
      exit(1);
    }

    printf("thread1 create success!\n");
    if(pthread_create(&thread2, &attr, thread_function, &lock))
```

```
{ perror("Failed to create thread2");  
    exit(1);  
}  
  
printf("thread2 create success!\n");  
// 等待线程结束  
pthread_join(thread1, NULL);  
pthread_join(thread2, NULL);  
// 输出共享变量的值  
printf("shared_value:%d\n",shared_value);  
return 0;  
}
```

```
[root@kp-test01 src]# ./spinlock  
shared_value:0  
thread1 create success!  
thread2 create success!  
shared_value:10000  
[root@kp-test01 src]# ./spinlock  
shared_value:0  
thread1 create success!  
thread2 create success!  
shared_value:10000  
[root@kp-test01 src]# ./spinlock  
shared_value:0  
thread1 create success!  
thread2 create success!  
shared_value:10000  
[root@kp-test01 src]# ./spinlock  
shared_value:0  
thread1 create success!  
thread2 create success!  
shared_value:10000  
[root@kp-test01 src]# ./spinlock  
shared_value:0  
thread1 create success!  
thread2 create success!  
shared_value:10000
```

分析：全局变量 `shared_value` 初始值为 0，在线程调用全局变量 `shared_value` 的前后分别使用了自旋锁的获取函数和释放函数，且在调用这两个函数的时候不会被中断，以此来保证在任意一个时刻 `shared_value` 最多只能被一个线程所修改，这就消除了多线程的竞争状态，实现了多线程的同步，最终 `shared_value` 的值跟预期一样是 10000。

1.3.5 测试数据设计

无

1.3.6 程序运行初值及运行结果分析

如何实现线程间的同步与互斥：

1. 互斥锁：

互斥锁是最常见的同步机制，用于保护共享资源，确保在任何时刻只有一个线程能够访问该资源。在 C 语言中，可以使用 `pthread_mutex_t` 来声明互斥锁。

2. 信号量：

信号量用于控制对共享资源的访问。一个信号量可以被多个线程同时访问，但是对资源的实际访问需要通过信号量的计数进行控制。

3. 条件变量：

条件变量用于线程间的协作，一个线程可以在条件变量上等待，而另一个线程在满足某个条件时发送信号通知等待线程继续执行。

4. 读写锁：

读写锁允许多个线程同时读取共享资源，但只允许一个线程写入。这对于读操作频繁、写操作较少的情况可以提高性能。

1.3.7 实验总结

1.3.7.1 实验中的问题与解决过程

在给多线程程序实现互斥锁的时候，我一开始想到的是课本 6.6 节有关互斥锁实现的内容，简单实现之后运行程序，发现结果还是和没有互斥锁时的结果一样，这代表我的互斥锁是不合格的。

signal() 最初称为 V (荷兰语 verhogen, 增加)。可按如下来定义 wait():

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

可按如下来定义 signal():

```
signal(S) {
    S++;
}
```

在 wait() 和 signal() 操作中, 信号量整数值的修改应不可分割地执行。也就是说, 当一个进程修改信号量值时, 没有其他进程能够同时修改同一信号量的值。另外, 对于

```
[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-1315[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:150[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:5213[root@kp-test01 src]# ./test
thread1 creat success!!!
thread2 creat success!!!
variable result:-3211[root@kp-test01 src]# ./test
```

在查阅一些资料后发现我实现的这个互斥锁有可能在运行过程中被中断, 于是我选用了教材 6.9.4 节 Pthreads 同步章节所介绍的 unnamed semaphore 作为信号量, 最终成功解决问题。

1.3.7.2 实验收获

1. 理解并发问题: 自旋锁的使用通常涉及到解决并发访问共享资源的问题。在实验中, 我深入理解多线程并发执行可能导致的问题, 如竞态条件、死锁等。
2. 学习自旋锁的原理: 通过实验, 我更加深入地理解自旋锁的工作原理, 包括如何进行加锁和解锁, 以及它是如何防止多个线程同时访问临界区的。
3. 处理锁的争用: 实验中我遇到多个线程争用同一个锁的情况。这时, 我考虑如何有效地处理锁的争用, 以避免性能问题和竞争条件。
4. 学习并发编程技能: 通过实验, 我提高了自己的并发编程技能, 包括线程的创建、同步、互斥等方面。

1.3.7.3 意见与建议

1. 实验设计：设计一个简单而具有挑战性的多线程应用程序，其中多个线程需要访问共享资源。确保在程序中存在竞态条件，以便演示自旋锁的必要性。
2. 选择适当的场景：考虑一些实际的应用场景，例如生产者-消费者问题、读者-写者问题等。在这些场景中使用自旋锁，以了解它在解决并发问题时的效果。
3. 使用工具进行分析：在实验中，使用调试工具和性能分析工具，例如 GDB、Valgrind 等，帮助你识别潜在的问题，如死锁或数据竞争。
4. 性能比较：比较自旋锁和其他同步机制（如互斥锁）在性能方面的差异。了解在不同负载和并发程度下自旋锁的表现如何，以及它是否是解决你的问题的最佳选择。

实验 2.1 进程的软中断通信

2.1.1 实验目的

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

2.1.2 实验内容

(1) 使用 man 命令查看 fork、kill、signal、sleep、exit 系统调用的帮助手册。

(2) 根据流程图（如图 2.1 所示）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!!

注：delete 会向进程发送 SIGINT 信号，quit 会向进程发送 SIGQUIT 信号。ctrl+c 为 delete，ctrl+\ 为 quit。

参考资料 <https://blog.csdn.net/mylzh/article/details/38385739>

(3) 多次运行所写程序，比较 5s 内按下 Ctrl+\ 或 Ctrl+Delete 发送中断，或 5s 内不进行任何操作发送中断，分别会出现什么结果？分析原因。

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

2.1.3 实验前准备

实验相关 UNIX 系统调用介绍:

(1)fork(): 创建一个子进程。

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

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

(2)exec(): 装入并执行相应文件。

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

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

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

(5)getpid():获得进程号。

(6)lockf(files,function,size):用于锁定文件的某些段或整个文件。本函数适用的头文件为:

#include<unistd.h>,

参数定义: int lockf(files,function,size)

int files,function;

long size;

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

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

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

头文件为: #include <signal.h>

参数定义: signal(sig,function)

int sig;

void (*func) ();

其中 sig 共有 19 个值

注意: signal 函数会修改进程对特定信号的响应方式。

(9)pipe(fd);

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

其中 fd[1]是写端，向管道中写入，fd[0]是读端，从管道中读出。

(10)暂停一段时间 sleep;

调用 sleep 将在指定的时间 seconds 内挂起本进程。

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

(11)暂停并等待信号 pause;

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

其调用格式为“int pause(void);”

在 linux 系统下，我们可以输入 kill -l 来观察所有的信号以及对应的编号：

```
● [root@localhost ~]# kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD  18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN    22) SIGTTOU  23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS     34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
○ [root@localhost ~]#
```

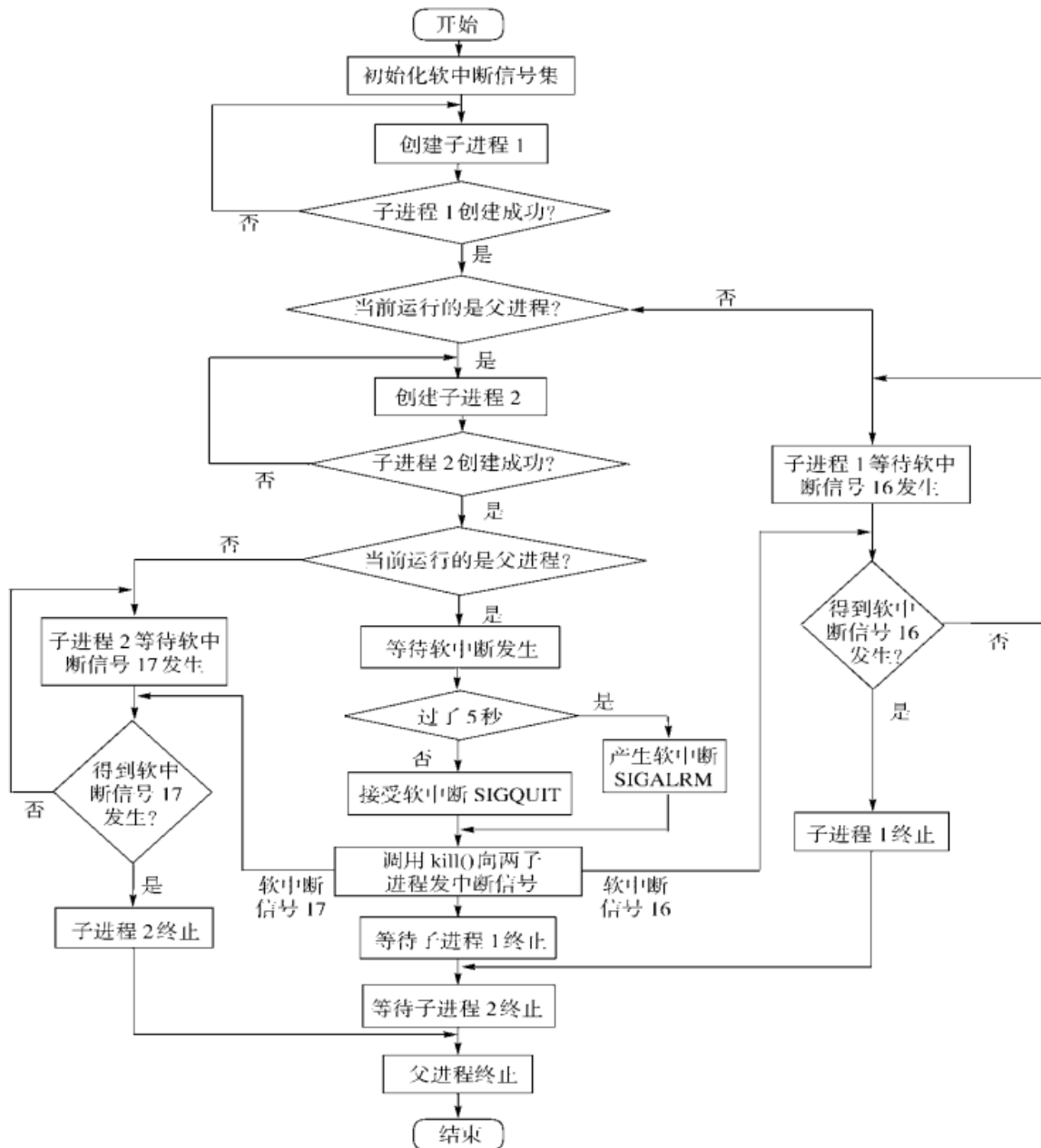



图 2.1 软中断通信程序流程图

2.1.4 实验步骤

编写代码：#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

int wait_flag=0;

void stop(int signum);

```
int main( ) {  
    int pid1, pid2;                                // 定义两个进程号变量  
    while((pid1 = fork( )) == -1);                  // 若创建子进程 1 不成功,则空循环  
    if(pid1 > 0) {                                    // 子进程创建成功,pid1 为进程号  
        while((pid2 = fork( )) == -1);            // 创建子进程 2  
        if(pid2 > 0) {  
wait_flag = 1;  
            signal(3,stop);  
            signal(2,stop);  
            signal(14,stop);  
            alarm(5);  
            while(wait_flag ==1);  
                kill(pid1,16);                      // 杀死进程 1 发中断号 16  
                kill(pid2,17);                      // 杀死进程 2  
                wait(0);                             // 等待第 1 个子进程 1 结束的信号  
                wait(0);                             // 等待第 2 个子进程 2 结束的信号  
                printf("\n Parent process is killed !!\n");  
                exit(0);                             // 父进程结束  
            }  
        else {  
            wait_flag = 1;  
            signal(17,stop);                          // 等待进程 2 被杀死的中断号 17  
            signal(2,SIG_IGN);  
            signal(3,SIG_IGN);  
            while(wait_flag);  
            printf("\n Child process 2 is killed by parent !!\n");  
            exit(0);  
        }  
    }  
    else {  
        wait_flag = 1;  
        signal(16,stop);                              // 等待进程 1 被杀死的中断号 16  
        signal(2,SIG_IGN);  
        signal(3,SIG_IGN);  
        while(wait_flag);  
    }  
}
```

```

    lockf(1,1,0);

    printf("\n Child process 1 is killed by parent !!\n");

    lockf(1,0,0);

    exit(0);
}
}

void stop( int signum) {

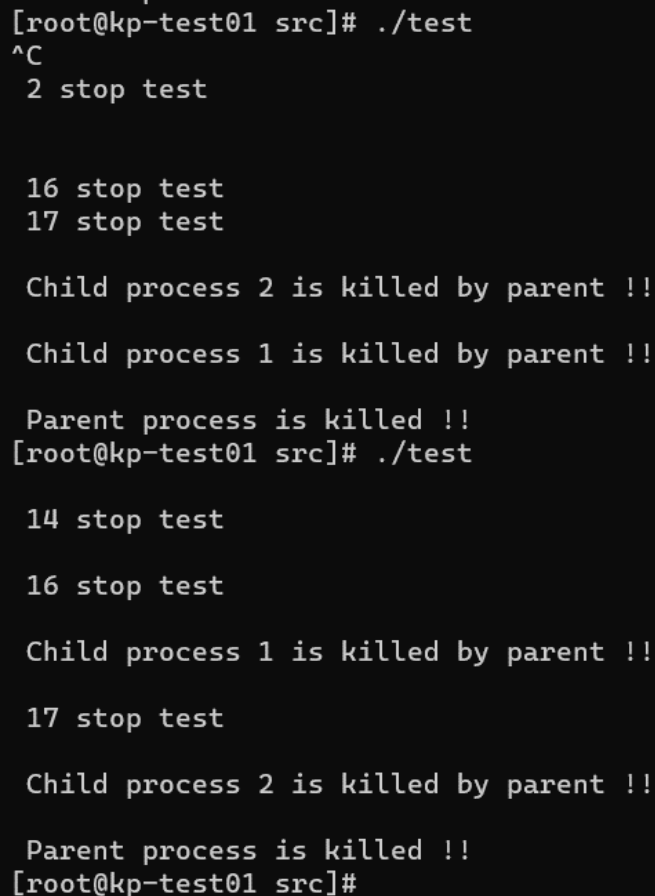
    wait_flag=0;

    printf("\n %d stop test \n",signum);

}

```

运行结果：



```

[root@kp-test01 src]# ./test
^C
2 stop test

16 stop test
17 stop test

Child process 2 is killed by parent !!
Child process 1 is killed by parent !!
Parent process is killed !!
[root@kp-test01 src]# ./test

14 stop test

16 stop test

Child process 1 is killed by parent !!

17 stop test

Child process 2 is killed by parent !!
Parent process is killed !!
[root@kp-test01 src]#

```

2.1.5 程序运行初值及运行结果分析

分析：创建两个子进程后在父进程中设置 `signal(3,stop);signal(2,stop); signal(14,stop);`；在 `stop` 函数中将 `wait_flag` 设置为 0，然后 `sleep(5)` 之后对 `wait_flag` 进行判断，如果 `wait_flag` 为 0，说明已经父进程已经收到 `SIGINT` 信号或者 `SIGQUIT` 信号，跳出 `if` 语句即可；如果 `wait_flag` 为 1，说明父进程还没有收到 `SIGINT` 信号或者 `SIGQUIT` 信号，此时需要给父进程发送一个 `SIGALARM` 信号，然后父进程调用 `signal(14,stop);`；`if` 语句结束后父进程给两个子进程分别发送信号 16 和信号 17，同时在两个子进程内设置 `signal(2,SIG_IGN); signal(3,SIG_IGN);` 来屏蔽 `SIGINT` 信号和 `SIGQUIT` 信号，子进程收到父进程的中断信号之后调用 `stop` 函数将 `wait_flag` 置 0，跳出了 `while(wait_flag);` 循环，打印之后调用 `exit(0)` 返回父进程。

2.1.6 实验总结

2.1.6.1 实验中的问题与解决过程

问题：不知道 signal 在程序中的作用，不知道怎么去设置那个 5 秒闹钟中断指令。

解决过程：查阅资料后得知 signal 是一个“异常”的函数，程序执行 signal 函数时如果对应的信号没有传递过来，那么程序会继续往下执行，如果在之后的某一个时刻 signal 函数对应的信号传递过来了那么程序会前往 signal 函数对应的函数指针参数来执行函数。

2.1.6.2 实验收获

我预想中的程序在 5s 内如果收到 SIGINT 信号或者 SIGQUIT 信号就会立刻调用 kill 函数，如果 5s 之内没有收到对应的信号，那么系统会自动给程序一个信号让程序调用 kill 函数。至于子进程 1 和子进程 2 执行的先后顺序则无法预测。实验结果与预期一致。

实际结果与预期一致，5s 之内如果有预期的信号则立刻调用 kill 函数，如果 5s 之内没有收到对应的信号，那么通过 alarm(5)给程序发送信号，进而让程序调用 kill 函数。

接收不同的中断

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

第一个调用 stop 函数的是中断信号，首先中断信号执行 stop 函数打印的结果就不一样，其次子进程 1 和子进程 2 的执行顺序也不一样

```
[root@kp-test01 src]# ./test
^C
2 stop test

16 stop test
17 stop test

Child process 2 is killed by parent !!

Child process 1 is killed by parent !!

Parent process is killed !!
[root@kp-test01 src]# ./test

14 stop test

16 stop test

Child process 1 is killed by parent !!

17 stop test

Child process 2 is killed by parent !!

Parent process is killed !!
[root@kp-test01 src]#
```

如果是 5s 内中断，那么中断信号会让父进程执行 stop 函数，直接将父进程的 wait_flag 置 0，进而让父进程跳出 while(wait_flag); 循环，执行后面的 kill 函数，同时打印中断信号的值。如果是 5s 后中断，说明在 5s 内一直没有对应的中断信号产生，此时 alarm(5) 会产生一个中断信号 14，同样的也会让父进程执行 stop 函数。改为闹钟中断后，不需要用户输入中断信号，过 5s 后打印的第一个信号就是 alarm(5) 产生的中断信号。

Kill 命令在程序中使用了两次，作用分别是父进程给子进程传递执行信号，子进程收到信号后执行 stop 函数，将子进程的 wait_flag 置 0，进而让子进程跳出 while(wait_flag); 循环，执行后面的 printf 函数。打印中断信号的值和自己已经收到信号的消息。

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

对于这个问题，我认为可以在进程内调用 exit(0)、return 等方式来退出。后者更好一些。kill 我理解的类似于 Windows 平台设备管理器的“结束进程”，或者说强制关机，这是一个具有风险的行为，尤其是对于易失性的信息，很可能直接丢失。因此 kill 不到万不得已不太应该使用。

2.1.6.3 意见与建议

验收等待时间太久，希望可以增加线上验收等验收形式

实验 2.2 进程的管道通信

2.2.1 实验目的

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

2.2.2 实验内容

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

(2) 根据流程图（如图 2.3 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

2.2.3 实验前准备

所谓“管道”，是指用于连接一个读进程和一个写进程以实现他们之间通信的一个共享文件，又名 pipe 文件。向管道(共享文件)提供输入的发送进程(即写进程)，以字符流形式将大量的数据送入管道；而接受管道输出的接收进程(即读进程)，则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。这种方式首创于 UNIX 系统，由于它能有效地传送大量数据，因而又被引入到许多其它操作系统中。

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

- ①互斥，即当一个进程正在对 pipe 执行读/写操作时，其它(另一)进程必须等待。
- ②同步，指当写(输入)进程把一定数量(如 4KB)的数据写入 pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。
- ③确定对方是否存在，只有确定了对方已存在时，才能进行通信。管道是进程间通信的一种简单易用的方法。管道分为匿名管道和命名管道两种。下面首先介绍匿名管道。

匿名管道只能用于父子进程之间的通信，它的创建使用系统调用 pipe()：

```
int pipe(int fd[2])
```

其中的参数 fd 用于描述管道的两端，其中 fd[0]是读端，fd[1]是写端。两个进

程分别使用读端和写端，就可以进行通信了。

一个父子进程间使用匿名管道通信的例子。

匿名管道只能用于父子进程之间的通信，而命名管道可以用于任何管道之间的通信。命名管道实际上就是一个 FIFO 文件，具有普通文件的所有性质，用 ls 命令也可以列表。但是，它只是一块内存缓冲区。

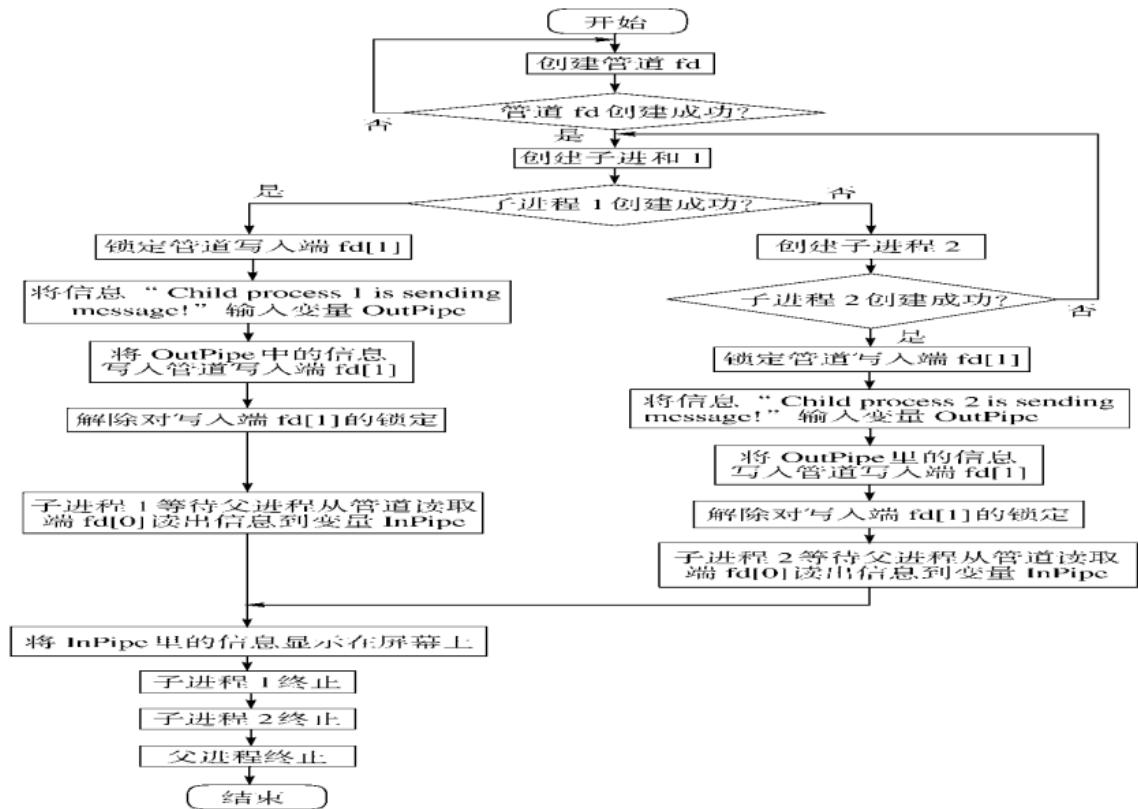


图 2.2 管道通信程序流程图

管道通信程序残缺版：

/*管道通信实验程序残缺版*/

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

```
#include <signal.h>
```

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

```
int pid1, pid2; // 定义两个进程变量
```

```
main() {
```

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

```
    char InPipe[1000]; // 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
    补充; // 锁定管道
    补充; // 分 2000 次每次向管道写入字符'1'
    sleep(5); // 等待读进程读出数据
    补充; // 解除管道的锁定
    exit(0); // 结束进程 1
}
else {
    while((pid2 = fork()) == -1); // 若进程 2 创建不成功,则空循环
    if(pid2 == 0) {
        lockf(fd[1],1,0);
        补充; // 分 2000 次每次向管道写入字符'2'
        sleep(5);
        lockf(fd[1],0,0);
        exit(0);
    }
    else {
        补充; // 等待子进程 1 结束
        wait(0); // 等待子进程 2 结束
        补充; // 从管道中读出 4000 个字符
        补充; // 加字符串结束符
        printf("%s\n",lnPipe); // 显示读出的数据
        exit(0); // 父进程结束
    }
}
```

其中两个进程各写入 2000 个字符，分有锁和无锁的情况。分别观察有锁和无锁情况下的写入情况。

2.2.4 实验步骤

运行代码：

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

```
#include <signal.h>
```



```
#include <unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
#include <sys/types.h>
#include<string.h>
int pid1,pid2; // 定义两个进程变量
int main( ){
    int fd[2],a;
    char OutPipe[100],InPipe[4100]; // 定义两个字符数组
    char c1='1',c2='2';
    pipe(fd); // 创建管道
    while((pid1 = fork( )) == -1); // 如果进程 1 创建不成功,则空循环
    if(pid1 == 0) { // 如果子进程 1 创建成功,pid1 为进程号
        lockf(fd[1],1,0); // 锁定管道
        sprintf(OutPipe,"\n Child process 1 is sending message!\n"); // 给 Outpipe 赋值
        write(fd[1],OutPipe,strlen(OutPipe));
        for(int i=0;i<2000;i++)
            write(fd[1],&c1,sizeof(c1));
        sleep(5);
        lockf(fd[1],0,0); // 解除管道的锁定
        exit(0); // 结束进程 1
    }
    else {
        while((pid2 = fork()) == -1); // 若进程 2 创建不成功,则空循环
        if(pid2 == 0) {
            lockf(fd[1],1,0);
            sprintf(OutPipe,"\n Child process 2 is sending message!\n");
            write(fd[1],OutPipe,strlen(OutPipe));
            for(int i=0;i<2000;i++)
                write(fd[1],&c2,1); // 向管道写入数据
            sleep(5);
            lockf(fd[1],0,0);
            exit(0);
        }
        else {
```

有锁:

无锁:

[illegible]

2.2.5 程序运行初值及运行结果分析

1. 预想中的结果：有锁的时候，先输出 2000 个 1，然后输出 2000 个 2。无锁的时候，1 和 2 随机输出，没有规律，无法预测。

2. 实际中的结果和预想中的结果几乎一致，当无锁的时候输出非常混乱，两个子进程的输出在父进程的输出中交错出现。由于没有进行同步和互斥控制，子进程 1 和子进程 2 在写入数据时，没有对管道进行互斥锁定，因此它们可以同时写入数据。而父进程也可以同时从管道中读取数据，父进程在读取数据时也没有进行同步操作，因此它可能在子进程 1 和子进程 2 还未完全写入数据时就开始读取。缺乏同步和互斥机制，导致进程之间的操作相互干扰，产生混乱的结果。导致数据混乱。

当有锁的时候，子进程 1 先上锁，将 2000 个 1 写到管道中，然后释放锁，接着子进程 2 上锁，将 2000 个 2 写到管道中。

3. 在这个实验中，使用了 **lockf** 函数对管道进行锁定和解锁的操作，实现了对管道通信的同步与互斥。**lockf(fd[1], 1, 0)**：锁定管道写入端，确保在一个进程写入时其他进程无法写入。**lockf(fd[1], 0, 0)**：解锁管道写入端，允许其他进程写入。这样做的目的是为了防止两个子进程同时向管道写入数据，造成混乱。

如果不控制同步与互斥，可能导致以下后果：

数据混乱：多个进程同时向管道写入数据，可能导致数据混在一起，难以区分哪个进程写入的哪部分数据。

读取不完整：父进程在子进程还未完全写入数据时就开始读取，可能导致读取不完整或不正确的数据。

程序崩溃：多个进程同时对管道进行读写，可能引发竞争条件，导致程序崩溃或产生未定义的行为。

2.2.6 实验总结

2.2.6.1 实验中的问题与解决过程

无

2.2.6.2 实验收获

1. 理解进程间通信的必要性：通过实验，我更好地理解为什么需要进程间通信以及在什么情况下使用管道是合适的。了解不同的 IPC 机制，例如管道、消息队列、共享内存等。
2. 学习管道的基本原理：通过实际编码和实验，更深入地理解管道的基本工作原理，包括创建管道、父子进程如何共享管道，以及管道的读写操作。
3. 处理同步和阻塞问题：在使用管道进行通信时，我学会如何确保数据的正确传输，如何处理阻塞等情况。
4. 解决进程间通信问题：在实验中，我学会使用调试工具和适当的编程技巧来解决这些问题，增加了对调试的认识。
5. 并发编程技能：通过进程间通信实验，我提高自己的并发编程技能，包括如何编写多进程程序、如何协调不同进程的活动等。

2.2.6.3 意见与建议

1. 实验目标明确：定义实验的明确目标，例如理解进程通信的基本概念、学会使用管道、处理进程同步等。确保让学生知道实验的目的是什么。
2. 提供清晰的实验说明：编写详细的实验说明，包括实验的背景、步骤、预期结果等。确保学生能够清楚地理解实验要求和完成步骤。
3. 逐步实施：建议学生逐步实施程序，先完成基本的功能，然后再逐步添加更复杂的部分。这有助于防止学生在实验的早期阶段就遇到太多问题。
4. 希望在实验过程中能用到更多的系统调用函数

实验 2.3 页面置换

2.3.1 实验目的

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。

2.3.2 实验内容

- (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 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

2.3.3 实验前准备

(1)FIFO 算法

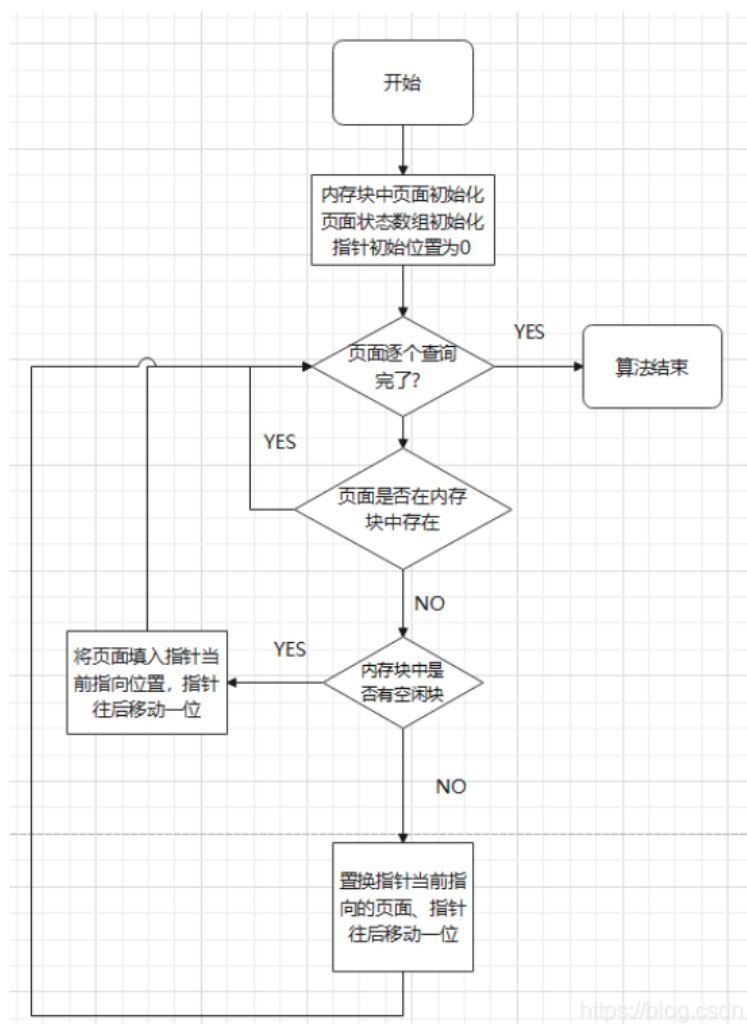


图 2.3 FIFO 算法流程图

在分配内存页面数（AP）小进程页面数（PP）时，当然是最先运行的 AP 个页面放入内存；这时又需要处理新的页面，则将原来放的内存中的 AP 个页中最先进入的调出（FIFO），再将新页面放入；总是淘汰在内存中停留时间最长的一页，即先进入内存的一页，先被替换出。

以后如果再有新页面需要调入，则都按上述规则进行。

算法特点：所使用的内存页面构成一个队列。

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)LRU 算法

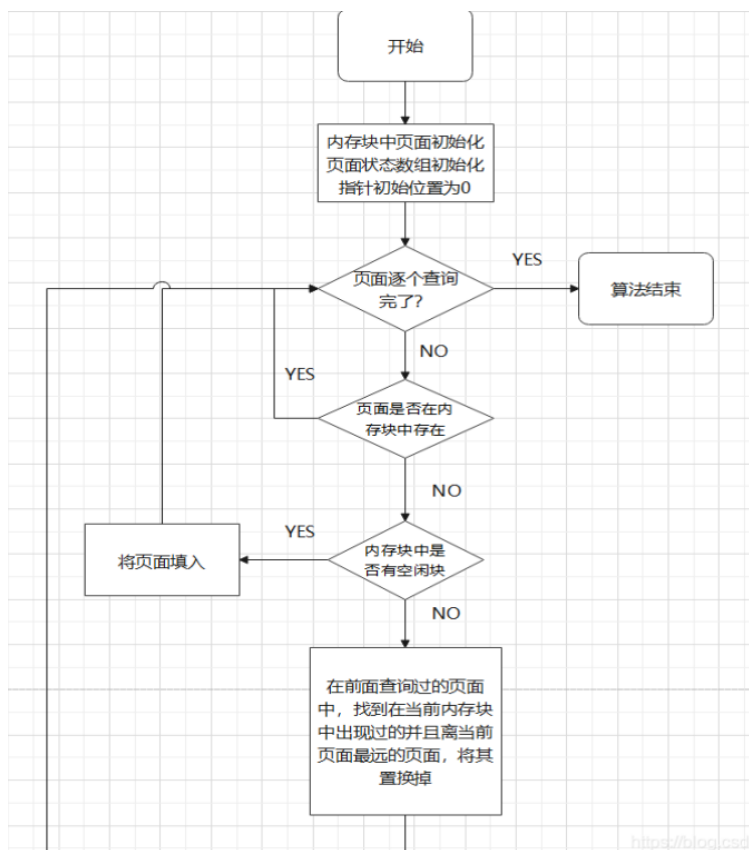


图 2.4 LRU 算法流程图

当内存分配页面数（AP）小于进程页面数（PP）时，把最先执行的 AP 个页面放入内存。

当需调页面进入内存，而当前分配的内存页面全部不空闲时，选择将其中最长时间没有用到的那一页调出，以空出内存来放置新调入的页面（LRU）。

算法特点：每个页面都有属性来表示有多长时间未被 CPU 使用的信息。

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，特别是在处理具有较好局部性的应用程序时效果良好。同时也讨论其可能的缺陷，如在面对长时间内不被使用的页面时性能下降。

2.3.4 实验步骤

编写代码：

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <time.h>
#define max 100000
typedef struct{
    // 1 2 3 4 1 2 5 1 2 3 4 5/12 个数据
    int pos;
    // 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1/20 个数
    int valid;
    int latest;
}S;
int main()
{
```

```
int total_instrucion=0; //页面数量
const int ap=20;           //内存分配页面数,随机生成的页面大小在 0 到 ap-1 之间

const int pp=3;           //进程页面数
int dise=0;
int front=0,rear=0,a;

int order[max];
S page[ap];
S pc[pp+1];
memset(page,0,sizeof(page));
memset(pc,0,sizeof(pc));
printf("手工输入请输入 1,随机输入请输入 0\n");
scanf("%d",&a);
if(a)    //1 表示手工输入页面序列, 0 表示随机输入页面序列
{
    printf("回车表示输入结束\n");
    scanf("%d",&order[0]);
    for(int i=1;getchar()!='\n';i++)    //手工输入, 回车作为结束符
    {scanf("%d",&order[i]);
        total_instrucion++;
    }
}
else {
    printf("请输入随机生成页面的个数: ");//定义随机生成的数组个数
    scanf("%d",&total_instrucion);
    srand((unsigned int)time(NULL));
    for(int i=0;i<total_instrucion;i++)    //随机输入
    {
        order[i]=rand()%ap;
        printf("%d ",order[i]);
    }
    printf("\n");
}
```



```
printf("FIFO:\n");
for(int i=0;i<total_instrucion;i++)
{
    if(page[order[i]].valid==0)
    if(front==(rear+1)%(pp+1))
    {
        pc[rear].pos=order[i];
        pc[rear].valid=1;
        page[order[i]].pos=rear;
        page[order[i]].valid=1;
        rear=(rear+1)%(pp+1);
        pc[front].valid=0;
        page[pc[front].pos].valid=0;
        front=(front+1)%(pp+1);
        dise++;
    }
    else {
        pc[rear].pos=order[i];
        pc[rear].valid=1;
        page[order[i]].pos=rear;
        page[order[i]].valid=1;
        rear=(rear+1)%(pp+1);
        dise++;
    }
    for(int j=front;j!=rear;j=(j+1)%(pp+1))
        printf("%d ",pc[j].pos);
    printf("\n");
}
printf("FIFO 的缺页率为:%f\n",(double)dise/total_instrucion);

memset(page,0,sizeof(page));
memset(pc,0,sizeof(pc));
dise=0;
int count=0;
printf("LRU:\n");
```

```
for(int i=0;i<total_instrucion;i++)
{
    if(page[order[i]].valid==0)
    {
        dise++;
        if(count<pp)
        {
            page[order[i]].pos=count;
            page[order[i]].valid=1;
            pc[count].pos=order[i];
            pc[count].valid=1;
            count++;
        }
        else{
            int temp=0x7fffffff,k;
            for(int j=0;j<pp;j++)
                if(pc[j].latest<temp)
                {
                    temp= pc[j].latest;
                    k=j;
                }
            page[pc[k].pos].valid=0;
            pc[k].valid=1;
            pc[k].pos=order[i];
            page[order[i]].valid=1;
            page[order[i]].pos=k;
        }
    }
    pc[page[order[i]].pos].latest=i;
    page[order[i]].latest=i;
    for(int j=0;j<pp&&pc[j].valid==1;j++)
        printf("%d ",pc[j].pos);
    printf("\n");
}
printf("LRU 的缺页率为:%f\n",(double)dise/total_instrucion);
```

}

```
[root@kp-test01 src]# ./test
0
3 0 2 2 3 6 3 1 6 3 0 0
FIFO:
3
3 0
3 0 2
3 0 2
3 0 2
3 0 2 6
3 0 2 6
0 2 6 1
0 2 6 1
2 6 1 3
6 1 3 0
6 1 3 0
0.583333
LRU:
3
3 0
3 0 2
3 0 2
3 0 2
3 0 2 6
3 0 2 6
3 1 2 6
3 1 2 6
3 1 2 6
3 1 0 6
3 1 0 6
0.500000
[root@kp-test01 src]#
```

```

● [root@localhost ~]# ./m
0
13 2 5 18 25 14 18 23 12 4 23 10 12 28 24
FIFO:
13
13 2
13 2 5
2 5 18
5 18 25
18 25 14
18 25 14
25 14 23
14 23 12
23 12 4
23 12 4
12 4 10
12 4 10
4 10 28
10 28 24
0.800000
LRU:
13
13 2
13 2 5
18 2 5
18 25 5
18 25 14
18 25 14
18 23 14
18 23 12
4 23 12
4 23 12
4 23 10
12 23 10
12 28 10
12 28 24
0.866667

```

FIFO 之 BLEADY 现象

```

[root@kp-test01 src]# gcc -o test test.c
[root@kp-test01 src]# ./test
1
1 2 3 4 1 2 5 1 2 3 4 5
1
1 2
1 2 3
2 3 4
3 4 1
4 1 2
1 2 5
1 2 5
1 2 5
2 5 3
5 3 4
5 3 4
0.750000
[root@kp-test01 src]# gcc -o test test.c
[root@kp-test01 src]# ./test
1
1 2 3 4 1 2 5 1 2 3 4 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4
1 2 3 4
2 3 4 5
3 4 5 1
4 5 1 2
5 1 2 3
1 2 3 4
2 3 4 5
0.833333
[root@kp-test01 src]#

```

分析：

1. S 结构：表示一个页面，具有 pos（在内存中的位置）、valid（页面是否有效）和 latest（最新访问时间）等属性。

page 数组：表示内存页面。

pc 数组：表示分配给进程的页面。

2. 常量：

max：数组大小的最大值。

ap：内存中的总页面数。

pp：每个进程的页面数。

3. 初始化：

使用零初始化数组 page 和 pc。

输入手动页面请求序列或生成随机序列。

4. FIFO 算法：

使用循环队列（pc）表示分配给进程的页面。

实现 FIFO 页面替换逻辑。

在每个页面请求后打印队列的状态。

计算并打印 FIFO 页面缺页率。

5. LRU 算法：

在 S 结构中使用额外的属性（latest）来存储最新的访问时间。

实现 LRU 页面替换逻辑。

在每个页面请求后打印分配的页面状态。

计算并打印 LRU 页面缺页率。

该程序使用循环队列来实现 FIFO，其中 front 和 rear 是队列前端和后端的指针。

对于 LRU，算法跟踪每个页面的最新访问时间，并在页面缺页时替换最近最少使用的页面。

该程序在每个页面请求后都会打印页面帧的状态，分别用于 FIFO 和 LRU。它还会打印两种算法的页面缺页率

2.3.5 程序运行初值及运行结果分析

①运行截图的前两张图表示在随机数生成页面序列的情况下，FIFO 和 LRU 两种页面置换算法的详细置换过程以及最终的缺页率。在第一张图中 FIFO 的缺页率差于 LRU 的缺页率，在第二张图中 FIFO 的缺页率优于 LRU 的缺页率，导致这种情况的根本原因是 LRU 的优秀缺页率是建立在页面序列满足一定的时间局部性和空间局部性的前提之上。由于页面序列是随机生成的，所以当页面序列不具有局部性的时候，LRU 算法可能还不如 FIFO 算法。但是程序在运行时用到的数据通常都具有一定的时间局部性和空间局部性，所以 LRU 算法很常用。

②在运行截图的最后一张图中，我们可以发现 FIFO 之 BLEADY 现象，同样的页面序列在内

存中的页面帧数增大的情况下缺页率不减反增，当发生页面缺页时，FIFO 算法会替换内存中最老的页面（存在时间最长的页面）。有了更多的页面帧，更多的页面会在内存中保留更长的时间。在某些情况下，增加页面帧数会导致本应被替换的页面在内存中停留的时间更长。结果是更多的页面缺页，因为老的页面被保留，而新的页面无法在可用的页面帧中找到位置。所以向系统添加更多资源（在这种情况下是页面帧）可能并不总是会带来预期的性能改善，而选择页面置换算法对结果起着至关重要的作用。

③在实现 LRU 的过程中，我在页面帧数数组 pc 中设置了一个 lastest 表示最近一次加入的时间，当需要选择牺牲页时，选择页面帧数数组 pc 中 lastest 最小的元素作为牺牲页，这样做使得越是最近加入数组的元素越晚被当成牺牲页，当页面序列具有局部性的时候，可以使得重复使用的数据尽可能的不会缺页。这种实现方式符合 LRU 算法的本质，即根据页面的最近使用情况进行页面替换，以更好地利用程序的局部性原理。

④在设计内存管理程序时,为了提高内存利用率可以采用如下策略：

1. **页面大小优化：**调整页面（页框）的大小，使之适应程序的访问模式。较小的页面大小可以减小内存碎片，而较大的页面大小可以降低页面表的开销。但是，过小或过大的页面大小都可能影响内存利用率，因此需要进行权衡。
2. **合理分配页面：**在设计内存管理算法时，考虑合理的页面分配策略，以确保每个进程都能够充分利用内存。考虑动态调整页面数量以适应工作负载的变化。
3. **内存压缩技术：**使用内存压缩技术来减小进程占用的内存空间。压缩技术可以通过压缩进程的内存页或者使用内存分页共享技术来实现。
4. **多进程合并：**对于多个相似的进程，尝试共享相同的内存页，而不是为每个进程分配独立的物理内存。这可以通过共享内存段或使用进程间通信机制来实现。
5. **内存映射文件：**利用内存映射文件的特性，将文件直接映射到进程的地址空间中，而无需实际加载整个文件到内存中。这可以减少内存的实际使用量。
6. **动态页面分配：**考虑使用动态页面分配机制，根据进程的实际需求动态调整分配的页面数量。这有助于避免过度分配或不足分配的问题。
7. **内存回收机制：**实现内存回收机制，及时释放不再需要的内存。这包括合理设计垃圾回收算法，释放被释放资源所占用的内存空间。
8. **智能页面置换策略：**选择合适的页面置换策略，例如 LRU、LFU 等，以最大程度地保留频繁使用的页面，减少不必要的页面置换，提高内存利用率。
9. **内存池管理：**使用内存池管理技术，将内存按照不同的大小分组管理。这有助于避免内存碎片问题，并提高内存的有效利用。
10. **优化数据结构：**使用紧凑的数据结构和算法，减小数据结构的内存占用，提高内存的有效利用率。

2.3.6 实验总结

2.3.6.1 实验中的问题与解决过程

问题：不知道采用什么样的数据结构来保存数据，不知道用什么数据结构来描述页面帧数

解决过程：在看完教材之后，我设置了一个数组来描述页面帧数，对于 FIFO 算法，我把这个数组当成循环队列来使用，队头所指的元素可以看成是在页面帧数中最早进入的元素，同时也当成了牺牲页，队尾所指的元素是待进入的元素。对于 LRU 算法，每次有元素进入数组的时候，都把数组对应元素的 latest 设置成最新值，在选择牺牲页的时候，将数组中 latest 最小的元素作为牺牲页。

2.3.6.2 实验收获

1. **理解页面置换算法的工作原理：** 实验能够帮助理解 FIFO 和 LRU 这两种常见的页面置换算法是如何工作的。了解它们的基本原理对于理解操作系统中的内存管理至关重要。
2. **观察不同算法的性能：** 通过实验，我观察并比较 FIFO 和 LRU 在处理页面置换时的性能差异。这包括页面缺页率、页面置换次数等指标。
3. **熟悉实际应用场景：** 在实验中，我使用了模拟的页面请求序列。通过分析这些序列，我可以更好地理解在实际应用中不同页面置换算法的表现。
4. **了解 Bélády's Anomaly：** 如果在实验中使用了不同数量的页面帧，我注意到了 Bélády's Anomaly 的现象，即增加页面帧数反而导致更多的页面缺页。这能够帮助理解算法的一些特殊性质。
5. **思考最佳实践：** 通过实验，我思考在不同情境下选择何种页面置换算法更为合适。不同的应用场景可能需要不同的算法来优化性能。

2.3.6.3 意见与建议

1. **深入分析实验结果：** 不仅仅观察实验结果，还应该深入分析不同算法在不同条件下的表现。考虑页面缺页率、页面置换次数等指标，并尝试理解结果背后的原因。
2. **尝试不同的输入序列：** 使用不同的页面请求序列进行实验，以更全面地了解算法在不同情境下的表现。这有助于更好地理解算法的鲁棒性和适用性。
3. **了解其他页面置换算法：** 除了 FIFO 和 LRU，还有其他页面置换算法，如最不经常使用 (LFU) 等。尝试比较不同算法的优缺点，以及它们在不同场景下的适用性。

实验三 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功能及内部实现，为提出更好的文件系统做好准备。掌握文件系统的工作原理。理解文件系统的主要数据结构。学习较为复杂的 Linux 下的编程。了解 EXT2 文件系统的结构

3.2 实验内容

在分析 Linux 的文件系统的基础上，设计实现一个完全类似于 Ext 文件系统的虚拟多级文件系统。

(1) 分析 EXT2 文件系统的结构。

(2) 基于 Ext2 的思想和算法，设计一个类 Ext2 的多级文件系统，实现 Ext2 文件系统的功能子集。

(3) 用现有操作系统上的文件来代替硬盘进行硬件模拟。

(4) 可以实现下列几条命令：

login,create ,delete ,cd ,close ,read ,write ,format ,password,pwd,ls

(5) 列目录时要列出文件类型、文件名、创建时间、最后访问时间和修改时间。

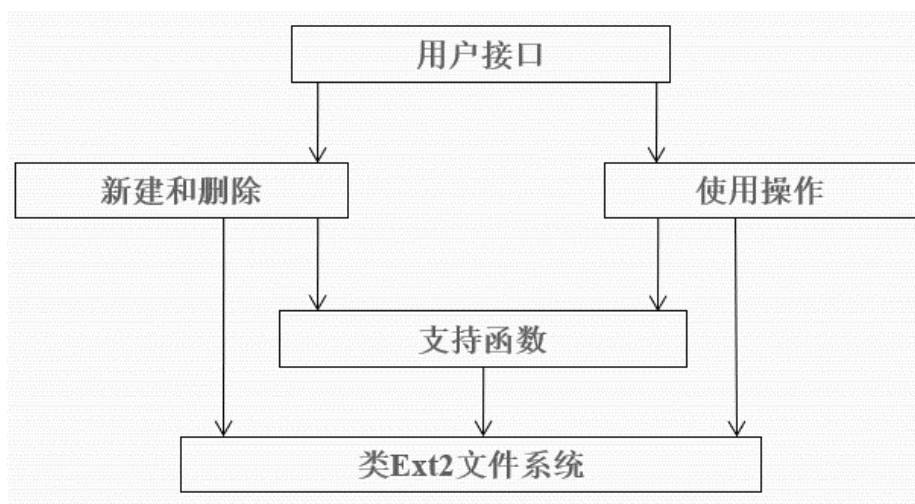
(6) 源文件可以进行读写保护。

3.3 实验思想

在分析 Linux 的文件系统的基础上，基于 Ext2 的思想和算法，设计一个类 Ext2 的虚拟多级文件系统，实现 Ext2 文件系统的功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。设计文件系统应该考虑的几个层次：①介质的物理结构；②物理操作——设备驱动程序完成；③文件系统的组织结构（逻辑组织结构）；④对组织结构其上的操作；⑤为用户使用文件系统提供的接口。

3.4 实验步骤

如图为文件系统架构图



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

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

```
typedef struct ext2_group_desc //组描述符 68 字节
{
    char bg_volume_name[16]; //卷名
    int bg_block_bitmap; //保存块位图的块号
    int bg_inode_bitmap; //保存索引结点位图的块号
    int bg_inode_table; //索引结点表的起始块号
    int bg_free_blocks_count; //本组空闲块的个数
    int bg_free_inodes_count; //本组空闲索引结点的个数
    int bg_used_dirs_count; //本组目录的个数
    char psw[16]; //password
    char bg_pad[24]; //填充(0xff)
} ext2_group_desc;
```

由于容量已经确定，文件最大即为 2MB。需要 4096 个数据块。索引结点的数据结构中，仍然采用多级索引机制。由于文件系统总块数必然小于 4096×2 ，所以只需要 13 个二进制位即可对块进行全局计数，实际实现用 int 32 位变量，即 4 字节表示 1 个块号。

在一级子索引中，如果一个数据块都用来存放块号，则可以存放 $512/4 = 128$ 个。因此，只使用一级子索引可以容纳最大的文件为 $128 \times 512 = 64KB$ 。需要使用二级子索引。只使用二级子索引时，索引结点中的一个指针可以指向 128×128 个块，即 $128 \times 128 \times 512B = 8MB$ ，已经可以满足要求了。为了尽量“像”ext2，也为了简单起见，索引结点的直接索引定义 6 个，一级子索引定义 1 个，二级子索引定义 1 个。总计 8 个指针。

```

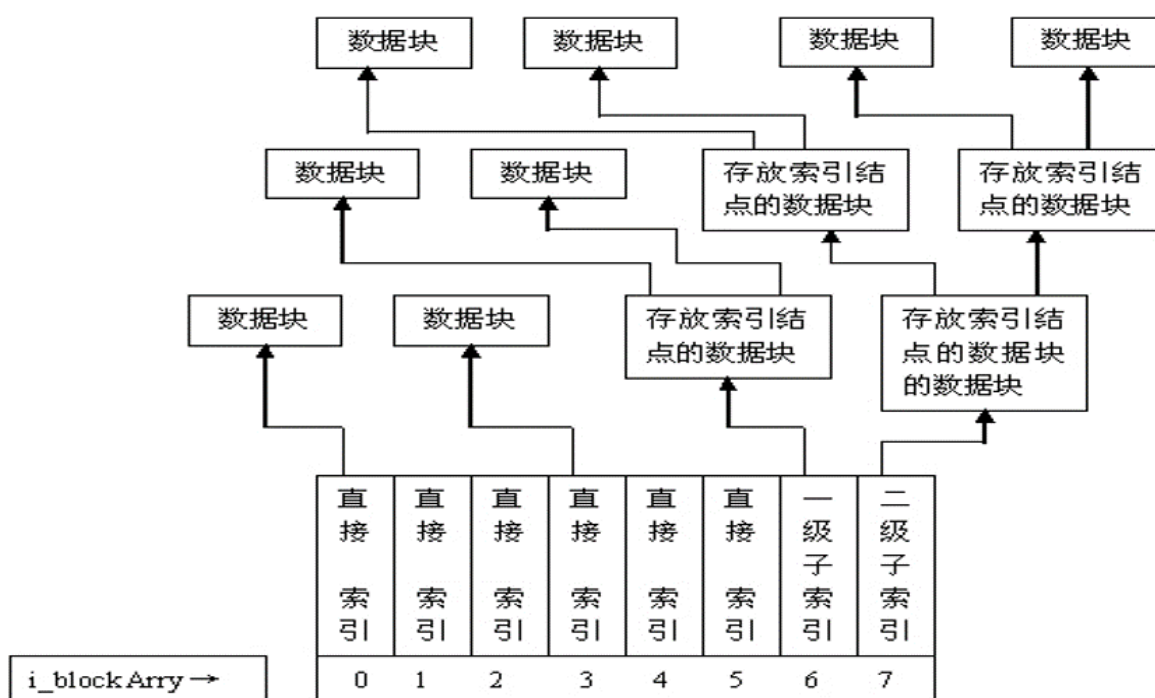
typedef struct ext2_inode //索引节点 64 字节
{
    int i_mode;           //文件类型及访问权限 1:普通文件, 2:目录
    int i_blocks;         //文件内容占用的数据块个数
    int i_size;           //大小(字节)
    time_t i_atime;       //访问时间
    time_t i_ctime;       //创建时间
    time_t i_mtime;       //修改时间
    time_t i_dtime;       //删除时间
    int i_block[8];       //指向数据块的指针
    char i_pad[24];       //填充 1(0xff)
} ext2_inode;

```

每个索引结点的长度为 64 字节

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

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



上图为 `i_block` 域与文件大小以及数据块的关系

(2) 当文件长度小于等于 $512 \times 6 = 3072$ 字节 (3KB) 时，只用到直接索引

(2)当文件长度大于 3072 字节，并且小于等于 3KB+64KB 即 67KB 时，除使用直接索引处，还将使用一级子索引。

(3) 当文件长度大于 67KB，并且小于 2MB（文件系统最大值）时，将开始使用二级子索引。

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

基于以上若干定义，得到模拟文件系统的“硬盘”数据结构

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

整个模拟文件系统所需要的“硬盘”空间为 $1+1+1+512+4096=4611$ 个块。共计 $4611 \times 512\text{bytes} = 2,360,832$ 字节 = 2305.5KB = 2.35MB。最多可容纳的文件数目为 $4096 - 17 = 4079$ 个。每个文件占用的数据空间最小为 512 字节，即一个块大小。

```
typedef struct ext2_dir_entry //目录体 32 字节
{
    int inode;           //索引节点号
    int rec_len;         //目录项长度
    int name_len;        //文件名长度
    int file_type;       //文件类型(1:普通文件, 2:目录...)
    char name[EXT2_NAME_LEN]; //文件名
    char dir_pad;        //填充
} ext2_dir_entry;
```

与 ext2 相同，目录作为特殊的文件来处理。将第 1 个索引结点指向根目录。根目录的索引结点中直接索引域指向数据块 0。

当文件系统在初始化时，根目录的数据块（即数据块 1）将被初始化。其所包含的所有索引节点号以及目录项长度域将被置 0。当文件被删除时，其所在目录项长度不变，索引节点号将被置 0。

当新建一个文件时，程序将从目录体的数据块查找索引节点号为 0 的目录项，并检查其长度是否够用。是，则分配给该文件，否则继续查找，直到找到长度够用，或者是长度为 0（即未被使用过）的地址，为文件建立目录项。

当建立的是一个目录时，将其所分配到的索引结点所指向的数据块清空。并且自动写入两个特殊的目录项。一个是当前目录“.”，其索引结点即指向本身的数据块。另一个是上一级目录“..”，其索引结点指向上一级目录的数据块。

作为简化，文件类型使用 1 表示普通文件，2 表示目录。

基于上述计算分析，进行如下宏定义

```
#define blocks 4611           // 1+1+1+512+4096,总块数
#define blocksiz 512          //每块字节数
#define inodesiz 64           //索引长度
#define data_begin_block 515  //数据开始块
#define dirsiz 32              //目录体长度
#define EXT2_NAME_LEN 15      //文件名长度
#define PATH "vdisk"          //文件系统
```

程序设计：

(1) 初始化模拟文件系统

在已有的文件系统的基础上建立一个大小为 2,360,832 字节(即 4611 个块)的文件 vdisk，这个文件即用来模拟硬盘。以后，文件系统的所有操作，均通过读写这个文件实现。并且，完全模拟硬盘读写方式，一次读取 1 个块，即 512 字节。即使只有 1 个字节的修改，也通过读写一个数据块来实现。

```
int format(ext2_inode *current)
{
    FILE *fp = NULL;
    int i;
    unsigned int zero[blocksiz / 4]; //零数组，用来初始化块为 0
    time_t now;
    time(&now);
    while (fp == NULL)
    {
        fp = fopen(PATH, "w+");
        for (i = 0; i < blocksiz / 4; i++)
            zero[i] = 0;
        for (i = 0; i < blocks; i++) //初始化所有 4611 块为 0
        {
            fseek(fp, i * blocksiz, SEEK_SET);
            fwrite(&zero, blocksiz, 1, fp);
        }
    }
}
```

在初始化过程中调用了 format 函数，format 函数是负责将整个文件格式化，在初始化过程中由于文件 vdisk 还没有被创建，所以在 fopen(PATH,"w+");的时候会创建文件 vdisk，然后将文件中每一块初始化为 0

```
//初始化组描述符
strcpy(group_desc.bg_volume_name, "Volume_name"); //初始化卷名为abcd
group_desc.bg_block_bitmap = 1; //保存块位图的块号
group_desc.bg_inode_bitmap = 2; //保存索引节点位图的块号
group_desc.bg_inode_table = 3; //索引节点表的起始块号
group_desc.bg_free_blocks_count = 4095; //除去一个初始化目录。空闲数据块的个数
group_desc.bg_free_inodes_count = 4095;
group_desc.bg_used_dirs_count = 1;
strcpy(group_desc.psw, "123");
fseek(fp, 0, SEEK_SET);
fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp); //第一块为组描述符
```

```
//初始化数据块位图和索引节点位图，第一位置为 1
zero[0] = 0x80000000;
fseek(fp, 1 * blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp); //第二块为块位图，块位图的第一位为 1
fseek(fp, 2 * blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp); //第三块为索引位图，索引节点位图的第一位为 1
```

```
//初始化索引节点表，添加一个索引节点
inode.i_mode = 2;
inode.i_blocks = 1;
inode.i_size = 64;
inode.i_ctime = now;
inode.i_atime = now;
inode.i_mtime = now;
inode.i_dtime = 0;
fseek(fp, 3 * blocksiz, SEEK_SET);
fwrite(&inode, sizeof(ext2_inode), 1, fp); //第四块开始为索引节点表
```

```
//向第一个数据块写 当前目录
dir.inode = 0;
dir.rec_len = 32; //默认目录体为32字节
dir.name_len = 1;
dir.file_type = 2;
strcpy(dir.name, "."); //当前目录
fseek(fp, data_begin_block * blocksiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp);
//当前目录之后写 上一目录
dir.inode = 0; //因为是根目录所以上一目录就是当前目录
dir.rec_len = 32;
dir.name_len = 2;
dir.file_type = 2;
strcpy(dir.name, ".."); //上一目录
fseek(fp, data_begin_block * blocksiz + dirsiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp); //第data_begin_block+1 =516 块开始为数据
```

随后将组描述符 group_desc，数据块位图，索引节点位图进行初始化，然后在索引节点表中设置根目录的索引节点，在目录体块（第一个数据块）中设置根目录的目录体，根目录上一级目录..（就是根目录）的目录体。

```
//current = &inode;
initialize(current); //将指针指向根目录
```

最后将 format 函数的参数 current 设置成索引节点中第一个节点（即根目录节点）结束 format 函数。

```
int initfs(ext2_inode *cu)
{
    f = fopen(PATH, "r+");
    if (f == NULL)
    { // char ch[20]; /* */
        char ch;
        int i;
        printf("File system couldn't be found. Do you want to create one?\n[Y/N]");
        i = 1;
        while (i)
        {
            scanf("%c", &ch); /* */
            switch (ch)
            {
                case 'Y':
                case 'y': /* */
                    if (format(cu) != 0)
                        return 1;
                    f = fopen(PATH, "r");
                    i = 0;
                    break;
                case 'N':
                case 'n': /* */
                    exitdisplay();
                    return 1;
                default:
                    printf("Sorry, meaningless command\n");
                    break;
            }
        }
    }
    fseek(f, 0, SEEK_SET);
    fread(&group_desc, sizeof(ext2_group_desc), 1, f);
    fseek(f, 3 * blocksiz, SEEK_SET);
    fread(&inode, sizeof(ext2_inode), 1, f);
    fclose(f);
    initialize(cu);
}
```

初始化调用函数 initfs，根据用户的输入决定下一步操作，若输入 y 或 Y，则调用 format 函数创建文件 vdisk 并对其进行格式化操作，然后将 cu 设置为根目录的索引节点指针

（2）文件系统级（底层）函数及其子函数

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


```
//寻找空索引
int FindInode()
{
    FILE *fp = NULL;
    unsigned int zero[blocksiz / 4];
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+");
    fseek(fp, 2 * blocksiz, SEEK_SET); // inode 位图
    fread(zero, blocksiz, 1, fp); // zero保存索引节点位图
    // unsigned int zero[128] ,每个int4字节, 共128个, 故一共能表示128*4*32位, =512*8 没问题!
```

设置一个寻找空索引节点的函数 FindInode,在函数中设置了 int 类型的数组 zero, 大小为 blocksiz/4,因为一个 int 占 4 个字节, 所以整个 zero 数据占的字节数为 blocksiz/4*4=blocksiz, 刚好能表示一个块。用 fp 打开文件 vdisk 后将其偏移两个块的距离, 这样就达到了表示索引节点位图的第三个块, 然后用 fread 函数在 fp 所指的位置读取一个块赋值给 zero, 这样 zero 就表示了整个索引节点位图。

```
for (i = last_allco_inode; i < (last_allco_inode + blocksiz / 4); i++) //一个inode号是int存储, 故为4字节,
// last_allco_inode + blocksiz / 4 其实i的绝对数值已经超出索引节点位图的存储范围
//但是因为要判断last_allco_inode之前的索引节点有无空闲, 所以后面计算的时候%取余即可。
{
    if (zero[i % (blocksiz / 4)] != 0xffffffff) //当还有空闲的索引节点时; 一个int4字节, 4*8=32位
    //i % (blocksiz / 4)是某个索引节点号
    // zero[i % (blocksiz / 4)]表示inode位图中的某段32位区域 (128*4*32) 一共有128个这样的区域
    {
        unsigned int j = 0x80000000, k = zero[i % (blocksiz / 4)], l = i;
        for (i = 0; i < 32; i++)
        {
            if (!(k & j)) // & 按位与, 再取非, 如果结果不为0, 说明第i位有空闲, 否则, j = j / 2, 考察下一位是否空闲
            {
                zero[l % (blocksiz / 4)] = zero[l % (blocksiz / 4)] | j; //如果空闲, 将此位置1

                group_desc.bg_free_inodes_count -= 1; //索引节点数减 1
                fseek(fp, 0, 0); //移动到起始位置—组描述符所在块
                fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp); //更新组描述符 (索引节点数目信息)

                fseek(fp, 2 * blocksiz, SEEK_SET);
                fwrite(zero, blocksiz, 1, fp); //更新inode位图, zero存储的是整个inode位图, 所以直接更新即可

                last_allco_inode = l % (blocksiz / 4);
                fclose(fp);
                return l % (blocksiz / 4) * 32 + i; // 返回空闲的inode号
            }
            else
            {
                j = j / 2; // 考察下一位
            }
        }
    }
}
```

last_allco_inode 表示最近一次找到的空节点在数组 zero 中的下标, 设置一个 for 循环, 循环变量为 int 类型变量 i, i 的初始值为 last_allco_inode, 每经历一次循环 i 就+1, 直到 i 的值等于 last_allco_inode+blocksiz/4 (因为整个 zero 数组只有 blocksiz/4 个元素) 才结束 for 循环, 在循环体内, 只有一个 if 语句, 如果 zero[i%(blocksiz/4)]=0xffffffff, 则表示 zero[i%(blocksiz/4)]对应的 32 个索引

节点全部已被分配，不能再被申请，所以结束 if 语句进入下一个循环；如果 $\text{zero}[\text{i} \% (\text{blocksiz}/4)] \neq 0\text{xffffffff}$ ，则表示 $\text{zero}[\text{i} \% (\text{blocksiz}/4)]$ 对应的 32 个索引节点至少有一个索引节点还没有被分配，进入 if 语句内，首先设置变量 k 和 i 分别保存 $\text{zero}[\text{i} \% (\text{blocksiz}/4)]$ 的值和 i 的值，再设置 int 类型的变量 $\text{j} = 0\text{x80000000}$ (即 32 位中只有最高位是 1)，然后进入一个 for 循环，将 i 初始化为 0，判断条件为 $\text{i} < 32$ ，末尾表达式为 $\text{i}++$ ，在循环体内设置一个 if 语句，对 $\text{k} \& \text{j}$ 进行判断，如果结果为 0，则说明此时没有匹配成功，需要将 j 的值右移一位，保证 j 的 32 位数中只有一位为 1，且 1 的位置右移了一位，再进行下一次循环。如果结果非 0，则说明此时 j 中的 1 正是空索引节点在数组 $\text{zero}[\text{i} \% (\text{blocksiz}/4)]$ 中的位置，此时进入 if 语句内，将组标识符对应的索引节点数-1 后再将组标识符写回文件对应的位置。将 $\text{zero}[\text{i} \% (\text{blocksiz}/4)]$ 和 j 进行按位或操作，其结果保存在 $\text{zero}[\text{i} \% (\text{blocksiz}/4)]$ 中，再将整个 zero 写回索引节点位图中，将找到的空节点在索引节点位图中的标识符改为 1。然后将 last_allco_inode 更新为 $\text{l} \% (\text{blocksiz}/4)$ ，再关掉 fp，执行 $\text{return l} \% (\text{blocksiz} / 4) * 32 + \text{i}$ ；返回找到的空索引节点的节点号结束函数 FindNode。如果 for 循环执行完后 $\text{zero}[\text{i} \% (\text{blocksiz}/4)] \neq 0\text{xffffffff}$ 一直不成立，则说明所有的索引节点已经全部被分配了，此次寻找空节点失败， return -1 ；返回 -1。


```

//寻找空block
int FindBlock()
{
    FILE *fp = NULL;
    unsigned int zero[blocksiz / 4];
    int i;
    while (fp == NULL)
    {
        fp = fopen(PATH, "r+");
        fseek(fp, 1 * blocksiz, SEEK_SET);
        fread(zero, blocksiz, 1, fp); // zero保存块位图
        for (i = last_allco_block; i < (last_allco_block + blocksiz / 4); i++)
        {
            if (zero[i % (blocksiz / 4)] != 0xffffffff)
            {
                unsigned int j = 0x80000000, k = zero[i % (blocksiz / 4)], l = i;
                for (i = 0; i < 32; i++)
                {
                    if (!(k & j))
                    {
                        zero[l % (blocksiz / 4)] = zero[l % (blocksiz / 4)] | j;
                        group_desc.bg_free_blocks_count -= 1; //块数减 1
                        fseek(fp, 0, 0);
                        fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp);
                        fseek(fp, 1 * blocksiz, SEEK_SET);
                        fwrite(zero, blocksiz, 1, fp);
                        last_allco_block = l % (blocksiz / 4);
                        fclose(fp);
                        return l % (blocksiz / 4) * 32 + i;
                    }
                    else
                    {
                        j = j / 2;
                    }
                }
            }
        }
    }
    fclose(fp);
    return -1;
}

```

寻找空数据块的函数 FindBlock 和刚才的寻找空索引节点函数 FindInode 如出一辙，都是先把对应的位图读出来，保存在数组 zero 中，再在 zero 中从上一次找到的空块位置开始找位图为 0 的地方，在这里就不加赘述。

```
//删除inode, 更新inode节点位图
void DelInode(int len) //len是inode号, 是一个unsigned int值
{
    unsigned int zero[blocksiz / 4], i;
    int j;
    f = fopen(PATH, "r+");
    fseek(f, 2 * blocksiz, SEEK_SET);
    fread(zero, blocksiz, 1, f);
    i = 0x80000000;
    for (j = 0; j < len % 32; j++)
        i = i / 2;
    zero[len / 32] = zero[len / 32] ^ i;
    fseek(f, 2 * blocksiz, SEEK_SET);
    fwrite(zero, blocksiz, 1, f);
    fclose(f);
}
```

函数以读写模式 ("r+") 打开文件。从文件系统中读取 inode 位图到名为 zero 的数组中。计算与提供的 inode 索引 (len) 对应的位在位图中的位置。使用位操作在计算的位置切换位, 将 inode 标记为空闲。定位到文件中对应 inode 位图的位置, 并将更新后的 zero 数组写回文件, 有效更新 inode 位图。

```
//删除block块, 更新块位图
void DelBlock(int len)
{
    unsigned int zero[blocksiz / 4], i;
    int j;
    f = fopen(PATH, "r+");
    fseek(f, 1 * blocksiz, SEEK_SET);
    fread(zero, blocksiz, 1, f);
    i = 0x80000000;
    for (j = 0; j < len % 32; j++)
        i = i / 2;
    zero[len / 32] = zero[len / 32] ^ i;
    fseek(f, 1 * blocksiz, SEEK_SET);
    fwrite(zero, blocksiz, 1, f);
    fclose(f);
}
```

DelBlock 函数机理和 DelInode 函数一样

```
int getch() // 使用方法，在需要不显示输入的是什么的地方调用，返回值为用户输入的字符。
{
    int ch;
    struct termios oldt, newt;
    tcgetattr(STDIN_FILENO, &oldt); // 用来获取终端参数，成功返回零；失败返回非零
    newt = oldt;
    newt.c_lflag &= ~(ECHO | ICANON);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    ch = getchar();
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
    return ch;
}
```

该函数使用 termios 结构体来获取和修改终端的设置，实现输入字符不回显和非规范模式。tcgetattr 用于获取终端的设置，并将其保存在 oldt 结构体中。创建一个新的结构体 newt，其中的标志位 c_lflag 被修改，以禁用回显和规范模式。使用 tcsetattr 将新的终端设置应用到终端。使用 getchar 获取用户输入的字符，由于禁用了回显，用户输入的字符不会在终端显示。最后，通过 tcsetattr 恢复终端的原始设置。

```
//返回目录的起始存储位置，每个目录 32 字节
int dir_entry_position(int dir_entry_begin, int i_block[8]) // dir_entry_begin目录体的相对开始字节
{
    int dir_blocks = dir_entry_begin / 512; // 存储目录需要的块数
    int block_offset = dir_entry_begin % 512; // 块内偏移字节数
    int a;
    FILE *fp = NULL;
    if (dir_blocks <= 5) //前六个直接索引
        return data_begin_block * blocksiz + i_block[dir_blocks] * blocksiz + block_offset;
    else //间接索引
    {
        while (fp == NULL)
            fp = fopen(PATH, "r+");
        dir_blocks = dir_blocks - 6;
        if (dir_blocks < 128) //一个块 512 字节，一个int为 4 个字节 一级索引有 512/4= 128 个
        {
            int a;
            fseek(fp, data_begin_block * blocksiz + i_block[6] * blocksiz + dir_blocks * 4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);
            return data_begin_block * blocksiz + a * blocksiz + block_offset;
        }
        else //二级索引
        {
            dir_blocks = dir_blocks - 128;
            fseek(fp, data_begin_block * blocksiz + i_block[7] * blocksiz + dir_blocks / 128 * 4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);
            fseek(fp, data_begin_block * blocksiz + a * blocksiz + dir_blocks % 128 * 4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);
            return data_begin_block * blocksiz + a * blocksiz + block_offset;
        }
        fclose(fp);
    }
}
```

dir_entry_position 函数用于确定目录项在文件系统中的实际存储位置。以下是该函数的详细运行逻辑：

参数解析： 函数接受两个参数：

dir_entry_begin: 目录项的相对起始字节偏移量，即在目录体中的位置。

i_block[8]: 一个数组，存储了索引节点中指向数据块的指针。

计算块数和偏移量： 首先，函数计算存储目录项所需的块数 dir_blocks 和块内偏移字节数 block_offset。dir_blocks 计算了目录项在文件系统中所需的块数，考虑了直接索引、一级间接索引和二级间接索引的情况。block_offset 表示目录项在所在块内的偏移字节数。

处理直接索引： 如果 dir_blocks 小于等于 5，表示目录项在索引节点的直接索引中，直接计算目录项的绝对位置并返回。

处理一级间接索引： 如果 dir_blocks 大于 5 且小于 128，表示目录项在索引节点的一级间接索引中。函数读取相应的一级索引块，找到实际存储目录项的块号。然后计算目录项的绝对位置，并返回。

处理二级间接索引： 如果 dir_blocks 大于等于 128，表示目录项在索引节点的二级间接索引中。函数首先读取相应的二级索引块，再读取一级索引块，找到实际存储目录项的块号。最后计算目录项的绝对位置，并返回。

返回实际存储位置： 函数返回目录项在文件系统中的实际存储位置。这个位置是相对于文件系统起始位置的偏移量。

通过这个逻辑，dir_entry_position 函数确保了能够根据相对偏移量和文件系统的索引结构准确地定位目录项的位置。

```
/*在当前目录 打开一个目录
current指向新打开的当前目录 (ext2_inode)
*/
int Open(ext2_inode *current, char *name)
{
    FILE *fp = NULL;
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+");
    for (i = 0; i < (current->i_size / 32); i++)
    {
        fseek(fp, dir_entry_position(i * 32, current->i_block), SEEK_SET); //定位目录的偏移位置
        fread(&dir, sizeof(ext2_dir_entry), 1, fp);
        if (!strcmp(dir.name, name))
        {
            if (dir.file_type == 2) //目录
            {
                fseek(fp, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET);
                fread(current, sizeof(ext2_inode), 1, fp);
                fclose(fp);
                return 0;
            }
        }
    }
    fclose(fp);
    return 1;
}
```

Open 函数的主要目标是在当前目录中打开一个子目录，并将当前目录指向新打开的子目录。以下是该函数的详细运行逻辑：

参数解析： ext2_inode *current：指向当前目录的索引节点。

char *name：要打开的子目录的名称。

文件打开循环： 函数使用一个循环遍历当前目录的所有目录项。在每次迭代中，通过 fread 从文件系统中读取一个目录项（ext2_dir_entry）。判断当前目录项的文件名是否与要打开的子目录名称匹配。

目录项匹配检查： 如果找到匹配的目录项，继续进行下一步检查。确保匹配的目录项是一个子目录（dir.file_type == 2）。

目录项匹配处理： 如果匹配的目录项是一个子目录，说明找到了要打开的目录。通过 fseek 和 fread 从文件系统中读取与该目录项对应的索引节点信息，并将 current 指针更新为该索引节点。关闭文件指针，并返回 0 表示打开成功。

未找到目录项处理： 如果循环结束时都没有找到匹配的目录项，说明目标目录不存在，返回 1 表示打开失败。

通过这个逻辑，Open 函数实现了在当前目录中打开子目录的功能。

```
/******关闭当前目录******/
/*
关闭时仅修改最后访问时间
返回时 打开上一目录 作为当前目录
*/
int Close(ext2_inode *current)
{
    time_t now;
    ext2_dir_entry bentry;
    FILE *fout;
    fout = fopen(PATH, "r+");
    time(&now);
    current->i_atime = now; //修改最后访问时间
    fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz, SEEK_SET);
    fread(&bentry, sizeof(ext2_dir_entry), 1, fout); // current's dir_entry

    fseek(fout, 3 * blocksiz + (bentry.inode) * sizeof(ext2_inode), SEEK_SET);
    fwrite(current, sizeof(ext2_inode), 1, fout); //写入文件系统中
    fclose(fout);
    return Open(current, "..");
}
```

Close 函数的主要目标是关闭当前目录，并在返回之前打开上一级目录作为新的当前目录。以下是该函数的详细运行逻辑：

参数解析： ext2_inode *current：指向当前目录的索引节点。

获取当前目录的目录项： 通过读取当前目录的第一个目录项，获取当前目录在文件系统中的目录项信息（ext2_dir_entry）。使用 fseek 定位到当前目录的第

一个目录项的位置，并使用 `fread` 读取该目录项。

修改最后访问时间： 获取到当前目录的目录项后，将当前目录的最后访问时间 (`i_atime`) 更新为当前时间。更新文件系统中的当前目录信息： 通过将修改后的索引节点信息写回文件系统，将最后访问时间的更新保存到文件系统中。

打开上一级目录： 调用 `Open` 函数打开上一级目录作为新的当前目录。`Open` 函数通过读取目录项信息将 `current` 指针更新为上一级目录的索引节点。

如果上一级目录不存在，`Open` 函数返回 1，表示打开失败。

返回打开结果： 如果打开上一级目录成功，`Close` 函数返回 0，表示关闭当前目录成功。如果上一级目录不存在，返回 1，表示关闭失败。

通过这个逻辑，`Close` 函数实现了关闭当前目录并打开上一级目录的功能。

```
void add_block(ext2_inode *current, int i, int j) // 空间不够，故增加一个数据块来存放内容
{
    FILE *fp = NULL;
    while (fp == NULL)
        fp = fopen(PATH, "r+");
    if (i < 6) //直接索引
    {
        current->i_block[i] = j;
    }
    else
    {
        i = i - 6;
        if (i == 0)
        {
            current->i_block[6] = FindBlock();
            fseek(fp, data_begin_block * blocksiz + current->i_block[6] * blocksiz, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp);
        }
        else if (i < 128) //一级索引
        {
            fseek(fp, data_begin_block * blocksiz + current->i_block[6] * blocksiz + i * 4, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp);
        }
        else //二级索引
    }
}
```

```

else //二级索引
{
    i = i - 128;
    if (i == 0)
    {
        current->i_block[7] = FindBlock();
        fseek(fp, data_begin_block * blocksiz + current->i_block[7] * blocksiz, SEEK_SET);
        i = FindBlock();
        fwrite(&i, sizeof(int), 1, fp);
        fseek(fp, data_begin_block * blocksiz + i * blocksiz, SEEK_SET);
        fwrite(&j, sizeof(int), 1, fp);
    }
    if (i % 128 == 0)
    {
        fseek(fp, data_begin_block * blocksiz + current->i_block[7] * blocksiz + i / 128 * 4, SEEK_SET);
        i = FindBlock();
        fwrite(&i, sizeof(int), 1, fp);
        fseek(fp, data_begin_block * blocksiz + i * blocksiz, SEEK_SET);
        fwrite(&j, sizeof(int), 1, fp);
    }
    else
    {
        fseek(fp, data_begin_block * blocksiz + current->i_block[7] * blocksiz + i / 128 * 4, SEEK_SET);
        fread(&i, sizeof(int), 1, fp);
        fseek(fp, data_begin_block * blocksiz + i * blocksiz + i % 128 * 4, SEEK_SET);
        fwrite(&j, sizeof(int), 1, fp);
    }
}
}
// 为当前目录寻找一个空目录块

```

add_block 函数的主要目标是在当前文件的索引节点中添加一个数据块，以扩展文件的存储空间。以下是该函数的详细运行逻辑：

参数解析： ext2_inode *current：指向当前文件的索引节点。

int i：要添加数据块的索引。

int j：要添加的数据块的块号。

判断数据块索引： 函数首先判断要添加的数据块的索引 i 的值。如果 i 小于 6，表示使用直接索引。如果 i 大于等于 6，表示使用一级索引或二级索引。

添加直接索引： 如果 i 小于 6，直接将数据块号 j 赋值给索引节点的相应位置。修改 current->i_block[i] = j;。

添加一级索引： 如果 i 大于等于 6 且小于 6 + temp（temp 为每个块可存放的 INT 数量），表示使用一级索引。通过 fseek 和 fread 从文件系统中读取一级索引块的块号。将 j 赋值给一级索引块中相应位置的 INT 值。

添加二级索引： 如果 i 大于等于 6 + temp，表示使用二级索引。计算二级索引块的剩余块数 remain_block。通过 fseek 和 fread 从文件系统中读取二级索引块的块号。根据 remain_block 的值定位到相应的一级索引块。通过 fseek 和 fread 读取一级索引块中的块号。将 j 赋值给一级索引块中相应位置的 INT 值。通过这个逻辑，add_block 函数实现了在当前文件的索引节点中添加一个数据块的功能，同时处理了直接索引、一级索引和二级索引的情况。


```

// 为当前目录寻找一个空目录体
int FindEntry(ext2_inode *current)
{
    FILE *fout = NULL;
    int location;           //条目的绝对地址
    int block_location;     //块号
    int temp;               //每个block 可以存放的INT 数量
    int remain_block;       //剩余块数
    location = data_begin_block * blocksiz;
    temp = blocksiz / sizeof(int);
    fout = fopen(PATH, "r+");
    if (current->i_size % blocksiz == 0) //一个BLOCK 使用完后增加一个块
    {
        add_block(current, current->i_blocks, FindBlock());
        current->i_blocks++;
    }
    if (current->i_blocks < 6) //前 6 个块直接索引
    {
        location += current->i_block[current->i_blocks - 1] * blocksiz;
        location += current->i_size % blocksiz;
    }
    else if (current->i_blocks < temp + 5) //一级索引
    {
        block_location = current->i_block[6];
        fseek(fout, (data_begin_block + block_location) * blocksiz + (current->i_blocks - 6) * sizeof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        location += block_location * blocksiz;
        location += current->i_size % blocksiz;
    }
    else //二级索引
    {
        block_location = current->i_block[7];
        remain_block = current->i_blocks - 6 - temp;
        fseek(fout, (data_begin_block + block_location) * blocksiz + (int)((remain_block - 1) / temp + 1) * sizeof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        remain_block = remain_block % temp;
        fseek(fout, (data_begin_block + block_location) * blocksiz + remain_block * sizeof(int),
            SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout);
        location += block_location * blocksiz;
        location += current->i_size % blocksiz + dirsiz;
    }
    current->i_size += dirsiz;
    fclose(fout);
    return location;
}

```

FindEntry 函数的主要目标是为当前目录寻找一个空的目录体。以下是该函数的详细运行逻辑：

文件指针和变量初始化： 函数开始时，初始化了文件指针 fout 为 NULL，并定义了几个用于存储目录体位置信息的变量 location、block_location、temp 和 remain_block。

计算目录体的位置： 如果当前目录的大小除以块的大小没有余数

(current->i_size % blocksiz == 0)，说明当前块已经用完，需要增加一个块。调用 add_block 函数，在当前目录的索引节点中添加一个新的数据块。更新 current->i_blocks，表示当前目录的块数增加了。计算目录体的绝对地址 location。

直接索引情况： 如果当前目录的块数小于 6，表示直接使用索引节点中的直接

索引。计算目录体在文件系统中的位置 `location`，通过当前目录的直接索引定位到相应的数据块。目录体的位置为 `location + current->i_size % blocksiz`，即当前块中的末尾。

一级索引情况： 如果当前目录的块数大于等于 6 且小于 `6 + temp`（每个块可存放的 INT 数量）。通过一级索引块找到相应的数据块。

计算目录体在文件系统中的位置 `location`，加上当前目录的块数减去 6（一级索引块的数量）。目录体的位置为 `location + current->i_size % blocksiz`。

二级索引情况： 如果当前目录的块数大于等于 `6 + temp`，表示使用二级索引。计算二级索引块的剩余块数 `remain_block`。通过二级索引块找到相应的一级索引块。计算一级索引块中的块号，找到相应的数据块。计算目录体在文件系统中的位置 `location`。目录体的位置为 `location + current->i_size % blocksiz + dirsiz`，其中 `dirsiz` 为目录体的长度。

更新当前目录的大小： 在任何情况下，都会更新当前目录的大小，将其增加一个目录体的长度。

关闭文件指针： 最后关闭文件指针。

通过这个逻辑，`FindEntry` 函数实现了为当前目录寻找一个空的目录体的功能，同时处理了直接索引、一级索引和二级索引的情况。

```
/******获取当前目录的目录名******/
void getstring(char *cs, ext2_inode node)
{
    ext2_inode current = node;
    int i, j;
    ext2_dir_entry dir;
    f = fopen(PATH, "r+");
    Open(&current, ".."); // current指向上一级目录
    for (i = 0; i < node.i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, node.i_block), SEEK_SET);
        fread(&dir, sizeof(ext2_dir_entry), 1, f);
        if (!strcmp(dir.name, "."))
        {
            j = dir.inode;
            break;
        }
    }
    for (i = 0; i < current.i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, current.i_block), SEEK_SET);
        fread(&dir, sizeof(ext2_dir_entry), 1, f);
        if (dir.inode == j)
        {
            strcpy(cs, dir.name);
            return;
        }
    }
}
```

输入参数：函数接收两个参数，一个是指向字符数组的指针 cs，用于存储目录名，另一个是表示当前目录的索引节点结构体 node。

打开文件系统文件：函数内部通过文件指针 f 打开文件系统文件，以读写方式打开。

获取上一级目录的信息：调用 Open 函数，将当前目录切换到上一级目录（..），并获取上一级目录的索引节点信息。

遍历当前目录的目录项：使用一个循环，遍历当前目录的目录项。在循环中，通过 fseek 定位到每个目录项的位置，使用 fread 读取目录项的内容。

比较目录项中的 inode 号：通过比较读取的目录项中的 inode 号，判断当前目录项是否是上一级目录中的条目。

获取上一级目录的目录名：如果找到与当前目录相对应的目录项，通过 getstring 递归调用，获取上一级目录的目录名。

将当前目录名添加到路径中：在递归调用返回后，将当前目录的名称（从目

录项中获取）追加到路径中，并使用路径分隔符（/）进行连接。

关闭文件：在函数结束前关闭文件系统文件。

总体来说，getstring 函数通过递归调用实现了获取当前目录的绝对路径，将路径以字符串形式存储在参数 cs 所指向的字符数组中。

(3) 命令层函数

```
int Read(ext2_inode *current, char *name)
{
    FILE *fp = NULL;
    int i;
    while (fp == NULL)    fp = fopen(PATH, "r+");
    for (i = 0; i < (current->i_size / 32); i++) //遍历当前目录的目录项，ext2_inode *current 指向当前目录，每个目录项32字节
    {
        fseek(fp, dir_entry_position(i * 32, current->i_block), SEEK_SET); // 返回目录项的起始存储位置
        fread(&dir, sizeof(ext2_dir_entry), 1, fp);
        if (!strcmp(dir.name, name)) // 比较文件名是否相同
            if (dir.file_type == 1) // 如果文件名相同，且文件类型是文件的话（文件类型有目录和文件两种）
            {
                time_t now;
                ext2_inode node;
                char content_char;
                fseek(fp, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET); //根据目录体中保存的索引节点号，找到文件的inode位置
                fread(&node, sizeof(ext2_inode), 1, fp); // original inode, node为文件的inode信息
                i = 0;
                for (i = 0; i < node.i_size; i++) // 读出大小为i_size的文件，一次读一个char
                {
                    //根据指向数据块的文件指针i_block，将默认的读写指针移动到文件的数据块中
                    fseek(fp, dir_entry_position(i, node.i_block), SEEK_SET);
                    fread(&content_char, sizeof(char), 1, fp);
                    if (content_char == 0xD) //0xD (ascii—回车\n)
                        printf("\n");
                    else
                        printf("%c", content_char);
                }
                printf("\n");
                time(&now);
                node.i_atime = now; // 修改访问时间
                fseek(fp, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET);
                fwrite(&node, sizeof(ext2_inode), 1, fp); // update inode 将修改写入文件系统中
                fclose(fp);
                return 0;
            }
    }
    fclose(fp);
    return 1;
}
```

Read 函数的主要目标是从文件系统中读取指定文件的内容，并在控制台上显示该文件的内容。以下是该函数的详细运行逻辑：

参数解析：ext2_inode *current：指向当前目录的索引节点。

char *name：要读取内容的文件的名称。

文件读取循环： 函数使用一个循环遍历当前目录的所有目录项。在每次迭代中，通过 `fread` 从文件系统中读取一个目录项（`ext2_dir_entry`）。判断当前目录项的文件名是否与要读取的文件名称匹配。

目录项匹配检查： 如果找到匹配的目录项，继续进行下一步检查。确保匹配的目录项是一个文件（`dir.file_type == 1`，文件类型为 1 表示普通文件）。

目录项匹配处理： 如果匹配的目录项是一个文件，说明找到了要读取的文件。通过 `fseek` 和 `fread` 从文件系统中读取与该目录项对应的索引节点信息。使用 `fseek` 和 `fread` 读取文件的内容，并在控制台上显示文件的内容。将文件的最后访问时间（`i_atime`）更新为当前时间。

未找到目录项处理： 如果循环结束时都没有找到匹配的目录项，说明要读取的文件不存在，返回 1 表示读取失败。

返回读取结果： 如果成功读取文件并显示内容，`Read` 函数返回 0，表示读取成功。如果未找到文件，返回 1，表示读取失败。

通过这个逻辑，`Read` 函数实现了从文件系统中读取文件内容并在控制台上显示的功能。

Create 函数：

```

/*****创建文件或者目录*****/
/*
 * type=1 创建文件
 * type=2 创建目录
 * current 当前目录索引节点
 * name 文件名或目录名
 */
int Create(int type, ext2_inode *current, char *name)
{
    FILE *fout = NULL;
    int i;
    int block_location; // block location
    int node_location; // node location
    int dir_entry_location; // dir entry location
    time_t now;
    ext2_inode ainode;
    ext2_dir_entry aentry, bentry; // bentry保存当前系统的目录体信息
    time(&now);
    fout = fopen(PATH, "r+");
    node_location = FindInode(); // 寻找空索引

    // 检查是否存在重复文件或目录名称
    for (i = 0; i < current->i_size / dirsiz; i++)
    {
        fseek(fout, dir_entry_position(i * sizeof(ext2_dir_entry), current->i_block), SEEK_SET);
        fread(&aentry, sizeof(ext2_dir_entry), 1, fout);
        if (aentry.file_type == type && !strcmp(aentry.name, name))
            return 1;
    }

    fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz, SEEK_SET);
    fread(&bentry, sizeof(ext2_dir_entry), 1, fout); // current's dir_entry
    if (type == 1) //文件

```

```
if (type == 1) //文件
{
    ainode.i_mode = 1;
    ainode.i_blocks = 0; //文件暂无内容
    ainode.i_size = 0; //初始文件大小为 0
    ainode.i_atime = now;
    ainode.i_ctime = now;
    ainode.i_mtime = now;
    ainode.i_dtime = 0;
    for (i = 0; i < 8; i++)
    {
        ainode.i_block[i] = 0;
    }
    for (i = 0; i < 24; i++)
    {
        ainode.i_pad[i] = (char)(0xff);
    }
}
else //目录
{
    ainode.i_mode = 2; //目录
    ainode.i_blocks = 1; //目录 当前和上一目录
    ainode.i_size = 64; //初始大小 32*2=64 //一旦新建一个目录，该目录下就有"."和"..".
    ainode.i_atime = now;
    ainode.i_ctime = now;
    ainode.i_mtime = now;
    ainode.i_dtime = 0;
    block_location = FindBlock();
    ainode.i_block[0] = block_location;
    for (i = 1; i < 8; i++)
    {
        ainode.i_block[i] = 0;
    }
    for (i = 0; i < 24; i++)
    {
        ainode.i_pad[i] = (char)(0xff);
    }
}
```

```
//当前目录
aentry.inode = node_location;
aentry.rec_len = sizeof(ext2_dir_entry);
aentry.name_len = 1;
aentry.file_type = 2;
strcpy(aentry.name, ".");
printf(".dir created.\n");
aentry.dir_pad = 0;
fseek(fout, (data_begin_block + block_location) * blocksiz, SEEK_SET);
fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);
//上一级目录
aentry.inode = bentry.inode;
aentry.rec_len = sizeof(ext2_dir_entry);
aentry.name_len = 2;
aentry.file_type = 2;
strcpy(aentry.name, "..");
aentry.dir_pad = 0;
fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);
printf("..dir created.\n");
//一个空条目
aentry.inode = 0;
aentry.rec_len = sizeof(ext2_dir_entry);
aentry.name_len = 0;
aentry.file_type = 0;
aentry.name[EXT2_NAME_LEN] = 0;
aentry.dir_pad = 0;
fwrite(&aentry, sizeof(ext2_dir_entry), 14, fout); //清空数据块
} // end else
//保存新建inode
fseek(fout, 3 * blocksiz + (node_location) * sizeof(ext2_inode), SEEK_SET);
fwrite(&ainode, sizeof(ext2_inode), 1, fout);
// 将新建inode 的信息写入current 指向的数据块
aentry.inode = node_location;
aentry.rec_len = dirsiz;
aentry.name_len = strlen(name);
```

```
if (type == 1)
{
    aentry.file_type = 1;
} //文件
else
{
    aentry.file_type = 2;
} //目录
strcpy(aentry.name, name);
aentry.dir_pad = 0;
dir_entry_location = FindEntry(current);
fseek(fout, dir_entry_location, SEEK_SET); //定位条目位置
fwrite(&aentry, sizeof(ext2_dir_entry), 1, fout);

//保存current 的信息,bentry 是current 指向的block 中的第一条
//ext2_inode cinode;
fseek(fout, 3 * blocksiz + (bentry.inode) * sizeof(ext2_inode), SEEK_SET);

// fread(&cinode, sizeof(ext2_inode), 1, fout);
// printf("after_cinode.i_size: %d\n", cinode.i_size);

fwrite(current, sizeof(ext2_inode), 1, fout);
fclose(fout);
return 0;
}
```

Create 函数的主要目的是在当前目录下创建一个新的文件或目录。以下是 Create 函数的运行逻辑的详细描述：

打开文件系统文件： 函数开始时，通过 `fopen(PATH, "r+")` 打开文件系统文件（vdisk），以便读取和写入文件系统的信息。

寻找空闲的索引节点： 调用 `FindNode` 函数寻找一个空闲的索引节点，用于存储新创建的文件或目录的信息。这个函数会更新索引节点位图，标记相应的索引节点为已使用，并返回空闲的索引节点号。

检查重名： 遍历当前目录下的目录项，检查是否存在同名的文件或目录。如果存在同名项，函数返回错误代码 1。

获取当前目录的目录项信息： 通过调用 `FindEntry` 函数获取当前目录下一个空闲的目录项位置。这个函数会返回一个用于存储新文件或目录信息的位置。

初始化新的索引节点： 如果创建的是文件，初始化一个新的索引节点（`ext2_inode`），设置相应的属性，包括文件类型、大小等。如果创建的是目录，初始化的内容包括目录项 `."`、`.."` 以及一个空目录项。

保存新的索引节点： 将新创建的索引节点写入文件系统的相应位置，更新文件系统的索引节点位图。

保存当前目录的信息： 将当前目录的信息写回文件系统。这涉及到更新当前

目录的目录项，增加目录项数量，以及更新当前目录的文件大小。

关闭文件系统文件： 关闭打开的文件系统文件，结束函数执行。

总体来说，Create 函数负责在文件系统中创建新的文件或目录，并确保相关的位图和目录项信息得到正确的更新。

```
int Write(ext2_inode *current, char *name)
{
    FILE *fp = NULL;
    ext2_dir_entry dir;
    ext2_inode node;
    time_t now;
    char str;
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+");
    while (1)
    {
        for (i = 0; i < (current->i_size / 32); i++)
        {
            fseek(fp, dir_entry_position(i * 32, current->i_block), SEEK_SET);
            fread(&dir, sizeof(ext2_dir_entry), 1, fp);
            if (!strcmp(dir.name, name))
            {
                if (dir.file_type == 1)
                {
                    fseek(fp, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET);
                    fread(&node, sizeof(ext2_inode), 1, fp);
                    break;
                }
            }
        }
        if (i < current->i_size / 32) // have file
            break;
        // Create(1,current,name); //have not file ,create a new file
        printf("There isn't this file,please create it first\n");
        return 0;
    }
    str = getch();
}
```



```
while (str != 27) // 没有检测到ESC (ascii = 27) 之前一直读
{
    printf("%c", str);
    if (!(node.i_size % 512)) // 需要增加数据块
    {
        add_block(&node, node.i_size / 512, FindBlock());
        node.i_blocks += 1;
    }
    fseek(fp, dir_entry_position(node.i_size, node.i_block), SEEK_SET);
    fwrite(&str, sizeof(char), 1, fp);

    node.i_size += sizeof(char);
    if (str == 0x0d)
        printf("%c", 0x0a);
    str = getch();
    if (str == 27)
        break;
}
time(&now);
node.i_mtime = now;
node.i_atime = now;
fseek(fp, 3 * blocksize + dir.inode * sizeof(ext2_inode), SEEK_SET);
fwrite(&node, sizeof(ext2_inode), 1, fp);
fclose(fp);
printf("\n");
return 0;
}
```

Write 函数的主要目的是向一个文件中写入数据。以下是 Write 函数的运行逻辑的详细描述：

打开文件系统文件： 函数开始时，通过 `fopen(PATH, "r+")` 打开文件系统文件（vdisk），以便读取和写入文件系统的信息。

查找目标文件的目录项： 通过遍历当前目录的目录项，查找要写入数据的目标文件。如果找不到目标文件，返回错误。

获取目标文件的索引节点信息： 通过目标文件的目录项中的索引节点号，读取文件系统中相应的索引节点信息。

循环读取用户输入的字符： 使用 `getch` 函数，循环读取用户输入的字符。这个循环一直持续，直到用户输入 ESC 键（ASCII 码 27）为止。

检查是否需要增加数据块： 在写入一个字符之前，检查目标文件的大小是否为块的整数倍，如果是，则调用 `add_block` 函数增加一个数据块。

写入字符到文件： 将用户输入的字符写入目标文件的数据块中，更新文件的大小和修改时间。

保存目标文件的索引节点信息： 将更新后的目标文件的索引节点信息写回

文件系统。

关闭文件系统文件： 关闭打开的文件系统文件，结束函数执行。

总体来说，Write 函数负责向目标文件中写入数据，涉及到数据块的管理和索引节点信息的更新。循环的读取用户输入字符保证了用户可以连续输入数据。

```
void ls(ext2_inode *current)
{
    ext2_dir_entry dir;
    int i, j;
    char timestr[150];
    ext2_inode node;
    f = fopen(PATH, "r+");
    printf("Type\t\tFileName\tCreateTime\t\t\tLastAccessTime\t\t\tModifyTime\n");
    printf("\n!!!!!!current->i_size:%d\n", current->i_size);
    for (i = 0; i < current->i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, current->i_block), SEEK_SET);
        fread(&dir, sizeof(ext2_dir_entry), 1, f); // 读出目录项内容
        fseek(f, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET);
        fread(&node, sizeof(ext2_inode), 1, f); // 读出索引节点的内容
        strcpy(timestr, "");
        strcat(timestr, asctime(localtime(&node.i_ctime)));
        strcat(timestr, asctime(localtime(&node.i_atime)));
        strcat(timestr, asctime(localtime(&node.i_mtime)));
        for (j = 0; j < strlen(timestr) - 1; j++)
        {
            if (timestr[j] == '\n')
            {
                timestr[j] = '\t';
            }
        }
        if (dir.file_type == 1)
            printf("File\t\t%s\t\t%s", dir.name, timestr);
        else
            printf("Directory\t%s\t\t%s", dir.name, timestr);
    }
    fclose(f);
}
```

Ls 函数的主要目的是列出当前目录的文件和子目录信息。以下是 Ls 函数的运行逻辑的详细描述：

打开文件系统文件： 函数开始时，通过 `fopen(PATH, "r+")` 打开文件系统文件（vdisk），以便读取文件系统的信息。

遍历当前目录的目录项： 使用循环，遍历当前目录的所有目录项。每个目录项包含文件或目录的相关信息。

读取目录项内容： 使用 `fread` 从当前目录的数据块中读取目录项的内容。目录项的结构是 `ext2_dir_entry`。

读取索引节点信息： 根据目录项中的索引节点号，使用 `fseek` 和 `fread` 读取相应索引节点的信息。索引节点结构为 `ext2_inode`。

格式化时间信息： 使用 `asctime` 函数将索引节点中的时间信息格式化为易读的字符串。

输出文件或目录信息： 将文件或目录的类型、名称、创建时间、最后访问时间、修改时间等信息输出到控制台。

关闭文件系统文件： 关闭打开的文件系统文件，结束函数执行。

总体来说，`Ls` 函数通过遍历当前目录的目录项，读取相关信息，然后格式化输出到控制台，实现了列出当前目录文件和子目录信息的功能。

```
int Password()
{
    char psw[16], ch[10];
    printf("Please input the old password\n");
    scanf("%s", psw);
    if (strcmp(psw, group_desc.psw) != 0)
    {
        printf("Password error!\n");
        return 1;
    }
    while (1)
    {
        printf("Please input the new password:");
        scanf("%s", psw);
        while (1)
        {
            printf("Modify the password?[Y/N]");
            scanf("%s", ch);
            if (ch[0] == 'N' || ch[0] == 'n')
            {
                printf("You canceled the modify of your password\n");
                return 1;
            }
            else if (ch[0] == 'Y' || ch[0] == 'y')
            {
                strcpy(group_desc.psw, psw);
                f = fopen(PATH, "r+");
                fseek(f, 0, 0);
                fwrite(&group_desc, sizeof(ext2_group_desc), 1, f);
                fclose(f);
                return 0;
            }
            else
            {
                printf("Meaningless command\n");
            }
        }
    }
}
```

获取旧密码：函数开始时，要求用户输入旧密码，并将输入的密码存储在字符数组 psw 中。

比对密码：通过 strcmp 函数比较输入的旧密码与存储在全局变量 group_desc 中的密码 (group_desc.psw) 是否一致。如果密码一致，继续执行下一步。如果密码不一致，打印密码错误的提示信息，并返回 1，表示密码错误。

修改密码：进入一个无限循环，提示用户输入新密码，并将输入的新密码存储在字符数组 psw 中。用户可以选择是否确认修改密码，输入 'Y' 或 'N'。如果输

入 'N', 表示取消修改密码, 函数返回 1。如果输入 'Y', 表示确认修改密码, 将新密码存储在 group_desc.psw 中, 并将更新后的组描述符写回文件系统文件。

结束循环: 循环结束后, 返回 0, 表示密码修改成功。

总体来说, Password 函数实现了用户修改密码的功能。用户需要输入旧密码进行身份验证, 然后可以选择输入新密码并确认修改, 或取消修改密码。函数通过读写文件系统文件, 更新密码信息。

```
int Delet(int type, ext2_inode *current, char *name)
{
    FILE *fout = NULL;
    int i, j, t, k, flag;
    // int Nlocation, Elocation, Blocation,
    int Blocation2, Blocation3;
    int node_location, dir_entry_location, block_location, e_location;
    int block_location2, block_location3;
    ext2_inode cinode, tmpinode, einode;
    ext2_dir_entry bentry, centry, dentry, eentry;
    //一个空条目
    dentry.inode = 0;
    dentry.rec_len = sizeof(ext2_dir_entry);
    dentry.name_len = 0;
    dentry.file_type = 0;
    strcpy(dentry.name, "");
    dentry.dir_pad = 0;

    fout = fopen(PATH, "r+");
    t = (int)(current->i_size / dirsiz); //总条目数
    flag = 0; //是否找到文件或目录
    for (i = 0; i < t; i++)
    {
        dir_entry_location = dir_entry_position(i * dirsiz, current->i_block);
        fseek(fout, dir_entry_location, SEEK_SET);
        fread(&centry, sizeof(ext2_dir_entry), 1, fout);
        if ((strcmp(centry.name, name) == 0) && (centry.file_type == type))
        {
            flag = 1;
            j = i;
            break;
        }
    }
    if (flag)
    {
        node_location = centry.inode; //inode号
        fseek(fout, 3 * blocksiz + node_location * sizeof(ext2_inode), SEEK SET); //定位INODE 位置
```

打开文件: 函数开始时, 打开文件系统文件以便进行读写操作。

查找目标文件或目录: 通过循环遍历当前目录的目录项, 查找指定名称和类型的文件或目录。如果找到目标文件或目录, 获取其相关信息, 包括索引节点号和数据块号。如果未找到目标, 关闭文件, 返回 1 表示删除失败。

递归删除目录内容: 如果找到的目标是目录, 需要递归删除目录中的所有文

件和子目录。通过循环读取目录中的条目，获取文件或子目录的信息，然后递归调用 Delet 函数。

释放数据块和索引节点：根据文件或目录的信息，释放其占用的数据块和索引节点。对于文件，释放直接指向的块、一级索引块和二级索引块。对于目录，递归删除子目录后，释放其占用的数据块和索引节点。

更新当前目录信息：删除目标文件或目录后，需要更新当前目录的相关信息。如果删除的是目录，还需要更新当前目录的大小和数据块信息。

关闭文件：操作完成后，关闭文件系统文件。

返回结果：返回 0 表示删除成功，返回 1 表示删除失败。

总体来说，Delet 函数实现了文件和目录的删除功能。函数会根据文件类型采取不同的处理方式，包括递归删除目录内容、释放数据块和索引节点等操作。最后，更新当前目录的信息。

```
int Delet(int type, ext2_inode *current, char *name)
{
    FILE *fout = NULL;
    int i, j, t, k, flag;
    // int Nlocation, Elocation, Blocation,
    int Blocation2, Blocation3;
    int node_location, dir_entry_location, block_location, e_location;
    int block_location2, block_location3;
    ext2_inode cinode, tmpinode, einode;
    ext2_dir_entry bentry, centry, dentry, eentry;
    // 一个空条目
    dentry.inode = 0;
    dentry.rec_len = sizeof(ext2_dir_entry);
    dentry.name_len = 0;
    dentry.file_type = 0;
    strcpy(dentry.name, "");
    dentry.dir_pad = 0;

    fout = fopen(PATH, "r+");
    t = (int)(current->i_size / dirsiz); // 总条目数
    flag = 0;                          // 是否找到文件或目录
    for (i = 0; i < t; i++)
    {
        dir_entry_location = dir_entry_position(i * dirsiz, current->i_block);
        fseek(fout, dir_entry_location, SEEK_SET);
        fread(&centry, sizeof(ext2_dir_entry), 1, fout);
        if ((strcmp(centry.name, name) == 0) && (centry.file_type == type))
        {
            flag = 1;
            j = i;
            break;
        }
    }
    if (flag)
    {
        node_location = centry.inode; // inode号
        fseek(fout, 3 * blocksiz + node_location * sizeof(ext2_inode), SEEK_SET); // 定位INODE 位置
    }
}
```

打开文件：函数开始时，打开文件系统文件以便进行读取操作。

获取当前目录的.和..目录项：通过读取当前目录的.和..目录项，获取当前目录和上一级目录的信息，包括索引节点号。

递归获取路径：如果当前目录不是根目录（.目录项的 inode 不等于 ..目录项的 inode），递归调用 pwd 函数。递归调用时，将当前目录切换为上一级目录，传递新的字符串（路径）。

构建路径字符串：在递归的过程中，不断在字符串前面追加目录名和斜杠，构建路径。

关闭文件：操作完成后，关闭文件系统文件。

返回结果：返回构建好的绝对路径字符串。

总体来说，pwd 函数通过递归获取上一级目录的信息，并在递归的过程中构建绝对路径字符串。如果当前目录是根目录，直接返回根目录的路径。最终，返回获取到的绝对路径。

3.5 程序运行初值及运行结果分析

create:

创建文件和目录

```
[.]> create f myfile
Congratulations! myfile is created

[.]> ls
Type          FileName      CreateTime          LastAccessTime      ModifyTime
!!!!!!!current->i_size:96
Directory     .             Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
Directory     ..            Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
File          myfile        Thu Nov 30 10:31:01 2023  Thu Nov 30 10:31:01 2023  Thu Nov 30 10:31:01 2023

[.]> create d dir
.dir created.
..dir created.
Congratulations! dir is created

[.]> ls
Type          FileName      CreateTime          LastAccessTime      ModifyTime
!!!!!!!current->i_size:128
Directory     .             Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
Directory     ..            Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
File          myfile        Thu Nov 30 10:31:01 2023  Thu Nov 30 10:31:01 2023  Thu Nov 30 10:31:01 2023
Directory     dir           Thu Nov 30 10:31:07 2023  Thu Nov 30 10:31:07 2023  Thu Nov 30 10:31:07 2023
```

delete:

删除文件

```
[.]> delete f myfile
num:4

last entryname: dir

last entrynamelen: 3
num:3
Congratulations! myfile is deleted!

[.]> ls
Type          FileName      CreateTime          LastAccessTime      ModifyTime
!!!!!!!current->i_size:96
Directory     .             Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
Directory     ..            Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023  Thu Nov 30 10:30:04 2023
Directory     dir           Thu Nov 30 10:31:07 2023  Thu Nov 30 10:31:07 2023  Thu Nov 30 10:31:07 2023
```

删除目录

```
[.]> ls
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
!!!!!!!current->i_size:64				
Directory	.	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023
Directory	..	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023

创建一个目录 dir，然后在目录 dir 里边创建新的目录和文件，最后在上级目录中直接删除目录 dir

```
o [root@localhost ~]# ./hello
Hello! Welcome to Ext2_like file system!
please input the password(init:123):123

[.]> create d dir
.dir created.
..dir created.
Congratulations! dir is created

[.]> cd dir

[dir]> create d mydir
.dir created.
..dir created.
Congratulations! mydir is created

[dir]> create f myfile
Congratulations! myfile is created

[dir]> ls
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
!!!!!!!current->i_size:128				
Directory	.	Thu Nov 30 10:37:45 2023	Thu Nov 30 10:37:45 2023	Thu Nov 30 10:37:45 2023
Directory	..	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023
Directory	mydir	Thu Nov 30 10:37:59 2023	Thu Nov 30 10:37:59 2023	Thu Nov 30 10:37:59 2023
File	myfile	Thu Nov 30 10:38:05 2023	Thu Nov 30 10:38:05 2023	Thu Nov 30 10:38:05 2023

```

[dir]> cd ..

[.]> delete d dir
num:4
last entryname: myfile
last entrynamelen: 6
num:3
The dir is deleted!Congratulations! dir is deleted!

[.]> ls
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
!!!!!!!current->i_size:64				
Directory	.	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023
Directory	..	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023	Thu Nov 30 10:30:04 2023

close:

```
[.]> ls
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
!!!!!!!current->i_size:64				
Directory	.	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023
Directory	..	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023

```

[.]> create d dir
.dir created.
..dir created.
Congratulations! dir is created

[.]> ls
```

Type	FileName	CreateTime	LastAccessTime	ModifyTime
!!!!!!!current->i_size:96				
Directory	.	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023
Directory	..	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023	Thu Nov 30 10:42:24 2023
Directory	dir	Thu Nov 30 10:45:03 2023	Thu Nov 30 10:45:03 2023	Thu Nov 30 10:45:03 2023

```

[.]> cd dir

[dir]> create d mydir
.dir created.
..dir created.
Congratulations! mydir is created

[dir]> cd mydir

[mydir]> close 2

[.]> █
```


read

```
[.]> ls
Type          FileName      CreateTime      LastAccessTime      ModifyTime
!!!!!!!current->i_size:128
Directory     .             Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023
Directory     ..            Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023
Directory     dir           Thu Nov 30 10:45:03 2023    Thu Nov 30 10:45:26 2023    Thu Nov 30 10:45:03 2023
File          file         Thu Nov 30 10:49:10 2023    Thu Nov 30 10:49:49 2023    Thu Nov 30 10:49:49 2023

[.]> read file

hello world!!!
This is OS lab

[.]> █
```

write

```
[.]> ls
Type          FileName      CreateTime      LastAccessTime      ModifyTime
!!!!!!!current->i_size:128
Directory     .             Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023
Directory     ..            Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023    Thu Nov 30 10:42:24 2023
Directory     dir           Thu Nov 30 10:45:03 2023    Thu Nov 30 10:45:26 2023    Thu Nov 30 10:45:03 2023
File          file         Thu Nov 30 10:49:10 2023    Thu Nov 30 10:50:44 2023    Thu Nov 30 10:49:49 2023

[.]> write file

now file is been writing

[.]> read file

hello world!!!
This is OS lab
now file is been writing

[.]> █
```

password (更改密码)

```
[.]> password
Please input the old password
1
Please input the new password:123
Modify the password?[Y/N]y

[.]> password
Please input the old password
123
Please input the new password:1
Modify the password?[Y/N]y

[.]> ^C
[root@localhost ~]# gcc hello.c -o hello
[root@localhost ~]# ./hello
Hello! Welcome to Ext2_like file system!
please input the password(init:123):123
Wrong password!It will terminate right away.
Thank you for using~ Byebye!
[root@localhost ~]# gcc hello.c -o hello
[root@localhost ~]# ./hello
Hello! Welcome to Ext2_like file system!
please input the password(init:123):1

[.]> █
```

format (格式化)

```
[.]> format
Do you want to format the filesystem?
It will be dangerous to your data.
[Y/N]y

!!!!!!!inode.i_size:64

[.]> ls
Type          FileName      CreateTime      LastAccessTime      ModifyTime
!!!!!!!current->i_size:64
Directory     .              Thu Nov 30 10:57:41 2023      Thu Nov 30 10:57:41 2023      Thu Nov 30 10:57:41 2023
Directory     ..             Thu Nov 30 10:57:41 2023      Thu Nov 30 10:57:41 2023      Thu Nov 30 10:57:41 2023

[.]> █
```

exit (退出)

```
[.]> exit
Do you want to exit from filesystem?[Y/N]
t

please input [Y/N]
Do you want to exit from filesystem?[Y/N]
y
Thank you for using~ Byebye!
[root@localhost ~]# █
```

pwd (寻找当前目录的绝对路径)

```
[.]> create d dir
.dir created.
..dir created.
Congratulations! dir is created

[.]> cd dir

[dir]> create d mydir
.dir created.
..dir created.
Congratulations! mydir is created

[dir]> cd mydir

[mydir]> cd mydir
path input error!

[mydir]> create d i
.dir created.
..dir created.
Congratulations! i is created

[mydir]> cd i

[i]> pwd
/dir/mydir/i

[i]> █
```

3.6 实验总结

3.6.1 实验中的问题与解决过程

1. 缺乏整体意识，没有全局观。看完 ppt 介绍后对于如何实现没有一个系统的认识，在实验的过程总是担心会缺少某一步进而导致严重的问题（比如内存泄漏）

解决过程：翻阅资料，询问同学

[模拟实现 EXT2 文件系统-CSDN 博客](#)

[EXT2 文件系统 - AlanTu - 博客园 \(cnblogs.com\)](#)

[Linux 内核之 IO2: EXT 文件系统详解\(案例解析\).ext2 文件系统实现-CSDN 博客](#)

[深入理解 EXT2 文件系统实现原理 - 知乎 \(zhihu.com\)](#)

[Unix/Linux 系统编程自学笔记-第十一章:EXT2 文件系统 - 20191314 汇仁 - 博客园 \(cnblogs.com\)](#)

2.对文件创建，文件读写，获取最新时间等功能不熟悉。寻址能力弱，容易把绝对寻址，直接寻址，间接寻址弄混

解决过程：翻阅资料，询问同学，编写简单的程序进行测试

[C 库函数 – fread\(\) | 菜鸟教程 \(runoob.com\)](#)

[C 中的 fseek 函数使用 - 极客星云 - 博客园 \(cnblogs.com\)](#)

[localtime\(\)函数：获取当前时间和日期并转换为本地时间_localtime 转 time-CSDN 博客](#)

3.6.2 实验收获

1. **理解文件系统的基本概念：**我深入了解文件系统的基本概念，包括磁盘块、inode、目录等。我将理解这些概念如何组成文件系统的基本结构。
2. **实际实现文件系统：**我自己动手实现 ext2 文件系统的一部分或整体。这种亲身经历有助于加深对文件系统内部机制的理解。在实现的过程中，我感觉自己是在写一个巨型的数据结构，同时也对一些用到的函数掌握程度更深入，比如 fopen，fseek，fwrite，fread 等函数，总的来说，这次实验让我收获颇丰。
3. **处理磁盘块分配和管理：**在实验中，我学会了如何分配和管理磁盘块。这涉及到空闲块的追踪、分配和释放，这是一个文件系统中重要的任务。
4. **inode 的理解和管理：**我进一步熟悉 inode 的概念，了解如何创建、读取和修改 inode。这对于文件和目录的管理至关重要。
5. **目录结构和管理：**我学到了如何设计和维护目录结构，包括创建、读取和删除目录项。这是构建文件系统的关键组成部分。
6. **文件的创建、读取和写入：**我了解到文件的创建、读取和写入操作是如何在文件系统中

进行的。这些操作涉及到磁盘块的分配和释放，以及对 inode 的操作。

3.6.3 意见与建议

意见：ppt 指导书上对于单个函数模块的讲解比较细致，但是对于整个系统的宏观描述以及各个模块之间的调用关系描述比较少，以至于看完 ppt 之后知道怎么写单个的模块，但是对于全局的认识很模糊

建议：增加对文件系统全局的讲解，将函数模块黑盒化来描述全局

