西安交通大學
XI'AN JIAOTONG UNIVERSITY

2024-2025 学年第一学期

《操作系统专题实验》

实 验 报 告

学　　院：　电子与信息学院

班　　级：

学　　号：

姓　　名：

二〇二四年 十二月

实验一 进程、线程相关编程实验

1.1 进程相关编程实验

1.1.1 实验目的

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

1.1.2 实验内容

（1）熟悉操作命令、编辑、编译、运行程序。完成图 1-1 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。

```
[root@kp-test1 1.1]# ./pid1
parent: pid = 2940
child: pid = 0
parent: pid1 = 2939
child: pid1 = 2940
[root@kp-test1 1.1]# ./pid1
parent: pid = 2942
child: pid = 0
parent: pid1 = 2941
child: pid1 = 2942
[root@kp-test1 1.1]# ./pid1
parent: pid = 2944
child: pid = 0
parent: pid1 = 2943
child: pid1 = 2944
```

1-1 中代码进行一个简单的 fork 调用，创建一个子进程。可以从结果中看到输出将分为两部分：子进程和父进程的输出。

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-test1 1.1]# ./pid1
child: pid = 0
parent: pid = 2962
child: pid1 = 2962
parent: pid1 = 2961
[root@kp-test1 1.1]# ./pid1
parent: pid = 2965
child: pid = 0
parent: pid1 = 2964
child: pid1 = 2965
```

从结果中看出，由于 fork() 后父子进程是并发的，没有 wait() 来同步，子进程和父进程输出顺序存在很大的不确定性，且每次运行的输出顺序可能不同，这取决于操作系统调度器如何调度父子进程的执行。

同时，由于父进程不会等待子进程结束，可能导致子进程成为孤儿进程，虽然这不会影

响当前程序的功能，无法从输出结果直观体现，但在其他程序中可能会有资源管理方面的影响。

```
[root@kp-test1 ~]# cd /usr/local/src/LAB1
[root@kp-test1 LAB1]# ps  -axj | grep test
   2351    2518    2518    2351 pts/0       2518 S+       0   0:00 ./test
   2518    2519    2518    2351 pts/0       2518 Z+       0   0:00 [test] <defunct>
   2474    2521    2520    2474 pts/1       2520 S+       0   0:00 grep --color=auto test
```

可以看出，让父进程 sleep(20)不对子进程进行 wait() 后，运行程序，发现子进程运行结束后变成了僵尸进程(Z+)。

步骤三：修改图 1-1 中代码，增加一个全局变量并在父子进程中对其进行不同的操作，观察并解释所做操作和输出结果。

```
[root@kp-test1 1.1]# gcc -o pid1 1_pid.c
[root@kp-test1 1.1]# ./pid1
parent: pid = 2675
child: pid = 0
parent: pid1 = 2674
child: pid1 = 2675
parent: global = 200
child: global = 100
parent: global_address = 42005c
child: global_address = 42005c
```

在修改后的代码中，添加了一个全局变量 global，并且在父进程和子进程中对它分别进行了不同的操作。在子进程中，global 增加了 100，并打印出子进程的 global 值和变量的内存地址。在父进程中，global 增加了 200，并同样打印出父进程的 global 值和变量的内存地址。

由于 fork 调用时，虽然子进程会复制父进程的所有数据，包括全局变量，但复制的结果是两个独立的地址空间，因此父进程和子进程虽然共享同样的代码和变量名，但它们修改的 global 变量实际上是各自进程中的副本，互不影响。　同时由于 fork()产生了独立的地址空间，虽然全局变量名是相同的，但它们在各自进程中的内存地址是相同的，因为它们是复制自父进程的，但值是独立修改的。

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

```
[root@kp-test1 1.1]# gcc -o pid1 1_pid.c
[root@kp-test1 1.1]# ./pid1
parent: pid = 2685
child: pid = 0
parent: pid1 = 2684
child: pid1 = 2685
parent: global = 200
child: global = 100
parent: global_address = 42005c
child: global_address = 42005c
final: global = 101
final: global = 201
```

在父进程和子进程结束后，程序再次对全局变量 global 进行加 1 操作并输出。由于 fork 后，父进程和子进程分别拥有各自的地址空间，因此它们对 global 的修改是独立的，互不影响。

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。

首先建立被调用程序 1_called.c，每次运行该程序时，系统会为它分配一个新的进程 ID，因此每次输出的 PID 可能不同。

```
[root@kp-test1 1.1]# gcc -o system 1_system.c
[root@kp-test1 1.1]# ./system
parent: pid = 2793
parent: pid1 = 2792
Welcome to this Process

The PID of this process  is: 2794
child: pid = 0
child: pid1 = 2793
```

运行时有两个独立的进程运行：父进程和子进程。子进程中通过 system()调用了 1_called 可执行文件，之后又输出自己的 PID。父进程则正常获取了子进程的 pid 并显示。

```
[root@kp-test1 1.1]# ./system
parent: pid = 2810
child: pid = 0
parent: pid1 = 2809
child: pid1 = 2810
Welcome to this Process

The PID of this process  is: 2811
```

若修改为子进程首先获取并打印了自己的 PID，然后使用 system 函数执行外部可执行文件 called。则可以看到父进程在 fork 之后首先执行，输出了父进程的 pid，然后父进程通过 getpid() 获取自己的 PID 并打印。

而子进程在 fork 后返回 0。后又通过 getpid 获取了它自己的 PID，表示这个子进程的真实 PID。再调用 called 创建了一个新的进程执行 called 程序。called 程序输出它自己的 PID，这个 PID 是 system 生成的新进程的 PID，与父子进程无关。

```
[root@kp-test1 1.1]# ./execl
parent: pid = 2824
parent: pid1 = 2823
Welcome to this Process

The PID of this process  is: 2824
```

由于 exec 族函数是专门用来替换当前进程地址空间的，因此一旦成功调用，子进程就会完全被 1_called.c 程序替代，不会执行子进程中的后续代码。

即如上图中，再子进程初始位置调用 exec 族函数，结果就不会执行后续子进程的打印语句。父进程继续执行它的代码，子进程则被替换成了 1_called.c 程序。由于 1_called.c 取代了子进程，因此可以看出它的 PID 与子进程相同。

```
[root@kp-test1 1.1]# ./execl
parent: pid = 2854
child: pid = 0
parent: pid1 = 2853
child: pid1 = 2854
Welcome to this Process

The PID of this process  is: 2854
```

将 exec 族函数放置在子进程结束位置，可以看出程序按照预期正常执行，子进程的打印语句没有被所调用的 called 可执行文件。

总结：

通过上述实验过程，可以看出 system()和 exec 族函数在功能、用法和效果上有显著的不同。在进行进程控制时，选择 system()或 exec 族函数取决于具体需求。

system()会创建一个子进程来执行给定的命令，通过调用相应的命令解释来解析并执行字符串命令。执行完后，如果没有特别的设置父进程会等待子进程结束。并且返回子进程的退出状态码

exec 族函数用于在当前进程中替换其地址空间为新的程序，即直接用新程序替换当前进程的映像。果执行成功，后续的代码不会被执行；如果执行失败，则会返回控制到原始进程，并可以进行错误处理。

1.1.5 测试数据设计

本部分无需数据测试。

1.1.6 程序运行初值及运行结果分析

运行结果及结果分析详见 1.1.4 部分。

1.1.7 实验总结

1.1.7.1 实验中的问题与解决过程

（1）使用 wait(NULL)函数，但没有包含<sys/wait.h>头文件以及在调用 system()函数时并未包含<stdlib.h>头文件，导致编译器发出"implicit declaration of function"警告。后续经过查询，在代码中加入相应头文件后解决此问题。

（2）在尝试在子进程中调用 system()和 exec 族函数时，由于对于其包含的参数内容不熟悉，导致函数报错，后通过查阅文档和网络教程，理解了其基本用法并应用它们。

1.1.7.2 实验收获

通过这个实验，我对进程管理的理解得到了深化。在 Linux 系统中，进程的创建、管理和调度的机制变得更加清晰，特别是在观察 wait()函数的行为后，我认识到父子进程间同步的重要性。此外，这个实验提升了我的编程技巧，尤其是涉及系统级调用和进程管理的函数，使我掌握了如何在程序中执行系统命令，以及如何用 exec 族函数有效替换当前进程的执行内容。最后，通过在一个程序中创建多个进程并管理它们的行为和状态，我对多进程编程模型有了更实际的认识和理解。

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

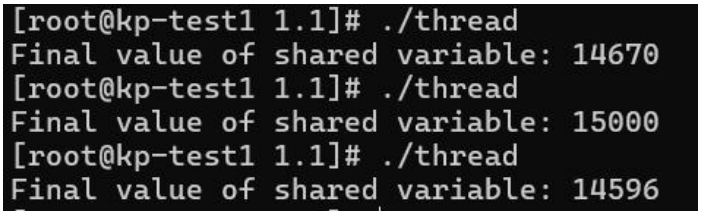
（1）在进程中给一变量赋初值并成功创建两个线程；
（2）在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果；
（3）多运行几遍程序观察运行结果，如果发现每次运行结果不同，请解释原因并修改程序解决，考虑如何控制互斥和同步；
（4）将任务一中第一个实验调用 system 函数和调用 exec 族函数改成在线程中实现，观察运行结果输出进程 PID 与线程 TID 进行比较并说明原因。

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。



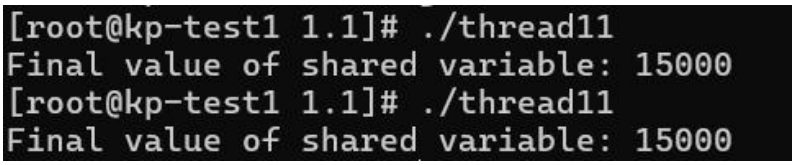
设计多线程程序并由其中两个线程分别对一个共享变量 shared_variable 进行操作。由于两个线程都在修改同一个共享变量而没有使用任何同步机制，这可能导致数据竞争，最终的输出结果会变得不可预测。

其中两个函数：op1() 函数会将 shared_variable 增加 1，循环 5000 次，总共增加 5000；op2() 函数会将 shared_variable 增加 2，循环 5000 次，总共增加 10000。再使用 pthread_create() 创建两个线程 thread1 和 thread2，分别执行 op1() 和 op2()。

由于 shared_variable 是在没有同步机制的情况下被两个线程并发修改，因此最终结果是不确定的。理论上，最终结果应该是 15000。但是，由于可能的竞争条件，最终结果可能会小于或等于 15000，甚至可能不稳定，即如上图结果所示。

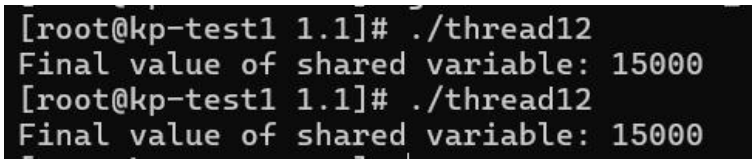
步骤二：修改程序，定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。运行程序，观察最终共享变量的值。

根据所学知识，同步在多线程编程中指的是协调多个线程的执行，以确保它们能够正常、可预测地访问共享资源或完成某些任务。互斥则是同步的一种形式，确保一次只有一个线程能访问某个特定资源或代码段，通常通过互斥锁来实现，但也可以通过信号量等其他机制实现。



上图为使用互斥锁来控制共享变量的访问与互斥，达到了稳定输出 15000 数值的结果，具体代码在附录中体现，使用互斥锁能够保证多个线程在访问共享资源时的安全性，避免数据竞争，从而使程序的输出结果稳定。

信号量比互斥锁更为通用。在信号量上下文中，P 操作用于申请或等待资源，V 操作用于释放或发出信号。P 操作会检查信号量的值，若大于 0，信号量的值减 1 并继续执行；若为 0，线程将被阻塞，直到信号量值大于 0。V 操作会增加信号量的值，若有线程因 P 操作阻塞在该信号量上，一个或多个线程将被解除阻塞。



使用 sem_t semaphore 定义了一个信号量，并通过 sem_init()初始化信号量，初始值为 1，表示可以让一个线程访问共享变量。在 op1()和 op2()两个操作中，使用 sem_wait()实现 P 操作，线程在操作共享变量前等待信号量，再使用 sem_post()实现 V 操作，线程在操作完后释放信号量。

由两个线程分别执行 op1() 和 op2()，对共享变量进行操作。并在程序结束时，通过 sem_destroy() 销毁信号量，释放资源。由于使用了信号量进行共享变量的互斥访问，可以看到最终输出结果将是稳定的。

通过使用信号量实现对共享变量的互斥访问，可以避免数据竞争问题，并确保最终结果的正确性。信号量提供了一种比互斥锁更灵活的同步机制，在更复杂的并发程序中，信号量

也可以用来控制多个线程或进程对共享资源的访问。

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。

通过修改 Part1 中 exec 族函数和 system()调用部分的代码改为在线程中执行。建立如下图所示的 op 线程函数。

```
void *op(void *params) {
    pthread_t tid = pthread_self();
    printf("Inner thread: tid = %ld\n", tid);
    execlp("./called", NULL);
    //system("./called");

    pthread_exit(0);
}
```

这个线程函数 op 是一个用来执行外部程序的函数，在线程上下文中被调用。

它调用了 pthread_self()函数，用于获取当前线程的 ID，并将其存储在变量 tid 中，并打印当前线程的 TID，帮助跟踪和理解程序在多线程环境下的执行过程。

同时，这个线程函数的主要目的是在子线程中执行外部程序 called。通过 execl 族函数或是 system()调用替换当前进程。

与 Part1 中相似，若将 exec 族函数放置在子进程初始位置，可以发现如下图所示。父进程首先输出自己的 PID。随后加入子进程的 execlp("./called", NULL)一旦执行成功，它将完全替换子进程，因此不会执行剩下的代码。

```
[root@kp-test1 1.1]# ./th1
parent: pid = 3571
parent: pid1 = 3570
Inner thread: tid = 2814621872911O4
Welcome to this Process

The PID of this process  is: 3571
```

在线程中调用了 execlp("./called", NULL)运行./called 可执行文件。可以看出 execlp()函数替换的是整个进程，因此整个子进程会变为./called，子进程的线程也就终止了，并接管了原本子进程的地址所在。

若将 exec 族函数部分重新放置于子进程的结束位置，与 Part1 中类似，观察结果如下。

```
[root@kp-test1 1.1]# ./th1
parent: pid = 3662
child: pid = 0
parent: pid1 = 3661
child: pid1 = 3662
Inner thread: tid = 281473223946720
Welcome to this Process

The PID of this process  is: 3662
```

可以看到子进程创建了一个线程，调换位置后的线程函数获取当前线程的 TID 并打印。在线程内部调用 execlp("./called", NULL)来替换当前子进程的地址空间，执行 called 程序。

子进程在 fork()之后，会输出子进程的 PID 和 pid1。在创建线程并执行 op()时，线程的 TID 会被打印。父进程在等待子进程完成之前，会打印父进程的 PID 和 pid1。父进程在

子进程结束后，通过 wait(NULL)回收子进程的资源。

同 exec 族函数部分在子进程中改用 system()创建一个线程。

可以看出虽然将 system()函数放置在子进程初始位置，其后续代码依旧在 op 操作执行完后执行。system()会创建一个新的子进程来运行命令，因此主线程不会被替换，并不会影响当前线程和进程的正常执行。线程的后续代码依然能够执行，程序会继续输出信息。

```
[root@kp-test1 1.1]#  gcc -o th1 1_thread2.c -lpthread
[root@kp-test1 1.1]# ./th1
parent: pid = 3653
parent: pid1 = 3652
Inner thread: tid = 281458165281248
Welcome to this Process

The PID of this process  is: 3655
child: pid = 0
child: pid1 = 3653
```

同 Part1,我们把线程创建和执行调换到子进程结束位置。相比于 exec 族函数,system()只是在调用它时创建了一个子进程来执行命令，并不会影响当前线程和进程的正常执行。因此与前面结果对比，发现只是输出的顺序改变了。

```
[root@kp-test1 1.1]#  gcc -o th1 1_thread2.c -lpthread
[root@kp-test1 1.1]# ./th1
parent: pid = 3712
parent: pid1 = 3711
child: pid = 0
child: pid1 = 3712
Inner thread: tid = 281461094871520
Welcome to this Process

The PID of this process  is: 3714
```

1.2.5 测试数据设计

本部分无数据进行测试。

1.2.6 程序运行初值及运行结果分析

运行初值及运行结果分析在 1.2.4 部分体现。

1.2.7 实验总结

1.2.7.1 实验中的问题与解决过程

在本次实验中，我遇到了多个问题，涵盖了多线程编程、进程与线程的 ID 差异、以及程序中的同步和互斥等方面。

（1）在 Linux 环境中使用 C 语言创建两个线程，并对共享变量分别进行不同操作。编译过程中遇到了 "undefined reference to 'pthread_create" 的报错。通过查询网络资料发现需要手动链接到 Pthreads 库来完成编译，即在链接阶段加上-lpthread 标志。

（2）在步骤一中多次运行同一个程序会产生不一致的输出。这是因为多个线程在访问和修

改共享变量时没有进行同步，导致了竞态。通过阅读附加的文档，发现可以使用互斥锁或信号量来进行同步解决这一问题。
（3）与 Part1 中不同，多线程环境中使用 system()与 exec 族函数。发现使用 system()调用的 PID 与主线程不同。经过查询相关文档发现，其原因是 system()内部会调用 fork()创建一个新的子进程来执行命令。

1.2.7.2 实验收获

在本次实验中，我学习了如何使用信号量进行同步，以及 PV 操作的具体用法。通过 PV 操作，可以确保在任何时刻只有一个线程能够访问共享变量，从而避免竞态条件，确保共享资源的正确访问。

同时本次实验加深了我对多线程和多进程编程的理解，使我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。此外，Part1 和 Part2 对与 system()函数和 excc 族函数的不同处理使我认识到了进程和线程的区别。应用互斥锁和信号量来解决竞争，增强了程序的健壮性。

实验不仅提升了我的技术能力，还锻炼了我的问题解决能力，使我能冷静认真地面对编译错误以及实验过程中遇到的问题。

1.3 自旋锁实验

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，待补充主函数的示例代码如下：

```
#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;
spinlock_t lock;
// 输出共享变量的值
// 初始化自旋锁
// 创建两个线程
// 等待线程结束
// 输出共享变量的值
return 0;
}
```

步骤二：补充完成代码后，编译并运行程序，分析运行结果。

对 spinlock.c 程序各主要部分代码进行分析。

1.3.4.1 自旋锁的定义和初始化

```
typedef struct {
    int flag;
} spinlock_t;
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}
```

对于自旋锁结构体，首先定义了一个结构体 spinlock_t，其中 flag 成员用于表示自旋锁的状态。flag 的值为 0 则表示锁是未被占用的，为 1 表示锁是被占用的。后对函数进行初始化，将 spinlock_init 函数的 flag 初始化为 0，表示锁在初始化时是可用的。

1.3.4.2 自旋锁的获取和释放

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

由 spinlock_lock 函数使用原子操作__sync_lock_test_and_set 来尝试获取锁：

如果 flag 的值为 0，它会将 flag 设为 1，表示锁已被占用，然后函数返回。

如果 flag 的值为 1，则线程会进入自旋状态，直到锁变为可用。

spinlock_unlock 函数使用__sync_lock_release 将 flag 设为 0，表示释放锁，以确保释放锁的过程是安全的。

1.3.4.3 主函数的实现

```
int main() {
    pthread_t thread1, thread2;
    spinlock_t lock;
    // 初始化自旋锁
    spinlock_init(&lock);
    // 创建两个线程
```

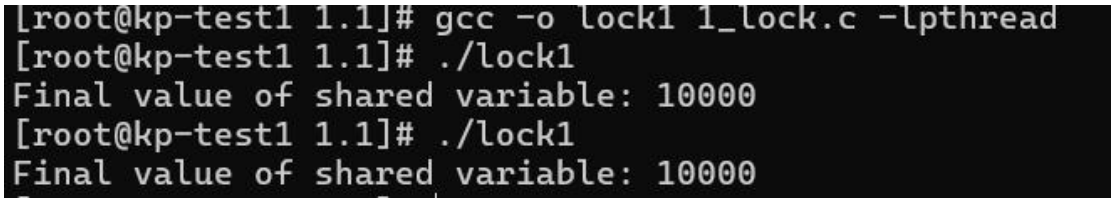
```
    pthread_create(&thread1, NULL, thread_function, &lock);
    pthread_create(&thread2, NULL, thread_function, &lock);
    // 等待线程结束
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    // 输出共享变量的值
    printf("Final value of shared variable: %d\n", shared_value);
    return 0;
}
```

首先定义两个线程 thread1 和 thread2。初始化自旋锁 lock。通过 pthread_create 创建两个线程，分别执行 thread_function，并传入相同的自旋锁 lock。后使用 pthread_join 等待两个线程的执行结束，确保主线程在输出共享变量值之前，两个子线程已完成操作。

最后，输出共享变量 shared_value 的最终值。因为使用了自旋锁来保护对 shared_value 的访问，最终值应为 10000。

通过上述各部分的代码实现，程序模拟了自旋锁的基本操作，确保在多线程环境中对共享资源的安全访问。自旋锁的实现方式虽然简单，但在多核处理器的情况下效率较高，因为它避免了线程上下文切换的开销。然而，自旋锁在长时间等待时可能会导致 CPU 时间的浪费，因此在实际应用中需根据具体情况选择合适的同步机制。

具体执行结果如下所示。



如上图所示结果，使用了自旋锁机制，确保同一时刻只有一个线程可以访问共享变量，因此避免了竞态条件。每个线程对 shared_value 加 5000 次，理论上总共会加 10000 次，图中结果符合预期。

自旋锁的使用确保了线程间对共享资源的安全访问，避免了竞态条件的发生。在这个实验中，通过对自旋锁的模拟，你可以更深入地理解线程同步与互斥的重要性。

1.3.5 测试数据设计

本部分无测试数据。

1.3.6 程序运行初值及运行结果分析

程序运行初值 shared_value 为 0 ，经过实验运行得到为 10000 的结果，与预期相符，验证了自旋锁的设定过程。运行结果分析于 1.3.4 中体现。

1.3.7 实验总结

1.3.7.1 实验中的问题与解决过程

（1）本 Part 遇到的问题较为简单，主要是遇到编译警告称关于 exit 函数未声明。通过查询网络资料，通过包含相应的头文件<stdlib.h>即可解决问题。

1.3.7.2 实验收获

在实验过程中遇到的问题以及解决过程锻炼了我分析问题和调试代码的能力。这使我对调试工具和方法有了更深入的理解，并能更有效地处理类似的问题。同时通过将理论知识应用到实际代码中，我的实际操作能力得到了增强。这不仅包括对各种编程概念的理解，还包括如何在实际环境中实现和测试这些概念。实际操作使我更加熟悉编程语言的特性和常见的开发工具，为我未来的编程实践打下了坚实的基础。

1.4 意见与建议

（1）建议实验中增加更多可能涉及的知识点。实验参考书较简练，可能存在较为难懂的地方。建议参考书中多提供搜索的索引。尽管实验手册提供了基础框架，但更多具体函数的使用示例和文档将有助于学生更好地理解和应用相关知识。

（2）当前实验内容涵盖进程、线程创建和管理，但实际应用中还有更多高级用法，如多进程并发处理、进程通信、管程等。

1.5 附件

程序文件：

第一部分：


1_pid.c


1_called.c


1_system.c


1_execl.c


test_pid.c

第二部分：


1_thread_pv.c


1_thread_lock.c


1_thread_sys_exe.c


1_unlock.c

第三部分：


1_spinlock.c

Readme 文件：


OS_LAB1_README.pdf

实验二 进程通信与内存管理

2.1 进程的软中断通信

2.1.1 实验目的

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，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 。

（3）多次运行所写程序，比较 5s 内按下 Ctrl+\或 Ctrl+Delete 发送中断，或 5s 内不进行任何操作发送中断，分别会出现什么结果？分析原因。
（4）将本实验中通信产生的中断通过 14 号信号值进行闹钟中断，体会不同中断的执行样式，从而对软中断机制有一个更好的理解。

2.1.3 实验思想

(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与进程特性有关的一些参数外,其他都与父进程相同,与父进程共享文本段和打开的文件,并都受进程调度程序的调度。

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,founction;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]； 38 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。

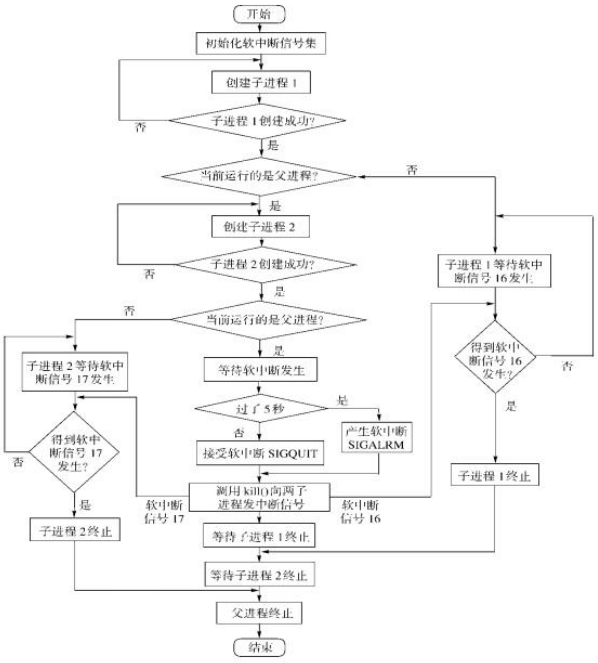


图 2.1 软中断通信程序流程图

2.1.4 实验步骤

见 2.1.2 实验内容部分。

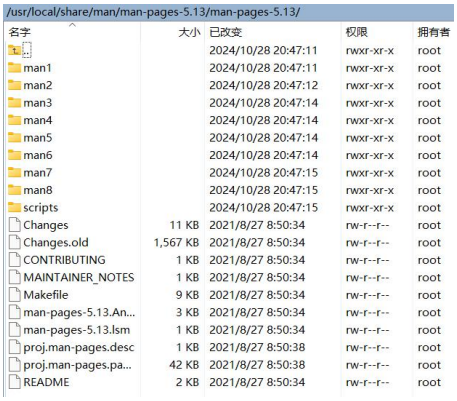
2.1.5 程序运行初值及运行结果分析

步骤一：使用 man 命令查看 fork 、kill 、signal、sleep、exit 系统调用的帮助手册。

由于虚拟机服务器中没有以上各个命令的手册信息，因此需要手动加入来完成查看。在下面网址中下载安装包。以完成相关操作。

http://www.kernel.org/pub/linux/docs/man-pages/man-pages-5.13.tar.xz

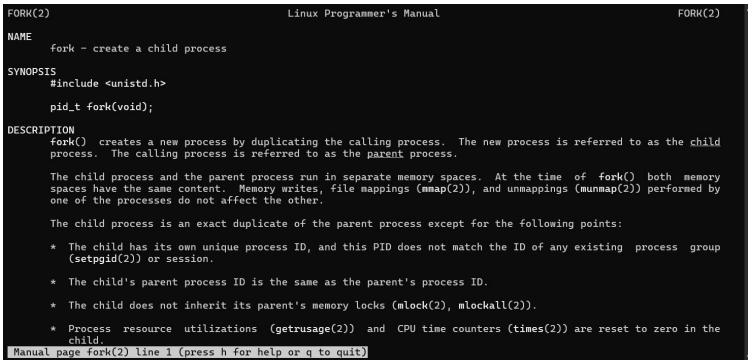
解压缩后，通过 winsSCP 上传至服务器/user/local/share/man 目录下。



名字	大小	已改变	权限	拥有者
..		2024/10/28 20:47:11	rwxr-xr-x	root
man1		2024/10/28 20:47:11	rwxr-xr-x	root
man2		2024/10/28 20:47:12	rwxr-xr-x	root
man3		2024/10/28 20:47:14	rwxr-xr-x	root
man4		2024/10/28 20:47:14	rwxr-xr-x	root
man5		2024/10/28 20:47:14	rwxr-xr-x	root
man6		2024/10/28 20:47:14	rwxr-xr-x	root
man7		2024/10/28 20:47:15	rwxr-xr-x	root
man8		2024/10/28 20:47:15	rwxr-xr-x	root
scripts		2024/10/28 20:47:15	rwxr-xr-x	root
Changes	11 KB	2021/8/27 8:50:34	rw-r--r--	root
Changes.old	1,567 KB	2021/8/27 8:50:34	rw-r--r--	root
CONTRIBUTING	1 KB	2021/8/27 8:50:34	rw-r--r--	root
MAINTAINER_NOTES	1 KB	2021/8/27 8:50:34	rw-r--r--	root
Makefile	9 KB	2021/8/27 8:50:34	rw-r--r--	root
man-pages-5.13.An...	3 KB	2021/8/27 8:50:34	rw-r--r--	root
man-pages-5.13.lsm	1 KB	2021/8/27 8:50:34	rw-r--r--	root
proj.man-pages.desc	1 KB	2021/8/27 8:50:38	rw-r--r--	root
proj.man-pages.pa...	42 KB	2021/8/27 8:50:38	rw-r--r--	root
README	2 KB	2021/8/27 8:50:34	rw-r--r--	root

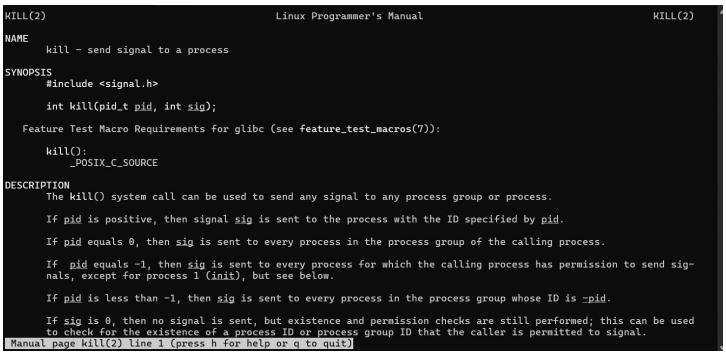
运行 sudo make install 指令，后即可通过 man 指令查看帮助手册。

①man fork 结果



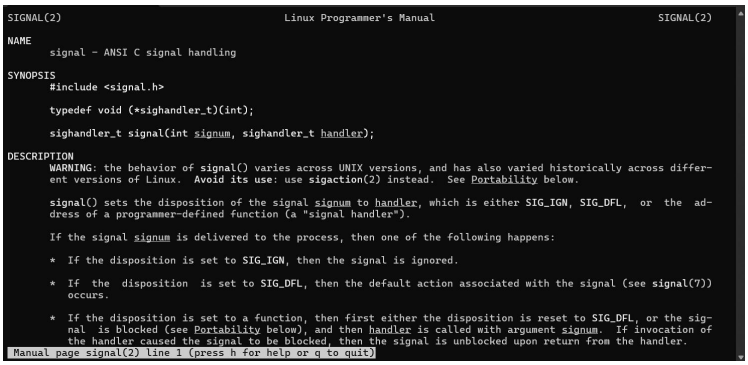
```
FORK(2)                          Linux Programmer's Manual                          FORK(2)

NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates a new process by duplicating the calling process.  The new process is referred to as the child
       process.  The calling process is referred to as the parent process.

       The child process and the parent process run in separate memory spaces.  At the time  of  fork()  both  memory
       spaces have the same content.  Memory writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed by
       one of the processes do not affect the other.

       The child process is an exact duplicate of the parent process except for the following points:

       *  The child has its own unique process ID, and this PID does not match the ID of any existing  process  group
          (setpgid(2)) or session.

       *  The child's parent process ID is the same as the parent's process ID.

       *  The child does not inherit its parent's memory locks (mlock(2), mlockall(2)).

       *  Process  resource  utilizations  (getrusage(2))  and  CPU time counters (times(2)) are reset to zero in the
          child.
 Manual page fork(2) line 1 (press h for help or q to quit)
```

② man kill 结果

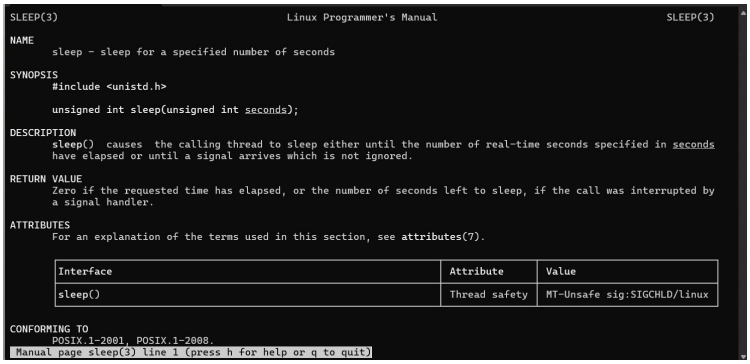


```
KILL(2)                          Linux Programmer's Manual                          KILL(2)

NAME
       kill - send signal to a process

SYNOPSIS
       #include <signal.h>

       int kill(pid_t pid, int sig);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       kill():
           _POSIX_C_SOURCE

DESCRIPTION
       The kill() system call can be used to send any signal to any process group or process.

       If pid is positive, then signal sig is sent to the process with the ID specified by pid.

       If pid equals 0, then sig is sent to every process in the process group of the calling process.

       If  pid equals -1, then sig is sent to every process for which the calling process has permission to send sig-
       nals, except for process 1 (init), but see below.

       If pid is less than -1, then sig is sent to every process in the process group whose ID is -pid.

       If sig is 0, then no signal is sent, but existence and permission checks are still performed; this can be used
       to check for the existence of a process ID or process group ID that the caller is permitted to signal.
 Manual page kill(2) line 1 (press h for help or q to quit)
```

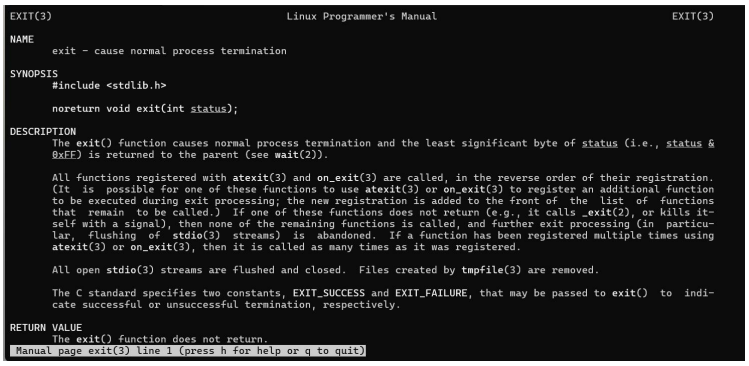
③ man signal 结果



④ man sleep 结果



⑤man exit 结果



步骤二：完成进程的软中断通信相关实验代码。

在 linux 系统下，输入 kill -l 来观察所有的信号以及对应的编号。得到以下的输出结果。

```
[root@localhost man-pages-5.13]# 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
```

发现表中显示在 Linux 系统中，父进程接收的中断信号 SIGINT 和 SIGQUIT 分别为 2

和 3。其中 SIGINT 通常由终端发送，表示用户请求中断进程的执行，通常通过 Ctrl+C 组合键触发。默认行为为终止当前进程。SIGQUIT 也由终端发送，表示用户请求进程退出，并生成核心转储文件。以及 SIGCHLD 是信号编号 17，通知父进程子进程的状态变化，通常用于子进程的清理工作。

下面补充完整代码，观察输出结果。首先解释代码实现逻辑，定义全局变量 pid1 和 pid2，初始化为 -1。再定义处理父进程接收到的中断信号，检查子进程的 ID 是否有效。向子进程 1 发送信号 16，向子进程 2 发送信号 17，表示杀死这两个子进程。设置两个函数分别处理子进程 1 收到信号 16 时的行为和子进程 2 收到信号 17 时的行为。进行输出消息并调用 exit(0)退出子进程。

主函数部分使用 signal()设置信号处理函数。先创建第一个子进程，使用 fork()创建子进程，父进程将 pid1 更新为子进程的 ID。后在父进程中，创建第二个子进程，同样使用 fork()，更新 pid2。为了增加直观性，父进程在创建子进程后，打印消息，等待 5 秒。后父进程调用 wait(NULL)等待两个子进程结束，并在结束后输出总结消息。子进程 1 和子进程 2 分别设置对应的信号处理，并使用 pause()等待信号。

步骤三：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，并分析原因。

首先观察在 5s 内按下中断信号的结果。

```
[root@localhost lab2]# ./kill
Start waiting for 5 seconds...
Child process 1 is ready
Child process 2 is ready
^C
Received soft interrupt
Sent 16 signal successfully
Sent 17 signal successfully

Parent process is killed!!
[root@localhost lab2]# ./kill
Start waiting for 5 seconds...
Child process 1 is ready
Child process 2 is ready
^C
Received soft interrupt
Sent 16 signal successfully
Sent 17 signal successfully
Child process 2 is killed by parent!!
Child process 1 is killed by parent!!

Parent process is killed!!
```

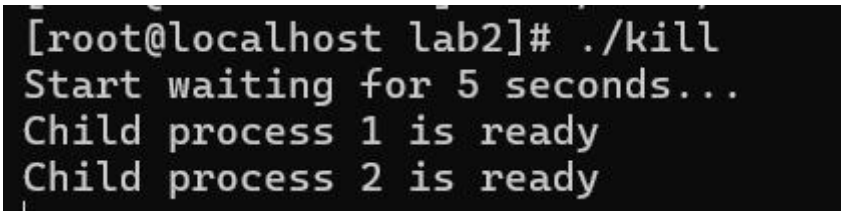
可以发现，当运行 kill 时，程序首先在父进程中打印 Start waiting for 5 seconds...，然后调用 sleep(5)使父进程暂停 5 秒。在父进程中创建两个子进程，在创建时，各个子进程分别会打印 Child process 1 is ready 和 Child process 2 is ready，表明两个子进程已经准备好并处于等待状态，准备接收信号。

在程序运行期间，通过按下 Ctrl + C 来中断程序。此时，父进程捕获到这个软中断信号，打印 Received soft interrupt，然后通过 kill(pid1, 16)向子进程 1 发送信号 16，再通过 kill(pid2, 17)向子进程 2 发送信号 17。此时，父进程会打印相关信息表明信号已成功发送。

但是子进程响应数据输出的顺序可能因为操作系统调度的不同而有所不同，但最终都会输出这两条信息。

最后父进程等待子进程结束，在所有子进程都终止后，输出 Parent process is killed!!，并结束自己的执行。

但如果观察五秒内不进行任何操作，则保持如下结果。



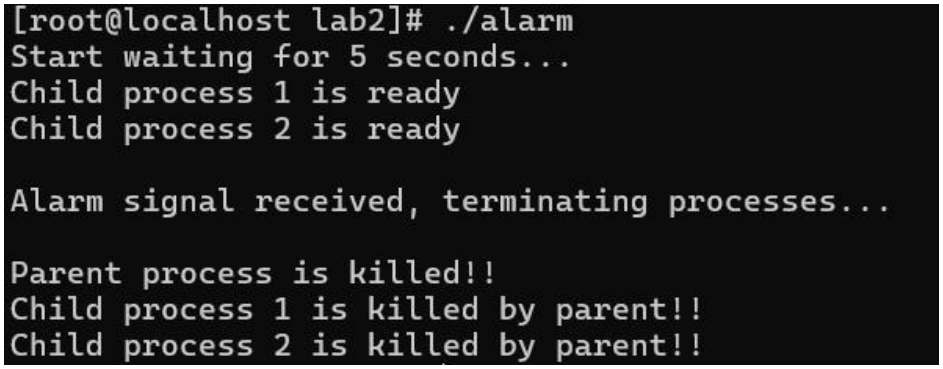
这是因为没有添加超时处理且父进程有阻塞状态，父进程必须等待子进程结束，而子进程始终在等待信号，因此程序将一直保持当前状态。

步骤四：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。

为了处理步骤三中，如果没有收到相关中断信号程序堵塞保持当前状态的问题。在代码部分中进行改进。添加一个信号处理函数 handle_alarm(int sig)，用于处理闹钟信号 SIGALRM，在接收到信号后向两个子进程发送相应的信号终止它们。

首先在父进程中调用 alarm(5)，设置在 5 秒后发送 SIGALRM 信号。使用 pause()让父进程进入等待状态，直到收到信号。再添加对 SIGALRM 的捕获，使得父进程能够在设定时间到达时接收闹钟信号并处理。

得到如下结果。



对比不同中断的执行样式。发现软中断，如 SIGINT 和 SIGQUIT 是通过用户按键触发，父进程能够实时响应这些信号并相应地终止子进程。而闹钟中断则通过定时器触发，表现为在设定时间后自动发送信号，父进程接收到后，能够批量终止子进程。

步骤五：相关问题回答。

（1）最初认为运行结果会怎么样？

我猜测在运行程序时，父进程会等待 5 秒，期间如果按下 Ctrl+C，发送 SIGINT 或 Ctrl+\ 发送"SIGQUIT，父进程会接收到信号，然后向两个子进程发送信号 16 和 17，分别终止它们。子进程应该会打印出相应的退出消息。

（2）实际的结果是什么样的？

在实际运行中如果在 5 秒内按下中断键，父进程会立即接收到信号并向子进程发送信号，子进程会输出相关显示信息并退出。如果 5 秒内没有按下任何中断键，父进程将接收到 SIGALRM 信号，触发闹钟中断，随后同样向子进程发送信号，子进程也会退出。

在接收不同中断前后，程序的响应时间不同：前者是实时响应，后者是基于定时机制。5 秒内中断会直接导致父进程和子进程之间的立即交互，而 5 秒后中断则显示了定时器的工作机制。

（3）改为闹钟中断后，程序运行的结果是什么样子？

改为闹钟中断后，程序的运行结果表现为在设置的 5 秒后，父进程接收到 SIGALRM，打印出相关输出，并向两个子进程发送信号，子进程输出相应的退出消息。与之前的软中断相比，闹钟中断的响应是基于时间的，而不是用户输入的。此外，父进程在没有用户干预的情

况下仍能够主动终止子进程，表现出更高的自动化。

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

kill 命令在程序中使用了 3 次。第一次在 handle_parent_interrupt 中，向子进程 1 发送信号 16。作用是终止子进程 1。执行后，子进程 1 会输出 Child process 1 is killed by parent!!并退出。第二次同样在 handle_parent_interrupt 中，向子进程 2 发送信号 17。作用是终止子进程 2。执行后，子进程 2 会输出 Child process 2 is killed by parent!!后退出。第三次在 handle_alarm 中，向 pid1 和 pid2 发送信号，终止两个子进程，执行后的现象与前两次相同。

（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？

进程主动退出可以通过调用 exit()函数来主动退出，或者通过返回主函数来完成退出。这种方式通常会执行清理操作，如释放资源、关闭文件描述符等。外部杀死进程则使用 kill 命令从外部终止进程，但这可能导致未完成的操作被中断，资源未被清理。

比较之下，主动退出更好，因为它允许进程执行必要的清理工作，确保系统资源得到妥善管理。外部杀死进程在紧急情况下可能是必要的，但它可能导致数据丢失或资源泄漏，因此应谨慎使用。

2.1.6 实验总结

2.1.6.1 实验中的问题与解决过程

首先是 man 手册安装。由于服务器中无法搜索到相关内容，且指导书中没有获得解决方法的渠道，因此查询网络后，在 http://www.kernel.org/pub/linux/docs/man-pages/ 下载源码，并使用正确相关命令，成功在服务器上安装 man 手册页进行查询。

在 Part1 中，发送信号后，进程的状态未能清晰地显示，导致输出信息不完整。后来通过在信号处理函数中添加详细的输出信息，确保在发送信号后打印确认信息。检查子进程的状态，并在输出之前确保状态已更新的方法解决相关问题。同样在 Part1 中，未能按预期捕获闹钟信号，如果程序中未能正确设置"SIGALRM"，父进程则无法响应闹钟。后来在调用 alarm()后，程序能正确进入等待状态。

2.1.6.2 实验收获

实验中我经历了许多挫折，比如信号处理、进程管理等方面的困惑。但正是这些挑战让我意识到，解决问题的过程本身就是一种学习。每次遇到问题，我都会尝试多种方法，查阅资料，或向他人请教。这种主动的学习态度让我不断进步，也让我更加熟悉了相关知识。。在这个实验中，我通过实践加深了对信号机制和进程管理的理解。这不仅提升了我的编程能力，也让我对计算机系统有了更全面的认识。我意识到，实验和实际应用是学习的重要组成部分，实践能够将理论知识与实际技能相结合。希望在未来的学习和工作中，继续保持这种态度，迎接更多的挑战！

2.2 进程的管道通信

2.2.1 实验目的

编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，

掌握管道通信的同步和互斥机制。

2.2.2 实验内容

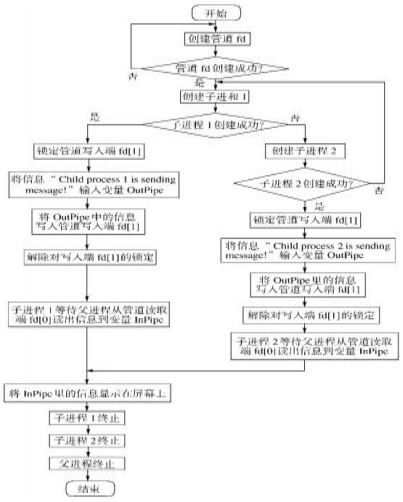
（1）学习 man 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读参考资料。
（2）根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。

2.2.3 实验思想

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，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，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)进程把一定数量的数据写入 pipe，便去睡眠等待，直到读进程取走数据后，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。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。

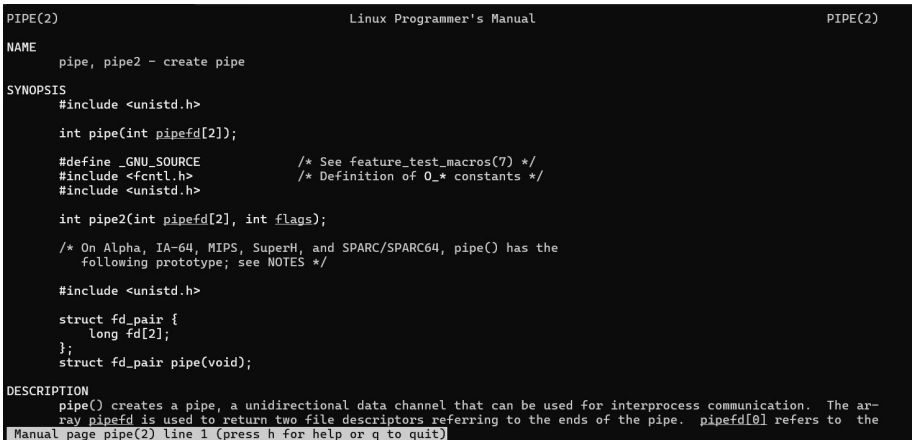
匿名管道只能用于父子进程之间的通信，而命名管道可以用于任何管道之间的通信。命名管道实际上就是一个 FIFO 文件，具有普通文件的所有性质，用 ls 命令也可以列表。但是，它只是一块内存缓冲区。



2.2.4 实验步骤

步骤一：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。

输入 man pipe 命令并执行，得到 pipe 的相关使用手册。



阅读手册，发现管道是一种用于进程间通信的机制。它允许一个进程的输出作为另一个进程的输入，从而实现数据的传输。它是由内核提供的一种缓冲区，通常用于实现流式数据传输，分为读端和写端。数据可以从写端写入，然后通过读端读取。

在系统中可以使用 pipe()系统调用来创建管道。该调用的语法为 int pipe(int fd[2]);。其中 fd[0]是读端的文件描述符。fd[1]是写端的文件描述符。

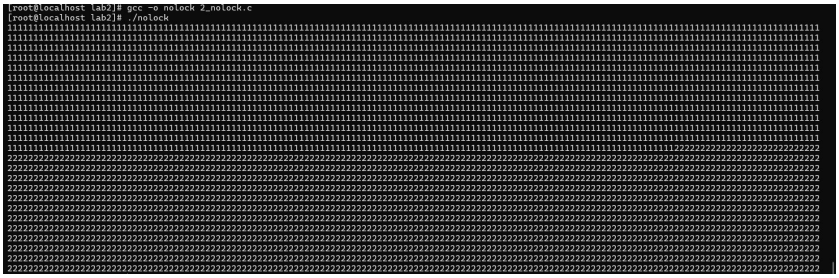
要向管道中写入数据，可以使用 write()函数，如 ssize_t write(int fd, const void *buf, size_t count);，其中 fd 是管道的写端文件描述符；buf 是指向要写入数据的缓冲区的指针；count 是要写入的字节数。相应的，如果需要从管道中读取数据，可以使用 read()函数。如 ssize_t read(int fd, void *buf, size_t count);，其中各参数定义同写操作。

步骤二：按照注释补充完整代码，运行程序，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。

首先处理不加锁情况。int pid1, pid2 定义两个变量用于存储子进程的进程 ID。主函数进行管道创建 fd[0]用于读，fd[1]用于写。创建管道后，数据可以在进程之间传递。后创建子进程，使用 fork()创建进程，如果创建成功，pid1 为 0，此时进入子进程 1 的代码块。子进程 1 向管道写入 2000 次字符 1。同样地，创建进程 2，写入 2000 次字符 2。父进程等待两个子进程结束，然后关闭写端。然后从管道中读取 4000 个字符，并在字符串末尾加上结束符。

由于父进程首先创建两个子进程。每个子进程向管道写入 2000 个字符。没有加锁机制，两个进程可能会交替写入，导致输出的字符顺序是无规律的。

但是在运行后得到如下结果。



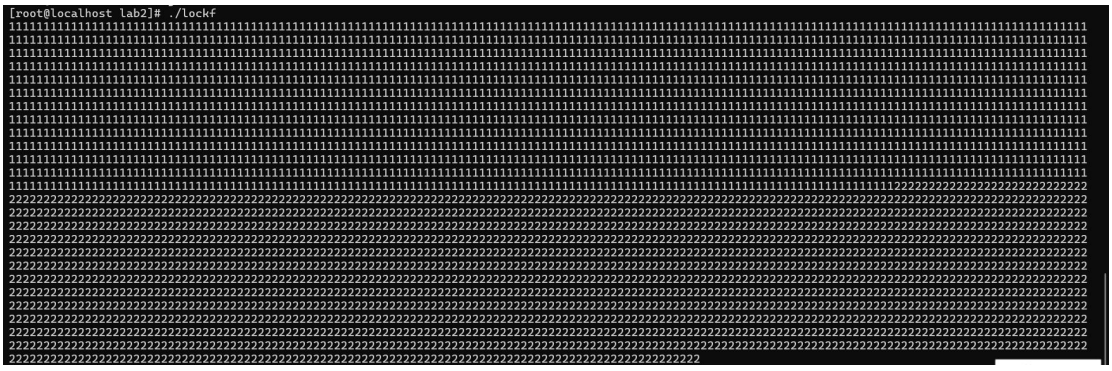
可以看到字符 1 和字符 2 并没有按照预想中的无规则交替输出。猜测可能子进程 1 在子进程 2 开始之前完成了所有的写入操作，就会导致输出中先有大量的'1'，然后才是 2。或是由于进程的写入速率不同。如果子进程 1 的 write 操作执行得更快，在子进程 2 之前完成写入。即使在没有互斥锁的情况下，输出的顺序仍然受进程调度、写入速率和管道的缓冲等因素的影响。

为了能更好观察到实验结果，选择在每个 write 操作后增加 usleep(100)，这会让当前进程暂停 100 微秒，从而给另一个子进程机会执行写入操作。这种短暂的休眠可以增加两个进程交替写入的可能性。

如图所示，由于首先读写不冲突，父进程使用 waitpid()等待两个子进程的写入，因此，读取数据过程时完整的。输出 1 和 2 的交替，说明，在一个子进程写入一个字符后，由于没有互斥锁的存在，另一个进程可以直接写入，因此不是全 1 和全 2 阵列先后出现。

再来处理加锁情况。

修改源代码，在每个子进程中调用 lockf(fd[1], 1, 0)；锁定写端，然后在写入完成后调用 lockf(fd[1], 0, 0)；解除锁定，以确保先将所有的 1 写入管道，随后再将所有的 2 写入管道。通过对管道上锁，当另一进程试图访问时，会阻塞在该处，类似于 wait()操作，当互斥锁被释放后进行访问。最终运行后得到以下结果。



发现按照预期，两个子进程以先后次序，分别写入 2000 个 1 以及 2000 个 2，而没有在时间上交叉。验证了互斥锁的功能。

或也可以采用信号量的 PV 操作，实现对有限管道的先后写读访问。

使用信号量 sem1 来完成进程 1 的互斥控制。sem2 用于控制进程 2 的执行，确保它在进程 1 完全写入 1 后才开始写入 2。进入临界区写入字符 1，然后释放 sem2 信号量以允许进程 2 进入。进程 2 等待 sem2 信号量，确保它在进程 1 完全写入后才开始写入字符 2，并释放 sem1 信号量以允许进程 1 继续。实现方法与 LAB1 内容相似。

以下是有锁和无锁程序的运行结果比较以及分析。

在无锁程序中，两个进程同时尝试向管道写入数据。由于没有任何机制来控制写入的顺序，最终的输出可能是交错的。例如：

1122112211221122112211221122112211221122112211221122112211221122...

其中字符 1 和 2 是混合在一起的，形成了交替的模式。这种情况通常发生在进程的调度中，两个进程几乎同时执行，没有保证哪个进程先写入数据。

有锁程序的运行结果中，通过 lockf 对管道进行锁定，确保同一时间只有一个进程可以向管道写入数据。这样输出的结果将是：

11111111111111111111...22222222222222222222...

在这种情况下，所有的 1 都在 2 之前被写入。这种输出结果保证了数据的顺序性。

管道通信本质上是异步的，进程之间可以独立运行。为了实现数据的同步，通常需要一些机制，如锁、信号量来控制数据的写入和读取顺序。在有锁的实现中，进程在写入数据之前获取锁，确保在写入完成之前不会有其他进程写入数据。

互斥是确保同一时间内只有一个进程访问某个资源。使用 lockf 函数可以实现互斥，避免了在写入管道时发生数据冲突。只有一个进程可以持有锁，其他进程必须等待，直到锁被释放。

步骤三：相关问题回答。

(1)你最初认为运行结果会怎么样？

最初我认为，在没有实现互斥的情况下应该输出无规则的字符 1 和字符 2，并且还可能发生同时写入操作的冲突导致发生字符缺失等问题。在实现互斥后，1 和 2 是有先后顺序的。

(2)实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

在不处理同步和互斥时，竟然出现了规则的"先 1 后 2".猜测是由于进程调度算法决定了进程执行的顺序导致子进程 1 在子进程 2 开始之前完成了所有的写入操作或写入速率不同导致的。在每个输出后增加缓冲时间后得到了预想中的结果。

验证子进程写入一个字符后，由于没有互斥锁的存在，另一个进程可以直接写入，因此出现无规律的 1、2 交替输出。

添加互斥锁后：运行结果则为规则的"先 1 后 2"，两个子进程以先后次序，写入 2000 个 1 以及 2000 个 2，而没有在时间上交叉。验证了互斥锁对于进程的控制作用。

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

在实验中，管道通信的同步与互斥主要通过操作系统提供的系统调用和文件锁定机制来实现。

同步机制用 pipe()、read()和 write()等系统调用，在写入和读取之间建立一种阻塞行为。具体来说，当一个进程写入数据时，如果管道的缓冲区已满，写操作会被阻塞，直到另一个进程读取数据，腾出空间。这种特性确保了进程按照一定的顺序进行读写操作，从而保证数据的正确性。

互斥机制使用 lockf()文件锁定机制来实现互斥。这些锁定机制可以防止多个进程同时进行写操作，从而避免竞争条件的发生。通过加锁，确保在同一时刻只有一个进程能够向管道写入数据，避免了多个进程同时写入时可能导致的数据覆盖或丢失问题。这种互斥控制有助于维护数据的一致性和完整性。

如果不控制同步与互斥可能发生数据竞争，如多个进程同时尝试写入管道，可能导致数据的交错写入，即实验中出现的字符 1 和 2 混合在一起，输出变得不可预测。同时，如果信号量或互斥锁的获取顺序不当，可能导致进程相互等待，最终导致死锁，使得程序无法继续执行。

在没有适当控制的情况下，一个进程可能长时间阻塞在写入操作上，而另一个进程未能及时读取，导致系统资源被浪费，甚至某些进程无法执行。以及，如果写入和读取没有得到同步控制，读取的数据可能是不完整或错误的，导致后续操作失败或产生错误的结果。不正确的资源管理也可能导致文件描述符或信号量未被释放，导致系统资源的泄露和潜在的崩溃。

2.2.5 运行结果分析

见 2.2.4 部分进行分析。

2.2.6 实验总结

2.2.6.1 实验中的问题与解决过程

由于操作系统调度的不可预测性，进程执行顺序可能会影响输出结果，尤其在没有互斥机制时，导致在无锁状态下并没有按照预期输出不规则的字符 1 和字符 2。经过搜索网络资料，了解到可以通过增加 sleep 时间来让进程有更多机会交替执行，或者使用信号量或条件变量来控制写入顺序。

以及父进程可能在子进程未完全写入数据时就开始读取，导致输出不完整。因此可以在父进程中使用 wait 确保所有子进程完成后再进行读取，或者使用信号机制通知父进程进行读取。

2.2.6.2 实验收获

通过这次实验，我深刻认识到互斥锁在多进程同步中的重要性。互斥锁能够确保同一时间只有一个进程访问共享资源，从而避免竞态条件和数据冲突。同时也让我我理解了如何通过管道进行进程间的通信，并结合互斥锁来确保管道的同步与互斥。通过使用管道进行进程间通信，我学到了管道的基本原理和使用方法。

这次实验使我对进程间的通信和同步有了更深入的理解。互斥锁和管道的结合为多进程编程提供了强有力的支持，使我我重视同步与互斥的设计，确保程序的正确性和高效性。

2.3 内存的分配与回收

2.3.1 实验目的

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

2.3.2 实验内容

（1）理解内存分配 FF，BF，WF 策略及实现的思路。
（2）参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。每种算法要实现内存分配、回收、空闲块排序以及合并、紧缩等功能。
（3）充分模拟三种算法的实现过程，并通过对比，分析三种算法的优劣。

2.3.3 实验思想

（1）首次适应算法。在采用空闲分区链作为数据结构时，该算法要求空闲分区链表以地址递增的次序链接。在进行内存分配时，从链首开始顺序查找，直至找到一个能满足进程大小要求的空闲分区为止。然后，再按照进程请求内存的大小，从该分区中划出一块内存空间分配给请求进程，余下的空闲分区仍留在空闲链中。
（2）循环首次适应算法。该算法是由首次适应算法演变而形成的，在为进程分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直至找到第一个能满足要求的空闲分区，并从中划出一块与请求的大小相等的内存空间分配给进程。
（3）最佳适应算法。将空闲分区链表按分区大小由小到大排序，在链表中查找第一个满足要求的分区。
（4）最差匹配算法。将空闲分区链表按分区大小由大到小排序，在链表中找到第一个满足要求的空闲分区。

内存分区回收的任务是释放被占用的内存区域,如果被释放的内存空间与其它空闲分区在地址上相邻接，还需要进行空间合并，分区回收流程如下：
1) 释放一块连续的内存区域。
2) 如果被释放区域与其它空闲区间相邻，则合并空闲区。
3) 修改空闲分区链表。

如果被释放的内存区域（回收区）与任何其它的空闲区都不相邻，则为该回收区建立一个空闲区链表的结点，使新建结点的起始地址字段等于回收区起始地址，空闲分区大小字段等于回收区大小，根据内存分配程序使用的算法要求（按地址递增顺序或按空闲分区大小由小到大排序），把新建结点插入空闲分区链表的适当位置。

如果被释放区域与其它空闲区间相邻，需要进行空间合并，在进行空间合并时需要考虑以下三种不同的情况：
1) 仅回收区的前面有相邻的空闲分区 R1。把回收区与空闲分区 R1 合并成一个空闲分区，把空闲链表中与 R1 对应的结点的分区起始地址作为新空闲区的起始地址,将该结点的分区大小字段修改为空闲分区 R1 与回收区大小之和。
2) 仅回收区的后面有相邻的空闲分区 R2。把回收区与空闲分区 R2 合并成一个空闲分区，把空闲链表中与 R2 对应的结点的分区起始地址改为回收区起始地址,将该结点的分区大小字段修改为空闲分区 R2 与回收区大小之和。
3) 回收区的前、后都有相邻的空闲分区，分别是 R1、R2。把回收区与空闲分区 R1、R2 合并成一个空闲分区,把空闲链表中与 R1 对应的结点的分区起始地址作为合并后新空闲分区的起始地址，将该结点的分区大小字段修改为空闲分区 R1、R2 与回收区三者大小之和，删去与 R2 分区对应的空闲分区结点。当然，也可以修改分区 R2 对应的结点，而删去 R1 对应的结点。还可以为新合并的空闲分区建立一个新的结点，插入空闲分区链表，删除 R1 和 R2 对应的分区结点。

一个空闲分区被分配给进程后，剩下的空闲区域有可能很小，不可能再分配给其他的进程，这样的小空闲区域称为内存碎片。最坏情况下碎片的数量会与进程分区的数量相同。大量碎片会降低内存的利用率，因此如何减少碎片就成为分区管理的关键问题。内存中的碎片太多时，可以通过移动分区将碎片集中，形成大的空闲分区。这种方法的系统开销显然很大，而且随着进程不断运行或退出，新的碎片很快就会产生。当然，回收分区时合并分区也会消除一些碎片。

当需要给进程 P 分配内存的大小为 R 时,假设当前的空闲分区有三个:R1,R2,R3,这三个空闲分区的大小都小于 R,且 R1+R2+R3>=R,这时就需要进行内存紧缩，即，将 R1,R2,R3 进行合并，形成一个新的大的连续空闲分区，之后就可以为进程 P 分配内存了。

2.3.4 实验步骤

步骤一：理解内存分配 FF，BF，WF 策略及实现的思路。

内存分配策略主要有三种：首次适应（FF）、最佳适应（BF）和最坏适应（WF）。它们的主要目标是在动态内存管理中有效地分配内存，以满足进程的需求。下面是这三种策略的简要说明及其实现思路：

首次适应即从头开始搜索内存块，找到第一个足够大的空闲块，并将其分配给请求的进程。实现思路为遍历内存块列表，找到第一个满足请求大小的空闲块。如果找到合适的块，则将其分配给进程，并更新空闲块的大小。如果剩余空间不足以满足其他进程的请求，可以将其标记为已分配。

最佳适应是遍历所有空闲内存块，找到最小的且大于或等于请求大小的块进行分配。实现思路为遍历所有空闲内存块，记录最小的满足请求的块。如果找到合适的块，则分配给进程，并更新剩余空间。此策略可以减少内存碎片，但搜索效率较低。

最坏适应则从所有空闲内存块中选择最大的块进行分配，以确保剩余的空闲内存块尽可能大。实现思路为遍历所有空闲内存块，找到最大的块进行分配。这样做的目的是希望保留较大的空闲内存块，以便未来的请求可以满足更大的内存需求。

FF 实现简单，速度快，但可能导致较多的内存碎片。BF 有效利用内存，减少碎片，但搜索时间较长。WF 通过保留大块内存来应对未来请求，但可能导致更多小块的碎片。这三种策略各有优缺点，选择哪种策略通常取决于具体的应用场景和系统需求。

步骤二：参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。

以下是代码中最重要的算法实现部分，包括内存分配算法和内存排序算法的具体分析。

内存分配算法 allocate_mem(struct allocated_block *ab)函数负责根据当前的内存分配算法分配内存块。其中变量定义分别为，fb 当前遍历的空闲块。pre 指向当前空闲块前一个块。request_size 请求的内存大小。total_free_size 遍历空闲块时统计的总空闲大小。

遍历空闲块链表，根据当前分配算法查找合适的空闲块。如果找到的空闲块能满足请求且剩余空间大于或等于 MIN_SLICE，则分割该空闲块并返回。如果空闲块的大小正好等于请求大小，则直接分配该块，并从链表中删除该块。如果在遍历过程中没有找到合适的块，但所有空闲块的总和可以满足请求，则调用 memory_compaction()进行内存紧缩，并再次遍历空闲块链表尝试分配。最后，如果分配成功返回 1，失败返回-1。

rearrange(int algorithm)根据所选的分配算法对空闲块进行排序。它调用了相应的重排函数。

rearrange_FF()该函数没有实际实现，首选适应的策略是直接在分配时查找第一个足够大的空闲块，因此不需要对空闲块进行排序。

rearrange_BF()调用 mergeSort(free_block, MA_BF)对空闲块进行排序。使用归并排序，根据空闲块的大小从小到大进行排序，以便于快速找到最小的足够大块。

rearrange_WF()类似于最佳适应，调用 mergeSort(free_block, MA_WF)，但排序的方式是从大到小，以便于快速找到最大的空闲块。

归并排序实现使用 mergeSort(struct free_block_type *head, int criterion)实现内存块排序的主要函数，采用了归并排序算法。具体流程为如果链表为空或只有一个元素，则直接返回该链表。随后使用快慢指针法找到中点，并将链表分为两部分。对分割后的两部分分别递归调用 mergeSort 进行排序。再使用 merge 函数将两个已排序链表合并成一个。合并时根据 criterion 选择合并策略，最佳适应选择较小的空闲块，最差适应选择较大的空闲块。

其中 merge(struct free_block_type *a, struct free_block_type *b, int criterion)函数用于合并两个已排序的链表。具体策略为如果是最佳适应则选择较小的块放入合并链表中。若为最差适应则选择较大的块放入合并链表中。最后，将合并后的结果返回。

步骤三：回答相关问题。

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

首次适应算法的算法思想是从空闲块链表的头部开始查找第一个能满足分配请求的空闲块，并将请求分配到该块。具有实现简单，查找速度较快，适合频繁分配的小内存请求的优点。缺点是可能会产生较多小而零散的空闲块，导致"外碎片"增加。查找性能提高方法可以采用指针优化，每次从上次查找到的位置继续查找，减少头部重复查找的时间。

最佳适应算法的算法思想是从空闲块中找到大小刚好或者稍大的空闲块，以减少内存浪费，将空闲块排序后查找最小适配块。具有可以减少内碎片的优点。缺点是查找过程耗时，可能导致小空闲块被大量保留，从而增多"外碎片"。查找性能提高方法可以通过对空闲块进行按大小排序或使用平衡树等数据结构实现更高效的查找和排序。

最坏适应算法的算法思想为在空闲块中选择最大的空闲块进行分配，确保剩余空间仍较大，便于后续分配。优点是可以减少外碎片的可能性，在分配时产生的碎片更大且便于后续利用。而缺点则为分配效率较低，可能造成大空闲块很快被分配完，导致较小的空闲块逐渐增多。可以对空闲块链表按大小降序排序，或选择合适的数据结构，如最大堆，以更快地找到最大空闲块来进行查找性能提高。

（2）3 种算法的空闲块排序分别是如何实现的。

FF 不需要排序，直接从链表头部开始查找，找到满足要求的第一个块即可。

BF 需要对空闲块按大小升序排序，每次查找时从最小块开始，以最小块满足分配要求。

WF 需要对空闲块按大小降序排序，每次查找时优先从最大块开始寻找。

（3）结合实验，举例说明什么是内碎片、外碎片，紧缩功能解决的是什么碎片。

分配内存时，空闲块大于请求的内存块大小，剩余的小碎片无法被利用。比如一个 20KB 空闲块分配给了需要 17KB 的请求，剩下 3KB 成为内碎片。外碎片则由于内存中零散的小空闲块无法整合成可用的大块，导致内存不足。比如分配了多个不相邻的小块，虽然总空闲内存满足请求，但无法分配一个大的连续空间。紧缩功能是通过将空闲块集中到一起，减少或消除外碎片，将零散的空闲块合并成一个大块，以便后续分配较大的内存请求。

（4）在回收内存时，空闲块合并是如何实现的？

在内存释放时，系统会尝试合并相邻的空闲块，合并的过程一般先将释放的内存块插入到空闲块链表中，按地址排序，以保证相邻块可以合并。再遍历链表，找到相邻的空闲块，即一个块的结束地址等于下一个块的起始地址。若存在相邻块，则将其合并，更新起始地址和大小信息，删除原来的两个块，保留合并后的一个块。

2.3.5 测试数据设计

为了能对上述三种算法进行充分模拟验证，设置在总计 1024 的内存空间中先后进入 4 个进程，分别占据 300,100,200,100。接着，进程 1，3 结束， P1 所占的 300 单位和 P3 所占的 200 单位被释放，产生两个空闲区，分别为 300 和 200 单位。 新进程 **Px** 尺寸为 150 单位，需要依据不同分配策略进行分配。按照不同算法，会得到不同结果。

首次适应算法策略下 Px 将从第一个足够大的空闲区开始分配。内存从头到尾查找，发现第一个空闲区为 300 单位，符合要求。因此，Px 会分配到起始地址为 0 的 300 单位空闲区内，将占据其前 150 单位，剩余 150 单位的空闲区。

最佳适应算法策略下，Px 将分配到最接近其大小的空闲区。现有两个空闲区，大小分别为 300 和 200 单位。200 单位是最接近 150 的空闲区，因此 Px 将分配到该空闲区。分配完成后，剩余空闲区为 50 单位。

最差适应算法策略下，Px 将分配到最大的空闲区。现有的空闲区中，所剩余的 324 单位是最大的。Px 将分配到起始地址为 700 的 1024 单位空间，使用前 150 单位，剩余 174 单位空闲。下面进行实验对比预期结果。

2.3.6 程序运行初值及运行结果分析

根据 2.3.5 的数据设计，将各个进程装填入内存如下。设定总计 1024 单位的内存空间，将先后进入 4 个进程，分别占据 300、100、200、100 单位的内存，依次插入内存并对内存进行分配和管理。

```
-------------------------------------------------------
Free Memory:
          start_addr                size
                 700                 324

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04        600        100
         3           PROCESS-03        400        200
         2           PROCESS-02        300        100
         1           PROCESS-01          0        300
-------------------------------------------------------
```

结束进程 P1 和 P3，释放出相应的空间。

```
-------------------------------------------------------
Free Memory:
          start_addr                size
                   0                 300
                 700                 324
                 400                 200

Used Memory:
       PID          ProcessName start_addr       size
         4           PROCESS-04        600        100
         2           PROCESS-02        300        100
-------------------------------------------------------
```

首先按照首次适应算法策略进行内存装填。

```
-------------------------------------------------------
Free Memory:
          start_addr                size
                 150                 150
                 700                 324
                 400                 200

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05          0        150
         4           PROCESS-04        600        100
         2           PROCESS-02        300        100
-------------------------------------------------------
```

观察到按照预期，内存从头到尾查找，发现第一个空闲区为 300 单位，符合要求，则分配到起始地址为 0 的 300 单位空闲区内，将占据其前 150 单位。

下面对最佳适应算法策略进行实验，Px 将分配到最接近其大小的空闲区。其中 200 单位是最接近 150 的空闲区，因此 Px 将分配到该空闲区。

```
------------------------------------------------------------
Free Memory:
          start_addr                  size
                 550                    50
                   0                   300
                 700                   324

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        400        150
         4           PROCESS-04        600        100
         2           PROCESS-02        300        100
------------------------------------------------------------
```

观察到符合预期。

而最差适应算法策略下，Px 将分配到最大的空闲区。因此，Px 将分配到起始地址为 700 的 324 单位空间，使用前 150 单位，剩余 174 单位空闲。

```
------------------------------------------------------
Free Memory:
          start_addr                size
                   0                 300
                 400                 200
                 850                 174

Used Memory:
       PID          ProcessName start_addr       size
         5           PROCESS-05        700        150
         4           PROCESS-04        600        100
         2           PROCESS-02        300        100
------------------------------------------------------
```

得到结果符合预期。为了对内存空间的合并进行进一步验证，再次终止进程与新进程创建。首先将进程 5 终止，释放出 150 单位的内存。此时如果不进行合并，空闲内存分布如下。

```
------------------------------------------------------------
Free Memory:
          start_addr                  size
                 520                   504

Used Memory:
       PID          ProcessName start_addr       size
         6           PROCESS-06        200        320
         4           PROCESS-04          0        100
         2           PROCESS-02        100        100
------------------------------------------------------------
```

其中 0-300 是被释放的部分，400-600 用于存储 PROCESS-04，而 700-850 则为释放后的 150 单位空间，850-1024 中则为 174 单位空闲。当请求分配 PROCESS-06 需要 320 单位时，由于没有足够的连续空闲块，必须进行合并否则将导致分配失败。

创建内存大小为 320 的进程 6 后可以看到，对空闲块进行了合并，使得进程 6 能够分配到连续的 200-520 这段空间中，成功验证合并功能。

```
------------------------------------------------------------
Free Memory:
          start_addr                  size
                 520                   504

Used Memory:
       PID          ProcessName start_addr       size
         6           PROCESS-06        200        320
         4           PROCESS-04          0        100
         2           PROCESS-02        100        100
------------------------------------------------------------
```

以上过程显示了动态内存管理中分配和释放的重要性，特别是合并空闲块的逻辑。如果在释放内存后能够及时合并相邻的空闲块，系统能够更有效地利用内存，从而降低内存碎片

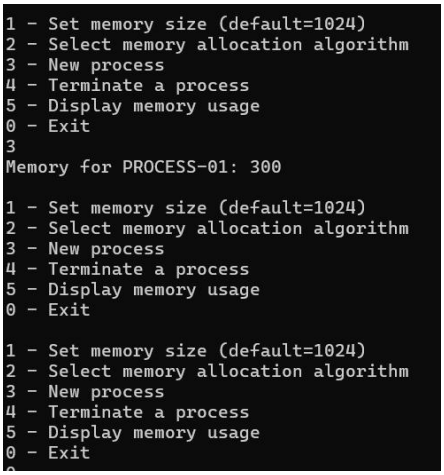
化，避免后续进程无法申请到所需内存的情况。

2.3.7 实验总结

2.3.7.1 实验中的问题与解决过程

本部分主要问题是对于参考代码的理解，由于本部分代码量较大且结合三种不同的算法，因此需要掌握理论基础和实践能力才能完成。阅读课本和相关文段后仔细理解参考代码，最终实现了三种内存分配算法，并通过模拟场景演示了它们的不同行为。成功处理了新建进程、结束进程、释放内存等操作，以及使用互斥锁和紧缩技术的内存管理问题。

以及在实验初期，发现运行代码会出现如下问题，即在进行创建新进程后输出两边菜单。



在当前的代码中，反复显示菜单的原因是 getchar()从输入缓冲区中读取字符时会包含多余的换行符。因此，可以在读取输入时忽略掉多余的换行符，以防止菜单被重复打印。在 main 函数的 choice 输入部分修改 getchar()使用 scanf(" %c", &choice);，来忽略前导空白符，这样每次选择输入后菜单将只出现一次。此外，为了使 fflush(stdin);对于不同平台的兼容性更强，可以将其移除，并使用 scanf(" %c", &choice);。

这样操作之后这样可以确保每次 display_menu 只显示一次，同时 scanf(" %c", &choice); 将有效处理多余的换行符，从而避免菜单重复显示。

2.3.7.2 实验收获

本次实验主要是锻炼了我对于课内所学的内存管理算法的掌握能力，通过实现三种内存分配算法来让我对于其原理和优劣性都有了深层次的了解，并通过模拟场景演示了它们的不同行为，深入理解了它们在不同情况下的适用性。为我后续的学习打下坚实基础。

2.4 意见与建议

本次实验提供了对管道通信和进程同步的基础理解，但在更复杂的场景中，可能会涉及到更多进程间的竞争和资源控制。因此，建议进行进一步的实验探索，如利用 lockf 进行文件锁管理、结合 wait 实现多进程的同步、深度学习 pipe 管道机制的原理，探索 fcntl 函数的应用，这些系统调用在实现多进程通信和同步控制中都十分重要。深入理解它们的底层工作机制，不仅有助于更稳定的系统编程实现，也能够提升系统编程的整体掌握。

2.5 附件

程序文件：

第一部分：  2_kill.c  2_alarm.c

第二部分：  2_nolock.c  2_nlsleep.c  2_lockf.c

第三部分：  2_alg.c

Readme 文件：  OS_LAB2_README.pdf

实验三 类 EXT2 文件系统的设计

3.1 实验目的

（1）深入理解文件系统的内部实现原理，掌握文件系统中目录结构、文件存储、索引机制等关键模块的实现细节。理解文件系统如何在硬盘上组织和管理数据，包括文件的分配、访问和回收机制。
（2）掌握文件系统的主要数据结构，理解系统中的数据组织方式，为文件操作提供高效的支持。
（3）了解 EXT2 文件系统的结构和设计思想，通过对 EXT2 文件系统的分析，理解其设计原则和关键实现细节。探索文件系统的改进方向，为设计高效、可靠的文件系统打下基础。提升综合设计与实现能力。

3.2 实验内容

（1）分析 EXT2 文件系统的核心结构，理解文件分配和管理的基本原理。
（2）基于 EXT2 文件系统的思想，设计一个类 Ext2 的多级文件系统，实现 Ext2 文件系统的一个功能子集。

（3） 文件和目录操作命令的实现，实现 login 和 password 命令，支持多用户登录和密码验证机制。实现文件的创建 create、删除 delete、读 read 和写 write。实现目录的切换 cd、列出当前目录内容 ls、显示当前路径 pwd。为文件和目录设置读写权限，确保源文件的读写保护功能有效，以及实现文件关闭 close，确保文件操作后的资源释放。
（4）在列目录时，展示以下文件或目录的详细信息：文件类型、文件名、创建时间、最后访问时间和修改时间。且源文件可以进行读写保护。

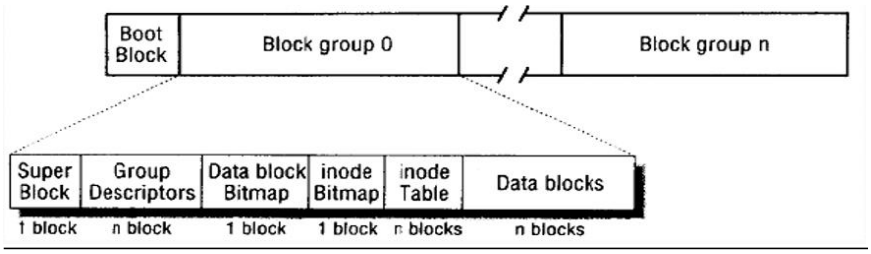
3.3 实验思想

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 文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。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。本实验只涉及后三个层次的内容。

3.4 实验步骤

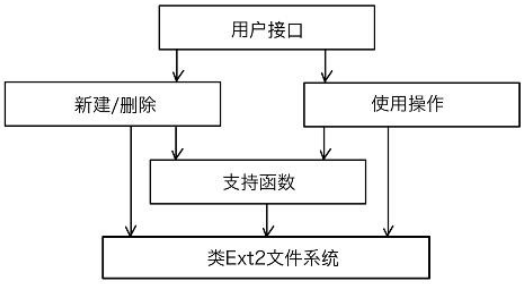
步骤一：进行宏观结构图设计。

Ext2 的磁盘布局使用了组块的结构，并且采用了多级索引使得单文件的大小得到了很大的提升.



在实现基于文件空间的类 Ext2 文件系统时，需要注意使用单文件空间来模拟整个磁盘空间、组织 ext2 系统所提供的各种功能以及其数据结构以及为用户提供 shell 和操作接口。

在满足上述各项要求的情况下，设计宏观结构图如下图所示。



为了后续的代码设计和使用方便，在初期进行宏定义与全局变量的设置，后续的设计过程中将使用这些变量。

宏定义总块数为 4611，即 1 超级块 + 1 块位图 + 1 索引节点位图 + 512 索引节点表 + 4096 数据块。这种分配方式符合简单文件系统的组织结构，保留了适当数量的块用于元数据管理和文件数据存储。确定虚拟磁盘中所有数据块的总数，以便对块号进行统一管理和分配。每块大小为 512 字节，这个值是文件系统常用的块大小，如 FAT 和 EXT 文件系统，便于磁盘分区和数据对齐。每个索引节点占用 64 字节，足够存储文件的元数据信息。数据块从第 515 块开始，其前的块分配给了超级块 、块位图、索引节点位图和索引节点表。每个目录项长度为 32 字节，能够存储基本的文件信息。32 字节的大小适合在块大小（512 字节）中均匀分配多个目录项，提高空间利用率。文件名长度设置为 15 字符，限制文件名长度，保证目录项的固定大小，便于文件系统管理。虚拟磁盘文件路径为 vdisk，表明文件系统的存储位置是当前路径下的一个文件。

全局变量 ext2_group_desc group_desc 表示组描述符，用于描述整个文件系统的元数据。包含卷名、块位图位置、空闲块计数等信息。ext2_inode inode 表示一个索引结点的结构体，用于记录文件或目录的元信息。ext2_dir_entry dir 目录体结构体，用于描述目录项，包括索引节点号、文件名、文件类型等信息。FILE *f 是文件指针，指向虚拟磁盘文件 vdisk。unsigned int last_allco_inode 和 unsigned int last_allco_block 用于记录上次分配的索引结点和数据块号，以优化分配算法。

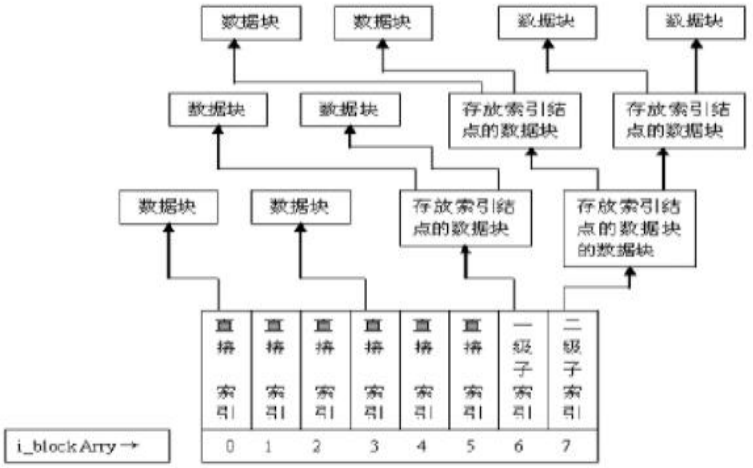
在设计过程中进行一定简化和约束，逻辑块大小、物理块大小均定义为 512 字节。由于位图只占用一个块,因此,每个组的数据块个数以及索引结点的个数均确定为 512*8=4096。进一步，每组的数据容量确定为 4096*512B=2MB。

同时在本模拟系统中，假设只有一个用户。由于只定义一个组，故组描述符只占用一个块，且 superblock 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，卷名等属于 superblock 的域。

定义 ext2_group_desc 为组描述符结构体，分别包含存储卷名、块位图的起始块号、索引结点位图的起始块号、索引结点表的起始块号、空闲块数、空闲索引结点数、文件系统的密码以及为保持数据结构的对齐填充字节。

定义 ext2_inode 为索引结点结构体，其中 int i_mode 指明文件权限和类型。高 8 位用于文件类型，低 8 位用于 rwx 权限控制。int i_blocks 是文件内容占用的块数。int i_size 为文件大小，单位为字节。时间戳字段 i_atime 表示访问时间、i_ctime 表示创建时间、i_mtime 表示修改时间、i_dtime 表示删除时间。int i_block[8]为块指针数组，6 个直接索引，1 个一级索引，1 个二级索引。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。

具体 i_block 域与文件大小以及数据块的关系图如下所示：



当文件长度小于等于 3072 字节时，只用到直接索引操作,当文件长度大于 3072 字节，并且小于等于 67KB 时，除使用直接索引处，还将使用一级子索引,当文件长度大于 67KB，并且小于 2MB 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 4096 个索引结点。故索引结点表大小为 256KB。为了与 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。 故有本系统的硬盘结构如下：

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

整个系统所需硬盘空间为 1+1+1+512+4096 个块，即 4611*512bytes=2,360,832 字节=2.35MB。最多可容纳文件数目为 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。

定义 ext2_dir_entry 目录体结构体，包括文件或目录对应的索引结点号、目录项的长度、文件名长度、文件类型，其中 1 表示普通文件，2 表示目录、文件名以及填充字节。

在文件系统初始化时，根目录的数据块将会被清空并初始化，所有包含的索引节点号和目录项长度域都会被置为 0。文件删除时，目录项的长度保持不变，但其索引节点号将被清除。当创建新文件时，程序会在目录体的数据块中查找索引节点号为 0 的目录项，并检查其长度是否足够。如果长度足够，则为文件分配该目录项；若不足够，程序将继续查找直到找到足够长度的目录项，或者找到一个未使用的目录项，并为该文件建立目录项。当创建目录时，程序将清空目录所分配的索引节点所指向的数据块，并自动插入两个特殊的目录项。一个是当前目录"."，其索引节点指向自身的数据块；另一个是上级目录".."，其索引节点指向上级目录的数据块。

步骤二：初始化模拟文件系统程序设计

为了后续各个函数设计便利，首先完成对 initialize 函数和 getch 函数的设计。

initialize 函数：

initialize 函数的目的是读取文件系统中的根目录 inode 信息，并将其存储在 cu 指向的 ext2_inode 结构体中。通过打开指定路径文件，并使用读写模式。使用 fseek 将文件指针定位到根目录的 inode 所在位置。此位置的计算可能是基于文件系统的结构。读取根目录 inode 的信息，并将其存储到 cu 指向的内存区域。

getch 函数：

函数 getch 用于从标准输入中读取一个字符，但不回显输入内容。使用 tcgetattr()获取当前终端参数，保存到变量 oldt 中，确保修改前的设置可以在函数结束时恢复。将 oldt 的副本赋值给 newt，然后修改 newt，ECHO 控制是否回显输入的字符。取消该标志会禁用输入回显。ICANON 控制是否使用规范模式。取消该标志会禁用行缓冲，从而使输入的字符在按键时立即可用，而无需按下回车键。使用 tcsetattr()将修改后的 newt 设置应用到终端，立即生效。调用 getchar()获取用户输入的字符，存储到 ch 中。读取完成后，将 oldt 重新设置为当前终端参数，恢复输入回显和规范模式。最后返回用户输入的字符。

初始化模拟文件系统的目的是创建一个大小为 2,360,832 字节,即 4611 个块的虚拟磁盘文件 vdisk，并通过此文件模拟硬盘进行文件系统的操作。文件系统操作均通过对 vdisk 文件的读写进行模拟，每次读取 512 字节，即一块，即使只修改 1 个字节，也会通过读取和写入一个完整的块来实现。

format 函数：

首先 format 函数使用 fopen(PATH, "w+")打开 vdisk 文件。如果文件不存在，则会创建该文件。循环直到成功打开文件，PATH 是文件的路径，"w+"表示以读写模式打开文件，并定义 zero 数组，用来初始化文件系统中的每个块为零。blocksiz 是一个常量，表示每个块的大小。通过循环，逐个访问磁盘的每个块，并将其内容全部初始化为 0。每次使用 fseek 定位到当前块的起始位置，然后使用 fwrite 将零数组写入磁盘。

初始化组描述符如下：

```
// 初始化组描述符
strcpy(group_desc.bg_volume_name, "Volume_name"); // 设置卷名
group_desc.bg_block_bitmap = 1;                   // 块位图所在块号
group_desc.bg_inode_bitmap = 2;                   // 索引节点位图所在块号
group_desc.bg_inode_table = 3;                    // 索引节点表起始块号
group_desc.bg_free_blocks_count = 4095;           // 可用块数（除去根目录占用块）
group_desc.bg_free_inodes_count = 4095;           // 可用索引节点数
group_desc.bg_used_dirs_count = 1;                // 已用目录数
strcpy(group_desc.psw, "123");                    // 设置默认密码
```

group_desc 是组描述符，包含了文件系统的关键元数据。这里设置了文件系统的卷名、位图的块号、索引节点表的位置、空闲块和空闲索引节点的数量等。通过 fseek()和 fwrite()将初始化后的 group_desc 写入虚拟磁盘的第一块。

随后初始化位图将 zero 数组的第一个字节设置为 0x80000000，表示位图的第一个块被标记为已使用。将位图写入第二和第三块，分别为数据块位图和索引节点位图。并初始化一个根目录的索引节点 inode，并将其写入到虚拟磁盘的第四块。设置索引节点的一些元数据，例如目录类型、文件大小、时间戳等。

为根目录创建一个目录项，表示当前目录"."，并将其写入磁盘的根目录数据块中。以及创建根目录的上一级目录".."，并将其写入数据块。

最后调用 initialize 函数将 current 指针指向根目录的索引节点。格式化过程完成后，关闭虚拟磁盘文件 fp。该函数完成了文件系统的初始化，并为后续的文件系统操作提供了基础设施。

exitdisplay 函数：

exitdisplay 函数为退出显示函数，输出感谢信息并退出。

initfs 函数：

initfs 函数的作用是初始化文件系统。如果文件系统文件不存在，它会询问用户是否创建一个新的文件系统。如果用户选择创建，则通过调用 format 函数来进行初始化。初始化完成后，它会读取文件系统的组描述符和根目录索引节点，并进行相应的初始化操作。

该部分代码首先尝试以读写模式（r+）打开指定路径（PATH）下的文件系统文件。如果文件存在，则会打开并继续执行；如果文件不存在，f 将为 NULL，则会进入文件系统创建流程。如果文件系统文件不存在，程序会提示用户是否创建一个新的文件系统。如果用户输入 Y 或 y，则调用 format(cu)函数进行初始化。如果返回值为 0 表示初始化成功，程序会重新以只读模式打开文件。如果用户选择 N 或 n，则调用 exitdisplay()退出程序并返回 1。

文件系统文件成功打开后，首先通过 fseek(f, 0, SEEK_SET)将文件指针移动到文件的起始位置，然后使用 fread 读取组描述符，即 ext2_group_desc。组描述符包含了文件系统的一些重要信息，例如块位图、inode 位图、inode 表等。接着，文件指针移动到文件的第 3 个块处，读取根目录的索引节点。最后，使用 fclose(f)关闭文件。

调用 initialize(cu)函数进行根目录初始化，将传入的指针指向根目录。cu 是 ext2_inode 类型的指针，表示当前目录的索引节点。如果文件系统成功初始化，则返回 0，表示初始化成功。

步骤三：文件系统级函数及其子函数设计

以下各个函数负责封装文件系统的底层操作，并为上层命令层提供支持。该层实现了与文件系统"硬盘"相关的所有块操作功能，包括索引节点和数据块的分配与回收、索引节点和数据块的读取与写入、位图的设置、组描述符的修改以及多级索引的实现等。

FindInode 函数：

函数 FindInode 寻找文件系统中一个空闲的索引节点并返回其编号。它通过查找索引节

点位图来确定空闲的索引节点，并且确保索引节点的分配是按顺序进行的。

函数首先尝试以读写模式打开文件（fopen(PATH, "r+")），直到成功打开为止。文件PATH是文件系统的虚拟磁盘。索引节点位图存储在文件的第二块。位图中每一个位表示一个索引节点的状态，1表示已占用，0表示空闲。从上次分配的索引节点last_allco_inode开始，检查位图中的索引节点。每次检查一个字节，即 4 位中的 32 个索引节点。如果i超过当前块的范围，会通过取余操作处理循环访问。

随后检查位图中的空闲位，如果某个字节中有空闲位，函数进一步检查该字节中的每一位。每次检查一个字节中的每一位，通过按位与和取反来检查该位是否为空闲。找到空闲位时，将该位标记为已占用，并更新索引节点位图和组描述符。

如果找到空闲索引节点，更新相关的位图和组描述符，并返回该索引节点的编号。如果没有找到空闲索引节点，函数返回-1。

FindBlock 函数：

函数 FindBlock 实现了一个寻找文件系统中空闲数据块的功能。通过读取块位图来检查哪些块是空闲的，并返回第一个找到的空闲块编号。

代码尝试以读写模式打开文件系统文件，直到成功打开为止。文件 PATH 表示文件系统虚拟磁盘。块位图存储在文件的第 1 个块，zero 是用于存储块位图的缓存数组。块位图中的每一位代表一个数据块的状态：1 表示已分配，0 表示空闲。

for (i = last_allco_block; i < (last_allco_block + blocksiz / 4); i++)从 上次分配的块编号开始搜索，以提高分配效率并尽量减少碎片化。blocksiz / 4 表示块位图按 4 字节分割的总块数，每个 unsigned int 代表 32 个块的状态。使用 i % (blocksiz / 4)保证可以循环遍历位图中的所有块。如果位图中的某个 unsigned int 不等于 0xffffffff，说明其中至少有一个块是空闲的。

通过按位操作检查 unsigned int 的每一位是否为空闲，从最高位开始，逐位向右移动检查。如果找到空闲位，则将该位设置为 1。找到空闲块后，将块位图对应位更新为已分配，将组描述符空闲块数量减 1，并更新到文件系统文件中。

计算并返回分配的块编号，1 % (blocksiz / 4)找到的 unsigned int 在位图中的索引。32 + i 计算该 unsigned int 的具体位数，得到实际块编号。如果没有空闲块，函数返回 -1。

DelInode 函数和 DelBlock 函数：

DelInode 函数和 DelBlock 函数以读写模式 ("r+") 打开文件系统文件，从文件的位图位置读取位图数据到数组中。根据提供的编号，确定对应的位图位置，后使用位操作切换计算位置的位状态，再定位到文件中位图的起始位置，将更新后的数组写回文件，从而完成 位图的更新。操作完成后，关闭文件以释放资源。

add_block 函数：

add_block 函数用于为文件或目录的 inode 增加一个新的数据块，用以存储更多内容。当现有数据块不足时，它通过直接索引、一级索引或二级索引机制来管理新的数据块。

如果 i 小于 6，直接使用 inode 的 i_block[i]存储新数据块号 j。若 i >= 6 且小于 6 + 128，使用一级索引管理数据块。一级索引块通过 i_block[6]指向，i 对应的偏移位置为 i * 4（每个索引占 4 字节）。将新块号 j 写入一级索引块的对应位置。当 i >= 134 时进入二级索引管理。二级索引通过 i_block[7]管理。当二级索引块未初始化时，先分配一个新的二级索引块并初始化它。当进入新的间接索引块时，分配一个新的间接索引块，并将其编号写入二级索引块的对应位置。若间接索引块已存在，直接读取间接索引块编号，并在其中写入新数据块号 j。

dir_entry_position 函数：

函数 dir_entry_position 根据目录项的相对起始字节偏移量 dir_entry_begin 和

i-node 的直接或间接块索引数组 i_block,计算并返回该目录项在文件系统中的绝对字节偏移地址。

如果 dir_blocks 小于等于 5，说明目录项所在的数据在直接索引块中。返回计算出的直接索引块的绝对地址。如果 dir_blocks 大于 5 且小于 5+128（一个间接索引块最多 128 个块地址），则读取一级间接索引块获取具体的物理块地址。如果 dir_blocks 超过一级间接索引范围，则需要从二级间接索引块中获取目标一级间接块地址，再从一级间接块中获取具体的数据块地址。返回计算出的目录项在文件系统中的绝对字节地址。如果 fp == NULL，使用循环等待文件打开。

FindEntry 函数：

FindEntry 函数的主要功能是为当前目录 inode 找到一个新的目录项存储位置。它根据目录项的相对字节偏移量，计算并返回该目录项在文件系统中的绝对字节地址。如果当前目录的空间已满，函数会自动分配新的数据块以容纳更多的目录项。

如果当前目录使用的块数少于 6，直接使用 i_block 数组中的直接索引块。计算绝对地址为 data_begin_block * blocksiz + 当前索引块号 * blocksiz + 块内偏移。如果当前块数在直接索引块之后，但在一级间接索引块范围内。则通过获取一级间接索引块号 i_block[6]，使用 fseek 来定位到一级间接索引块中对应的位置，读取实际的数据块号。计算绝对地址为 data_begin_block * blocksiz + 一级间接块号 * blocksiz + 块内偏移。如果当前块数超过直接索引和一级间接索引范围，使用二级间接索引。获取二级间接索引块号 i_block[7]后计算剩余块数 remain_block，定位到二级间接索引块中对应的位置，读取一级间接块号然后进一步定位到一级间接块中对应的位置，读取实际的数据块号。计算绝对地址为 data_begin_block * blocksiz + 二级间接块号 * blocksiz + 块内偏移 + dirsiz。最后更新后关闭文件系统文件，返回计算出的目录项绝对地址。

这段代码的功能是通过遍历一个 EXT2 文件系统节点的目录条目,从上一级目录中获取当前目录的名称，并将其存储在字符串"cs" 中。

login 函数：

login 函数用于验证用户输入的密码是否正确。定义了一个字符数组 psw 来存储用户输入的密码，最大长度为 15 个字符。使用 scanf 从用户输入中读取密码，并将其存储在 psw 数组中。用户输入的密码将覆盖该数组的内容。调用 strcmp 函数比较存储的密码和用户输入的密码。如果密码正确，函数返回 0，表示登录成功。否则返回非 0 值，表示登录失败。

getstring 函数：

getstring 使用 Open(&current, "..")将 current 指针指向上一层目录。在当前目录中找到目录条目".", 获取其对应的 inode 值 j。遍历上一层目录"current"，找到 inode 值等于 j 的目录条目，并将其名称拷贝到 cs 中。

步骤四：命令层函数的设计

命令层函数是一类在文件系统设计中用于实现文件操作的高层次函数,主要负责对用户提供的命令进行解释、处理并完成相应的操作。它们是文件系统接口的一部分，直接面向用户或应用程序。

Open 函数：

Open 函数在给定目录 inode 和目标文件/目录名的情况下，尝试在当前目录中找到匹配的文件或目录。遍历当前目录的所有目录项使用 dir_entry_position 函数计算每个目录项的物理位置并定位到该位置。再使用 fread 从文件中读取一个目录项的内容到 dir 结构中。通过检查 dir.name 是否与目标名称 name 匹配如果匹配，且 dir.file_type 为 2，则读取该目录的 inode 信息。使用 inode 号 dir.inode 定位到对应 inode 的物理位置并更新 current 的值。返回 0 表示成功。如果遍历完所有目录项未找到匹配项，则关闭文件并返回 1，表示

失败。

Close 函数：

Close 函数用于关闭当前目录并返回上一级目录。首先以读写模式打开文件系统映像文件 PATH。在通过 time 获取当前时间并更新 current 的 i_atime 字段，记录目录的最后访问时间。使用当前目录的第一个数据块地址 current->i_block[0]定位到当前目录项的起始位置并读取当前目录的目录项信息 bentry。使用 bentry.inode 定位到当前目录对应的 inode 的存储位置然后更新后的 current 写回文件系统。关闭文件后调用 Open 函数，尝试打开父目录。

Read 函数：

Read 函数的目的是打开指定路径的文件，并将该文件的内容输出到控制台。它通过遍历当前目录的目录项，遍历当前目录的所有目录项，使用 fseek 定位每个目录项的存储位置，然后使用 fread 读取目录项数据到 dir 结构。如果目录项的文件名与目标文件名匹配，并且文件类型是普通文件，则继续处理。根据目录项中的 inode 索引，定位到文件的 inode 信息，读取文件的元数据。遍历文件的所有数据块，根据 inode 中存储的 i_block 数据块索引，逐块读取文件内容并输出到控制台。遇到回车符时，打印换行。在读取完文件内容后，更新文件的最后访问时间，并将修改后的数据写回文件系统。读取和更新文件系统后，关闭文件指针。如果文件成功读取并显示内容，返回 0，如果找不到指定的文件，返回 1。

Write 函数：

Write 函数的主要功能是在指定的文件中写入数据。通过打开文件系统文件，以读写模式打开。如果 fopen 返回 NULL，则会持续尝试打开该文件，直到成功为止。遍历当前目录，检查是否存在指定名字的文件。若找到了对应的文件，读取其 inode 信息。如果遍历后找到了文件，则跳出循环；否则输出提示信息，并返回 0，要求用户先创建文件。通过先前设置的 getch()获取用户的每一个字符，并将其写入文件。当输入的字符数量达到当前数据块的最大值，即 512 字节时，通过 add_block 为文件分配一个新的数据块。如果用户按下 ESC，则结束输入。在用户输入结束后，更新文件的修改时间和访问时间。然后将更新后的 inode 信息写回文件，并关闭文件。

Create 函数：

Create 函数的主要功能是创建一个新文件或目录，并将其信息写入模拟的文件系统中。通过 FindInode()和 FindBlock()函数查找空闲的 inode 和数据块。遍历当前目录的所有目录项，检查是否存在同名文件或目录。如果找到了相同名字的文件或目录，返回 1，表示创建失败。根据文件类型 type，为新文件或目录分配一个 inode，设置其基本属性。对于目录，创建当前目录和父目录的目录项。随后创建一个空目录项以填充数据块，防止数据块碎片化，并将新创建的 inode 信息写入文件系统。将新创建的文件或目录的目录项写入当前目录。更新并保存当前目录的 inode 信息。如果文件或目录创建成功，返回 0，否则返回 1。

Delete 函数：

Delete 函数用于文件系统中删除指定文件或目录的操作。打开文件系统文件进行读写操作。计算当前目录下的条目总数，然后遍历每一个目录项。通过获取当前目录项的位置，并读取该目录项的内容到 centry 中。如果找到匹配的条目，则说明找到要删除的文件或目录。如果要删除的是目录，首先会递归删除该目录中的所有条目。通过调用 Delete 函数递归删除目录中的子目录和文件。删除目录时，会释放其占用的数据块和 inode。如果要删除的是文件，则删除文件所占用的数据块。删除文件或目录后，更新当前目录的内容。如果删除的条目不是最后一项，则用最后一项覆盖被删除的条目，并更新当前目录的 inode 信息。

ls 函数：

ls 函数用于列出当前目录中的文件和子目录，以及它们的详细信息。打开指定路径 文

件，并使用读写模式。打印目录内容的标题行，说明每列的内容。使用 asctime 函数将时间戳转换为字符串格式，并用 localtime 将时间转换为本地时间。替换时间字符串中的换行符\n 为制表符\t，使输出更加整齐。后判断 dir.file_type 是否为文件类型。根据文件类型打印"File"或"Directory"标签，并输出文件/目录的名称以及格式化后的时间戳。

pwd 函数：

pwd 函数会递归遍历当前目录及其上级目录，直到到达根目录，构建出从根目录到当前目录的完整路径。在每一次递归中，函数都会从当前目录的条目读取到该目录的信息，再读取其父目录的 inode，通过递归向上追溯直到根目录。每次递归时，会将上级目录的路径与当前目录名称拼接起来，逐渐构建出当前目录的绝对路径。

假设当前目录是"/home/user/docs"，那么递归的过程是：

1. 第一次递归：当前目录"docs"，上级目录"user"，路径拼接为"/home/user/docs"。
2. 第二次递归：当前目录"user"，上级目录"home"，路径拼接为"/home/user/docs"。
3. 第三次递归：当前目录"home"，上级目录"/"，路径拼接为"/home/user/docs"。

最终结果会得到从根目录到当前目录的完整路径"/home/user/docs"。

步骤五：界面及 main 函数设计

shellloop 函数：

模拟的 Shell 环境，用于管理一个基于 ext2 文件系统的操作。在 shellloop 函数中，程序进入一个无限循环，等待用户输入命令，并根据不同命令执行相应的操作。

首先，ctable 数组保存了可用的命令。每次循环开始时，程序会获取当前目录的路径，并显示在命令提示符中，等待用户输入命令。用户输入的命令被存储在 command 变量中，程序通过 strcmp 比较输入命令和 ctable 中的命令，找到对应的命令。

用户输入命令后，程序检查是创建文件还是目录，然后调用 Create 或 Delete 函数来创建或删除文件/目录。如果操作成功，则输出相应提示。

cd 命令用于改变当前目录。程序根据输入的路径字符串 var2，解析路径并调用 Open 函数改变当前目录。如果路径格式错误，则输出提示信息。

用户输入要关闭的文件层级数，程序通过循环调用 Close 函数逐层关闭文件。

用户输入要读取的文件名，通过调用 Read 函数读取文件内容，如果失败则提示错误。

用户输入文件名，通过 Write 函数向文件写入数据。

用户输入 password 命令，程序调用 Password 函数更改密码。

用户确认是否格式化文件系统，程序调用 format 函数执行文件系统格式化操作。

用户确认是否退出文件系统，若确认则退出。

login 命令表示用户尝试登录，若用户已经登录，则输出提示信息。logout 命令则注销当前用户，重新进入登录界面。ls 命令列出当前目录中的文件和目录项。

pwd 命令显示当前目录的绝对路径，通过递归调用 pwd 函数，逐层获取父目录路径并拼接成完整的绝对路径。help 命令输出可用命令的帮助信息，列出所有支持的命令及其使用方法。

main 函数：

简化版的文件系统程序的主函数。调用 initfs(&cu)函数进行文件系统初始化。如果初始化失败，则程序退出。通过调用 login()函数要求用户进行登录操作。假设该函数会处理密码验证，并返回 0 表示登录成功。如果 login()返回非 0 值，表示密码验证失败后调用 exitdisplay()函数显示退出信息，最后程序结束。

如果登录成功，程序进入文件系统命令的主循环，通过调用 shellloop(cu)进入交互式命令行，用户可以执行如创建文件、改变目录、查看文件等操作。shellloop 函数会一直循环直到用户退出文件系统。程序在退出之前调用 exitdisplay()函数。最后 return 0 表示

程序正常结束。

步骤六：相关问题回答

（1）如何实现将文件作为硬盘来使用？

要将文件作为硬盘使用，通常是通过在操作系统中模拟文件系统的方式来实现。可以通过将文件作为原始数据存储区，可以模拟硬盘的操作。在代码中，使用 fopen() 打开文件并进行读写操作来访问该文件。或者可以实现类似 ext2_inode 和 ext2_group_desc 这样的结构体来表示文件系统中的 inode、超级块、目录项等。通过这些结构体的操作，模拟硬盘上的数据存取与管理。

（2）对于删除文件后的空闲区应该怎样处理？

在文件系统中，删除文件后应当将文件占用的空间标记为“空闲”以便后续可以复用。

当文件被删除时，其占用的 inode 和数据块应该被释放。可以将 inode 标记为“未使用”，并将其对应的数据块也标记为“空闲”。在代码中，需要修改 ext2_inode 和 ext2_group_desc 中的相应字段来反映这些改变。当文件被删除后，相关的块被标记为空闲并合并到空闲链表中，供以后分配使用。这样可以避免空闲块过多而导致碎片化。

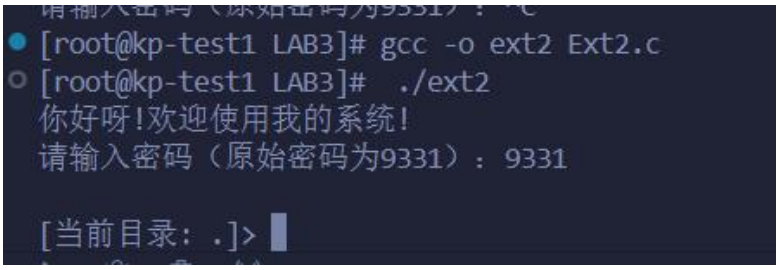
（3） 多级索引如何实现？

多级索引的实现通常是为了支持大文件，避免在 inode 中直接存储过多的指向数据块的指针。inode 中有一定数量的直接指针，直接指向文件的数据块。如果文件的数据块数量超过直接块的数量，使用一级间接指针。一级间接块是一个指针数组，每个指针指向一个数据块。一级间接块指针存储在 inode 中的一个单独的块中。如果文件依然很大，一级间接块指向的块也可能无法完全存储所有的数据块。这时，使用二级间接块指针。二级间接块指向一级间接块，从而间接地指向数据块。

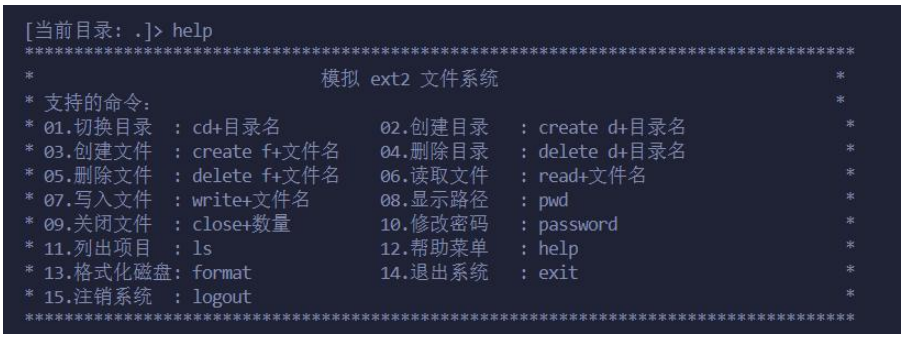
在文件写入或读取时，代码会根据数据块的位置和大小选择使用直接块、一级间接块、二级间接块或三级间接块进行访问。

3.5 程序运行初值及运行结果分析

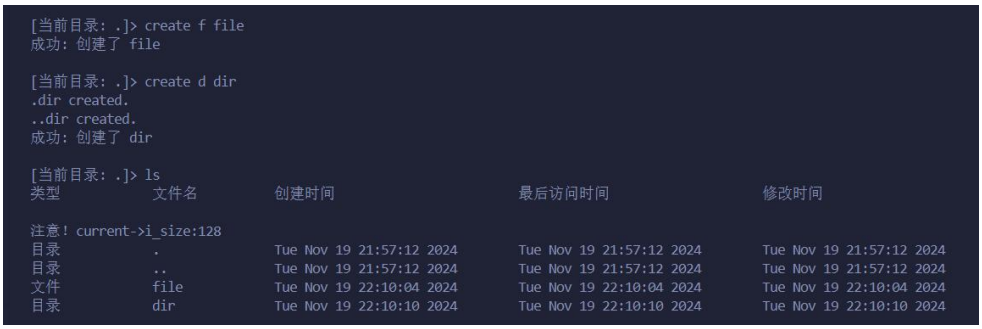
对各个功能进行测试，首先登录系统。



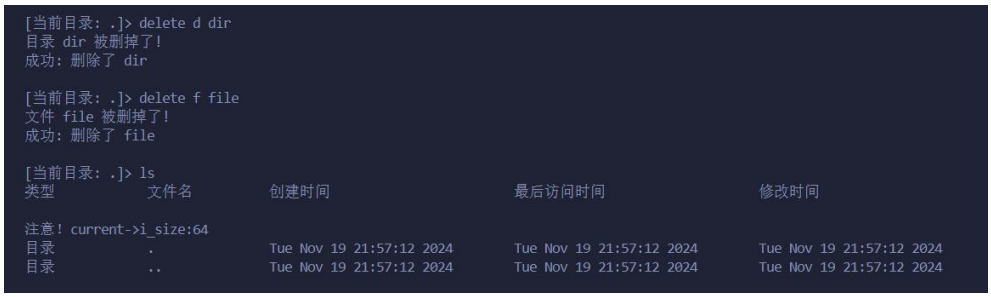
列出 help 目录如下。



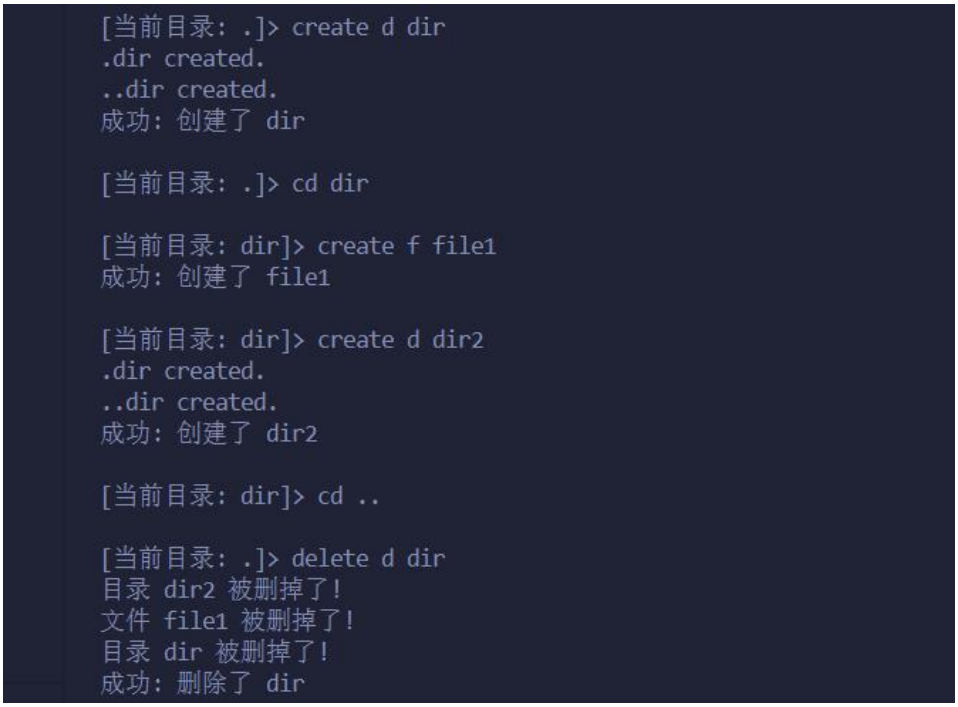
创建文件和目录如下。



删除文件和目录如下：

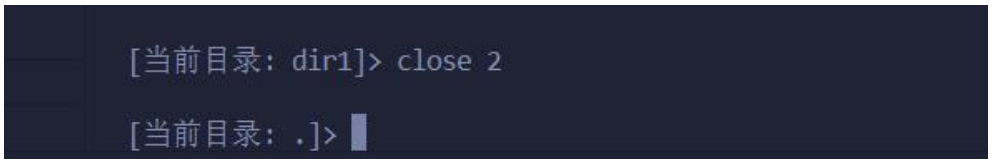


进一步验证功能，创建一个目录 dir，然后在目录 dir 里边创建新的目录和文件，最后在上级目录中直接删除目录 dir。具体如下：

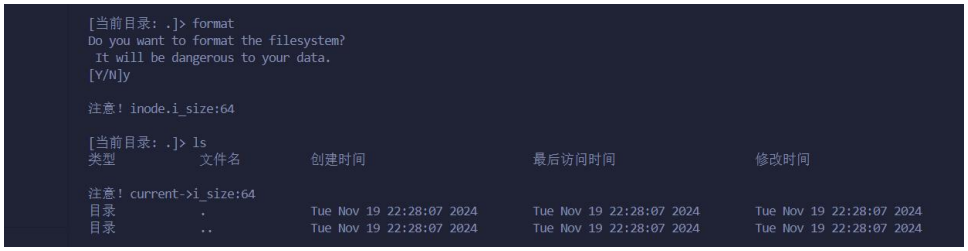


同上步，建立目录并使用 pwd 命令寻找当前目录的绝对路径，操作如下：

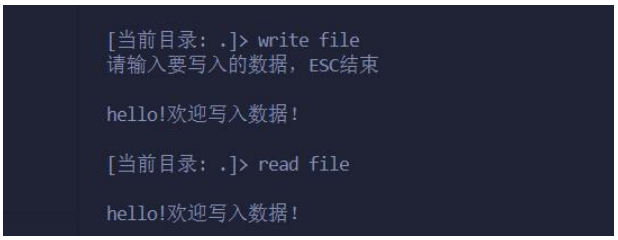
close 关闭命令测试，在上一步基础上直接调用 close 命令关闭两级目录。操作如下：



测试 format 格式化命令，将上面各个步骤创建的内容全部一次性清空。操作如下：

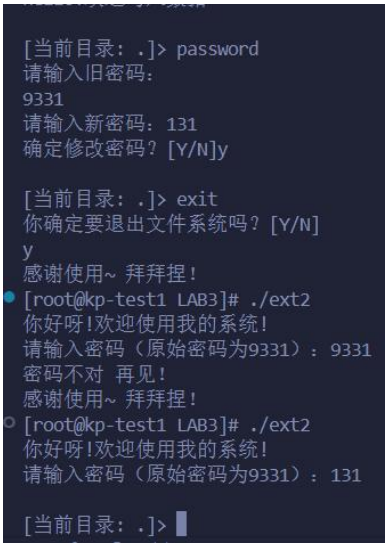


下面测试 write 写命令和 read 读命令，在创建的 file 文件中写入内容后读出。



。

使用 password 命令更改密码，后使用 exit 命令退出后用新密码登录。操作如下：



3.6 实验总结

3.6.1 实验中的问题与解决过程

（1）在初步了解文件系统的概念后，对于如何系统地实现文件系统缺乏全面的认识。在实验过程中，我常常担心会遗漏某些步骤，进而引发一系列问题。这使得我在进行文件系统设计和实现时心里不安。通过翻阅了多篇关于 EXT2 文件系统的实现文章，借此获取更多的背景知识。

（2）在实际编写文件系统时，我发现自己对文件创建、读写操作以及获取当前时间等基础功能不够熟悉，特别是在寻址方式上，容易混淆绝对寻址、直接寻址、间接寻址等不同概念。通过资料查阅与阅读相关实验指导文档，帮助我理清了问题，增强了对文件系统实现细节的掌握。

（3）在文件系统中，读取目录项需要通过目录文件的 inode 数据块部分，按照一定的流程解析目录项。然而，当目录项变多时，读取的复杂性可能增加，特别是涉及长目录项或嵌套目录的场景。通过按最大长度的方式读取目录项，对目录文件的结构进行统一约束以简化读取流程。

（4）当分配的数据块数量超过系统预期时，可能导致溢出问题，如空闲块不足、超出文件系统的限制等。预先判断空闲块数量，即在分配数据块前，检查当前文件系统中的空闲块数是否足够满足所需块数。如果不足，则直接拒绝分配，并返回错误提示。对于特殊场景，如大文件分配，引入动态分配机制，例如在多级索引中扩展一级或二级间接块。

（5）当目录项的长度可变时，存储和解析变得复杂，直接存储内容可能导致占用不必要的空间，或者需要频繁的调整和迁移。通过使用指针而非直接存储内容，在目录项中存储指针如 inode 指针或文件名指针以指向实际内容的存储位置，而不是直接存入文件名等变量长度内容。这种方式能够显著提高目录项的存储效率。

3.6.2 实验收获

通过这次实验，我深入理解了文件系统的基本构成元素，并认识到这些元素如何通过协同工作形成文件系统的核心架构。且通过实际操作，我加深了对文件系统内部机制的理解。尤其在实现过程中，我逐渐掌握了文件操作函数的使用，这些函数的灵活运用对文件系统的实现至关重要。在实验过程中，我深入学习了如何进行磁盘块的分配和管理。学会了如何管理磁盘空间并保证数据的持久性。

通过对 inode 的深入学习，我理解了它在文件和目录管理中的重要性。我不仅学习了如何设计并维护目录结构，还加深了对目录结构管理的理解，并学会了如何高效地组织文件和目录。这次实验使我收获满满，对文件系统有了更加深入全面的理解，巩固了理论知识。

3.7 意见与建议

（1）建议在教学过程中提供更多类似的实践机会，尤其是与文件系统、内核编程和系统级开发相关的实验。通过实践，能够更加深入地理解概念、掌握技能，同时也能够发现和解决问题，提升解决实际问题的能力。

（2）鼓励积极利用现有的文档和社区资源，尤其是开源社区中的 Linux 内核源代码、C 标准库文档、文件系统实现文献以及技术论坛。这些资源提供了详细的技术说明、最佳实践、解决方案和开发技巧，可以帮助学生更好地理解复杂的概念，并在遇到问题时找到有效的解决方法。社区的讨论和开发经验也是非常宝贵的，尤其是在学习新技术时。

（3）在进行内核级编程，特别是涉及到文件系统实现等低级操作时，强调代码的安全性和稳定性至关重要。应该加强对潜在风险的认知，如内存泄漏、越界访问、并发问题、资源释放等常见问题。

3.8 附件

程序文件： 
Ext2.c

Readme 文件： 
OS_LAB3_README.pdf

具体附件

1_called.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    printf("Welcome to this Process\n");
    pid_t pid;
    pid = getpid();
    printf("The PID of this process  is: %d\n", pid);
    return 0;
}
```

1_execl.c

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

int shared_variable = 0; // 共享变量，用于线程之间的协作
pthread_mutex_t lock; // 互斥锁，用于保护共享变量

// 线程 1 执行的函数
void *op1() {
    for (int i = 0; i < 5000; i++) { // 循环 5000 次
        pthread_mutex_lock(&lock); // 获取互斥锁，以保护共享变量的访问
        shared_variable++; // 将共享变量自增 1
        pthread_mutex_unlock(&lock); // 释放互斥锁
    }
    return NULL; // 线程结束
}

// 线程 2 执行的函数
void *op2() {
    for (int i = 0; i < 5000; i++) { // 循环 5000 次
        pthread_mutex_lock(&lock); // 获取互斥锁，以保护共享变量的访问
        shared_variable += 2; // 将共享变量自增 2
        pthread_mutex_unlock(&lock); // 释放互斥锁
```

```
    }
    return NULL; // 线程结束
}

int main() {
    pthread_t thread1, thread2; // 定义两个线程的 ID

    // 初始化互斥锁
    pthread_mutex_init(&lock, NULL);

    // 创建线程 1
    if (pthread_create(&thread1, NULL, op1, NULL) != 0) {
        perror("Failed to create thread1"); // 输出创建线程失败的信息
        exit(1); // 退出程序
    }

    // 创建线程 2
    if (pthread_create(&thread2, NULL, op2, NULL) != 0) {
        perror("Failed to create thread2"); // 输出创建线程失败的信息
        exit(1); // 退出程序
    }

    // 等待线程 1 结束
    pthread_join(thread1, NULL);
    // 等待线程 2 结束
    pthread_join(thread2, NULL);

    // 输出共享变量的最终值
    printf("Final value of shared variable: %d\n", shared_variable);

    // 销毁互斥锁，释放资源
    pthread_mutex_destroy(&lock);

    return 0; // 正常结束程序
}
```

1_pid.c

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

int global = 0; // 全局变量，初始值为 0
```

```
int main() {
    pid_t pid, pid1; // 定义进程 ID

    /* 创建子进程 */
    pid = fork(); // 调用 fork()创建一个新进程
    if (pid < 0) {
        /* 发生错误 */
        fprintf(stderr, "Fork Failed"); // 输出错误信息
        return 1; // 返回 1 表示错误
    }
    else if (pid == 0) { /* 子进程 */
        pid1 = getpid(); // 获取子进程的 ID
        printf("child: pid = %d\n", pid); /* A 输出子进程的 PID（应为 0） */
        printf("child: pid1 = %d\n", pid1); /* B 输出子进程的真实 PID */

        global += 100; // 子进程中全局变量加 100
        printf("child: global = %d\n", global); // 输出子进程中的全局变量值
        printf("child: global_address = %x\n", &global); // 输出全局变量的地址
    }
    else { /* 父进程 */
        pid1 = getpid(); // 获取父进程的 ID
        printf("parent: pid = %d\n", pid); /* C 输出子进程的 PID */
        printf("parent: pid1 = %d\n", pid1); /* D 输出父进程的真实 PID */

        global += 200; // 父进程中全局变量加 200
        printf("parent: global = %d\n", global); // 输出父进程中的全局变量值
        printf("parent: global_address = %x\n", &global); // 输出全局变量的地址

        wait(NULL); // 等待子进程结束
    }

    global += 1; // 在父子进程结束后，全局变量加 1
    printf("final: global = %d\n", global); // 输出最终的全局变量值

    return 0; // 正常结束程序
}
```

1_spinlock.c

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

```c
#include <pthread.h>
#include <stdlib.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;
    spinlock_t lock;

    // 初始化自旋锁
```

```
    spinlock_init(&lock);

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

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

    // 输出共享变量的值
    printf("Final value of shared variable: %d\n", shared_value);

    return 0;
}
```

1_system.c

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

int main() {
    pid_t pid, pid1; // 定义进程 ID

    /* 创建子进程 */
    pid = fork();
    if (pid < 0) {
        /* 发生错误 */
        fprintf(stderr, "Fork Failed"); // 输出错误信息
        return 1; // 返回 1 表示错误
    }
    else if (pid == 0) { /* 子进程 */
        // 执行外部程序"./called"
        system("./called");

        pid1 = getpid(); // 获取子进程 ID
        printf("child: pid = %d\n", pid); /* 输出子进程的 PID（应为 0） */
        printf("child: pid1 = %d\n", pid1); /* 输出子进程的 PID1 */
    }
    else { /* 父进程 */
        pid1 = getpid(); // 获取父进程 ID
```

```
        printf("parent: pid = %d\n", pid); /* 输出子进程的 PID */
        printf("parent: pid1 = %d\n", pid1); /* 输出父进程的 PID1 */

        wait(NULL); // 等待子进程结束
    }
    return 0; // 正常结束程序
}
```

1_thread_lock.c

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

int shared_variable = 0; // 共享变量
pthread_mutex_t lock; // 互斥锁

// 线程 1 执行的函数
void *op1() {
    for (int i = 0; i < 5000; i++) {
        pthread_mutex_lock(&lock); // 获取锁
        shared_variable++; // 共享变量自增 1
        pthread_mutex_unlock(&lock); // 释放锁
    }
    return NULL; // 线程结束
}

// 线程 2 执行的函数
void *op2() {
    for (int i = 0; i < 5000; i++) {
        pthread_mutex_lock(&lock); // 获取锁
        shared_variable += 2; // 共享变量自增 2
        pthread_mutex_unlock(&lock); // 释放锁
    }
    return NULL; // 线程结束
}

int main() {
    pthread_t thread1, thread2; // 线程 ID

    // 初始化互斥锁
    pthread_mutex_init(&lock, NULL);

    // 创建线程 1
```

```
    if (pthread_create(&thread1, NULL, op1, NULL) != 0) {
        perror("Failed to create thread1"); // 输出错误信息
        exit(1); // 退出程序
    }

    // 创建线程 2
    if (pthread_create(&thread2, NULL, op2, NULL) != 0) {
        perror("Failed to create thread2"); // 输出错误信息
        exit(1); // 退出程序
    }

    // 等待线程 1 结束
    pthread_join(thread1, NULL);
    // 等待线程 2 结束
    pthread_join(thread2, NULL);

    // 输出共享变量的最终值
    printf("Final value of shared variable: %d\n", shared_variable);

    // 销毁互斥锁
    pthread_mutex_destroy(&lock);

    return 0; // 正常结束程序
}
```

1_thread_pv.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h> // 用于信号量

int shared_variable = 0;
sem_t semaphore; // 定义信号量

void *op1() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore); // P 操作 - 等待
        shared_variable++;
        sem_post(&semaphore); // V 操作 - 释放
    }
    return NULL;
}
```

```
void *op2() {
    for (int i = 0; i < 5000; i++) {
        sem_wait(&semaphore); // P 操作 - 等待
        shared_variable += 2;
        sem_post(&semaphore); // V 操作 - 释放
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // 初始化信号量，初始值为 1（相当于互斥锁）
    sem_init(&semaphore, 0, 1);

    if (pthread_create(&thread1, NULL, op1, NULL) != 0) {
        perror("Failed to create thread1");
        exit(1);
    }

    if (pthread_create(&thread2, NULL, op2, NULL) != 0) {
        perror("Failed to create thread2");
        exit(1);
    }

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    printf("Final value of shared variable: %d\n", shared_variable);

    // 销毁信号量
    sem_destroy(&semaphore);

    return 0;
}
```

1_thread_sys_exe.c

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

```
// 线程执行的函数
void *op(void *params) {
    pthread_t tid = pthread_self(); // 获取当前线程 ID
    printf("Inner thread: tid = %ld\n", tid); // 输出线程 ID
    execlp("./called", NULL); // 执行外部程序"./called"
    // system("./called"); // 也可以使用 system 函数执行外部程序

    pthread_exit(0); // 结束线程
}

int main() {
    pid_t pid, pid1;

    /* 创建子进程 */
    pid = fork();
    if (pid < 0) {
        /* 发生错误 */
        fprintf(stderr, "Fork Failed"); // 输出错误信息
        return 1; // 返回 1 表示错误
    }
    else if (pid == 0) { /* 子进程 */
        pid1 = getpid(); // 获取子进程 ID
        printf("child: pid = %d\n", pid); /* 输出子进程 ID */
        printf("child: pid1 = %d\n", pid1); /* 输出子进程 ID1 */

        // 创建线程并执行
        pthread_t tid; // 线程 ID
        pthread_attr_t attr; // 线程属性

        pthread_attr_init(&attr); // 初始化线程属性

        pthread_create(&tid, &attr, op, NULL); // 创建线程
        pthread_join(tid, NULL); // 等待线程结束
    }
    else { /* 父进程 */
        pid1 = getpid(); // 获取父进程 ID
        printf("parent: pid = %d\n", pid); /* 输出父进程 ID */
        printf("parent: pid1 = %d\n", pid1); /* 输出父进程 ID1 */
        wait(NULL); // 等待子进程结束
    }

    return 0; // 正常结束程序
}
```

1_unlock.c

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

// 全局变量
int shared_variable = 0;
pthread_mutex_t lock;

// 循环次数常量
#define ITERATIONS 5000

// 线程函数 1
void *op1(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&lock); // 加锁
        shared_variable++;
        pthread_mutex_unlock(&lock); // 解锁
    }
    return NULL;
}

// 线程函数 2
void *op2(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&lock); // 加锁
        shared_variable += 2;
        pthread_mutex_unlock(&lock); // 解锁
    }
    return NULL;
}

int main() {
    pthread_t thread1, thread2;

    // 初始化互斥锁
    if (pthread_mutex_init(&lock, NULL) != 0) {
        perror("Mutex initialization failed");
        return EXIT_FAILURE;
    }

    // 创建线程 1
```

```
    if (pthread_create(&thread1, NULL, op1, NULL) != 0) {
        perror("Failed to create thread1");
        pthread_mutex_destroy(&lock); // 销毁互斥锁
        return EXIT_FAILURE;
    }

    // 创建线程 2
    if (pthread_create(&thread2, NULL, op2, NULL) != 0) {
        perror("Failed to create thread2");
        pthread_mutex_destroy(&lock); // 销毁互斥锁
        return EXIT_FAILURE;
    }

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

    // 输出共享变量的最终值
    printf("Final value of shared variable: %d\n", shared_variable);

    // 销毁互斥锁
    pthread_mutex_destroy(&lock);

    return EXIT_SUCCESS;
}
```

test_pid.c

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

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();
    if (pid < 0) {
        /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
```

```
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d\n",pid); /* A */
        printf("child: pid1 = %d\n",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d\n",pid); /* C */
        printf("parent: pid1 = %d\n",pid1); /* D */
        wait(NULL);
    }
    return 0;
}
```

2_alarm.c

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

pid_t pid1 = -1, pid2 = -1;

// 处理父进程接收到的中断信号
void handle_parent_interrupt(int sig) {
    if (pid1 <= 0 || pid2 <= 0) return; // 子进程空操作
    printf("\nReceived soft interrupt\n");
    kill(pid1, 16);
    printf("Sent 16 signal successfully\n");
    kill(pid2, 17);
    printf("Sent 17 signal successfully\n");
}

// 处理子进程 1 收到的信号
void handle_child1_signal(int sig) {
    printf("Child process 1 is killed by parent!!\n");
    exit(0); // 直接退出子进程
}

// 处理子进程 2 收到的信号
void handle_child2_signal(int sig) {
    printf("Child process 2 is killed by parent!!\n");
    exit(0); // 直接退出子进程
```

```
}

// 处理闹钟信号
void handle_alarm(int sig) {
    printf("\nAlarm signal received, terminating processes...\n");
    if (pid1 > 0) kill(pid1, 16);
    if (pid2 > 0) kill(pid2, 17);
}

int main() {
    signal(SIGINT, handle_parent_interrupt);   // 捕获 SIGINT
    signal(SIGQUIT, handle_parent_interrupt);  // 捕获 SIGQUIT
    signal(SIGALRM, handle_alarm);              // 捕获 SIGALRM

    // 创建子进程1
    while (pid1 == -1) pid1 = fork();
    if (pid1 > 0) { // 父进程
        // 创建子进程2
        while (pid2 == -1) pid2 = fork();
        if (pid2 > 0) { // 父进程
            printf("Start waiting for 5 seconds...\n");
            alarm(5); // 设置5秒后发送SIGALRM
            pause();  // 等待信号
            printf("\nParent process is killed!!\n");
        } else { // 子进程 2
            signal(17, handle_child2_signal);
            printf("Child process 2 is ready\n");
            pause(); // 等待信号
        }
    } else { // 子进程 1
        signal(16, handle_child1_signal);
        printf("Child process 1 is ready\n");
        pause(); // 等待信号
    }

    return 0;
}
```

2_alg.c

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

#define PROCESS_NAME_LEN 32
```

```
#define MIN_SLICE 10
#define DEFAULT_MEM_SIZE 1024
#define DEFAULT_MEM_START 0

#define MA_FF 1
#define MA_BF 2
#define MA_WF 3

struct free_block_type {
  int size;
  int start_addr;
  struct free_block_type *next;
};
struct free_block_type *free_block = NULL;

struct allocated_block {
  int pid;
  int size;
  int start_addr;
  char process_name[PROCESS_NAME_LEN];
  struct allocated_block *next;
};
struct allocated_block *allocated_block_head = NULL;

int allocate_mem(struct allocated_block *ab);
void kill_process();
struct allocated_block *find_process(int pid);
void free_mem(struct allocated_block *ab);
void dispose(struct allocated_block *free_ab);
void kill_block(struct allocated_block *ab);
void display_mem_usage();
void set_mem_size();
void set_algorithm();
void display_menu();
void new_process();
void rearrange(int algorithm);
void sort_free_blocks();
void compact();
void memory_compaction();
void do_exit();
int mem_size = DEFAULT_MEM_SIZE;
int ma_algorithm = MA_FF;
static int pid = 0;
int flag = 0;
```

```
struct free_block_type *init_free_block(int mem_size) {
  struct free_block_type *fb;
  fb = (struct free_block_type *)malloc(sizeof(struct free_block_type));
  if (fb == NULL) {
    printf("No mem\n");
    return NULL;
  }
  fb->size = mem_size;
  fb->start_addr = DEFAULT_MEM_START;
  fb->next = NULL;
  return fb;
}

void set_mem_size() {
  int size;
  if (flag != 0) {
    printf("Cannot set memory size again\n");
    return;
  }
  printf("set memory_size to: ");
  scanf("%d", &size);
  if (size > 0) {
    mem_size = size;
    free_block->size = mem_size;
  }
  flag = 1;
}

void set_algorithm() {
  int algorithm;
  printf("\t1 - First Fit\n");
  printf("\t2 - Best Fit \n");
  printf("\t3 - Worst Fit \n");
  scanf("%d", &algorithm);
  if (algorithm >= MA_FF && algorithm <= MA_WF) {
    ma_algorithm = algorithm;
  }
  rearrange(ma_algorithm);
}

struct free_block_type *merge(struct free_block_type *a,
                              struct free_block_type *b, int criterion) {
  struct free_block_type dummy;
```

```
  struct free_block_type *current = &dummy;
  if (criterion == MA_BF) {
    while (a && b) {
      if (a->size < b->size) {
        current->next = a;
        a = a->next;
      } else {
        current->next = b;
        b = b->next;
      }
      current = current->next;
    }
    current->next = a ? a : b;
  } else if (criterion == MA_WF) {
    while (a && b) {
      if (a->size > b->size) {
        current->next = a;
        a = a->next;
      } else {
        current->next = b;
        b = b->next;
      }
      current = current->next;
    }
    current->next = a ? a : b;
  }
  return dummy.next;
}
struct free_block_type *mergeSort(struct free_block_type *head, int criterion) {
  if (!head || !head->next) return head;

  struct free_block_type *a = head, *b = head->next;
  while (b && b->next) {
    head = head->next;
    b = b->next->next;
  }
  b = head->next;
  head->next = NULL;

  return merge(mergeSort(a, criterion), mergeSort(b, criterion), criterion);
}

void rearrange_FF() { return; }
void rearrange_BF() { free_block = mergeSort(free_block, MA_BF); }
```

```
void rearrange_WF() { free_block = mergeSort(free_block, MA_WF); }
void rearrange(int algorithm) {
  switch (algorithm) {
    case MA_FF:
      rearrange_FF();
      break;
    case MA_BF:
      rearrange_BF();
      break;
    case MA_WF:
      rearrange_WF();
      break;
    default:
      printf("Invalid algorithm.\n");
  }
}

void sort_free_blocks() {
  // insertion sort
  struct free_block_type *current, *next, *tmp, *pre;

  if (!free_block || !free_block->next) return;

  current = free_block->next;
  free_block->next = NULL;

  while (current) {
    pre = NULL;
    next = current->next;

    tmp = free_block;
    while (tmp->next && tmp->next->start_addr < current->start_addr) {
      pre = tmp;
      tmp = tmp->next;
    }
    if (!pre) {
      current->next = free_block;
      free_block = current;
    } else {
      current->next = pre->next;
      pre->next = current;
    }
    current = next;
  }
```

```
}

int allocate_mem(struct allocated_block *ab) {
  struct free_block_type *fb, *pre;
  int request_size = ab->size;
  fb = pre = free_block;
  int total_free_size = 0;
  int check = 0;
  // 根据当前算法在空闲分区链表中搜索合适空闲分区进行分配，分配时注意以下情况：
  while (fb) {
    if (fb->size - request_size >= MIN_SLICE) {
      //  1. 找到可满足空闲分区且分配后剩余空间足够大，则分割
      ab->start_addr = fb->start_addr;
      ab->size = request_size;
      fb->start_addr += request_size;
      fb->size -= request_size;
      goto success;
    } else if (fb->size >= request_size) {
      //  2. 找到可满足空闲分区且但分配后剩余空间比较小，则一起分配
      ab->start_addr = fb->start_addr;
      ab->size = fb->size;
      pre->next = fb->next;
      if (fb == free_block) check = 1;
      free(fb);
      if (check) free_block = NULL;  // avoid xuangua pointer
      goto success;
    } else {
      total_free_size += fb->size;
      pre = fb;
      fb = fb->next;
    }
  }
  if (total_free_size >= request_size) {
    memory_compaction();
    //  3.
    //  找不可满足需要的空闲分区但空闲分区之和能满足需要，则采用内存紧缩技术，进行空闲分区的合并，然后再分配
    fb = pre = free_block;
    while (fb) {
      if (fb->size - request_size >= MIN_SLICE) {
        ab->start_addr = fb->start_addr;
        ab->size = request_size;
        fb->start_addr += request_size;
        fb->size -= request_size;
```

```
        goto success;
      } else if (fb->size >= request_size) {
        ab->start_addr = fb->start_addr;
        ab->size = fb->size;
        pre->next = fb->next;
        if (fb == free_block) check = 1;
        free(fb);
        if (check) free_block = NULL;  // avoid xuangua pointer
        goto success;
      } else {
        pre = fb;
        fb = fb->next;
      }
    }
  }
  return -1;

success:
  rearrange(ma_algorithm);
  return 1;
  //  4. 在成功分配内存后，应保持空闲分区按照相应算法有序
  //  5. 分配成功则返回 1，否则返回-1
}
void memory_compaction() {
  // Step 1: 将所有已分配的块移动到内存的起始地址
  int new_start_addr = 0;
  struct allocated_block *cur = allocated_block_head;
  while (cur) {
    cur->start_addr = new_start_addr;
    new_start_addr += cur->size;
    cur = cur->next;
  }
  // Step 2: 释放所有旧的空闲块
  struct free_block_type *fb = free_block;
  while (fb) {
    struct free_block_type *tmp = fb;
    fb = fb->next;
    free(tmp);
  }

  // Step 3: 创建一个新的空闲块，其起始地址为最后一个已分配块的结束地址
  free_block = (struct free_block_type *)malloc(sizeof(struct free_block_type));
  free_block->start_addr = new_start_addr;
  free_block->size = mem_size - new_start_addr;
```

```
  free_block->next = NULL;
}

void new_process() {
  struct allocated_block *ab;
  int size;
  int ret;
  ab = (struct allocated_block *)malloc(sizeof(struct allocated_block));
  if (!ab) exit(-5);
  ab->next = NULL;
  pid++;
  sprintf(ab->process_name, "PROCESS-%02d", pid);
  ab->pid = pid;
  printf("Memory for %s: ", ab->process_name);
  scanf("%d", &size);
  if (size > 0) ab->size = size;
  ret = allocate_mem(ab);
  if ((ret == 1) && (allocated_block_head == NULL)) {
    allocated_block_head = ab;
  } else if (ret == 1) {
    ab->next = allocated_block_head;
    allocated_block_head = ab;
  } else if (ret == -1) {
    printf("Allocation fail\n");
    free(ab);
  }
}

void kill_process() {
  struct allocated_block *ab;
  int pid;
  printf("Kill Process, pid = ");
  scanf("%d", &pid);
  ab = find_process(pid);
  if (ab != NULL) {
    free_mem(ab); /*释放 ab 所表示的分配区*/
    dispose(ab);  /*释放 ab 数据结构节点*/
  }
}
void free_mem(struct allocated_block *ab) {
  int algorithm = ma_algorithm;
  struct free_block_type *fb;
  fb = (struct free_block_type *)malloc(sizeof(struct free_block_type));
  if (!fb) {
```

```
    printf("malloc fail in free_mem\n");
    return;
  }
  fb->size = ab->size;
  fb->start_addr = ab->start_addr;
  fb->next = free_block;
  free_block = fb;
  sort_free_blocks();
  compact();
  rearrange(ma_algorithm);
  // 进行可能的合并，基本策略如下
  // 1. 将新释放的结点插入到空闲分区队列末尾 I choose the head not end.
  // 2. 对空闲链表按照地址有序排列
  // 3. 检查并合并相邻的空闲分区
  // 4. 将空闲链表重新按照当前算法排序
}
void dispose(struct allocated_block *free_ab) {
  struct allocated_block *pre, *ab;
  if (free_ab == allocated_block_head) { /*如果要释放第一个节点*/
    allocated_block_head = allocated_block_head->next;
    free(free_ab);
    return;
  }
  pre = allocated_block_head;
  ab = allocated_block_head->next;
  while (ab != free_ab) {
    pre = ab;
    ab = ab->next;
  }
  pre->next = ab->next;
  free(ab);
}

void compact() {
  struct free_block_type *fb = free_block;
  struct free_block_type *next = NULL;

  while (fb && fb->next) {
    next = fb->next;
    if (fb->start_addr + fb->size == next->start_addr) {
      fb->size += next->size;
      fb->next = next->next;
      free(next);
    } else {
```

```
      fb = fb->next;
    }
  }
}

struct allocated_block *find_process(int pid) {
  struct allocated_block *ab = allocated_block_head;
  while (ab != NULL) {
    if (ab->pid == pid) {
      return ab;
    }
    ab = ab->next;
  }
  return NULL;
}

void kill_block(struct allocated_block *ab) {
  struct allocated_block *pre_ab = allocated_block_head;
  if (ab == allocated_block_head) {
    allocated_block_head = ab->next;
    free(ab);
    return;
  }
  while (pre_ab) {
    if (pre_ab->next == ab) {
      pre_ab->next = ab->next;
      free(ab);
      return;
    }
    pre_ab = pre_ab->next;
  }
}

void display_mem_usage() {
  struct free_block_type *fbt = free_block;
  struct allocated_block *ab = allocated_block_head;

  if (fbt == NULL) {
    printf("No free memory blocks\n");
    return;
  }

  printf("----------------------------------------------------------\n");
  printf("Free Memory:\n");
```

```
  printf("%20s %20s\n", "start_addr", "size");
  while (fbt != NULL) {
    printf("%20d %20d\n", fbt->start_addr, fbt->size);
    fbt = fbt->next;
  }
  printf("\nUsed Memory:\n");
  printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start_addr", "size");
  while (ab != NULL) {
    printf("%10d %20s %10d %10d\n", ab->pid, ab->process_name, ab->start_addr,
           ab->size);
    ab = ab->next;
  }

  printf("------------------------------------------------------------\n");
}

int main() {
  char choice;
  pid = 0;
  free_block = init_free_block(mem_size);  // 初始化空闲区
  while (1) {
    display_menu();
    // 使用 scanf 来读取选择，忽略前导空白符
    scanf(" %c", &choice);
    switch (choice) {
      case '1':
        set_mem_size();
        break;
      case '2':
        set_algorithm();
        flag = 1;
        break;
      case '3':
        new_process();
        flag = 1;
        break;
      case '4':
        kill_process();
        flag = 1;
        break;
      case '5':
        display_mem_usage();
        flag = 1;
        break;
```

```
      case '0':
        do_exit();
        exit(0);
      default:
        break;
    }
  }
}

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

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

2_kill.c

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

pid_t pid1 = -1, pid2 = -1;
```

```
// 处理父进程接收到的中断信号
void handle_parent_interrupt(int sig) {
    if (pid1 <= 0 || pid2 <= 0) return; // 子进程空操作
    printf("\nReceived soft interrupt\n");
    kill(pid1, 16);
    printf("Sent 16 signal successfully\n");
    kill(pid2, 17);
    printf("Sent 17 signal successfully\n");
}

// 处理子进程1收到的信号
void handle_child1_signal(int sig) {
    printf("Child process 1 is killed by parent!!\n");
    exit(0); // 直接退出子进程
}

// 处理子进程2收到的信号
void handle_child2_signal(int sig) {
    printf("Child process 2 is killed by parent!!\n");
    exit(0); // 直接退出子进程
}

int main() {
    signal(SIGINT, handle_parent_interrupt);   // 捕获 SIGINT
    signal(SIGQUIT, handle_parent_interrupt);  // 捕获 SIGQUIT

    // 创建子进程1
    while (pid1 == -1) pid1 = fork();
    if (pid1 > 0) { // 父进程
        // 创建子进程2
        while (pid2 == -1) pid2 = fork();
        if (pid2 > 0) { // 父进程
            printf("Start waiting for 5 seconds...\n");
            sleep(5);
            while (wait(NULL) != -1); // 等待子进程
            printf("\nParent process is killed!!\n");
        } else { // 子进程 2
            signal(17, handle_child2_signal);
            printf("Child process 2 is ready\n");
            pause(); // 等待信号
        }
    } else { // 子进程 1
        signal(16, handle_child1_signal);
        printf("Child process 1 is ready\n");
```

```
        pause(); // 等待信号
    }

    return 0;
}
```

2_lockf.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h> // 包含此头文件以使用 lockf

int pid1, pid2; // 定义两个进程变量

int main() {
    int fd[2];
    char InPipe[10000] = {0}; // 定义读缓冲区，并初始化
    char c1 = '1', c2 = '2';
    pipe(fd); // 创建管道

    // 创建进程 1
    while ((pid1 = fork()) == -1); // 如果进程 1 创建不成功，则空循环
    if (pid1 == 0) { // 子进程 1
        lockf(fd[1], 1, 0); // 锁定管道
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1); // 向管道写入字符 '1'
        }
        lockf(fd[1], 0, 0); // 解除管道的锁定
        exit(0); // 结束进程 1
    }
    else {
        // 创建进程 2
        while ((pid2 = fork()) == -1); // 如果进程 2 创建不成功，则空循环
        if (pid2 == 0) {
            lockf(fd[1], 1, 0); // 锁定管道
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1); // 向管道写入字符 '2'
            }
            lockf(fd[1], 0, 0); // 解除管道的锁定
            exit(0); // 结束进程 2
        }
```

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

            close(fd[1]); // 关闭写端

            // 从管道中读出 4000 个字符
            read(fd[0], InPipe, 4000);
            InPipe[4000] = '\0'; // 加字符串结束符
            printf("%s\n", InPipe); // 显示读出的数据

            close(fd[0]); // 关闭读端
            exit(0); // 父进程结束
        }
    }
}
```

2_nlsleep.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h> // 添加此行以使用 usleep

int pid1, pid2; // 定义两个进程变量

int main() {
    int fd[2];
    char InPipe[10000] = {0}; // 定义读缓冲区,并初始化
    char c1 = '1', c2 = '2';
    pipe(fd); // 创建管道

    // 创建进程 1
    while ((pid1 = fork()) == -1); // 如果进程 1 创建不成功,则空循环
    if (pid1 == 0) { // 子进程 1
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1); // 向管道写入字符 '1'
            usleep(100); // 暂停 100 微秒
        }
        exit(0); // 结束进程 1
    }
    else {
```

```
        // 创建进程 2
        while ((pid2 = fork()) == -1); // 如果进程 2 创建不成功,则空循环
        if (pid2 == 0) {
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1); // 向管道写入字符 '2'
                usleep(100); // 暂停 100 微秒
            }
            exit(0); // 结束进程 2
        }
        else {
            waitpid(pid1, NULL, 0); // 等待子进程 1 结束
            waitpid(pid2, NULL, 0); // 等待子进程 2 结束

            close(fd[1]); // 关闭写端

            // 从管道中读出 4000 个字符
            read(fd[0], InPipe, 4000);
            InPipe[4000] = '\0'; // 加字符串结束符
            printf("%s\n", InPipe); // 显示读出的数据

            close(fd[0]); // 关闭读端
            exit(0); // 父进程结束
        }
    }
}
```

2_nolock.c

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

int pid1, pid2; // 定义两个进程变量

int main() {
    int fd[2];
    char InPipe[10000] = {0}; // 定义读缓冲区,并初始化
    char c1 = '1', c2 = '2';
    pipe(fd); // 创建管道

    // 创建进程 1
    while ((pid1 = fork()) == -1); // 如果进程 1 创建不成功,则空循环
```

```
    if (pid1 == 0) { // 子进程 1
        for (int i = 0; i < 2000; i++) {
            write(fd[1], &c1, 1); // 向管道写入字符 '1'
        }
        exit(0); // 结束进程 1
    }
    else {
        // 创建进程 2
        while ((pid2 = fork()) == -1); // 如果进程 2 创建不成功,则空循环
        if (pid2 == 0) {
            for (int i = 0; i < 2000; i++) {
                write(fd[1], &c2, 1); // 向管道写入字符 '2'
            }
            exit(0); // 结束进程 2
        }
        else {
            waitpid(pid1, NULL, 0); // 等待子进程 1 结束
            waitpid(pid2, NULL, 0); // 等待子进程 2 结束

            close(fd[1]); // 关闭写端

            // 从管道中读出 4000 个字符
            read(fd[0], InPipe, 4000);
            InPipe[4000] = '\0'; // 加字符串结束符
            printf("%s\n", InPipe); // 显示读出的数据

            close(fd[0]); // 关闭读端
            exit(0); // 父进程结束
        }
    }
}
```

Ext2.c

```
#include <stdio.h>
#include "string.h"
#include "stdlib.h"
#include "time.h"
#include <sys/ioctl.h>
#include <sys/file.h>
#include <termios.h>
#include <unistd.h> // 用于 STDIN_FILENO

// 文件系统相关宏定义
```

```
#define blocks 4611             // 总块数 (1+1+1+512+4096)
#define blocksiz 512            // 每块大小 (字节数)
#define inodesiz 64             // 索引节点大小 (字节数)
#define data_begin_block 515    // 数据块起始块号
#define dirsiz 32               // 目录项长度 (字节数)
#define EXT2_NAME_LEN 15        // 文件名最大长度
#define PATH "MY_DISK"            // 虚拟磁盘文件路径

// 组描述符结构体，描述文件系统组的相关信息，占 68 字节
typedef struct ext2_group_desc {
    char bg_volume_name[16];  // 卷名
    int bg_block_bitmap;      // 块位图所在块号
    int bg_inode_bitmap;      // 索引节点位图所在块号
    int bg_inode_table;       // 索引节点表起始块号
    int bg_free_blocks_count; // 组内空闲块数
    int bg_free_inodes_count; // 组内空闲索引节点数
    int bg_used_dirs_count;   // 组内目录数
    char password[16];              // 文件系统密码
    char bg_pad[36];          // 填充
} ext2_group_desc;

// 索引节点结构体，描述文件或目录的元数据，占 64 字节
typedef struct ext2_inode {
    int i_mode;     // 文件类型和访问权限 (1: 普通文件, 2: 目录)
    int i_blocks;   // 文件内容占用的数据块数量
    int i_size;     // 文件大小 (字节)
    time_t i_atime; // 文件最近访问时间
    time_t i_ctime; // 文件创建时间
    time_t i_mtime; // 文件最近修改时间
    time_t i_dtime; // 文件删除时间
    int i_block[8]; // 数据块指针数组 (包括直接索引和间接索引)
    char i_pad[36]; // 填充
} ext2_inode;

// 目录项结构体，描述目录中的一个文件或子目录，占 32 字节
typedef struct ext2_dir_entry {
    int inode;                // 关联的索引节点号
    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;
```

```
// 全局变量定义
ext2_group_desc group_desc;      // 组描述符实例
ext2_inode inode;                // 索引节点实例
ext2_dir_entry dir;              // 目录项实例 (存储文件或目录的元数据)
FILE *f;                         // 文件指针 (用于文件系统操作)
unsigned int last_allco_inode = 0; // 上次分配的索引节点号
unsigned int last_allco_block = 0; // 上次分配的数据块号

/**********第一部分**********/
/**********初始化模拟文件系统程序设计**********/

/*从文件系统中读取根目录的 inode 数据，并将其存储到 cu 指针所指的 ext2_inode 结构体中。*/
int initialize(ext2_inode *cu)
{
    f = fopen(PATH, "r+");                          // 打开文件以读写模式操作
    fseek(f, 3 * blocksiz, 0);                      // 定位到第 3 个块，读取根目录的 inode
    fread(cu, sizeof(ext2_inode), 1, f);            // 将数据读入 cu 指针所指的结构体
    fclose(f);                                      // 关闭文件
    return 0;
}

/*捕获用户输入字符，不显示输入内容。*/
int getch() {
    int ch; // 用于保存读取的字符
    struct termios oldt, newt; // 终端参数结构体，用于保存和修改终端设置

    // 获取当前终端的参数设置，并保存到 `oldt` 中
    // STDIN_FILENO 是标准输入（文件描述符 0）
    tcgetattr(STDIN_FILENO, &oldt);
    // 复制当前终端设置到 `newt`，准备修改
    newt = oldt;
    // 修改终端设置：
    // 关闭回显（ECHO）：输入字符不会在终端显示
    // 关闭规范模式（ICANON）：输入字符不需要按 Enter 键即可读取
    newt.c_lflag &= ~(ECHO | ICANON);
    // 将修改后的参数立即生效到终端，TCSANOW 表示立即生效
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
    // 从标准输入读取单个字符
    ch = getchar();
    // 恢复原始终端设置（如启用回显和规范模式）
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
```

```
    // 返回用户输入的字符
    return ch;
}

/*退出显示函数，输出感谢信息并退出*/
void exitdisplay()
{
    printf("感谢使用~ 拜拜捏！\n");
    return;
}

int format(ext2_inode *current);
/*初始化文件系统,如果文件系统初始化成功，返回 0;如果文件系统初始化失败，返回 1*/
int initfs(ext2_inode *cu)
{
    f = fopen(PATH, "r+");
    if (f == NULL) // 如果文件系统文件不存在
    {
        char ch; // 用于存储用户输入的命令
        int i;
        printf("未找到文件系统。是否要创建一个新的文件系统？\n[Y/N]");
        i = 1;
        while (i) // 循环等待用户输入
        {
            scanf("%c", &ch); // 获取用户输入
            switch (ch)
            {
            case 'Y':
            case 'y': // 用户选择创建新文件系统
                if (format(cu) != 0) // 格式化文件系统
                    return 1; // 格式化失败，返回 1
                f = fopen(PATH, "r"); // 打开文件
                i = 0; // 停止循环
                break;
            case 'N':
            case 'n': // 用户选择不创建文件系统
                exitdisplay(); // 显示退出信息
                return 1; // 返回 1，表示初始化失败
            default:
                printf("无效命令，请重新输入\n"); // 输入无效命令，提示用户
                break;
            }
        }
    }
```

```
    // 如果文件存在，读取文件系统信息
    fseek(f, 0, SEEK_SET);
    fread(&group_desc, sizeof(ext2_group_desc), 1, f); // 读取组描述符
    fseek(f, 3 * blocksiz, SEEK_SET);
    fread(&inode, sizeof(ext2_inode), 1, f); // 读取根目录的索引节点
    fclose(f);

    initialize(cu); // 初始化当前目录
    return 0; // 返回 0，表示初始化成功
}

/**********第二部分**********/
/**********文件系统级函数及其子函数设计**********/

/*查找空闲索引节点*/
int FindInode()
{
    FILE *fp = NULL;
    unsigned int zero[blocksiz / 4]; // 用于保存 inode 位图
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+"); // 打开文件系统
    fseek(fp, 2 * blocksiz, SEEK_SET); // 定位到 inode 位图
    fread(zero, blocksiz, 1, fp); // 读取 inode 位图

    for (i = last_allco_inode; i < (last_allco_inode + blocksiz / 4); i++) // 遍
历 inode 位图
    {
        if (zero[i % (blocksiz / 4)] != 0xffffffff) // 检查是否有空闲的 inode
        {
            unsigned int j = 0x80000000, k = zero[i % (blocksiz / 4)], l = i;
            for (i = 0; i < 32; i++) // 遍历 32 位，找到空闲的位
            {
                if (!(k & j)) // 判断当前位是否空闲
                {
                    zero[l % (blocksiz / 4)] |= j; // 将空闲位置为已占用
                    group_desc.bg_free_inodes_count -= 1; // 更新组描述符中的空
闲 inode 计数
                    fseek(fp, 0, 0); // 定位到组描述符块
                    fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp); // 写回
组描述符

                    fseek(fp, 2 * blocksiz, SEEK_SET); // 定位到 inode 位图
```

```
                    fwrite(zero, blocksiz, 1, fp); // 更新 inode 位图

                    last_allco_inode = 1 % (blocksiz / 4); // 记录最后分配的 inode
                    fclose(fp);
                    return 1 % (blocksiz / 4) * 32 + i; // 返回空闲 inode 编号
                }
                else
                    j /= 2; // 检查下一位
            }
        }
    }
    fclose(fp);
    return -1; // 没有空闲 inode
}

/*查找空闲块*/
int FindBlock()
{
    FILE *fp = NULL;
    unsigned int zero[blocksiz / 4]; // 用于保存块位图
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+"); // 打开文件系统
    fseek(fp, 1 * blocksiz, SEEK_SET); // 定位到块位图
    fread(zero, blocksiz, 1, fp); // 读取块位图

    for (i = last_allco_block; i < (last_allco_block + blocksiz / 4); i++) // 遍
历块位图
    {
        if (zero[i % (blocksiz / 4)] != 0xffffffff) // 检查是否有空闲块
        {
            unsigned int j = 0x80000000, k = zero[i % (blocksiz / 4)], l = i;
            for (i = 0; i < 32; i++) // 遍历 32 位，找到空闲的位
            {
                if (!(k & j)) // 判断当前位是否空闲
                {
                    zero[l % (blocksiz / 4)] |= j; // 将空闲位置为已占用
                    group_desc.bg_free_blocks_count -= 1; // 更新组描述符中的空
闲块计数
                    fseek(fp, 0, 0); // 定位到组描述符块
                    fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp); // 写回
组描述符

                    fseek(fp, 1 * blocksiz, SEEK_SET); // 定位到块位图
```

```
                    fwrite(zero, blocksiz, 1, fp); // 更新块位图

                    last_allco_block = l % (blocksiz / 4); // 记录最后分配的块
                    fclose(fp);
                    return l % (blocksiz / 4) * 32 + i; // 返回空闲块编号
                }
                else
                    j /= 2; // 检查下一位
            }
        }
    }
    fclose(fp);
    return -1; // 没有空闲块
}

// 删除指定的 inode 节点，并更新 inode 位图
void DelInode(int len) // len 是 inode 号
{
    unsigned int zero[blocksiz / 4], i;
    int j;
    f = fopen(PATH, "r+"); // 打开文件系统路径
    fseek(f, 2 * blocksiz, SEEK_SET); // 定位到 inode 位图的起始位置
    fread(zero, blocksiz, 1, f); // 读取 inode 位图到 zero 数组
    i = 0x80000000; // 用于按位操作的初始值
    for (j = 0; j < len % 32; j++)
        i = i / 2; // 将 i 右移，定位到对应的位
    zero[len / 32] = zero[len / 32] ^ i; // 通过按位异或清除指定 inode
    fseek(f, 2 * blocksiz, SEEK_SET); // 定位回 inode 位图的起始位置
    fwrite(zero, blocksiz, 1, f); // 写回更新后的 inode 位图
    fclose(f); // 关闭文件
}

// 删除指定的数据块，并更新块位图
void DelBlock(int len)
{
    unsigned int zero[blocksiz / 4], i;
    int j;
    f = fopen(PATH, "r+"); // 打开文件系统路径
    fseek(f, 1 * blocksiz, SEEK_SET); // 定位到块位图的起始位置
    fread(zero, blocksiz, 1, f); // 读取块位图到 zero 数组
    i = 0x80000000; // 用于按位操作的初始值
    for (j = 0; j < len % 32; j++)
        i = i / 2; // 将 i 右移，定位到对应的位
    zero[len / 32] = zero[len / 32] ^ i; // 通过按位异或清除指定块
```

```
    fseek(f, 1 * blocksiz, SEEK_SET); // 定位回块位图的起始位置
    fwrite(zero, blocksiz, 1, f); // 写回更新后的块位图
    fclose(f); // 关闭文件
}

// 增加一个数据块到当前文件中，支持直接索引、一级索引和二级索引
void add_block(ext2_inode *current, int i, int j) // i 表示数据块序号，j 是新分配的数据块号
{
    FILE *fp = NULL;
    while (fp == NULL)
        fp = fopen(PATH, "r+"); // 打开文件系统路径

    if (i < 6) // 使用直接索引
    {
        current->i_block[i] = j; // 将新数据块号直接写入 i_block 数组
    }
    else
    {
        i = i - 6; // 索引号减去直接索引的数量
        if (i == 0) // 为一级索引分配新数据块
        {
            current->i_block[6] = FindBlock(); // 分配一个新的数据块存储一级索引
            fseek(fp, data_begin_block * blocksiz + current->i_block[6] * blocksiz, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp); // 写入第一个数据块号
        }
        else if (i < 128) // 一级索引中分配数据块
        {
            fseek(fp, data_begin_block * blocksiz + current->i_block[6] * blocksiz + i * 4, SEEK_SET);
            fwrite(&j, sizeof(int), 1, fp); // 写入对应数据块号
        }
        else // 二级索引
        {
            i = i - 128; // 索引号减去一级索引的数量
            if (i == 0) // 分配二级索引块
            {
                current->i_block[7] = FindBlock(); // 分配一个新的二级索引块
                fseek(fp, data_begin_block * blocksiz + current->i_block[7] * blocksiz, SEEK_SET);
                i = FindBlock(); // 为二级索引块分配一个新的数据块
                fwrite(&i, sizeof(int), 1, fp);
                fseek(fp, data_begin_block * blocksiz + i * blocksiz, SEEK_SET);
```

```
                fwrite(&j, sizeof(int), 1, fp); // 写入数据块号
            }
            if (i % 128 == 0) // 当前索引需要分配新的块
            {
                fseek(fp, data_begin_block * blocksiz + current->i_block[7] *
blocksiz + i / 128 * 4, SEEK_SET);
                i = FindBlock(); // 分配新的块
                fwrite(&i, sizeof(int), 1, fp);
                fseek(fp, data_begin_block * blocksiz + i * blocksiz, SEEK_SET);
                fwrite(&j, sizeof(int), 1, fp); // 写入数据块号
            }
            else // 二级索引中已存在块
            {
                fseek(fp, data_begin_block * blocksiz + current->i_block[7] *
blocksiz + i / 128 * 4, SEEK_SET);
                fread(&i, sizeof(int), 1, fp); // 读取二级索引块号
                fseek(fp, data_begin_block * blocksiz + i * blocksiz + i % 128 *
4, SEEK_SET);
                fwrite(&j, sizeof(int), 1, fp); // 写入数据块号
            }
        }
    }
}

// 返回目录的存储位置偏移量，每个目录项占 32 字节
int dir_entry_position(int dir_entry_begin, int i_block[8]) // dir_entry_begin 表
示目录项的相对起始字节
{
    int dir_blocks = dir_entry_begin / 512;   // 目录项跨越的块数
    int block_offset = dir_entry_begin % 512; // 当前块内的字节偏移量
    int a; // 临时变量用于存储索引值
    FILE *fp = NULL;

    if (dir_blocks <= 5) // 如果目录项在直接索引块中
    {
        return data_begin_block * blocksiz + i_block[dir_blocks] * blocksiz +
block_offset;
    }
    else // 如果目录项在间接索引块中
    {
        while (fp == NULL) // 确保文件成功打开
            fp = fopen(PATH, "r+");

        dir_blocks -= 6; // 减去直接索引块数量
```

```
        if (dir_blocks < 128) // 一级间接索引处理（128 为单块可容纳的索引数）
        {
            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 // 二级间接索引处理
        {
            dir_blocks -= 128; // 减去一级间接索引块数量

            // 定位二级间接索引块并读取其值
            fseek(fp, data_begin_block * blocksiz + i_block[7] * blocksiz +
(dir_blocks / 128) * 4, SEEK_SET);
            fread(&a, sizeof(int), 1, fp);

            // 定位对应的一级间接索引块并读取最终地址
            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); // 关闭文件
    }
}

// 为当前目录寻找一个空目录条目位置并返回绝对地址
int FindEntry(ext2_inode *current)
{
    FILE *fout = NULL;
    int location;       // 条目的绝对位置
    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; // 从数据块起始位置开始计算
    temp = blocksiz / sizeof(int); // 计算每个块可以存放多少个 INT
    fout = fopen(PATH, "r+"); // 以读写模式打开文件
    if (current->i_size % blocksiz == 0) // 如果当前目录的大小是块的整数倍，说明
当前块已满，需要增加一个新块
    {
        add_block(current, current->i_blocks, FindBlock()); // 增加一个新的数据
```

```
块
        current->i_blocks++; // 更新块计数
    }
    if (current->i_blocks < 6) // 前6个块为直接索引块
    {
        location += current->i_block[current->i_blocks - 1] * blocksiz; // 计算当前目录条目的位置
        location += current->i_size % blocksiz; // 在当前块内找到位置
    }
    else if (current->i_blocks < temp + 5) // 一级索引
    {
        block_location = current->i_block[6]; // 获取一级索引的块号
        fseek(fout, (data_begin_block + block_location) * blocksiz + (current->i_blocks - 6) * sizeof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout); // 读取块号
        location += block_location * blocksiz; // 更新位置
        location += current->i_size % blocksiz; // 在该块内找到位置
    }
    else // 二级索引
    {
        block_location = current->i_block[7]; // 获取二级索引的块号
        remain_block = current->i_blocks - 6 - temp; // 计算剩余块数
        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); // 读取二级索引块号
        remain_block = remain_block % temp; // 计算剩余块数
        fseek(fout, (data_begin_block + block_location) * blocksiz + remain_block * sizeof(int), SEEK_SET);
        fread(&block_location, sizeof(int), 1, fout); // 读取块号
        location += block_location * blocksiz; // 更新位置
        location += current->i_size % blocksiz + dirsiz; // 在该块内找到位置并加上目录条目大小
    }
    current->i_size += dirsiz; // 更新当前目录的大小
    fclose(fout); // 关闭文件
    return location; // 返回找到的空目录条目的位置
}

/*登录函数，用于验证输入的密码与存储的密码是否匹配，如果密码匹配，则返回 0；如果密码不匹配，则返回非零值*/
int login()
{
    char psw[16]; // 用于存储输入的密码
    printf("请输入密码（原始密码为9331）：");
```

```
    scanf("%s", psw); // 输入密码
    return strcmp(group_desc.password, psw); // 比较输入的密码与存储的密码
}

int Open(ext2_inode *current, char *name);//声明 Open 函数

/*获取当前目录的目录名，参数 cs_name_name：用于存储获取到的目录名，参数 node：当前目录的索引节点*/
void getstring(char *cs_name, ext2_inode node)
{
    ext2_inode current = node; // 当前目录节点
    int i, j;
    ext2_dir_entry dir; // 目录项
    f = fopen(PATH, "r+"); // 打开文件系统

    // 打开父目录
    Open(&current, ".."); // current 指向父目录（上级目录）

    // 查找当前目录中的"."（表示当前目录）项，获取其对应的索引节点
    for (i = 0; i < node.i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, node.i_block), SEEK_SET); // 定位到当前目录项
        fread(&dir, sizeof(ext2_dir_entry), 1, f); // 读取目录项
        if (!strcmp(dir.name, ".")) // 如果目录项为"."，即当前目录
        {
            j = dir.inode; // 保存该目录项对应的索引节点
            break;
        }
    }

    // 查找父目录中与当前目录相对应的目录项，获取其名称
    for (i = 0; i < current.i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, current.i_block), SEEK_SET); // 定位到父目录项
        fread(&dir, sizeof(ext2_dir_entry), 1, f); // 读取目录项
        if (dir.inode == j) // 如果该目录项的索引节点与当前目录相同
        {
            strcpy(cs_name, dir.name); // 将目录名复制到传入的字符串 cs_name 中
            return; // 返回
        }
    }
}
```

```
/**********第三部分**********/
/**********命令层函数的设计**********/

/*打开指定目录并将其设置为当前目录,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) // 确保文件成功打开
        fp = fopen(PATH, "r+");

    for (i = 0; i < (current->i_size / 32); i++) // 遍历当前目录中的所有目录项
    {
        // 定位当前目录项的存储位置
        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)) // 匹配目标目录名
        {
            if (dir.file_type == 2) // 如果是目录类型
            {
                // 读取目标目录的索引节点信息
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fread(current, sizeof(ext2_inode), 1, fp);
                fclose(fp); // 关闭文件
                return 0;   // 打开成功
            }
        }
    }

    fclose(fp); // 关闭文件
    return 1;   // 打开失败
}

/*关闭当前目录,仅更新最后访问时间，返回上一目录作为新的当前目录*/
int Close(ext2_inode *current)
{
    time_t now;
    ext2_dir_entry parent_entry; // 父目录项信息
    FILE *fout;

    fout = fopen(PATH, "r+"); // 打开文件进行读写
```

```
    time(&now); // 获取当前时间

    current->i_atime = now; // 更新最后访问时间

    // 定位并读取当前目录对应的目录项
    fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz, SEEK_SET);
    fread(&parent_entry, sizeof(ext2_dir_entry), 1, fout);

    // 更新索引节点信息到文件系统
    fseek(fout, 3 * blocksiz + (parent_entry.inode) * sizeof(ext2_inode),
SEEK_SET);
    fwrite(current, sizeof(ext2_inode), 1, fout);

    fclose(fout); // 关闭文件

    // 打开父目录并设置为当前目录
    return Open(current, "..");
}

/*从当前目录中读取文件内容，name 为文件名*/
int Read(ext2_inode *current, char *name) {
    FILE *fp = NULL;
    int i;
    while (fp == NULL)
        fp = fopen(PATH, "r+"); // 打开文件系统

    int fd = fileno(fp);
    if (flock(fd, LOCK_SH) == -1) { // 添加共享锁
        perror("无法加读锁");
        fclose(fp);
        return -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) {
                time_t now;
                ext2_inode node;
                char content_char;
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fread(&node, sizeof(ext2_inode), 1, fp);
```

```
                for (i = 0; i < node.i_size; i++) {
                    fseek(fp, dir_entry_position(i, node.i_block), SEEK_SET);
                    fread(&content_char, sizeof(char), 1, fp);
                    if (content_char == 0xD)
                        printf("\n");
                    else
                        printf("%c", content_char);
                }
                printf("\n");

                time(&now);
                node.i_atime = now;
                fseek(fp, 3*blocksiz+dir.inode*sizeof(ext2_inode), SEEK_SET);
                fwrite(&node, sizeof(ext2_inode), 1, fp);

                flock(fd, LOCK_UN); // 释放锁
                fclose(fp);
                return 0;
            }
        }
    }

    flock(fd, LOCK_UN); // 释放锁
    fclose(fp);
    return 1; // 文件未找到
}


/*向目录 'current' 中的文件 'name' 写入数据。如果该文件在目录中不存在，则提示用户先创建文件*/
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+");

    int fd = fileno(fp);
    if (flock(fd, LOCK_EX) == -1) { // 添加独占锁
        perror("无法加写锁");
```

```
        fclose(fp);
        return -1;
    }

    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)
            break;

        printf("该文件不存在，请先创建文件\n");
        flock(fd, LOCK_UN); // 释放锁
        fclose(fp);
        return 0;
    }

    str = getch();
    while (str != 27) {
        printf("%c", str);

        if (!(node.i_size % 512)) {
            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 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET);
    fwrite(&node, sizeof(ext2_inode), 1, fp);

    flock(fd, LOCK_UN); // 释放锁
    fclose(fp);
    printf("\n");
    return 0;
}


/*创建目录，type=1 创建文件、type=2 创建目录、current 当前目录的索引节点、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
    {
        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("创建了.dir\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("创建了..dir\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;
}

/*在当前目录删除目录或文件*/
int Delete(int type, ext2_inode *current, char *name)
{
    FILE *fout = NULL;
    int i, j, t, k, flag;
    int node_location, dir_entry_location, block_location, e_location;
    int block_location2, block_location3;
    ext2_inode cinode;
    ext2_dir_entry centry, dentry, eentry;
    // 初始化删除条目的空结构
    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); // 计算当前目录条目数量
    flag = 0; // 用于标记是否找到目标文件或目录

    // 查找目录项，定位到目标文件或目录
    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;
        }
    }
    // 如果找到了目标文件或目录
    if (flag)
    {
        node_location = centry.inode;  // 获取 inode 号
        fseek(fout, 3 * blocksiz + node_location * sizeof(ext2_inode), SEEK_SET);
// 定位到 inode 位置
        fread(&cinode, sizeof(ext2_inode), 1, fout); // 读取 inode 信息

        // 删除目录
        if (type == 2)
        {
            while (cinode.i_size > 2 * dirsiz) // 删除目录中的内容，至少保留当前
目录项和"."目录项
            {
                fseek(fout,    dir_entry_position(cinode.i_size    -    dirsiz,
cinode.i_block), SEEK_SET);
                fread(&eentry, sizeof(ext2_dir_entry), 1, fout);
                Delete(eentry.file_type, &cinode, eentry.name); // 递归删除子目
录或文件
            }

            // 删除当前目录的块和 inode
            DelBlock(cinode.i_block[0]);
            DelInode(node_location);

            // 更新当前目录条目，删除目录项
            dir_entry_location = dir_entry_position(current->i_size - dirsiz,
current->i_block);
            fseek(fout, dir_entry_location, SEEK_SET);
            fread(&centry, dirsiz, 1, fout); // 读取最后一条目录项
            fseek(fout, dir_entry_location, SEEK_SET);
            fwrite(&dentry, dirsiz, 1, fout); // 清空该位置

            // 释放多余的数据块
            dir_entry_location -= data_begin_block * blocksiz;
            if (dir_entry_location % blocksiz == 0)
            {
                DelBlock(dir_entry_location / blocksiz);
                current->i_blocks--;
                if (current->i_blocks == 6)
                    DelBlock(current->i_block[6]);
```

```
                else if (current->i_blocks == (blocksiz / sizeof(int) + 6))
                {
                    int a;
                    fseek(fout, data_begin_block * blocksiz + current->i_block[7]
* blocksiz, SEEK_SET);
                    fread(&a, sizeof(int), 1, fout);
                    DelBlock(a);
                    DelBlock(current->i_block[7]);
                }
                else if (!((current->i_blocks - 6 - blocksiz / sizeof(int)) %
(blocksiz / sizeof(int))))
                {
                    int a;
                    fseek(fout, data_begin_block * blocksiz + current->i_block[7]
* blocksiz + ((current->i_blocks - 6 - blocksiz / sizeof(int)) / (blocksiz /
sizeof(int))), SEEK_SET);
                    fread(&a, sizeof(int), 1, fout);
                    DelBlock(a);
                }
            }
            current->i_size -= dirsiz;

            // 如果删除的条目不是最后一条，用最后一条目录项覆盖删除项
            if (j * dirsiz < current->i_size)
            {
                dir_entry_location    =    dir_entry_position(j    *    dirsiz,
current->i_block);
                fseek(fout, dir_entry_location, SEEK_SET);
                fwrite(&centry, dirsiz, 1, fout);
            }
            printf("目录 %s 被删掉了!\n", name);
        }
        // 删除文件
        else
        {
            // 删除直接指向的块
            for (i = 0; i < 6; i++)
            {
                if (cinode.i_blocks == 0)
                    break;
                block_location = cinode.i_block[i];
                DelBlock(block_location); // 删除块
                cinode.i_blocks--;
            }
```

```
            // 删除一级索引中的块
            if (cinode.i_blocks > 0)
            {
                block_location = cinode.i_block[6];
                fseek(fout, (data_begin_block + block_location) * blocksiz,
SEEK_SET);
                for (i = 0; i < blocksiz / sizeof(int); i++)
                {
                    if (cinode.i_blocks == 0)
                        break;
                    fread(&block_location2, sizeof(int), 1, fout);
                    DelBlock(block_location2); // 删除块
                    cinode.i_blocks--;
                }
                DelBlock(block_location); // 删除一级索引
            }

            // 删除二级索引中的块
            if (cinode.i_blocks > 0)
            {
                block_location = cinode.i_block[7];
                for (i = 0; i < blocksiz / sizeof(int); i++)
                {
                    fseek(fout, (data_begin_block + block_location) * blocksiz +
i * sizeof(int), SEEK_SET);
                    fread(&block_location2, sizeof(int), 1, fout);
                    fseek(fout, (data_begin_block + block_location2) * blocksiz,
SEEK_SET);
                    for (k = 0; i < blocksiz / sizeof(int); k++)
                    {
                        if (cinode.i_blocks == 0)
                            break;
                        fread(&block_location3, sizeof(int), 1, fout);
                        DelBlock(block_location3); // 删除块
                        cinode.i_blocks--;
                    }
                    DelBlock(block_location2); // 删除二级索引
                }
                DelBlock(block_location); // 删除一级索引
            }

            // 删除文件的 inode
            DelInode(node_location);
```

```
            // 更新当前目录的条目
            dir_entry_location = dir_entry_position(current->i_size - dirsiz,
current->i_block);
            fseek(fout, dir_entry_location, SEEK_SET);
            fread(&centry, dirsiz, 1, fout); // 读取最后一条目录项
            fseek(fout, dir_entry_location, SEEK_SET);
            fwrite(&dentry, dirsiz, 1, fout); // 清空该位置

            // 释放数据块
            dir_entry_location -= data_begin_block * blocksiz;
            if (dir_entry_location % blocksiz == 0)
            {
                DelBlock(dir_entry_location / blocksiz);
                current->i_blocks--;
                if (current->i_blocks == 6)
                    DelBlock(current->i_block[6]);
                else if (current->i_blocks == (blocksiz / sizeof(int) + 6))
                {
                    int a;
                    fseek(fout, data_begin_block * blocksiz + current->i_block[7]
* blocksiz, SEEK_SET);
                    fread(&a, sizeof(int), 1, fout);
                    DelBlock(a);
                    DelBlock(current->i_block[7]);
                }
                else if (!((current->i_blocks - 6 - blocksiz / sizeof(int)) %
(blocksiz / sizeof(int))))
                {
                    int a;
                    fseek(fout, data_begin_block * blocksiz + current->i_block[7]
* blocksiz + ((current->i_blocks - 6 - blocksiz / sizeof(int)) / (blocksiz /
sizeof(int))), SEEK_SET);
                    fread(&a, sizeof(int), 1, fout);
                    DelBlock(a);
                }
            }
            current->i_size -= dirsiz;

            // 如果删除的条目不是最后一条，用最后一条目录项覆盖删除项
            if (j * dirsiz < current->i_size)
            {
                dir_entry_location    =    dir_entry_position(j    *    dirsiz,
current->i_block);
                fseek(fout, dir_entry_location, SEEK_SET);
```

```
                fwrite(&centry, dirsiz, 1, fout);
            }

            printf("文件 %s 被删掉了!\n", name);
        }

        // 更新当前目录 inode
        fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz,
SEEK_SET);
        fread(&centry, sizeof(ext2_dir_entry), 1, fout);
        fseek(fout, 3 * blocksiz + (centry.inode) * sizeof(ext2_inode), SEEK_SET);
        fwrite(current, sizeof(ext2_inode), 1, fout); // 更新目录 inode
    }
    fclose(fout);
}

/* 列出当前目录中的文件和子目录*/
void ls(ext2_inode *current)
{
    ext2_dir_entry dir; // 目录项
    int i, j;
    char timestr[150]; // 用于存储时间的字符串
    ext2_inode node;   // 文件或目录的索引节点

    // 打开文件
    f = fopen(PATH, "r+");
    printf("类型\t\t 文件名\t\t 创建时间\t\t\t 最后访问时间\t\t\t 修改时间\n");
    printf("\n 注意! current->i_size:%d\n", current->i_size);

    // 遍历当前目录的所有条目
    for (i = 0; i < current->i_size / 32; i++)
    {
        fseek(f, dir_entry_position(i * 32, current->i_block), SEEK_SET); // 定
位到目录项
        fread(&dir, sizeof(ext2_dir_entry), 1, f);  // 读取目录项
        fseek(f, 3 * blocksiz + dir.inode * sizeof(ext2_inode), SEEK_SET); // 定
位到索引节点
        fread(&node, sizeof(ext2_inode), 1, f);     // 读取索引节点

        // 格式化时间字符串
        strcpy(timestr, "");
        strcat(timestr, asctime(localtime(&node.i_ctime))); // 创建时间
        strcat(timestr, asctime(localtime(&node.i_atime))); // 最后访问时间
        strcat(timestr, asctime(localtime(&node.i_mtime))); // 修改时间
```

```
        // 替换时间字符串中的换行符为制表符
        for (j = 0; j < strlen(timestr) - 1; j++)
            if (timestr[j] == '\n')
            {
                timestr[j] = '\t';
            }

        // 输出文件或目录的信息
        if (dir.file_type == 1)
            printf("文件\t\t%s\t\t%s", dir.name, timestr);
        else
            printf("目录\t\t%s\t\t%s", dir.name, timestr);
    }

    fclose(f); // 关闭文件
}

/*此函数用于修改文件系统的密码，如果密码修改成功，则返回 0；如果密码修改失败或用户取消修改，则返回 1*/
int Password()
{
    char psw[16], ch[10]; // 用于存储输入的密码和确认修改的命令
    printf("请输入旧密码：\n");
    scanf("%s", psw); // 输入当前密码
    if (strcmp(psw, group_desc.password) != 0) // 比对输入的旧密码与存储的密码
    {
        printf("密码错误！\n");
        return 1; // 密码错误，返回 1
    }
    while (1)
    {
        printf("请输入新密码：");
        scanf("%s", psw); // 输入新密码
        while (1)
        {
            printf("确定修改密码？[Y/N]");
            scanf("%s", ch); // 输入是否确认修改密码
            if (ch[0] == 'N' || ch[0] == 'n') // 用户选择取消修改
            {
                printf("您已取消密码修改\n");
                return 1; // 用户取消修改，返回 1
            }
            else if (ch[0] == 'Y' || ch[0] == 'y') // 用户确认修改
```

```
            {
                strcpy(group_desc.password, psw); // 更新密码
                f = fopen(PATH, "r+"); // 重新打开文件
                fseek(f, 0, 0); // 定位到文件开头
                fwrite(&group_desc, sizeof(ext2_group_desc), 1, f); // 保存新的
密码
                fclose(f);
                return 0; // 密码修改成功，返回 0
            }
            else
                printf("无效命令\n"); // 输入无效命令，提示用户
        }
    }
}

 /*显示当前目录的绝对路径*/
void pwd (char *str, ext2_inode *current) {
    FILE *fout = NULL;
    char string[100]; // 用于存储路径字符串
    char *slash = "/"; // 用于拼接路径时的分隔符

    int node_location, dir_entry_location, block_location;
    ext2_inode cinode; // 当前目录的 inode 结构
    ext2_dir_entry pentry, centry;  // 上级目录条目、当前目录条目

    fout = fopen(PATH, "r+"); // 打开文件系统

    // 定位到当前目录的"."条目，即当前目录自身
    fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz, SEEK_SET);
    fread(&centry, sizeof(ext2_dir_entry), 1, fout); // 读取当前目录的条目信息

    // 定位到当前目录的".."条目，即上一级目录
    fseek(fout, (data_begin_block + current->i_block[0]) * blocksiz + dirsiz,
SEEK_SET);
    fread(&pentry, sizeof(ext2_dir_entry), 1, fout); // 读取上一级目录的条目信息

    // 定位到上一级目录的 inode，并读取该 inode 的信息
    fseek(fout, 3 * blocksiz + (pentry.inode) * sizeof(ext2_inode), SEEK_SET);
    fread(&cinode, sizeof(ext2_inode), 1, fout); // 读取上一级目录的 inode 信息

    // 获取上一级目录的路径并保存到 string 中
    getstring(string, cinode);

    // 更新当前目录为上一级目录
```

```
    current = &cinode;

    // 如果当前目录的 inode 与上一级目录的 inode 不相同，说明没有到达根目录，需要递归
    if (centry.inode != pentry.inode) {
        // 拼接当前目录路径到上级路径
        strcat(string, slash);
        strcat(string, str); // 将当前目录名加到路径字符串后面
        strcpy(str, string); // 更新路径

        // 递归调用 pwd，继续向上查找上一级目录
        pwd(str, current);
    }
}

/*格式化模拟文件系统,包括初始化组描述符、位图和根目录。current 指向 ext2_inode 类型的指针，用于指向根目录。返回 0，表示成功。*/
int format(ext2_inode *current)
{
    FILE *fp = NULL;
    int i;
    unsigned int zero[blocksiz / 4];                // 用于填充块的零数组
    time_t now;
    time(&now);                                     // 获取当前时间
    // 保证文件打开成功
    while (fp == NULL)
        fp = fopen(PATH, "w+");                     // 打开文件以写模式操作
    // 初始化零数组
    for (i = 0; i < blocksiz / 4; i++)
        zero[i] = 0;
    // 清空所有数据块，将其初始化为零
    for (i = 0; i < blocks; i++)
    {
        fseek(fp, i * blocksiz, SEEK_SET);          // 定位到块的起始位置
        fwrite(&zero, blocksiz, 1, fp);             // 写入零数据
    }
    // 初始化组描述符
    strcpy(group_desc.bg_volume_name, "Volume_name"); // 设置卷名
    group_desc.bg_block_bitmap = 1;                  // 块位图所在块号
    group_desc.bg_inode_bitmap = 2;                  // 索引节点位图所在块号
    group_desc.bg_inode_table = 3;                   // 索引节点表起始块号
    group_desc.bg_free_blocks_count = 4095;          // 可用块数（除去根目录占用块）
    group_desc.bg_free_inodes_count = 4095;          // 可用索引节点数
```

```
group_desc.bg_used_dirs_count = 1;                // 已用目录数
strcpy(group_desc.password, "9331");                    // 设置默认密码

// 将组描述符写入第一块
fseek(fp, 0, SEEK_SET);
fwrite(&group_desc, sizeof(ext2_group_desc), 1, fp);

// 初始化块位图和索引节点位图，第一位标记为已用
zero[0] = 0x80000000;
fseek(fp, 1 * blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp);                   // 写入块位图
fseek(fp, 2 * blocksiz, SEEK_SET);
fwrite(&zero, blocksiz, 1, fp);                   // 写入索引节点位图

// 初始化索引节点表，设置根目录节点信息
inode.i_mode = 2;                                 // 目录类型
inode.i_blocks = 1;                               // 占用块数
inode.i_size = 64;                                // 目录大小
inode.i_ctime = now;                              // 创建时间
inode.i_atime = now;                              // 访问时间
inode.i_mtime = now;                              // 修改时间
inode.i_dtime = 0;                                // 删除时间（未删除）
fseek(fp, 3 * blocksiz, SEEK_SET);
fwrite(&inode, sizeof(ext2_inode), 1, fp);        // 写入索引节点表

// 初始化根目录的 "." 和 ".." 目录项
dir.inode = 0;                                    // 当前目录 inode 号
dir.rec_len = 32;                                 // 目录项长度
dir.name_len = 1;                                 // 名称长度
dir.file_type = 2;                                // 类型（目录）
strcpy(dir.name, ".");                            // 当前目录
fseek(fp, data_begin_block * blocksiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp);      // 写入当前目录

dir.inode = 0;                                    // 根目录上级目录仍为自身
dir.rec_len = 32;
dir.name_len = 2;                                 // 名称长度
dir.file_type = 2;                                // 类型（目录）
strcpy(dir.name, "..");                           // 上级目录
fseek(fp, data_begin_block * blocksiz + dirsiz, SEEK_SET);
fwrite(&dir, sizeof(ext2_dir_entry), 1, fp);      // 写入上级目录

// 调用初始化函数，设置当前目录指针为根目录
initialize(current);
```

```
    // 打印根目录 inode 大小调试信息
    printf("\n 注意! inode.i_size:%d\n", inode.i_size);
    fclose(fp);                                          // 关闭文件
    return 0;
}

/**********第四部分**********/
/**********界面及 main 函数设计**********/

/*模拟的 Shell 环境，程序进入一个无限循环，等待用户输入命令，并根据不同命令执行相应的操作。 */
void shellloop(ext2_inode currentdir)
{
    char command[10], var1[10], var2[128], path[10];
    ext2_inode temp;
    int i, j;
    char currentstring[20];
    // 命令表，存储支持的命令
    char ctable[14][10] = {"create", "delete", "cd", "close", "read", "write",
"password", "format", "exit", "login", "logout", "ls", "pwd", "help"};

    // 无限循环，等待用户输入命令
    while (1)
    {
        // 获取当前目录名称并打印提示符
        getstring(currentstring, currentdir);
        printf("\n[当前目录: %s]> ", currentstring);

        // 读取用户输入的命令
        scanf("%s", command);

        // 遍历命令表，找到对应的命令索引
        for (i = 0; i < 14; i++)
            if (!strcmp(command, ctable[i]))
                break;

        // 根据命令执行对应操作
        if (i == 0 || i == 1) // 创建或删除文件/目录
        {
            scanf("%s", var1); // 输入类型（f: 文件, d: 目录）
            scanf("%s", var2); // 输入文件/目录名称
            if (var1[0] == 'f')
                j = 1; // 文件
```

```
            else if (var1[0] == 'd')
                j = 2; // 目录
            else
            {
                printf("错误: 第一个参数必须是 [f/d]\n");
                continue;
            }

            if (i == 0) // 创建操作
            {
                if (Create(j, &currentdir, var2) == 1)
                    printf("失败: 无法创建 %s\n", var2);
                else
                    printf("成功: 创建了 %s\n", var2);
            }
            else // 删除操作
            {
                if (Delete(j, &currentdir, var2) == 1)
                    printf("失败: 无法删除 %s\n", var2);
                else
                    printf("成功: 删除了 %s\n", var2);
            }
        }
    else if (i == 2) // cd - Change Directory
        {
            scanf("%s", var2); // 输入目标目录
            i = 0;
            j = 0;
            temp = currentdir;
            while (1)
            {
                path[i] = var2[j]; // 构建路径
                if (path[i] == '/')
                {
                    if (j == 0)
                        initialize(&currentdir); // 如果是根目录，初始化
                    else if (i == 0) // 目录名中不能包含 '/'
                    {
                        printf("路径错误!\n");
                        break;
                    }
                    else // 进入指定目录
                    {
                        path[i] = '\0'; // 临时存储目录路径
```

```
                if (Open(&currentdir, path) == 1)
                {
                    printf("路径不对!\n");
                    currentdir = temp;
                }
            }
            i = 0; // 重新设置路径
        }
        else if (path[i] == '\0') // 路径结束
        {
            if (i == 0)
                break;
            if (Open(&currentdir, path) == 1)
            {
                printf("路径错误!\n");
                currentdir = temp;
            }
            break;
        }
        else
            i++; // 增加路径字符索引
        j++; // 增加用户输入字符索引
    }
}
else if (i == 3) // 关闭文件
{
    scanf("%d", &i); // 输入要关闭的文件数量
    for (j = 0; j < i; j++)
        if (Close(&currentdir) == 1)
        {
            printf("警告: 数量 %d 超过了打开的文件数\n", i);
            break;
        }
}
else if (i == 4) // 读取文件
{
    scanf("%s", var2); // 输入文件名
    if (Read(&currentdir, var2) == 1)
        printf("失败: 无法读取文件 %s\n", var2);
}
else if (i == 5) // 写入文件
{
    printf("请输入要写入的数据，ESC 结束\n");
    scanf("%s", var2); // 输入文件名
```

```
            if (Write(&currentdir, var2) == 1)
                printf("失败: 无法写入文件 %s\n", var2);
        }
        else if (i == 6) // 修改密码
            Password();
        else if (i == 7) //格式化
        {
            while (1)
            {
                printf("Do you want to format the filesystem?\n It will be dangerous
to your data.\n");
                printf("[Y/N]");
                scanf("%s", var1);
                if (var1[0] == 'N' || var1[0] == 'n')
                    break;
                else if (var1[0] == 'Y' || var1[0] == 'y')
                {
                    format(&currentdir);
                    break;
                }
                else
                    printf("please input [Y/N]");
            }
        }
       else if (i == 8) // exit - 退出文件系统
        {
            while (1)
        {
        printf("你确定要退出文件系统吗? [Y/N]\n");
        scanf("%s", var2); // 读取用户输入
        if (var2[0] == 'N' || var2[0] == 'n') // 用户选择不退出
            break;
        else if (var2[0] == 'Y' || var2[0] == 'y') // 用户选择退出
            return; // 退出文件系统
        else
            printf("\n 请输入 [Y/N]\n"); // 提示用户重新输入
         }
        }
        else if (i == 9) // 登录
            printf("错误: 您已经登录\n"); // 提示用户已登录
        else if (i == 10) // logout - 从文件系统退出
        {
            while (i)
            {
```

```
printf("你确定要从文件系统退出吗? [Y/N]"); // 提示用户是否退出
scanf("%s", var1); // 读取用户输入
if (var1[0] == 'N' || var1[0] == 'n') // 用户选择不退出
    break;
else if (var1[0] == 'Y' || var1[0] == 'y') // 用户选择退出
{
    initialize(&currentdir); // 重新初始化文件系统
    while (1)
    {
        printf("请输入命令: ");
        scanf("%s", var2); // 读取用户输入的命令
        if (strcmp(var2, "login") == 0) // 如果输入的是 "login" 命令
        {
            if (login() == 0) // 调用登录函数，如果登录成功
            {
                i = 0; // 设置 i 为 0，退出外部循环
                break;
            }
        }
        else if (strcmp(var2, "exit") == 0) // 如果输入的是 "exit" 命令
            return; // 退出程序
    }
}
else
    printf("请输入 [Y/N]"); // 提示用户重新输入
    }
}
else if (i == 11) // ls - 列出目录内容
    ls(&currentdir); // 调用 ls 函数显示目录内容
else if (i == 12) // pwd - 打印工作目录
{
    char string[100];
    for (int j = 0; j < 100; j++)
        string[j] = 0; // 初始化字符串
    getstring(currentstring, currentdir); // 获取当前目录的目录名
    pwd(string, &currentdir); // 获取绝对路径
    strcat(string, currentstring); // 拼接当前目录名
    int i = 0;
    for (i = 0; string[i + 1]; i++)
        string[i] = string[i + 1]; // 去掉多余的 '/'
    string[i] = 0; // 添加字符串结尾标志
    printf("%s\n", string); // 输出绝对路径
}
else if (i == 13) // 帮助信息
```

```
        {

printf("*************************************************************************
*************\n");
            printf("*                                        模拟 ext2 文件系统
*\n");
            printf("*        支        持        的        命        令        :
*\n");
            printf("*01.切换目录  :cd+目录名          02.创建目录   :create d+
目录名                *\n");
            printf("*03.创建文件  :create f+文件名    04.删除目录   :delete d+
目录名                *\n");
            printf("* 05.删除文件  : delete f+文件名    06.读取文件   : read+文
件名                   *\n");
            printf("* 07.写入文件  : write+文件名       08.显示路径   : pwd
*\n");
            printf("*09.关闭文件  :close+数量         10.修改密码   :password
*\n");
            printf("* 11.列出项目  : ls                 12.帮助菜单   : help
*\n");
            printf("* 13.格式化磁盘: format             14.退出系统   : exit
*\n");
            printf("*     15.    注    销    系    统              :     logout
*\n");

printf("*************************************************************************
*************\n");
        }
        else
        {
            printf("错误: 无效命令，请输入 help 查看支持的命令。\n");
        }
    }
}

/*main 函数，简化版的文件系统程序的主函数。*/
int main()
{
    ext2_inode cu; /* 当前用户的 inode 结构，表示用户所在的目录或文件系统状态 */

    // 输出欢迎信息，提示用户进入 Ext2 类似文件系统
    printf("你好呀!欢迎使用我的系统!\n");

    // 初始化文件系统，如果初始化失败则退出程序
```

```
    if (initfs(&cu) == 1)
        return 0;

    // 提示用户输入密码进行登录
    if (login() != 0) /* 登录失败时退出 */
    {
        // 密码错误时显示错误信息，并退出程序
        printf("密码不对 再见! \n");

        // 显示退出信息并结束程序
        exitdisplay();

        return 0;
    }

    // 登录成功后进入命令行交互模式
    shellloop(cu);

    // 退出文件系统并显示退出信息
    exitdisplay();

    // 正常结束程序
    return 0;
}
```