西安交通大学

操作系统专题实验报告

班级：计算机 2101

学号：2211410824

姓名：杜建宇

2023 年 12 月 15 日

目 录

1 openEuler 系统环境实验

1.1 实验目的

进程相关编程实验 ：

（1） 熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看

系统基本信息以了解系统；

（2）编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管

理等机制在操作系统实际运行中的作用。

线程相关编程实验 ：

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。

自旋锁实验 ：

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；

1.2 实验内容

进程相关编程实验 ：

（1） 熟悉操作命令、编辑、编译、运行程序。完成图 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；

线程相关编程实验 ：

（1） 在进程中给一变量赋初值并成功创建两个线程；

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

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

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

自旋锁实验 ：

（1） 在进程中给一变量赋初值并成功创建两个线程；

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

（ 3）使用自旋锁实现互斥和同步；

1.3 实验思想

进程相关编程实验 ：

（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。

线程相关编程实验 ：

本实验旨在通过创建两个线程，它们分别对一个共享的变量进行多次循环操

作，并观察在多次运行实验时可能出现的不同结果。在观察到结果不稳定的情况下，引入互斥和同步机制来确保线程间的正确协同操作。

（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。

自旋锁实验：

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

（ 1） 初始化锁： 自旋锁的开始是一个共享的标志变量（ flag），最初为未锁定状态（ 0）。这个标志变量用于表示资源是否已被其他线程占用。

（ 2） 获取锁： 当一个线程尝试获取锁时，它会循环检查标志变量的状态。如果发现标志变量是未锁定状态（ 0），那么该线程将通过原子操作将标志变量设置为锁定状态（ 1），从而成功获取锁。如果标志变量已经是锁定状态，线程会一直在循环中等待，直到标志变量变为未锁定状态为止。

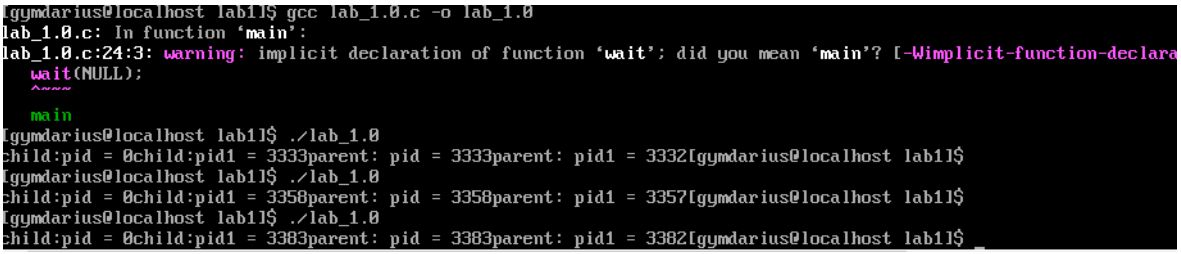
（ 3） 释放锁： 当持有锁的线程完成对共享资源的操作后，它会通过原子操作将标志变量设置回未锁定状态（ 0），从而释放锁，允许其他等待的线程尝试获取锁。

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。

1.4 实验步骤

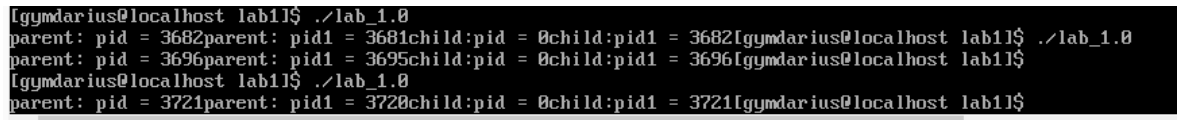
进程相关编程实验 ：

步骤一：编写并多次运行图 1-1 中代码，运行结果如下



对该过程的解释：fork()函数通过复制当前进程产生一个子进程，在父进程中，`fork()` 返回子进程的进程 ID，而在子进程中，它返回 0。getpid()函数返回当前进程的 pid_t 值。父进程的 pid 值为子进程的 pid 值，pid1 值为父进程的 pid 值，父进程的 pid 值比子进程的 pid 值要小 1.子进程的 pid 值为 0，pid1 值为自己的 pid 值。

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

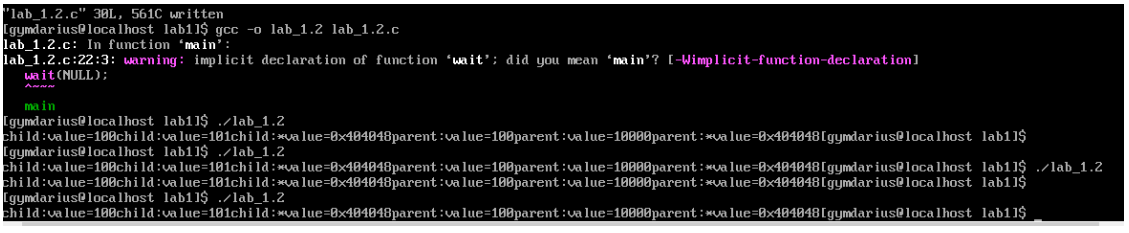


该运行结果中 parent 的输出早于 child 的输出，正好与不删去 wait()函数时相反。

解释：wait(NULL)函数会使父进程阻塞，直到它的一个子进程结束为止，一旦子进程终止，父进程将不再阻塞。在删去 wait()之前，父进程运行到 wait()时阻塞，等到子结束、输出之后，父进程才可以结束，输出。因此 child 输出提前于 parent。而删去 wait()后，则恰好相反，这里需要考虑 printf()的特性：缓冲区。

步骤三：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。

运行结果为：



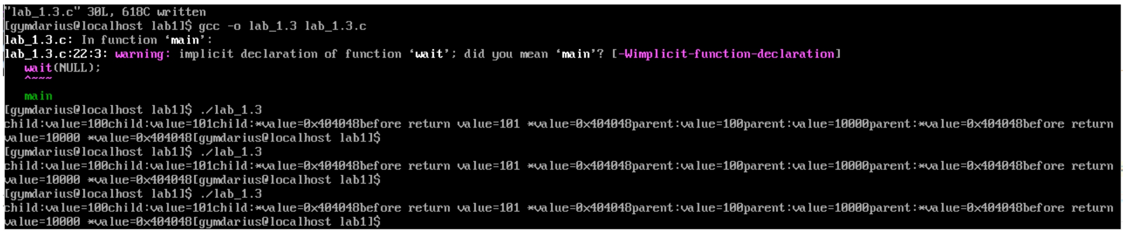
操作：定义一个全局变量，在子进程和父进程中进行不同的操作，并且输出其地址。

可以看到子进程和父进程并不共享全局变量，但是由于 COW（写时复制）机制，子进程和

父进程中的全局变量仍然指向相同的物理内存页。

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。

输出结果为：



操作：在 return 前加上了一句 printf()函数，用来输出 value 的值和地址。

子进程先结束先输出，父进程由于阻塞，后结束后输出。进一步说明了父进程和子进程并不共享全局变量。

步骤五：修改图中程序，在子进程中调用 system()与 exec 族函数。

system()

运行结果：

首先输出父进程的 PID，然后父进程中由于 wait(NULL)阻塞、等待子进程结束，子进程中输出子进程 PID，然后用 system()函数调用 system_call 函数，输出当前进程 system_call 进程 PID。由于 system() 函数不会替换当前进程，而是在当前进程的上下文中启动一个新的 shell，并在该 shell 中执行指定的命令。执行完命令后，shell 进程结束，控制返回到原始进程。**所以 system()执行完后还会执行输出子进程 PID 的代码**，而 exec 函数族会完全替换当前进程的代码和数据，而不会启动一个新的 shell。它会从新程序的 main 函数开始执行，当前进程的代码和数据将不再存在。

exec 族函数

在这个过程中，可以通过进程号、进程数来判断调用的是 system()/exec()族函数

运行结果为：

```
[root@kp-test01 lab1]# gcc -o lab_1.4 lab_1.4.c
[root@kp-test01 lab1]# ./lab_1.4
parent process PID: 9902
child process1 PID: 9903
system_call PID: 9903
[root@kp-test01 lab1]# ./lab_1.4
parent process PID: 9905
child process1 PID: 9906
system_call PID: 9906
[root@kp-test01 lab1]# ./lab_1.4
parent process PID: 9907
child process1 PID: 9908
system_call PID: 9908
```

线程相关编程实验 ：

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进行多次操作，观察输出结果。代码中定义共享变量初始值为 0，两线程分别对其进行 100000 次 +/-操作，最终在主进程中输出处理后的变量值。

运行结果为：

```
[root@kp-test01 lab2]# gcc -o lab_2.0 lab_2.0.c -lpthread
[root@kp-test01 lab2]# ./lab_2.0
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 3738
[root@kp-test01 lab2]# ./lab_2.0
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: -1902
[root@kp-test01 lab2]# ./lab_2.0
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 4733
[root@kp-test01 lab2]# ./lab_2.0
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: -132
```

没有使用互斥锁或其它同步机制。这将导致两个线程同时访问和修改 shared_variable，由于竞争条件，输出结果可能是不确定的，并且在每次运行时都可能不同。

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。

定义了一个信号量，初值赋为 0，在进程 1 的循环增值操作后进行 V 操作，在进程 2 的循环减值操作前进行 P 操作，确保同一时间内只有一个线程在工作。所以最终结果输出为 0。

运行结果为：

```
[root@kp-test01 lab2]# gcc -o lab_2.1 lab_2.1.c -pthread
[root@kp-test01 lab2]# ./lab_2.1
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 0
[root@kp-test01 lab2]# ./lab_2.1
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 0
[root@kp-test01 lab2]# ./lab_2.1
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 0
[root@kp-test01 lab2]# ./lab_2.1
Thread 1 created successfully.
Thread 2 created successfully.
Final shared_variable value: 0
```

步骤三：在第一部分实验了解了 system()与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

在主函数中新建两个线程，两线程分别输出 tid 和 pid 并且调用 system()或 exec 族函数来调用./system_call

调用 system()

运行结果为：

```
[root@kp-test01 lab2]# gcc -o lab_2.2 lab_2.2.c -pthread
[root@kp-test01 lab2]# ./lab_2.2
thread1 create success!
thread1 tid = 10158, pid = 10157
thread2 create success!
thread2 tid = 10159, pid = 10157
system_call PID: 10160
thread1 systemcall return
system_call PID: 10161
thread2 systemcall return
[root@kp-test01 lab2]# ./lab_2.2
thread1 create success!
thread1 tid = 10163, pid = 10162
thread2 create success!
thread2 tid = 10164, pid = 10162
system_call PID: 10165
system_call PID: 10166
thread1 systemcall return
thread2 systemcall return
```

进程和线程中调用 exec 族函数的区别：当调用 exec 族函数时，进程会完全替代其执行内容，

而线程只会替代当前线程的执行内容。这意味着在多线程应用程序中，一个线程调用 exec 函数不会影响其他线程，而在多进程应用程序中，一个进程调用 exec 函数会替代整个进程。

输出结果为：

```
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11149, pid = 11148
system_call PID: 11148
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11154, pid = 11153
system_call PID: 11153
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11157, pid = 11156
system_call PID: 11156
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11160, pid = 11159
system_call PID: 11159
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11163, pid = 11162
system_call PID: 11162
[root@kp-test01 lab2]#
```

调整代码结构后，输出结果为：

```
[root@kp-test01 lab2]# gcc -o lab_2.3 lab_2.3.c -pthread
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11175, pid = 11174
system_call PID: 11174
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11178, pid = 11177
system_call PID: 11177
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11182, pid = 11181
thread2 create success!
thread2 tid = 11183, pid = 11181
system_call PID: 11181
[root@kp-test01 lab2]# ./lab_2.3
thread1 create success!
thread1 tid = 11185, pid = 11184
thread2 create success!
thread2 tid = 11186, pid = 11184
system_call PID: 11184
```

自旋锁实验 ：

共享变量的输出结果为 10000，说明我们已经使用自旋锁来实现线程间的同步 。

输出结果为：

```
[root@kp-test01 lab3]# gcc -o lab_3.0 lab_3.0.c -pthread
[root@kp-test01 lab3]# ./lab_3.0
shared_value = 0
Thread 1 created successfully.
Thread 2 created successfully.
shared_value = 10000
[root@kp-test01 lab3]# ./lab_3.0
shared_value = 0
Thread 1 created successfully.
Thread 2 created successfully.
shared_value = 10000
[root@kp-test01 lab3]# ./lab_3.0
shared_value = 0
Thread 1 created successfully.
Thread 2 created successfully.
shared_value = 10000
```

1.5 实验总结

1.5.1 实验中的问题与解决过程

问题 1：在实验过程中，在 printf 输出的最后添加\n 导致进程调度输出不同。

运行结果为：

```
[gymdarius@localhost lab1]$ gcc lab_1.1.c -o lab_1.1
lab_1.1.c: In function 'main':
lab_1.1.c:24:3: warning: implicit declaration of function 'wait'; did you me
   wait(NULL);
   ^~~~
   main
[gymdarius@localhost lab1]$ ./lab_1.1
parent: pid = 3498
child:pid = 0
parent: pid1 = 3497
child:pid1 = 3498
[gymdarius@localhost lab1]$ ./lab_1.1
parent: pid = 3512
child:pid = 0
parent: pid1 = 3511
child:pid1 = 3512
```

问题描述：

父子进程调度输出时，\n 添加与否导致进程调度输出不同。

解决过程：

printf 函数是一个行缓冲函数，先将内容写到缓冲区，满足一定条件后，才会将内容写入对应的文件或流中。满足条件如下：

1. 缓冲区填满。

2. 写入的字符中有‘\n’ '\r'。

3. 调用 fflush 或 stdout 手动刷新缓冲区。

4. 调用 scanf 等要从缓冲区中读取数据时，也会将缓冲区内的数据刷新。

5. 程序结束时。

不加\n 时，即使 parent 进程先执行输出函数，但由于该进程未结束，所以内容仍然保留在缓冲区中。而子进程由于先于父进程结束，所以子进程的输出先行输出到屏幕上，造成看上去是先调用子进程，后调用父进程的效果。

lab1.0 的代码输出结果（没有注释 wait(NULL),没有添加/n）

```
[gymdarius@localhost lab1]$ ./lab_1.0
child:pid = 0child:pid1 = 2909parent: pid = 2909parent: pid1 = 2908[gymdarius@localhost lab1]$ onmouse
-bash: onmouse: command not found
```

而加入\n 之后， \n 强制刷新，每 printf 输出一次就显示在屏幕上一次，这个顺序是真实的进程调度顺序

```
lab_1.0  lab_1.0.1  lab_1.0.c
[gymdarius@localhost lab1]$ ./lab_1.0.1
parent: pid = 3251
child:pid = 0
parent: pid1 = 3250
child:pid1 = 3251
[gymdarius@localhost lab1]$
```

为了更好地进行实验探索，后面的实验过程中均不添加/n。

问题 2：gcc 编译时没有链接成功，报错 undefined reference topthread_create'`，因此编译时使用 -pthread 标志来链接 pthread 库

1.5.2 实验收获

在进程实验中，我熟悉了 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了解了系统；编写并运行简单的进程调度相关程序，体会了进程调度、进程间变量的管理等机制在操作系统实际运行中的作用。 在线程实验中，我学习了多线程编程中的线程共享进程信息的原理，深入了解了线程共享资源时可能出现的问题。在自旋锁实验中，我了解了自旋锁的基本概念，并通过研究自旋锁的工作原理和特点，深入理解了自旋锁相对于其他锁机制的优势和局限性；我实现自旋锁的设计与同步，并使用自

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；

1.5.3 意见与建议

指导书可以向同学们提出一些建设性的建议，以便于同学们更深入的进行实验。如果仅仅凭着一个人的力量去进行实验，那么结果难免有局限性。

可以以小组的形式有一个大型实验，加强同学们的小组合作能力。

1.6 附件

1.6.1 附件 1　程序

进程相关编程实验

1)图中程序不删除 wait()

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

int main(){
    pid_t pid,pid1;

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

2)图中程序删除 wait()

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

```
4.
5.  int main(){
6.      pid_t pid,pid1;
7.
8.      pid=fork();
9.      if(pid<0){
10.         fprintf(stderr,"Fork Failed");
11.         return 1;
12.     }
13.     else if(pid==0){
14.         pid1=getpid();
15.         printf("child:pid= %d\n",pid);
16.         printf("child:pid1= %d\n",pid1);
17.     }
18.     else{
19.         pid1=getpid();
20.         printf("parent:pid= %d\n",pid);
21.         printf("parent:pid1= %d\n",pid1);
22.     }
23.     return 0;
24. }
```

3)添加全局变量

```
1.  #include<sys/types.h>
2.  #include<stdio.h>
3.  #include<unistd.h>
4.  int value=100;
5.  int main(){
6.      pid_t pid,pid1;
7.
8.      pid=fork();
9.      if(pid<0){
10.         fprintf(stderr,"Fork Failed");
11.         return 1;
12.     }
13.     else if(pid==0){
14.         printf("child:value=%d",value);
15.         value+=1;
16.         printf("child:value=%d",value);
17.         printf("child:*value=%p",&value);
18.     }
19.     else{
20.         wait(NULL);
21.         printf("parent:value=%d",value);
22.         value*=value;
23.         printf("parent:value=%d",value);
24.         printf("parent:*value=%p",&value);
25.     }
26.     return 0;
27. }
```

4)在 return 前增加对全局变量的操作

```
1.  #include<sys/types.h>
2.  #include<stdio.h>
```

```c
#include<unistd.h>
int value=100;
int main(){
    pid_t pid,pid1;

    pid=fork();
    if(pid<0){
        fprintf(stderr,"Fork Failed");
        return 1;
    }
    else if(pid==0){
        printf("child:value=%d",value);
        value+=1;
        printf("child:value=%d",value);
        printf("child:*value=%p",&value);
    }
    else{
        wait(NULL);
        printf("parent:value=%d",value);
        value*=value;
        printf("parent:value=%d",value);
        printf("parent:*value=%p",&value);
    }
    printf("before return value=%d *value=%p",value,&value);
    return 0;
}
```

5)在子进程中调用 exec 族函数

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        return 1;
    } else if (pid == 0) {
        pid1 = getpid();
        printf("child process1 PID: %d\n", pid1);
        char *args[] = {"./system_call", NULL};
        execv(args[0], args);
        perror("execv");
        exit(1);
        printf("child process PID: %d\n", pid1);
    } else {
        pid1 = getpid();
        printf("parent process PID: %d\n", pid1);
        wait(NULL);
    }
    return 0;
```

```
}
```

6)在子进程中调用 system()函数

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    pid_t pid, pid1;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        return 1;
    } else if (pid == 0) {
        pid1 = getpid();
        printf("child process1 PID: %d\n", pid1);
        fflush(stdout);
        system("./system_call");
        printf("child process PID: %d\n", pid1);
    } else {
        pid1 = getpid();
        printf("parent process PID: %d\n", pid1);
        wait(NULL);
    }
    return 0;
}
```

7)创建两个子线程， 两线程分别对同一个共享变量多次操作

```
#include <pthread.h>
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int shared_variable = 0;
void *thread_function1(void *thread_id) {
    int i;
    for (i = 0; i < 100000; i++) {
        shared_variable++;
    }
    pthread_exit(NULL);
}
void *thread_function2(void *thread_id) {
    int i;
    for (i = 0; i < 100000; i++) {
        shared_variable--;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;
    if (pthread_create(&thread1, NULL, thread_function1, (void *)1) == 0) {
        printf("Thread 1 created successfully.\n");
```

```
    } else {
        printf("Thread 1 creation failed.\n");
    }

    if (pthread_create(&thread2, NULL, thread_function2, (void *)2) == 0) {
        printf("Thread 2 created successfully.\n");
    } else {
        printf("Thread 2 creation failed.\n");
    }
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final shared_variable value: %d\n", shared_variable);
    return 0;
}
```

8)定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。

```
#include <pthread.h>
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
#include <semaphore.h>
int shared_variable = 0;
sem_t sem2;
void *thread_function1(void *thread_id) {
    int i;
    for (i = 0; i < 100000; i++) {
        shared_variable++;
    }
    sem_post(&sem2);
    pthread_exit(NULL);
}
void *thread_function2(void *thread_id) {
    int i;
    sem_wait(&sem2);
    for (i = 0; i < 100000; i++) {
        shared_variable--;
    }
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    sem_init(&sem2, 0, 0);
    if (pthread_create(&thread1, NULL, thread_function1, (void *)1) == 0) {
        printf("Thread 1 created successfully.\n");
    } else {
        printf("Thread 1 creation failed.\n");
    }
    if (pthread_create(&thread2, NULL, thread_function2, (void *)2) == 0) {
        printf("Thread 2 created successfully.\n");
    } else {
        printf("Thread 2 creation failed.\n");
    }
    pthread_join(thread1, NULL);
```

```
    pthread_join(thread2, NULL);
    sem_destroy(&sem2);
    printf("Final shared_variable value: %d\n", shared_variable);
    return 0;
}
```

9)将 system()函数的调用改为在线程中实现

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/syscall.h>
void *thread_function1(void *arg) {
    pid_t pid = getpid();
    pid_t tid = syscall(SYS_gettid);
    printf("thread1 create success!\n");
    printf("thread1 tid = %d, pid = %d\n", tid, pid);
    system("./system_call");
    printf("thread1 systemcall return\n");
    pthread_exit(NULL);
}
void *thread_function2(void *arg) {
    pid_t pid = getpid();
    pid_t tid = syscall(SYS_gettid);
    printf("thread2 create success!\n");
    printf("thread2 tid = %d, pid = %d\n", tid, pid);
    system("./system_call");
    printf("thread2 systemcall return\n");
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    if (pthread_create(&thread1, NULL, thread_function1, NULL) == 0) {
    } else {
        printf("Thread 1 creation failed.\n");
    }
    if (pthread_create(&thread2, NULL, thread_function2, NULL) == 0) {
    } else {
        printf("Thread 2 creation failed.\n");
    }
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

10)将 exec 族函数的调用改为在线程中实现

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

```
void *thread_function1(void *arg) {
    pid_t pid = getpid();
    pid_t tid = syscall(SYS_gettid);
    printf("thread1 create success!\n");
    printf("thread1 tid = %d, pid = %d\n", tid, pid);
    char *args[] = {"./system_call", NULL};
    execv(args[0], args);
    perror("execv");
    exit(1);
    printf("thread1 systemcall return\n");
    pthread_exit(NULL);
}
void *thread_function2(void *arg) {
    pid_t pid = getpid();
    pid_t tid = syscall(SYS_gettid);
    printf("thread2 create success!\n");
    printf("thread2 tid = %d, pid = %d\n", tid, pid);
    char *args[] = {"./system_call", NULL};
    execv(args[0], args);
    exit(1);
    printf("thread2 systemcall return\n");
    pthread_exit(NULL);
}
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread_function1, NULL);
    pthread_create(&thread2, NULL, thread_function2, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

11)自旋锁实验

```
include <stdio.h>
#include <pthread.h>
typedef struct {
        int flag;
    } spinlock_t;
void spinlock_init(spinlock_t *lock) {
    lock->flag = 0;
}
void spinlock_lock(spinlock_t *lock) {
    while (__sync_lock_test_and_set(&lock->flag, 1)) {
    }
}
void spinlock_unlock(spinlock_t *lock) {
    __sync_lock_release(&lock->flag);
}
int shared_value = 0;
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;
    printf("shared_value = %d\n",shared_value);
    spinlock_init(&lock);
     if (pthread_create(&thread1, NULL, thread_function, &lock) == 0) {
        printf("Thread 1 created successfully.\n");
    } else {
        printf("Thread 1 creation failed.\n");
    }

    if (pthread_create(&thread2, NULL, thread_function, &lock) == 0) {
        printf("Thread 2 created successfully.\n");
    } else {
        printf("Thread 2 creation failed.\n");
    }
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("shared_value = %d\n", shared_value);
    return 0;
}
```

1.6.2 附件 2 Readme

见第一次实验文件夹下 readme.md 文档

2 进程通信与内存管理

2.1 实验目的

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

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

内存的分配和回收：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。

2.2 实验内容

进程的软中断通信：

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

（ 2）根据流程图（如图 2.1 所示） 编制实现软中断通信的程序： 使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号（即 5s 内按下 delete 键或 quit 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 kill()向两个子进程分别发出整数值为 16 和 17 软中断信号，子进程获得对应软中断信号，然后分别输出下列信息后终止：

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

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

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

参考资料 *https://blog.csdn.net/mylizh/article/details/38385739*

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

进程的管道通信：

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

（ 2）根据流程图（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 。

内存的分配和回收：

（ 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 实验思想

进程的软中断通信：

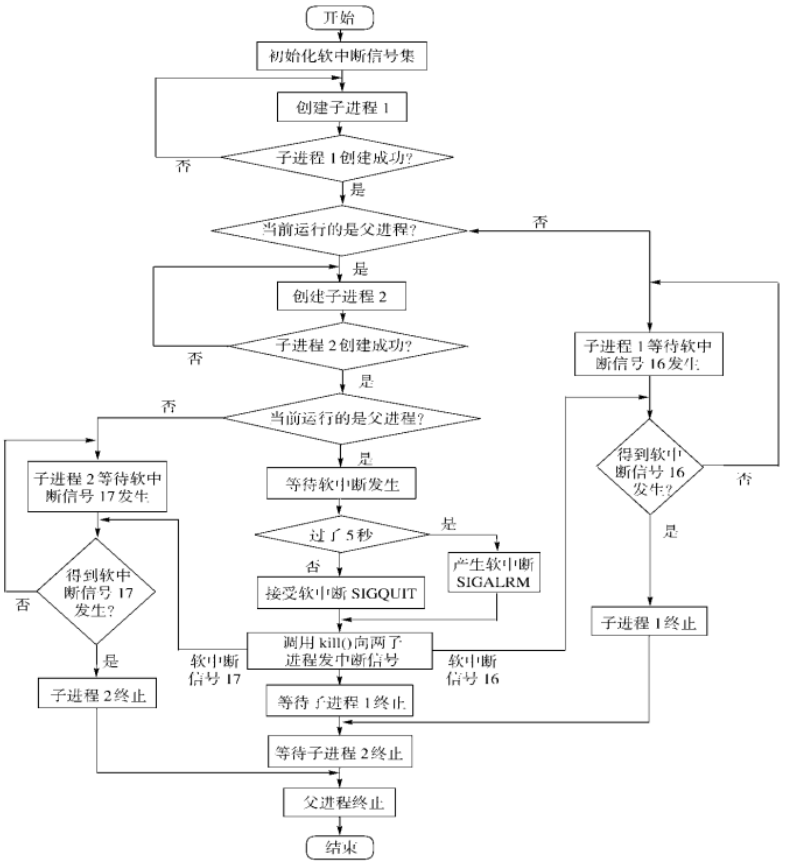


图 2.1 软中断通信程序流程图

根据实验要求编写程序， 其中，中断处理程序实现了输出中断号的功能。

父进程创建两个子进程，两个子进程中分别处理 16、 17 号中断，并在父进程中分别杀死两个进程，并传递对应进程号到两个子进程

进程的管道通信:

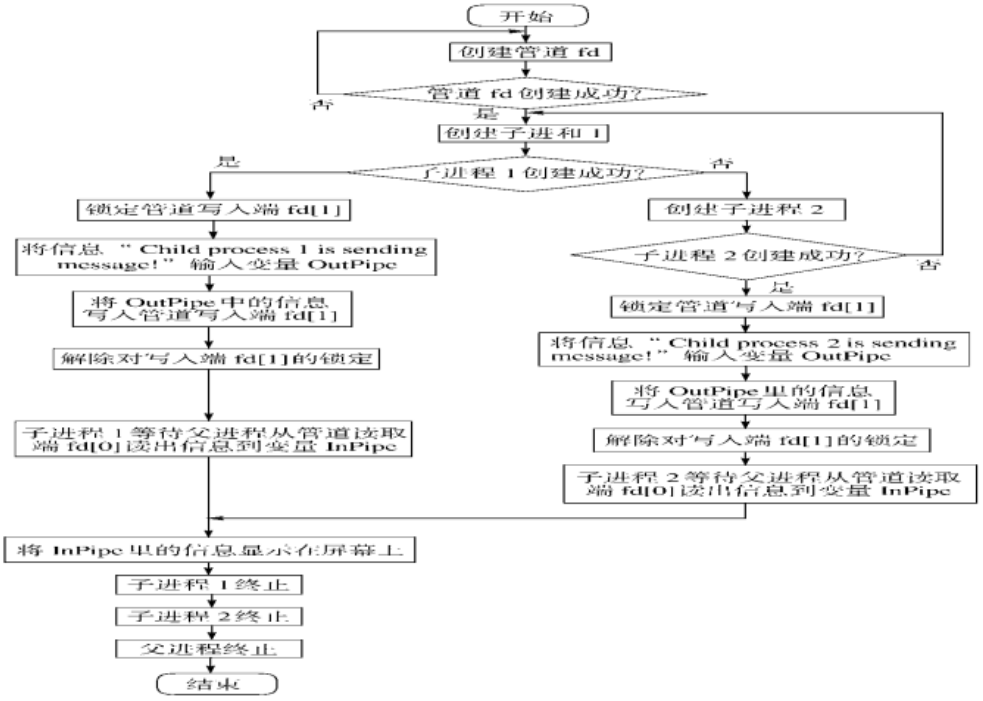


图 2.2 管道通信程序流程图

要实现这个功能，首先需要学习进程间管道通信。进程通过调用 pipe 函数，在内核中开辟一块缓冲区用来进行进程间通信，这块缓冲区称为管道，它有一个读端和一个写端。 pipe 函数接受一个参数，是包含两个整数的数组，如果调用成功，会通过 pipefd[2]传出给用户程序两个文件描述符，需要注意 pipefd [0]指向管道的读端, pipefd [1]指向管道的写端，那么此时这个管道对于用户程序就是一个文件，可以通过 read(pipefd [0])；或者 write(pipefd [1])进行操作， pipe 函数调用成功返回 0，否则返回-1.

内存的分配和回收:

内存管理中实现了三种内存管理算法（FF， BF， WF），具体原理见实验代码注释.代码设计：将程序分为显示信息、初始化、进程创建、进程销毁、退出等几个模块，并分别实现代码

2.4 实验结果

进程的软中断通信

```
[root@kp-test01 LAB2]# ./lab2_1
^C
2 stop test

16 stop test

Child process 1 is killed by parent!

17 stop test

Child process 2 is killed by parent!

Parent process is killed!
```

```
[root@kp-test01 LAB2]# ./lab2_1

16 stop test

Child process 1 is killed by parent!

17 stop test

Child process 2 is killed by parent!

Parent process is killed!
```

```
[root@kp-test01 LAB2]# ./lab2_11
after 4 seconds kill the first

14 stop test
after 6 seconds kill the second

16 stop test

Child process 1 is killed by parent!

14 stop test

17 stop test

Child process 2 is killed by parent!

Parent process is killed!
```

进程的管道通信

内存的分配和回收

```
[root@kp-test01 LAB2]# ./lab2_33

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

Used Memory:
       PID          ProcessName start_addr       size
----------------------------------------------------------
```

2.5 回答问题

2.5.1 软中断通信

1、最初认为的运行结果

我最初认为运行结果是 如果 5s 内从键盘手动输出`Ctrl+C`发出中断信号，那么应当首先输出截获 SIGINT 信号对应的输出。接下来两个子进程的输出顺序是随机的，但是要求每个子进程需要满足，先输出捕获信号后执行函数对应的输出，然后打印输出子进程被父进程杀死的语句，最后输出父进程被回收的对应语句。拓扑结构即为 2 stop test 在最前，17 stop test 在 Child process 2 is killed 前，16 stop test 在 Child process 1 is killed 前，parent process is killed 在最后，其余顺序可以随意出现。

2 实际的运行结果

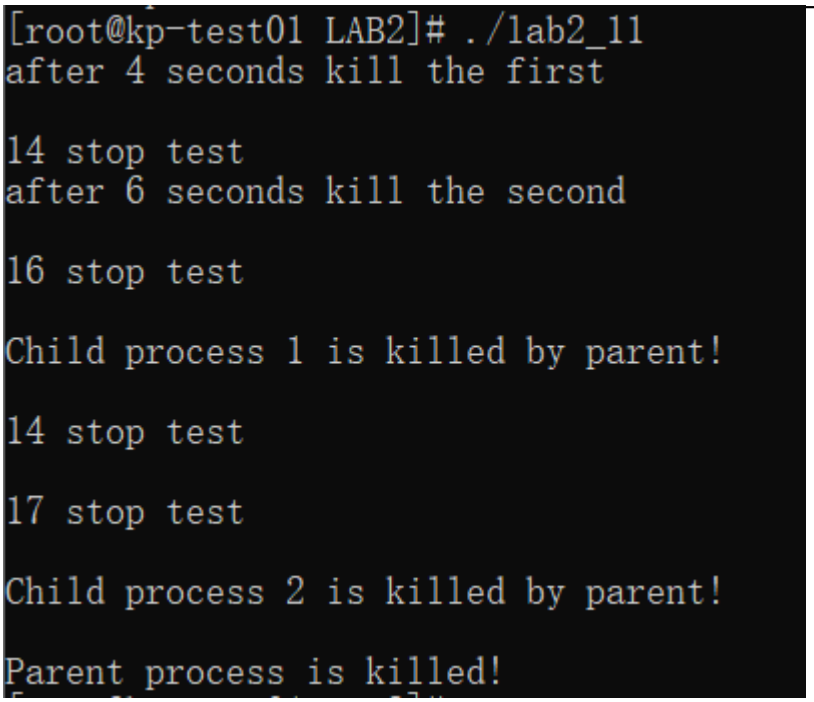
5s 内中断

```
[root@kp-test01 LAB2]# ./lab2_1
^C
2 stop test

16 stop test

Child process 1 is killed by parent!

17 stop test

Child process 2 is killed by parent!

Parent process is killed!
```

5s 后中断

```
[root@kp-test01 LAB2]# ./lab2_1

16 stop test

Child process 1 is killed by parent!

17 stop test

Child process 2 is killed by parent!

Parent process is killed!
```

3 闹钟中断



4kill 命令的分析

kill 命令在程序中使用了两次，作用是向子进程发送中断信号结束此进程，第一次执行后，子进程接受 16 信号并打印 16 stop test，并打印子进程 1 被杀死的信息，第二次执行同理，打印 17 stop test，并打印子进程 2 被杀死的信息。

5kill 命令拓展

进程可以通过 exit 函数来自主退出，进程自主退出的方式会更好一些，进程在退出时，`exit()` 函数会执行一系列清理操作，包括关闭文件、释放内存等。这有助于防止资源泄漏；exit()允许进程向父进程传递一个退出状态。这个退出状态可以被父进程获取，用于了解子进程的退出情况。`kill` 命令不能提供这种方式。使用 kill 命令在进程外部强制其退出可能导致进程的数据丢失或资源泄漏，但当进程出现异常无法退出时，可能需要 kill 命令来中止进程。

2.5.2 管道通信

1.你最初认为运行结果会怎么样？

加锁时的运行结果应该是先输出 1000 个 1 再输出 1000 个 2，与实际结果一样。不加锁时应该 12 交替输出。

2.实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

加锁运行状态

分析：有锁状态下，同一时间内只有一个进程可以写入，因而是先写入了 2000 个 1 而后写入了 2000 个 2

不加锁运行状态

分析：无锁状态下，两进程对 InPipe 的写入在时序上是随机的，因而打印结果也是 1 与 2 交

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？

同步与互斥的实现：在对管道写入前，先对管道用 lockf 函数加锁，等管道写入完毕后，在将管道解锁。

不控制的后果：由于多个进程同时向管道中写入数据，那么数据就很容易发生交错和覆盖，导致数据错误。错打印。

2.5.3 内存的分配与回收

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。

First Fit (FF):

思想：分配时从内存的起始位置开始找到第一个足够大的空闲块进行分配。

优点：简单，实现容易。

缺点：可能会导致大块的内存碎片。

Best Fit (BF):

思想：分配时找到所有足够大的空闲块中最小的一个进行分配。

优点：尽量减小内存碎片。

缺点：查找最小空闲块可能比较耗时。

Worst Fit (WF):

思想：分配时找到所有足够大的空闲块中最大的一个进行分配。

优点：可以避免产生大量小碎片。

缺点：可能导致大块的内存碎片。

在实现上如何提高算法的查找性能：

使用更高效的数据结构，如平衡二叉树或哈希表，来存储空闲块信息，以提高查找效率。

改进排序算法

采取内存紧缩技术以提高内存利用率，减少空闲内存块的个数，从而缩短算法查找时间

2. 3 种算法的空闲块排序分别是如何实现的。

在代码中，空闲块的排序是通过三个函数实现的：

rearrange_FF()：对空闲块按照起始地址进行冒泡排序。

rearrange_BF()：对空闲块按照大小进行冒泡排序。

rearrange_WF()：对空闲块按照大小进行逆向冒泡排序。

这些函数在每次内存分配或释放后都被调用，以保持空闲块的有序性。

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。

内碎片：指已分配给进程的内存块中，没有被进程使用的部分。例如，如果一个进程请求分配 100 个字节，但系统只能提供 102 个字节的内存块，那么有 2 个字节就是内碎片。

外碎片：指分配给各进程的内存块之间存在的不可用的、无法分配的小块内存。例如，多次的进程分配和释放导致内存中存在很多不连续的小块，这些小块之间的未分配空间就是外碎片。

紧缩功能解决的碎片：当有大量外碎片时，紧缩功能可以将已分配的内存块整理并移动，以便合并这些碎片，形成更大的连续可用空间。这样可以提高内存的利用率，减少外碎片对系统的影响。

4.在回收内存时，空闲块合并是如何实现的？

在回收内存时，空闲块合并是通过在 `free_mem()` 函数中实现的。具体步骤如下：

1. 将被释放的内存块信息创建为一个新的空闲块节点 `head`。

2. 将这个新的空闲块节点的 `next` 指向当前的空闲块链表的头部（即 `free_block`)。

3. 更新当前的空闲块链表的头部为这个新的空闲块节点 `head`。

4. 对当前空闲块链表进行 rearrange_FF()即按照起始地址进行冒泡排序。

5. 对当前的空闲块链表进行遍历，检查是否有相邻的空闲块可以合并。

6. 如果找到相邻的空闲块，则合并它们，更新相应的信息。

7. 继续遍历直到整个空闲块链表。

这样，通过合并相邻的空闲块，可以尽量减小内存碎片，提高内存利用率。

2.6 实验总结

2.6.1 实验中的问题与解决过程

问题 1：

通过 man 命令查看 fork 等系统调用的手册时，显示”No manual entry for fork in section 1”,上网搜查以后发现是 man 命令手册没有更新导致的，重新下载最新的 man 命令手册以后成功查看。

问题 2：

一开始闹钟中断不知道如何设计，通过与同学交流以后成功解决

问题 3：

allocate_mem 和 free_mem 函数的实现遇到困难，通过上网查阅资料，理解思路以后，编写完成

2.6.2 实验收获

通过这次实验，我对于管道通信有了一定程度的掌握，并掌握了用管道实现不同进程间通信的方法。此外，我还掌握了内存分配 FF， BF， WF 策略，以及页面替换 OPT、 LRU、 FIFO、 NRU 等算法

2.6.3 意见与建议

对进程的管道通信可以设计更多的内容

2.7 附件

2.7.1 附件 1 程序

1）进程软中断通信

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


void inter_handler(int sign)
{
    switch(sign)
    {
        case SIGINT:
            printf("\n2 stop test\n");
            break;
        case SIGSTKFLT:
            printf("\n16 stop test\n");
            break;
        case SIGCHLD:
            printf("\n17 stop test\n");
            break;
        default:;
    }
}

int main()
{
    pid_t pid1, pid2;
    while((pid1 = fork()) == -1);
    if(pid1 > 0)
    {
        while((pid2 = fork()) == -1);
        if(pid2 > 0)
        {
            signal(2, inter_handler);
            sleep(5);
            kill(pid1, SIGSTKFLT); //kill the first child
            kill(pid2, SIGCHLD); //kill the second child
            wait(NULL);   //wait the first child's finish
            wait(NULL);   //wait the second child's finish
            printf("\nParent process is killed! \n");
            exit(0);
        }
        else
        {
```

```
            signal(2, SIG_IGN); //用于忽略 SIGINT 信号
            signal(17, inter_handler); //the second child is waiting
            pause();    //暂停等待
            printf("\nChild process 2 is killed by parent! \n");
            exit(0);
        }
    }
    else
    {
        signal(2, SIG_IGN); //用于忽略 SIGINT 信号
        signal(16, inter_handler); //the first child is waiting
        pause();    //暂停等待
        printf("\nChild process 1 is killed by parent! \n");
        exit(0);
    }
    return 0;
}
```

2) 进程软中断通信(闹钟中断)

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

void inter_handler(int sign)
{
    switch(sign)
    {
        case SIGINT:
            printf("\n2 stop test\n");
            break;
        case SIGSTKFLT:
            printf("\n16 stop test\n");
            break;
        case SIGCHLD:
            printf("\n17 stop test\n");
            break;
        case SIGALRM:
            printf("\n14 stop test\n");
            break;
        default:;
    }
}

int main()
{
    pid_t pid1, pid2;
    while((pid1 = fork()) == -1);
    if(pid1 > 0)
    {
        while((pid2 = fork()) == -1);
        if(pid2 > 0)
```

```
        {
            signal(14, inter_handler);
            printf("after 4 seconds kill the first\n");
            alarm(4);
            sleep(8);       //sleep 会被 alarm 函数发出的信号中断
            kill(pid1, SIGSTKFLT); //kill the first child
            printf("after 6 seconds kill the second\n");
            alarm(6);
            sleep(8);
            kill(pid2, SIGCHLD); //kill the second child
            wait(NULL);   //wait the first child's finish
            wait(NULL);   //wait the second child's finish
            printf("\nParent process is killed! \n");
            exit(0);
        }
        else
        {
            signal(17, inter_handler); //the second child is waiting
            pause();
            printf("\nChild process 2 is killed by parent! \n");
            exit(0);
        }
    }
    else
    {
        signal(16, inter_handler); //the first child is waiting
        pause();
        printf("\nChild process 1 is killed by parent! \n");
        exit(0);
    }
    return 0;
}
```

3) 进程的管道通信

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

int pid1,pid2;     // 定义两个进程变量

//define read/write end code
#define READEND 0
#define WRITEEND 1

int main(int argc, char* argv[])
{
    int fd[2];
    char InPipe[1000];                  // 定义读缓冲区(buffer)
    char c1='1', c2='2';
    pipe(fd);                           // 创建管道
    while((pid1 = fork( )) == -1);      // 如果进程 1 创建不成功,则空循环
```

```
19.
20.     if(pid1 == 0)                       // 子进程 1
21.     {
22.         //lockf(fd[WRITEEND],F_LOCK,0);   // 锁定管道
23.         int i;
24.         for(i=1;i<=2000;i++)             //  分 200 次每次向管道写入字符’1’
25.             write(fd[WRITEEND],&c1,sizeof(char));
26.         //sleep(5);                        // 等待读进程读出数据
27.         //lockf(fd[WRITEEND],F_ULOCK,0);  // 解除管道的锁定
28.         printf("pid1 over\n");
29.         exit(0);                         // 结束进程 1
30.     }
31.
32.     else
33.     {
34.         while((pid2 = fork()) == -1);       // 若进程 2 创建不成功,则空循环
35.         if(pid2 == 0)                       // 子进程 2
36.         {
37.             //lockf(fd[WRITEEND],F_LOCK,0);   // 锁定管道
38.             int i;
39.             for(i=1;i<=2000;i++)             //  分 200 次每次向管道写入字符’2’
40.                 write(fd[WRITEEND],&c2,sizeof(char));
41.             //sleep(5);                        // 等待读进程读出数据
42.             //lockf(fd[WRITEEND],F_ULOCK,0);  // 解除管道的锁定
43.             printf("pid2 over\n");
44.             exit(0);
45.         }
46.         else                                //父进程
47.         {
48.             waitpid(pid1,NULL,0);           // 等待子进程 1 结束
49.             waitpid(pid2,NULL,0);           // 等待子进程 2 结束
50.             printf("read\n");
51.             read(fd[READEND],InPipe,4000*sizeof(char));
52.
53.             printf("read over\n");
54.             InPipe[4000]='\0';               //  加字符串结束符
55.             printf("%s\n",InPipe);          // 显示读出的数据
56.             exit(0);                        // 父进程结束
57.         }
58.     }
59.     return 0;
60. }
```

4) 内存的分配与回收

```
1.  #include<stdio.h>
2.  #include<stdlib.h>
3.  #include<unistd.h>
4.  #define PROCESS_NAME_LEN 32
5.  #define MIN_SLICE 10
6.  #define DEFAULT_MEM_SIZE 1024
7.  #define DEFAULT_MEM_START 0
8.  #define MA_FF 1
9.  #define MA_BF 2
```

```
#define MA_WF 3
int mem_size=DEFAULT_MEM_SIZE;
int ma_algorithm = MA_FF;
static int pid = 0;
int flag = 0;
struct free_block_type{
    int size;
    int start_addr;
    struct free_block_type *next;
};
struct free_block_type *free_block;
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;
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 do_exit(){
    while(free_block->next!= NULL&&free_block !=NULL){
        struct free_block_type *temp = free_block;
        free_block = free_block->next;
        free(temp);
    }
    if(free_block)
        free(free_block);
}
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");
}
int set_mem_size(){
    int size;
    if(flag!=0){
    printf("Cannot set memory size again\n");
    return 0;
    }
```

```
64.     printf("Total memory size =");
65.     scanf("%d", &size);
66.     if(size>0) {
67.         mem_size = size;
68.         free_block->size = mem_size;
69.     }
70.     flag=1; return 1;
71. }
72. int rearrange_FF(){     //对起始地址进行冒泡排序
73.     struct free_block_type *head = free_block;
74.     struct free_block_type *pre;
75.      while(head->next != NULL){
76.         pre = head->next;
77.         while(pre != NULL){
78.             if(head->start_addr > pre->start_addr){
79.
80.                 int temp_size = head->size;
81.                 int temp_start = head->start_addr;
82.                 head->size = pre->size;
83.                 head->start_addr = pre->start_addr;
84.                 pre->size = temp_size;
85.                 pre->start_addr = temp_start;
86.             }
87.             pre = pre->next;
88.         }
89.         head = head->next;
90.      }
91.      return 1;
92.
93. }
94. int rearrange_BF(){ //对大小进行冒泡排序
95.      struct free_block_type *head = free_block;
96.
97.      struct free_block_type *pre;
98.      while(head->next != NULL){
99.          pre = head->next;
100.         while(pre != NULL){
101.             if(head->size > pre->size){
102.
103.                 int temp_size = head->size;
104.                 int temp_start = head->start_addr;
105.                 head->size = pre->size;
106.                 head->start_addr = pre->start_addr;
107.                 pre->size = temp_size;
108.                 pre->start_addr = temp_start;
109.             }
110.             pre = pre->next;
111.         }
112.         head = head->next;
113.     }
114.    return 1;
115. }
116. int rearrange_WF(){ //对大小进行逆向冒泡排序
```

```
    struct free_block_type *head = free_block;
    struct free_block_type *pre;
    while(head->next != NULL){
        pre = head->next;
        while(pre != NULL){
            if(head->size < pre->size){
                int temp_size = head->size;
                int temp_start = head->start_addr;
                head->size = pre->size;
                head->start_addr = pre->start_addr;
                pre->size = temp_size;
                pre->start_addr = temp_start;
            }
            pre = pre->next;
        }
        head = head->next;
    }
    return 1;
}
int rearrange(int algorithm){
    switch(algorithm){
    case MA_FF: return rearrange_FF(); break;
    case MA_BF: return rearrange_BF(); break;
    case MA_WF: return rearrange_WF(); break;
 }
}
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>=1 && algorithm <=3)
        ma_algorithm=algorithm;
    rearrange(ma_algorithm);
}
int allocate_mem(struct allocated_block *ab){//按照链表 free_block 进行遍历，如果第一轮不行的话。就重新分配后再次尝试分配
    struct free_block_type *head, *pre;
    int request_size=ab->size;
    head = pre = free_block;
    while(head != NULL){
        if(head->size >= request_size){
            if(head->size - request_size <= MIN_SLICE){
                ab->start_addr = head->start_addr;
                ab->size = head->size;
                if(head == free_block){
                    free_block = head->next;
                }
                else{
                     pre->next = head->next;
                }
                free(head);
        }
```

```
            else{
                ab->start_addr = head->start_addr;
                ab->size = request_size;
                head->start_addr += request_size;
                head->size -= request_size;
            }
        rearrange(ma_algorithm);
        return 1;
       }
    pre = head;
    head = head->next;
    }
    if(rearrange(ma_algorithm) == 1){
        pre = head = free_block;
        while(head != NULL){
            if(head->size >= request_size){
                if(head->size - request_size <= MIN_SLICE){
                    ab->start_addr = head->start_addr;
                    ab->size = head->size;
                    if(head == free_block){
                        free_block = head->next;
                    }else{
                        pre->next = head->next;
                    }
                    free(head);
                }else{
                    ab->start_addr = head->start_addr;
                    ab->size = request_size;
                    head->start_addr += request_size;
                    head->size -= request_size;
                }
                rearrange(ma_algorithm);
                return 1;
            }
            else if(head->size < MIN_SLICE) return -1;
            pre = head;
            head = head->next;
        }
    }
 return -1;
}
int new_process(){
    struct allocated_block *ab;
    int size; int ret;
    ab=(struct allocated_block *)malloc(sizeof(struct allocated_block));
    if(!ab) exit(-5);
    ab->next = NULL;
    pid++;
    sprintf(ab->process_name, "PROCESS-%02d", pid);
    ab->pid = pid;
    printf("Memory for %s:", ab->process_name);
    scanf("%d", &size);
    if(size>0) ab->size=size;
    ret = allocate_mem(ab);
```

```
    if((ret==1) &&(allocated_block_head == NULL)){
        allocated_block_head=ab;
        return 1; }
    else if (ret==1) {
        ab->next=allocated_block_head;
        allocated_block_head=ab;
        return 2; }
    else if(ret==-1){
        pid--;
        printf("Allocation fail\n");
        free(ab);
        return -1;
    }
    return 3;
}
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;
}
int free_mem(struct allocated_block *ab){
    int algorithm = ma_algorithm;
    struct free_block_type *head, *pre;
    head=(struct free_block_type*) malloc(sizeof(struct free_block_type));
    if(!head) return -1;    //分配失败
    head->size=ab->size;
    head->start_addr=ab->start_addr;
    head->next=free_block;
    free_block = head;
    rearrange_FF(); //按照起始地址进行冒泡排序
    pre = free_block;
    struct free_block_type *next = pre->next;
    while(next != NULL){
        if(pre->start_addr + pre->size != next->start_addr){
            pre = pre->next;
            next = next->next;
        }
        else{
            pre->size += next->size;
            struct free_block_type *temp = next;
            pre->next = next->next;
            free(temp);
            next = pre->next;
        }
    }
    rearrange(ma_algorithm);
    return 0;
}
```

```
277. int dispose(struct allocated_block *free_ab){
278.     struct allocated_block *pre, *ab;
279.     if(free_ab == allocated_block_head) {
280.     allocated_block_head = allocated_block_head->next;
281.     free(free_ab);
282.     return 1;
283.     }
284.     pre = allocated_block_head;
285.     ab = allocated_block_head->next;
286.     while(ab!=free_ab){ pre = ab; ab = ab->next; }
287.     pre->next = ab->next;
288.     free(ab);
289.     return 2;
290.  }
291. void kill_process(){
292.     struct allocated_block *ab;
293.     int pid;
294.     printf("Kill Process, pid=");
295.     scanf("%d", &pid);
296.     ab = find_process(pid);
297.     if(ab!=NULL){
298.         free_mem(ab);
299.         dispose(ab);
300.         printf("\nProcess %d has been killed\n", pid);
301.     }
302.     else
303.         printf("\nThere is no Process %d\n",pid);
304. }
305. int display_mem_usage(){
306.     struct free_block_type *fbt= free_block;
307.     struct allocated_block *ab=allocated_block_head;
308.     printf("----------------------------------------------------------\n");
309.     printf("Free Memory:\n");
310.     printf("%20s %20s\n", " start_addr", " size");
311.     while(fbt!=NULL){
312.         printf("%20d %20d\n", fbt->start_addr, fbt->size);
313.         fbt=fbt->next;
314.  }
315.  printf("\nUsed Memory:\n");
316.  printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start_addr", " size");
317.  while(ab!=NULL){
318.     printf("%10d %20s %10d %10d\n", ab->pid, ab->process_name,
319.     ab->start_addr, ab->size);
320.     ab=ab->next;
321.  }
322.  printf("----------------------------------------------------------\n");
323.  return 0;
324.  }
325. int main(){
326.     char choice; pid=0;
327.     free_block = init_free_block(mem_size);
328.     while(1) {
329.     display_menu();
330.         fflush(stdin);
```

```
        choice=getchar();
        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;
            }
        }
    return 0;
}
```

2.7.2 附件 2 Readme

见第二次实验文件夹下 readme.md 文档

3. 文件系统

3.1 实验目的

通过一个简单文件系统的设计，加深理解文件系统的内部实现原理

3.2 实验内容

模拟 EXT2 文件系统原理设计实现一个类 EXT2 文件系统

3.3 实验思想

为了进行简单的模拟，基于 Ext2 的思想和算法，设计一个类 Ext2 的文件系统，实现 Ext2 文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。

设计文件系统应该考虑的几个层次

介质的物理结构

物理操作——设备驱动程序完成

文件系统的组织结构（逻辑组织结构）

对组织结构其上的操作

为用户使用文件系统提供的接口

文件系统的基本实现

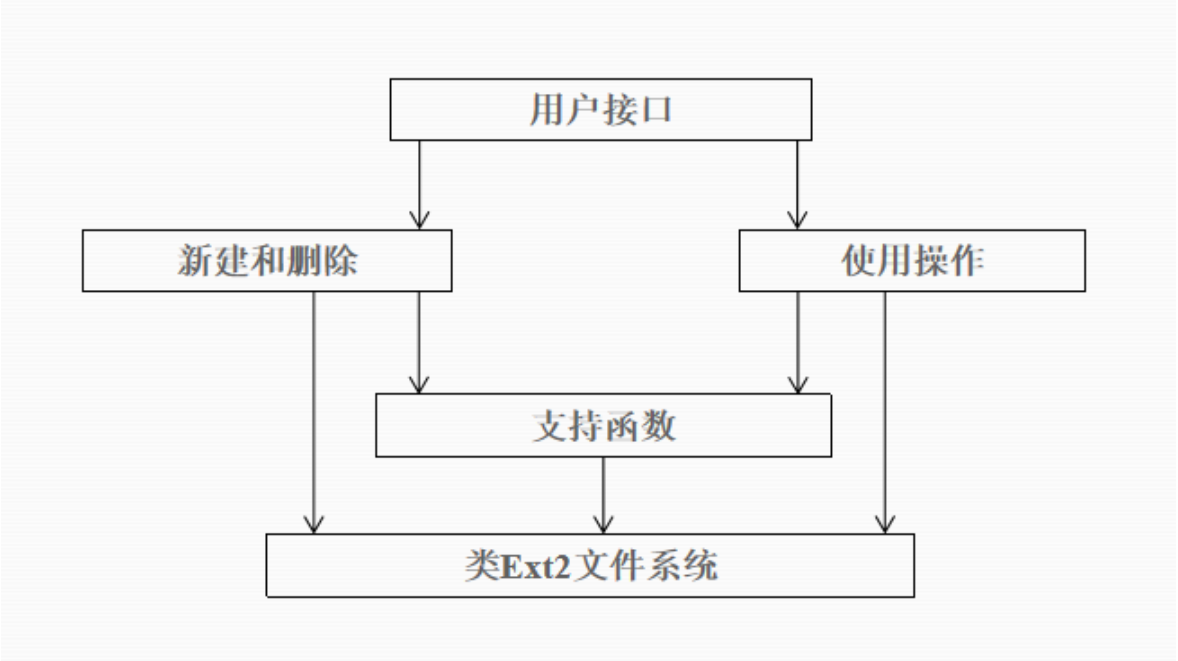
数据结构及其磁盘布局

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

整体布局、超级块、文件的索引、目录的存储、空闲块管理、块组描述符

文件的基本操作

接口和用户界面设计、节点创建删除设计、节点操作设计



类 ext2 文件系统的数据结构

块的定义

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

组描述符

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

struct ext2_group_desc

{

类型　bytes 域　　释意

char[ ] 16　　bg_volume_name[16];　　卷名

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

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

索引结点数据结构

由于容量已经确定，文件最大即为 2MB。需要 4096 个数据块。索引结点的数据结构中，仍然采用多级索引机制。由于文件系统总块数必然小于 4096*2，所以只需要 13 个二进制位即可对块进行全局计数，实际实现用 unsigned int 16 位变量，即 2 字节表示 1 个块号。在一级子索引中，如果一个数据块都用来存放块号，则可以存放 512/2＝256 个。因此，只使用一级子索引可以容纳最大的文件为 256*512=128KB。需要使用二级子索引。只使用二级子索引时，索引结点中的一个指针可以指向 256＊256 个块，即 256*256*512=8MB，已经可以满足要求了。为了尽量“像”ext2，也为了简单起见，索引结点的直接索引定义 6 个，一级子索引定义 1 个，二级子索引定义 1 个。总计 8 个指针。

索引结点的数据结构定义

struct ext2_inode {

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

}

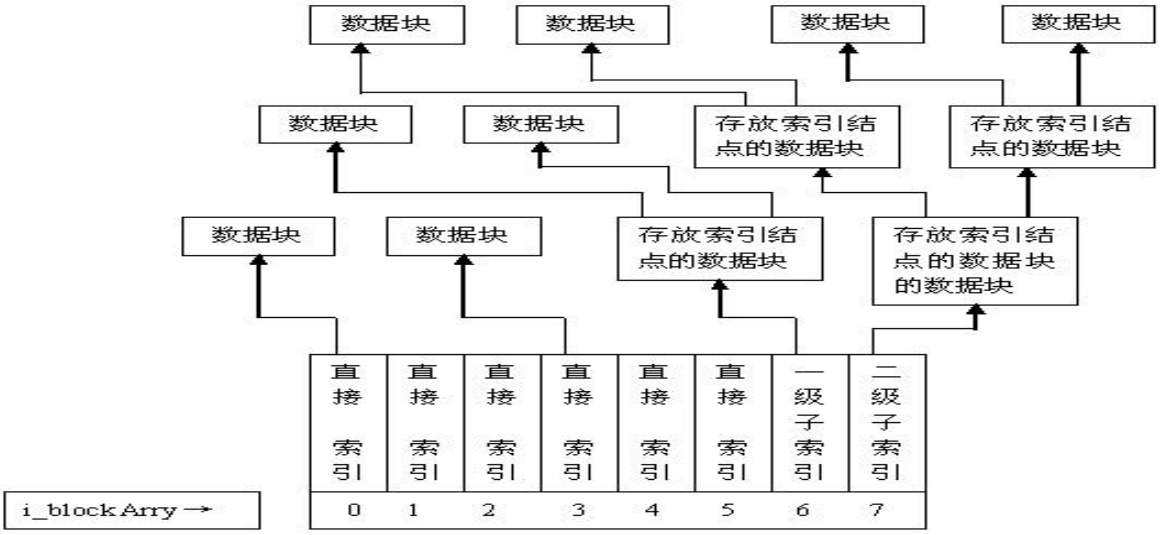
即，每个索引结点的长度为 64 字节。

索引节点各字段说明

i_mode 域构成一个文件访问类型及访问权限描述。即 linux 中的 drwxrwxrwx 描述。d 为目录，r 为读控制，w 为写控制，x 为可执行标志。并且，3 个 rwx 分别是所有者(owner)，组(group)，全局(universe)这三个对象的权限。为了简单的起见，模拟系统中，i_mode 的 16 位如下分配。高 8 位(high_i_mode)，是目录项中文件类型码的一个拷贝。低 8 位(low_i_mode)中的最低 3 位分别用来标识 rwx3 个属性。高 5 位不用，用 0 填充。在显示文件访问权限时，3 个对象均使用低 3 位的标识。这样，由这 16 位，即可生成一个文件的完整的 drwxrwxrwx 描述。

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

i_block 域与文件大小以及数据块的关系



⑴当文件长度小于等于 512*6=3072 字节（3KB）时，只用到直接索引

⑵当文件长度大于 3072 字节，并且小于等于 3KB+128KB 即 131KB 时，除使用直接索引处，还将使用一级子索引。

⑶当文件长度大于 131KB，并且小于 2MB（文件系统最大值）时，将开始使用二级子索引。

文件系统的数据容量

假设仅保存一个大文件，现在来计算这个文件最多能有多大。由于索引结点表本身要占用数据块，文件最大长度并没有 2MB。文件系统初始化时，根目录用去一个数据块。当文件大于 131KB 时，一级子索引本身还要占用一个数据块 512 字节。即，已经用去 131KB+0.5KB(根目录)+0.5KB(索引数据块)=132KB 的空间。

此外，由于文件系统的限制，二级子索引并不会全部使用。这里定义二级子索引的使用顺序是深度优先。即，必须先使用完第 1 个二级子索引 512*256=128KB 后，再使用第 2 个二级子索引。这样，每使用一个 2 级索引，多占用 1 个数据块，512 字节。假设使用了 n(<256)个二级子索引，占用的空间大小为(1+n)*512=(1+n)/2 KB。加上已经使用的 132KB 的空间，总共使用了的容量为 131.5+128*n+(1+n)/2　KB。令其等于文件系统的容量，2MB，解方程: 132+128*n+(1+n)/2=2048　得 n=14.91。即最多只能使用满 14 个二级子索引，最多可以使用到 15 个二级子索引。

由上可得，对一个文件，实际中 子索引系统所占用的数据块最多为 1+1+15=17 个。加上根目录，总共用了 18 个数据块来存放文件系统的信息。当 14 个二级子索引全部使用满时，用来存储文件数据的数据块最为 6+256+14*256=3846 个。当使用第 15 个二级子索引时，又用 1 个数据块来存放索引结点。文件系统总计有 4096 个数据块。那么留下给第 15 个二级子索引使用的数据块为 4096-18-3846=232。这是第 15 个二级子索引的索引结点数的上限。文件系统的实际容量为(4096-18)*512=2087936 字节＝2039 KB＝1.99MB。

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

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

假设每个文件大小平均分布，并把目录当成普通文件则平均占用数据块 4078/2=2039 个，平均使用索引结点块为：

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

个。则硬盘空间利用效率为 $\frac{2039}{2039+16}=99.22\%$

则文件系统容量大约为(4096－1)*512*99.22%＝2080214 字节＝2031.4KB=1.98MB。

索引结点表

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

模拟文件系统的“硬盘”数据结构

基于以上若干定义，得到模拟文件系统的“硬盘”数据结构：

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

整个模拟文件系统所需要的“硬盘”空间为 1+1+1+512+4096=4611 个块。共计 4611*512bytes=2,360,832 字节＝2305.5KB＝2.25MB。

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

目录与文件

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

目录体的数据结构与 ext2 基本相同，唯一的区别在于索引节点号用 16 位来表示：

struct ext2_dir_entry {

Type Bytes Field 释意

__u16 2 inode; 索引节点号

__u16 2 rec_len; 目录项长度

__u8 1 name_len; 文件名长度

__u8 1 file_type; 文件类型(1:普通文件，2:目录…)
char[ ] 8*_LEN name[EXT2_NAME_LEN]; 文件名
};

其中，文件名最大长度为 255 字符（节）。因此，目录项的长度范围是 7 至 261 字节。
当文件系统在初始化时，根目录的数据块（即数据块 1）将被初始化。其所包含的所有索引节点号以及目录项长度域将被置 0。当文件被删除时，其所在目录项长度不变，索引节点号将被置 0。
当新建一个文件时，程序将从目录的数据块查找索引节点号为 0 的目录项，并检查其长度是否够用。是，则分配给该文件，否则继续查找，直到找到长度够用，或者是长度为 0（即未被使用过）的地址，为文件建立目录项。
当建立的是一个目录时，将其所分配到的索引结点所指向的数据块清空。并且自动写入两个特殊的目录项。一个是当前目录“.”,其索引结点即指向本身的数据块。另一个是上一级目录“..”，其索引结点指向上一级目录的数据块。例如，/root 目录。其索引结点号为 1。并且，第 1 个数据块存放着该目录的目录项。/root 目录在文件系统初始化时自动生成。同时，在目录项中自动生成以下两项 ：

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

文件类型
文件类型项与 ext2 完全一样：

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

模拟文件系统的操作

		操作对象	
操作(命令)	功能	索引结点	数据块
dir	列当前目录	√	
mkdir	建立目录	√	√
rmdir	删除目录	√	√
create	建立文件	√	
delete	删除文件	√	√
cd	进入目录	√	
attrib	更改文件保护码	√	
open	打开文件	√	
close	关闭文件	√	
read	读文件	√	√
write	写文件	√	√

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

“存储空间”的管理

这里涉及到模拟文件系统的 5 个底层操作：索引结点的分配与释放、数据块的分配与释放以及数据块的寻址。

这些操作将采用 ext2 基本相同的方法实现。区别在于：Ext2 中对 superblock 的操作将变成对组描述符的操作。此外，数据块在分配时不采取预先分配策略。查找空闲块的方法可采用从某个起始点开始线性查找。

程序结构

初始化模拟文件系统

在已有的文件系统的基础上建立一个大小为 FS_SPACE=2,360,832 字节(即 2.25MB)的文件 FS.txt，这个文件即用来模拟硬盘。以后，文件系统的所有操作，均通过读写这个文件实现。并且，完全模拟硬盘读写方式，一次读取 1 个块，即 512 字节。即使只有 1 个字节的修改，也通过读写一个数据块来实现。

常驻内存的数据结构也被初始化。

文件系统级（底层）函数及其子函数

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

命令层函数

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

用户接口层

主要功能是接收及识别用户命令，词法分析，提取命令及参数。组织调用命令层对应的命令实现相应功能。本层实际上是一个基于命令层基础上的 shell。

为了完善接口的功能，shell 程序中增加了一些附加命令。这些命令无需调用文件系统级函数。这些附加命令有：quite 命令退出程序，format 命令重新建立文件系统。

各层函数列表

初始化文件系统

```
 initialize_disk()        /*建立文件系统*/
initialize_memory()     /*初始化文件系统的内存数据*/
```

底层

```
update_group_desc() /*将内存中的组描述符更新到"硬盘".*/
reload_group_desc() /*载入可能已更新的组描述符*/
load_inode_entry()   /*载入特定的索引结点*/
update_inode_entry() /*更新特定的索引结点*/
load_block_entry() /*载入特定的数据块*/
update_block_entry()/*更新特定的数据块*/
update_inode_i_block() /*根据多级索引机制更新索引结点的数据块信息域*/
ext2_new_inode() /*分配一个新的索引结点*/
ext2_alloc_block() /*分配一个新的数据块*/
ext2_free_inode() /*释放特定的索引结点*/
ext2_free_block_bitmap() /*释放特定块号的数据块位图*/
ext2_free_blocks()    /*释放特定文件的所有数据块*/
search_filename()    /*在当前目录中查找文件*/
test_fd()              /*检测文件打开 ID(fd)是否有效*/
```

命令层

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

用户接口层及附加命令

```
shell()                /*启动用户接口*/
format()               /*重新建立文件系统,无参数*/
quit()         /*退出 shell(),无参数*/
```

常驻内存的数据结构释意

```
unsigned short fopen_table[16] ; /*文件打开表，最多可以同时打开 16 个文件*/
unsigned short last_alloc_inode; /*上次分配的索引结点号*/
unsigned short last_alloc_block; /*上次分配的数据块号*/
unsigned short current_dir ;   /*当前目录(索引结点）*/
```

```
char current_path[256];          /*当前路径(字符串) */
struct ext2_group_desc;          /*组描述符*/
```

3.4 实验步骤

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

3.5 程序运行初值及运行结果分析

1 启动

```
[root@localhost LAB3]# ./main
Hello! Welcome to Ext2_like file system!
please input the password(init:123):***
[root@ext2]# dir
Type            FileName        CreateTime                      LastAccessTime                  ModifyTime
Directory       .               Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023
Directory       ..              Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023
[root@ext2]#
```

2 新建文件操作

```
[root@ext2]# create file1
Congratulations! file1 is created

[root@ext2]# open file1
Open successfully!

[root@ext2]# close file1
Close successfully!

[root@ext2]# dir
Type            FileName        CreateTime                      LastAccessTime                  ModifyTime
Directory       .               Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:18:36 2023
Directory       ..              Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:18:36 2023
File            file1           Tue Nov 28 16:18:41 2023        Tue Nov 28 16:18:44 2023        Tue Nov 28 16:18:41 2023
[root@ext2]#
```

3 对于文件的读写

```
[root@ext2]# write file1

hello world!


[root@ext2]# read file1

hello world!


[root@ext2]# open file1
The file has already been opened!

[root@ext2]# close file1
Close successfully!

[root@ext2]#
```

4 对于文件的删除

```
[root@ext2]# dir
Type            FileName        CreateTime                      LastAccessTime                  ModifyTime
Directory       .               Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:18:36 2023
Directory       ..              Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:18:36 2023
File            file1           Tue Nov 28 16:18:41 2023        Tue Nov 28 16:20:40 2023        Tue Nov 28 16:20:34 2023
[root@ext2]# delete file1
Congratulations! file1 is deleted!

[root@ext2]# dir
Type            FileName        CreateTime                      LastAccessTime                  ModifyTime
Directory       .               Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:21:59 2023
Directory       ..              Tue Nov 28 11:27:16 2023        Tue Nov 28 11:27:16 2023        Tue Nov 28 16:21:59 2023
[root@ext2]#
```

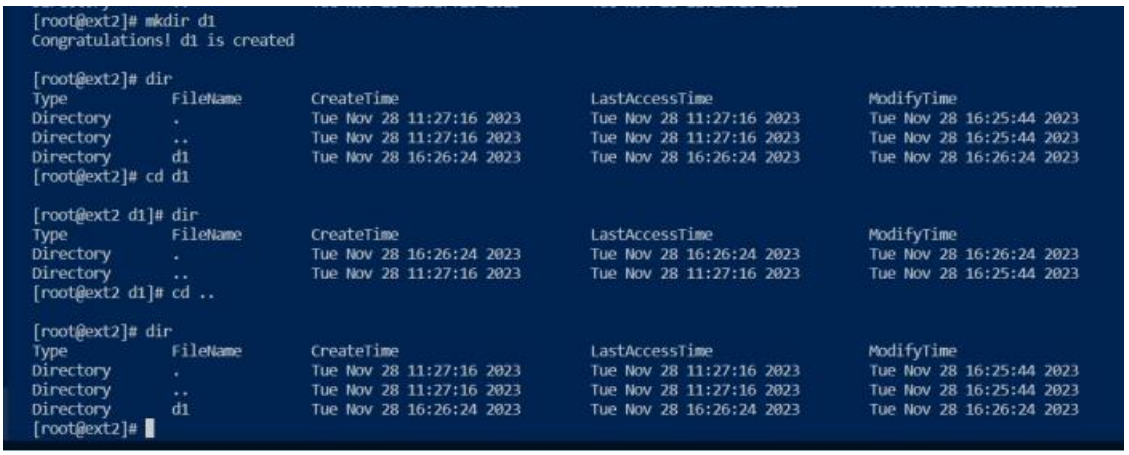
5 修改文件的读写权限

```
[root@ext2]# create file1
Congratulations! file1 is created

[root@ext2]# attrib file1 r
Modify successfully!

[root@ext2]# write file1
Failed! The file can't be written

[root@ext2]# attrib file1 w
Modify successfully!

[root@ext2]# read file1
Failed! The file can't be read

[root@ext2]# attrib file1 rw
Modify successfully!

[root@ext2]# read file1


[root@ext2]# write file1

hello world!


[root@ext2]# read file1

hello world!


[root@ext2]#
```

6 对文件夹的创建和删除

```
[root@ext2]# dir
Type          FileName       CreateTime                LastAccessTime            ModifyTime
Directory     .              Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:21:59 2023
Directory     ..             Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:21:59 2023
File          file1          Tue Nov 28 16:23:20 2023  Tue Nov 28 16:24:20 2023  Tue Nov 28 16:24:17 2023
[root@ext2]# delete file1
Congratulations! file1 is deleted!

[root@ext2]# dir
Type          FileName       CreateTime                LastAccessTime            ModifyTime
Directory     .              Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:24 2023
Directory     ..             Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:24 2023
[root@ext2]# mkdir d1
Congratulations! d1 is created

[root@ext2]# dir
Type          FileName       CreateTime                LastAccessTime            ModifyTime
Directory     .              Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:24 2023
Directory     ..             Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:24 2023
Directory     d1             Tue Nov 28 16:25:31 2023  Tue Nov 28 16:25:31 2023  Tue Nov 28 16:25:31 2023
[root@ext2]# rmdir d1
Congratulations! d1 is deleted!

[root@ext2]# dir
Type          FileName       CreateTime                LastAccessTime            ModifyTime
Directory     .              Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:44 2023
Directory     ..             Tue Nov 28 11:27:16 2023  Tue Nov 28 11:27:16 2023  Tue Nov 28 16:25:44 2023
[root@ext2]#
```

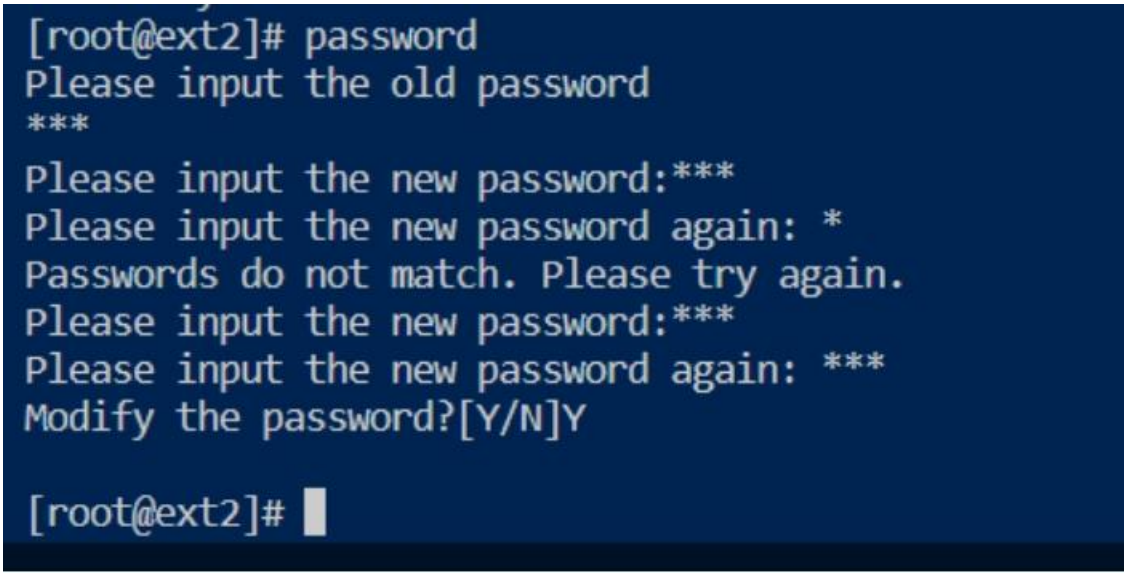
7 cd 命令



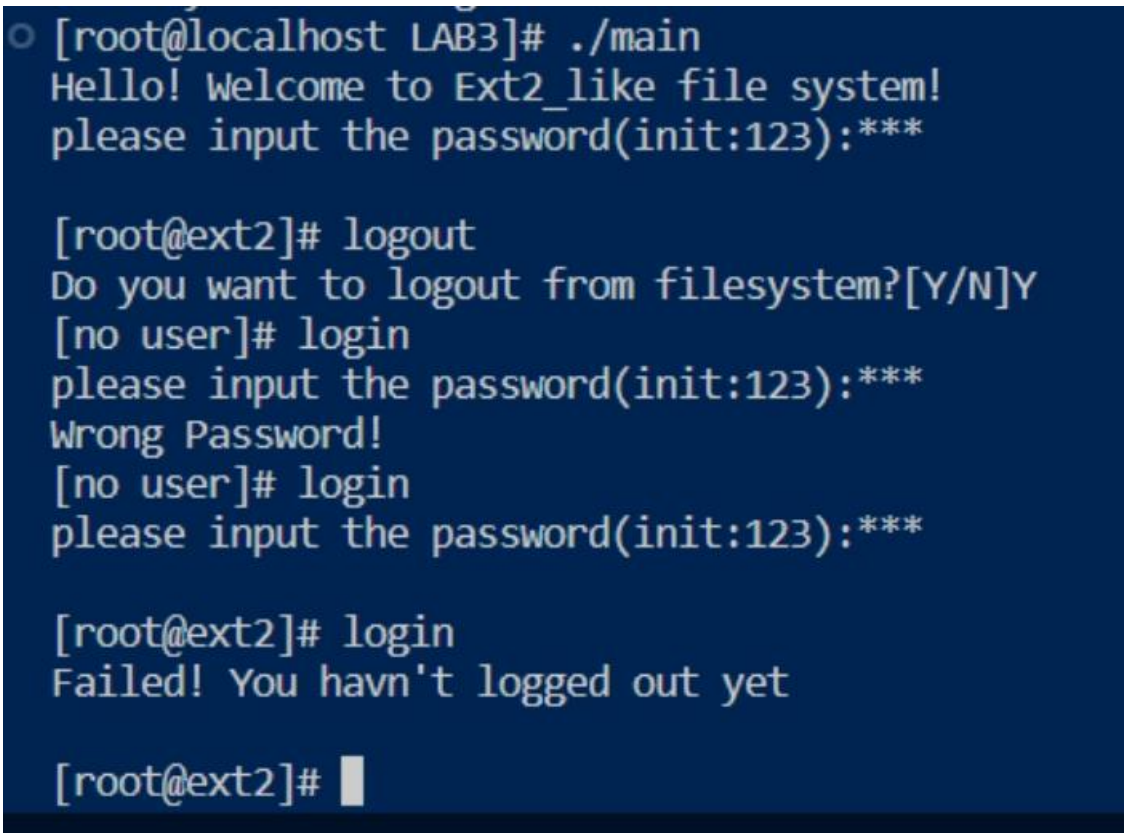
8password 命令

初始密码为 123，可以修改，在修改时需要先输入旧密码，之后输入两次新密码，若两次新密码一致，则可以修改，否则需重新输入新密码。

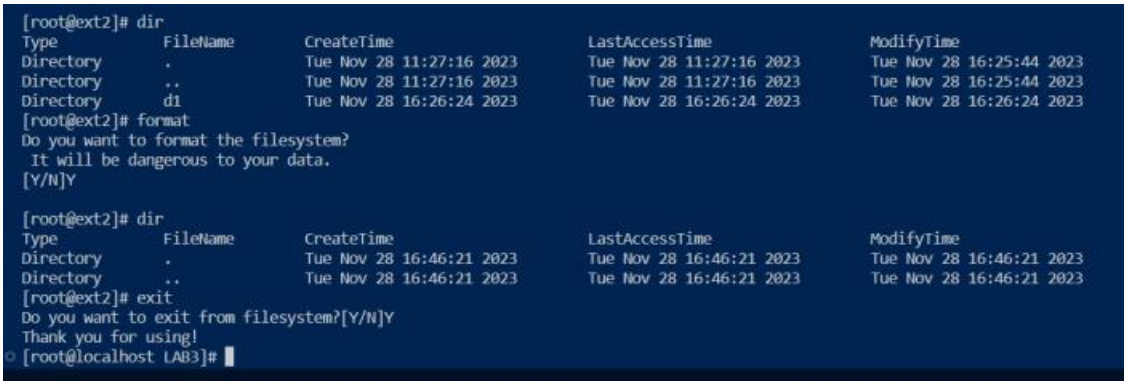
在任何需要输入密码的情况下，用户输入的密码不会回显，并且具有回删的功能



9login 和 logout 命令展示



10format 和 exit 命令展示



3.6 实验总结

3.6.1 实验中的问题与解决过程

1.为了实现密码和文件打开表等功能，对 PPT 中给出的数据结构进行了扩充修改。

2.在实现过程中，将头文件，全局变量，函数实现分开实现，使用预处理命令进行条件编译(一开始会报错，后面上网搜索以后发现问题)

3.底层函数实现，如系统初始化 format 函数，删除和创建文件操作在实现时遇到很大困难，通过参考实验流程图以及上网查阅资料解决

4.在实现对文件的操作时，需要用到 fseek、fread、fwrite、fopen、fclose 等函数，一开始不了解，通过查阅资料后了解了函数的参数、输出结果、各个参数的含义。

5.调试代码遇到的问题：

1)关于缓冲区的问题 clearbuffer 解决

在实现 logout 之后重新 login 的过程时，logout 之后，输入 login 则重新登录，由于在用 scanf 读取输出时，没有将回车符读走，所以重新定义了 clearBuffer 函数来情况缓存区。之后成功实现了 login logout

2)回退键无效

对于 login、password 等函数，想实现的功能时，输入时隐藏，同时具有回退的功能。将输入设置为*很容易实现，但是添加回退功能时遇到困难。原因是 getch 函数的头文件 conio.h 在 Linux 中没有，通过查找资料，找到了一个在 Linux 中类似的实现为

```
1.  //in windows
2.  #include<stdio.h>
3.  #include<conio.h>
4.  int mian(){
5.      char c;
6.      printf("input a char:");
7.      c=getch();
8.      printf("You have inputed:%c \n",c);
9.      return 0;
10. }
11.
12. //in linux
13. #include<stdio.h>
14. int main(){
15.     char c;
16.     printf("Input a char:");
17.     system("stty -echo");   //不显示输入内容
18.     c=getchar();
19.     system("stty echo");
20.     printf("You have inputed:%c \n",c);
21.     return 0;
22. }
```

3.6.2 实验收获

在本实验中通过设计和实现一个类 Ext2 文件系统，使我深入了解了文件系统的内部工作原理，包括文件的创建、删除、读取、写入等基本操作的实现方式。

在模拟 EXT2 文件系统的过程中，我对 EXT2 文件系统的设计和实现有了更为深刻的理解。这也有助于你对其他文件系统的理解，因为许多文件系统共享一些基本的设计原理。

我成功地实现了一个包括文件创建、删除、目录切换、文件关闭、读取、写入、密码修改、文件系统格式化、退出、登录、显示当前目录、新建文件夹、删除文件夹、修改文件权限、打开文件等功能的文件系统。

在实验过程中，我遇到了一些异常情况和错误，例如文件不存在、权限不足等，学会如何处理这些情况对于一个健壮的文件系统至关重要。这有助于我提高代码的稳定性和鲁棒性。

总体来说，这个实验使我在文件系统领域获得了丰富的经验，涵盖了文件操作、用户管理、异常处理等多个方面，为我在计算机科学和系统设计领域的深入学习打下了坚实的基础。

3.6.3 意见与建议

目前的实验设计已经很完善，可以增加多用户的实现要求。

3.7 附件

3.7.1 附件 1 程序

见第三次实验文件夹下代码部分

3.7.2 附件 2 Readme

见第三次实验文件夹下 readme.md 文档