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

1.1 实验目的 ............................................................................................................................... 1

1.2 实验内容 ............................................................................................................................... 2

1.3 实验思想 ............................................................................................................................... 3

1.4 实验步骤 ............................................................................................................................... 5

1.5 实验总结 ............................................................................................................................. 10

1.5.1 实验中的问题与解决过程...................................................................................... 10

1.5.2 实验收获.................................................................................................................. 11

1.5.3 意见与建议.............................................................................................................. 12

1.6 附件 ..................................................................................................................................... 12

1.6.1 附件 1 程序............................................................................................................. 12

1.6.2 附件 2 Readme......................................................................................................... 19

2 进程通信与内存管理..................................................................................................................... 19

2.1 实验目的 ............................................................................................................................. 19

2.2 实验内容 ............................................................................................................................. 20

2.3 实验思想 ............................................................................................................................. 21

2.4 实验结果 ............................................................................................................................. 22

2.5 回答问题 ............................................................................................................................. 24

2.5.1 软中断通信.............................................................................................................. 24

2.5.2 管道通信.................................................................................................................. 26

2.5.3 内存的分配与回收................................................................................................... 27

2.6 实验总结 ............................................................................................................................. 29

2.6.1 实验中的问题与解决过程...................................................................................... 29

2.6.2 实验收获.................................................................................................................. 29

2.7 附件 ................................................................................................................................... 30

2.7.1 附件 1 程序........................................................................................................... 30

2.7.2 附件 2 Readme....................................................................................................... 40

3 文件系统 ........................................................................................................................................ 40

3.1 实验目的 ............................................................................................................................. 40

3.2 实验内容 ............................................................................................................................. 40

3.3 实验思想 ............................................................................................................................. 40

3.4 实验步骤 ........................................................................................................................... 48

3.5 程序运行初值及运行结果分析..........................................................................................48

3.6 实验总结 ........................................................................................................................... 53

3.6.1 实验中的问题与解决过程.................................................................................... 53

3.6.2 实验收获................................................................................................................ 54

I

3.6.3 意见与建议............................................................................................................ 54

3.7 附件 ................................................................................................................................... 54

3.7.1 附件 1 程序........................................................................................................... 54

3.7.2 附件 2 Readme....................................................................................................... 54

I

# 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。

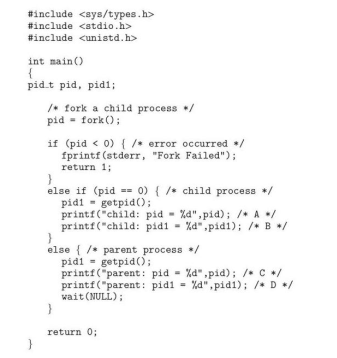


图 1-1 教材中所给代码（p103 作业 3.7）

（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。

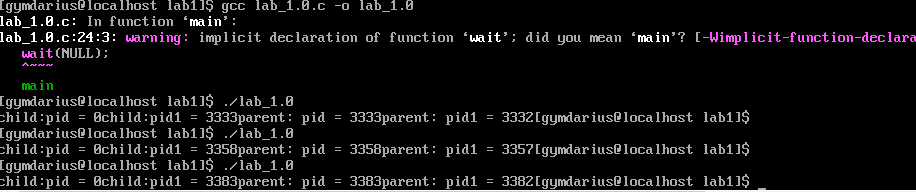
（ 3） 释放锁： 当持有锁的线程完成对共享资源的操作后，它会通过原子操作将标志变量设置回未锁定状态（ 0），从而释放锁，允许其他等待的线程尝试获取锁。

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

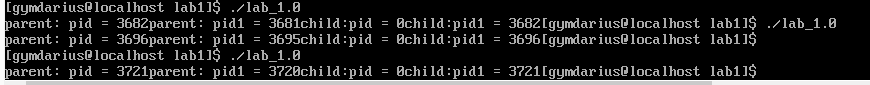
**进程相关编程实验 ：**

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：

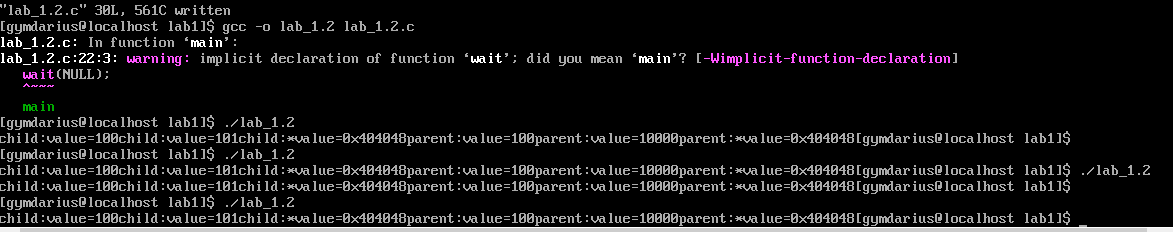


该运行结果中parent的输出早于child的输出，正好与不删去wait()函数时相反。

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：

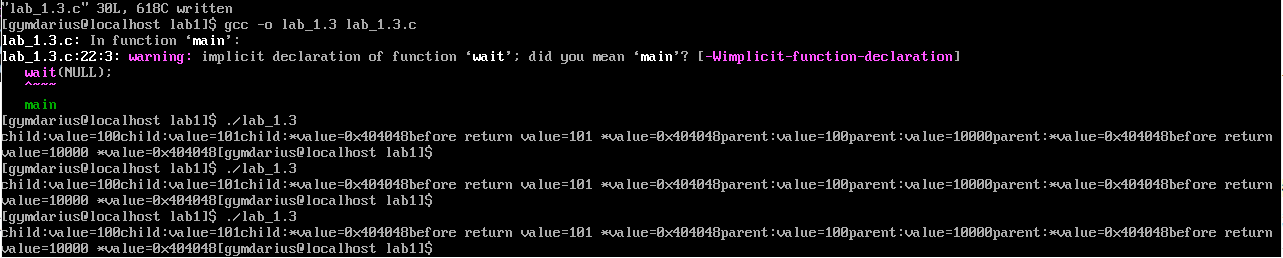


操作：定义一个全局变量，在子进程和父进程中进行不同的操作，并且输出其地址。

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()**

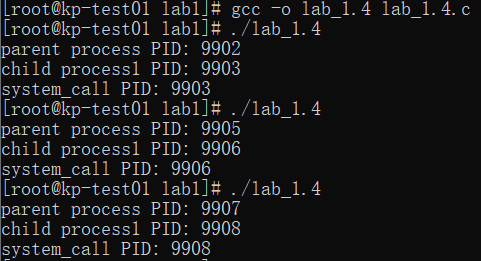
运行结果：

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

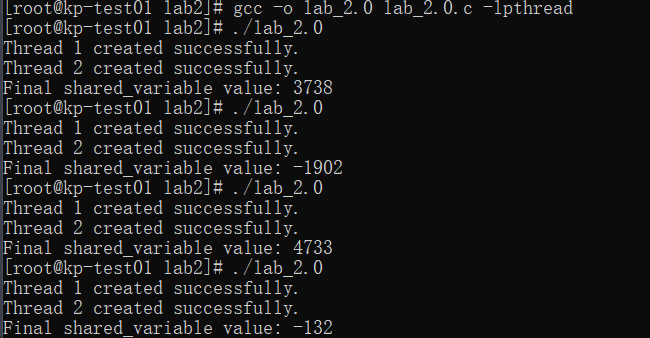
运行结果为：



**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：

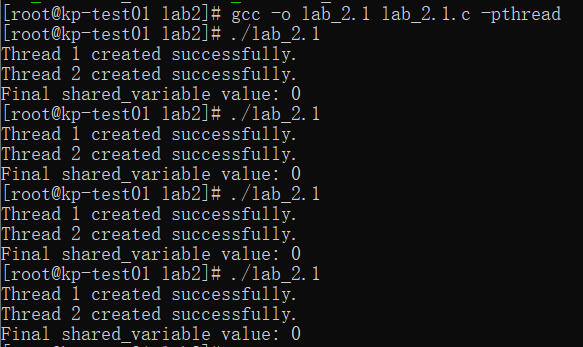


没有使用互斥锁或其它同步机制。这将导致两个线程同时访问和修改shared\_variable，由于竞争条件，输出结果可能是不确定的，并且在每次运行时都可能不同。

**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：

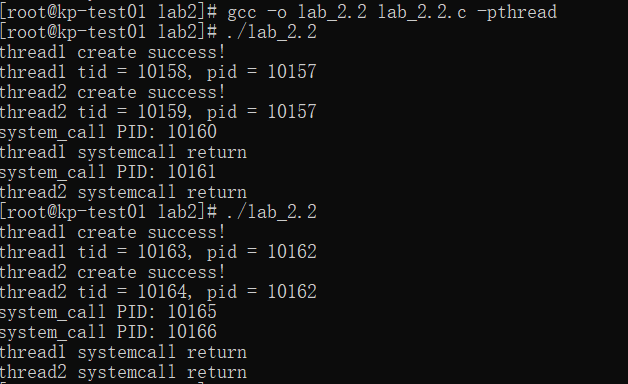


**步骤三：**在第一部分实验了解了 system()与 exec 族函数的基础上，将这两个函数的调用改为在线程中实现，输出进程 PID 和线程的 TID 进行分析。

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

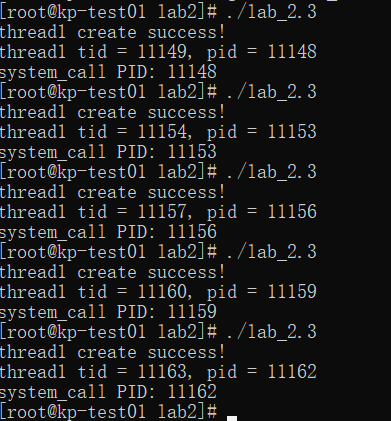
调用system()

运行结果为：

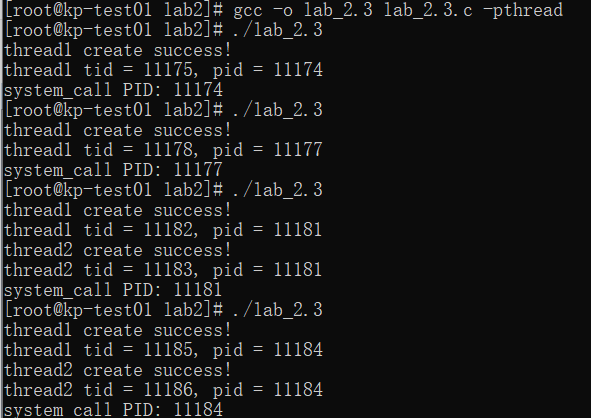


**进程和线程中调用exec族函数的区别：当调用exec族函数时，进程会完全替代其执行内容，而线程只会替代当前线程的执行内容。这意味着在多线程应用程序中，一个线程调用exec函数不会影响其他线程，而在多进程应用程序中，一个进程调用exec函数会替代整个进程。**

输出结果为：



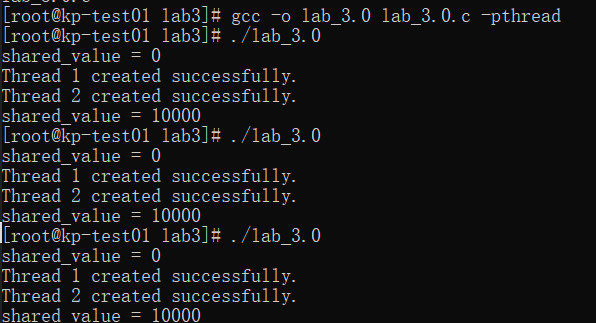
调整代码结构后，输出结果为：



**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：

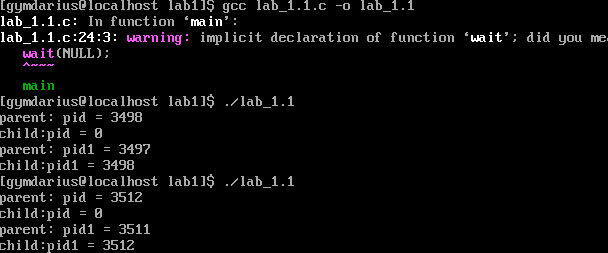


## 1.5 实验总结

### 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：



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。

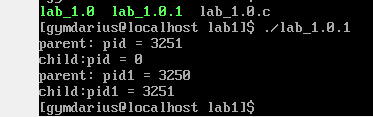
5. 程序结束时。

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）



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



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()

1. #include<sys/types.h>
2. #include<stdio.h>
3. #include<unistd.h>
5. **int** main(){
6. pid\_t pid,pid1;
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",pid);
16. printf("child:pid1= %d",pid1);
17. }
18. **else**{
19. pid1=getpid();
20. printf("parent:pid= %d",pid);
21. printf("parent:pid1= %d",pid1);
22. wait(NULL);
23. }
24. **return** 0;
25. }

2)图中程序删除wait()

1. #include<sys/types.h>
2. #include<stdio.h>
3. #include<unistd.h>
5. **int** main(){
6. pid\_t pid,pid1;
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;
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>
3. #include<unistd.h>
4. **int** value=100;
5. **int** main(){
6. pid\_t pid,pid1;
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. printf("before return value=%d \*value=%p",value,&value);
27. **return** 0;
28. }

5)在子进程中调用exec 族函数

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <sys/wait.h>
6. **int** main() {
7. pid\_t pid, pid1;
8. pid = fork();
9. **if** (pid < 0) {
10. fprintf(stderr, "Fork Failed\n");
11. **return** 1;
12. } **else** **if** (pid == 0) {
13. pid1 = getpid();
14. printf("child process1 PID: %d\n", pid1);
15. **char** \*args[] = {"./system\_call", NULL};
16. execv(args[0], args);
17. perror("execv");
18. exit(1);
19. printf("child process PID: %d\n", pid1);
20. } **else** {
21. pid1 = getpid();
22. printf("parent process PID: %d\n", pid1);
23. wait(NULL);
24. }
25. **return** 0;
26. }

6)在子进程中调用system()函数

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <unistd.h>
4. #include <sys/types.h>
5. #include <sys/wait.h>
6. **int** main() {
7. pid\_t pid, pid1;
8. pid = fork();
9. **if** (pid < 0) {
10. fprintf(stderr, "Fork Failed\n");
11. **return** 1;
12. } **else** **if** (pid == 0) {
13. pid1 = getpid();
14. printf("child process1 PID: %d\n", pid1);
15. fflush(stdout);
16. system("./system\_call");
17. printf("child process PID: %d\n", pid1);
18. } **else** {
19. pid1 = getpid();
20. printf("parent process PID: %d\n", pid1);
21. wait(NULL);
22. }
23. **return** 0;
24. }

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

1. #include <pthread.h>
2. #include<sys/types.h>
3. #include<stdio.h>
4. #include<unistd.h>
5. **int** shared\_variable = 0;
6. **void** \*thread\_function1(**void** \*thread\_id) {
7. **int** i;
8. **for** (i = 0; i < 100000; i++) {
9. shared\_variable++;
10. }
11. pthread\_exit(NULL);
12. }
13. **void** \*thread\_function2(**void** \*thread\_id) {
14. **int** i;
15. **for** (i = 0; i < 100000; i++) {
16. shared\_variable--;
17. }
18. pthread\_exit(NULL);
19. }
21. **int** main() {
22. pthread\_t thread1, thread2;
23. **if** (pthread\_create(&thread1, NULL, thread\_function1, (**void** \*)1) == 0) {
24. printf("Thread 1 created successfully.\n");
25. } **else** {
26. printf("Thread 1 creation failed.\n");
27. }
29. **if** (pthread\_create(&thread2, NULL, thread\_function2, (**void** \*)2) == 0) {
30. printf("Thread 2 created successfully.\n");
31. } **else** {
32. printf("Thread 2 creation failed.\n");
33. }
34. pthread\_join(thread1, NULL);
35. pthread\_join(thread2, NULL);
36. printf("Final shared\_variable value: %d\n", shared\_variable);
37. **return** 0;
38. }

8)定义信号量 signal，使用 PV 操作实现共享变量的访问与互斥。

1. #include <pthread.h>
2. #include<sys/types.h>
3. #include<stdio.h>
4. #include<unistd.h>
5. #include <semaphore.h>
6. **int** shared\_variable = 0;
7. sem\_t sem2;
8. **void** \*thread\_function1(**void** \*thread\_id) {
9. **int** i;
10. **for** (i = 0; i < 100000; i++) {
11. shared\_variable++;
12. }
13. sem\_post(&sem2);
14. pthread\_exit(NULL);
15. }
16. **void** \*thread\_function2(**void** \*thread\_id) {
17. **int** i;
18. sem\_wait(&sem2);
19. **for** (i = 0; i < 100000; i++) {
20. shared\_variable--;
21. }
22. pthread\_exit(NULL);
23. }
24. **int** main() {
25. pthread\_t thread1, thread2;
26. sem\_init(&sem2, 0, 0);
27. **if** (pthread\_create(&thread1, NULL, thread\_function1, (**void** \*)1) == 0) {
28. printf("Thread 1 created successfully.\n");
29. } **else** {
30. printf("Thread 1 creation failed.\n");
31. }
32. **if** (pthread\_create(&thread2, NULL, thread\_function2, (**void** \*)2) == 0) {
33. printf("Thread 2 created successfully.\n");
34. } **else** {
35. printf("Thread 2 creation failed.\n");
36. }
37. pthread\_join(thread1, NULL);
38. pthread\_join(thread2, NULL);
39. sem\_destroy(&sem2);
40. printf("Final shared\_variable value: %d\n", shared\_variable);
41. **return** 0;
42. }

9)将system()函数的调用改为在线程中实现

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pthread.h>
4. #include <unistd.h>
5. #include <sys/types.h>
6. #include <sys/syscall.h>
7. **void** \*thread\_function1(**void** \*arg) {
8. pid\_t pid = getpid();
9. pid\_t tid = syscall(SYS\_gettid);
10. printf("thread1 create success!\n");
11. printf("thread1 tid = %d, pid = %d\n", tid, pid);
12. system("./system\_call");
13. printf("thread1 systemcall return\n");
14. pthread\_exit(NULL);
15. }
16. **void** \*thread\_function2(**void** \*arg) {
17. pid\_t pid = getpid();
18. pid\_t tid = syscall(SYS\_gettid);
19. printf("thread2 create success!\n");
20. printf("thread2 tid = %d, pid = %d\n", tid, pid);
21. system("./system\_call");
22. printf("thread2 systemcall return\n");
23. pthread\_exit(NULL);
24. }
25. **int** main() {
26. pthread\_t thread1, thread2;
27. **if** (pthread\_create(&thread1, NULL, thread\_function1, NULL) == 0) {
28. } **else** {
29. printf("Thread 1 creation failed.\n");
30. }
31. **if** (pthread\_create(&thread2, NULL, thread\_function2, NULL) == 0) {
32. } **else** {
33. printf("Thread 2 creation failed.\n");
34. }
35. pthread\_join(thread1, NULL);
36. pthread\_join(thread2, NULL);
37. **return** 0;
38. }

10)将exec族函数的调用改为在线程中实现

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pthread.h>
4. #include <unistd.h>
5. #include <sys/types.h>
6. #include <sys/syscall.h>
7. **void** \*thread\_function1(**void** \*arg) {
8. pid\_t pid = getpid();
9. pid\_t tid = syscall(SYS\_gettid);
10. printf("thread1 create success!\n");
11. printf("thread1 tid = %d, pid = %d\n", tid, pid);
12. **char** \*args[] = {"./system\_call", NULL};
13. execv(args[0], args);
14. perror("execv");
15. exit(1);
16. printf("thread1 systemcall return\n");
17. pthread\_exit(NULL);
18. }
19. **void** \*thread\_function2(**void** \*arg) {
20. pid\_t pid = getpid();
21. pid\_t tid = syscall(SYS\_gettid);
22. printf("thread2 create success!\n");
23. printf("thread2 tid = %d, pid = %d\n", tid, pid);
24. **char** \*args[] = {"./system\_call", NULL};
25. execv(args[0], args);
26. exit(1);
27. printf("thread2 systemcall return\n");
28. pthread\_exit(NULL);
29. }
30. **int** main() {
31. pthread\_t thread1, thread2;
32. pthread\_create(&thread1, NULL, thread\_function1, NULL);
33. pthread\_create(&thread2, NULL, thread\_function2, NULL);
34. pthread\_join(thread1, NULL);
35. pthread\_join(thread2, NULL);
36. **return** 0;
37. }

11)自旋锁实验

1. include <stdio.h>
2. #include <pthread.h>
3. **typedef** **struct** {
4. **int** flag;
5. } spinlock\_t;
6. **void** spinlock\_init(spinlock\_t \*lock) {
7. lock->flag = 0;
8. }
9. **void** spinlock\_lock(spinlock\_t \*lock) {
10. **while** (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {
11. }
12. }
13. **void** spinlock\_unlock(spinlock\_t \*lock) {
14. \_\_sync\_lock\_release(&lock->flag);
15. }
16. **int** shared\_value = 0;
17. **void** \*thread\_function(**void** \*arg) {
18. spinlock\_t \*lock = (spinlock\_t \*)arg;**for** (**int** i = 0; i < 5000; ++i) {
19. spinlock\_lock(lock);
20. shared\_value++;
21. spinlock\_unlock(lock);
22. }
23. **return** NULL;
24. }
25. **int** main() {
26. pthread\_t thread1, thread2;
27. spinlock\_t lock;
28. printf("shared\_value = %d\n",shared\_value);
29. spinlock\_init(&lock);
30. **if** (pthread\_create(&thread1, NULL, thread\_function, &lock) == 0) {
31. printf("Thread 1 created successfully.\n");
32. } **else** {
33. printf("Thread 1 creation failed.\n");
34. }
36. **if** (pthread\_create(&thread2, NULL, thread\_function, &lock) == 0) {
37. printf("Thread 2 created successfully.\n");
38. } **else** {
39. printf("Thread 2 creation failed.\n");
40. }
41. pthread\_join(thread1, NULL);
42. pthread\_join(thread2, NULL);
43. printf("shared\_value = %d\n", shared\_value);
44. **return** 0;
45. }

#### 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：

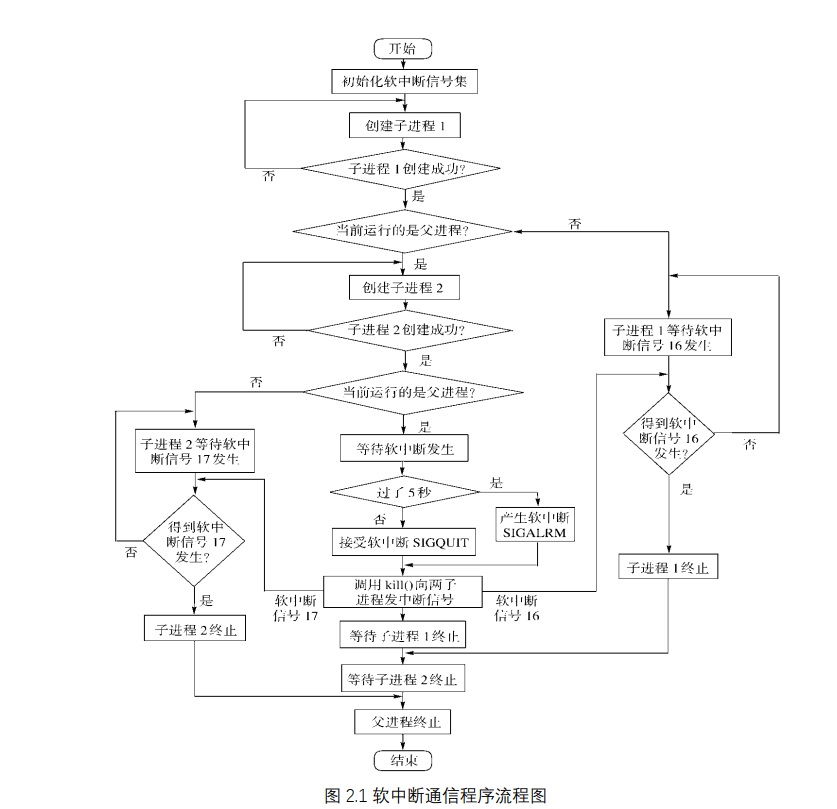
（ 1）理解内存分配 FF， BF， WF 策略及实现的思路。

（ 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

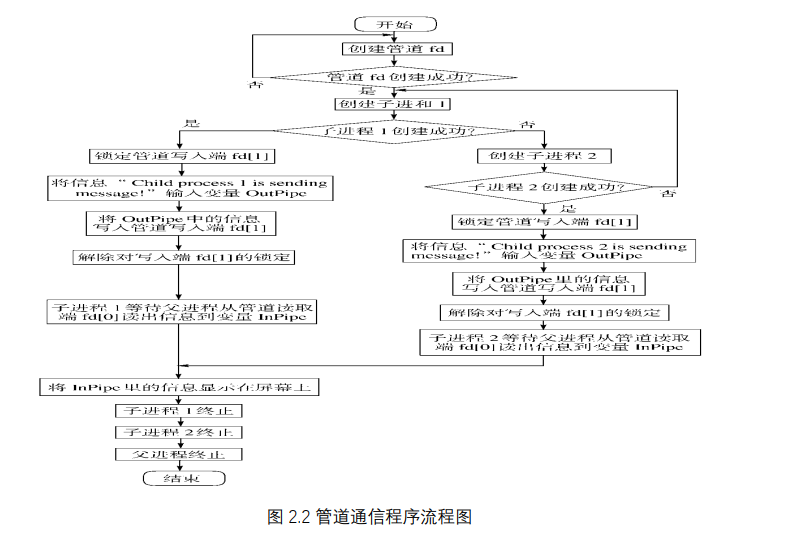
进程的软中断通信：



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:



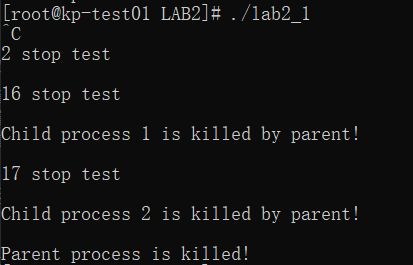
要实现这个功能，首先需要学习进程间管道通信。进程通过调用 pipe 函数，在内核中开辟一块缓冲区用来进行进程间通信，这块缓冲区称为管道，它有一个读端和一个写端。 pipe 函数接受一个参数，是包含两个整数的数组，如果调用成功，会通过pipefd[2]传出给用户程序两个文件描述符，需要注意 pipefd [0]指向管道的读端, pipefd [1]指向管道的写端，那么此时这个管道对于用户程序就是一个文件，可以通过 read(pipefd [0])；或者 write(pipefd [1])进行操作， pipe 函数调用成功返回 0，否则返回-1.

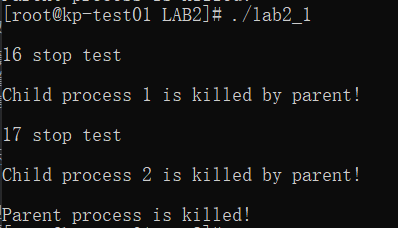
内存的分配和回收:

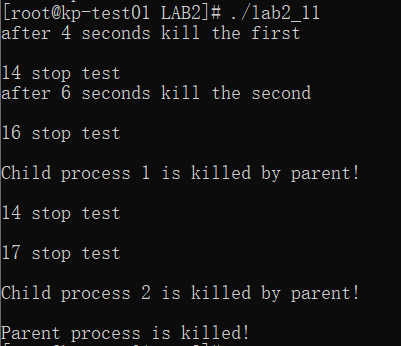
内存管理中实现了三种内存管理算法（FF， BF， WF），具体原理见实验代码注释.代码设计：将程序分为显示信息、初始化、进程创建、进程销毁、退出等几个模块，并分别实现代码

## 2.4 实验结果

进程的软中断通信

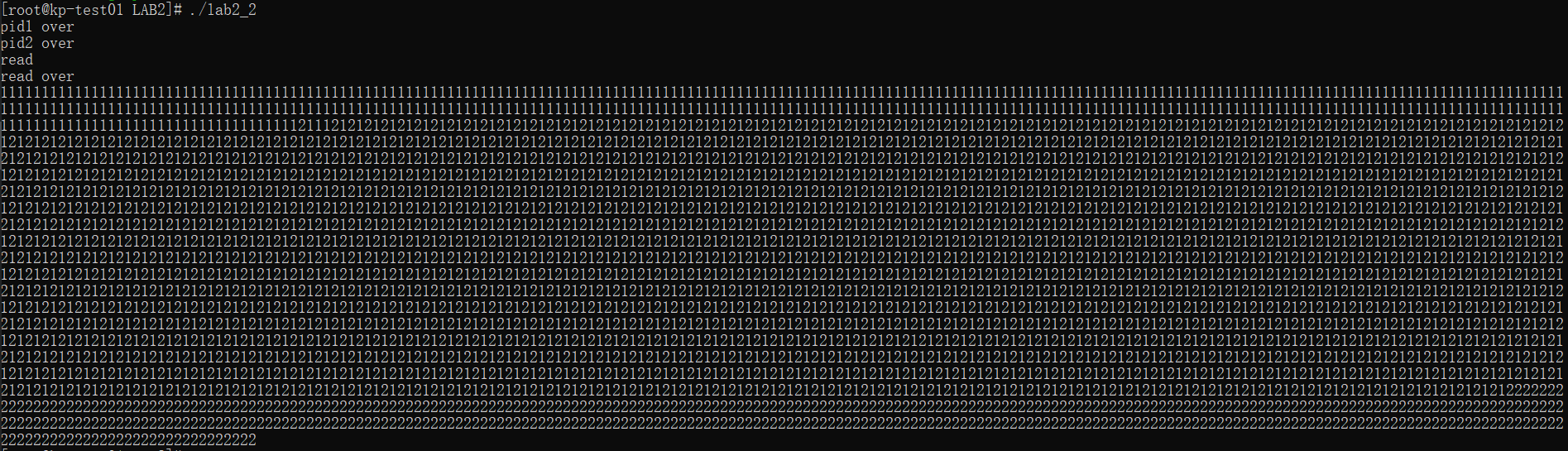




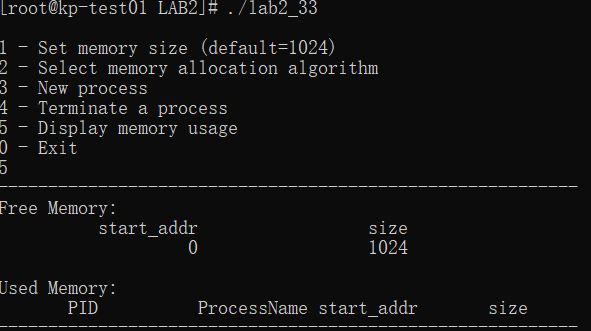


进程的管道通信





内存的分配和回收



## 2.5 回答问题

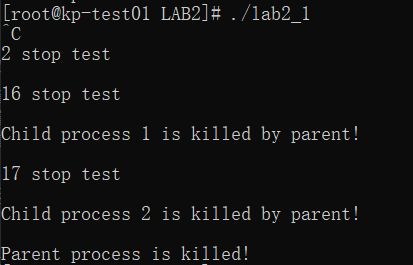
#### 2.5.1软中断通信

1、最初认为的运行结果

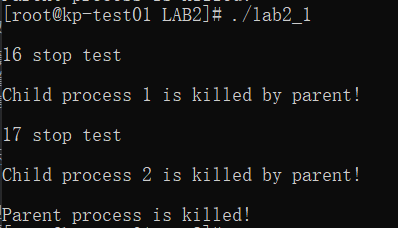
我最初认为运行结果是 如果5s内从键盘手动输出`Ctrl+C`发出中断信号，那么应当首先输出截获SIGINT信号对应的输出。接下来两个子进程的输出顺序是随机的，但是要求每个子进程需要满足，先输出捕获信号后执行函数对应的输出，然后打印输出子进程被父进程杀死的语句，最后输出父进程被回收的对应语句。拓扑结构即为 2 stop test在最前，17 stop test 在 Child process 2 is killed 前，16 stop test 在 Child process 1 is killed 前，parent process is killed 在最后，其余顺序可以随意出现。

2实际的运行结果

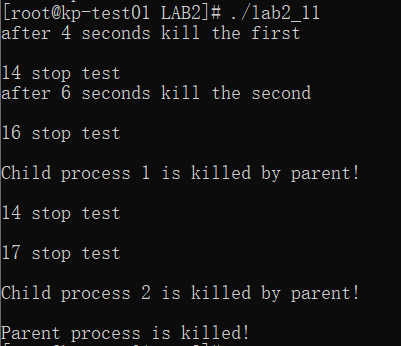
5s内中断



5s后中断



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。

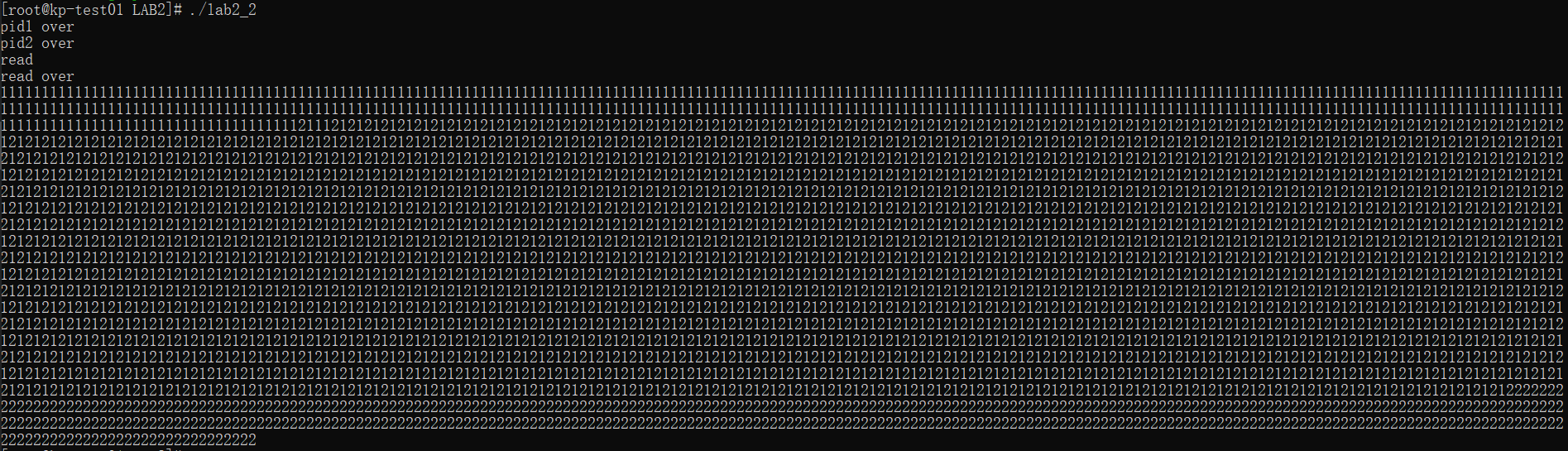
2.实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

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**

1. #include <stdio.h>
2. #include <unistd.h>
3. #include <sys/wait.h>
4. #include <stdlib.h>
5. #include <signal.h>


9. **void** inter\_handler(**int** sign)
10. {
11. **switch**(sign)
12. {
13. **case** SIGINT:
14. printf("\n2 stop test\n");
15. **break**;
16. **case** SIGSTKFLT:
17. printf("\n16 stop test\n");
18. **break**;
19. **case** SIGCHLD:
20. printf("\n17 stop test\n");
21. **break**;
22. **default**:;
23. }
24. }
26. **int** main()
27. {
28. pid\_t pid1, pid2;
29. **while**((pid1 = fork()) == -1);
30. **if**(pid1 > 0)
31. {
32. **while**((pid2 = fork()) == -1);
33. **if**(pid2 > 0)
34. {
35. signal(2, inter\_handler);
36. sleep(5);
37. kill(pid1, SIGSTKFLT); //kill the first child
38. kill(pid2, SIGCHLD); //kill the second child
39. wait(NULL);   //wait the first child's finish
40. wait(NULL);   //wait the second child's finish
41. printf("\nParent process is killed! \n");
42. exit(0);
43. }
44. **else**
45. {
46. signal(2, SIG\_IGN); //用于忽略SIGINT信号
47. signal(17, inter\_handler); //the second child is waiting
48. pause();    //暂停等待
49. printf("\nChild process 2 is killed by parent! \n");
50. exit(0);
51. }
52. }
53. **else**
54. {
55. signal(2, SIG\_IGN); //用于忽略SIGINT信号
56. signal(16, inter\_handler); //the first child is waiting
57. pause();    //暂停等待
58. printf("\nChild process 1 is killed by parent! \n");
59. exit(0);
60. }
61. **return** 0;
62. }
63. **进程软中断通信(闹钟中断)**
64. #include <stdio.h>
65. #include <unistd.h>
66. #include <sys/wait.h>
67. #include <stdlib.h>
68. #include <signal.h>
70. **void** inter\_handler(**int** sign)
71. {
72. **switch**(sign)
73. {
74. **case** SIGINT:
75. printf("\n2 stop test\n");
76. **break**;
77. **case** SIGSTKFLT:
78. printf("\n16 stop test\n");
79. **break**;
80. **case** SIGCHLD:
81. printf("\n17 stop test\n");
82. **break**;
83. **case** SIGALRM:
84. printf("\n14 stop test\n");
85. **break**;
86. **default**:;
87. }
88. }
90. **int** main()
91. {
92. pid\_t pid1, pid2;
93. **while**((pid1 = fork()) == -1);
94. **if**(pid1 > 0)
95. {
96. **while**((pid2 = fork()) == -1);
97. **if**(pid2 > 0)
98. {
99. signal(14, inter\_handler);
100. printf("after 4 seconds kill the first\n");
101. alarm(4);
102. sleep(8);       //sleep会被alarm函数发出的信号中断
103. kill(pid1, SIGSTKFLT); //kill the first child
104. printf("after 6 seconds kill the second\n");
105. alarm(6);
106. sleep(8);
107. kill(pid2, SIGCHLD); //kill the second child
108. wait(NULL);   //wait the first child's finish
109. wait(NULL);   //wait the second child's finish
110. printf("\nParent process is killed! \n");
111. exit(0);
112. }
113. **else**
114. {
115. signal(17, inter\_handler); //the second child is waiting
116. pause();
117. printf("\nChild process 2 is killed by parent! \n");
118. exit(0);
119. }
120. }
121. **else**
122. {
123. signal(16, inter\_handler); //the first child is waiting
124. pause();
125. printf("\nChild process 1 is killed by parent! \n");
126. exit(0);
127. }
128. **return** 0;
129. }
130. **进程的管道通信**
131. #include <unistd.h>
132. #include <signal.h>
133. #include <stdio.h>
134. #include <stdlib.h>
135. #include <fcntl.h>
136. **int** pid1,pid2;     // 定义两个进程变量
138. //define read/write end code
139. #define READEND 0
140. #define WRITEEND 1
142. **int** main(**int** argc, **char**\* argv[])
143. {
144. **int** fd[2];
145. **char** InPipe[1000];                  // 定义读缓冲区(buffer)
146. **char** c1='1', c2='2';
147. pipe(fd);                           // 创建管道
148. **while**((pid1 = fork( )) == -1);      // 如果进程1创建不成功,则空循环
150. **if**(pid1 == 0)                       // 子进程1
151. {
152. //lockf(fd[WRITEEND],F\_LOCK,0);   // 锁定管道
153. **int** i;
154. **for**(i=1;i<=2000;i++)             //  分200次每次向管道写入字符’1’
155. write(fd[WRITEEND],&c1,**sizeof**(**char**));
156. //sleep(5);                       // 等待读进程读出数据
157. //lockf(fd[WRITEEND],F\_ULOCK,0);  // 解除管道的锁定
158. printf("pid1 over\n");
159. exit(0);                        // 结束进程1
160. }
162. **else**
163. {
164. **while**((pid2 = fork()) == -1);       // 若进程2创建不成功,则空循环
165. **if**(pid2 == 0)                       // 子进程2
166. {
167. //lockf(fd[WRITEEND],F\_LOCK,0);   // 锁定管道
168. **int** i;
169. **for**(i=1;i<=2000;i++)             //  分200次每次向管道写入字符’2’
170. write(fd[WRITEEND],&c2,**sizeof**(**char**));
171. //sleep(5);                       // 等待读进程读出数据
172. //lockf(fd[WRITEEND],F\_ULOCK,0);  // 解除管道的锁定
173. printf("pid2 over\n");
174. exit(0);
175. }
176. **else**                                //父进程
177. {
178. waitpid(pid1,NULL,0);           // 等待子进程1 结束
179. waitpid(pid2,NULL,0);           // 等待子进程2 结束
180. printf("read\n");
181. read(fd[READEND],InPipe,4000\***sizeof**(**char**));
183. printf("read over\n");
184. InPipe[4000]='\0';               //  加字符串结束符
185. printf("%s\n",InPipe);          // 显示读出的数据
186. exit(0);                        // 父进程结束
187. }
188. }
189. **return** 0;
190. }
191. **内存的分配与回收**
192. #include<stdio.h>
193. #include<stdlib.h>
194. #include<unistd.h>
195. #define PROCESS\_NAME\_LEN 32
196. #define MIN\_SLICE 10
197. #define DEFAULT\_MEM\_SIZE 1024
198. #define DEFAULT\_MEM\_START 0
199. #define MA\_FF 1
200. #define MA\_BF 2
201. #define MA\_WF 3
202. **int** mem\_size=DEFAULT\_MEM\_SIZE;
203. **int** ma\_algorithm = MA\_FF;
204. **static** **int** pid = 0;
205. **int** flag = 0;
206. **struct** free\_block\_type{
207. **int** size;
208. **int** start\_addr;
209. **struct** free\_block\_type \*next;
210. };
211. **struct** free\_block\_type \*free\_block;
212. **struct** allocated\_block{
213. **int** pid; **int** size;
214. **int** start\_addr;
215. **char** process\_name[PROCESS\_NAME\_LEN];
216. **struct** allocated\_block \*next;
217. };
218. **struct** allocated\_block \*allocated\_block\_head = NULL;
219. **struct** free\_block\_type \*init\_free\_block(**int** mem\_size){
220. **struct** free\_block\_type \*fb;
221. fb=(**struct** free\_block\_type \*)malloc(**sizeof**(**struct** free\_block\_type));
222. **if**(fb==NULL){
223. printf("No mem\n");
224. **return** NULL;
225. }
226. fb->size = mem\_size;
227. fb->start\_addr = DEFAULT\_MEM\_START;
228. fb->next = NULL;
229. **return** fb;
230. }
231. **void** do\_exit(){
232. **while**(free\_block->next!= NULL&&free\_block !=NULL){
233. **struct** free\_block\_type \*temp = free\_block;
234. free\_block = free\_block->next;
235. free(temp);
236. }
237. **if**(free\_block)
238. free(free\_block);
239. }
240. **void** display\_menu(){
241. printf("\n");
242. printf("1 - Set memory size (default=%d)\n", DEFAULT\_MEM\_SIZE);
243. printf("2 - Select memory allocation algorithm\n");
244. printf("3 - New process \n");
245. printf("4 - Terminate a process \n");
246. printf("5 - Display memory usage \n");
247. printf("0 - Exit\n");
248. }
249. **int** set\_mem\_size(){
250. **int** size;
251. **if**(flag!=0){
252. printf("Cannot set memory size again\n");
253. **return** 0;
254. }
255. printf("Total memory size =");
256. scanf("%d", &size);
257. **if**(size>0) {
258. mem\_size = size;
259. free\_block->size = mem\_size;
260. }
261. flag=1; **return** 1;
262. }
263. **int** rearrange\_FF(){     //对起始地址进行冒泡排序
264. **struct** free\_block\_type \*head = free\_block;
265. **struct** free\_block\_type \*pre;
266. **while**(head->next != NULL){
267. pre = head->next;
268. **while**(pre != NULL){
269. **if**(head->start\_addr > pre->start\_addr){
271. **int** temp\_size = head->size;
272. **int** temp\_start = head->start\_addr;
273. head->size = pre->size;
274. head->start\_addr = pre->start\_addr;
275. pre->size = temp\_size;
276. pre->start\_addr = temp\_start;
277. }
278. pre = pre->next;
279. }
280. head = head->next;
281. }
282. **return** 1;
284. }
285. **int** rearrange\_BF(){ //对大小进行冒泡排序
286. **struct** free\_block\_type \*head = free\_block;
288. **struct** free\_block\_type \*pre;
289. **while**(head->next != NULL){
290. pre = head->next;
291. **while**(pre != NULL){
292. **if**(head->size > pre->size){
294. **int** temp\_size = head->size;
295. **int** temp\_start = head->start\_addr;
296. head->size = pre->size;
297. head->start\_addr = pre->start\_addr;
298. pre->size = temp\_size;
299. pre->start\_addr = temp\_start;
300. }
301. pre = pre->next;
302. }
303. head = head->next;
304. }
305. **return** 1;
306. }
307. **int** rearrange\_WF(){ //对大小进行逆向冒泡排序
308. **struct** free\_block\_type \*head = free\_block;
309. **struct** free\_block\_type \*pre;
310. **while**(head->next != NULL){
311. pre = head->next;
312. **while**(pre != NULL){
313. **if**(head->size < pre->size){
314. **int** temp\_size = head->size;
315. **int** temp\_start = head->start\_addr;
316. head->size = pre->size;
317. head->start\_addr = pre->start\_addr;
318. pre->size = temp\_size;
319. pre->start\_addr = temp\_start;
320. }
321. pre = pre->next;
322. }
323. head = head->next;
324. }
325. **return** 1;
326. }
327. **int** rearrange(**int** algorithm){
328. **switch**(algorithm){
329. **case** MA\_FF: **return** rearrange\_FF(); **break**;
330. **case** MA\_BF: **return** rearrange\_BF(); **break**;
331. **case** MA\_WF: **return** rearrange\_WF(); **break**;
332. }
333. }
334. **void** set\_algorithm(){
335. **int** algorithm;
336. printf("\t1 - First Fit\n");
337. printf("\t2 - Best Fit \n");
338. printf("\t3 - Worst Fit \n");
339. scanf("%d", &algorithm);
340. **if**(algorithm>=1 && algorithm <=3)
341. ma\_algorithm=algorithm;
342. rearrange(ma\_algorithm);
343. }
344. **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
345. **struct** free\_block\_type \*head, \*pre;
346. **int** request\_size=ab->size;
347. head = pre = free\_block;
348. **while**(head != NULL){
349. **if**(head->size >= request\_size){
350. **if**(head->size - request\_size <= MIN\_SLICE){
351. ab->start\_addr = head->start\_addr;
352. ab->size = head->size;
353. **if**(head == free\_block){
354. free\_block = head->next;
355. }
356. **else**{
357. pre->next = head->next;
358. }
359. free(head);
360. }
361. **else**{
362. ab->start\_addr = head->start\_addr;
363. ab->size = request\_size;
364. head->start\_addr += request\_size;
365. head->size -= request\_size;
366. }
367. rearrange(ma\_algorithm);
368. **return** 1;
369. }
370. pre = head;
371. head = head->next;
372. }
373. **if**(rearrange(ma\_algorithm) == 1){
374. pre = head = free\_block;
375. **while**(head != NULL){
376. **if**(head->size >= request\_size){
377. **if**(head->size - request\_size <= MIN\_SLICE){
378. ab->start\_addr = head->start\_addr;
379. ab->size = head->size;
380. **if**(head == free\_block){
381. free\_block = head->next;
382. }**else**{
383. pre->next = head->next;
384. }
385. free(head);
386. }**else**{
387. ab->start\_addr = head->start\_addr;
388. ab->size = request\_size;
389. head->start\_addr += request\_size;
390. head->size -= request\_size;
391. }
392. rearrange(ma\_algorithm);
393. **return** 1;
394. }
395. **else** **if**(head->size < MIN\_SLICE) **return** -1;
396. pre = head;
397. head = head->next;
398. }
399. }
400. **return** -1;
401. }
402. **int** new\_process(){
403. **struct** allocated\_block \*ab;
404. **int** size; **int** ret;
405. ab=(**struct** allocated\_block \*)malloc(**sizeof**(**struct** allocated\_block));
406. **if**(!ab) exit(-5);
407. ab->next = NULL;
408. pid++;
409. sprintf(ab->process\_name, "PROCESS-%02d", pid);
410. ab->pid = pid;
411. printf("Memory for %s:", ab->process\_name);
412. scanf("%d", &size);
413. **if**(size>0) ab->size=size;
414. ret = allocate\_mem(ab);
415. **if**((ret==1) &&(allocated\_block\_head == NULL)){
416. allocated\_block\_head=ab;
417. **return** 1; }
418. **else** **if** (ret==1) {
419. ab->next=allocated\_block\_head;
420. allocated\_block\_head=ab;
421. **return** 2; }
422. **else** **if**(ret==-1){
423. pid--;
424. printf("Allocation fail\n");
425. free(ab);
426. **return** -1;
427. }
428. **return** 3;
429. }
430. **struct** allocated\_block \*find\_process(**int** pid){
431. **struct** allocated\_block \*ab = allocated\_block\_head;
432. **while**(ab != NULL){
433. **if**(ab->pid == pid){
434. **return** ab;
435. }
436. ab = ab->next;
437. }
438. **return** NULL;
439. }
440. **int** free\_mem(**struct** allocated\_block \*ab){
441. **int** algorithm = ma\_algorithm;
442. **struct** free\_block\_type \*head, \*pre;
443. head=(**struct** free\_block\_type\*) malloc(**sizeof**(**struct** free\_block\_type));
444. **if**(!head) **return** -1;    //分配失败
445. head->size=ab->size;
446. head->start\_addr=ab->start\_addr;
447. head->next=free\_block;
448. free\_block = head;
449. rearrange\_FF(); //按照起始地址进行冒泡排序
450. pre = free\_block;
451. **struct** free\_block\_type \*next = pre->next;
452. **while**(next != NULL){
453. **if**(pre->start\_addr + pre->size != next->start\_addr){
454. pre = pre->next;
455. next = next->next;
456. }
457. **else**{
458. pre->size += next->size;
459. **struct** free\_block\_type \*temp = next;
460. pre->next = next->next;
461. free(temp);
462. next = pre->next;
463. }
464. }
465. rearrange(ma\_algorithm);
466. **return** 0;
467. }
468. **int** dispose(**struct** allocated\_block \*free\_ab){
469. **struct** allocated\_block \*pre, \*ab;
470. **if**(free\_ab == allocated\_block\_head) {
471. allocated\_block\_head = allocated\_block\_head->next;
472. free(free\_ab);
473. **return** 1;
474. }
475. pre = allocated\_block\_head;
476. ab = allocated\_block\_head->next;
477. **while**(ab!=free\_ab){ pre = ab; ab = ab->next; }
478. pre->next = ab->next;
479. free(ab);
480. **return** 2;
481. }
482. **void** kill\_process(){
483. **struct** allocated\_block \*ab;
484. **int** pid;
485. printf("Kill Process, pid=");
486. scanf("%d", &pid);
487. ab = find\_process(pid);
488. **if**(ab!=NULL){
489. free\_mem(ab);
490. dispose(ab);
491. printf("\nProcess %d has been killed\n", pid);
492. }
493. **else**
494. printf("\nThere is no Process %d\n",pid);
495. }
496. **int** display\_mem\_usage(){
497. **struct** free\_block\_type \*fbt= free\_block;
498. **struct** allocated\_block \*ab=allocated\_block\_head;
499. printf("----------------------------------------------------------\n");
500. printf("Free Memory:\n");
501. printf("%20s %20s\n", " start\_addr", " size");
502. **while**(fbt!=NULL){
503. printf("%20d %20d\n", fbt->start\_addr, fbt->size);
504. fbt=fbt->next;
505. }
506. printf("\nUsed Memory:\n");
507. printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start\_addr", " size");
508. **while**(ab!=NULL){
509. printf("%10d %20s %10d %10d\n", ab->pid, ab->process\_name,
510. ab->start\_addr, ab->size);
511. ab=ab->next;
512. }
513. printf("----------------------------------------------------------\n");
514. **return** 0;
515. }
516. **int** main(){
517. **char** choice; pid=0;
518. free\_block = init\_free\_block(mem\_size);
519. **while**(1) {
520. display\_menu();
521. fflush(stdin);
522. choice=getchar();
523. **switch**(choice){
524. **case** '1': set\_mem\_size(); **break**;
525. **case** '2': set\_algorithm();flag=1; **break**;
526. **case** '3': new\_process(); flag=1;**break**;
527. **case** '4': kill\_process(); flag=1; **break**;
528. **case** '5': display\_mem\_usage(); flag=1; **break**;
529. **case** '0': do\_exit(); exit(0);
530. **default**: **break**;
531. }
532. }
533. **return** 0;
534. }

#### 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

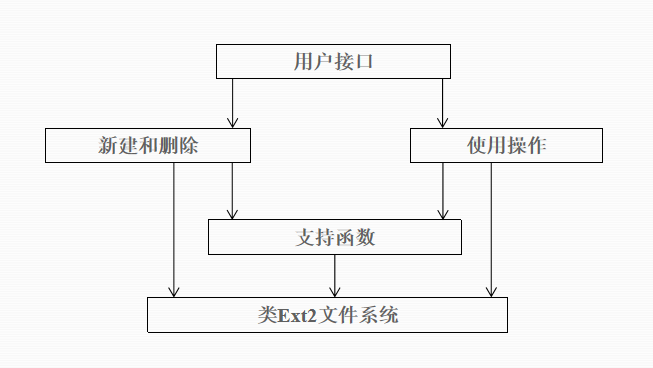
通过一个简单文件系统的设计，加深理解文件系统的内部实现原理

## 3.2实验内容

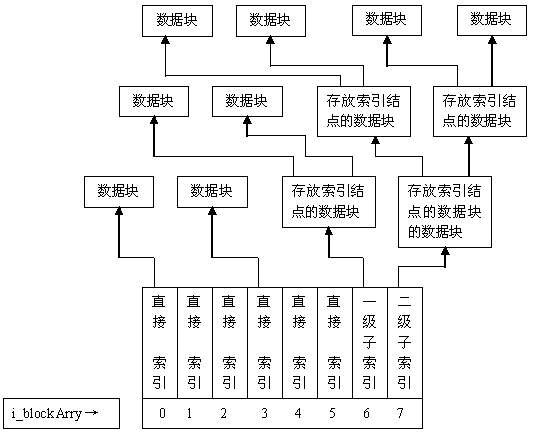
模拟EXT2文件系统原理设计实现一个类EXT2文件系统

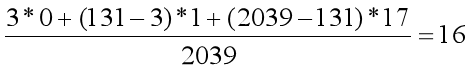
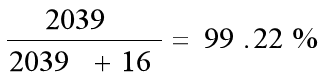
## 3.3实验思想

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：个。则硬盘空间利用效率为  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：

2 MB

256 KB

512 Bytes

512 Bytes

512 Bytes

4096 blocks

512 blocks

1 block

1 block

1 block

数据块

索引结点表

索引结点位图

数据块位图

组描述符

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 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。

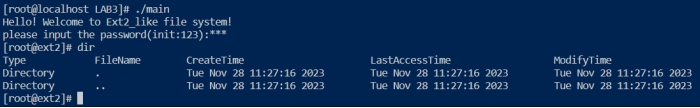
程序结构 初始化模拟文件系统 在已有的文件系统的基础上建立一个大小为FS\_SPACE=2,360,832字节(即2.25MB)的文件FS.txt，这个文件即用来模拟硬盘。以后，文件系统的所有操作，均通过读写这个文件实现。并且，完全模拟硬盘读写方式，一次读取1个块，即512字节。即使只有1个字节的修改，也通过读写一个数据块来实现。常驻内存的数据结构也被初始化。 文件系统级（底层）函数及其子函数 这些函数完成了所有文件系统底层的操作封装。并为上层即命令层提供服务。该层实现了所有对文件系统“硬盘”的块操作功能。例如：分配和回收索引结点与数据块，索引结点的读取与写入，数据块的读取与写入，索引结点及数据块位图的设置，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

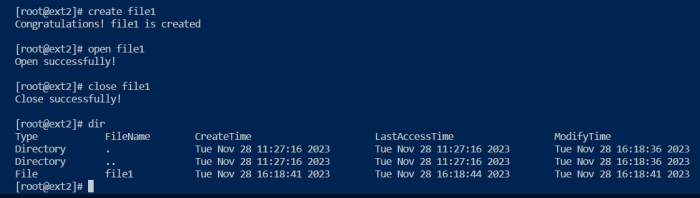
定义类 EXT2 文件系统所需的数据结构，包括组描述符、索引结点和目录项。实现底层函数，包括分配数据块等 操作。实现命令层函数，包括 dir 等操作。完成 shell 的设计。测试整个文件系统的功能。

## 3.5程序运行初值及运行结果分析

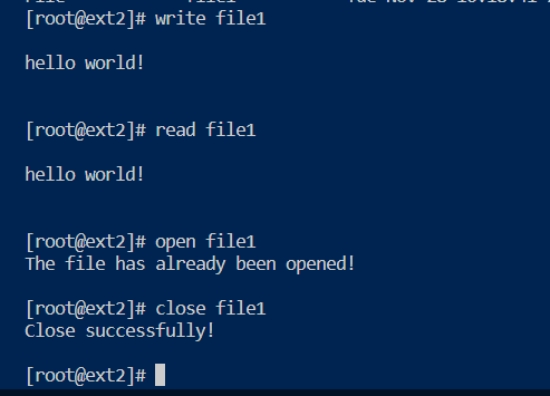
1启动



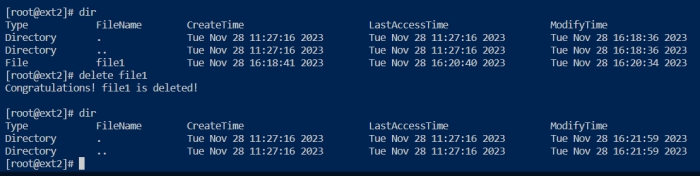
2新建文件操作



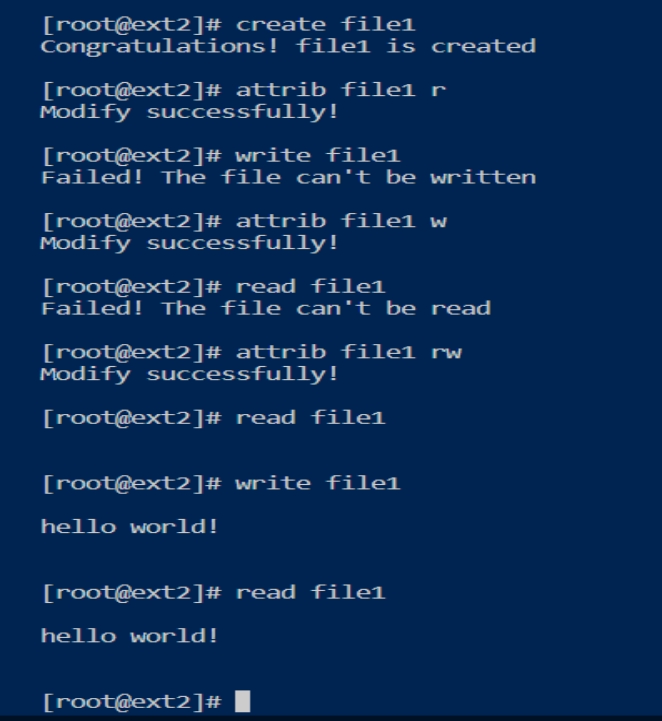
3对于文件的读写



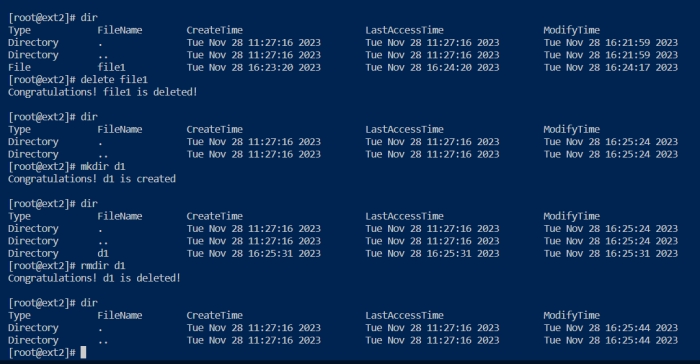
4对于文件的删除



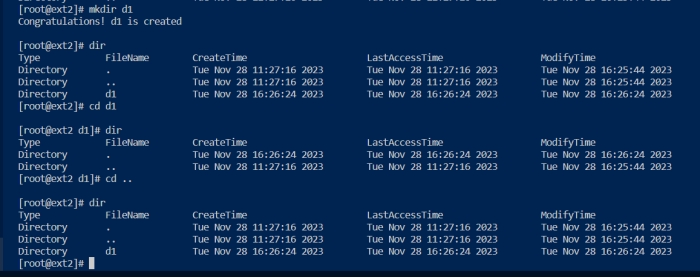
5修改文件的读写权限



6对文件夹的创建和删除



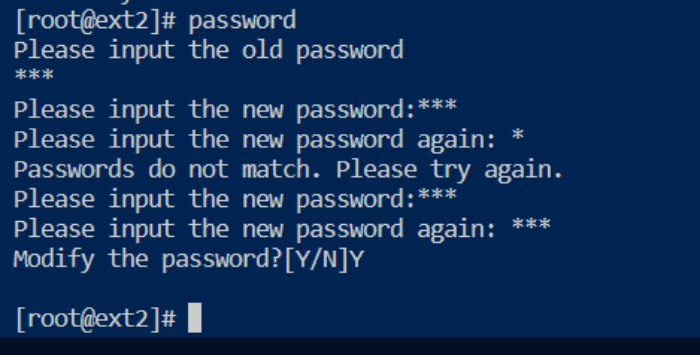
7 cd命令



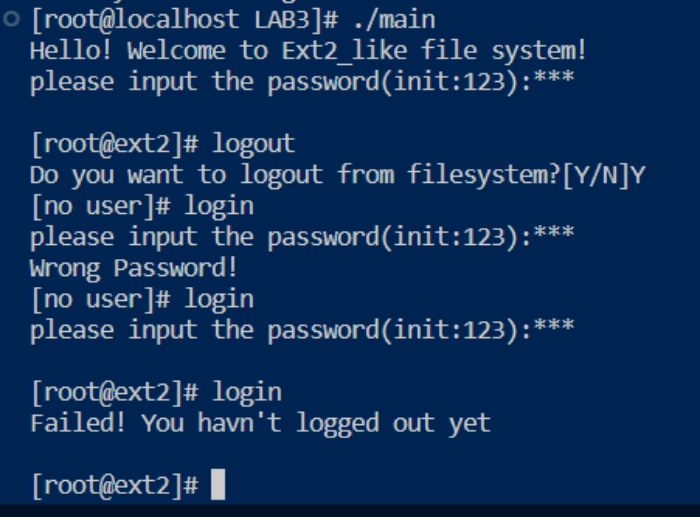
8password命令

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。

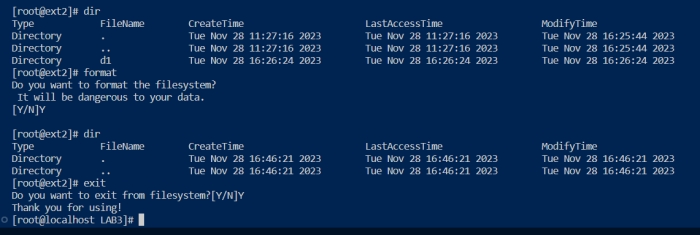
在任何需要输入密码的情况下，用户输入的密码不会回显，并且具有回删的功能



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. }
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