西安交通大学

**操作系统专题实验报告**

班级： 计算机91

学号： 2184214539

姓名： 叶文

2021年12月8日

目录

[1进程相关编程实验 1](#_Toc89822111)

[1.1实验目的 1](#_Toc89822112)

[1.2实验内容 1](#_Toc89822113)

[1.3实验思想（或流程图） 1](#_Toc89822114)

[1.4实验步骤 2](#_Toc89822115)

[1.5测试数据 2](#_Toc89822116)

[1.6程序运行初值及运行结果分析 3](#_Toc89822117)

[1.7实验总结 7](#_Toc89822118)

[1.7.1实验中的问题与解决过程 7](#_Toc89822119)

[1.1问题描述 7](#_Toc89822120)

[1.1解决过程：（网址，参考资料），具体解决方法 7](#_Toc89822121)

[2.1问题描述 8](#_Toc89822122)

[2.2解决过程： 8](#_Toc89822123)

[3.1问题描述 8](#_Toc89822124)

[3.2 解决过程 8](#_Toc89822125)

[1.7.2实验收获 8](#_Toc89822126)

[1.7.3意见与建议 9](#_Toc89822127)

[1.8附件 9](#_Toc89822128)

[1.8.1附件1 程序 9](#_Toc89822129)

[1.8.2附件2 Readme 11](#_Toc89822130)

[2 进程通信与内存分配 12](#_Toc89822131)

[2.1实验目的 12](#_Toc89822132)

[2.2实验内容 12](#_Toc89822133)

[2.3实验思想 12](#_Toc89822134)

[2.4实验步骤 13](#_Toc89822135)

[2.5程序运行初值及运行结果分析 13](#_Toc89822136)

[2.6实验总结 20](#_Toc89822137)

[2.6.1实验中的问题与解决过程 20](#_Toc89822138)

[1.问题描述 20](#_Toc89822139)

[2.解决过程： 20](#_Toc89822140)

[2.3.2实验收获 20](#_Toc89822141)

[2.3.3意见与建议 20](#_Toc89822142)

[2.4附件 21](#_Toc89822143)

[1.8.2附件2 Readme 34](#_Toc89822144)

[3 EXT2文件系统 35](#_Toc89822145)

[3.1实验目的 35](#_Toc89822146)

[3.2实验内容 35](#_Toc89822147)

[3.3实验思想 35](#_Toc89822148)

[3.4实验步骤及运行结果分析 36](#_Toc89822149)

[3.5实验总结 38](#_Toc89822150)

[3.5.1实验中的问题与解决过程 38](#_Toc89822151)

[1.1问题描述 38](#_Toc89822152)

[1.2解决过程： 38](#_Toc89822153)

[2.1问题描述 38](#_Toc89822154)

[2.2解决过程： 38](#_Toc89822155)

[3.5.2实验收获 39](#_Toc89822156)

[3.5.3意见与建议 39](#_Toc89822157)

[3.4附件 39](#_Toc89822158)

# 1进程相关编程实验

## 1.1实验目的

1. 学会fork()函数的用法及特点
2. 观察fork()函数调用后子父进程Pid的特点
3. 深入理解子父进程运行机制
4. 学会使用system及exec组函数并了解函数功能
5. 学会使用Pthread实现多线程

## 1.2实验内容

1. 通过利用fork()调用来观察父子进程的进程调度的结果
2. 利用父子进程来观察程序中全局变量的改变情况
3. 在子程序中执行system函数并观察情况
4. 在子程序中执行exec函数观察其特点
5. 对程序多次调用并观察其输出顺序
6. 使用 Pthread 实现多线程并观察其结果

## 1.3实验思想（或流程图）

1. **有关子进程的基础运行：**

通过调用fork()函数生成一个子进程，构造一个if语句在父子进程中分别打印pid 与pid1的值观察打印值，当去掉wait(NULL)后重新打印，加上sleep函数后打印输出结果，在以上不同的情况中分别多次打印查看输出结果有何不同

1. **进程调度中的全局变量的改变**

设置一个全局变量example在父子进程中分别进行操作然后输出全局变量，在return前增加对全局变量的操作并输出结果

1. **在子进程中分别执行system函数与exec函数**

通过在子进程中添加system模块与exec模块，在system’模块中调用echo输出信息，在exec组函数中调用Hello执行，来分别观察system函数作用及exec组函数的作用

1. **运用Pthread创建两个线程在同一进程下执行信息交换**

通过创建两个线程mythread1，mythread2函数对一变量进行操作，在两个线程中对变量操作结束后输出所操作的变量一次来验证线程间的信息交换。

## 1.4实验步骤

1. **有关子进程的基础运行：**

在Open Euler中运行通过 **./Testpid** 执行相应程序多次运行得到运行结果，见1.6

去掉初始程序中的 **wait(NULL)** 后多次执行相应程序得到运行结果，见1.6

在以上的基础上在父进程程序处添加sleep函数多次执行得到运行结果

1. **进程调度中的全局变量的改变**

在**main**函数前添加全局变量**example** 并在父子进程中分别对example进行操作的到运行结果，更进一步在**return** 前对**example**执行**\*2**操作并输出结果，多次运行得到运行结果

1. **在子进程中分别执行system函数与exec函数**

首先在子进程中添加：

**system("echo 'This is a system\_child process'");** 进行调用system函数

其次在子进程中添加：

**if(execl("/usr/local/os1/Hello",NULL) == -1){ printf("error in execl\n")； exit(1);}**

用以执行**exec** 族函数

1. **运用Pthread创建两个线程在同一进程下执行信息交换**

通过创建线程函数

**void \* mythread1(void \*t)，void \* mythread1(void \*t)**

用 **pthread\_create** 创建两个线程并执行对应对变量的操作，打印观察线程中对变量的操作

## 1.5测试数据

在全局变量中，子进程对其进行 **example = example + 1** 操作

父进程对其进行 **example = example +2** 操作

在其中对于全局变量**example = 2；**父进程对其加2为偶数；子进程对其加1为奇数

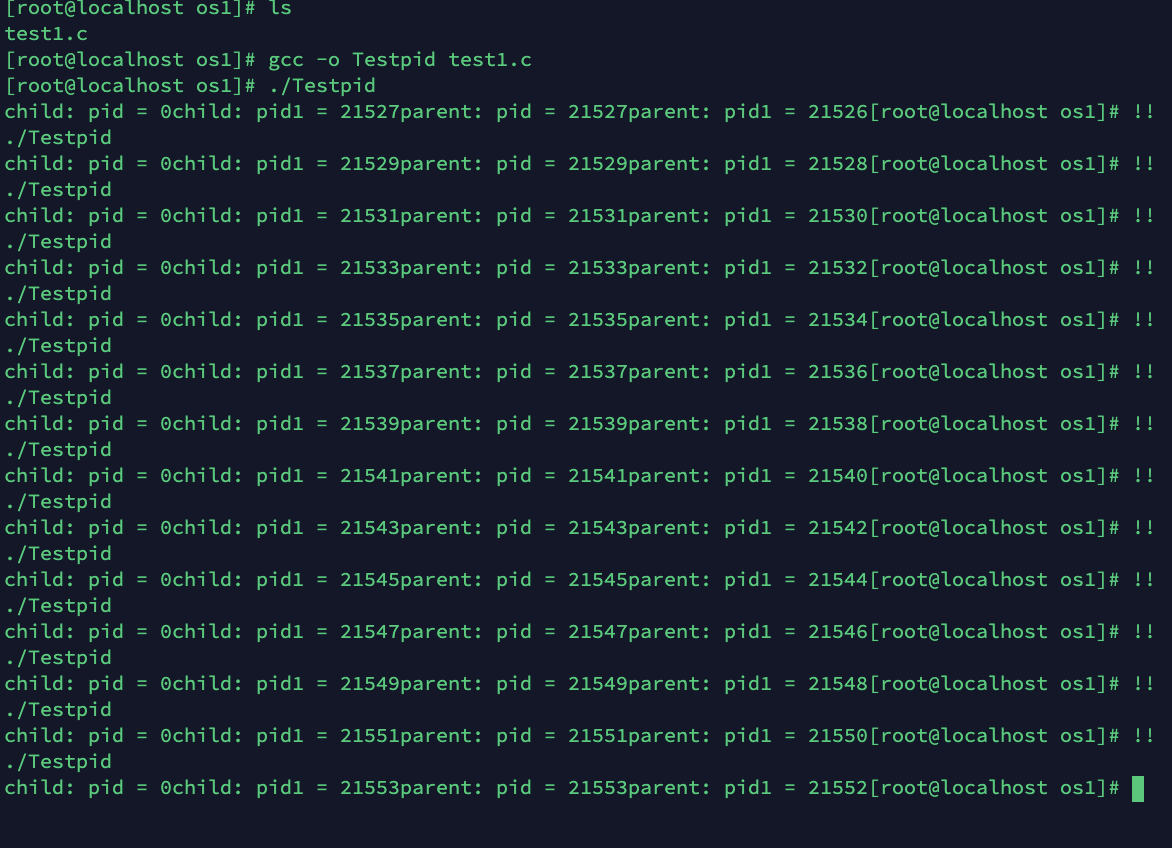
在system调用echo 有：**system("echo 'This is a system\_child process'");**

在exec调用中调用 **execl("/usr/local/os1/Hello")** 调用一个Hello World函数

在pthread中对mythread1线程执行**\*(int\*)t = \*(int\*)t + 1;，**对共享变量执行加1操作

在pthread中对mythread2线程执行**\*(int\*)t = \*(int\*)t + 2;，**对共享变量执行加2操作

## 1.6程序运行初值及运行结果分析



**对程序进行多次执行有如上结果：**

可见每次child pid 均为0 这是由于fork调用对于子进程来说，每次返回为0

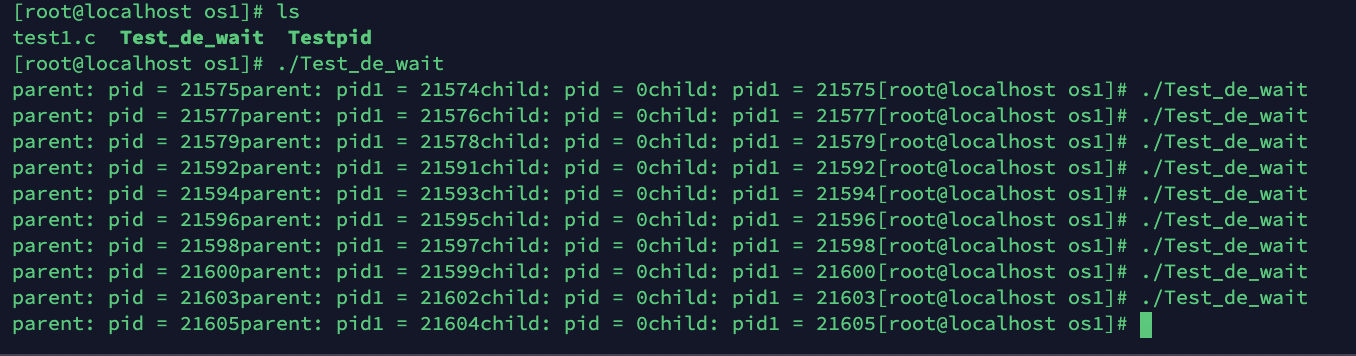
每次的child pid1均不相同 这是由于getpid操作返回当前pid值，每次执行时进程不同

每次的 parent pid 均不相同且 parent pid与child pid1相同 这是由于fork调用对于父进程来说，每次返回为子进程的pid值，所以子进程getpid与对于父进程返回相同

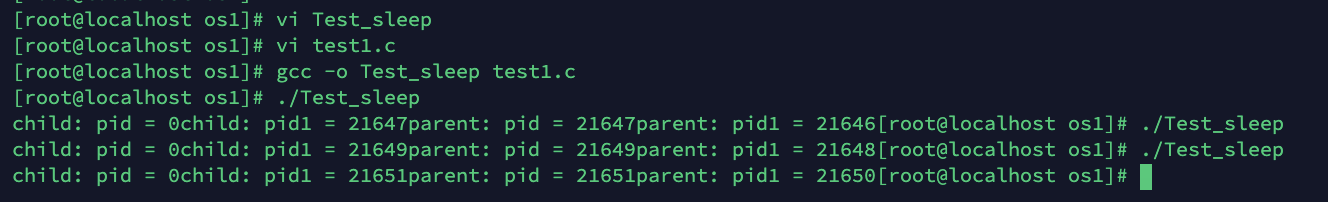
每次的parent pid1 均不相同 这是由于getpid操作返回当前pid值，每次执行时进程不同

而且由于父进程中有wait 将会等待子进程运行，所以输出结果不会改变

**去掉wait后运行截图为：**

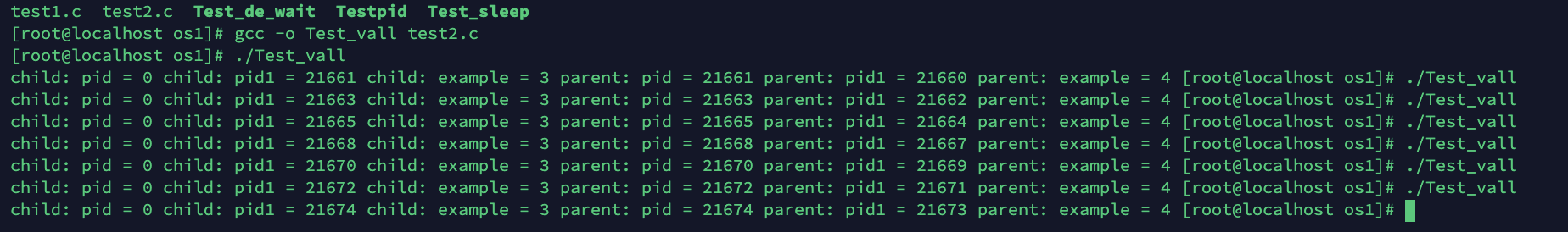
去掉wait后由于父进程不在等待子进程后，因此父子进程运行的顺序是随机的，因此parent可能出现在前，也可能出现在后无法确定，上图是由于运行次数有限所以一直parent一直在前

如果此时在parent进程上加上sleep运行顺序反转



此时child在前，是由于添加了sleep函数，此时父进程沉睡1s，所以子进程将会先运行

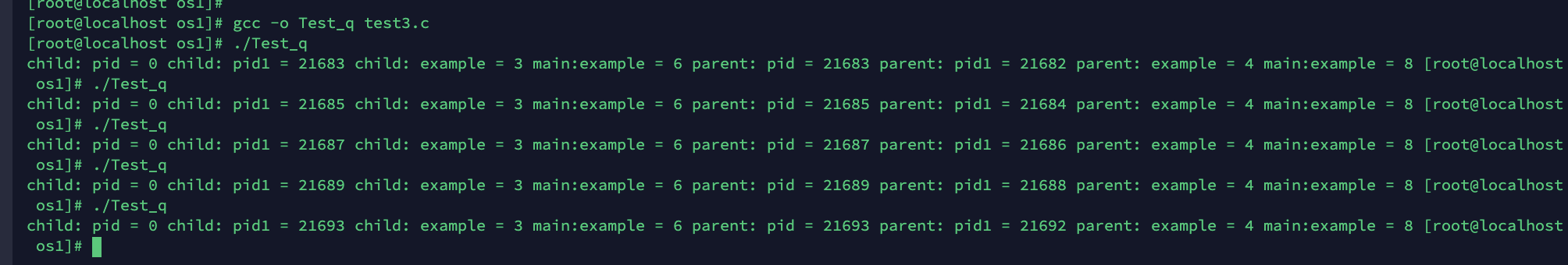
**父子进程对全局变量操作截图：**



通过添加example = 2 的全局变量，在子进程中**example = example + 1;**在父进程中有**example = example +2;**

根据结果有：子进程的**example = 3** 父进程的**example = 4**可知，及父子进程均是对**example = 2** 的初值进行操作的

故父子进程均将example作为自己单独的全局变量进行处理，父子进程的exampel各是独自的全局变量



若在return前继续将其输出有如下结果：

由此可知在父子进程的return前对example的操作子进程输出 example = 6，而父进程对example = 8 结合上面可知，父子进程对全局example的操作相当于父子进程有各自相互独立的全局变量分别处理，两者没有数据交流

**对子进程调用system函数及exec组函数**

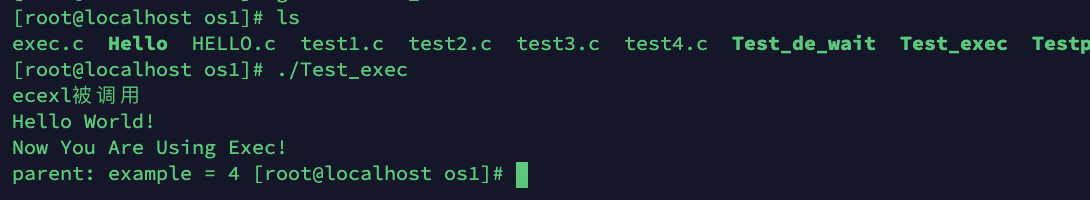
**System：**

在子进程中调用system函数，在system中嵌套echo函数

输出 ：**This is a system\_child process**

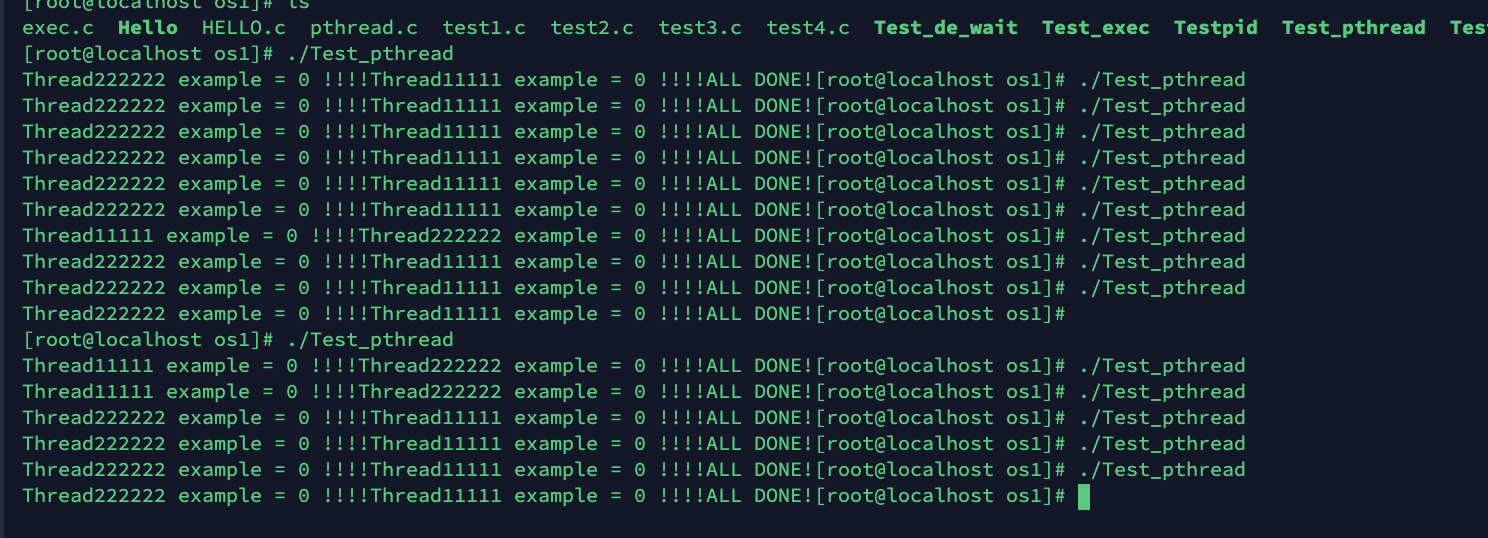
同时在输出之后继续执行输出**example = 3**

由此可见system’函数仅仅作为一个顺序执行中的一环，在system执行过程中首先fork出父子进程，父进程等待子进程的执行，子进程则调用exec函数去执行echo命令，所以在system函数fork的进程结束后将**继续执行子进程剩下的语句**

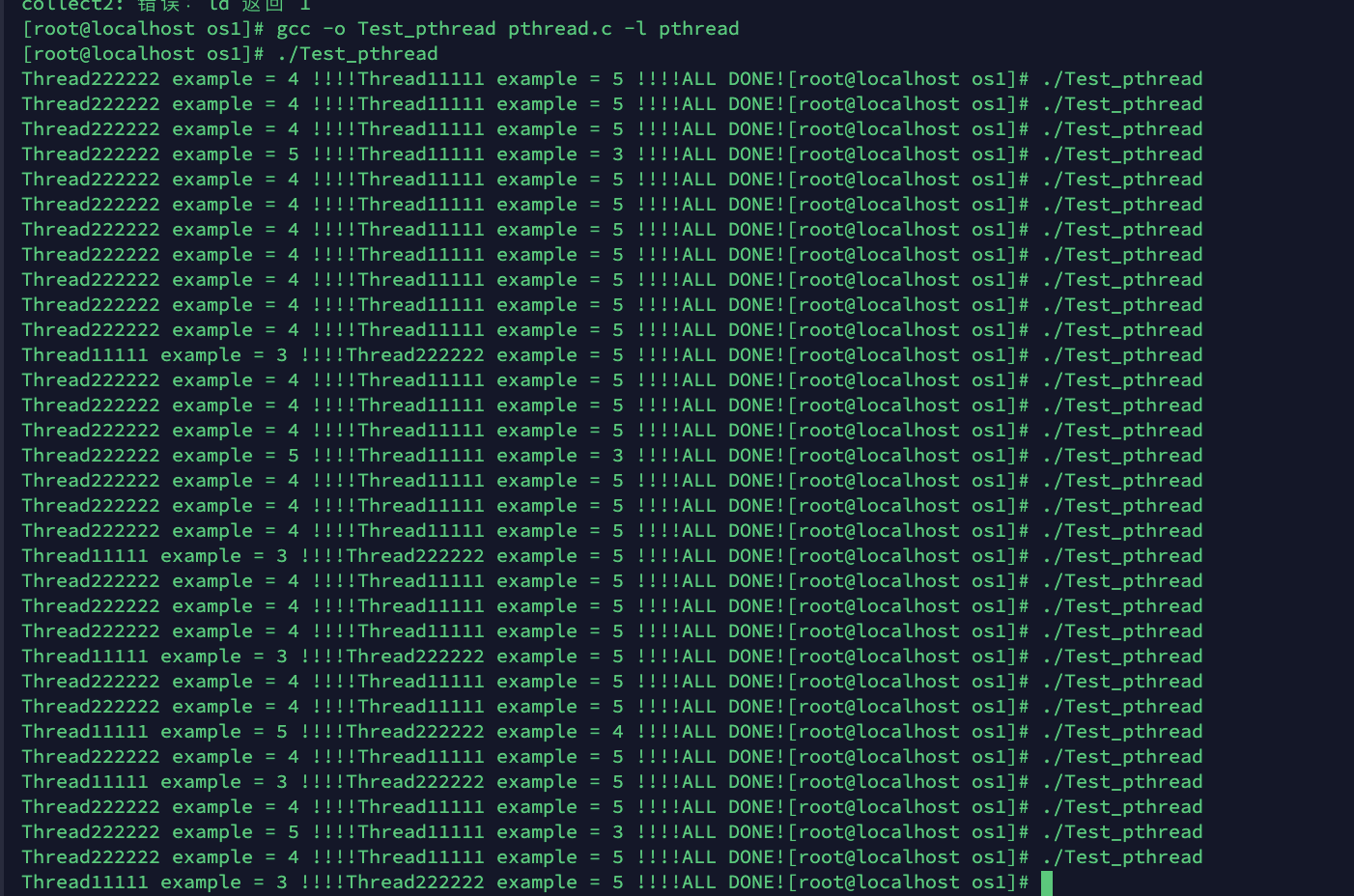
Exec：

通过调用execl来执行之前编译的Hello程序，输出Hello World，但是发现在Exec调用结束之后子进程剩下的语句不在执行，相当于通过execl将fork出来的子进程替换成了execl中执行的程序Hello，让原本的子进**程获得了重生。**

**Pthread创建两个线程进行信息交换：**

****

**通过pthread创建两个线程对同一个全局变量进行操作**

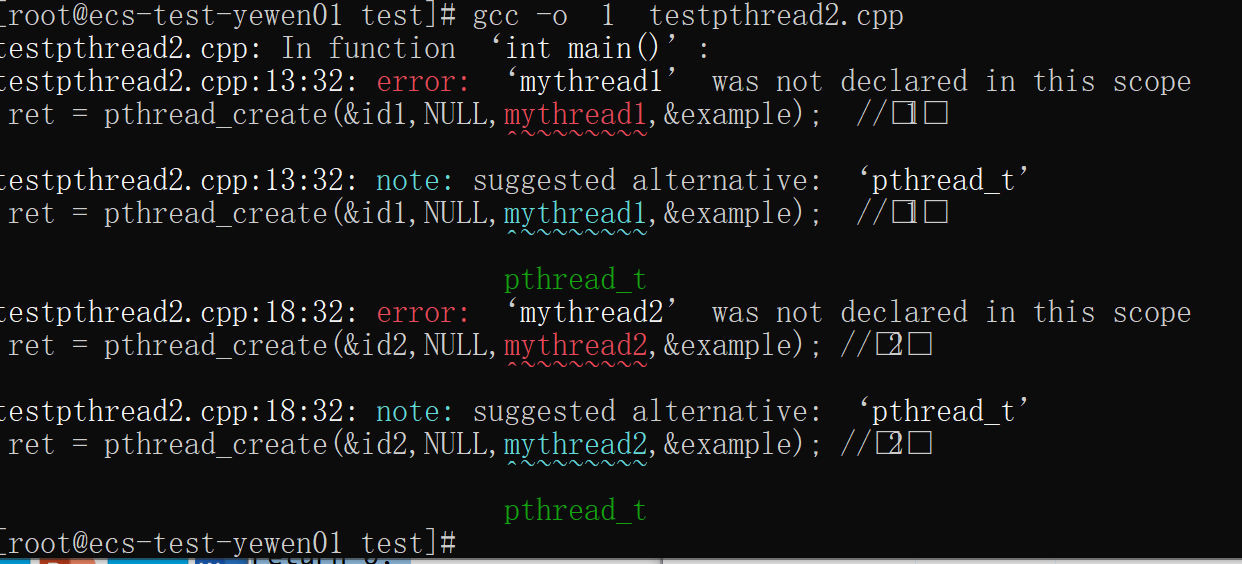
通过不加互斥的对全局变量进行操作，发现有时线程2先输出，有时候线程1先进行输出，而且有时候Thread多线程的输出**example输出错误，比如Thread1111 输出 example = 5 ；Thread2222 输出 example = 4,输出错误**

可见使用多线程可以对同一变量进行操作，但是如果不加互斥的话，顺序不一定，而且有时由于同时访问，产生错误，添加互斥后则输出一定

## 1.7实验总结

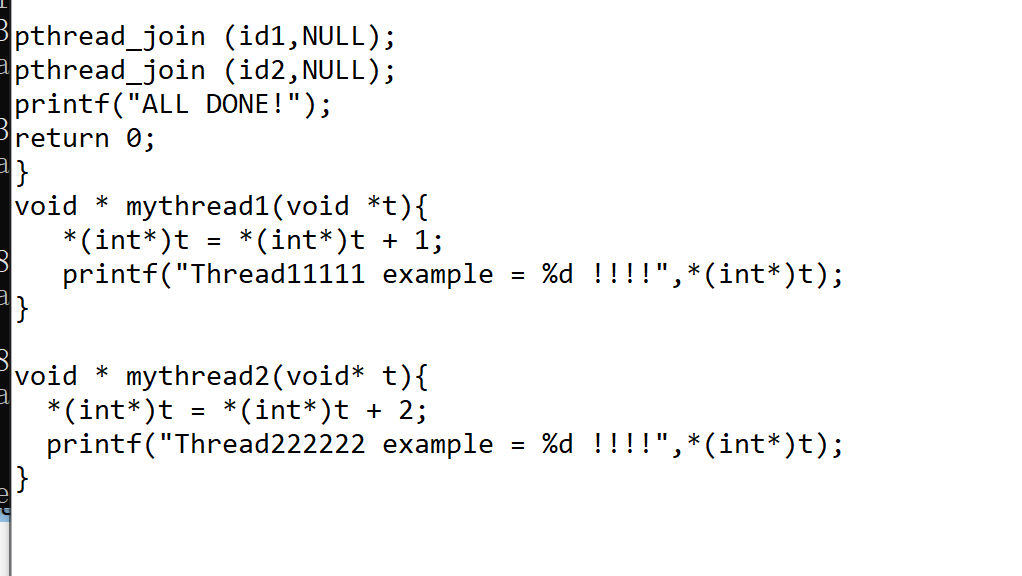
### 1.7.1实验中的问题与解决过程

### 1.1问题描述

在进行编译生成可执行文件时出现报错

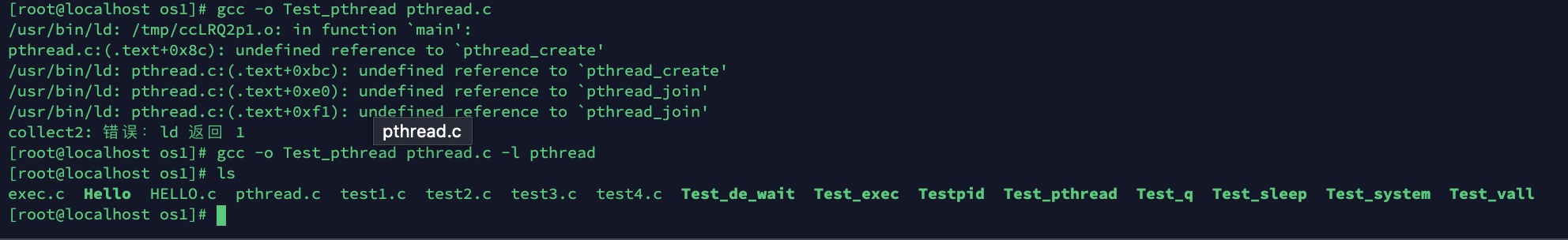
### 1.1解决过程：（网址，参考资料），具体解决方法

该问题的原因是由于函数写才了return 0 之后，之前未进行申明而出错



通过将函数移动到main函数之前即可解决

### 2.1问题描述

编译过程中出现未定义的**pthreadcreat**报错

### 2.2解决过程：

通过在编译操作是进行添加 -l pthread 编译时添加多线程库即可解决错误

### 3.1问题描述

出现空指针报错

### 3.2 解决过程

通过对函数的空指针进行替换为\*(int \*) 替换即可，将空指针明确替换成int\*型指针即可

### 1.7.2实验收获

1. 通过进程与线程实验我掌握并理解了fork调用的用法及返回值的区别
2. 通过进程的应用，我理解了wait(NULL)的用法。及等待子进程的执行
3. 在全局变量的实验我理解了父子进程具有相同的代码，但是对于同一个变量名的操作其实是针对与两个地址空间的操作
4. System调用是创建一个子进程执行system调用中的语句，调用结束后将会返回原先位置继续执行剩下语句
5. Exec组函数是将执行的内容对子进程内容进行替换，Exec结束后不在返回继续执行程序，因此相当于将子进程进行了替换
6. 通过Pthread创建线程时两个线程同时进行运行，此时当对同一个地址空间进行操作时需要进行互斥，否则将会出现错误

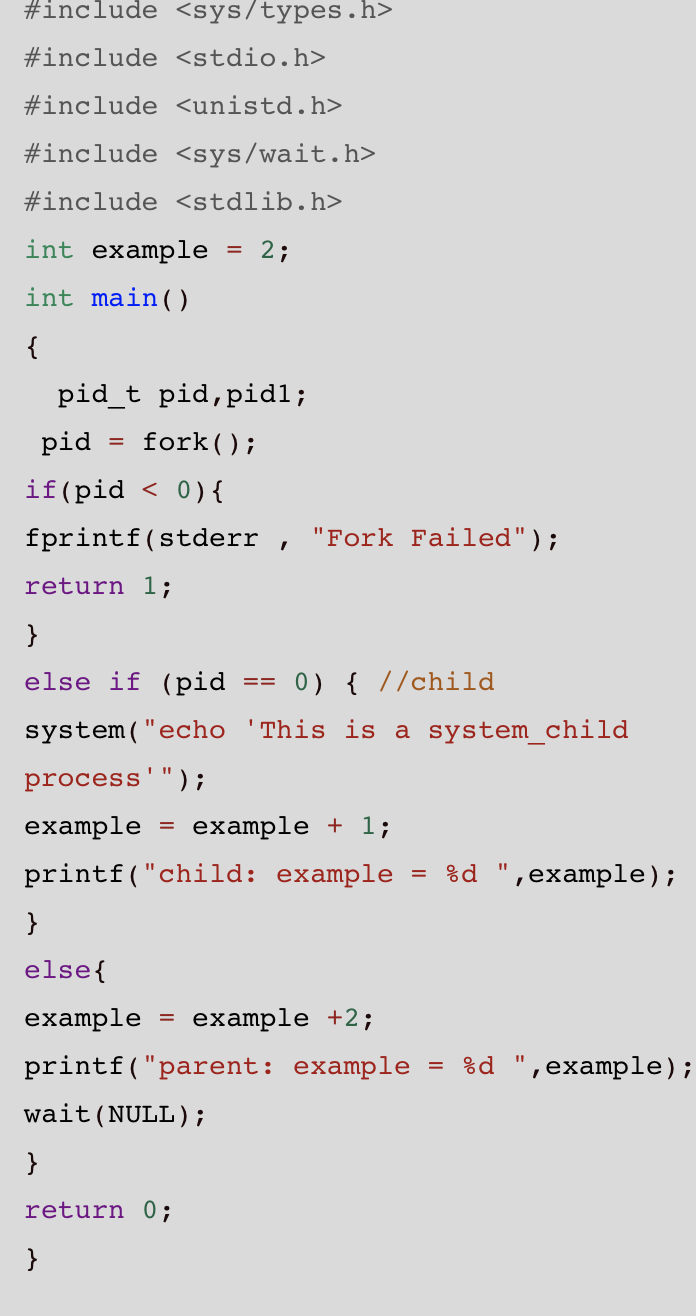
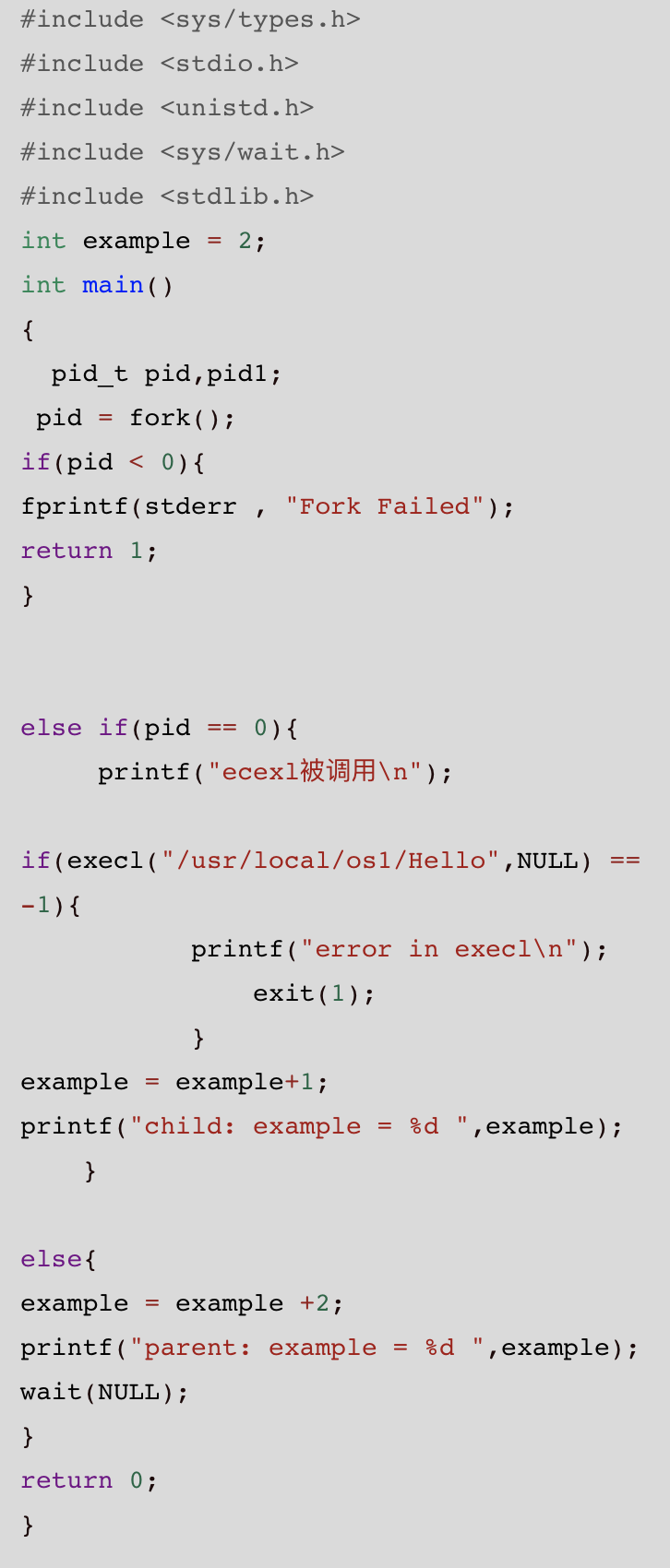
### 1.7.3意见与建议

建议可以在实现多线程时给定一个具体框架用于算法设计，比如提示是否需要互斥

## 1.8附件

### 1.8.1附件1 程序

### 





### 1.8.2附件2 Readme

# 

# 2 进程通信与内存分配

## 2.1实验目的

1. 了解软中断实现的原理
2. 学会软中断通信的过程，以及如何在程序中实现软中断通信
3. 理解进程及进程在调度执行和内存空间等方面的特点
4. 掌握在POSIX 规范中系统调用的功能和使用
5. 熟悉并掌握signal，alarm等各种形式的软中断通信
6. 深入理解进程管道通信的特点
7. 掌握管道通信的同步和互斥机制
8. 实现一个管道通信，并加入互斥机制
9. 深入理解内存分配管理的三种算法
10. 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
11. 掌握内存分配FF，BF，WF策略及实现的思路
12. 掌握内存回收过程及实现思路
13. 学会内存的紧缩策略的实现

## 2.2实验内容

**软中断通信：**

编制实现软中断通信的程序，通过系统调用signal()捕捉中断信号，父进程用系统调用kill()向两个子进程分别16和17软中断信号，子进程获得对应软中断信号输出特定信号

**管道通信：**

创建一个管道，用两个子进程互斥的在管道中实现通信

**内存分配与管理：**

建立三种内存分配的策略，通过设定一个固定大小的内存，通过不同的策略将不同大小的内存分配进内存，并管理空闲空间。

### 2.3实验思想

**软中断通信：**

创建一个父进程与两个子进程，可以通过父进程对两个子进程发送中断信号，或者通过键盘发送中断信号。子进程接受到中断信号后执行对应操作。

**管道通信：**

通过创建一个pipe管道，创建两个子进程进行管道与父进程之间进行管道的通信，在子进程将数据输入，在父进程将所输入的数据输出

**内存分配与管理：**

设置一个内存的总空间大小，通过指定内存的分配策略，同时管理进程链表与空闲链表，通过添加进程，删除进程进行来进行进程链表及空闲链表的更新，根据维护的空闲空间与占用的空间来确定此时是否可以进行进程内存空间的分配，同时制定一个紧缩策略，当紧缩策略可用时给进程分配进程空间同时更新系统所管理的进程及空闲链表。

## 2.4实验步骤

**软中断通信：**

通过父进程发出中断信号17，同时通过ctl +c 发出中断信号，分别在父进程发出中断信号前与信号后键盘ctl+c 发出中断信号多次运行看输出结果

在alarm软中断通信，分别在不进行任何操作下输出结果，在未执行完时键入ctl+c发出中断信号输出结果，针对不同结果分析

**管道通信：**

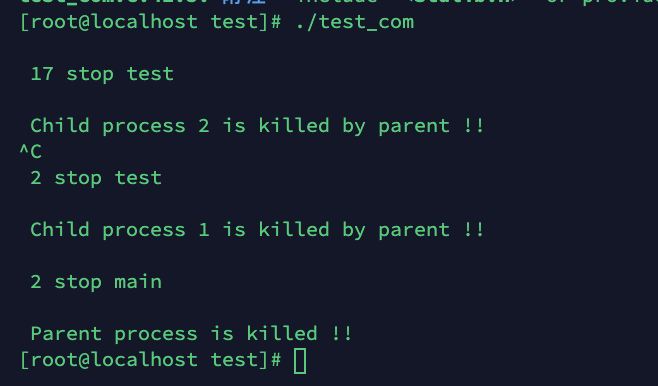
创建两个子进程往管道中分别写入信息，在父进程将写入消息读出，分别在有锁和无锁的条件下进行写入，读出操作，多次运行，分析两者输出结果的不同

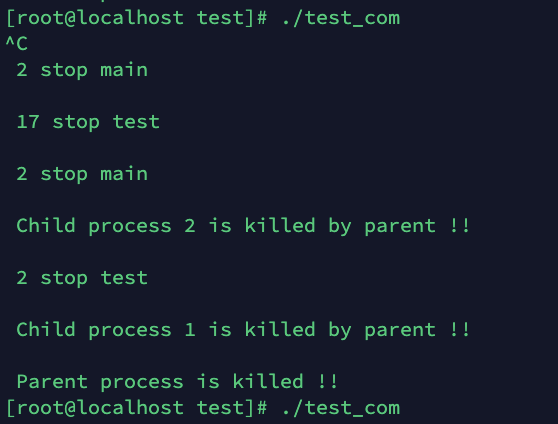
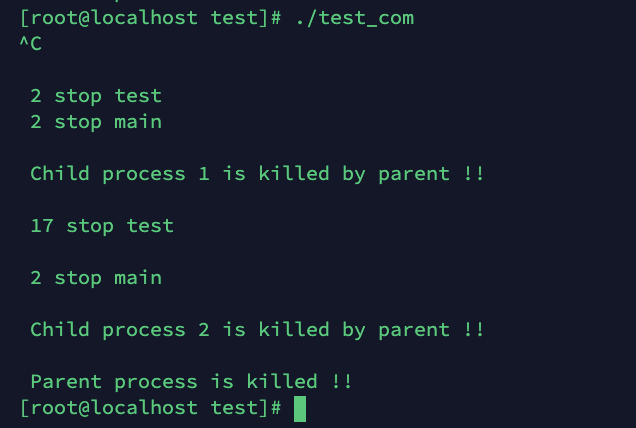
**内存分配与管理：**

创建一系列进程直到空间满，然后执行删除操作，同时修改分配操作，验证内存分配，删除是否正确，然后设置一个刚刚好大小的进行触发紧缩策略，验证紧缩策略的正确性

## 2.5程序运行初值及运行结果分析

**软中断通信：**

**5s后键入ctl+c：**

**5s前键入：**

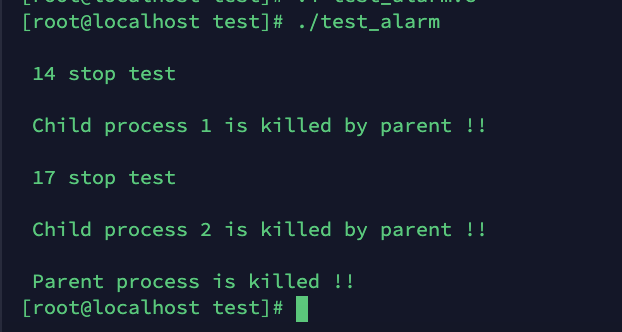
首先pid1,pid2均在pause处停止等待 父进程sleep(5)发出中断信号，process2停止

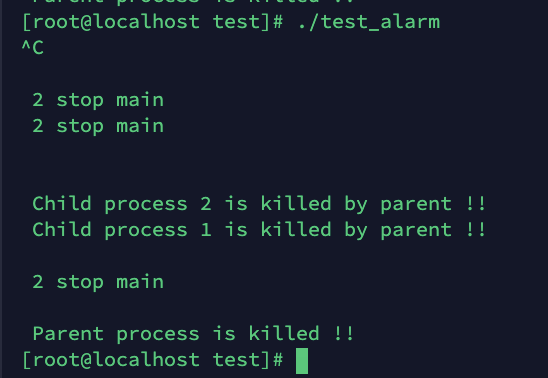
自己键入ctl + c process收到中断信号并打印

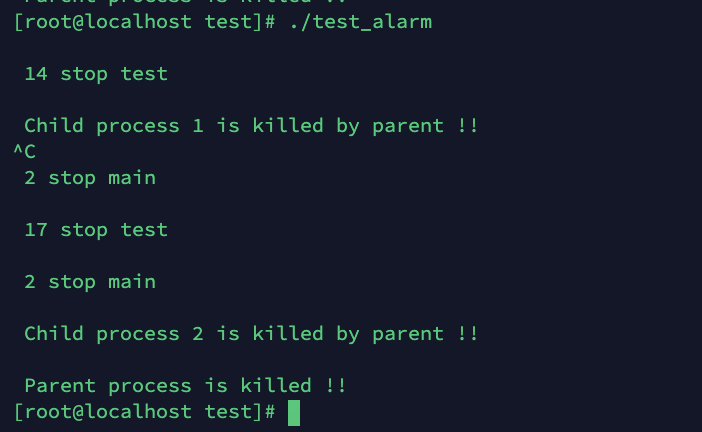
此时的ctl + c 被main中的signal收到打印

最后父进程终止**。**

**Alarm软中断通信：**

**不加操作：**

**执行ctl+c**

**在中间进行ctl+c：**

由于父进程未发出中断信号，子进程alarm未发出中断信号，由ctl+c发出中断信号，先中止子进程，后终止了父亲进程。

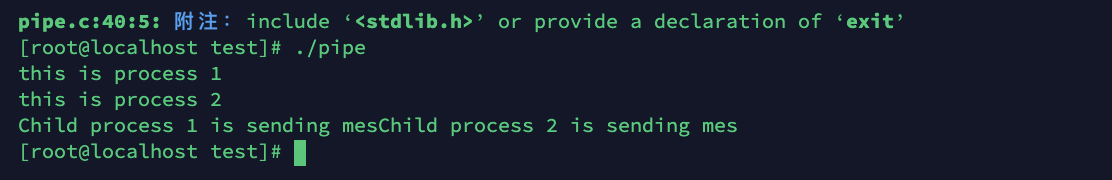
**不加操作ctrl+c时：**

此时alarm先发送信号中断子进程1，打印，然后父进程sleep完后发出中断信号,子进程2打印

**加入ctrl+c后：**

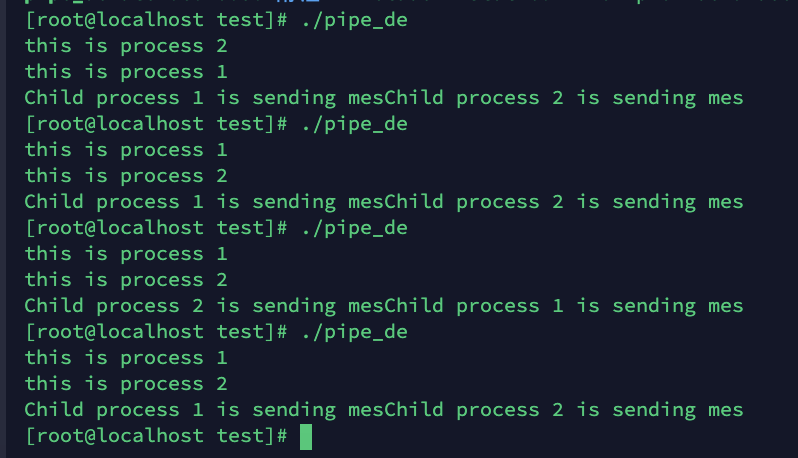
会触发进程前段signal，导致多次打印 2 stop main

**Pipe管道通信：**

**有锁：**

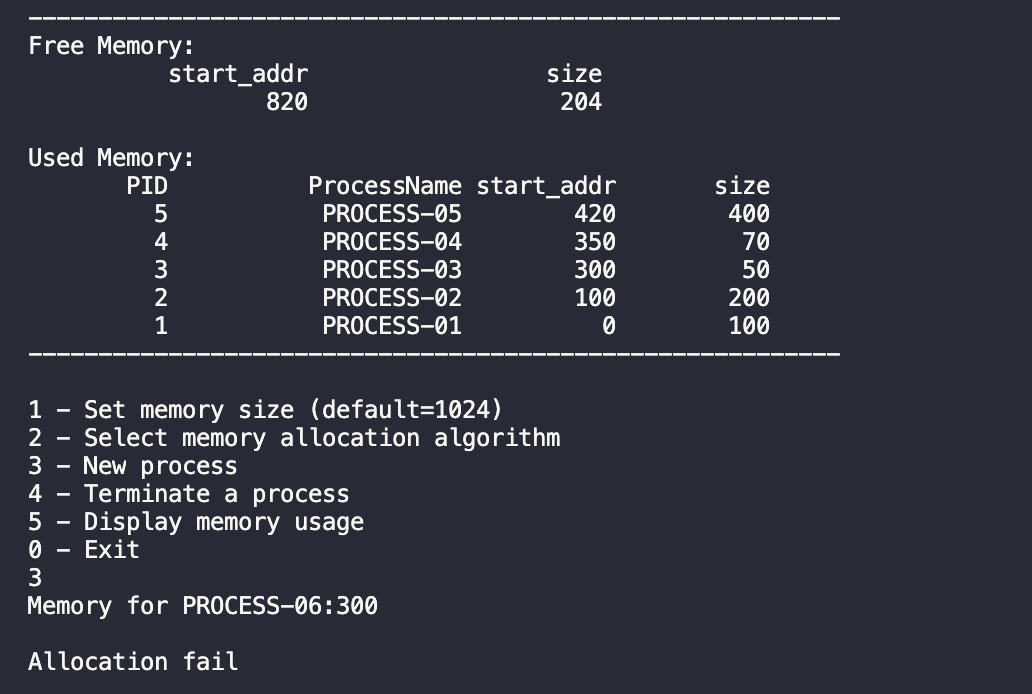
**在process1 前面添加sleep()后变换**

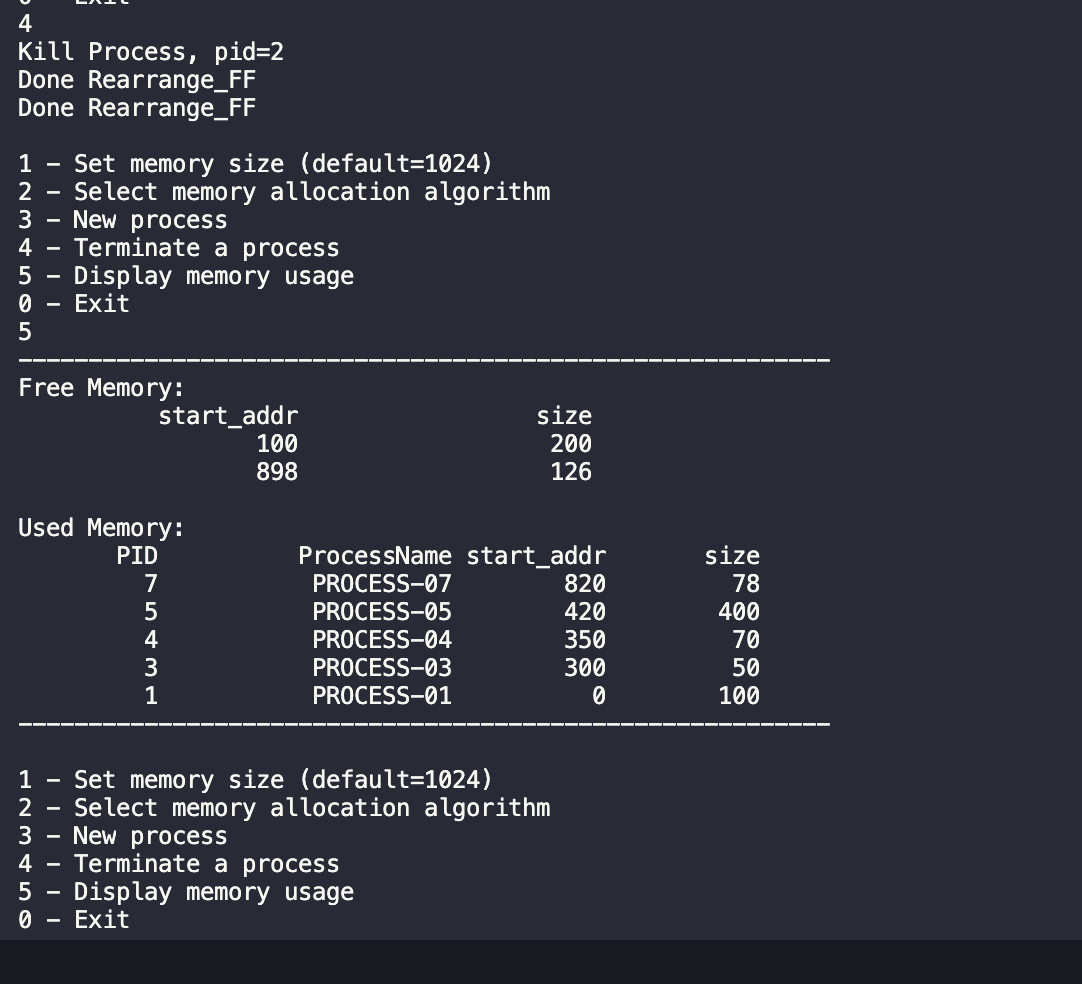
在有锁固定时此时输出顺序是确定的，不会改变，而且由于互斥，输出不会出错

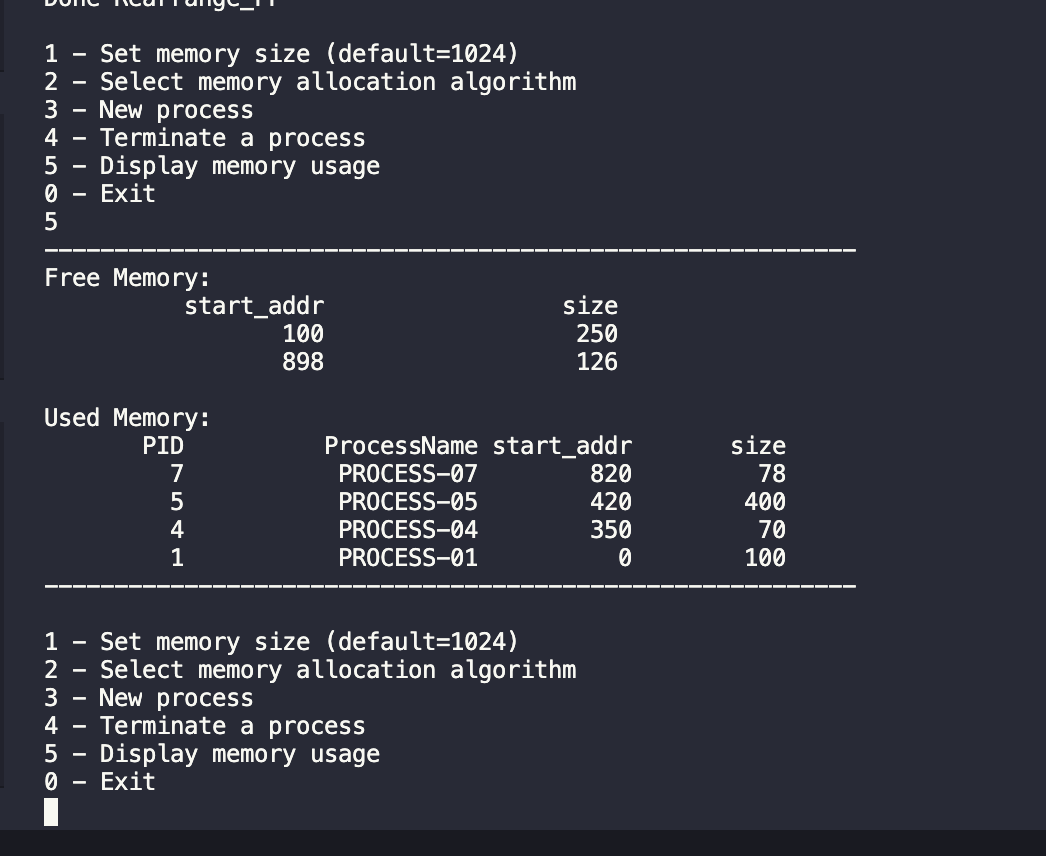
**无锁：**

在无锁是发现写入管道时顺序出现问题，在输出时有时会出现写入顺序和读出顺序的不一致，所以当无锁时写入，读出的次序得不到保证

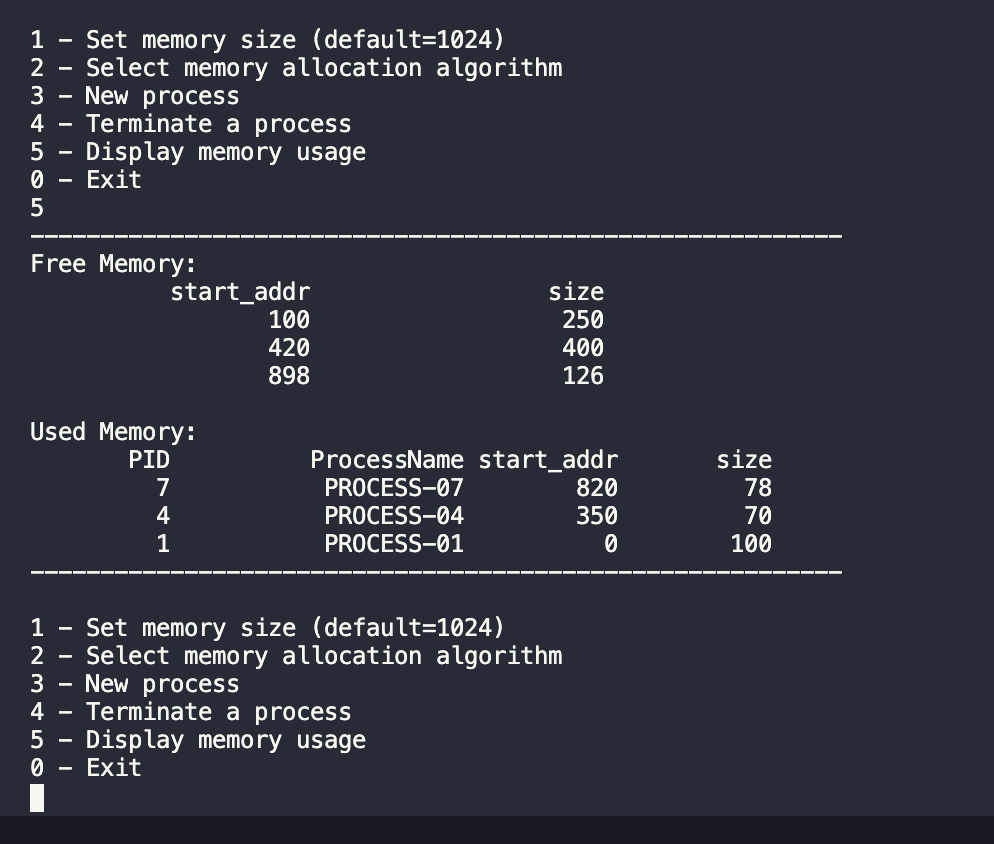
**内存分配：**

**1.多次创建进程知道不可创建为止输出Fail信号**

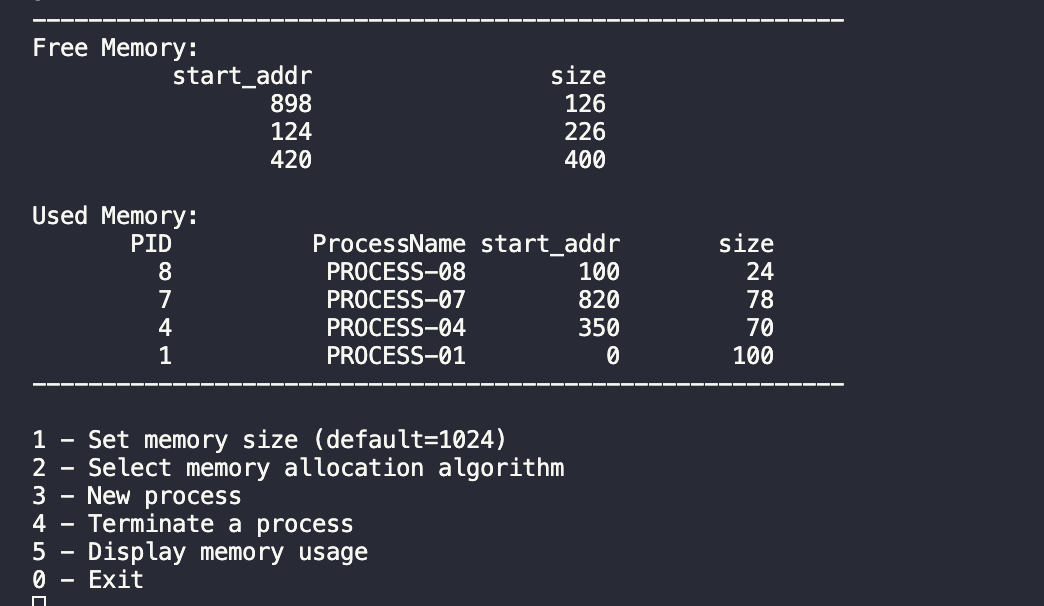
**2.终止进程pid = 2 后空闲链表更新**

**3.** **终止另一个相邻进程pid=3后，空闲相邻出合并**

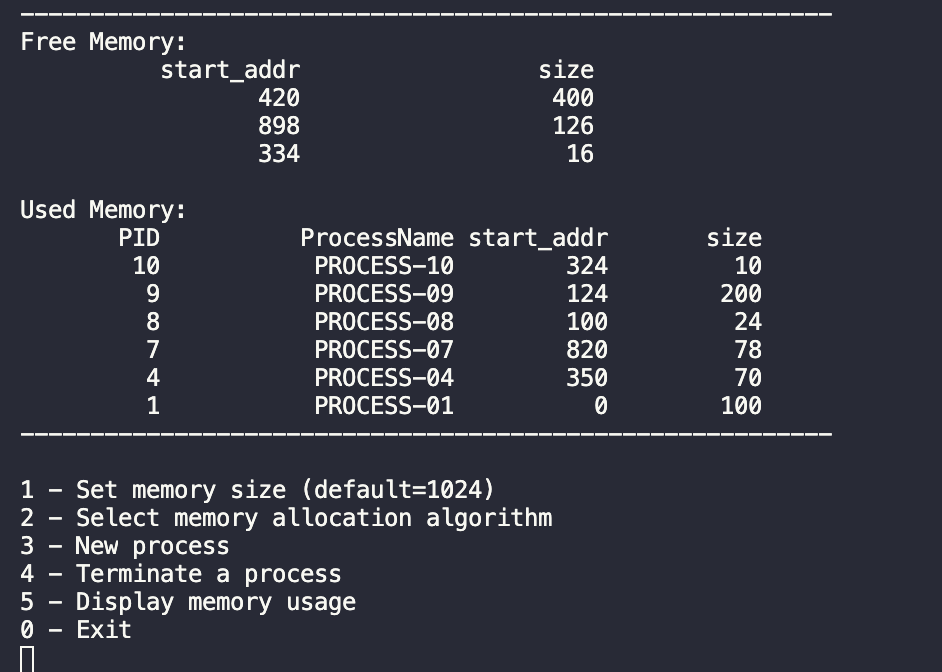
**4.** **再次终结一个进程 pid =5，出现了3个空闲节点**

****

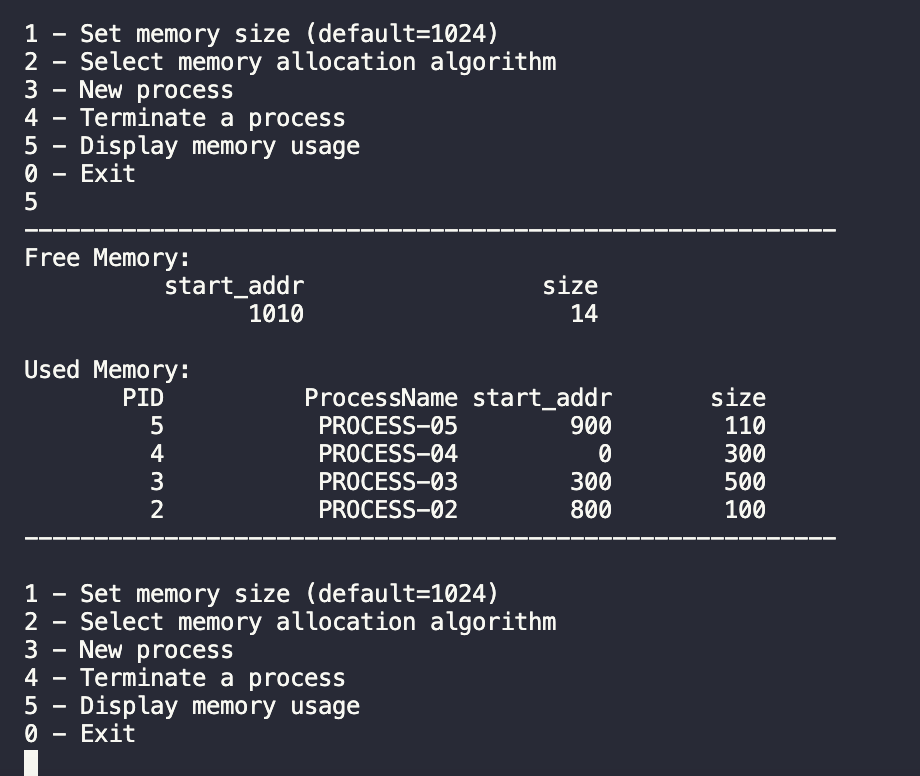
**5.** **创建进程pid =8，并修改策略最佳适应Best Fit，低地址空间在前**

****

**6.选择最差适应，空间从大到小**

****

**7.内存紧缩策略：**

**分配成功！**

**从以上所有功能均正常，内存分配系统设计成功**

## 2.6实验总结

### 2.6.1实验中的问题与解决过程

### 1.问题描述

紧缩策略在有空间时仍然无法对其进行分配空间

### 2.解决过程：

通过重新设置紧缩策略来解决，将所有分配进程重新进行连续的地址分配，最后剩下的空间分配给新进程，以此策略完成分配

### 2.3.2实验收获

1. 学会了软中断通信的实现方法，深刻理解了软中断的机制，学会了使用软中断通信在父子进程之间进行通信
2. 掌握了软中断通信中的实现具体细节，学会了使用signal及alarm及kill等在软中断通信中常用的调用
3. 掌握了管道通信中的同步互斥机制，理解并熟悉了管道通信的原理
4. 学会了自己设计一个管道用于父子进程间的管道通信，掌握了同步互斥的实现机制
5. 意识到了管道通信中没有同步互斥机制的后果
6. 掌握并理解了内存管理及分配背后的实现机制
7. 掌握内存分配FF，BF，WF策略及实现的思路
8. 掌握内存回收过程及实现思路
9. 可以自己设计一个内存分配的整体系统并且可以实现内存紧缩等不同策略
10. 对于OS系统中内存分配策略理解的更加透彻

### 2.3.3意见与建议

无

## 2.4附件

软中断：

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

void stop( int *signum*);

void stopmain(int *signum*);

main( ) {

int pid1, pid2; // 定义两个进程号变量

signal(SIGINT,stopmain); // 或者 signal(14,stop);

while((pid1 = fork( )) == -1); // 若创建子进程1不成功,则空循环

if(pid1 > 0) { // 子进程创建成功,pid1为进程号

while((pid2 = fork( )) == -1); // 创建子进程2

if(pid2 > 0) {

sleep(5);

// 杀死进程1发中断号16

kill(pid2, 17); // 杀死进程2

wait(0); // 等待第1个子进程1结束的信号

wait(0); // 等待第2个子进程2结束的信号

printf("\n Parent process is killed !!\n");

exit(0); // 父进程结束

}

else { //子进程

signal(17,stop); // 等待进程2被杀死的中断号17

pause();

printf("\n Child process 2 is killed by parent !!\n");

exit(0);

}

}

else { //pid1 = 0; 孩子进程

signal(SIGINT,stop); // 等待进程1被杀死的中断号16

pause();

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

exit(0);

}

}

void stop(int *signum*) {

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

}

void stopmain(int *signum*){

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

}

管道通信：

#include <stdio.h>

#include <signal.h>

#include <unistd.h>

#include <sys/types.h>

void stop( int *signum*);

void stopmain(int *signum*);

main( ) {

int pid1, pid2; // 定义两个进程号变量

signal(SIGINT,stopmain); // 或者 signal(14,stop);

while((pid1 = fork( )) == -1); // 若创建子进程1不成功,则空循环

if(pid1 > 0) { // 子进程创建成功,pid1为进程号

while((pid2 = fork( )) == -1); // 创建子进程2

if(pid2 > 0) {

sleep(10);

// 杀死进程1发中断号16

kill(pid2, 17); // 杀死进程2

wait(0); // 等待第1个子进程1结束的信号

wait(0); // 等待第2个子进程2结束的信号

printf("\n Parent process is killed !!\n");

exit(0); // 父进程结束

}

else { //子进程

signal(17,stop); // 等待进程2被杀死的中断号17

pause();

printf("\n Child process 2 is killed by parent !!\n");

exit(0);

}

}

else { //pid1 = 0; 孩子进程1

signal(SIGALRM,stop); // 等待进程1被杀死的中断号14

alarm(3);

pause();

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

exit(0);

}

}

void stop(int *signum*) {

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

}

void stopmain(int *signum*){

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

}

内存分配：

#include <stdio.h>

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<unistd.h>

#include<ctype.h>

#include<pthread.h>

#define PROCESS\_NAME\_LEN 32 /\*进程名长度\*/

#define MIN\_SLICE 10 /\*最小碎片的大小\*/

#define DEFAULT\_MEM\_SIZE 1024 /\*内存大小\*/

#define DEFAULT\_MEM\_START 0 /\*起始位置\*/

/\* 内存分配算法 \*/

#define MA\_FF 1

#define MA\_BF 2

#define MA\_WF 3

int mem\_size=DEFAULT\_MEM\_SIZE; /\*剩余内存大小\*/

int ma\_algorithm = MA\_FF; /\*当前分配算法\*/

static int pid = 0; /\*初始pid\*/

int flag = 0; //设置内存大小标志

/\*描述每一个空闲块的数据结构\*/ //空闲快的数据链表

struct *free\_block\_type*{

int size;

int start\_addr;

struct *free\_block\_type* \*next;

};

/\*指向内存中空闲块链表的首指针\*/

struct *free\_block\_type* \*free\_block=NULL;

/\*每个进程分配到的内存块的描述\*/

struct *allocated\_block*{ //已分配进程的链表

int size; //大小

int start\_addr; //开始的地址

char process\_name[PROCESS\_NAME\_LEN]; //名称

int pid; //pid值

struct *allocated\_block* \*next;

};

/\*进程分配内存块链表的首指针\*/

struct *allocated\_block* \*allocated\_block\_head = NULL;

//初始化空闲区

//初始化一整个大的空闲区

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 display\_menu(){

printf("\n");

printf("1 - Set memory size (default=%d)\n", DEFAULT\_MEM\_SIZE);

printf("2 - Select memory allocation algorithm\n");

printf("3 - New process \n");

printf("4 - Terminate a process \n");

printf("5 - Display memory usage \n");

printf("0 - Exit\n");

}

void set\_mem\_size(){ //可以自定义设置内存大小

int size;

if(flag!=0){ //防止重复设置

printf("Cannot set memory size again\n");

}

printf("Total memory size =");

scanf("%d", &size);

if(size>0) {

mem\_size = size;

free\_block->size = mem\_size;

}

flag=1;

}

void rearrange\_FF(){ //按照地址来进行FF排序，优先使用低地址

struct *free\_block\_type*\* p1,\* p2; //p1在前

p1 = free\_block;

int temp=0;

while(( p1->next) != NULL){

p2 = p1->next;

while(p2 != NULL){

if(p1->start\_addr > p2->start\_addr ) { //交换低地址在前

temp = p1->size;

p1->size = p2->size;

p2->size = temp;

temp = p1->start\_addr;

p1->start\_addr = p2->start\_addr;

p2->start\_addr = temp;

}

p2 = p2->next;

}

p1 = p1->next;

}

printf("Done Rearrange\_FF\n");

}

void rearrange\_BF(){ //空闲分区按照从小到大的顺序排序

struct *free\_block\_type*\* p1,\* p2; //p1在前

p1 = free\_block;

int temp = 0;

while(p1->next != NULL)

{

p2 = p1->next;

while(p2 != NULL){ //空间从小到大排序

if(p1->size > p2->size){

temp = p1->size;

p1->size = p2->size;

p2->size = temp;

temp = p1->start\_addr;

p1->start\_addr = p2->start\_addr;

p2->start\_addr = temp;

}

p2 = p2->next;

}

p1 = p1->next;

}

printf("Done Rearrange\_BF\n");

}

void rearrange\_WF() {

struct *free\_block\_type*\* p1,\* p2; //p1在前

p1 = free\_block;

int temp;

while(p1->next != NULL)

{

p2 = p1->next;

while(p2 != NULL){ //空间从大到小排序

if(p1->size < p2->size){

temp = p1->size;

p1->size = p2->size;

p2->size = temp;

temp = p1->start\_addr;

p1->start\_addr = p2->start\_addr;

p2->start\_addr = temp;

}

p2 = p2->next;

}

p1 = p1->next;

}

printf("Done Rearrange\_WF\n");

}

void rearrange(int *algorithm*){ //整个的调用

switch(*algorithm*){

case MA\_FF: rearrange\_FF(); break;

case MA\_BF: rearrange\_BF(); break;

case MA\_WF: rearrange\_WF(); break;

}

}

void set\_algorithm(){//用户自定义算法

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;

//按指定算法重新排列空闲区链表

rearrange(ma\_algorithm);

}

/\*struct allocated\_block{ //已分配进程的链表

int size; //大小

int start\_addr; //开始的地址

char process\_name[PROCESS\_NAME\_LEN]; //名称

int pid; //pid值

struct allocated\_block \*next;

};

\*/

int allocate\_mem(struct *allocated\_block* \**ab*){

int flag\_m = 0;

struct *free\_block\_type* \*fbt, \*pre;

int request\_size=*ab*->size;

fbt = pre = free\_block; //指向空闲头指针

struct *allocated\_block* \*p1,\*p2;

p1 = p2 =allocated\_block\_head;

struct *free\_block\_type* \*p3,\*p4;

p3 = p4 = free\_block;

while(fbt != NULL){ //寻找空闲节点

//找到节点

if(fbt->size >= request\_size){ //此时有空间可以给其分配

if(fbt->size >= request\_size + MIN\_SLICE){

flag\_m = 1;//内存剩余空间则够大则分配

mem\_size = mem\_size - request\_size;

*ab*->start\_addr = fbt->start\_addr; //初始位置处理完毕

fbt->size -= request\_size;

fbt->start\_addr = fbt->start\_addr + request\_size; //空闲处理完毕

break;

}

else if(fbt->size <= request\_size +MIN\_SLICE && fbt->size > request\_size){ //此时剩余空间较小一起分配

flag\_m = 1;

mem\_size = mem\_size - fbt->size; //空间

*ab*->start\_addr = fbt->start\_addr; //初始位置处理完毕

pre->next = fbt->next;

//由于删除节点所以pre应该指向

free(fbt); //此空闲节点释放

break;

}

}

pre = fbt;

fbt = fbt->next;

}

//遍历结束任然未找到空闲区

if(fbt == NULL){

if(mem\_size > request\_size){ //剩余足够

p2 = p1->next; //p1 =head;

p1->start\_addr = 0;

while(p2 != NULL){

//首先将进程链合并

p2->start\_addr = p1->start\_addr + p1->size;

p1 = p2;

p2 = p2->next;

}

//此时将空闲区整体合并

free\_block->start\_addr=(struct *free\_block\_type*\*) malloc(sizeof(struct *free\_block\_type*));

free\_block->next =NULL;

free\_block->size = mem\_size;

free\_block->start\_addr = p1->start\_addr+p1->size;

//开始分配

*ab*->start\_addr = free\_block->start\_addr;

*ab*->next = allocated\_block\_head;

allocated\_block\_head = *ab*;

free\_block->start\_addr = free\_block->start\_addr + request\_size;

free\_block->size -=request\_size;

mem\_size = mem\_size - request\_size;

}

}

if(flag\_m == 1){

rearrange(ma\_algorithm); //进行重整

return 1;

}

else return -1;

//根据当前算法在空闲分区链表中搜索合适空闲分区进行分配，分配时注意以下情况：

// 1. 找到可满足空闲分区且分配后剩余空间足够大，则分割

// 2. 找到可满足空闲分区且但分配后剩余空间比较小，则一起分配

// 3. 找不可满足需要的空闲分区但空闲分区之和能满足需要，则采用内存紧缩技术，进行空闲分区的合并，然后再分配

// 4. 在成功分配内存后，应保持空闲分区按照相应算法有序

// 5. 分配成功则返回1，否则返回-1

}

int new\_process(){ //进程创建

struct *allocated\_block* \*ab; //进程链表

int size; int ret;

ab=(struct *allocated\_block* \*)malloc(sizeof(struct *allocated\_block*));

if(!ab) exit(-5);

ab->next = NULL;

pid++; //进程唯一标识符

sprintf(ab->process\_name, "PROCESS-%02d", pid); //进程名

ab->pid = pid;

printf("Memory for %s:", ab->process\_name);

scanf("%d", &size); //设置大小

if(size>0) ab->size=size;

ret = allocate\_mem(ab); /\* 从空闲区分配内存，ret==1表示分配ok\*/

if((ret==1) &&(allocated\_block\_head == NULL)){

allocated\_block\_head=ab;

return 1; }

/\*分配成功，将该已分配块的描述插入已分配链表\*/

else if (ret==1) {

ab->next=allocated\_block\_head;

allocated\_block\_head=ab;

return 2; }

else if(ret==-1){ /\*分配不成功\*/

printf("\nAllocation fail\n");

free(ab);

return -1;

}

return 3;

}

int free\_mem(struct *allocated\_block* \**ab*){

mem\_size += *ab*->size;

int algorithm = ma\_algorithm;

struct *free\_block\_type* \*fbt=NULL, \*pre=NULL, \*work=NULL;

fbt=(struct *free\_block\_type*\*) malloc(sizeof(struct *free\_block\_type*));

if(!fbt) return -1;

//将释放节点插入到队列首部

fbt->size = *ab*->size;

fbt->start\_addr = *ab*->start\_addr;

fbt->next = free\_block;

free\_block =fbt; //差到头节点

//按照地址从低到高排列

rearrange(MA\_FF);

//此时检查合并分区

fbt = free\_block;

while(fbt != NULL){

work = fbt->next;

if(work != NULL){

if(fbt->start\_addr + fbt->size ==work->start\_addr) //可合并

{

fbt->size = fbt->size + work->size;

fbt->next = work->next;

free(work);

work = NULL;

}}

fbt = fbt->next;

}

rearrange(algorithm); //按照当前算法重新排序

// 进行可能的合并，基本策略如下

// 1. 将新释放的结点插入到空闲分区队列末尾

// 2. 对空闲链表按照地址有序排列

// 3. 检查并合并相邻的空闲分区5

// 4. 将空闲链表重新按照当前算法排序

return 1;

}

int dispose(struct *allocated\_block* \**free\_ab*){

struct *allocated\_block* \*pre, \*ab;

if(*free\_ab* == allocated\_block\_head) { /\*如果要释放第一个节点\*/

allocated\_block\_head = allocated\_block\_head->next;

free(*free\_ab*);

return 1;

}

pre = allocated\_block\_head;

ab = allocated\_block\_head->next;

while(ab!=*free\_ab*){ pre = ab; ab = ab->next; }

pre->next = ab->next;

free(ab);

return 2;

}

struct *allocated\_block* \*find\_process(int *pid*){

struct *allocated\_block* \*p;

p=allocated\_block\_head;

while(p){

if(p->pid == *pid*){

return p;

break;

}

else p=p->next;}

}

kill\_process(){

struct *allocated\_block* \*ab;

int pid;

printf("Kill Process, pid=");

scanf("%d", &pid);

ab=find\_process(pid);

if(ab!=NULL){

free\_mem(ab); /\*释放ab所表示的分配区\*/

dispose(ab); /\*释放ab数据结构节点\*/

}

}

display\_mem\_usage(){

struct *free\_block\_type* \*fbt=free\_block;

struct *allocated\_block* \*ab=allocated\_block\_head;

if(fbt==NULL) return(-1);

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

/\* 显示空闲区 \*/

printf("Free Memory:\n");

printf("%20s %20s\n", " start\_addr", " size");

while(fbt!=NULL){

printf("%20d %20d\n", fbt->start\_addr, fbt->size);

fbt=fbt->next;

}

/\* 显示已分配区 \*/

printf("\nUsed Memory:\n");

printf("%10s %20s %10s %10s\n", "PID", "ProcessName", "start\_addr", " size");

while(ab!=NULL){

printf("%10d %20s %10d %10d\n", ab->pid, ab->process\_name, ab->start\_addr, ab->size);

ab=ab->next;

}

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

return 0;

}

void do\_exit(){

struct *free\_block\_type* \*p1 = free\_block;

struct *free\_block\_type* \*pre = p1->next;

while(pre != NULL){

free(p1);

p1 = pre;

pre = pre->next;

}

free(p1);

struct *allocated\_block* \*p2 = allocated\_block\_head;

struct *allocated\_block* \*pre1 =p2->next;

while(pre1 != NULL){

free(p2);

p2 = pre1;

pre1 = pre1->next;

}

free(p2);

exit(0);

}

main(){

char choice;

pid=0;

free\_block = init\_free\_block(mem\_size); //初始化空闲区

while(1) {

display\_menu(); //显示菜单

fflush(stdin);

choice=getchar(); //获取用户输入

switch(choice){

case '1': set\_mem\_size(); break; //设置内存大小

case '2': set\_algorithm();flag=1;break;//设置算法

case '3': new\_process(); flag=1;break;//创建新进程

case '4': kill\_process(); flag=1;break;//删除进程

case '5': display\_mem\_usage(); flag=1; break; //显示内存使用

case '0': do\_exit(); exit(0); //释放链表并退出

default: break; } }

}

### 1.8.2附件2 Readme

# 3 EXT2文件系统

## 3.1实验目的

通过一个简单的文件系统的设计，加深理解文件系统内部的实现原理

## 3.2实验内容

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

## 3.3实验思想

根据EXT2文件系统设置文件系统的结构，在文件系统中设置一系列的结构

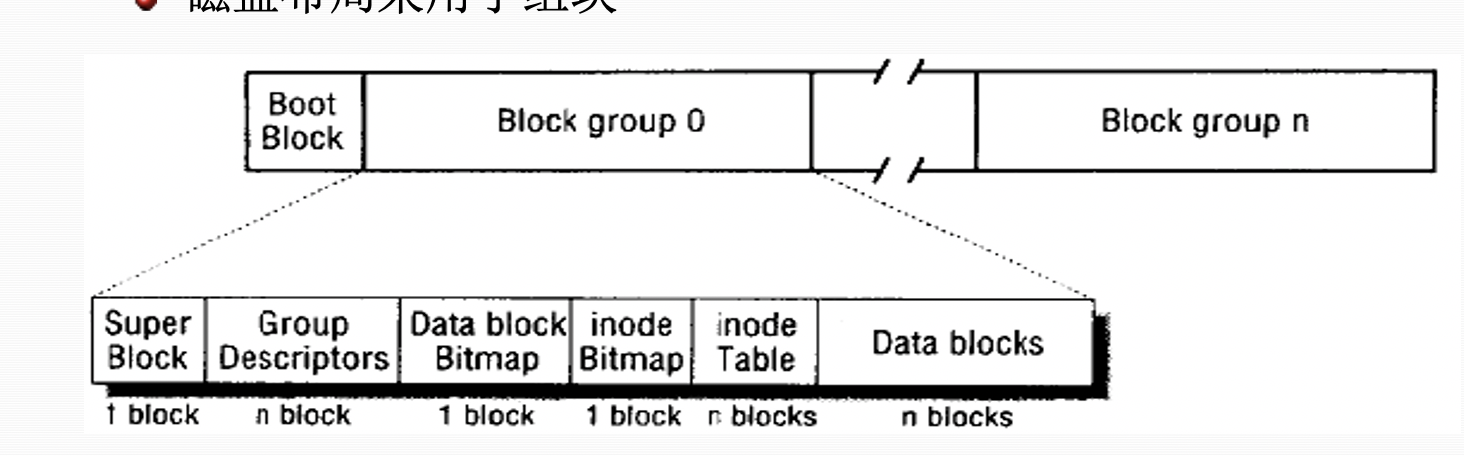
1.SuperBlock中定义了一些文件系统的初始信息，比如inode数量等。

2.组描述符中定义了系统的文件名，起始块号，索引位图起始等

3.Bata block的bitmap 用来标示哪些数据块没有使用，用来分配空间

4.Inode的bitmap用来标示哪些inode已经使用过了，用来释放与分配inode

5.Inode中，每个索引节点对应一个文件或是目录，在inode存储着文件的一些列参数，包括文件类型，创建时间，访问时间，修改时间，文件所占的块数及数据块的指针，通过inode节点可以找到文件的存储空间

通过以上我们可以得到其中结构如图：

接下来为算法结构的设计：

1. 删除文件或目录：根据inode号指向的数据块的到在数据及索引bitmap中的位置置为0，同时索引节点指向0数据块，表示清空
2. 5.open及close的操作：设置一个打开文件表实现，用一个数组来记录打开文件
3. 6.write操作：通过dir找到inode，通过inode找到数据块，查看打开文件表后即可对指向的数据块进行操作

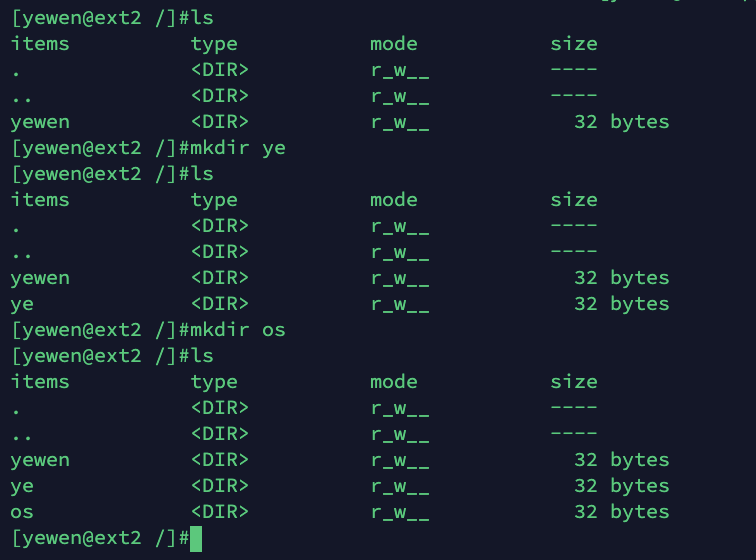
4. 创建文件：首先从DataBlock为其分配一个新块用来存储目录项，同时在inode的位图中为其查找一个可分配的空间分配inode同时根据文件大小计算出所分的数据块，在data bitmap中查找连续的这么多块后为其分配数据块并将数据块写入inode中同时更新bitmap位图

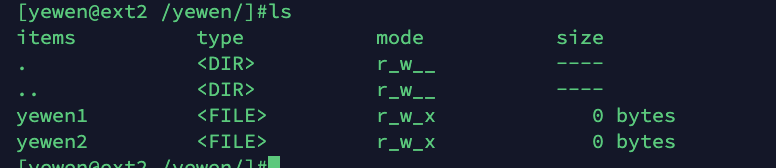
5. 创建目录：同理按照文件创建的方法，不过在inode指向数据块后为其写入两个目录项一个是当前目录“.”,及上一级目录”..”

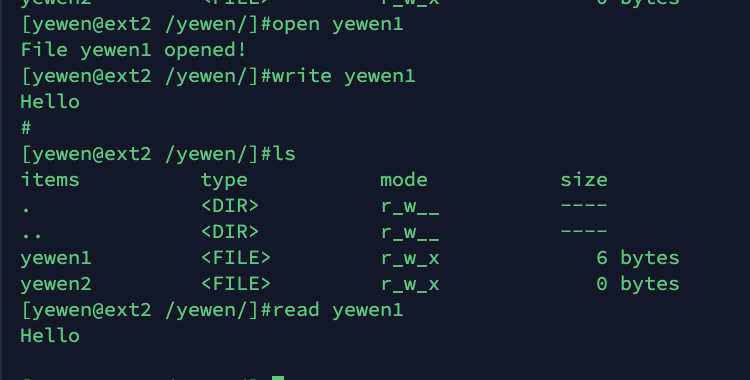
6. cd操作：首先找到当前目录的Inode号，block号，首先判断是否为 “.”及当前目录，若是则不进行操作，如果为“..”则为返回上一级目录修改current\_path中的值为当前输入的char[]temp即可，如果前两者都不是则将当前的current\_path+”/”+char[]temp及表示当前所进入的目录

7.quit操作：直接exit即可

## 3.4实验步骤及运行结果分析

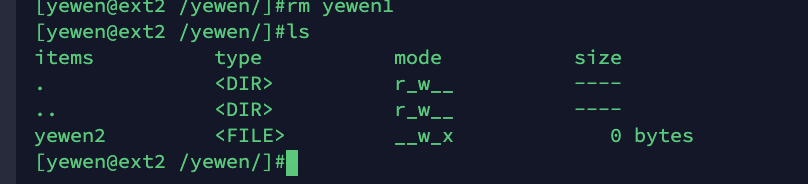
1.多次创建目录mkdir及ls

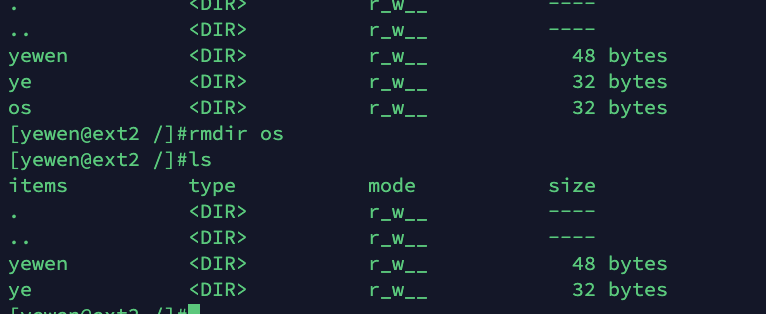
2.cd命令进入目录及touch文件

文件的写入write操作，打开open写入open表后进行操作

文件模式位转换

**删除文件：**

****

**删除目录：**

**由此可见文件系统的基本功能已经实现！**

## 3.5实验总结

### 3.5.1实验中的问题与解决过程

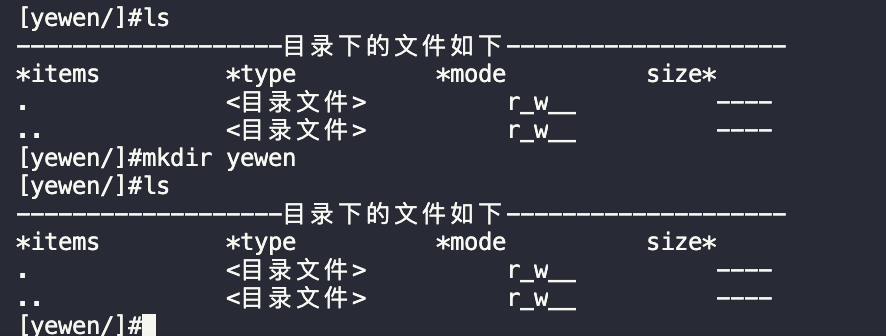
### 1.1问题描述

出现断错误（核心已转储）

### 1.2解决过程：

通过修改文件的模式位，由于无法对于FS文件进行操作，修改其模式位=为可操作即可完成

### 2.1问题描述

在VScode上无法对其进行写入操作. 

### 2.2解决过程：

由于inode中索引结构每次指向同一文件到导致，通过修改索引位分配节点即可

### 3.5.2实验收获

1. 掌握了文件系统的基本实现方法
2. 掌握并熟悉了EXT2文件系统的初始结构
3. 掌握并熟悉了文件系统中通过目录找inode然后找数据块的查找方法
4. 掌握了文件系统中需要设计的数据结构
5. 掌握并熟悉了文件系统实现中所需要定义的结构
6. 掌握并透彻理解了文件系统操作内部的实现方法

### 3.5.3意见与建议

无

### 3.4附件

Cd函数：

// 进入某个目录，实际上是改变当前路径

void cd(char *tmp*[100]) {

unsigned short i, j, k, flag;

flag = research\_file(*tmp*, 2, &i, &j, &k); //找到当前文件的Inode号，block号，目录号

if (flag) {

current\_dir = i;

//返回到上一次目录

if (!strcmp(*tmp*, "..") && dir[k - 1].name\_len) /\* 到上一级目录且不是..目录

{

current\_path[strlen(current\_path) - dir[k - 1].name\_len - 1] = '\0';

current\_dirlen = dir[k].name\_len; //名字长度

}

else if (!strcmp(*tmp*, ".")) {

return;

}

else if (strcmp(*tmp*, "..")) // cd 到上一级目录

{

current\_dirlen = strlen(*tmp*); //将其链接到当前目录

strcat(current\_path, *tmp*);

strcat(current\_path, "/");

}

}

else {

printf("The directory %s not exists!\n", *tmp*);

}

}

Touch操作

void touch(char *tmp*[100], int *type*) {

unsigned short no, i, j, k, flag;

reload\_inode\_entry(current\_dir);

if (!research\_file(*tmp*, *type*, &i, &j, &k)) {//寻找文件

flag = 1;

inode\_area[0].i\_block[inode\_area[0].i\_blocks] = alloc\_block();

inode\_area[0].i\_blocks++;

reload\_dir(inode\_area[0].i\_block[inode\_area[0].i\_blocks - 1]);

no = dir[0].inode = get\_inode();

dir[0].name\_len = strlen(*tmp*);

dir[0].file\_type = *type*;

strcpy(dir[0].name, *tmp*);

//初始化新块其他项目为0

for (flag = 1; flag < 32; flag++) {

dir[flag].inode = 0;

}

update\_dir(inode\_area[0].i\_block[inode\_area[0].i\_blocks - 1]);

inode\_area[0].i\_size += 16;

update\_inode\_entry(current\_dir);

//将新增文件的inode节点初始化

dir\_prepare(no, strlen(*tmp*), *type*);

}

else {

printf("File has already existed!\n");

}

}

Rm:

// 删除文件 通过删除位图，更新inode中数据块 目录指针指向0

void rm(char *tmp*[100]) {

unsigned short i, j, k, m, n, flag;

m = 0;

flag = research\_file(*tmp*, 1, &i, &j, &k);

if (flag) {

flag = 0;

entry(i); // 加载删除文件 inode

//删除文件对应的数据块

while (m < inode\_area[0].i\_blocks) {

rm\_block(inode\_area[0].i\_block[m++]);

}

inode\_area[0].i\_blocks = 0;

inode\_area[0].i\_size = 0;

remove\_inode(i);

// 更新父目录

entry(current\_dir);

reload\_dir(inode\_area[0].i\_block[j]);

dir[k].inode = 0; //删除inode节点

update\_dir(inode\_area[0].i\_block[j]);

inode\_area[0].i\_size -= 16;

up\_block(current\_dir);

}

}

Openfile:

//在open表里面查询是否打开，如果没有则在open表中更改

void open\_file(char *tmp*[100]) {

unsigned short flag, i, j, k;

flag = research\_file(*tmp*, 1, &i, &j, &k);

if (flag) {

if (search\_file(dir[k].inode)) {

printf("The file %s has opened!\n", *tmp*);

}

else {

flag = 0;

while (fopen\_table[flag]) {

flag++;

}

fopen\_table[flag] = (short) dir[k].inode;

printf("File %s opened!\n", *tmp*);

}

}

else

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

}

Close file:

// 关闭文件 修改open表使得其关闭

void close\_file(char *tmp*[100]) {

unsigned short flag, i, j, k;

flag = research\_file(*tmp*, 1, &i, &j, &k);

if (flag) {

if (search\_file(dir[k].inode)) {

flag = 0;

while (fopen\_table[flag] != dir[k].inode) {

flag++;

}

fopen\_table[flag] = 0;

printf("File %s closed!\n", *tmp*);

}

else {

printf("The file %s has not been opened!\n", *tmp*);

}

}

else {

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

}

}

写文件操作：

void write\_file(char *tmp*[100]) {

unsigned short flag, i, j, k, size = 0, need\_blocks, length;

flag = research\_file(*tmp*, 1, &i, &j, &k);

if (flag) {

if (search\_file(dir[k].inode)) {

entry(dir[k].inode);

if (!(inode\_area[0].i\_mode & 2)) // i\_mode:111b:读,写,执行

{

printf("The file %s can not be writed!\n", *tmp*);

return;

}

fflush(stdin);

while (1) {

tempbuf[size] = getchar();

if (tempbuf[size] == '#') {

tempbuf[size] = '\0';

break;

}

if (size >= 4095) {

printf("Sorry,the max size of a file is 4KB!\n");

break;

}

size++;

}

if (size >= 4095) {

length = 4096;

}

else {

length = strlen(tempbuf);

}

//计算需要的数据块数目

need\_blocks = length / 512;

if (length % 512) {

need\_blocks++;

}

if (need\_blocks < 9) // 文件最大 8 个 blocks(512 bytes)

{

// 分配文件所需块数目

//因为以覆盖写的方式写，要先判断原有的数据块数目

if (inode\_area[0].i\_blocks <= need\_blocks) {

while (inode\_area[0].i\_blocks < need\_blocks) {

inode\_area[0].i\_block[inode\_area[0].i\_blocks] = alloc\_block();

inode\_area[0].i\_blocks++;

}

}

else {

while (inode\_area[0].i\_blocks > need\_blocks) {

rm\_block(inode\_area[0].i\_block[inode\_area[0].i\_blocks - 1]);

inode\_area[0].i\_blocks--;

}

}

j = 0;

while (j < need\_blocks) {

if (j != need\_blocks - 1) {

reload\_block(inode\_area[0].i\_block[j]);

memcpy(Buffer, tempbuf + j \* BLOCK\_SIZE, BLOCK\_SIZE);

update\_block(inode\_area[0].i\_block[j]);

}

else {

reload\_block(inode\_area[0].i\_block[j]);

memcpy(Buffer, tempbuf + j \* BLOCK\_SIZE, length - j \* BLOCK\_SIZE);

inode\_area[0].i\_size = length;

update\_block(inode\_area[0].i\_block[j]);

}

j++;

}

up\_block(dir[k].inode);

}

else {

printf("Sorry,the max size of a file is 4KB!\n");

}

}

else {

printf("The file %s has not opened!\n", *tmp*);

}

}

else {

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

}

}

读文件操作:

// 读文件

void read\_file(char *tmp*[100]) {

unsigned short flag, i, j, k, t;

flag = research\_file(*tmp*, 1, &i, &j, &k);

if (flag) {

if (search\_file(dir[k].inode)) //读文件的前提是该文件已经打开

{

for (flag = 0; flag < inode\_area[0].i\_blocks; flag++) {

reload\_block(inode\_area[0].i\_block[flag]);

for (t = 0; t < inode\_area[0].i\_size - flag \* 512; ++t) {

printf("%c", Buffer[t]);

}}}}

}

Readme：

文件系统根据

[https://blog.csdn.net/qq\_43791817/article/details/117822506?ops\_request\_misc=&request\_id=&biz\_id=102&utm\_term=类EXT2文件系统的设计&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduweb~default-5-117822506.pc\_search\_mgc\_flag&spm=1018.2226.3001.418](https://blog.csdn.net/qq_43791817/article/details/117822506?ops_request_misc=&request_id=&biz_id=102&utm_term=类EXT2文件系统的设计&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduweb~default-5-117822506.pc_search_mgc_flag&spm=1018.2226.3001.4187)

进行构造