西安交通大学

操作系统专题实验报告

班级： 计算机2401

学号： 2244113131

姓名： 王豪源

2022年12月8日

目录

[实验一 进程、线程相关编程经验 8](#_Toc5448)

[1.1 进程相关编程实验 8](#_Toc19150)

[1.1.1 实验目的 8](#_Toc27260)

[1.1.2 实验内容 8](#_Toc17821)

[1.1.3 实验思想 10](#_Toc21739)

[1.1.4 实验步骤 10](#_Toc5981)

[1.1.5 程序运行初值及运行结果分析 17](#_Toc18345)

[1.1.6 实验总结 17](#_Toc13779)

[1.1.6.1 实验中的问题与解决过程 17](#_Toc31835)

[1.1.6.2 实验收获 18](#_Toc2845)

[1.1.6.3 意见与建议 18](#_Toc14195)

[1.1.7 附件 18](#_Toc27024)

[1.2 线程相关编程实验 19](#_Toc11022)

[1.2.1 实验目的 19](#_Toc10111)

[1.2.2 实验内容 19](#_Toc22045)

[1.2.3 实验思想 19](#_Toc27229)

[1.2.4 实验步骤 21](#_Toc27009)

[1.2.5 程序运行初值及运行结果分析 29](#_Toc9177)

[1.2.6 实验总结 30](#_Toc19254)

[1.2.6.1 实验中的问题与解决过程 30](#_Toc6368)

[1.2.6.2 实验收获 30](#_Toc20431)

[1.2.6.3 意见与建议 30](#_Toc18860)

[1.2.7 附件 31](#_Toc17609)

[1.3 自旋锁实验 31](#_Toc2886)

[1.3.1 实验目的 31](#_Toc31362)

[1.3.2 实验内容 31](#_Toc26152)

[1.3.3 实验思想 32](#_Toc13013)

[1.3.4 实验步骤 32](#_Toc10980)

[1.3.5 程序运行初值及运行结果分析 37](#_Toc16809)

[1.3.6 实验总结 37](#_Toc29938)

[1.3.6.1 实验中的问题与解决过程 37](#_Toc22268)

[1.3.6.2 实验收获 37](#_Toc21789)

[1.3.6.3 意见与建议 38](#_Toc26155)

[1.3.7 附件 38](#_Toc26660)

[1.3.7.1 附件1 程序 38](#_Toc31871)

[1.3.7.2 附件2 Readme 38](#_Toc902)

[实验二 进程通信与内存管理 39](#_Toc10586)

[2.1 进程的软中断通信 39](#_Toc12694)

[2.1.1 实验目的 39](#_Toc27754)

[2.1.2 实验内容 39](#_Toc16200)

[2.1.3 实验前准备 40](#_Toc715)

[2.1.4 实验步骤 44](#_Toc29020)

[2.1.5 程序运行初值及运行结果分析 50](#_Toc29196)

[2.1.6 实验总结 54](#_Toc26726)

[2.1.6.1 实验中的问题与解决过程 54](#_Toc28047)

[2.1.6.2 实验收获 54](#_Toc938)

[2.1.6.3 意见与建议 54](#_Toc29824)

[2.1.7 附件 54](#_Toc23737)

[2.1.7.1 附件1 程序 54](#_Toc14161)

[2.1.7.2 附件2 Readme 54](#_Toc25461)

[2.2 进程的管道通信 55](#_Toc2615)

[2.2.1 实验目的 55](#_Toc14225)

[2.2.2 实验内容 55](#_Toc8459)

[2.2.3 实验前准备 56](#_Toc19640)

[2.2.4 实验步骤 57](#_Toc6995)

[2.2.5 运行结果分析 60](#_Toc23756)

[2.2.6实验总结 61](#_Toc27129)

[2.2.6.1 实验中的问题与解决过程 61](#_Toc342)

[2.2.6.2 实验收获 62](#_Toc9089)

[2.2.6.3 意见与建议 62](#_Toc21865)

[2.2.7 附件 62](#_Toc17129)

[2.2.7.1 附件1 程序 62](#_Toc19316)

[2.2.7.2 附件2 Readme 63](#_Toc28320)

[2.3 内存的分配与回收 63](#_Toc32481)

[2.3.1 实验目的 63](#_Toc4159)

[2.3.2 实验内容 63](#_Toc18291)

[2.3.3 实验前准备 63](#_Toc2782)

[2.3.4 实验步骤 68](#_Toc17594)

[2.3.5 程序运行初值及运行结果分析 86](#_Toc3389)

[2.3.6 实验总结 88](#_Toc2593)

[2.3.6.1 实验中的问题与解决过程 88](#_Toc9758)

[2.3.6.2 实验收获 88](#_Toc6233)

[2.3.6.3 意见与建议 88](#_Toc13441)

[2.3.7 附件 88](#_Toc30512)

[2.3.7.1 附件1 程序 88](#_Toc23198)

[2.3.7.2 附件2 Readme 89](#_Toc6936)

[实验三 类EXT2文件系统编写 89](#_Toc4559)

[3.1 实验目的 89](#_Toc23527)

[3.2 实验内容 89](#_Toc11990)

[3.3 实验思想 89](#_Toc11928)

[3.4 实验步骤 89](#_Toc24849)

[3.5 程序运行初值及运行结果分析 102](#_Toc8727)

[3.6 实验总结 103](#_Toc19619)

[3.6.1 实验中的问题与解决过程 103](#_Toc24409)

[3.6.2 实验收获 103](#_Toc7836)

[3.6.3 意见与建议 103](#_Toc12113)

[3.7 附件 104](#_Toc13220)

[3.7.1 附件1 程序 104](#_Toc23943)

[3.7.2 附件2 Readme 104](#_Toc27829)

# 实验一 进程、线程相关编程经验

# 1.1 进程相关编程实验

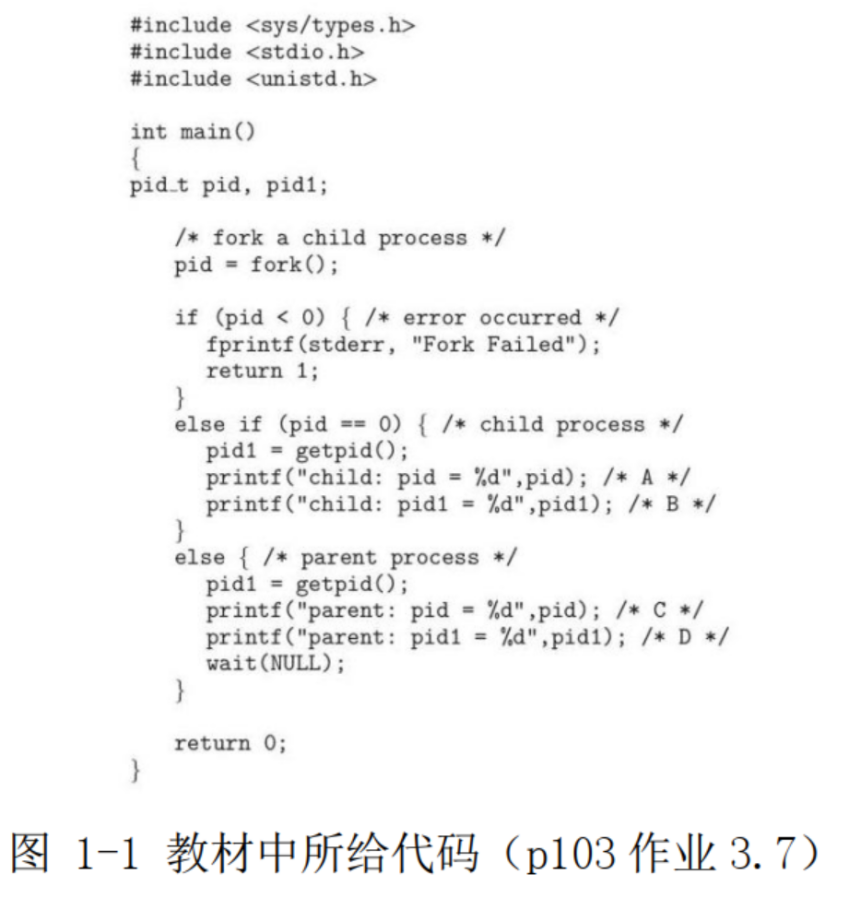
## 1.1.1 实验目的

（1） 熟悉 Linux 操作系统的基本环境和操作方法，通过运行系统命令查看系统基本信息以了解系统；

（2）编写并运行简单的进程调度相关程序，体会进程调度、进程间变量的管理等机制在操作系统实际运行中的作用。

## 1.1.2 实验内容

（1） 熟悉操作命令、编辑、编译、运行程序。完成图 1-1 程序的运行验证，多运行几次程序观察结果；去除 wait 后再观察结果并进行理论分析。



（2）扩展图 1-1 的程序：

a） 添加一个全局变量并在父进程和子进程中对这个变量做不同操作，输出操作结果并解释；

b） 在 return 前增加对全局变量的操作并输出结果，观察并解释；

c） 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数；

## 1.1.3 实验思想

（1） 进程： 进程是计算机科学中的一个重要概念，它是操作系统中的基本执行单位。进程代表着一个正在执行的程序实例，它包括了程序的代码、数据和执行状态等信息。操作系统通过进程管理来实现对计算机资源的有效分配和控制；

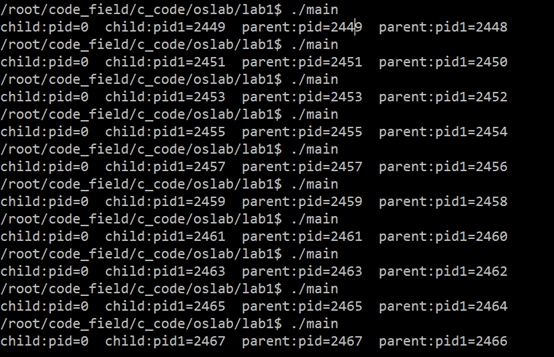
（2） PID： PID 是进程标识符（Process Identifier）的缩写，它是用来唯一标识一个操作系统中的进程的数值。每个正在运行或已经终止的进程都会被分配一个唯一的 PID，这个标识符可以用来在操作系统内部识别和管理进程；

（3） fork()函数： fork() 是一个在类 Unix 操作系统中常见的系统调用，用于创建一个新的进程，新进程是原进程（父进程）的副本。新进程被称为子进程，它与父进程共享很多资源，但也有一些独立的属性。 fork() 被用于实现多进程编程，常见于操作系统和并发编程中。 函数返回一个整数，如果返回值为负数，则表示创建进程失败。如果返回值为 0，表示当前正在执行的代码是在子进程中。如果返回值大于 0，表示当前正在执行的代码是在父进程中，返回值是子进程的 PID。 调用 fork() 函数时，操作系统会创建一个新的进程，该进程是调用进程的一个副本，称为子进程。子进程几乎与父进程相同，包括代码、数据、文件描述符等。但是子进程拥有自己的独立的内存空间和资源。

## 1.1.4 实验步骤

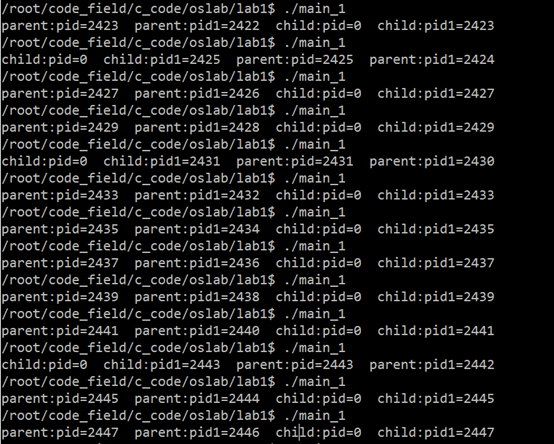
本实验通过在程序中输出父、子进程的 pid，分析父子进程 pid 之间的关系，进一步加入 wait()函数分析其作用。

步骤一： 编写并多次运行图 1-1 中代码



可以看到，始终是先打印子进程，再打印父进程。

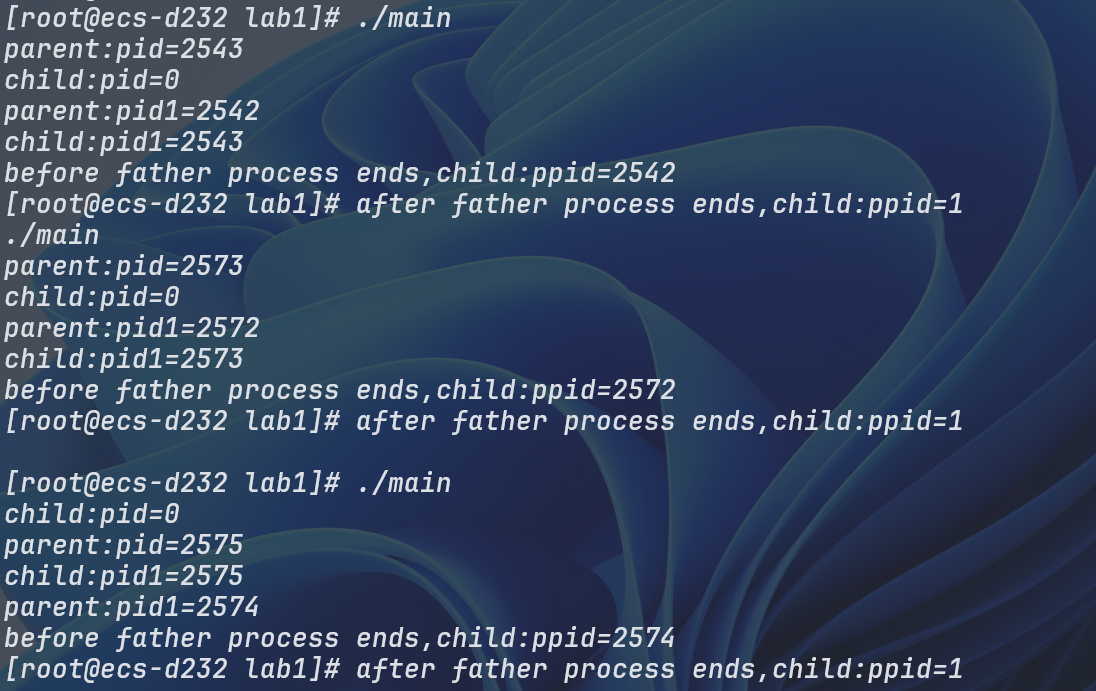
步骤二： 删去图 1-1 代码中的 wait()函数并多次运行程序，分析运行结果。



可以发现当前情况下存在child有可能先于parent进程运行。

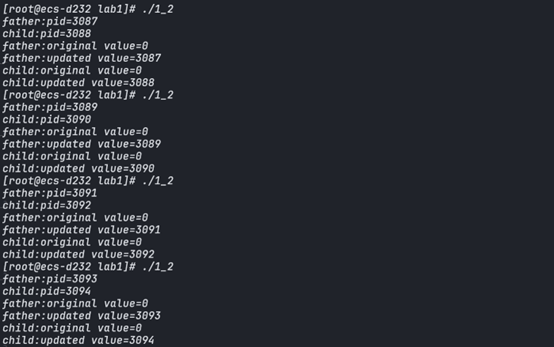
原因在于wait会阻塞父进程的执行知道子进程执行完成。去除之后，父子进程并发执行，打印顺序将不确定，并会出现僵尸进程。

可以将父进程先结束，打印子进程的ppid实验如下



从运行结果中可以看到，当父进程结束后，子进程的ppid变为了1。

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



我定义了一个全局变量value，在每个进程中，先输出原始值，在输出更改后的值，对value的更改都是将他设置为当前进程的PID。

在程序中打印value的地址

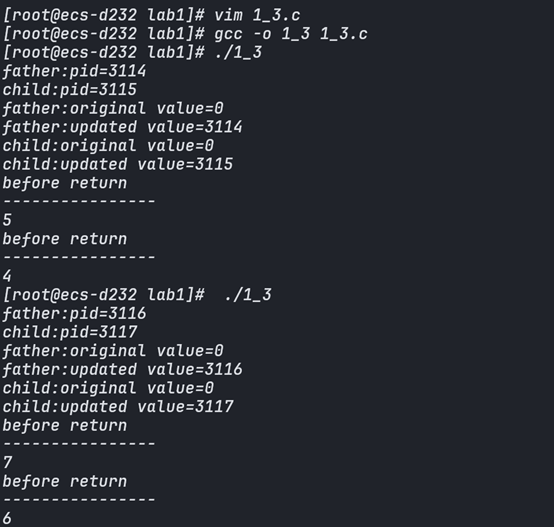


从这个结果，可以更好的理解操作系统的内存机制。

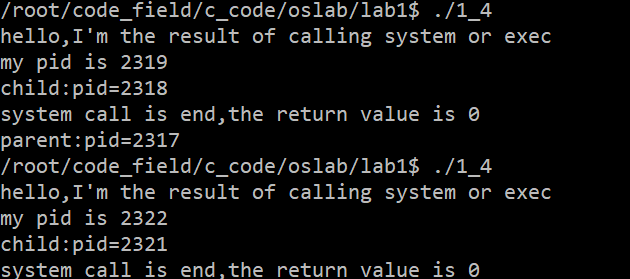
第一，每次运行时，父子进程的value地址是相同的，这是因为在fork时，父子进程共享页表，因此地址是一样的,value的独立副本是它们采取写时复制机制，当需要对某个地址空间的值做修改时，会使该页表指向新的物理帧。

第二，每次运行value的虚拟地址空间都是不变的，这是因为代码在编译链接阶段就已经确定好了变量的虚拟地址。

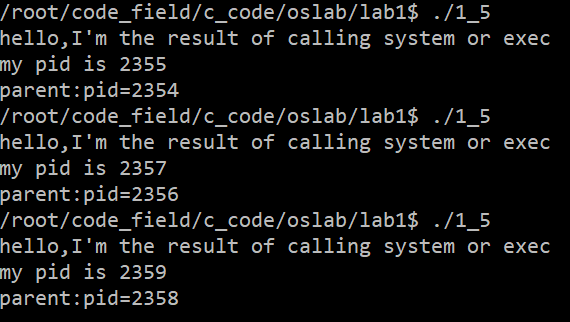
步骤四： 在步骤三基础上，在 return 前增加对全局变量的操作（自行设计）并输出结果，观察并解释所做操作和输出结果

进行的操作是对10取余，再次印证了value是独立的副本。

步骤五： 修改图 1-1 程序，在子进程中调用 system()与 exec 族函数。 编写system*call.c 文件输出进程号 PID，编译后生成 system*call 可执行文件。在子进程中调用 system\_call,观察输出结果并分析总结。

调用system()运行结果

可以看到system会新建一个进程。调用exec族函数运行结果



运行结果分析，调用exec族函数，会将代码段加载进来，替换掉原来的代码段，使得原来得代码在调用exec之后的代码段是没有继续运行的。

但exec并不会创建一个新进程，他还是原来的那个进程。

## 1.1.5 程序运行初值及运行结果分析

运行结果分析如上。

## 1.1.6 实验总结

### 1.1.6.1 实验中的问题与解决过程

1. 刚开始不熟悉wait的基本用法，通过上网搜索资料，了解wait的工作原理，也再次加深了对僵尸进程的理解。

2. 不了解system和exec族函数的用法，通过浏览各大技术博客，理解了system相当于命令行的用法，并新建立进程。学习了exec后缀-l，-e，-v，-p的含义，并理解了为什么加载不熟悉的程序要用-e后缀创建新的环境变量。

3. 在用命令编译文件时，含有线程相关的源文件编译错误。学习之后了解到要手动链接到POSIX库，加上后缀-pthread解决。

4. 在线程实验的步骤三中，exec前的线程PID以及TID无法输出。原因是printf要缓冲区刷新时才输出，用fprintf或者exec前加上fflush强制刷新缓冲区执行。

### 1.1.6.2 实验收获

1. **进程管理理解深化**：通过这个实验，更加深入地了解了 Linux 系统中进程的创建、管理和调度。特别是通过观察 wait() 函数的行为，理解了父子进程间同步的重要性。
2. **编程技巧提升**：这个实验让我更熟悉了 C 语言的编程模式，尤其是涉及到系统级调用和进程管理的函数。对 fork(), wait(), system(), 和 exec() 等函数有了更深入的了解。
3. **系统调用与命令行工具**：实验中涉及到 system() 和 exec() 系列函数，使我了解了如何在程序中执行系统命令，以及如何用 exec() 替换当前进程的执行内容。
4. **多进程编程模型**：通过在一个程序中创建多个进程，以及管理这些进程的行为和状态，我对多进程编程有了更实际的认识和理解。

### 1.1.6.3 意见与建议

1. **增加更多的进程管理实验**：当前实验内容虽然涵盖了基础的进程创建和管理，但在实际应用中还有更多高级的用法，比如多进程并发处理，进程通信等，建议加入这部分内容。
2. **提供更详细的函数文档和示例代码**：尽管实验手册给出了基础框架，但更多具体函数的使用例子和文档将会更有助于理解。
3. **加强对错误处理的教学**：在实际编程中，错误处理是非常重要的一环。本次实验虽然有简单的错误处理，但没有详细介绍这方面的最佳实践。

## 1.1.7 附件

附件内容在1-3下给出。

# 1.2 线程相关编程实验

## 1.2.1 实验目的

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

## 1.2.2 实验内容

（1） 在进程中给一变量赋初值并成功创建两个线程；

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

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

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

## 1.2.3 实验思想

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

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

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

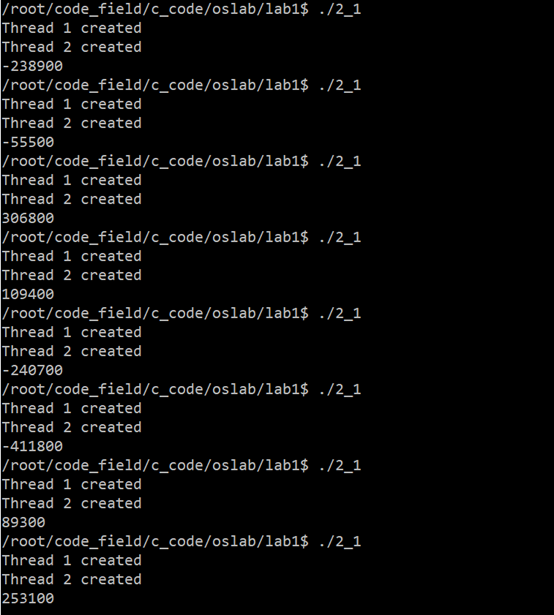
（3）互斥与同步： 为了解决竞态条件带来的问题，可以使用互斥锁（Mutex）来保护共享变量的访问。在每个线程对变量进行操作之前，先获取互斥锁，操作完成后再释放锁。这样一来，每次只有一个线程能够访问变量，从而避免了并发访问带来的不稳定性。

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

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

## 1.2.4 实验步骤

步骤一： 设计程序，创建两个子线程， 两线程分别对同一个共享变量多次操作，观察输出结果。



可以发现，此时输出的结果并不固定。这是因为两个线程并发执行，对共享变量res读取到寄存器中时，读取到的是同一个值，一个线程要+100，一个线程要-100，本来效果应该是相互抵消的，但写回内存时，只将自己操作后的结果写回内存，此时就表现出+100或-100，导致值的改变。

考虑一个简单的情境：

1. \*\*res\*\* 当前值为 0。

2. 线程1 读取其值（0），准备加1。

3. 线程2 被调度，读取其值（还是0），准备减1。

4. 线程1 继续执行，将值加1，结果为1。

5. 线程2 继续执行，减1（认为之前的值是0），结果为 -1。

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。

同步（Synchronization）在多线程编程中指的是协调多个线程的执行，以确保它们能够正常、可预测地访问共享资源或完成某些任务。互斥（Mutual Exclusion）是同步的一种特殊形式，确保一次只有一个线程能访问某个特定的资源或代码段。互斥通常通过互斥锁（Mutex）来实现，但它也可以通过其他机制来实现，比如信号量（Semaphore）。信号量和互斥锁类似，但更为通用。信号量可以用来解决除了互斥之外的其他同步问题。

PV操作是信号量（Semaphores）操作的传统术语，源自荷兰语的Proberen（尝试）和Verhogen（增加）。在信号量的上下文中，P操作通常用于申请或等待资源，而V操作用于释放或发出信号。

**P操作（也叫wait或down或sem\_wait）**

* 当一个线程执行P操作时，它会检查信号量的值。
  + 如果信号量的值大于0，那么它将减少信号量的值（通常是减1）并继续执行。
  + 如果信号量的值为0，线程将被阻塞，直到信号量的值变为大于0。

**V操作（也叫signal或up或sem\_post）**

* V操作增加信号量的值（通常是加1）。
  + 如果有线程因执行P操作而阻塞在这个信号量上，一个或多个线程将被解除阻塞，并被允许减少信号量的值。

在C语言中，使用POSIX信号量

* sem\_wait(&semaphore);：执行P操作。
* sem\_post(&semaphore);：执行V操作。

其中，semaphore是一个sem\_t类型的变量，代表信号量。

sem\_init和sem\_destroy是POSIX信号量（Semaphores）的初始化和销毁函数，它们用于设置和清理信号量。

sem\_init

这个函数用于初始化一个未命名的信号量。

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

**sem:** 一个指向信号量对象的指针。

**pshared:** 如果这个参数是0，信号量就是当前进程的局部信号量。如果这个参数非0，则该信号量在多个进程间共享。

**value:** 信号量的初始值。

这个函数成功时返回0，失败时返回-1。

**sem\_destroy**

这个函数用于销毁一个未命名的信号量，释放其占用的资源。

**int sem\_destroy(sem\_t \*sem);**

* **sem**: 一个指向信号量对象的指针。

这个函数成功时返回0，失败时返回-1。在使用sem\_destroy之前，确保没有线程被阻塞在该信号量上。

整体的代码为：

#include<stdio.h>

#include<pthread.h>

#include<semaphore.h>

int res=0;

sem\_t sem;

void \*add(void \*target){

for(int i=0;i<100000;i++){

sem\_wait(&sem);

res+=100;

sem\_post(&sem);

}

}

void \*minus(void \*target){

for(int i=0;i<100000;i++){

sem\_wait(&sem);

res-=100;

sem\_post(&sem);

}

}

int main(){

pthread\_t thread1,thread2;

sem\_init(&sem, 0, 1); // *初始化信号量，初始值为1，用于互斥*

if(pthread\_create(&thread1,NULL,add,&res)==0){

printf("Thread 1 created\n");

}

if(pthread\_create(&thread2,NULL,minus,&res)==0){

printf("Thread 2 created\n");

}

pthread\_join(thread1,NULL);

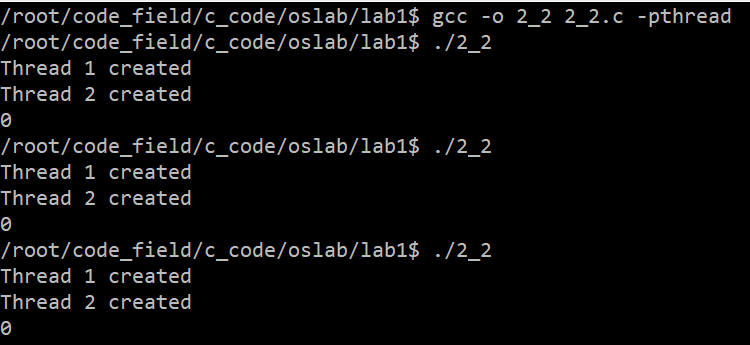
pthread\_join(thread2,NULL);

printf("%d\n",res);

sem\_destroy(&sem); // 销毁信号量

return 0;

}



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

System\_call源程序

#include<stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main(){

printf("thread\_system\_call pid:%d\n",getpid());

}

调用system的函数源码

#include<stdio.h>

#include<pthread.h>

#include<stdlib.h>

#include <sys/types.h>

void\* system\_call(void\* arg){

int num=\*((int\*)arg);

printf("Thread%d created\n",num);

printf("Thread%d TID:%d, PID:%d\n",num,pthread\_self(),getpid());

system("/root/code\_field/c\_code/oslab/lab1/thread\_system\_call");

}

int main(){

pthread\_t thread1,thread2;

int arg1=1,arg2=2;

if(pthread\_create(&thread1,NULL,system\_call,&arg1)!=0){

printf("Thread 1 creation failed\n");

}

if(pthread\_create(&thread2,NULL,system\_call,&arg2)!=0){

printf("Thread 2 creation failed\n");

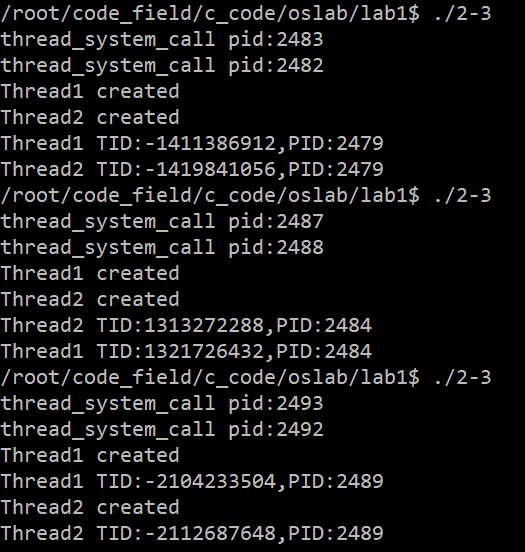
}

pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

}

运行结果



结果显示线程的TID不同，但都有一个相同的PID，也就是父进程的PID。

每一个线程的system调用都会创建一个新进程

应用exec族函数源码

#include<stdio.h>

#include<pthread.h>

#include<unistd.h>

#include<sys/types.h>

void\* exec\_call(void\* arg){

int num=\*((int\*)arg);

fprintf(stderr,"Thread%d created\n",num);

fprintf(stderr,"Thread%d TID:%d,PID:%d\n",num,pthread\_self(),getpid());

execl("/root/code\_field/c\_code/oslab/lab1/exec\_call","exec\_call",NULL);

printf("if execl fails,this line will be printed\n");

}

int main(){

pthread\_t thread1,thread2;

int arg1=1,arg2=2;

if(pthread\_create(&thread1,NULL,exec\_call,&arg1)!=0){

printf("Thread 1 creation failed\n");

}

if(pthread\_create(&thread2,NULL,exec\_call,&arg2)!=0){

printf("Thread 2 creation failed\n");

}

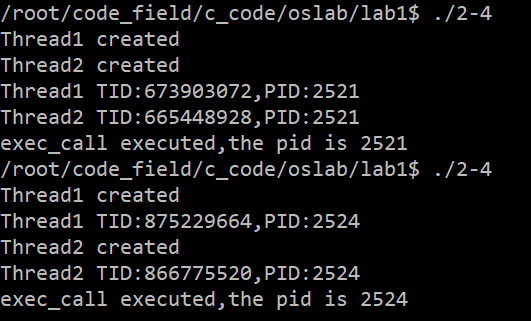
pthread\_join(thread1,NULL);

pthread\_join(thread2,NULL);

printf("if exec succeed,this will no be prinrfed");

}

运行结果分析



结果分析，父线程的最后一行的printf并没有执行，并且exec只有的代码实际上只执行了一遍，这是因为exec的调用实际上是进程级别的，调用exec会导致整个进程的代码段被替换，而不仅仅是当前线程执行的函数。

## 1.2.5 程序运行初值及运行结果分析

运行结果分析如上。

## 1.2.6 实验总结

### 1.2.6.1 实验中的问题与解决过程

### 1.2.6.2 实验收获

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

我研究了多线程对共享资源访问的问题，尤其是竞态条件和其对程序输出的影响，成功地应用了互斥锁和信号量来解决这些问题，增强了程序的健壮性。

除了技术细节，实验还锻炼了我的问题解决能力。遇到编译错误、不一致的输出和程序终止问题时，都能够分析问题的根本原因，并找到相应的解决方案。这些都是软件开发中非常重要的技能。

### 1.2.6.3 意见与建议

1. **更多的实例操作**：尽管本次实验已经涵盖了多个重要的概念，但实践是检验真理的唯一标准。在未来的实验中，更多的实例操作可能会更有助于理解这些复杂的概念。
2. **代码审查与反思**：完成代码和实验后，进行代码审查和反思总结也是非常有价值的。这不仅能帮助你找出可能的错误或不足，还能促使思考如何优化和改进。
3. **深入理解同步机制**：本实验主要集中在使用信号量进行同步，但还有其他同步机制，如条件变量、读写锁等，也值得进一步研究。
4. **探索更多系统调用**：除了system()和exec族函数，还有很多其他有用的系统调用，如pipe()、dup()等，这些都是进程和线程间通信的重要手段。

## 1.2.7 附件

1-3下给出。

# 1.3 自旋锁实验

## 1.3.1 实验目的

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

（1） 了解自旋锁的基本概念： 通过研究自旋锁的工作原理和特点，深入理解自旋锁相对于其他锁机制的优势和局限性；

（2） 实验自旋锁的应用： 在一个多线程的实验环境中，设计一个竞争资源的场景，让多个线程同时竞争对该资源的访问；

（3） 实现自旋锁的同步： 使用自旋锁来保护竞争资源的访问，确保同一时间只有一个线程可以访问该资源，避免数据不一致和竞态条件；

## 1.3.2 实验内容

（1） 在进程中给一变量赋初值并成功创建两个线程；

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

（3） 使用自旋锁实现互斥和同步；

## 1.3.3 实验思想

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

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

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

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

自旋锁的工作原理中关键的部分在于“自旋”这一概念，即等待获取锁的线程会循环忙等待，不断检查标志变量的状态，直到能够成功获取锁。这种方式在锁的占用时间很短的情况下可以减少线程切换的开销，提高程序性能。

## 1.3.4 实验步骤

步骤一： 根据实验内容要求，编写模拟自旋锁程序代码 spinlock.c， 待补充主函数的示例代码如下：

*/\*\*\*spinlock.c\*in xjtu\*2023.8*  
*\*/*  
#include <stdio.h>#include <pthread.h>*// 定义自旋锁结构体typedef struct {*  
int flag;} spinlock\_t;  
*// 初始化自旋锁*  
void spinlock\_init(spinlock\_t \*lock) {lock->flag = 0;  
}  
*// 获取自旋锁*  
void spinlock\_lock(spinlock\_t \*lock) {  
while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {*// 自旋等待*  
}}  
*// 释放自旋锁*  
void spinlock\_unlock(spinlock\_t \*lock) {\_\_sync\_lock\_release(&lock->flag);  
}  
*// 共享变量int shared\_value = 0;*  
*// 线程函数*  
void \*thread\_function(void \*arg) {spinlock\_t \*lock = (spinlock\_t \*)arg;for (int i = 0; i < 5000; ++i) {  
spinlock\_lock(lock);shared\_value++;spinlock\_unlock(lock);  
}  
return NULL;}  
int main() {  
pthread\_t thread1, thread2;  
spinlock\_t lock;*// 输出共享变量的值*  
*// 初始化自旋锁*  
*// 创建两个线程*  
*// 等待线程结束*  
*// 输出共享变量的值return 0;*  
}

补充完整代码为：

#include <stdio.h>

#include <pthread.h>

// 定义自旋锁结构体

typedef struct {

int flag;

}spinlock\_t;

// 初始化自旋锁

void spinlock\_init(spinlock\_t \*lock) {

lock->flag = 0;

}

// 获取自旋锁

void spinlock\_lock(spinlock\_t \*lock) {

while (\_\_sync\_lock\_test\_and\_set(&lock->flag, 1)) {

// 自旋等待

}

}

// 释放自旋锁

void spinlock\_unlock(spinlock\_t \*lock) {

\_\_sync\_lock\_release(&lock->flag);

}

// 共享变量

int shared\_value = 0;

// 线程函数

void \*thread\_function(void \*arg) {

spinlock\_t \*lock = (spinlock\_t \*)arg;

for (int i = 0; i < 5000; ++i) {

spinlock\_lock(lock);

shared\_value++;

spinlock\_unlock(lock);

}

return NULL;

}

int main() {

pthread\_t thread1, thread2;

spinlock\_t lock;

// 输出共享变量的值

printf("original shared\_value: %d\n", shared\_value);

// 初始化自旋锁

spinlock\_init(&lock);

// 创建两个线程

if(pthread\_create(&thread1, NULL, thread\_function, (void \*)&lock) != 0) {

perror("Failed to create thread 1");

return 1;

}

printf("Thread 1 created\n");

if(pthread\_create(&thread2, NULL, thread\_function, (void \*)&lock) != 0) {

perror("Failed to create thread 2");

return 1;

}

printf("Thread 2 created\n");

// 等待线程结束

pthread\_join(thread1, NULL);

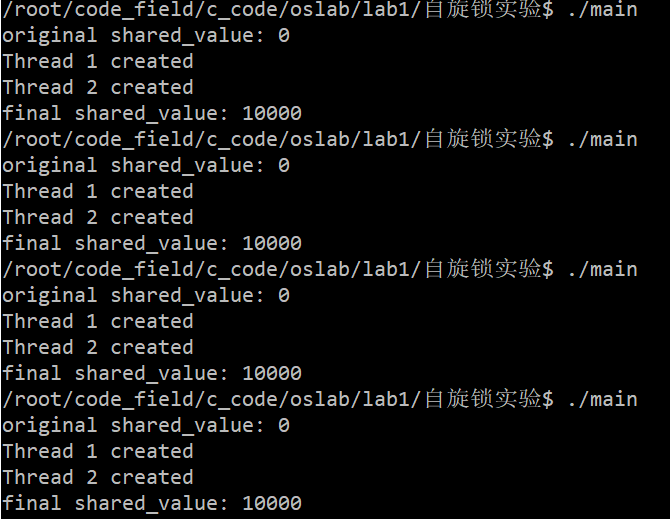
pthread\_join(thread2, NULL);

// 输出共享变量的值

printf("final shared\_value: %d\n", shared\_value);

return 0;

}

补充完成代码后，编译并运行程序，分析运行结果

自旋锁设定成功，每个线程都正确地对shared\_value进行了修改

## 1.3.5 程序运行初值及运行结果分析

shared\_value初始值为0

运行后获得了正确的结果：10000，这说明自旋锁设定有效

## 1.3.6 实验总结

### 1.3.6.1 实验中的问题与解决过程

### 1.3.6.2 实验收获

1. **提高了问题解决能力**: 遇到的问题和解决过程锻炼了分析问题和调试代码的能力，尤其是如何定位和修复内存访问相关的错误。
2. **加强了实际操作能力**: 不仅仅是理论知识，还有实际将理论应用到代码中的能力也得到了加强。

### 1.3.6.3 意见与建议

1. **注意代码的健壮性**: 在编写代码时，应考虑各种边界条件和异常情况，例如空指针、非法输入等。
2. **充分利用编译器**: 使用编译器的警告和错误信息作为调试的线索，而不仅仅是编译的障碍。
3. **注重代码可读性**: 代码是给人看的，其次才是给机器执行。应确保代码结构清晰，命名规范，同时也要充分注释。
4. **持续学习和实践**: 技术是日新月异的，通过不断的学习和实践，才能跟上技术的步伐，更好地应对各种问题。

## 1.3.7 附件

### 1.3.7.1 附件1 程序



### 1.3.7.2 附件2 Readme



# 实验二 进程通信与内存管理

# 2.1 进程的软中断通信

## 2.1.1 实验目的

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

## 2.1.2 实验内容

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

（2）根据流程图（如图 2.1 所示） 编制实现软中断通信的程序： 使用系统调用 fork()创建两个子进程，再用系统调用 signal()让父进程捕捉键盘上发出的中断信号（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 。

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

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

## 2.1.3 实验前准备

实验相关 UNIX 系统调用介绍：

(1)fork(): 创建一个子进程。

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

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

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

(2)exec(): 装入并执行相应文件。

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

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

(5)getpid():获得进程号。

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

参数定义： int lockf(files,function,size) int files,founction;

long size;

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

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

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

头文件为： #include <signal.h>

参数定义： signal(sig,function) int sig;

void (\*func) ();其中 sig 共有 19 个值

注意： signal 函数会修改进程对特定信号的响应方式。

(9)pipe(fd);

int fd[2];

其中 fd[1]是写端，向管道中写入， fd[0]是读端，从管道中读出。

(10)暂停一段时间 sleep；

调用 sleep 将在指定的时间 seconds 内挂起本进程。

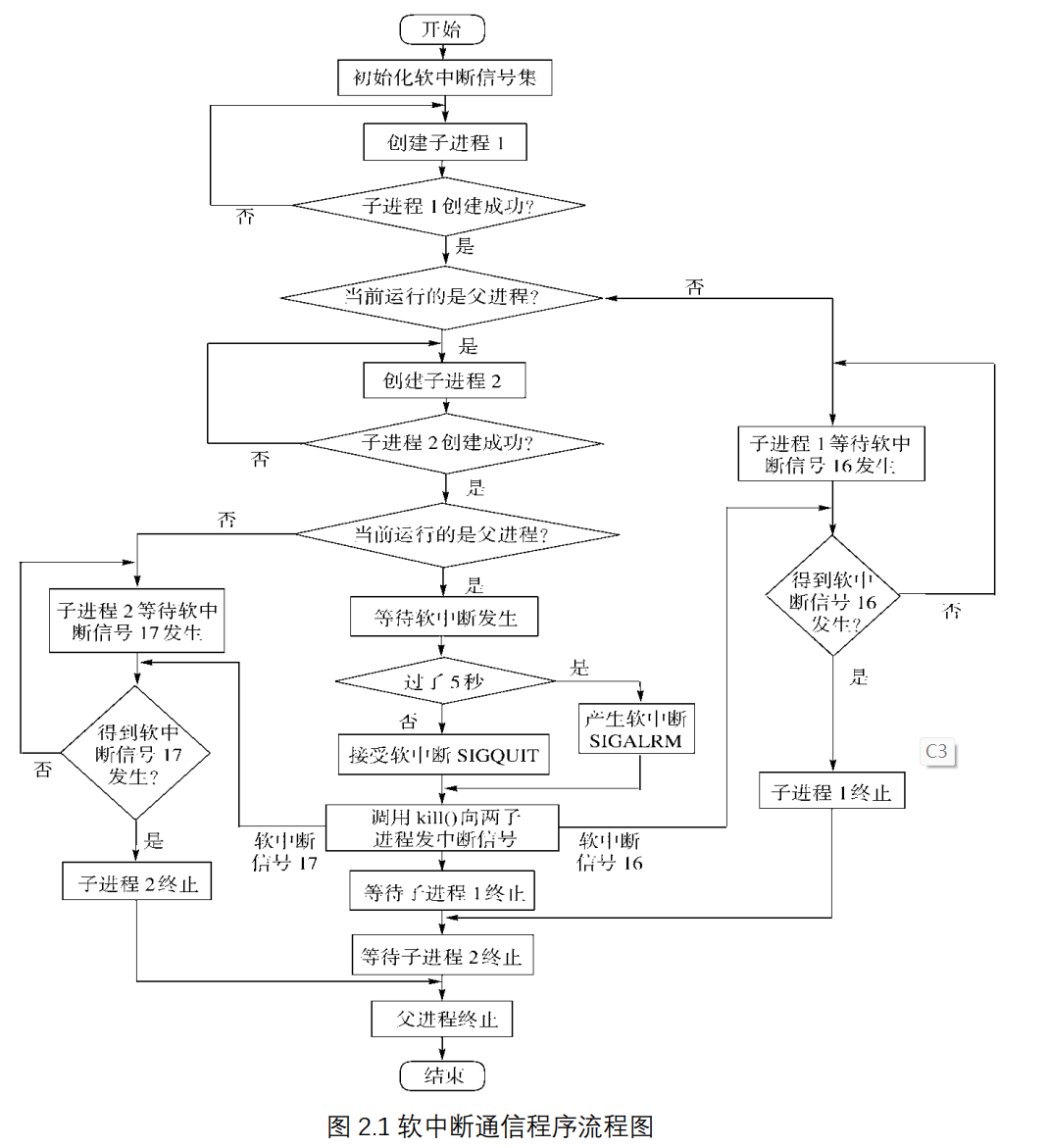
其调用格式为：“ unsigned sleep(unsigned seconds);” ；返回值为实际的挂起时间。

(11)暂停并等待信号 pause；

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

其调用格式为“int pause(void);”

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



编写实验代码需要考虑的问题：

(1)父进程向子进程发送信号时，如何确保子进程已经准备好接收信号？

(2)如何阻塞住子进程，让子进程等待父进程发来信号？

## 2.1.4 实验步骤

对于第一个问题，我设置了两个变量child1\_isready和child2\_isready，分别表示子进程1和子进程2是否已经准备好接收信号。子进程在启动时会向父进程发送一个信号，通知父进程它已经准备好了。在父进程中，将信号与signal函数绑定在一起，当他接受到某个子进程的信号后，就将相应进程的ready位设置为1.父进程在发送信号之前会检查这两个变量，确保子进程已经准备好。

对于第二个问题，我使用了pause函数来阻塞子进程。pause函数会使进程进入睡眠状态，直到接收到一个信号为止。子进程在启动后会调用pause函数，等待父进程发送信号。当父进程发送信号后，子进程会被唤醒并执行相应的信号处理函数。

代码展示

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <signal.h>

int flag=0;

int is\_timeout=0;

int child1\_isready=0;

int child2\_isready=0;

void inter\_handler(int sig) {

if(sig==SIGINT || sig==SIGQUIT){

flag++;

}

}

void alarm\_handler(int sig) {

if(sig==SIGALRM){

is\_timeout=1;

flag=1;

}

}

void child1\_isready\_handler(int sig) {

if(sig==16){

child1\_isready=1;

}

}

void child2\_isready\_handler(int sig) {

if(sig==17){

child2\_isready=1;

}

}

void waiting() {

while(flag<1){

pause();

}

}

void child\_handler(int sig) {

if(sig==16){

printf("\nChild process1 is killed by parent!!\n");

exit(0);

}

if(sig==17){

printf("\nChild process2 is killed by parent!!\n");

exit(0);

}

}

int main() {

signal(SIGQUIT, inter\_handler);

signal(SIGINT, inter\_handler);

signal(SIGALRM, alarm\_handler);

signal(16, child1\_isready\_handler);

signal(17, child2\_isready\_handler);

// TODO: 五秒之后或接收到两个信号

pid\_t pid1=-1, pid2=-1;

while (pid1 == -1)pid1=fork();

if (pid1 > 0) {

while (pid2 == -1)pid2=fork();

if (pid2 > 0) {

alarm(5);

waiting();

alarm(0);

if(is\_timeout){

//printf("\nThe signal is timeout proc stopped auto!!\n");

}

while(!child1\_isready){

pause();

}

kill(pid1, 16);

while(!child2\_isready){

pause();

}

kill(pid2, 17);

wait(NULL);

wait(NULL);

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

} else {

// TODO: 子进程 2

kill(getppid(), 17);

signal(17, child\_handler);

while(1){

pause();

}

}

} else {

// TODO:子进程 1

kill(getppid(), 16);

signal(16, child\_handler);

while(1){

pause();

}

printf("\nChild process1 is killed by parent!!\n");

return 0;

}

return 0;

}

### 代码解释

这份代码是有时钟中断的，如果要使程序没有时钟中断，只需要注释掉signnal(SIGALRM, alarm\_handler);这一行即可。

flag变量用于记录父进程接收到的信号数量。当父进程接收到SIGINT或SIGQUIT信号时，inter\_handler函数会将flag加1。当父进程接收到SIGALRM信号时，alarm\_handler函数会将is\_timeout设置为1，并将flag加1。

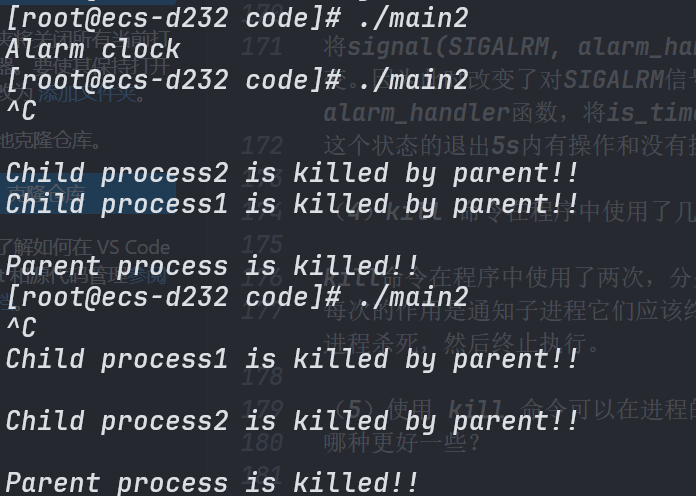
waiting函数会使父进程进入睡眠状态，直到flag的值大于等于1，即接收到至少一个信号为止。

在父进程中设置5秒的闹钟，如果在5秒内没有接收到任何信号，闹钟会触发SIGALRM信号，调用alarm\_handler函数，将is\_timeout设置为1，并将flag加1，从而使父进程退出等待状态。

如果接收到了SIGINT或SIGQUIT信号，父进程会退出waiting，并执行后续代码清除定时器，准备向子进程发送信号。

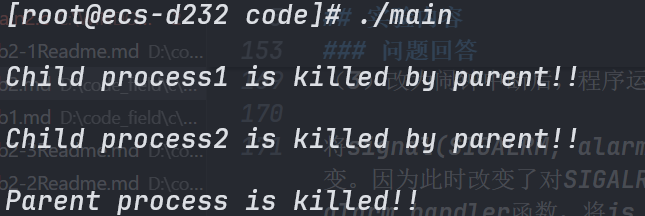
## 2.1.5 程序运行初值及运行结果分析

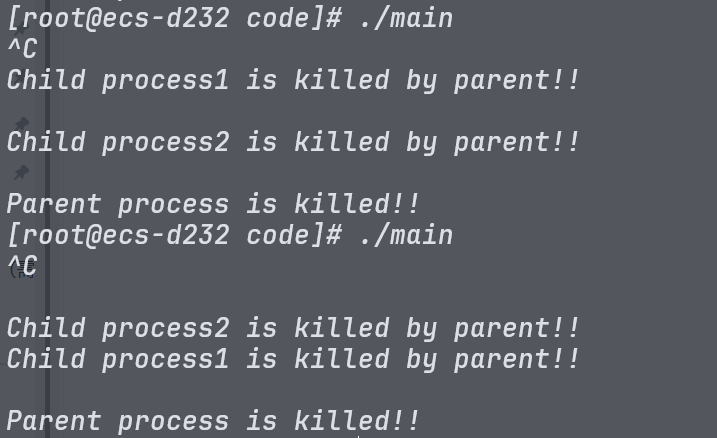
**没有定时器**



**此时如果五秒内，没有任何操作，会执行默认行为，终止进程并输出Alarm clock**

**有定时器**





将signal(SIGALRM, alarm\_handler);取消注释后，程序运行结果并不因为5s内有无操作而改变。因为此时改变了对SIGALRM信号的处理方式，父进程在5秒后会接收到SIGALRM信号，并调用alarm\_handler函数，将is\_timeout设置为1，并将flag加1，使父进程退出waiting状态。

这个状态的退出5s内有操作和没有操作都是一样的。

**其他问题回答**

（1）你最初认为运行结果会怎么样？写出你猜测的结果。

我最初认为两个进程会按照代码顺序那样，先杀死子进程1，再杀死子进程2。

（2）实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将 5 秒内中断和 5 秒后中断的运行结果截图，试对产生该现象的原因进行分析。

实际上，两个子进程的杀死顺序并不确定。并且如果5秒内有任何操作，父进程会向子进程发送信号，而如果五秒内没有任何操作，终端会输出"Alarm clock"。

原因分析：对于杀死顺序不确定这个问题，可能是因为编译代码时，两个kill的顺序并没有严格保证执行顺序，不过这个可能性比较低。

更可能的原因是父进程只负责发送信号，在发送信号后，子进程的执行顺序是由操作系统调度决定的，因此子进程1和子进程2被杀死的顺序是不确定的。

对于5s内中断和5秒后中断的区别：由于alarm信号没有被处理，因此当5秒内没有任何操作时，闹钟会触发SIGALRM信号，导致父进程退出等待状态，并输出"Alarm clock"。如果5秒内有任何操作，父进程会接收到SIGINT或SIGQUIT信号，正常向子进程发送信号。

（3）改为闹钟中断后，程序运行的结果是什么样子？与之前有什么不同？

将signal(SIGALRM, alarm\_handler);取消注释后，程序运行结果并不因为5s内有无操作而改变。因为此时改变了对SIGALRM信号的处理方式，父进程在5秒后会接收到SIGALRM信号，并调用alarm\_handler函数，将is\_timeout设置为1，并将flag加1，使父进程退出waiting状态。

这个状态的退出5s内有操作和没有操作都是一样的。

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

kill命令在程序中使用了两次，分别用于向子进程1和子进程2发送信号。

每次的作用是通知子进程它们应该终止执行。执行后的现象是子进程会输出相应的消息，表示它们被父进程杀死，然后终止执行。

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

进程可以通过调用exit函数来主动退出。exit函数会终止进程的执行，并返回一个状态码给操作系统。

这两种退出方式各有优缺点。使用kill命令可以在进程的外部强制终止进程，适用于需要立即停止进程的情况，但可能会导致资源未被正确释放。而主动调用exit函数可以确保进程有机会进行清理工作，释放资源，但可能需要更多的时间来完成退出过程。总体来说，主动退出更好一些，因为它允许进程有机会进行必要的清理工作。

## 2.1.6 实验总结

### 2.1.6.1 实验中的问题与解决过程

不了解kill、signal相关的调用。查询相关文档学习。

### 2.1.6.2 实验收获

学会了如何使用signal\_handler来协调不同进程，线程之间的运行。

### 2.1.6.3 意见与建议

## 2.1.7 附件

### 2.1.7.1 附件1 程序



### 2.1.7.2 附件2 Readme

Readme在最后一起给出

# 2.2 进程的管道通信

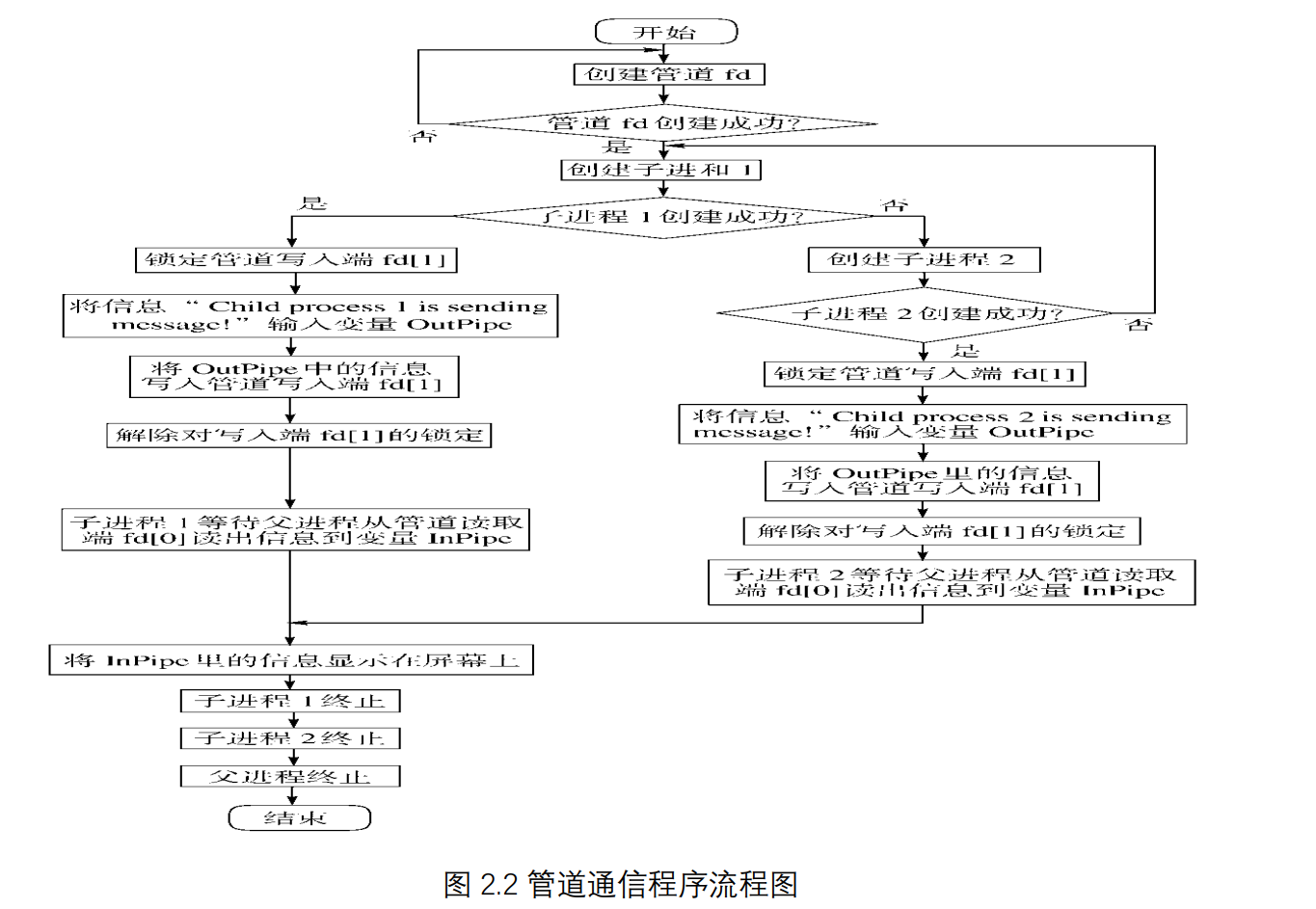
## 2.2.1 实验目的

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

## 2.2.2 实验内容

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

（2）根据流程图（如图 2.2 所示）和所给管道通信程序，按照注释里的要求把代码补充完整，运行程序，体会互斥锁的作用，比较有锁和无锁程序的运行结果，分析管道通信是如何实现同步与互斥的。



## 2.2.3 实验前准备

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

为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

①互斥，即当一个进程正在对 pipe 执行读/写操作时，其它(另一)进程必须等待。

②同步，指当写(输入)进程把一定数量(如 4KB)的数据写入 pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把他唤醒。当读进程读一空 pipe 时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。

③确定对方是否存在，只有确定了对方已存在时，才能进行通信。

管道是进程间通信的一种简单易用的方法。管道分为匿名管道和命名管道两种。下面首先介绍匿名管道。

匿名管道只能用于父子进程之间的通信，它的创建使用系统调用 pipe()： int pipe(int fd[2])

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

读端和写端，就可以进行通信了。

一个父子进程间使用匿名管道通信的例子。

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

2.2.4 实验步骤

根据残缺代码完善。补充完整后的代码为：

/\*管道通信实验程序残缺版 \*/

#include <unistd.h>

#include <signal.h>

#include <stdio.h>

#include <stdlib.h>

int pid1,pid2; // 定义两个进程变量

int main(){

int fd[2];

char InPipe[5000]; // 定义读缓冲区

char c1='1', c2='2';

while(pipe(fd)==-1); // 创建管道

FILE \*fp;

char buffer[1024];

char command[100];

sprintf(command, "ls -l /proc/%d/fd/", getpid());

system(command);

printf("\n");

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

if(pid1 == 0) { // 如果子进程 1 创建成功,pid1 为进程号

close(fd[0]); // 关闭管道读端

lockf(fd[1],F\_LOCK,0);// 锁定管道

char \*OutPipe="Child process 1 is sending message!";

for(int i=1;i<=2000;i++){

write(fd[1],&c1,1);

} // 分 2000 次每次向管道写入字符’1’

sleep(3); // 等待读进程读出数据

lockf(fd[1],F\_ULOCK,0); // 解除管道的锁定

close(fd[1]);

exit(0); // 结束进程 1

}

else {

while((pid2 = fork()) == -1); // 若进程 2 创建不成功,则空循环

if(pid2 == 0) {

close(fd[0]); // 关闭管道读端

lockf(fd[1],1,0);

char \*OutPipe="Child process 2 is sending message!";

for(int i=1;i<=2000;i++){

write(fd[1],&c2,1);

}// 分 2000 次每次向管道写入字符’2’

sleep(3);

lockf(fd[1],0,0);

close(fd[1]);

exit(0);

} else {

close(fd[1]); // 关闭管道写端

wait(NULL); // 等待子进程 1 结束

wait(NULL); // 等待子进程 2 结束

int total\_read = 0;

int bytes\_read;

while (total\_read < 4000 &&

(bytes\_read = read(fd[0], InPipe + total\_read, 4000 - total\_read)) > 0) {

total\_read += bytes\_read;

}

InPipe[total\_read] = '\0';

printf("%s\n",InPipe); // 显示读出的数据

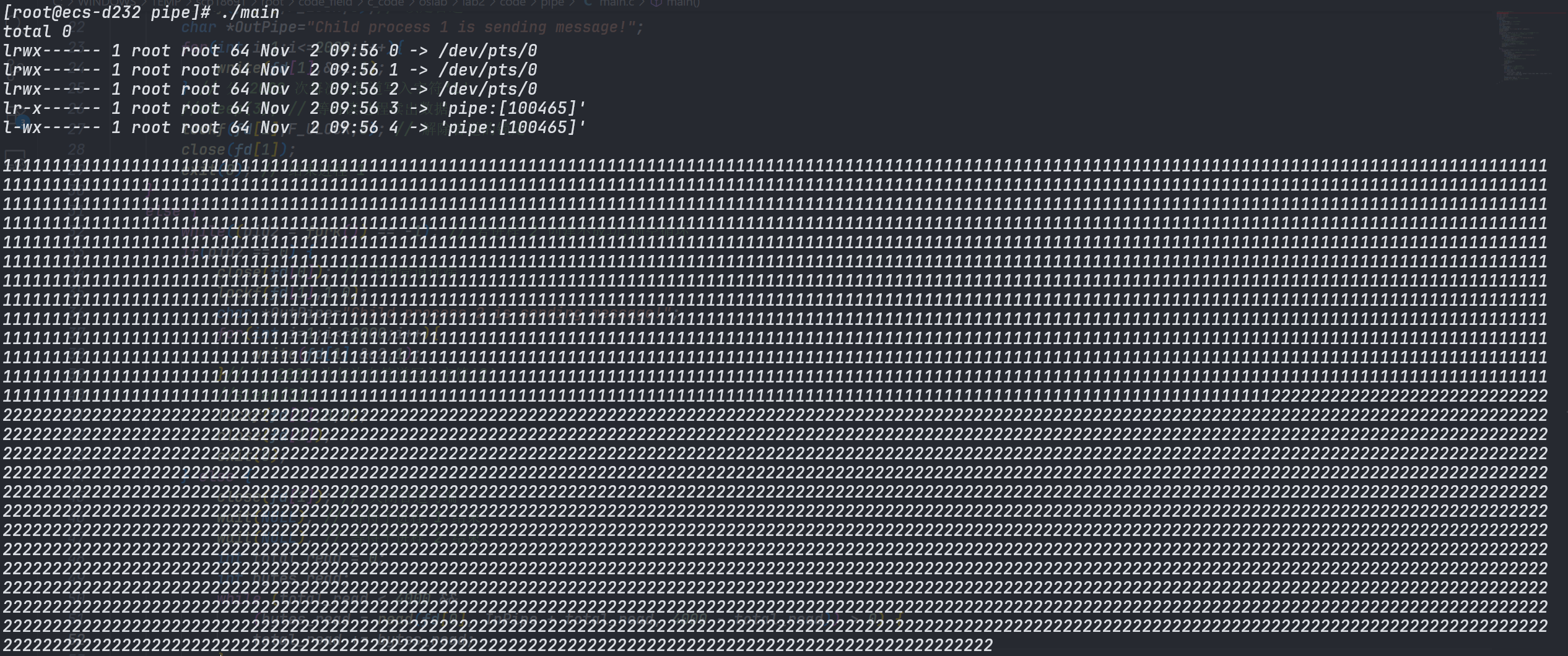
exit(0); // 父进程结束

}

}

}

## 2.2.5 运行结果分析



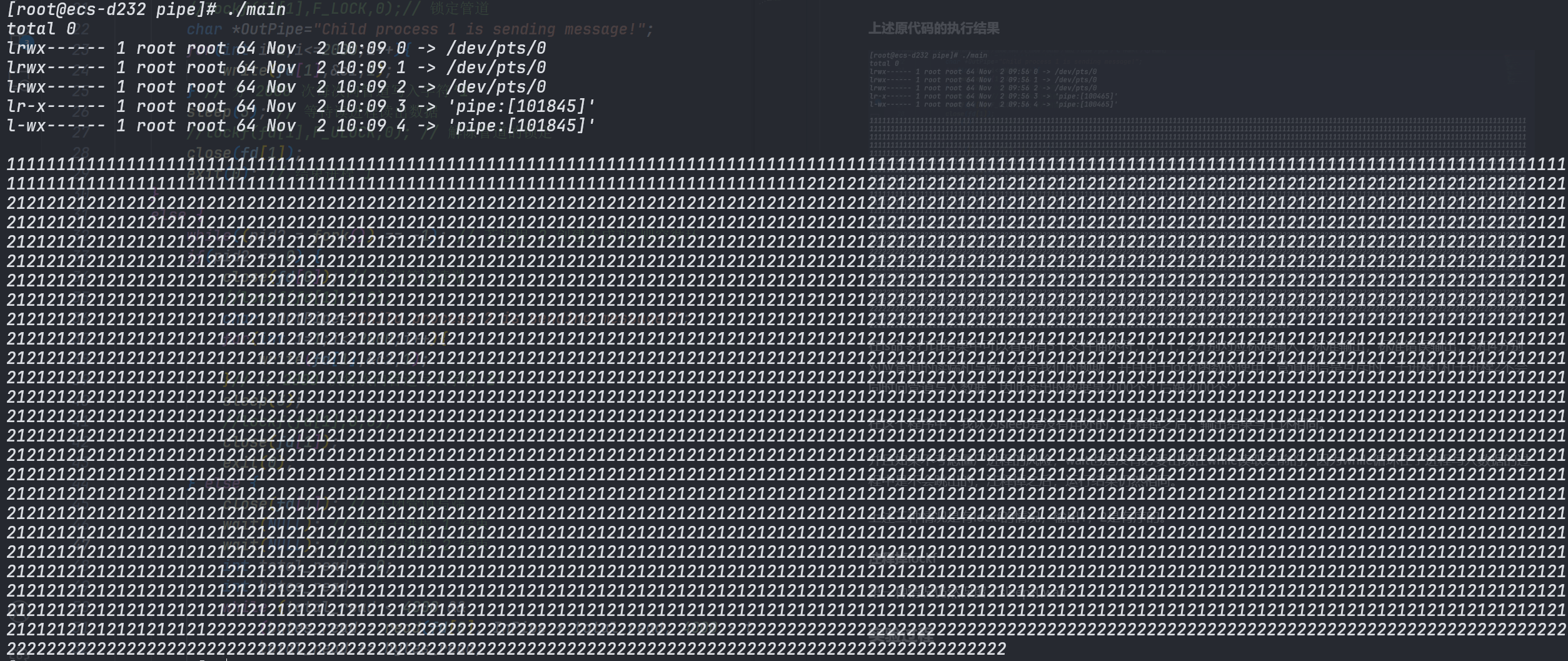
在ls命令打印结果中可以看到有5个文件描述符，0，1，2分别对应标准输入，标准输出，标准错误输出，3和4分别对应管道的读端和写端。符合我们的预期，并且由于lockf函数的使用，管道通信是互斥的，子进程1和子进程2不会同时向管道写入数据，因此读出的数据是2000个'1'后跟2000个'2'。

并且如果不考虑僵尸进程的风险，wait也是没有必要出现在while读取之前的，因为while循环在子进程写入数据的过程中是不会跳出的，注释掉之后，运行结果仍然相同。

上述三种情况是有lockf的情况，输出1，2是有序的。

#### 注释掉lockf

##### 第一种情况仍然保留了sleep和wait

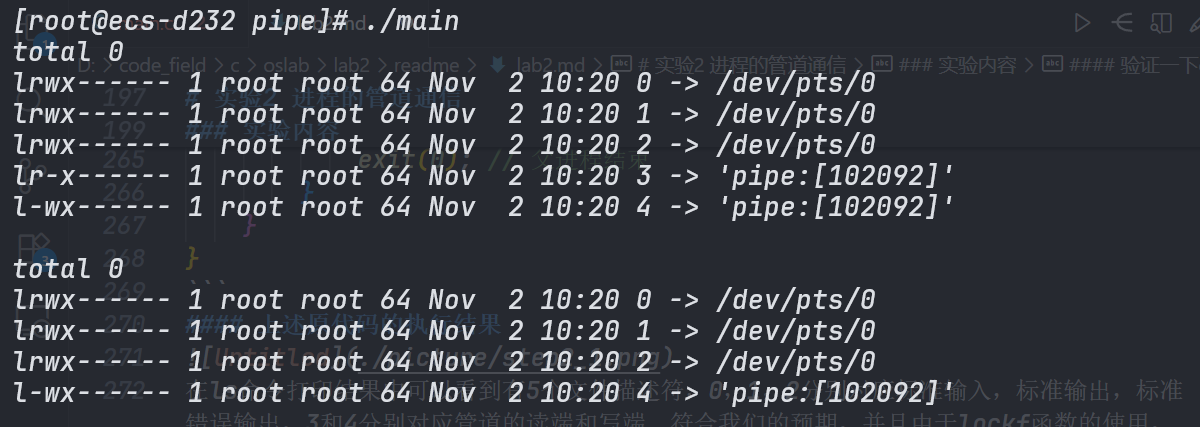


可以看到输出结果是交错的，说明两个子进程在没有锁的情况下同时向管道写入数据，导致读出的数据是混乱的。

##### 注释掉wait，运行结果相同

#### 验证一下close的作用

在其中一个子进程中也调用打印当前进程文件描述符的命令，结果如下：



可见close函数的作用是关闭文件描述符，使得该文件描述符不再可用。

## 2.2.6实验总结

### 2.2.6.1 实验中的问题与解决过程

1. 不理解pipe中的整形数组的作用。上网搜集资料，了解到了文件描述符。
2. 不了解pipe在内存中是如何存在的。上网查阅，创建pipe后，在用户空间我们可以看到两个文件描述符，这是我们访问pipe的钥匙，在内核中创建两个file结构体，对应读端，写端。还有struct innode对象这是管道核心对象。File结构体可以指向管道核心对象。
3. 不理解lockf和close的区别以及作用机理。在file结构体中，有一个引用计数属性，记录有多少个进程正在使用它，当close后，引用计数减1，当减到零时，就表明这个端不在需要了，close常被用作多个子进程写入数据，父进程处理数据，当没有写入端后，read会返回eof，此时就不需要处理数据了。Lockf则锁定文件，避免并发写入造成数据混乱。

### 2.2.6.2 实验收获

1. 互斥锁的重要性：通过这次实验，明确了互斥锁在多进程同步中的重要性。
2. 管道的同步与互斥：理解了如何通过管道来进行进程间的通信，并通过互斥锁来保证管道的同步与互斥。

### 2.2.6.3 意见与建议

### 2.2.7 附件

### 2.2.7.1 附件1 程序



### 2.2.7.2 附件2 Readme

# 2.3 内存的分配与回收

## 2.3.1 实验目的

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

## 2.3.2 实验内容

（1）理解内存分配 FF， BF， WF 策略及实现的思路。

（2）参考给出的代码思路，定义相应的数据结构，实现上述 3 种算法。每种算法要实现内存分配、回收、空闲块排序以及合并、紧缩等功能。

（3）充分模拟三种算法的实现过程，并通过对比，分析三种算法的优劣。

## 2.3.3 实验前准备

关于 OS 的连续内存管理知识

（1） 系统区与用户区

内存管理的基础是对内存的划分。最简单的划分就是每部分仅包含连续的内存区域，这样的区域称为分区。内存一般被划分为两部分：操作系统分区和用户进程分区。当系统中同时运行多个进程时，每个进程都需要有自己独占的分区，这样就形成了多个用户进程分区。每个分区都是连续的，作为进程访问的物理内存，其位置和大小可以由基址寄存器和界限寄存器唯一确定。通过这两个寄存器，系统可以将一个进程的逻辑地址空间映射到其物理地址空间，整个系统仅需要一对这样的寄存器用于当前正在执行的进程。进程分区的起址和大小是保存在进程控制块中的，仅在进程运行时才会装载到基址和界限寄存器中。

操作系统内存管理的任务就是为每个进程分配内存分区，并将进程代码和数据装入分区。每当进程被调度，执行前由操作系统为其设置好基址和界限寄存器。进程在执行过程中CPU 会依据这两个寄存器的值进行地址转换，得到要访问的物理地址。进程执行结束并退出内存后，操作系统回收进程所占的分区。下面讨论三种连续内存管理的特点及管理方法。

（2） 连续内存管理方法

1) 单一连续区分配

这种分配方式仅适用于单用户单任务的系统，它把内存分为系统区和用户区。系统区仅供 OS 使用，用户区供用户使用，任意时刻内存中只能装入一道程序。

2) 固定分区分配

固定分区分配将用户内存空间划分为若干个固定大小的区域，在每个用户分区中可以装入一个用户进程。内存的用户区被划分成几个分区，便允许几个进程驻留内存。操作系统为了完成对固定分区的管理，必须定义一个记录用户分区大小、分区起始地址及分区是否空闲的数据结构。

3) 动态分区分配

这种分配方式根据用户进程的大小，动态地对内存进行划分，根据进程需要的空间大小分配内存。内存中分区的大小和数量是变化的。动态分区方式比固定分区方式显著地提高了内存利用率。

操作系统刚启动时，内存中仅有操作系统分区和一个空闲分区。随着进程不断运行和退出，原始的空闲分区被分割成了大量的进程分区和不相邻的空闲分区。当一个新的进程申请内存时，系统为其分配一个足够大的空闲分区，当一个进程结束时，系统回收进程所占内存。采用动态分区分配方式时，通常可以建立一个空闲分区链以管理空闲的内存区域。

一般不会存在一个空闲分区，其大小正好等于需装入的进程的大小。操作系统不得不把一个大的空闲分区进行拆分后分配给新进程，剩下的放入空闲分区表。空闲分区表中可能有多个大于待装入进程的分区，应该按照什么策略选择分区，会影响内存的利用率。

（3） 常见的动态分区分配算法

1) 首次适应算法。

在采用空闲分区链作为数据结构时，该算法要求空闲分区链表以地址递增的次序链接。在进行内存分配时，从链首开始顺序查找，直至找到一个能满足进程大小要求的空闲分区为止。然后，再按照进程请求内存的大小，从该分区中划出一块内存空间分配给请求进程，余下的空闲分区仍留在空闲链中。

2) 循环首次适应算法。

该算法是由首次适应算法演变而形成的，在为进程分配内存空间时，从上次找到的空闲分区的下一个空闲分区开始查找，直至找到第一个能满足要求的空闲分区，并从中划出一块与请求的大小相等的内存空间分配给进程。

3) 最佳适应算法。

将空闲分区链表按分区大小由小到大排序，在链表中查找第一个满足要求的分区。

4) 最差匹配算法。

将空闲分区链表按分区大小由大到小排序，在链表中找到第一个满足要求的空闲分区。

（4） 动态分区的回收

内存分区回收的任务是释放被占用的内存区域，如果被释放的内存空间与其它空闲分区在地址上相邻接，还需要进行空间合并，分区回收流程如下：

1) 释放一块连续的内存区域。

2) 如果被释放区域与其它空闲区间相邻，则合并空闲区。

3) 修改空闲分区链表。

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

如果被释放区域与其它空闲区间相邻，需要进行空间合并，在进行空间合并时需要考虑以下三种不同的情况：

1) 仅回收区的前面有相邻的空闲分区。如图 3-7a 所示，把回收区与空闲分区 R1 合并成一个空闲分区，把空闲链表中与 R1 对应的结点的分区起始地址作为新空闲区的起始地址，将该结点的分区大小字段修改为空闲分区 R1 与回收区大小之和。

2) 仅回收区的后面有相邻的空闲分区。如图 3-7b 所示，把回收区与空闲分区 R2 合并成一个空闲分区，把空闲链表中与 R2 对应的结点的分区起始地址改为回收区起始地址，将该结点的分区大小字段修改为空闲分区 R2 与回收区大小之和。

3) 回收区的前、后都有相邻的空闲分区。如图 3-7c 所示，把回收区与空闲分区 R1、 R2 合并成一个空闲分区，把空闲链表中与 R1 对应的结点的分区起始地址作为合并后新空闲分区的起始地址，将该结点的分区大小字段修改为空闲分区 R1、 R2 与回收区三者大小之和，删去与 R2 分区对应的空闲分区结点。当然，也可以修改分区 R2 对应的结点，而删去 R1 对应的结点。还可以为新合并的空闲分区建立一个新的结点，插入空闲分区链表，删除 R1 和 R2 对应的分区结点。

(5) 内存碎片

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

## 2.3.4 实验步骤

* 明确主要功能
* 1 - Set memory size (default=1024)
* 2 - Select memory allocation algorithm
* 3 - New process
* 4 - Terminate a process
* 5 - Display memory usage
* 0 - Exit
* 通过键盘输入选择。

完整代码

#include<stdio.h>

#include<stdlib.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; /\*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\*/

// 内存空闲块结构体

struct free\_block\_type{

int size;

int start\_addr;

struct free\_block\_type \*next;

};

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

// 已分配内存块结构体

struct allocated\_block{

int pid;

int size;

int start\_addr;

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

struct allocated\_block \*next;

};

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

/\*初始化空闲块，默认为一块，可以指定大小及起始地址\*/

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");

}

int set\_mem\_size(){

int size;

if(flag!=0){ //防止重复设置

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

return 0;

}

printf("Total memory size =");

scanf("%d", &size);

if(size>0) {

mem\_size = size;

free\_block->size = mem\_size;

}

flag=1;

return 1;

}

int free\_num(){

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

int count=0;

while(fbt!=NULL){

count++;

fbt=fbt->next;

}

return count;

}

/\*按 FF 算法重新整理内存空闲块链表\*/

void rearrange\_FF(){

//请自行补充

int n=free\_num();

if(n<=1) return;

int i=1;

while(i<n){

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

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

while(current!=NULL && current->next!=NULL){

if(current->start\_addr > current->next->start\_addr){

struct free\_block\_type \*temp=current->next;

current->next=temp->next;

temp->next=current;

if(pre==NULL){

free\_block=temp;

}else{

pre->next=temp;

}

pre=temp;

}else{

pre=current;

current=current->next;

}

}

i++;

}

//将空闲区链表合并

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

while(fbt!=NULL && fbt->next!=NULL){

if(fbt->start\_addr + fbt->size == fbt->next->start\_addr){

struct free\_block\_type \*temp=fbt->next;

fbt->size += temp->size;

fbt->next=temp->next;

free(temp);

}else{

fbt=fbt->next;

}

}

}

/\*按 BF 算法重新整理内存空闲块链表\*/

void rearrange\_BF(){

//请自行补充

int n=free\_num();

if(n<=1) return;

int i=1;

while(i<n){

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

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

while(current!=NULL && current->next!=NULL){

if(current->size > current->next->size){

struct free\_block\_type \*temp=current->next;

current->next=temp->next;

temp->next=current;

if(pre==NULL){

free\_block=temp;

}else{

pre->next=temp;

}

pre=temp;

}else{

pre=current;

current=current->next;

}

}

i++;

}

}

/\*按 WF 算法重新整理内存空闲块链表\*/

void rearrange\_WF(){

//请自行补充

int n=free\_num();

if(n<=1) return;

int i=1;

while(i<n){

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

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

while(current!=NULL && current->next!=NULL){

if(current->size < current->next->size){

struct free\_block\_type \*temp=current->next;

current->next=temp->next;

temp->next=current;

if(pre==NULL){

free\_block=temp;

}else{

pre->next=temp;

}

pre=temp;

}else{

pre=current;

current=current->next;

}

}

i++;

}

}

/\*按指定的算法整理内存空闲块链表\*/

void rearrange(int algorithm){

switch(algorithm){

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

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

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

}

}

/\*设置当前的分配算法\*/

void set\_algorithm(){

int algorithm;

printf("\t1 - First Fit\n");

printf("\t2 - Best Fit \n");

printf("\t3 - Worst Fit \n");

scanf("%d", &algorithm);

if(algorithm>=1 && algorithm <=3)

ma\_algorithm=algorithm;

//按指定算法重新排列空闲区链表

rearrange(ma\_algorithm);

}

/\*创建新的进程，主要是获取内存的申请数量\*/

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

int new\_process(){

struct allocated\_block \*ab;

int size; int ret;

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

if(!ab) exit(-5);

ab->next = NULL;

pid++;

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

ab->pid = pid;

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

scanf("%d", &size);

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

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

/\*如果此时 allocated\_block\_head 尚未赋值，则赋值\*/

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("Allocation fail\n");

free(ab);

return -1;

}

return 3;

}

int process\_num(){

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

int count=0;

while(ab!=NULL){

count++;

ab=ab->next;

}

return count;

}

void delete\_free\_list(){

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

struct free\_block\_type \*temp;

while(fbt!=NULL){

temp=fbt;

fbt=fbt->next;

free(temp);

}

free\_block=NULL;

}

void mem\_comp(){

int n=process\_num();

if(n<=1) return;

int i=1;

//先按地址排序

while(i<n){

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

struct allocated\_block \*pre=NULL;

while(current!=NULL && current->next!=NULL){

if(current->start\_addr > current->next->start\_addr){

struct allocated\_block \*temp=current->next;

current->next=temp->next;

temp->next=current;

if(pre==NULL){

allocated\_block\_head=temp;

}else{

pre->next=temp;

}

pre=temp;

}else{

pre=current;

current=current->next;

}

}

i++;

}

//重新计算已分配区的起始地址

int total\_size=0;

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

if(ab==NULL) return;

total\_size+=ab->size;

ab->start\_addr=DEFAULT\_MEM\_START;

while(ab!=NULL&&ab->next!=NULL){

ab->next->start\_addr=total\_size;

total\_size+=ab->next->size;

ab=ab->next;

}

delete\_free\_list();

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

free\_block->start\_addr=total\_size;

free\_block->size=mem\_size - total\_size;

free\_block->next=NULL;

}

/\*分配内存模块\*/

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

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

int request\_size=ab->size;

fbt = pre = free\_block;

//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但分配后剩余空间比较小，则一起分配

// 3. 找不到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

//请自行补充。。。。。

int total\_free\_size = 0;

while(fbt != NULL){

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

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

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

fbt->start\_addr += request\_size;

fbt->size -= request\_size;

}else{

if(pre == fbt){

free\_block = fbt->next;

}else{

pre->next = fbt->next;

}

ab->size = fbt->size;

free(fbt);

}

rearrange(ma\_algorithm);

return 1;

}

total\_free\_size += fbt->size;

pre = fbt;

fbt = fbt->next;

}

if(total\_free\_size >= request\_size){

mem\_comp();

return allocate\_mem(ab);

}

return -1;

}

/\*删除进程，归还分配的存储空间，并删除描述该进程内存分配的节点\*/

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

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

while(ab!=NULL){

if(ab->pid==pid) return ab;

ab=ab->next;

}

return NULL;

}

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

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

void kill\_process(){

struct allocated\_block \*ab;

int pid;

printf("Kill Process, pid=");

scanf("%d", &pid);

ab=find\_process(pid);

if(ab!=NULL){

free\_mem(ab); /\*释放 ab 所表示的分配区\*/

dispose(ab); /\*释放 ab 数据结构节点\*/

}

}

/\*将 ab 所表示的已分配区归还，并进行可能的合并\*/

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

int algorithm = ma\_algorithm;

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

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

if(!fbt) return -1;

// 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

// 4. 将空闲链表重新按照当前算法排序

//请自行补充……

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

fbt->size = ab->size;

fbt->next = NULL;

if(free\_block == NULL){

free\_block = fbt;

}

else{

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

while(temp->next != NULL){

temp = temp->next;

}

temp->next = fbt;

}

rearrange\_FF(); //先按地址排序

if(ma\_algorithm != MA\_FF)

rearrange(ma\_algorithm); //再按当前算法排序

return 1;

}

/\*释放 ab 数据结构节点\*/

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;

}

/\* 显示当前内存的使用情况，包括空闲区的情况和已经分配的情况 \*/

int display\_mem\_usage(){

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

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

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

struct allocated\_block \*temp;

while(ab!=NULL){

temp=ab;

ab=ab->next;

free(temp);

}

allocated\_block\_head=NULL;

delete\_free\_list();

}

int 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

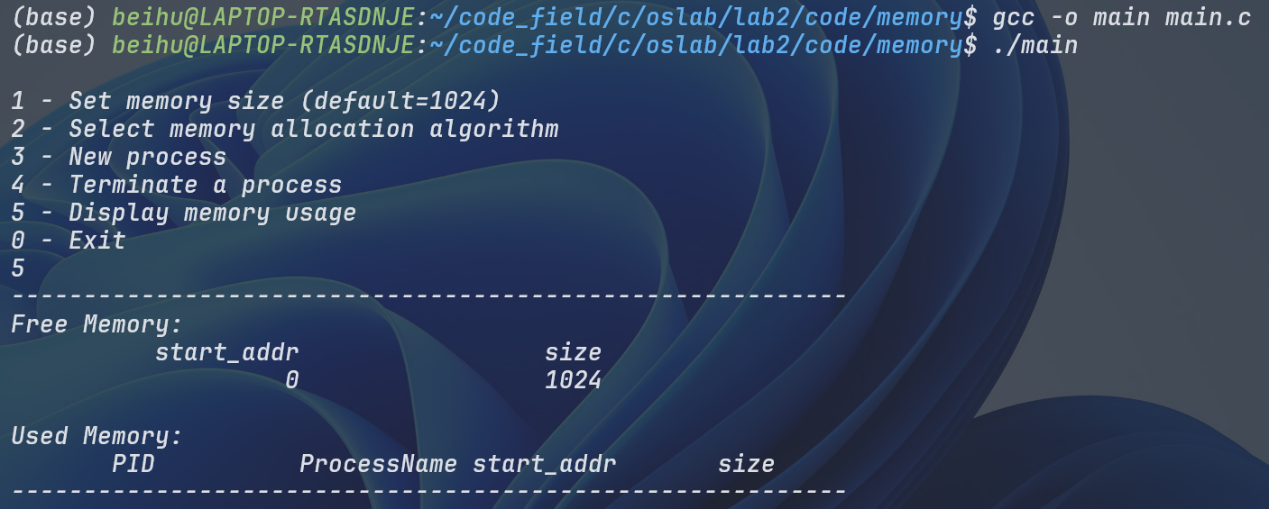
case '0': do\_exit(); exit(0); //释放链表并退出

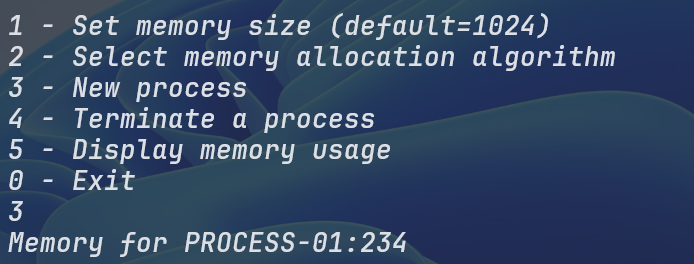
default: break;

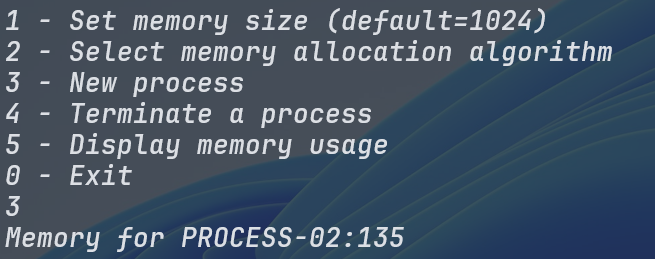
} }

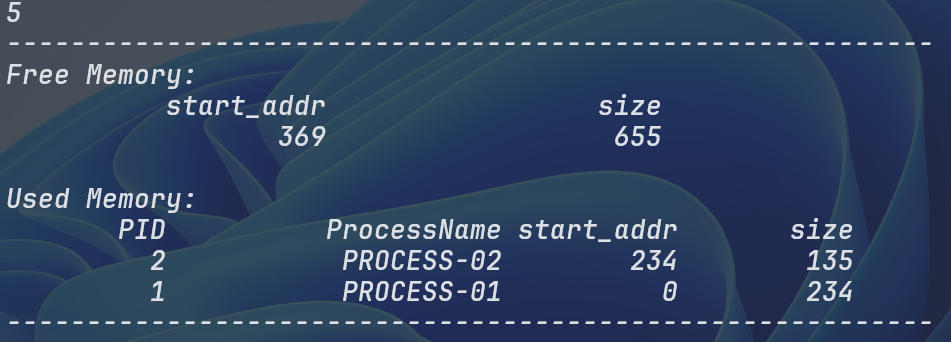
}

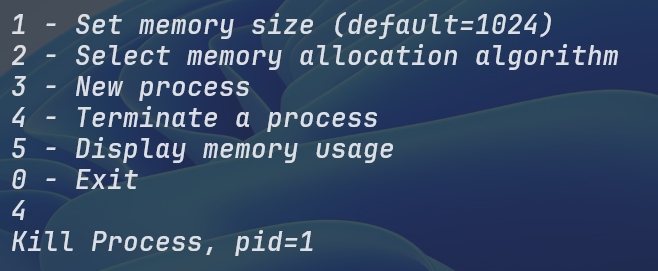
## 2.3.5 程序运行初值及运行结果分析



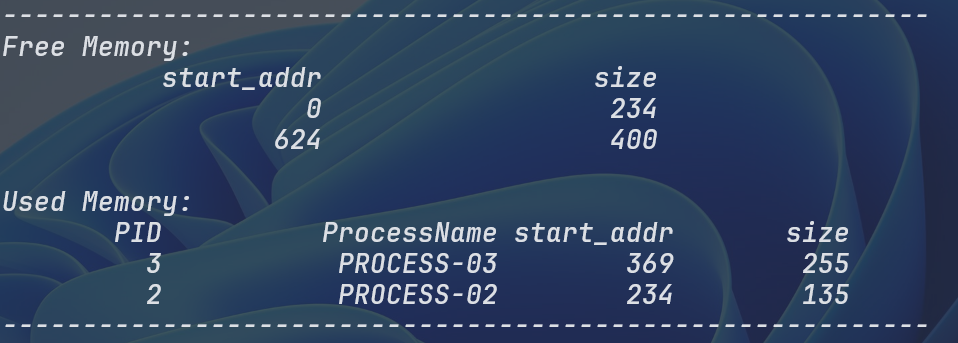












## 2.3.6 实验总结

### 2.3.6.1 实验中的问题与解决过程

实验中的问题大部分都是coding时的思路问题。

### 2.3.6.2 实验收获

对于FF的空闲块排序（事实上是不需要排序的，因为无论如何排列，只要找到第一个可用的就行），对于BF、WF我采用了简单的冒泡排序，这种代码没有递归，避免栈溢出。

### 2.3.6.3 意见与建议

本次实验难度中等，适合作为初学内存分配算法时的实践作业

## 2.3.7 附件

### 2.3.7.1 附件1 程序



### 2.3.7.2 附件2 Readme



# 实验三 类EXT2文件系统编写

## 3.1 实验目的

通过一个简单文件系统的设计，理解文件系统的内部结构、基本功能及实现

## 3.2 实验内容

1) 分析 EXT2 文件系统的结构；

2) 基于 Ext2 的思想和算法，设计一个类 Ext2 的多级文件系统，实现 Ext2 文件系统的一

个功能子集；

3) 用现有操作系统上的文件来代替硬盘进行硬件模拟。

## 3.3 实验思想

在分析 Linux 的文件系统的基础上，基于 Ext2 的思想和算法，设计一个类 Ext2 的虚拟多级文件系统，实现 Ext2 文件系统的一个功能子集。并且用现有操作系统上的文件来代替硬盘进行硬件模拟。设计文件系统应该考虑的几个层次：①介质的物理结构；②物理操作——设备驱动程序完成；③文件系统的组织结构（逻辑组织结构）；④对组织结构其上的操作；⑤为用户使用文件系统提供的接口。本实验只涉及后三个层次的内容。

## 3.4 实验步骤

代码部分太长，我附到附件里面了。

1. **进行基本的宏定义**

#define VOLUME\_NAME "EXT2FS" //卷名

#define BLOCK\_SIZE 512 //块大小

#define NUM\_BLOCK 4611 //磁盘总块数

#define GDT\_START 0

#define GD\_SIZE 32

//一些起始字节定义

#define BLOCK\_BITMAP 512

#define INODE\_BITMAP 1024

#define INODE\_TABLE 1536

#define INODE\_SIZE 64

#define INODE\_TABLE\_COUNTS 4096

#define DATA\_BLOCK 263680

#define DATA\_BLOCK\_COUNTS 4096

#define USER\_MAX 3

#define FOPEN\_TABLE\_MAX 16

#define MAX\_FILE\_SIZE (2\*1024\*1024-18\*512)

//文件类型宏定义

#define FILE\_UNKNOWN 0

#define FILE\_FILE 1

#define FILE\_DIR 2

1. **基本数据结构定义**

本实验要求实现变长目录，因此我的目录项中使用了柔性数组，并定义了辅助函数来根据名字获取对齐后所占用的实际字节数。

struct dir\_entry {

unsigned short inode;

unsigned short rec\_len;

unsigned char name\_len;

unsigned char file\_type;

char name[]; // 柔性数组

};

static inline unsigned short calculate\_rec\_len(unsigned char name\_len)

{

unsigned short base\_len = DIR\_ENTRY\_HEADER\_SIZE + name\_len;

return (base\_len + 3) & ~3; // 4字节对齐

}

组描述符，inode节点的数据结构定义

struct group\_desc {

char bg\_volume\_name[16];

unsigned short bg\_block\_bitmap;

unsigned short bg\_inode\_bitmap;

unsigned short bg\_inode\_table;

unsigned short bg\_free\_blocks\_count;

unsigned short bg\_free\_inodes\_count;

unsigned short bg\_used\_dirs\_count;

char bg\_pad[4];

};

struct inode {

unsigned short i\_mode;

unsigned short i\_blocks;

unsigned short i\_uid;

unsigned short i\_gid;

unsigned short i\_links\_count;

unsigned short i\_flags;

unsigned long i\_size;

unsigned long i\_atime;

unsigned long i\_ctime;

unsigned long i\_mtime;

unsigned long i\_dtime;

unsigned short i\_block[8]; // 0-5 direct, 6: single indirect, 7: double

char i\_pad[16];

};

1. **一些在内存中存在的量的定义**

static unsigned short last\_alloc\_inode;

static unsigned short last\_alloc\_block;

static unsigned short current\_dir; //当前所在的inode号

static unsigned short current\_dirlen;

static short fopen\_table[FOPEN\_TABLE\_MAX]; //文件打开表

static struct group\_desc group\_desc\_buf; //组描述符缓冲区，用于读取与更新

static struct inode inode\_buf; //inode信息的缓冲区

static unsigned char bitbuf[BLOCK\_SIZE]; //bit\_map的存储变量

static unsigned char ibuf[BLOCK\_SIZE]; //inode\_map

static char Buffer[BLOCK\_SIZE]; //非目录块的数据缓冲区

static char tempbuf[MAX\_FILE\_SIZE]; //辅助缓冲区，用于一些辅助操作，防止//污染其他缓冲区

static char dir\_buf[BLOCK\_SIZE]; //目录缓冲区

char current\_path[256];

char current\_user[10];

char current\_disk[64];

1. **模拟各种磁盘信息加载到内存中**

static void update\_group\_desc();

static void load\_group\_desc();

static void update\_block\_bitmap();

static void load\_block\_bitmap();

static void update\_inode\_bitmap();

static void load\_inode\_bitmap();

static void update\_inode\_entry(unsigned short i);

static void load\_inode\_entry(unsigned short i);

static void update\_dir(unsigned short i);

static void load\_dir(unsigned short i);

static void update\_data\_block(unsigned short i) ;

static void load\_data\_block(unsigned short i);

//临时将新的inode信息读到指定缓冲区

static void load\_inode\_to\_buf(unsigned short i, struct inode\* buf) ;

1. **定义inode，block的分配释放函数**

分配释放的基本流程

读入相应位图--->查找--->更新位图--->更新组描述符

static int alloc\_block()；

static int alloc\_inode() ；

static void remove\_block();

static void remove\_inode();

1. **多级索引机制的一些辅助函数**

在我的设计中，删除某一个块后，我会将该块的后续块进行前移，因此他们逻辑上是连续的，我可以根据逻辑块号来定位到他在inode中的位置，依此来获取它的实际块号。

下面是一些根据一级及二级索引中的逻辑块号获取物理块号的函数

unsigned short get\_index\_one(unsigned short logic\_block\_num, unsigned short inode\_i) ；

unsigned short get\_index\_two(unsigned short logic\_block\_num, unsigned short inode\_i) ；

1. **目录项构造的辅助函数**

在新建一个文件时，我需要给他分配inode信息，但暂时不分配数据块，但是对于文件夹，不仅要分配inode，还要分配一个数据块并填入基本的目录信息，由于要实现变长目录，需要根据name计算rec\_len,所以我定义了一个创建目录项结构体的辅助函数。

//根据名字以及类型返回它的相应结构体

struct dir\_entry\* create\_dir\_entry(unsigned short inode, unsigned char file\_type, const char\* name) {

size\_t nlen = strlen(name);

unsigned char name\_len = (unsigned char)nlen;

unsigned short rec\_len = calculate\_rec\_len(name\_len);

struct dir\_entry\* entry = (struct dir\_entry\*)malloc(rec\_len);

if (!entry) return NULL;

entry->inode = inode;

entry->rec\_len = rec\_len;

entry->name\_len = name\_len;

entry->file\_type = file\_type;

memset(entry->name, 0, name\_len + 1);

memcpy(entry->name, name, name\_len);

return entry;

}

1. **mkdir与create的实现**

这两者的实现逻辑上大致是一样的，不一样的就是目录项的创建要分配数据块并写入”.”和”..“两个目录项，因此定义了如下函数

static void init\_dir\_entry(unsigned short inode\_i, unsigned char file\_type, unsigned short mode) ；

该函数可以接受一个文件类型。

基本思路就是。

先加载该文件夹的inode信息至内存中，如果是目录，就分配一个数据块，并用memset字节级别的操作将目录项信息写入内存的目录项缓存中，然后更新用该缓存更新新分配的块，最后更新inode信息。

如果类型是文件的话，就不需要分配数据块了，更新inode即可。

有了这个辅助函数，还要再定义一个help\_code函数，它用于辅助mkdir和create，原因很简单，mkdir和create的思路类似，代码有大部分的重合之处。

代码实现：

void help\_code(const char \*name, unsigned char file\_type)；

help\_code的作用：

1. 在当前目录下，创建文件或目录，在这之前，要首先遍历，查找是否存在。
2. 如果不存在，就分配inode给要创建的文件或目录，然后调用上面的init\_dir\_entry,初始话该文件或目录的inode信息以及可能存在的数据块信息。
3. 最后将新目录项插入到当前目录项中，基本思路是，先尝试直接块，将直接块读取到dir\_buf中，然后遍历寻找空缺的项插入即可，然后是一级索引，二级索引，一级如果块不够，要追加块，在这里就不再细说了。
4. 有一个重要的点就是，我设计的删除目录项，是和EXT2类似的，就是我并不真正删除它，而是将inode置0，并将它的rec\_len并入到上一个目录项中，所以我的find\_file实现如下:

/\* find\_file: 在 current\_dir 下查找 name，返回 inode\_num / block逻辑索引 / start\_byte / prev\_start\_byte

file\_type: FILE\_FILE 或 FILE\_DIR

\*/

static unsigned short find\_file(const char \*name, unsigned char file\_type,

unsigned short\* inode\_num, unsigned short\* block\_num,unsigned short\* start\_byte, unsigned short\* prev\_start\_byte) ；

查找是否存在的同时，如果存在，要返回上一个目录项的起始字节，以及它的起始字节，方便遍历与删除。

上面的函数定义好后，mkdir和create就很简单了。

void mkdir\_cmd(char \*name) {

help\_code(name, FILE\_DIR);

}

void create\_file\_cmd(char \*name) {

help\_code(name, FILE\_FILE);

}

1. **rmdir,rm以及delete的实现**

他们的思路也是类似的，rmdir是删除空目录，rm是递归删除目录，delete是删除文件。

基本思路就是，先清理掉要删除的那一项的inode信息以及释放掉它分配的数据块，然后更新它的父目录，在更新时，涉及到如果删除后数据块为空，要将数据块释放掉，并前移inode的逻辑块。

下面的del\_help就是做这个工作的。

void del\_help(const char \*name, unsigned char file\_type) ；

定义好del\_help后，如果要删除文件，直接调用del\_help即可。

Rmdir的实现，就是先检查所有块判断是否为空，为空就调用del\_help即可。

Rm的实现采用递归，函数remove\_dir()是递归删除当前目录下的所有内容，实现思路就是，遍历当前inode的所有块，并将块读入到dir\_buf中，然后遍历dir\_buf,如果是目录，就调用remove\_dir,否则，是文件，直接调用del\_help,最后，在删除所有内容后，目录为空，调用rmdir即可。

1. **文件读写**

文件打开操作：首先查找文件是否存在，如果存在，在文件打开表中寻找该文件是否已经打开，如果没有，就在文件打开表中添加一项，并修改文件的访问时间即可。

Close:首先查找文件是否存在，如果存在，在文件打开表中寻找该文件是否已经打开，如果已经打开，就在修改文件打开表，删除一项即可。

Read操作：首先检查文件是否打开，如果打开，则遍历所有逻辑块，获得对应的数据块，将数据块内容读取到buffer中，输出即可。

Write覆盖写入：直接删除文件的所有块，重新写入数据

Write追加写入：定位到最后一个逻辑块的最后一个字节，在该字节后面追加写入数据即可。

1. **Cd操作**

首先定义cd\_onestep,它首先解析cd的名字，如果是”.”修改访问时间即可。

否则，使用find\_file查找文件或目录，并返回inode(包括查找”..”项),并将current\_dir设置为inode，并根据名字，更新current\_path，最后更新访问时间，更新inode即可。

然后定义cd，cd就用strtok以”/”为分隔符解析字符串，依次调用cd\_onestep即可。

1. **Ls操作**

Ls操作就是先把当前目录inode写入inode的buffer中，然后遍历它的所有块，将块信息加载到dir\_buf中，使用offset作为偏移量，来遍历目录项，获得inode号，然后使用一个临时缓冲区，防止污染当前的缓冲区，将它的inode信息读入到该缓冲区中，打印名字，时间即可。

1. **Format**

Format操作就是初始化磁盘和初始化内存。

Initialize\_disk初始化磁盘就是先检查是否有打开的模拟磁盘的文件，如果没有的话，就创建。

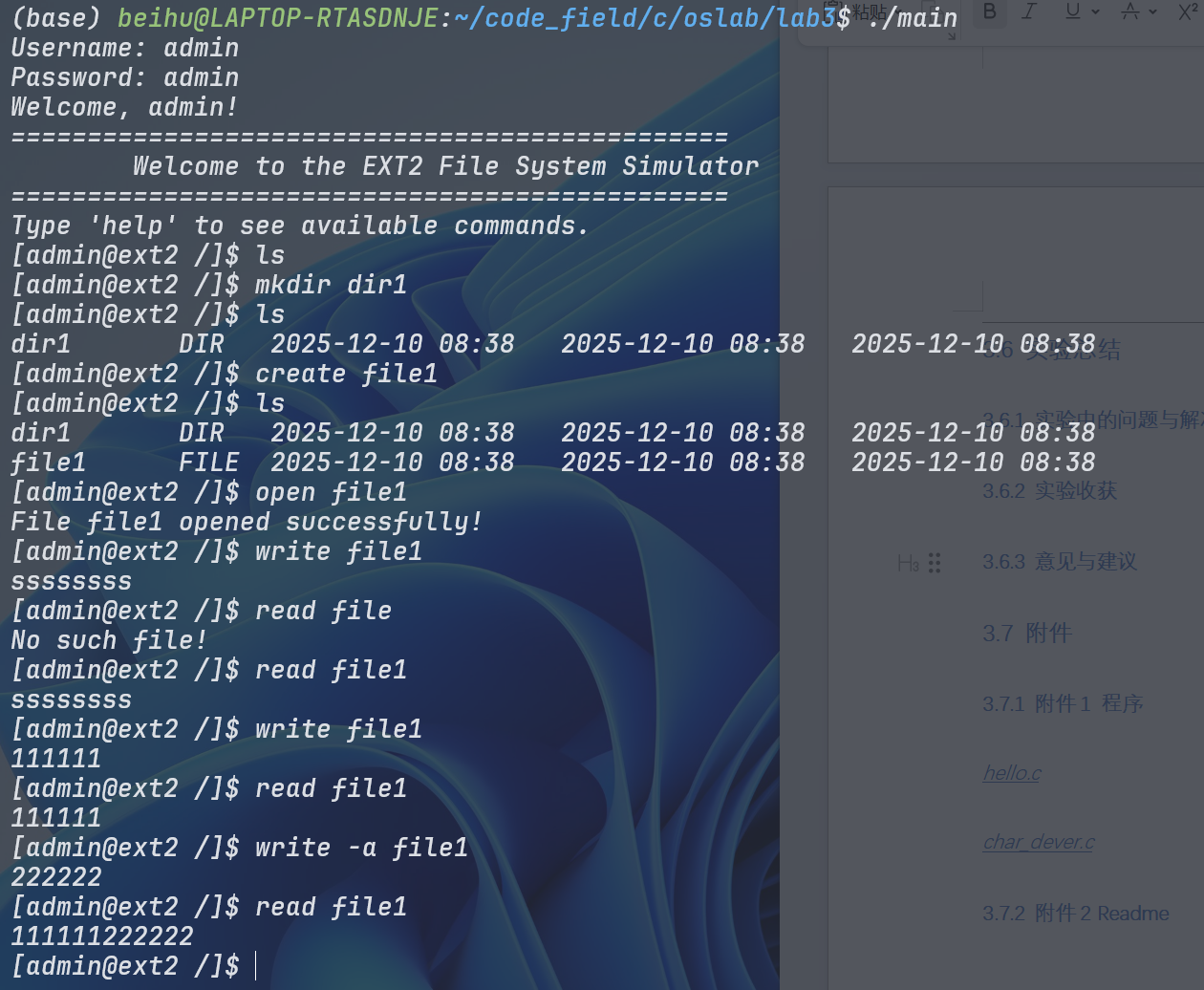
之后就是初始化存在与内存中的一些变量，将其内容全部设为0(空)。

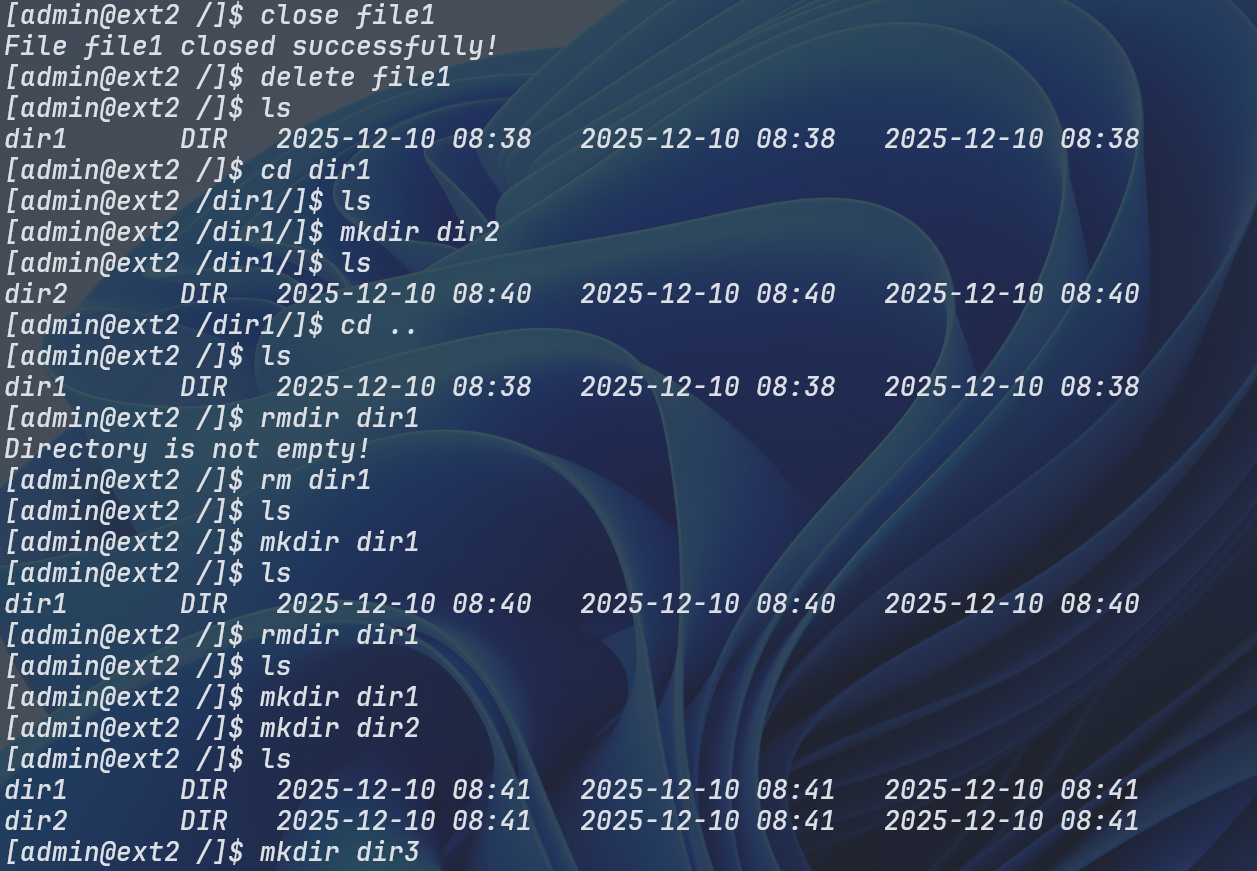
然后是initialize\_memory,该函数初始化current\_path，构建一个初始的路径字符串显示在终端，然后current\_dir=0,current\_dirlen=0,最后是分配一个root\_inode，并创建初始化根目录项信息。

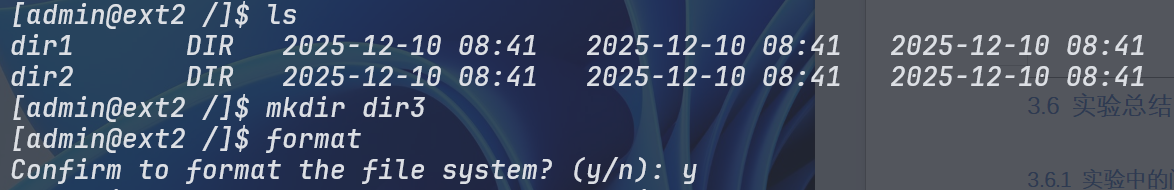
1. **Login**

打开保存的userlist.txt文件，然后输入用户名，和密码，和文件中保存的作比较即可。

# 3.5 程序运行初值及运行结果分析









## 3.6 实验总结

### 3.6.1 实验中的问题与解决过程

1.不知道从何写起。借鉴网络上的一些源码慢慢有了思路

2.对于变长目录项，不知道如何遍历与管理。通过学习memcpy等一系列字节级别操作的函数实现了变长目录项。

3.对于大量字符串操作，包括解析等等感到很困难，不太熟悉c的一些库函数的用法。阅读技术文档学习了一些库函数的用法。

### 3.6.2 实验收获

通过类EXT2简单文件系统的编写，我对文件系统的各种细节有了更好，更深入的理解，也提高了我的编程能力。

### 3.6.3 意见与建议

我个人觉得从零写起的难度偏高，或许可以给同学们一些参考源码。

## 3.7 附件

### 3.7.1 附件1 程序



### 3.7.2 附件2 Readme

