 华中农业大学课程项目报告

课程名称：操作系统实验

姓名：唐平

班级：计科1502

学号：2015317200405

信息学院计算机科学系

2017年12月27日

**实验内容：**

1. **进程调度**
2. **死锁**
3. **存储管理**
4. **文件系统**

**实验一：进程调度**

1. **实验目的**

进程调度是处理及管理的核心内容，本次实验要求用C语言编写和调试一个简单的进程调度程序。调度算法可以任意选择或自行设计，例如简单轮转法和优先数法等。通过本次实验可以加深各使用进程控制块进行进程调度和各种调度算法的理解及其实施方法。

1. **实验内容**
2. **题目**

采用C语言编写程序，选用优先数调度算法或简单轮转法对五个进程进行调度，每个进程处于运行（Run）、就绪（Ready）和完成（Finish）三种状态之一，并假定起始状态为就绪状态。

1. **进程控制块结构**

进程标识符

连接指针

优先数 | 轮转时间片

占用的CPU时间片数

进程所需时间片数

进程状态

1. **算法说明**

（1）为了便于处理，程序中进程的运行时间以时间片为单位进行计算，各进程的优先数或轮转时间片数，以及进程需要运行的时间片数，其初始值均有用户给定

( 2 ) 优先数法

进程就绪队列按优先数大小从高到低排列，链首进程首先投入运行。进程每执行一次，进程需要的时间片数减1、该进程的优先数减3。这样，该进程如果在一个时间片中没有完成，其优先数降低一级。接着仍是用该进程降低一级后的优先数与就绪队列中链首进程的优先数进行比较，如果仍是该进程的优先数高或相同，便让该进程继续执行；否则，调度就绪队列的链首进程投入运行。原运行过的进程按其现行优先数大小插入就绪队列，且改变它们对应的进程状态，一直到所有进程都运行完各自的时间片数。

1. **实验步骤**
2. **任务分析**

本实验要实现的是使用C语言模拟实现进程调度，此处采用的算法是优先数算法。本实验重在理解进程调度的过程，明白操作系统进程调度程序是怎么对多个进程进行调度，不产生饥饿现象**。**

1. **输入的形式和输入值的范围**

输入的是PCB结构体的优先数、进程所需时间片数（都是用int类型表示）；为保证进程有意义，输入范围大于0；进程生成后，当前状态自动初始化为ready

1. **输出的形式**

每个进程输出有三个属性:优先数、所需时间片数、进程当前状态

1. **程序所能达到的功能**

在手动控制进程输入后，对进程调度的每一步进行检测，按步执行。

1. **测试数据：**

优先数：5 时间片数：3

优先数：9 时间片数：3

优先数：15 时间片数：3

1. **概要设计**
2. *//进程三种状态*#define **PROCESS\_RUN** 1  
   #define **PROCESS\_READY** 0  
   #define **PROCESS\_FINISH** 2

*//对进程控制块进行初始化***void** init(**struct** P\_PCB \*pre );

*//优先数法***typedef struct** P\_PCB{  
 **int** process\_sign; *//进程标识符* **int** process\_priority\_num; *//进程优先数  
 // int cpu\_time; //进程所占CPU时间片数* **int** cpu\_need\_time; *//进程所需时间片数* **int** process\_status; *//进程当前状态* **struct** P\_PCB \* next;  
}\*pPCB;  
  
 *//将链表中的结构体按优先数从大到小的规律排序***void** insert\_for\_sort\_operation(**struct** P\_PCB\*\* ppNode, **struct** P\_PCB\* pNode);  
  
**void** sort\_for\_link\_node(**struct** P\_PCB\*\* ppNode);  
  
*//优先数法***void** priority(**struct** P\_PCB \*\*run,**struct** P\_PCB \*\*ready,**struct** P\_PCB \*\*tail,**struct** P\_PCB \*\*finish);

1. **详细设计**
2. **进程控制块的链结构：**

Run Ready Tail

1

0

……

^

……

……

……

^

Finish

……

……

……

^

其中：Run——表是当前运行进程指针

Ready——就绪队列头指针

Tail——就绪队列尾指针

Finish——完成队列指针

1. **程序流程图：**

开始

生成并按优先数大小排列进程控制块

链首进程投入运行

时间片到，进程时间片数减1，优先数减3

进程时间片数为0？

撤销该进程

从链首取一个进程投入运行

优先数大于链首进程？

运行进程推出，按优先数插入进程链

进程队列空？

结束

是

否

1. **各函数之间的调用关系**

进程链表创建：**void** init(**struct** P\_PCB \*pre );

创建完之后完成对链表的排序： **void** insert\_for\_sort\_operation(**struct** P\_PCB\*\* ppNode, **struct** P\_PCB\* pNode);将进程节点插入已经排好序的链表ppNode  
**void** sort\_for\_link\_node(**struct** P\_PCB\*\* ppNode);将传入的链表差分，然后调用insert\_for\_sort\_operation()函数进行重新组装,完成排序

进程调度算法： **void** priority(**struct** P\_PCB \*\*run,**struct** P\_PCB \*\*ready,**struct** P\_PCB \*\*tail,**struct** P\_PCB \*\*finish);进程排序之后，将进程链链首传入函数进行进程调度的实现

1. **调试分析**

程序实现过程出现的一个比较大的问题是指针指向的问题：指向准备就绪部分首地址的指针ready，因为在执行过程中未及时进行更新，导致进程链重新排序后通过给链首指针run指针赋值 \*run = \*ready，使得run未能正确得到更新后的链表序列，执行几轮后报错。解决的办法是在每次进程执行一次之后，使用run来刷新ready的值，保证ready及时更新，就不会出现这类错误。

指针在使用过程中，一定得小心，时刻注意指针是否正确指向对的位置，另外，当链表发生更新及顺序变换后，各个记录不同段位置的指针也要及时更新

1. **测试结果**

****

****

****

1. **使用说明**

程序运行后有详细提示，按照程序执行步骤操作即可。

1. **实验总结**

本次实验是使用C语言模拟实现操作系统的进程调度功能，本次主要使用优先数算法实现。在优先数算法中，首先需要根据各进程的优先数对进程进行从大到小的排序，优先级最高的进程最先获得时间片运行。当然，为了保证优先数小的进程不产生饥饿现象，时间片不能过大并且优先数要以合适的数量级进行递减。

但是存在一种现象：因为实验中使用的是int类型的数据代表优先数，可能存在的情况是当优先数为负后进程仍未运行结束，这时进程优先数为负,由此联想到真实系统中的优先数设置，为了避免这种情况的出现，在表示优先数的二进制位中必须设置符号位（如果真实系统也是这种调度方法的话），因为进程的优先数不能为0后就不继续往下递减，如果不往下继续递减的话，要是有多个进程优先数同时为0并且都为结束，调度算法就无法对运行进程进行选择，由此出现混乱的情况。

PCB结构体中有个‘占用CPU时间片数’属性，一直没搞懂其作用是什么，最开始以为是进程当前占用时间片数，但是进程执行是以时间片为单位，执行一次之后，需要再次进行优先数比较后才能判断是否继续执行，所以占用多个时间片意义不大。所以在程序中，我将该属性删除掉未使用。

1. **附录**

**Pcb.h**

#ifndef **PCBPROJECT\_PCB\_H**#define **PCBPROJECT\_PCB\_H**#include **<stdio.h>**#include **<stdlib.h>**#include **<string.h>***//进程三种状态*#define **PROCESS\_RUN** 1  
#define **PROCESS\_READY** 0  
#define **PROCESS\_FINISH** 2  
  
*//优先数法***typedef struct** P\_PCB{  
 **int** process\_sign; *//进程标识符* **int** process\_priority\_num; *//进程优先数  
 // int cpu\_time; //进程所占CPU时间片数* **int** cpu\_need\_time; *//进程所需时间片数* **int** process\_status; *//进程当前状态* **struct** P\_PCB \* next;  
}\*pPCB;  
  
*////简单轮转法  
//typedef struct R\_PCB{  
// int process\_sign; //进程标识符  
// int process\_rotation\_num; //轮转时间片  
// int cpu\_time; //进程所占CPU时间  
// int cpu\_need\_time; //进程所需时间片数  
// int process\_status; //进程当前状态  
// struct PCB \* next;  
//}\*pRPCB;  
  
//将链表中的结构体按优先数从大到小的规律排序***void** insert\_for\_sort\_operation(**struct** P\_PCB\*\* ppNode, **struct** P\_PCB\* pNode);  
  
**void** sort\_for\_link\_node(**struct** P\_PCB\*\* ppNode);  
  
  
*//优先数法***void** priority(**struct** P\_PCB \*\*run,**struct** P\_PCB \*\*ready,**struct** P\_PCB \*\*tail,**struct** P\_PCB \*\*finish);  
  
  
#endif *//PCBPROJECT\_PCB\_H*

**Pcb.c**

#include **"pcb.h"***//将链表中的结构体按优先数从大到小的规律排序***void** insert\_for\_sort\_operation(**struct** P\_PCB \*\*ppNode, **struct** P\_PCB \*pNode)  
{  
 **struct** P\_PCB\* prev;  
 **struct** P\_PCB\* cur;  
  
 */\* 在第一个数据之前插入pNode \*/* **if**(pNode->process\_priority\_num > (\*ppNode)->process\_priority\_num){  
 pNode->next = \*ppNode;  
 \*ppNode = pNode;  
 **return**;  
 }  
  
 cur = \*ppNode;  
 **while**(cur){  
 **if**(pNode->process\_priority\_num > cur->process\_priority\_num)  
 **break**;  
  
 prev = cur;  
 cur = cur->next;  
 }  
  
 pNode->next = prev->next;  
 prev->next = pNode;  
 **return**;  
}  
  
  
**void** sort\_for\_link\_node(**struct** P\_PCB \*\*ppNode)  
{  
 **struct** P\_PCB\* prev;  
 **struct** P\_PCB\* curr;  
  
 **if**(**NULL** == ppNode || **NULL** == \*ppNode)  
 **return**;  
  
 curr = (\*ppNode) ->next;  
 (\*ppNode) ->next = **NULL**;  
  
 **while**(curr){  
 prev = curr;  
 curr = curr->next;  
 insert\_for\_sort\_operation(ppNode, prev);  
 }  
  
 **return**;  
}  
  
  
*//优先数法  
/\*\*  
 \*  
 \* @param p 链表指针，指向链首  
 \* @param run 表示当前运行进程指针  
 \* @param ready 就绪队列头指针  
 \* @param tail 就绪队列尾指针  
 \* @param finish 完成队列头指针  
 \*/***void** priority(**struct** P\_PCB \*\*run,**struct** P\_PCB \*\*ready,**struct** P\_PCB \*\*tail,**struct** P\_PCB \*\*finish){  
 **char** sign = **'y'**;  
 **if** (\*run == **NULL**){  
 *//如果链表为空，结束程序* printf(**"PCB进程控制块为空 !\n"**);  
 exit(0);  
 }  
 **while** (sign == **'y'** || sign==**'Y'**){  
 \*ready = (\*run)->next;  
 **struct** P\_PCB \*pReady = \*ready,\*pFinish = \*finish;  
 *// struct P\_PCB \*tailHead = \*tail;* **struct** P\_PCB\* head = \*run,\*ptr = \*run,\*str = **NULL**,\*runHead = \*run;  
 head->cpu\_need\_time --;  
 head->process\_priority\_num = head->process\_priority\_num - 3;  
  
 *// str = head->next;  
 //将链首与下一个结构体的优先数进行对比，如果少于下一个优先数,将下一个置为链首  
 //将原链首按优先数从大到小插入链表中* **int** count = head->cpu\_need\_time ;  
 **if** ((head->process\_priority\_num < pReady->process\_priority\_num) && count > 0 ){  
 str = \*ready;  
 *//\*ready = (\*run)->next;* **if**((head->next)->process\_status == **PROCESS\_FINISH**){  
 *//当只有一个进程未完成时* } **else**{  
 \*run = (\*run)->next;  
 printf(**"%d\n"**,head->process\_priority\_num);  
 (\*run)->process\_status = **PROCESS\_RUN**;  
 ptr = \*run;  
 **while** (ptr != **NULL** ){  
 printf(**"ptr = %d\n"**,ptr->process\_priority\_num);  
 **if** ((head->process\_priority\_num > ptr->process\_priority\_num) && ptr->process\_status != **PROCESS\_FINISH**){  
 *//从链首开始遍历就绪链表段，将进程插入合适位置* head->process\_status = **PROCESS\_READY**;  
 str->next = head;  
 head ->next = ptr;  
 **break**;  
 }**else if**((ptr->process\_priority\_num == (\*tail)->process\_priority\_num) && (head->process\_priority\_num < ptr->process\_priority\_num)){  
 *//当进程未完成并且优先数最低的时候，将进程置于就绪段的段位* head->process\_status = **PROCESS\_READY**;  
 head->next = ptr->next;  
 ptr->next = head;  
 \*tail = head;  
 \*finish = (\*tail)->next;  
 **break**;  
 } **else**{  
 str = ptr; *//让str指向当前进程的前一个进程* ptr = ptr->next;  
 }  
 }  
 }  
  
 }**else if**(count == 0){  
 *//当进程完成后* **if** ((\*ready)->process\_status == **PROCESS\_FINISH**){  
 printf(**"all process finished\n"**);  
 **return**;  
 }  
 \*run = \*ready;  
 (\*run)->process\_status = **PROCESS\_RUN**;  
 \*ready = (\*run)->next;  
 *// ptr = \*run;* pFinish = \*run;  
 **if** (\*finish == **NULL**){  
 *//完成指针指向NULL* \*finish = ptr;  
 pFinish = \*finish;  
 }  
 **while** (pFinish->next != **NULL**){  
 *// printf(" %d \n",pFinish->process\_priority\_num);* pFinish = pFinish->next;  
 }  
 head->next = **NULL**;  
 head->process\_status = **PROCESS\_FINISH**;  
 pFinish->next = head; *//将完成的进程置于finish队列的最后* }  
 runHead = \*run;  
 **while**(runHead != **NULL**){  
 *//printf("优先数:%d 进程所需时间片数:%d 进程所占时间片数：%d",runHead->process\_priority\_num,runHead->cpu\_need\_time,runHead->cpu\_time);* printf(**"%d %d %d"**,runHead->process\_priority\_num,runHead->cpu\_need\_time,runHead->process\_status);  
 printf(**"\n"**);  
 runHead = runHead->next;  
 }  
 printf(**"is continue(Y/N)?"**);  
 getchar();  
 scanf(**"%c"**,&sign);  
 }  
  
}

**Main.c**

#include **"pcb.h"** *//对进程控制块进行初始化***void** init(**struct** P\_PCB \*pre ){  
 **char** an ;  
 **int** count = 0;  
 *//RamPre tou = pre;* printf(**"输入进程优先数："**);  
 scanf(**"%d"**,&(pre->process\_priority\_num));  
*// printf("输入进程所占CPU时间片数：");  
// scanf("%d",&(pre->cpu\_time));* printf(**"输入进程所需时间片数:"**);  
 scanf(**"%d"**,&(pre->cpu\_need\_time));  
 pre->process\_status = **PROCESS\_READY**;  
 pre->process\_sign = count;  
 pre->next = **NULL**;  
 **do**{  
 printf(**"是否继续创建(y/n)?"**);  
 getchar();  
 scanf(**"%c"**,&an);  
 **if** (an == **'y'** || an == **'Y'**){  
 **struct** P\_PCB \*p = (**struct** P\_PCB\*)malloc(**sizeof**(**struct** P\_PCB));  
 count++;  
 printf(**"输入进程优先数："**);  
 scanf(**"%d"**,&(p->process\_priority\_num));  
*// printf("输入进程所占CPU时间片数：");  
// scanf("%d",&(p->cpu\_time));* printf(**"输入进程所需时间片数:"**);  
 scanf(**"%d"**,&(p->cpu\_need\_time));  
 p->process\_status = **PROCESS\_READY**;  
 p->process\_sign = count;  
 p->next = **NULL**;  
 pre->next = p;  
 pre = pre->next;  
 }  
  
 }**while** (an == **'y'** ||an == **'Y'**);  
}  
**int** main() {  
  
 **struct** P\_PCB\* p\_pcb = (**struct** P\_PCB \*)malloc(**sizeof**(**struct** P\_PCB));  
 **struct** P\_PCB\* run=**NULL**,\*ready = **NULL**,\*tail = **NULL**,\*finish = **NULL**;  
 **char** an;  
 printf(**"是否创建进程(Y/N)?"**);  
 scanf(**"%c"**,&an);  
 **if** (an == **'Y'**){  
 init (p\_pcb);  
 }  
 **struct** P\_PCB \*s = p\_pcb;  
 **while**(s != **NULL**){  
 printf(**"%d-"**,s->process\_sign);  
 s= s->next;  
 };  
 printf(**"\n"**);  
 sort\_for\_link\_node(&p\_pcb);  
 s = p\_pcb;  
 **while**(s->next != **NULL**){  
 printf(**"%d-"**,s->process\_priority\_num);  
 s= s->next;  
 };  
 printf(**"\n"**);  
 ready = p\_pcb->next;  
 tail = s;  
 finish = **NULL**;  
 p\_pcb->process\_status = **PROCESS\_RUN**;  
 priority(&p\_pcb,&ready,&tail,&finish);  
 s = p\_pcb;  
 **while**(s != **NULL**){  
 printf(**"%d-"**,s->process\_sign);  
 s= s->next;  
 };  
 **return** 0;  
}

**实验二：死锁**

1. **实验目的**

学生应独立的采用高级语言编写一个动态分配系统资源的程序，模拟死锁现象，观察死锁发生的条件，并采用适当的算法，有效地防止死锁的发生。学生应通过本次实验，更直观的了解死锁发生的原因，初步掌握防止死锁、解除死锁的简单方法，加深理解教材中有关死锁的内容。

1. **实验内容**
2. **题目**

本次实验采用银行算法防止死锁的发生。设有3个并发进程共享10个系统资源。在3个进程申请系统资源之和不超过10时，当然不可能发生死锁，因为各个进程资源都能满足。在有一个进程申请的系统资源数超过10时，必然会发生死锁。应该排队这两种情况。程序采用人工输入各进程的申请资源序列。如果随机给各进程分配资源，就可能发生死锁，这也就是不采用防止死锁算法的情况。假如，按照一定的规则，为各进程分配资源，就可以防止死锁的发生。示例采用银行算法。这是一种犹如“瞎子爬山”的方法，即探索一步，前进一步，行不通，再往其它方向试探，直至爬上山顶。这种方法是比较保守的，所花的代价也不小。

1. **算法说明**

*银行算法*，顾名思义是来源于银行的借贷业务，一定数量的本金要应付各种客户的借贷周转，为了防止银行因资金无法周转而倒闭，对每一笔贷款，必须考察其最后是否能归还。研究死锁现象时就碰到类似的问题，有限资源为多个进程共享，分配不好就会发生每个进程都无法继续下去的死锁僵局。

银行算法的原理是先假定某一次分配成立，然后检查由于这次分配是否会引起死锁，即剩下的资源是不是能满足任一进程完成的需要。如这次分配是安全的（不会引起死锁），就实施本次分配，再作另一种分配试探，一直探索到各进程均满足各自的资源请求，防止了死锁的发生。

**3.实验步骤**

1. **任务分析**

本实验使用C语言模拟动态分配系统资源的程序，观察斯所发生的条件，并设计算法，防止死锁发生。

* 1. **输入的形式和输入值的范围**

输入：第一行：资源种类数 进程数

第二行：各类资源数目

第三行：各个进程对各类进程的最大需求

第四行：系统已经分配给各进程的资源数

第五行：本次请求资源的进程号

第六行：所申请的各类资源数

范围为大于0的整数

* 1. **输出的形式**

第一行：输入结束后各进程还需各类资源数

第二行：能够分配的话输出分配的过程不能分配给出提示

* 1. **程序所能达到的功能**

实现银行家算法

1. **程序设计**

输出出错信息

找出一个未完成的

进程作为现行进程

现行进程

已完成所有

请求

收回该进程占用的资源，标志该进程结束，如果全部进程结束，置ADVANCE为‘flase’，否则’true’

找出一个未完成的进程作为现行进程

申请量

超过最大需

求量

死锁处理程序

进程已全部结束？

开始

输入选择标志

OPTION

输入各进程动态申请

资源数组AP

输入各进程最资源需求

量数组VPMAXCLAIM

最大资源

需求量超过系统

资源数

输出出错信息

关闭输出文件

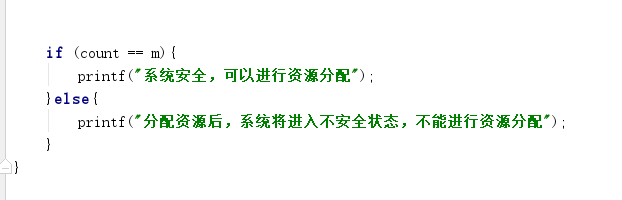
结束

1. **调试分析**

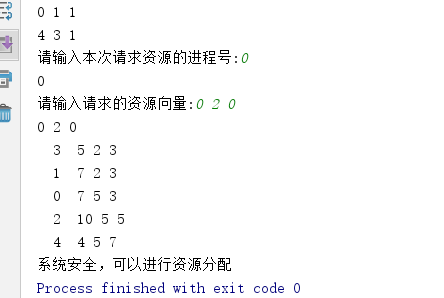
程序关键部分在银行家算法中的安全性检查的实现，这里我通过三重for循环实现



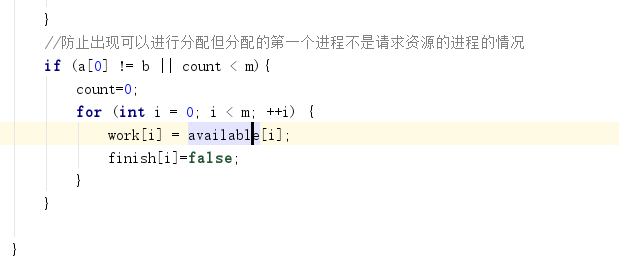
在这里面设置了一个count计数器，记录每次检测所能分配的进程的数量，如果循环结束后，count的值等于进程数量的话，就代表有一种分配方法。



这样虽然能避免通常情况下不能执行完所有进程的情况，但是，还不够完整：可能会存在这样一种情况：



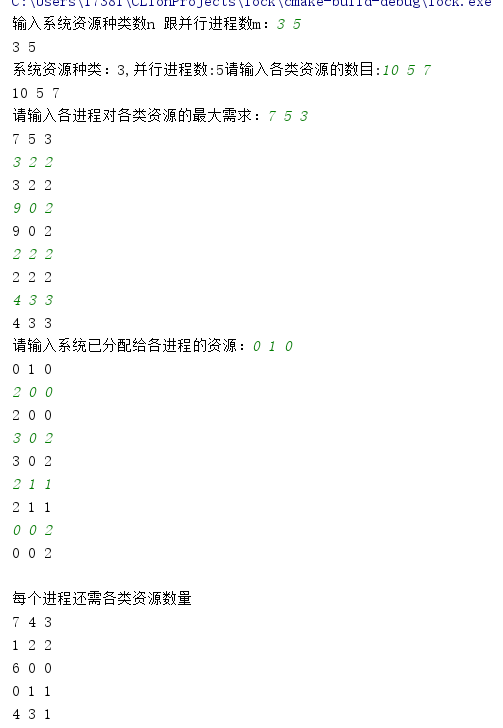
即分配方案第一个分配进程不是请求资源的进程，因此要进行进一步完善，于是，我进行了以下处理：



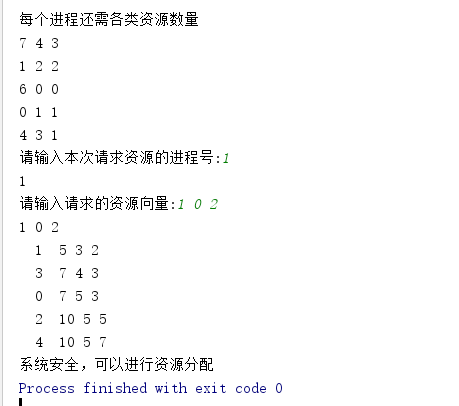
如图所示，添加一个数组记录每次循环分配的进程号，在每次循环完之后，对第一个进程号进行检查，如果不是请求资源的进程，则重新遍历。

1. **测试结果**

**输入**

****

**输出及检测**

****

1. **使用说明**

程序运行后有详细提示，按照程序执行步骤操作即可。

1. **实验总结**

在实际现代操作系统中，都是多个进程并发执行，不再像dos系统单道执行，这样提高了系统资源的利用率，但同时也带来了不少新的问题：多个进程对同样的资源都有需求，要是系统现有资源不能满足进程需求，进程就会停下来等待，要是多个进程都进入等待状态，这时每个进程都在等别的进程运行完释放资源，这就进入了死锁。这种现象会严重影响进程的执行，所以我们必须想办法避免出现这种情况。有以下几种方法：

* + 1. 为每一个进程设置一个等待临界时间，重进程开始等待开始计时，当等待时间到达该临界时间后进程销毁，释放资源。
    2. 设置算法，避免出现死锁。算法设计过程中需要考虑到系统对当前申请资源的进程请求响应后，会不会影响到之后其他进程的运行，确保不会影响后，才能对其进行响应。

1. **附录**

**Main.c**

#include **<stdio.h>**#include **<stdbool.h>**#define **AVAILABLE** 20  
#define **MAX\_X** 20  
#define **MAX\_Y** 20  
#define **ALLOCATION\_X** 20  
#define **ALLOCATION\_Y** 20  
#define **NEED\_X** 20  
#define **NEED\_Y** 20  
  
*//系统资源量***int** total[**AVAILABLE**];  
*//可用资源***int** available[AVAILABLE];  
*//最大需求***int** max[MAX\_X][MAX\_Y];  
*//分配数据,系统已分配给定义第i个进程多少个资源***int** allocation[ALLOCATION\_X][ALLOCATION\_Y];  
*//表示每一个进程尚需的资源***int** need[NEED\_X][NEED\_Y];  
  
  
*//初始化数据***void** init(**int** n,**int** m){  
 *//int n,m;* printf(**"请输入各类资源的数目:"**);  
 **for** (**int** i = 0; i < n; ++i) {  
 scanf(**"%d"**,&total[i]);  
 }  
 printf(**"请输入各进程对各类资源的最大需求："**);  
 **for** (**int** j = 0; j < m; ++j) {  
 **for** (**int** i = 0; i < n; ++i) {  
 scanf(**"%d"**,&max[j][i]);  
 }  
  
 }  
 printf(**"请输入系统已分配给各进程的资源："**);  
 **for** (**int** k = 0; k <m ; ++k) {  
 **for** (**int** i = 0; i <n ; ++i) {  
 scanf(**"%d"**,&allocation[k][i]);  
 }  
  
 }  
  
 *//计算可用资源available* **for** (**int** i1 = 0; i1 <m ; ++i1) {  
 **for** (**int** i = 0; i <n ; ++i) {  
 total[i] = total[i] - allocation[i1][i];  
  
 }  
  
 }  
 **for** (**int** j1 = 0; j1 <n ; ++j1) {  
 available[j1] = total[j1];  
 *// printf("%d ",available[j1]);* }  
 printf(**"\n"**);  
*// printf("请输入每个进程尚需的各类资源：");  
// for (int l = 0; l <m ; ++l) {  
// for (int i = 0; i <n ; ++i) {  
// scanf("%d",&need[l][i]);  
//  
// }  
//  
// }* printf(**"每个进程还需各类资源数量\n"**);  
 **for** (**int** i = 0; i <m ; ++i) {  
 **for** (**int** j = 0; j <n ; ++j) {  
 need[i][j] = max[i][j]-allocation[i][j];  
 printf(**"%d "**,need[i][j]);  
 }  
 printf(**"\n"**);  
 }  
  
}  
*//检查request是否符合要求*bool exam(**int** request[MAX\_X],**int** object[AVAILABLE],**int** n){  
 **int** i = 0;  
 **while** (request[i]<=object[i] && i<n){  
 i++;  
 }  
 **if** (i<n){  
 **return** false;  
 }  
 **return** true;  
  
}  
  
*//计算need***void** calculate(**int** need[NEED\_X][NEED\_Y],**int** max[MAX\_X][MAX\_Y],**int** allocation[ALLOCATION\_X][ALLOCATION\_Y],**int** m,**int** n){  
 **for** (**int** i = 0; i <m ; ++i) {  
 **for** (**int** j = 0; j <n ; ++j) {  
 need[i][j] = max[i][j]-allocation[i][j];  
 printf(**"%d "**,need[i][j]);  
 }  
 printf(**"\n"**);  
 }  
  
  
}  
*//安全性算法检查***void** checkSecurity(**int** need[MAX\_X][AVAILABLE],**int** allcation[MAX\_X][MAX\_Y],**int** n,**int** m,**int** b){  
 **int** count=0,k,t,o; *//记录已检查的进程个数//初始化finish跟work* **int** work[n],a[n]; *//表示系统可分配给进程继续运行所需的各类进程数* bool finish[m],exam[m]; *//表示系统是否有足够的资源分配给进程* **for** (**int** i = 0; i <n ; ++i) {  
 work[i] = available[i];  
 finish[i] = false;  
 exam[i] = false;  
 a[i] = n+1;  
 }  
 *//三重循环进行安全性检查* **for** (**int** l = 0; l < m; ++l) {  
 **for** (**int** j = 0; j <m; ++j) {  
 k=0;  
 *//exam[]为了在遍历的时候控制不重复去遍历系统不足以满足的进程* **if** (exam[j] == false){  
 **for** (**int** i = 0; i <n && finish[j] == false; ++i) {  
 *//条件语句中的finish[] == false为了是程序不再遍历已经执行完的进程,节约时间* k=i;  
 **if** (work[i] < need[j][i]){  
 exam[j]=true; *//对于某个进程，如果需求的某类资源不能满足需求，记录下来* **break**;  
 }  
 }  
 **if** (k==n-1){  
 t = count;  
 a[count] = j;  
 count++;  
 printf(**" %d "**,j);  
 finish[j]=true; *//能够满足该进程的资源需求，则进行分配，并将finish[] 置为true* **for** (**int** i = 0; i <n ; ++i) {  
 work[i] = work[i] + allcation[j][i];  
 exam[i] = false; *//这里的exam[]为了保证下次可以遍历该进程* printf(**"%d "**,work[i]);  
 }  
 printf(**"\n"**);  
 **break**;  
 }  
 }  
  
 }  
 *//防止出现可以进行分配但分配的第一个进程不是请求资源的进程的情况* **if** (a[0] != b || count < m){  
 count=0;  
 **for** (**int** i = 0; i < m; ++i) {  
 work[i] = available[i];  
 finish[i]=false;  
 }  
 }  
 }  
  
 **if** (count == m){  
 printf(**"系统安全，可以进行资源分配"**);  
 }**else**{  
 printf(**"分配资源后，系统将进入不安全状态，不能进行资源分配"**);  
 }  
}  
**int** main() {  
 **int** n,m,count=0,b;  
 bool exam1=false,exam2=false;  
 **int** request[MAX\_X]; *//表示当前请求向量* printf(**"输入系统资源种类数n 跟并行进程数m："**);  
 scanf(**"%d %d"**,&n,&m);  
 printf(**"系统资源种类：%d,并行进程数:%d"**,n,m);  
 init(n,m);  
 **do**{  
 printf(**"请输入本次请求资源的进程号:"**);  
 scanf(**"%d"**,&b);  
 printf(**"请输入请求的资源向量:"**);  
 **for** (**int** i = 0; i <n ; ++i) {  
 scanf(**"%d"**,&request[i]);  
  
 }  
 *//检查是否request合理* exam1 = exam(request,need[b],n);  
 exam2 = exam(request,available,n);  
 **if** (exam1 == true && exam2 ==true){  
 **for** (**int** i = 0; i < n; ++i) {  
 available[i] = available[i] - request[i];  
 allocation[b][i] = allocation[b][i]+request[i];  
 need[b][i] = need[b][i] - request[i];  
 }  
  
 }checkSecurity(need,allocation,n,m,b);  
  
 }**while**(count==m);  
 *//可用资源* **return** 0;  
}

**实验三：存储管理**

1. **实验目的**

提高内存管理的效率始终是操作系统研究的重要课题之一，虚拟存储技术是用来提高存储容量的一种重要方法，所以，本项实验的目的是让学生独立地设计几个常用的存储分配算法，并用高级语言编写程序对各种算法进行分析比较，评测其性能的优劣，从而加深对这些算法的了解。

1. **实验内容**
2. **分区分配算法**

本实验要求采用首次适应算法和最佳适应算法两种分区分配的内存管理算法。

(1) 建立分区描述器：分区描述器可根据自己编写程序的需要来建立，描述器本身所包含的内容以描述清楚内存分区情况为准。

(2) 建立自由主存队列：针对两种不同的放置策略（首次适应和最佳适应）来建立相应的队列结构。

(3) 用C语言编写实现首次适应算法和最佳适应算法的程序。

(4) 用C语言编写回收算法。

1. **请求式分页存储管理算法**

本实验要求采用请求式分页存储算法，淘汰算法采用先进先出算法FIFO和最近最少使用页面淘汰算法（LRU）。

设逻辑空间大小为128K，页面尺寸分别为2、4、6、8、10、12、14、16K，内存容量为8至64页。

1. 先进先出算法FIFO：该算法的实质是选择作业中在主存驻留时间最长的一页淘汰，这种算法容易实现，例如分配一个作业的存储块数为m，则只需建立一张m个元素的队列表Q(0)、Q(1)、…、Q(m-1)和一个替换指针。这个队列是按页调入主存的一页。如图4-1所示，某时刻调入主存四个块，（即m=4），它们按页进入主存的先后顺序为4、5、1、2，当需要置换时，总是淘汰替换指针所指向的那一页，见图3.1.

（指向最老的一页）

页 号

2

4

5

1

替换指针

图3.1 队列表的实现示意图

新调进的页面装入主存后，修改相应的队列元素，然后将替换指针往前移动，使其指向当前最老的一页。

(2) 最近最少使用页面淘汰算法（LRU）：这是一种经常使用的方法，有多种不同的实施方案。这里采用不断调整页表链的方法，即总是淘汰页表链链首的页面，而把新访问的页面插入链尾。如果当前调用页面已在页表内，则把它再次调整到链尾。这样就能保证最近使用的页面总处于靠近链尾部分，而不常使用的页面被移到链首，逐个被淘汰。

**3.实验步骤**

**（1）任务分析**

**输入的形式和输入值的范围**

输入内存大小（int）

**输出的形式**

1. 输入后内存块链表
2. 内存进行分配后的内存块链表
   1. **实验设计**

**1.分区分配法**

**1）首次适应算法**

创建好内存块链表并设置玩申请内存大小后，对内存链进行遍历，挨个检查内存块的大小，当找到第一个适合申请内存要求的内存块时停止遍历，将内存进行划分，这时需要注意该内存结构体的状态为未使用，这样才能进行分配。

**2）最佳适应算法**

基本上与首次适应算法一致，区别是必须遍历完内存链，找出最合适所要申请的内存大小的内存块。

设置一个临界值，当将适合分配的内存大小与申请的内存大小进行比较，差值大于这个临界值时对该内存进行划分，若小于临界值则将内存全部给所申请内存。

**2.请求式分页存储管理算法**

**1）先进先出算法**

该算法的实质是选择作业中在主存驻留时间最长的一页淘汰，这种算法容易实现

**2）最近最少使用页面淘汰算法**

这是一种经常使用的方法，有多种不同的实施方案。这里采用不断调整页表链的方法，即总是淘汰页表链链首的页面，而把新访问的页面插入链尾。如果当前调用页面已在页表内，则把它再次调整到链尾。这样就能保证最近使用的页面总处于靠近链尾部分，而不常使用的页面被移到链首，逐个被淘汰。

**程序流程图：**

开 始

产生给定长度符合某种假定的指令地址流

Alg=FIFO/LRU?

为每一指令地址形

成对应的访问页号

置初值：size=1

assigned=4

输入淘汰算法Alg

用FIFO淘汰算法计算命中率

最近最少使用页面淘汰算法计算命中率

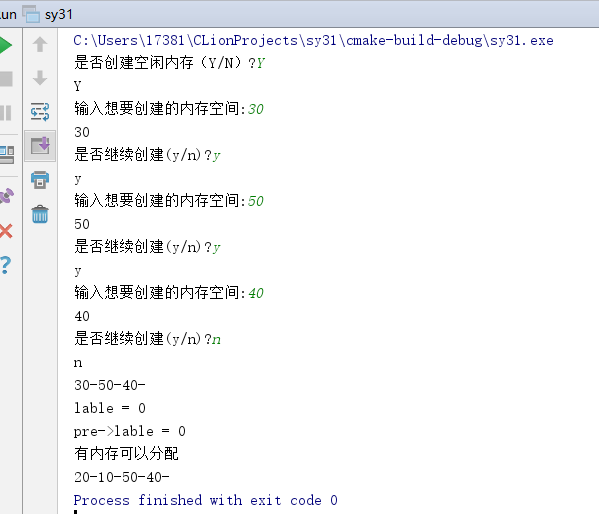
输出完整结果

结 束

图3.2 请求式分页存储算法流程图

* 1. **测试结果**

**分区分配算法中的最佳适应算法**

****

**分区分配算法中的首次适应算法**

****

* 1. **使用说明**

程序运行后有详细提示，按照程序执行步骤操作即可。

**4.实验总结**

内存管理是操作系统中重要的一部分，对内存进行有效的管理有利于更好的实现多进程并发执行。分区分配算法是在分页式管理机制出现之前的一种内存分配方式。他包括多种算法，有名的有首次适应算法和最佳适应算法。

首次适应算法顾名思义，就是对空闲内存进行遍历，当找到第一块满足需求的内存时就对内存进行划分，分配给所需进程。最佳适应算法同样，不过是找出最合适的一块内存进行划分。但是以上两种算法都会产生内零头跟外零头。造成内存的浪费。针对这种情况，提出了分页式管理机制，就是无论所需内存大小，都以页为单位进行内存分配。这样消除了外零头。同时如果按照适合的大小进行分页，也能大大减少内零头的产生。

**5. 附录**

**Main.c**

#include **<stdio.h>**#include **<malloc.h>**#include **<string.h>**#define **SIZE** 10  
  
*//结构体定义每块内存块的可使用容量***typedef struct** ram{  
 *// struct ram \* precious;* **int** lable;  
 **int** size;  
 **int** status;  
 **struct** ram \*next;  
}Ram ,\*RamPre;  
  
*//首次适应算法***void** firstFit(**int** demand,RamPre \*preHead){  
 *//对链表进行遍历，找出第一个适合的内存块* **struct** ram\* pre = \*preHead,\*p = \*preHead,\*pri = **NULL**;  
 **int** result = pre->size - demand;  
 *// RamPre pri = pre;* **while** (pre != **NULL**){  
 **if**( result >= **SIZE** ){ *//当剩余内存大于size的时候，只划出需要的部分* p = (**struct** ram\*)malloc(**sizeof**(**struct** ram)); *//将指向新建节点* **if** (p == **NULL**){  
 printf(**"No enough memory !\n"**);  
 **break**;  
 }  
 p->size = demand;  
 p->status = 1;  
  
 **if** (\*preHead == **NULL**){  
 *//若原链表为空* \*preHead = p;  
 }**else**{  
 pri = pre; *//temp中保存当前节点的指针* }  
 **if** ( pre == \*preHead){  
 *//若在头节点插入新节点* p->next = \*preHead;  
 \*preHead = p;  
 }**else**{  
 *// pre = pri;* pri->next = p;  
 p->next = pre;  
 }  
 pre->size -= demand;  
 printf(**"yes\n"**);  
 printf(**"%d\n"**,p->size);  
 printf(**"%d\n"**,p->next->size);  
 **break**;  
 }**else if**(result < **SIZE** && result >= 0 ){ *//当剩余内存小于size且大于等于0的时候，将内存全部分配给* pre->status = 1;  
 printf(**"有内存可以分配\n"**);  
 **break**;  
 } **else**{  
 *// pri->next = pre;* pri = pre; *//保存当前节点的上一个节点* pre = pre->next;  
 result = pre->size - demand;  
 **if** (pre == **NULL**){  
 printf(**"没有足够大的内存可以分配"**);  
 }  
 }  
 }  
  
}  
  
*//最佳适应算法***void** bestFit(**int** demand,RamPre \*preHead){  
 *//RamPre str = pre;* **struct** ram \*str = \*preHead,\*p = \*preHead ,\*pre = \*preHead,\*pri = **NULL**;  
 **int** result = pre->size - demand;  
 **int** n;  
 **int** lable = -1;  
 **while** (str != **NULL**){  
 n = str->size - demand;  
 *// n = str->size;* **if**(n > 0 && n < result){  
 result = n;  
 lable = str->lable;  
 }  
 str = str->next;  
 }  
  
 printf(**"lable = %d\n"**,lable);  
 *//再次便利找到最适合的内存块，进行分配* **while** (pre != **NULL**){  
 **if** (lable == pre->lable && **SIZE** <= result){  
 p = (**struct** ram\*)malloc(**sizeof**(**struct** ram)); *//将指向新建节点* **if** (p == **NULL**){  
 printf(**"No enough memory !\n"**);  
 **break**;  
 }  
 p->size = demand;  
 p->status = 1;  
  
 **if** (\*preHead == **NULL**){  
 *//若原链表为空* \*preHead = p;  
 }**else**{  
 pri = pre; *//temp中保存当前节点的指针* }  
 **if** ( pre == \*preHead){  
 *//若在头节点插入新节点* p->next = \*preHead;  
 \*preHead = p;  
 }**else**{  
 *// pre = pri;* pri->next = p;  
 p->next = pre;  
 }  
 pre->size -= demand;  
 printf(**"yes\n"**);  
 printf(**"%d\n"**,p->size);  
 printf(**"%d\n"**,p->next->size);  
 **break**;  
 } **else if**(lable == pre->lable && result >= 0){  
 printf(**"pre->lable = %d\n"**,pre->lable);  
 pre->status = 1;  
 printf(**"有内存可以分配\n"**);  
 **break**;  
 } **else**{  
 pri = pre; *//保存当前节点的上一个节点* pre = pre->next;  
 result = pre->size - demand;  
 **if** (pre == **NULL**){  
 printf(**"没有足够大的内存可以分配"**);  
 }  
 }  
  
 }  
  
  
  
}  
  
*//回收内存算法***void** recycleRam(RamPre idle){  
  
}  
  
*//初始化空闲内存***void** init(RamPre pre){  
 **char** an ;  
 **int** count = 0;  
 *//RamPre tou = pre;* printf(**"输入想要创建的内存空间:"**);  
 scanf(**"%d"**,&(pre->size));  
 pre->lable = count;  
 pre->status = 0;  
 pre->next = **NULL**;  
 **do**{  
 printf(**"是否继续创建(y/n)?"**);  
 getchar();  
 scanf(**"%c"**,&an);  
 **if** (an == **'y'** || an == **'Y'**){  
 **struct** ram\* p = (**struct** ram\*)malloc(**sizeof**(**struct** ram));  
 p->next = **NULL**;  
 printf(**"输入想要创建的内存空间:"**);  
 scanf(**"%d"**,&(p->size));  
 count++;  
 p->lable = count;  
 p->status = 0;  
 pre->next = p;  
 pre = pre->next;  
 }  
  
 }**while** (an == **'y'** ||an == **'Y'**);  
}  
  
**int** main() {  
 **struct** ram \*p = (**struct** ram\*)malloc(**sizeof**(**struct** ram));  
 **char** an,chr[10];  
 printf(**"是否创建空闲内存（Y/N）?"**);  
 scanf(**"%c"**,&an);  
 **if** (an == **'Y'**){  
 init (p);  
 }  
 **struct** ram \*s = p;  
 **while**(s != **NULL**){  
 printf(**"%d-"**,s->size);  
 s= s->next;  
  
 };  
 printf(**"\n"**);  
  
 *// printf("%d",p->size);* printf(**"请选择算法（bestFit/firstFit）:"**);  
 getchar();  
 scanf(**"%s"**,&chr);  
 **if** (strcmp(chr,**"bestFit"**) == 0){  
 bestFit(20,&p);  
 }**else if**(strcmp(chr,**"firstFit"**) == 0){  
 firstFit(20,&p);  
 }**else**{  
 printf(**"input error!\n"**);  
 }  
  
 **while**(p != **NULL**){  
 printf(**"%d(%d)-"**,p->size,p->status);  
 p= p->next;  
  
 };  
 **return** 0;  
}

**实验四：文件系统**

**1.实验目的**

文件系统是操作系统的一个重要组成部分，也是与用户关系极为密切的部分。学生应独立的用高级语言编写和调试一个简单的文件系统，模拟文件管理的工作过程，从而对各种文件操作命令的实质内容和执行过程有比较深入的了解，掌握它们的实施方法，加深对教材中有关内容的理解

**2.实验内容**

**要求：**

1. 设计一个n个用户的文件系统，每个用户最多可保存m个文件。
2. 限制用户在一次运行中只能打开一个文件。
3. 系统应能检查输入命令的正确性，出错时要能显示出错原因。
4. 对文件必须设置保护措施，如只能执行，允许读，允许写等。在每次打开文件时，根据本次打开的要求，再次设置保护级别，即可有二级保护。
5. 对文件的操作至少应有下面几条命令：

create 建立文件

delete 删除文件

open 打开文件

close 关闭文件

read 读文件

write 写文件

**题目：**

1. 本次实验设计一个共有10个用户的文件系统，每个用户最多可保存10个文件，一次运行过程中用户可同时打开5个文件。
2. 程序采用二级文件目录，即设置了主文件目录（MFD）和用户文件目录（UFD）。前者应包含文件主（即用户）及他们的目录区指针；后者应给出每个文件主占有的文件目录，即文件名、保护码、文件长度以及它们存放的位置等。另外为打开文件设置了运行文件目录（AFD），在文件打开时应填入打开文件号，本次打开保护码和读写指针等。
3. 为了便于实现，对文件的读写作了简化，在执行读写命令时，只修改读写指针，并不进行实际文件的读写操作。

**算法：**

1. 因系统小，文件目录的检索使用了简单的线性搜索，而没有采用Hash等有效算法。
2. 文件保护简单使用了三位保护码，对应于允许读，允许写和允许执行，如下所示：

1 1 1

允许读 允许写 允许执行

如对应位为0，则不允许。

1. 程序中使用的主要数据结构如下：
   1. 主文件目录（MFD）和用户文件目录（UFD）

MFD UFD

用户名1

文件目录指针1

用户名2

文件目录指针2

用户名3

文件目录指针3

┋

┋

文件名

┋

文件长度

保护码

文件名

┋

文件长度

保护码

图4.1主文件目录（MFD）和用户文件目录（UFD）

② 打开文件目录（AFD）

打开文件号

打开保护码

读写指针

图4.2 打开文件目录（AFD）

**3.实验步骤**

**（1）任务分析**

本实验使用C语言模拟文件系统的运行，实现几种简单的命令如：create，delete，close，read等。

* 1. **输入的形式和输入值的范围**

指令名称 + 要操作的文件名，如创建文件test: create test

* 1. **输出的形式**

根据指令要求显示，如ls指令，执行显示当前用户目录下所有文件的文件名

* 1. **程序所能达到的功能**

程序实现几种简单的文件操作指令：

create 建立文件

delete 删除文件

open 打开文件

close 关闭文件

read 读文件

write 写文件

ls 显示当前目录下的文件

**（2）概要设计**

#define **USER\_NUM** 10  
#define **USER\_FILE\_NUM** 10  
#define **USER\_RUN\_FILE\_NUM** 5  
#define **PROTECT\_READ** 4  
#define **PROTECT\_WRITE** 2  
#define **PROTECT\_RUN** 1  
*//用户文件目录***struct** UFD{  
 **char** file\_name[50]; *//文件名* **int** protect\_num; *//保护码  
 // int file\_length; //文件长度* **unsigned int** file\_address; *//文件存放地址* **struct** UFD \*next; *//下一条文件信息*}FILE\_UFD,\*pFILE\_UFD;  
  
  
*//主文件目录***struct** MFD{  
 **char** user\_name[50]; *//用户名* **struct** FILE\_UFD \*ptr; *//文件目录指针*}FILE\_MFD,\*pFILE\_MFD;  
  
*//打开文件目录***struct** AFD{  
 **int** file\_num; *//打开文件号* **char** file\_name[50]; *//文件名* **int** file\_protect\_num; *//打开文件保护码* **unsigned int** read\_write\_ptr; *//读写指针,这里为了简化工作量,直接定义成整形,实际项目中不能这样* **struct** AFD \*next;  
  
}FILE\_AFD,\*pFILE\_AFD;  
  
  
*//创建文件***void** createFile(**struct** UFD \*\*p,**char** fileName[30]);  
  
*//删除文件***void** deleteFile(**struct** UFD \*p,**struct** AFD \*\* str,**char** fileName[30]);  
  
*//打开文件***void** openFile(**struct** UFD \*p,**struct** AFD \*\*str,**char** fileName[30]);  
  
*//关闭文件***void** closeFile(**struct** AFD \*str,**char** fileName[30]);  
  
*//读文件内容***void** readFile(**struct** UFD \*P,**struct** AFD \*\*strHead,**char** fileName[30]);  
  
*//写文件内容***void** writeFile(**struct** UFD \*p,**struct** AFD \*\*str,**char** fileName[30]);  
  
*//显示当前用户的文件目录***void** showFile(**struct** UFD \*p);

**（3）详细设计**

1. 创建三种结构体：用户文件目录UFD，主文件目录MFD,打开文件目录AFD

2. 程序流程图：

输入用户名

开始

在MFD中找到该用户？

报告无此用户文件

“try again”

否

是

结束

显示该用户目录表UFD中所有文件

初始化运行

文件表AFD

保存当前文件目录

Say good bye

打印当前文件目录

输入操作命令

是什么命令？

建立文件处理程序

删除文件处理程序

打开文件处理程序

建立文件处理程序

关闭文件处理程序

建立文件处理程序

3.程序需要实现的各个功能模块

a. 创建文件 b.删除文件 c.打开文件 d.关闭文件 e.读取文件 f.往文件中写 g.显示当前目录下文件

上述几种操作命令通过用户输入进行选择执行，使用循环控制进行多次访问，操作。

4.因为涉及到文件的读写，所以选择链表这类数据结构来存储文件信息。好处是灵活，操作方便，不需要对每个节点进行检查。

5.各函数之间的关系：

各函数之间基本上互不影响但是必须有一定的先后关系，比如要删除一个文件，前提是用户目录下有这个文件，要对一个文件进行读写操作，必须要打开这个文件才能进行读写等。

**（4）调试分析**

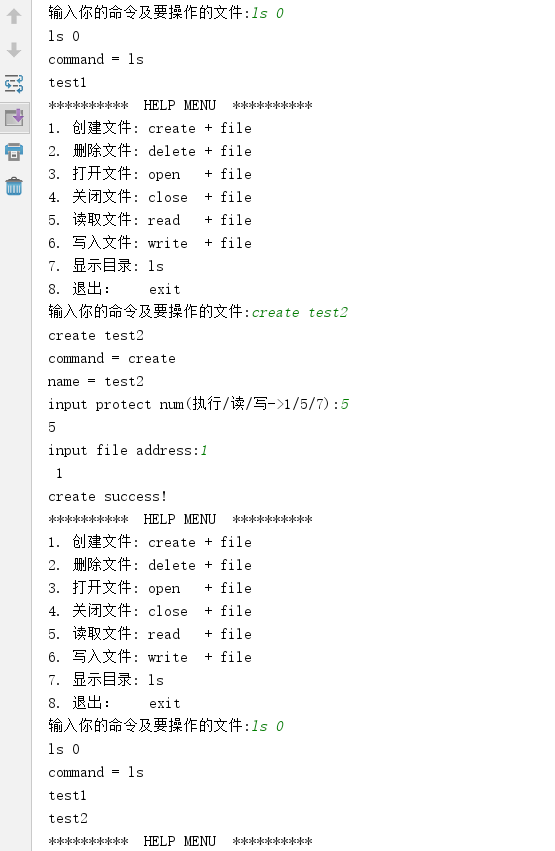
最开始使用的是一重指针，所以对链表进行修改后返回不了修改后的链表，于是想到了指针函数，指针函数可以实现对修改后链表的返回，但是后来想了想，指针函数只能对只有一个指针类型的形参起作用，要是形参中有多个指针并且都需要进行修改并返回的时候，指针函数就起不了作用。所以对程序进行了第二次该进，这次使用了二重指针去实现指针函数的功能。对了拥有多个指针形参的函数能很好的适用。

**（5）测试结果**

**创建文件test1**

****

**显示ls->create->ls**

****

**Ls->read test1->read test2(read指令执行前先查看打开目录中是否存在需要操作的文件，如果没有则先去执行open命令后再执行read指令)**

****

**（6）使用说明**

程序运行后有详细提示，按照程序执行步骤操作即可。

**4. 实验总结**

文件系统在操作系统中占有重要地位，文件系统主要负责磁盘中文件的管理和存储信息。个人理解中，文件系统作用在磁盘上：当一个进程创建之后，该进程会根据程序或文件存放的地址再通过文件系统提供的树状路径去访问该文件或程序。将代码和数据或者文件复制到内存中供进程使用，但是一但数据到达内存中之后，进程对该文件的一切操作均与文件系统没有任何关系（当然，此处指的是已经copy到内存中的部分），当进程对文件或者数据操作完成后，再保存到磁盘中，这时，就存在一个问题：修改后的文件或数据怎么与磁盘中的文件或数据进行合并。个人猜测是通过覆盖的方式进行更新。

实验过程中，我分别使用UNIX系统跟Windows系统分别进行了以下操作：

创建一个文件🡪打开文件写入一些数据,关闭🡪再次打开该文件，修改🡪在打开的状态下删除源文件🡪最后保存

会发生什么有意思的事?

Windows系统下不会让你在文件打开的状态下对其源文件进行删除，UNIX则会，在mac电脑上，执行到最后一步后，发现原本删除后到了回收箱里的文件内容发生了改变，也就是说，在保存后，文件更新内容保存到了原本已经被删除的文件里！这个现象我解释不了，可能需要看了内核源码后才能试着解释吧。

**5. 附录**

**Filesystem.h**

#ifndef **FILESYSTEM\_FILESYSTEM\_H**#define **FILESYSTEM\_FILESYSTEM\_H**#include **<stdio.h>**#include **<string.h>**#include **<stdlib.h>**#define **USER\_NUM** 10  
#define **USER\_FILE\_NUM** 10  
#define **USER\_RUN\_FILE\_NUM** 5  
#define **PROTECT\_READ** 4  
#define **PROTECT\_WRITE** 2  
#define **PROTECT\_RUN** 1  
  
  
*//用户文件目录***struct** UFD{  
 **char** file\_name[50]; *//文件名* **int** protect\_num; *//保护码  
 // int file\_length; //文件长度* **unsigned int** file\_address; *//文件存放地址* **struct** UFD \*next; *//下一条文件信息*}FILE\_UFD,\*pFILE\_UFD;  
  
  
*//主文件目录***struct** MFD{  
 **char** user\_name[50]; *//用户名* **struct** FILE\_UFD \*ptr; *//文件目录指针*}FILE\_MFD,\*pFILE\_MFD;  
  
*//打开文件目录***struct** AFD{  
 **int** file\_num; *//打开文件号* **char** file\_name[50]; *//文件名* **int** file\_protect\_num; *//打开文件保护码* **unsigned int** read\_write\_ptr; *//读写指针,这里为了简化工作量,直接定义成整形,实际项目中不能这样* **struct** AFD \*next;  
  
}FILE\_AFD,\*pFILE\_AFD;  
  
  
*//创建文件***void** createFile(**struct** UFD \*\*p,**char** fileName[30]);  
  
*//删除文件***void** deleteFile(**struct** UFD \*p,**struct** AFD \*\* str,**char** fileName[30]);  
  
*//打开文件***void** openFile(**struct** UFD \*p,**struct** AFD \*\*str,**char** fileName[30]);  
  
*//关闭文件***void** closeFile(**struct** AFD \*str,**char** fileName[30]);  
  
*//读文件内容***void** readFile(**struct** UFD \*P,**struct** AFD \*\*strHead,**char** fileName[30]);  
  
*//写文件内容***void** writeFile(**struct** UFD \*p,**struct** AFD \*\*str,**char** fileName[30]);  
  
*//显示当前用户的文件目录***void** showFile(**struct** UFD \*p);  
  
  
#endif *//FILESYSTEM\_FILESYSTEM\_H*

**Filesystem.c**

*//  
// Created by 17381 on 2017/12/23.  
//*#include **"fileSystem.h"**#include **<stdlib.h>**#include **<malloc.h>**#include **<stdbool.h>  
  
int** FILE\_RUNNING\_NUM = 0;  
  
*//创建文件***void** createFile(**struct** UFD \*\*head,**char** fileName[30]){  
 **struct** UFD \*pFile = **NULL**,\*p = \*head;  
 pFile = (**struct** UFD\*)malloc(**sizeof**(**struct** UFD));  
*// printf("input file name: ");  
// gets(pFile->file\_name);  
// getchar();  
 //如果申请内存失败,退出程序* **if** (pFile == **NULL**){  
 printf(**"No enough memory to allocation ! \n"**);  
 exit(0);  
 }  
 **if** (\*head == **NULL**){  
 *//若原链表为空* \*head = pFile; *//将新建节点设为头节点* }**else**{  
 **while** (p->next != **NULL**) {  
 p = p->next;  
 }  
 p->next = pFile; *//让末节点的指针域 指向新建节点* }  
 strcpy(pFile->file\_name,fileName);  
 printf(**"name = %s\n"**,pFile->file\_name);  
 printf(**"input protect num(执行/读/写->1/5/7):"**);  
 scanf(**"%d"**,&pFile->protect\_num);  
 printf(**"input file address: "**);  
 scanf(**"%d"**,&pFile->file\_address);  
 pFile->next = **NULL**;  
 printf(**"create success!\n"**);  
 *//printf("name2 = %s",p->file\_name);  
 //return head;*}  
  
*//删除文件  
/\*\*  
 \*删除一个文件前，首先遍历打开文件目录,查看是否是打开状态,  
 \* 打开状态下不让删除  
 \* @param p 用户文件目录  
 \* @param str 打开文件目录  
 \* @param fileName 需要操作的文件  
 \*/***void** deleteFile(**struct** UFD \*head,**struct** AFD \*\*strHead,**char** fileName[30]){  
 **int** count = 0; *//记录是否存在该文件，当用户目录下有该文件的时候，将count改变为1* **struct** UFD\* p = head;  
 **struct** AFD\* str = \*strHead;  
 **while** (str != **NULL**){  
 **if** (strcmp(str->file\_name,fileName) == 0){  
 printf(**"文件当前处于打开状态,不允许删除\n"**);  
 **return**;  
 }**else**{  
 *// struct AFD \*delete = NULL;  
 // delete = str;* str = str->next;  
 *// free(delete);* }  
 }  
  
 **while** (p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
 **struct** UFD \*o = **NULL**; *//删除当前文件节点* o = p;  
 p = p->next;  
 free(o);  
 count = 1;  
 **break**;  
 }**else**{  
 p = p->next;  
 }  
  
 }  
  
 **if** (count){  
 printf(**"delete successful !\n"**);  
 }**else**{  
 printf(**"no such file !\n"**);  
 }  
  
}  
  
*//打开文件***void** openFile(**struct** UFD \*p,**struct** AFD \*\*strHead,**char** fileName[30]){  
 **int** count = 0;  
 **struct** AFD \* ptr = **NULL**,\*str = \*strHead;  
 ptr = (**struct** AFD \*)malloc(**sizeof**(**struct** AFD));  
 **if** (FILE\_RUNNING\_NUM == **USER\_RUN\_FILE\_NUM** ){  
 *//判断当前打开文件的个数,如果超过最大值,则不允许再打开* printf(**"当前打开文件已经到达可打开上限,不能再打开了\n"**);  
 **return**;  
 } **else if**(\*strHead == **NULL**){  
 *//原链表为空* \*strHead = ptr;  
 } **else**{  
 **while**(str->next != **NULL**){ *//将指针指向AFD链表的末尾* str = str->next;  
 }  
 str->next = ptr; *//将新创建的AFD置于AFD链表的末尾* }  
  
 **while** (p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
  
 strcpy(ptr->file\_name,p->file\_name);  
 ptr->file\_num = FILE\_RUNNING\_NUM + 1;  
*// printf("input file protect num: ");  
// scanf("%d",&ptr->file\_protect\_num);* ptr->file\_protect\_num = p->protect\_num;  
 ptr->next = **NULL**;  
 ptr->read\_write\_ptr = 0;  
 count = 1;  
 FILE\_RUNNING\_NUM ++;  
 **break**;  
 }  
 }  
  
 **if** (count){  
 printf(**"open success !\n"**);  
 }**else**{  
 printf(**"open failed \n"**);  
 }  
  
}  
  
*//关闭文件***void** closeFile(**struct** AFD \*str,**char** fileName[30]){  
 **int** count = 0;  
 *//遍历打开文件目录* **while**(str != **NULL**){  
 **if** (strcmp(str->file\_name,fileName) == 0){  
 **struct** AFD\* o;  
 o = str;  
 str = str->next;  
 free(o);  
 count = 1;  
 **break**;  
 }**else**{  
 str = str->next;  
 }  
 }  
 **if** (count == 1){  
 printf(**"colse successful !\n"**);  
 }**else**{  
 printf(**"no such file in openFile !\n"**);  
 }  
  
}  
  
*//读文件内容***void** readFile(**struct** UFD \*p,**struct** AFD \*\*strHead,**char** fileName[30]){  
 **struct** AFD\* str = \*strHead;  
 **int** count = 0;  
 *//遍历打开文件目录* **while** (str != **NULL**){  
 **if** (strcmp(str->file\_name,fileName) == 0){  
 **if** (str->file\_protect\_num >= 5){  
 *//当保护码大于等于5的时候，文件可读  
 //遍历用户文件目录,将文件存放的地址赋给读写指针* **while** (p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
 str->read\_write\_ptr = p->file\_address;  
 **break**;  
 }**else**{  
 p = p->next;  
 }  
 }  
 count = 1;  
 **break**;  
 }  
 }**else**{  
 str = str->next;  
 }  
  
 }  
 **if** (count == 0){  
 *//如果打开文件目录中没有当前文件,则进行对用户目录的遍历* **while**(p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
 *//可读访问权限* **if** (p->protect\_num >= 5){  
 openFile(p,strHead,fileName);  
 count = 1;  
 **break**;  
 }  
 } **else**{  
 p = p->next;  
 }  
 }  
 }  
  
 **if** (count == 1){  
 printf(**"read success !\n"**);  
 }**else**{  
 printf(**"no such file or no root !\n"**);  
 }  
  
}  
  
*//写文件内容***void** writeFile(**struct** UFD \*p,**struct** AFD \*\*strHead,**char** fileName[30]){  
 **struct** AFD\* str = \*strHead;  
 **int** count = 0;  
 *//遍历打开文件目录* **while** (str != **NULL**){  
 **if** (strcmp(str->file\_name,fileName) == 0){  
 **if** (str->file\_protect\_num == 7){  
 *//当保护码等于7的时候，文件可写  
 //遍历用户文件目录,将文件存放的地址赋给读写指针* **while** (p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
 str->read\_write\_ptr = p->file\_address;  
 **break**;  
 }**else**{  
 p = p->next;  
 }  
 }  
 count = 1;  
 **break**;  
 }  
 }**else**{  
 str = str->next;  
 }  
 }  
  
 **if** (count == 0){  
 *//如果打开文件目录中没有当前文件,则进行对用户目录的遍历* **while**(p != **NULL**){  
 **if** (strcmp(p->file\_name,fileName) == 0){  
 *//可写访问权限* **if** (p->protect\_num == 7){  
 openFile(p,strHead,fileName);  
 count = 1;  
 **break**;  
 }  
 } **else**{  
 p = p->next;  
 }  
 }  
 }  
  
 **if** (count == 1){  
 printf(**"write success !\n"**);  
 }**else**{  
 printf(**"no such file or no root !\n"**);  
 }  
}  
  
*//显示当前用户的文件目录  
/\*\*  
 \*  
 \* @param p  
 \* @param fileName  
 \*/***void** showFile(**struct** UFD \*p){  
 *//遍历用户文件目录* **if** (p == **NULL**){  
 printf(**"no file\n"**);  
 }**else**{  
 **while** (p != **NULL**){  
 printf(**"%s \n"**,p->file\_name);  
 p = p->next;  
 }  
 }  
}

**Main.c**

#include **<stdio.h>**#include **"fileSystem.h"***//比较命令,选择函数***void** choose(**char** command[30],**char** fileName[50],**struct** UFD \*\*p,**struct** AFD \*\*str,**struct** MFD \*m,**int** \*count){  
 **int** i;  
 printf(**"command = %s \n"**,command);  
  
 **if**(strcmp(command,**"create"**) == 0){  
 i = 0;  
 }**else if**(strcmp(command,**"exit"**) == 0){  
 \*count = 0;  
 i = 7;  
 }**else if**(strcmp(command,**"delete"**) == 0){  
 i = 1;  
 }**else if**(strcmp(command,**"open"**) == 0){  
 i = 2;  
 }**else if**(strcmp(command,**"close"**) == 0){  
 i = 3;  
 }**else if**(strcmp(command,**"read"**) == 0){  
 i = 4;  
 }**else if**(strcmp(command,**"write"**) == 0){  
 i = 5;  
 }**else if**(strcmp(command,**"ls"**) == 0){  
 i = 6;  
 }**else** {  
 printf(**"please input legal input \n"**);  
 **return**;  
 }  
 *//printf("i = %d\n",i);* **switch**( i ){  
 **case** 0:  
 *// printf("fileName = %s \n",fileName);* createFile(p,fileName);  
 *// printf("fileName = %s",(\*p)->file\_name);* **break**;  
 **case** 1:  
 deleteFile(\*p,str,fileName);  
 **break**;  
 **case** 2:  
 openFile(\*p,str,fileName);  
 **break**;  
 **case** 3:  
 closeFile(\*str,fileName);  
 **break**;  
 **case** 4:  
 readFile(\*p,str,fileName);  
 **break**;  
 **case** 5:  
 writeFile(\*p,str,fileName);  
 **break**;  
 **case** 6: *//ls  
 //printf("%s\n",(\*p)->file\_name);* showFile(\*p);  
 **break**;  
 **case** 7:  
 printf(**"exit\n"**);  
 exit(0);  
 }  
}  
  
**void** init(**struct** UFD \*p, **struct** MFD\* m, **struct** AFD\* str){  
 *//表头指针* p = **NULL**;  
 m = **NULL**;  
 str = **NULL**;  
  
}  
  
**int** main() {  
 **char** command[30],fileName[30];  
 **struct** UFD \*p = **NULL**;  
 **struct** AFD \*str = **NULL**;  
 **struct** MFD \*m = **NULL**;  
 **int** count = 1;  
 **while** (count){  
 printf(**"\*\*\*\*\*\*\*\*\*\* HELP MENU \*\*\*\*\*\*\*\*\*\*\n"**);  
 printf(**"1. 创建文件: create + file\n"**);  
 printf(**"2. 删除文件: delete + file\n"**);  
 printf(**"3. 打开文件: open + file\n"**);  
 printf(**"4. 关闭文件: close + file\n"**);  
 printf(**"5. 读取文件: read + file\n"**);  
 printf(**"6. 写入文件: write + file\n"**);  
 printf(**"7. 显示目录: ls \n"**);  
 printf(**"8. 退出： exit \n"**);  
 printf(**"输入你的命令及要操作的文件:"**);  
 scanf(**"%s %s"**,command,fileName);  
 *//getchar();  
 //init(p,m,str);* choose(command,fileName,&p,&str,m ,&count);  
 *//printf("%d",count);* }  
 **return** 0;  
}