**武汉大学计算机学院**

**本科生实验报告**

**操作系统设计与实现**

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：操作系统设计

指 导 教 师 ：曾平

学 生 学 号 ：2017302580198

学 生 姓 名 ：邹鑫

二○一九年五月

**郑 重 声 明**

本人呈交的实验报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本实验报告不包含他人享有著作权的内容。对本实验报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本实验报告的知识产权归属于培养单位。

本人签名： 邹鑫 日期： 2019.05.13

**摘要**

为了进一步对操作系统对系统各个方面的处理进行更深入的了解，本文做了五次实验，从五个不同的角度去探讨操作系统的作用。五个实验的基本内容如下：

实验一：处理器调度，设计一个调度算法，对处理器调度过程进行简单的模拟。本文设计了一个按优先数调度算法[1]执行的处理器模拟过程。

实验二：主存空间的分配和回收，采用首次适应算法[2]，对可变分区管理方式下主存的分配和回收。

实验三：磁盘存储空间的分配和回收，用位示图管理[3]存储空间，实现对磁盘存储的分配和回收的模拟。

实验四：进城创建，利用fork()函数[4]进行进程的创建，由于windows系统不再支持fork()函数的调用，本文使用linux虚拟机为环境进行实验。

实验五：进程同步，以生产者-消费者问题[5]为原型，模拟操作系统的P、V操作[6]以及其同步管理机制，模拟进程同步过程。

**关键词**

操作系统、处理器调度、主存、分配和回收、进程同步。

目录

[1 处理器调度 - 1 -](#_Toc30571)

[1.1 实习内容及上机实验所用平台 - 1 -](#_Toc7110)

[1.1.1 实习内容 - 1 -](#_Toc9868)

[1.1.2 设计思路 - 1 -](#_Toc17702)

[1.1.3 上机实验所用平台 - 2 -](#_Toc31047)

[1.2 数据结构及代码段分析 - 2 -](#_Toc1801)

[1.2.1 数据结构 - 2 -](#_Toc4481)

[1.2.2 代码段分析 - 2 -](#_Toc85)

[1.3 调试过程 - 4 -](#_Toc19664)

[1.3.1 数据输入 - 4 -](#_Toc4454)

[1.3.2 人为结果预测 - 4 -](#_Toc9921)

[1.3.3 运行结果 - 8 -](#_Toc22988)

[1.4 实验总结 - 8 -](#_Toc21000)

[2 主存空间的分配和回收 - 9 -](#_Toc13608)

[2.1 实习内容及上机实验所用平台 - 9 -](#_Toc26752)

[2.1.1 实习内容 - 9 -](#_Toc14268)

[2.1.2 设计思路 - 9 -](#_Toc20459)

[2.1.3 上机实验所用平台 - 9 -](#_Toc28841)

[2.2 数据结构及代码段分析 - 10 -](#_Toc8642)

[2.2.1 数据结构 - 10 -](#_Toc9969)

[2.2.2 代码段分析 - 10 -](#_Toc5796)

[2.3 调试过程 - 16 -](#_Toc12215)

[2.3.1 测试为进程分配空间 - 16 -](#_Toc14886)

[2.3.2 测试释放空间，不与前、后空闲区合并 - 17 -](#_Toc12118)

[2.3.3 测试释放空间，与前空闲区合并 - 18 -](#_Toc4609)

[2.3.4 测试释放空间，与后空闲区合并 - 18 -](#_Toc25660)

[2.3.5 测试释放空间，与前、后空闲区均合并 - 19 -](#_Toc31746)

[2.4 实验总结 - 20 -](#_Toc20922)

[3 磁盘存储空间的分配和回收 - 20 -](#_Toc24613)

[3.1 实习内容及上机实验所用平台 - 20 -](#_Toc4638)

[3.1.1 实习内容 - 20 -](#_Toc181)

[3.1.2 上机实验所用平台 - 21 -](#_Toc4820)

[3.2 数据结构及代码段分析 - 21 -](#_Toc16509)

[3.2.1 数据结构 - 21 -](#_Toc21093)

[3.2.2 代码段分析 - 21 -](#_Toc14137)

[3.3 调试过程 - 25 -](#_Toc14213)

[3.3.1 初始界面 - 25 -](#_Toc28405)

[3.3.2 分配长度为10的进程p1 - 25 -](#_Toc4508)

[3.3.3 分配长度为5的进程p2 - 26 -](#_Toc14584)

[3.3.4 输入一个过大的进程 - 26 -](#_Toc5726)

[3.3.5 显示位示图 - 26 -](#_Toc16235)

[3.3.6 回收p1并显示位示图 - 27 -](#_Toc3449)

[3.3.7 回收一个不存在的进程 - 27 -](#_Toc24801)

[3.3.8 创建已存在的进程 - 28 -](#_Toc8802)

[3.3.9 回收进程p2并显示位示图 - 28 -](#_Toc3427)

[3.4 实验总结 - 28 -](#_Toc21952)

[4 进程创建 - 29 -](#_Toc22714)

[4.1 实习内容及上机实验所用平台 - 29 -](#_Toc4877)

[4.1.1 实习内容 - 29 -](#_Toc21501)

[4.1.2 上机所用平台 - 29 -](#_Toc21604)

[4.2 数据结构及代码段分析 - 29 -](#_Toc4280)

[4.2.1 数据结构 - 29 -](#_Toc19861)

[4.2.2 代码段分析 - 30 -](#_Toc549)

[4.3 调试过程 - 31 -](#_Toc8431)

[4.4 实验总结 - 31 -](#_Toc21837)

[5 进程同步 - 32 -](#_Toc29551)

[5.1 实习内容及上机实验所用平台 - 32 -](#_Toc24330)

[5.1.1 实习内容 - 32 -](#_Toc8997)

[5.1.2 上机实验所用平台 - 32 -](#_Toc30529)

[5.2 数据结构及代码段分析 - 32 -](#_Toc14869)

[5.2.1 数据结构 - 32 -](#_Toc4817)

[5.2.2 代码段分析 - 33 -](#_Toc18660)

[5.3 调试过程 - 39 -](#_Toc14852)

[5.3.1 执行P操作前后对比 - 40 -](#_Toc28679)

[5.3.2 生产者放入产品 - 40 -](#_Toc10122)

[5.3.3 消费者取出产品 - 41 -](#_Toc21327)

[5.4 实验总结 - 41 -](#_Toc31811)

[6 参考文献 - 41 -](#_Toc21263)

# 1 处理器调度

## 1.1 实习内容及上机实验所用平台

### 1.1.1 实习内容

设计一个按优先数调度算法实现处理器调度的程序。

假定系统有5个进程，每个进程用一个PCB来表示，PCB的组成为：进程名、指向下一个进程的指针、要求的运行时间、优先数、状态。其中，要求的运行时间是假定的进程运行的单位时间数；优先数越高，进程的优先级越高；进程只有两种状态——就绪和结束，用R表示就绪状态、E表示结束状态，进程的初始状态都为就绪状态。

运行步骤：

（1）开始运行之前，通过键盘输入为每个进程确定优先数和要求运行时间。

（2）处理器总是选择队首的进程运行，进程每运行一次，优先数减一，要求运行时间减一。

（3）进程运行一次之后，若要求运行时间不为0，则将其加入就绪队列排队，否则，将其状态改为“结束”，退出就绪队列。

（4）若就绪队列为空，则结束，否则转（2）重复，直至就绪队列为空为止。

### 1.1.2 设计思路

要完成这项任务，首先要确定数据结构，即PCB用什么来实现。可以用类，也可以用结构体，由于此问题比较简单，故本文决定使用结构体实现。将PCB做成一个链表当做队列，开始时进行交互，人为从命令行输入每个进程的要求运行时间和优先数。创建完成后，用一个while(链表不为空)的循环，在循环里做进程的调度——每次取队首的PCB，输出“P1 is running”（第一个进程），然后将其要求时间和优先数减一，判断要求运行时间是否为0，若不为0，继续加入队列（按优先数排序，如果优先数相同，则放在前面），直至队列为空。

### 1.1.3 上机实验所用平台

本实验所用的计算机操作系统为windows系统（win10家庭版），使用的集成开发环境为Visual Studio 2017，使用C++编程语言进行实现。

## 1.2 数据结构及代码段分析

### 1.2.1 数据结构

这个题目需要用到的数据结构只有一个，即PCB：

struct PCB

{

const char\* name;//进程名称

PCB\* next;//下一个pcb首地址

int runTime;//要求运行时间

int priority;//优先级

char state;//E 结束 R就绪

};

### 1.2.2 代码段分析

#### 1.2.2.1 insert函数

void insert(PCB\* H,PCB\* p) {//将p插入链表H中，并按从大到小的顺序排队

PCB \* pb = H;

PCB \* t = H->next;

while (t != NULL)

{

if (t->priority > p->priority) {

pb = pb->next;

t = t->next;

}

else {// 相等时p在前面

break;

}

}

p->next = t;

pb->next = p;

}

主要作用：将一个进程PCB按顺序插入队列中，插在优先数比它大的进程后面，插在优先数比他小的进程前面，对于优先数和它相等的进程，则将其插入在前面（这其实无伤大雅）。

#### 1.2.2.2 pop函数

PCB\* pop(PCB\* H) {

if (H->next != NULL) {

PCB \* ret = H->next;

H->next = H->next->next;

return ret;

}

else {

return NULL;

}

}

主要作用，将队首的进程取出来。

#### 1.2.2.3 main函数

由于main函数内容较多，不方便放在正文，本文将其放在了附录中。主要作用：实现本次实验要求的业务逻辑。

## 1.3 调试过程

### 1.3.1 数据输入

本文决定随机输入测试数据如表1.1。

表1.1 测试初始输入表

|  |  |  |
| --- | --- | --- |
| 进程名 | 初始优先数 | 初始要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 5 | 2 |
| P4 | 2 | 1 |
| P5 | 4 | 4 |

### 1.3.2 人为结果预测

现在，本文将一步一步地进行分析，求出预测的结果。

初始状态下，各进程状态如表1.2

表1.2 初始状态

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 5 | 2 |
| P4 | 2 | 1 |
| P5 | 4 | 4 |

队列（队首在左）：P3,P5,P1,P4,P2(因为进程2先入队，所以根据上文的规则，进程4在前面)

（1）P3运行后，各进程状态如表1.3

表1.3 P3运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 4 | 1 |
| P4 | 2 | 1 |
| P5 | 4 | 4 |

队列：P3,P5,P1,P4,P2

（2）P3运行后，各进程状态如表1.4

表1.4 P3运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 4 | 4 |

队列：P5,P1,P4,P2

（3）P5运行后，各进程状态如表1.5

表1.5 P5运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 3 | 3 |

队列：P5,P1,P4,P2

（4）P5运行后，各进程状态如表1.6

表1.6 P5运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 3 | 3 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 2 | 2 |

队列：P1,P5,P4,P2

（5）P1运行后，各进程状态如表1.7

表1.7 P1运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 2 | 2 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 2 | 2 |

队列：P1,P5,P4,P2

（6）P1运行后，各进程状态如表1.8

表1.8 P1运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 2 | 2 |

队列：P5,P4,P2,P1

（7）P5运行后，各进程状态如表1.9

表1.9 P5运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 2 | 1 |
| P5 | 1 | 1 |

队列：P4,P2,P5,P1

（8）P4运行后，各进程状态如表1.10

表1.10 P4运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 2 | 3 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 1 | 1 |

队列：P2,P5,P1

（9）P2运行后，各进程状态如表1.11

表1.11 P2运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 1 | 2 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 1 | 1 |

队列：P2,P5,P1

（10）P2运行后，各进程状态如表1.12

表1.12 P2运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 0 | 1 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 1 | 1 |

队列：P5,P1,P2

（11）P5运行后，各进程状态如表1.13

表1.13 P5运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 1 | 1 |
| P2 | 0 | 1 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 0 | 0 |

队列：P1,P2

（12）P1运行后，各进程状态如表1.14

表1.14 P1运行后

|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 0 | 0 |
| P2 | 0 | 1 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 0 | 0 |

队列：P2

（13）P2运行后，各进程状态如表1.15

表1.15 P2运行后

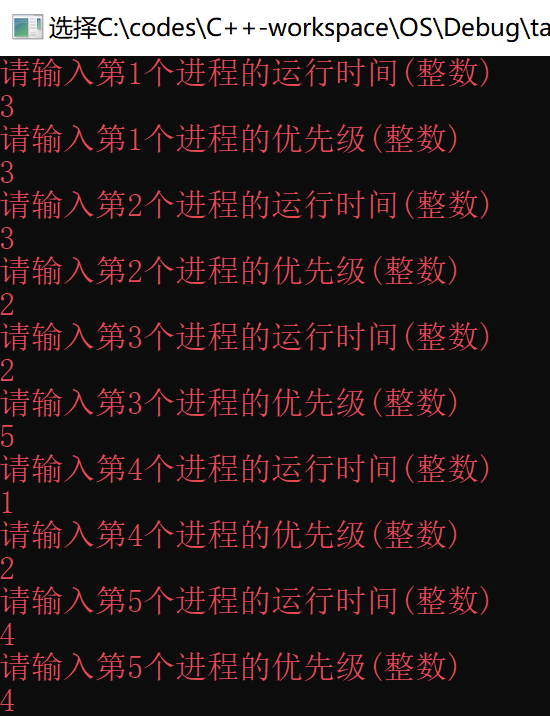
|  |  |  |
| --- | --- | --- |
| 进程名 | 优先数 | 要求运行时间 |
| P1 | 0 | 0 |
| P2 | -1 | 0 |
| P3 | 3 | 0 |
| P4 | 1 | 0 |
| P5 | 0 | 0 |

队列：空

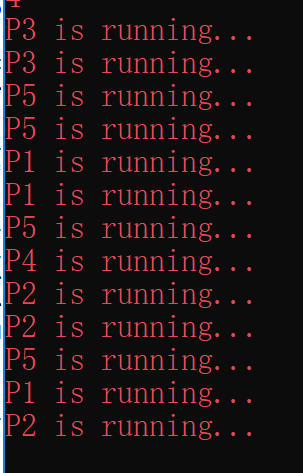
综上所述，我们可以得到预测的运行顺序为:P3,P3,P5,P5,P1,P1,P5,P4,P2,P2,P5,P1,P2。

### 1.3.3 运行结果

输入数据：



运行结果：



可以看到，运行的结果和预测的结果是一致的。

## 1.4 实验总结

本次实验相对来说较简单，还没有遇到什么麻烦。通过本次实验，我对操作系统原理中的进程调度又有了更深一步的理解，也让我复习了一下我的C++。

# 2 主存空间的分配和回收

## 2.1 实习内容及上机实验所用平台

### 2.1.1 实习内容

可变分区管理方式下采用首次适应算法实现主存分配和回收。当要装入一个作业时，根据作业需要的主存容量查看是否有足够的空闲空间，若有，则按需分配，否则，作业无法装入。假定内存大小为128K，空闲区说明表要说明分区号、起始地址、长度。采用首次适应算法分配回收内存空间。运行时，输入一系列分配请求和回收请求。

要求能接受来自键盘的空间申请及释放请求，能显示分区分配及回收后的内存布局情况。

### 2.1.2 设计思路

要完成这个实验任务，首先要创建一个进程的数据结构，由于此进程属性较少，结构简单，本文采用结构体来实现，同样地，也用一个结构体来实现空闲区。

要想实现首次适应算法，必须要将空闲区按首地址大小从小到大排列，空闲区大小满足，要将内存“分配”给进程，然后修改空闲区的大小。

还需要注意的一个问题时空闲区回收问题，当进程“释放”内存，应当考虑要不要将当前的空闲与其他空闲区合并为一个空闲区，这也是本实验的难点之一。解决方案为：遍历空闲区队列，当一个空闲区的首地址加上它的地址长度正好等于要释放的进程占用“内存”的首地址时，则表示该进程释放所得空闲区要和前一个空闲区合并，同理，当有一个空闲区的首地址等于要释放的进程加上其长度时，该空闲区即为即将释放内存的进程所占位置应该向后合并的空闲区。

### 2.1.3 上机实验所用平台

本实验所用的计算机操作系统为windows系统（win10家庭版），使用的集成开发环境为Visual Studio 2017，使用C++编程语言进行实现。

## 2.2 数据结构及代码段分析

### 2.2.1 数据结构

空闲区FreeZone：

struct FreeZone

{

int num;//空闲区号

int startAddress;//首地址

int length;//长度

FreeZone \* next;//指向下一个空闲区的指针

};

进程PCB：

struct PCB

{

int startAddress;//首地址

string name;//进程名

int length;//长度

PCB \* next;//指向下一个进程的指针

};

### 2.2.2 代码段分析

#### 2.2.2.1 searchPreOfPreZone函数

FreeZone \* searchPreOfPreZone(FreeZone \* fHead, PCB \* p) {

FreeZone \* tF = fHead->next;

FreeZone \* tPF = fHead;

while (tF != NULL) {

if (tF->startAddress + tF->length == p->startAddress) {

break;

}

else {

tF = tF->next;

tPF = tPF->next;

}

}

if (tF == NULL) {

return NULL;

}

else {

return tPF;

}

}

功能：寻找要释放的进程向前连接的空闲区的前一个空闲区的指针，如果没有，则返回null。

#### 2.2.2.2 searchPreOfNextZone函数

FreeZone \* searchPreOfNextZone(FreeZone \* fHead, PCB \* p) {

FreeZone \* tF = fHead->next;

FreeZone \* tPF = fHead;

while (tF != NULL) {

if (tF->startAddress == p->startAddress + p->length) {

break;

}

else {

tF = tF->next;

tPF = tPF->next;

}

}

if (tF == NULL) {

return NULL;

}

else {

return tPF;

}

}

功能：寻找要释放的进程向后连接的空闲区的前一个空闲区的指针，如果没有，则返回null。

#### 2.2.2.3 allocate函数

void allocate(FreeZone \* fHead, PCB \* pHead, int length, string name) {

if (nameRepeated(pHead, name)) {

cout << "已存在的进程名，申请失败" << endl;

return;

}

if (fHead->next != NULL) {

FreeZone \* tF = fHead->next, \*tBF = fHead;

while (tF != NULL) {

if (tF->length < length) {

tF = tF->next;

tBF = tBF->next;

}

else if (tF->length == length) {

PCB \* tPCB = new PCB();

tPCB->length = length;

tPCB->name = name;

tPCB->startAddress = tF->startAddress;

tPCB->next = NULL;

insertPCB(pHead, tPCB);

tBF->next = tF->next;

tBF = tF->next;

delete tF;

while (tBF != NULL) {

tBF->num--;

tBF = tBF->next;

}

break;

}

else {

PCB \* tPCB = new PCB();

tPCB->length = length;

tPCB->name = name;

tPCB->startAddress = tF->startAddress;

tPCB->next = NULL;

insertPCB(pHead, tPCB);

tF->length -= length;

tF->startAddress += length;

break;

}

}

if (tF == NULL) {

cout << "分配失败，内存空闲区大小不足" << endl;

}

}

else

{

cout << "分配失败，内存无空闲区" << endl;

showState(fHead, pHead);

}

}

功能：创建一个名为name，长度为length的进程，并为之分配空间，如果无空余空间，则输出提示语句。

#### 2.2.2.4 takeBack函数

void takeBack(FreeZone \* fHead, PCB \* pHead, string name) {

PCB \* p = getPCB(pHead, name);

if (p == NULL) {

cout << "回收出错，不存在名为\"" << name << "\"的进程" << endl;

}

else {

FreeZone \* preOfPreF = searchPreOfPreZone(fHead, p);

FreeZone \* preOfNextF = searchPreOfNextZone(fHead, p);

if (preOfPreF == NULL && preOfNextF == NULL) {//前后都不可连接

FreeZone \* f = new FreeZone();

f->length = p->length;

f->startAddress = p->startAddress;

f->next = NULL;

f->num = 1;

FreeZone \* tF = fHead;

while (tF->next != NULL && tF->next->startAddress < f->startAddress) {

tF = tF->next;

f->num++;

}

f->next = tF->next;

tF->next = f;

if (tF == fHead) {

f->num = 1;

}

f = f->next;

while (f != NULL) {

f->num++;

f = f->next;

}

}

else if (preOfPreF != NULL && preOfNextF != NULL) {//前后都要连接

FreeZone \* pre = preOfPreF->next;

FreeZone \* minusPoint = preOfNextF->next->next;

FreeZone \* delP = preOfNextF->next;

pre->length += p->length + preOfNextF->next->length;

pre->next = minusPoint;

while (minusPoint != NULL) {

minusPoint->num--;

minusPoint = minusPoint->next;

}

delete delP;

}

else if (preOfPreF != NULL && preOfNextF == NULL) {//只和前面连接

FreeZone \* pre = preOfPreF->next;

pre->length += p->length;

}

else {//只和后面连接

FreeZone \* next = preOfNextF->next;

next->startAddress = p->startAddress;

next->length += p->length;

}

delete p;

}

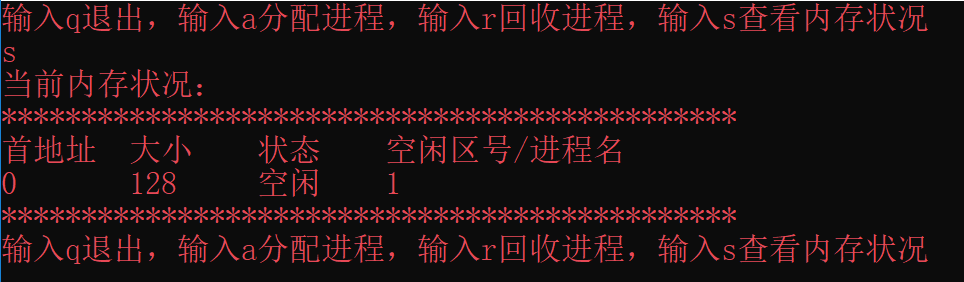
}

功能：让名为name的进程释放内存，并加入空闲区。

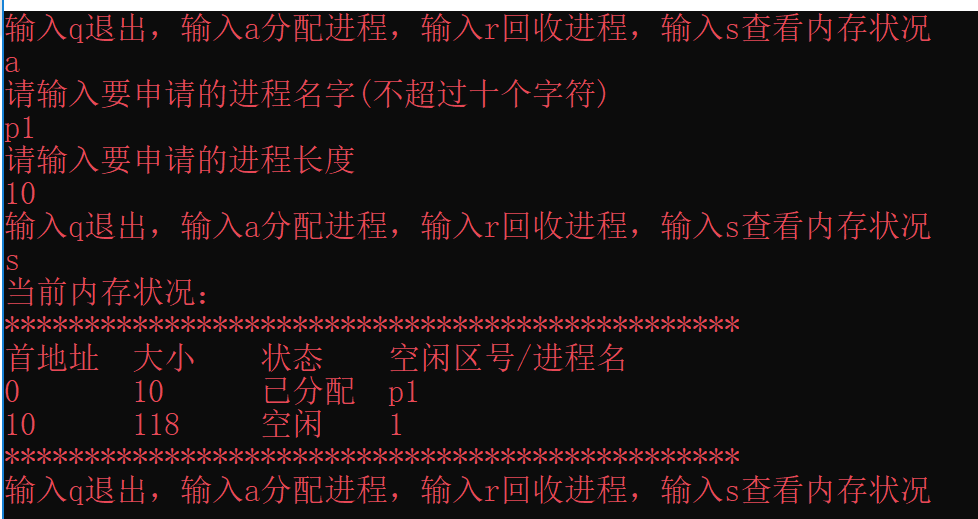
## 2.3 调试过程

### 2.3.1 测试为进程分配空间

#### 2.3.1.1 初始状态

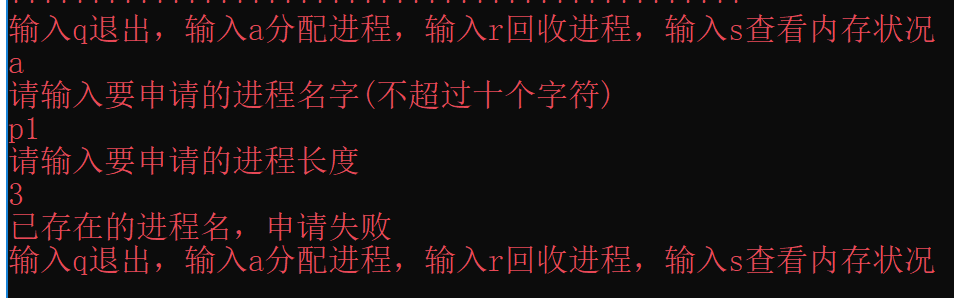


#### 2.3.1.2 为一个进程分配空间



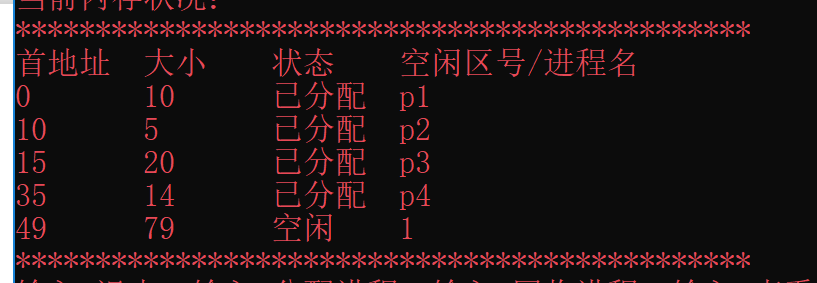
如图所示，创建了名为p1的进程，并为其分配了10K的空间。

#### 2.3.1.3 输入一个已存在的进程名

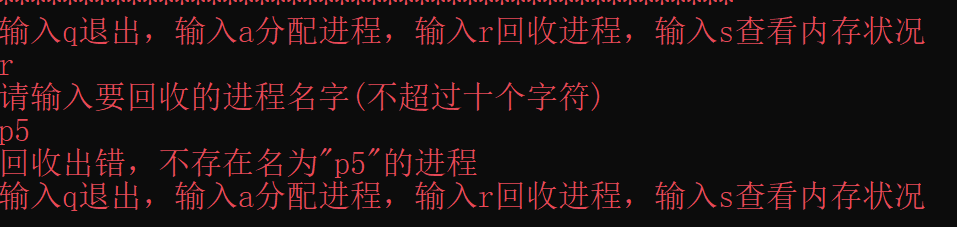


### 2.3.2 测试释放空间，不与前、后空闲区合并

先多申请几个进程，结果如下：

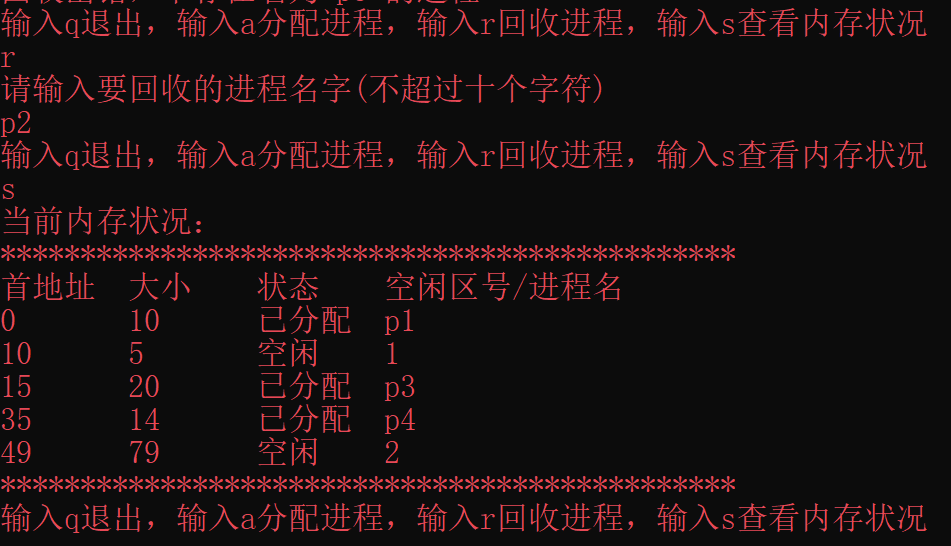


#### 2.3.2.1 释放一个不存在的进程



回收失败，并提示进程不存在，达到了想要的效果。

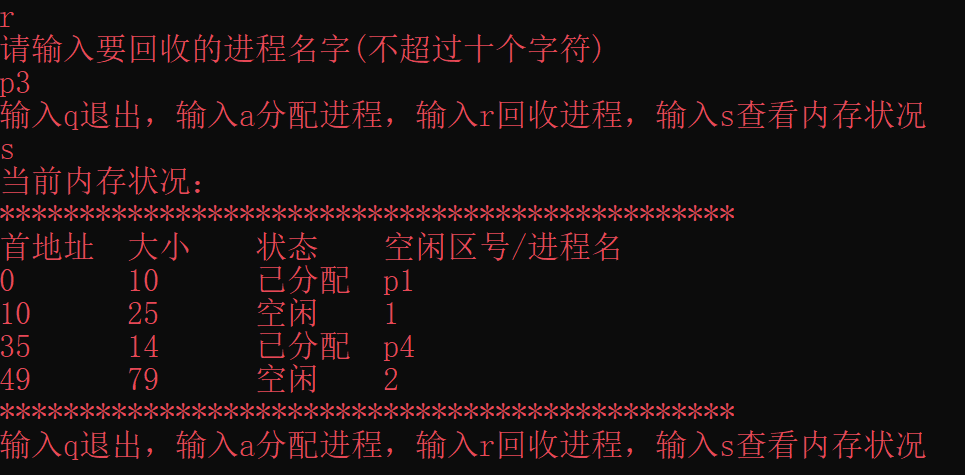
#### 2.3.2.2 释放p2



p2的内存已被回收，并且p2位置的空闲区编号变成了1，后面的空闲区编号变成了2，自动编号条件满足。

### 2.3.3 测试释放空间，与前空闲区合并

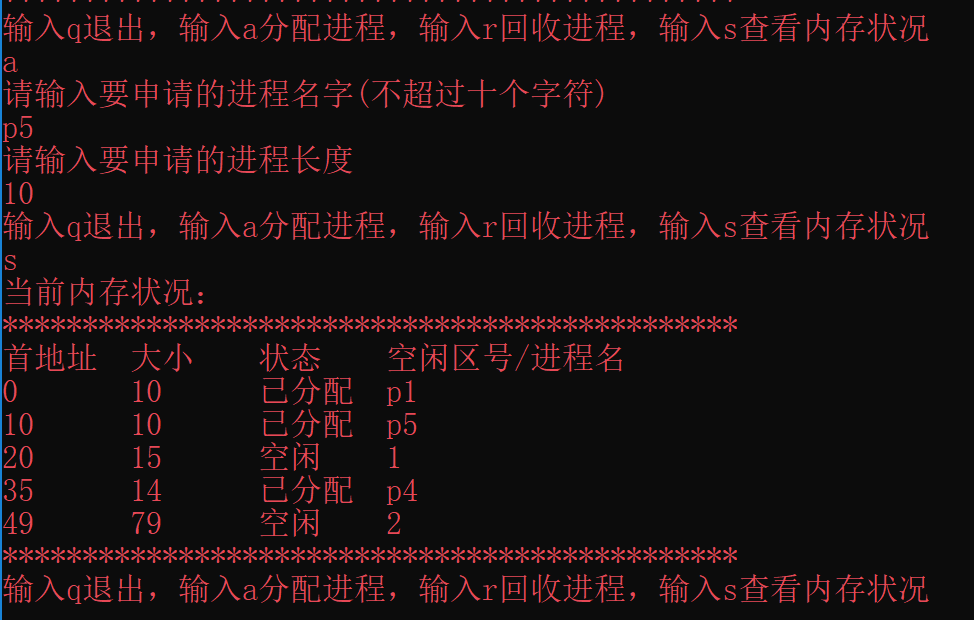
接着上面的结果，释放p3：



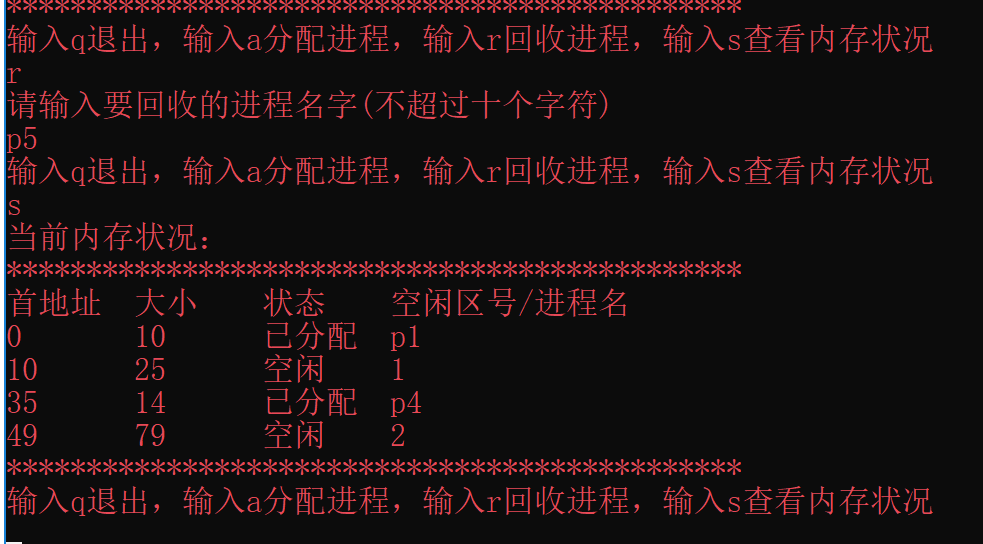
空闲区合并了，地址也变成了5+20 = 25，结果正确。

### 2.3.4 测试释放空间，与后空闲区合并

为方便起见，我们再在上文的基础上分配一个小于25的进程p5：



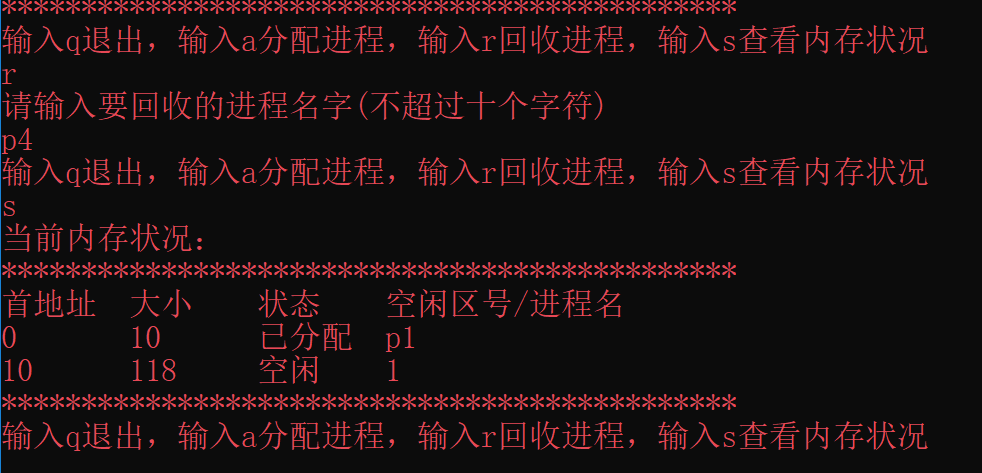
释放p5，看是否与空闲区1合并：



他们合并了。

### 2.3.5 测试释放空间，与前、后空闲区均合并

在刚才的基础上释放p4，看是否与空闲区1和空闲区2合并：



空闲区合并了，至此，各种检验都没出问题，完成了实验（源码见附录）。

## 2.4 实验总结

这次实验不仅使我对内存管理的首次适应算法有了更深层次的了解，有了更深的印象，还让我复习了一下数据结构和C++语言，我可以说是受益匪浅。

# 3 磁盘存储空间的分配和回收

## 3.1 实习内容及上机实验所用平台

### 3.1.1 实习内容

在磁盘储存管理中，为了表示哪些磁盘空间已被占用，哪些磁盘空间是空闲的，可用位示图来指出。位示图由若干字节构成，每一位与磁盘上的一块对应，“1”状态表示相应块已占用，“0”状态表示该块为空闲。本实验将模拟磁盘管理系统，采用位示图表明磁盘分配状态，并从通过命令行操作创建、撤销一个进程，以及显示创建时占用的磁盘空间物理地址，以及回收时的字节号和位数。并显示分配前后的位示图状态。

### 3.1.2 上机实验所用平台

本次实验使用的操作系统是windows10操作系统，使用的集成开发环境是VisualStudio2019。

## 3.2 数据结构及代码段分析

### 3.2.1 数据结构

进程表示：PCB

struct PCB

{

vector<int> bit\_num;//位号

int length;

friend bool operator < (const struct PCB& p1, const struct PCB& p2);

};

此数据结构表示进程PCB，其中，length表示此进程占用的空间大小（单位：bit），而vector——bit\_num则是一个存放此进程所占用空间的位数的动态数组，如：一个进程占了4个bit，分别的位数号码为：0,4,5,7，则vector中就有四个值：0,4,5,7。

bool bit\_map[8][8];

一个用于存位示图状态的二维bool型数组，若为true，则表示该处空闲，反之为占用。

### 3.2.2 代码段分析

#### 3.2.2.1 init函数

void init() {

for (int i = 0; i < 8; i++) {

for (int j = 0; j < 8; j++) {

bit\_map[i][j] = true;

}

}

}

作用：初始化位示图，全置为空闲。

#### 3.2.2.2 getFree函数

int getFree() {//返回一个空闲的位数 若无空闲，返回-1

for (int i = 0; i < 8; i++) {

for (int j = 0; j < 8; j++) {

if (bit\_map[i][j]) {

return i \* 8 + j;

}

}

}

return -1;

}

作用：找到当前一个空闲块并返回其位数，若全都被占用，则返回-1。

#### 3.2.2.3 free函数

void free(string p\_name) {

if (map\_PCB.count(p\_name) == 0) {//不存在

cout << "不存在该进程！" << endl;

}

else if (map\_PCB.count(p\_name) == 1) {

PCB free\_pcb = map\_PCB[p\_name];

map\_PCB.erase(p\_name);

for (int i = 0; i < free\_pcb.bit\_num.size(); i++) {

int bit = free\_pcb.bit\_num[i];

int byte = bit / 8;

cout << "回收 --- 字节号：" << byte << '\t' << "位数:" << bit << endl;

bit\_map[bit / 8][bit % 8] = true;

free\_bit\_num++;

}

}

}

作用，通过传入进程名释放该进程所占空间并显示释放的地址，若该进程不存在，则输入“不存在该进程！”。

#### 3.2.2.4 main函数中分配新进程的代码段

if (c == 'a') {

cout << "请输入进程名：" << endl;

string name;

cin >> name;

cout << "请输入进程长度：(单位：位)" << endl;

int plength;

cin >> plength;

if (plength > free\_bit\_num) {

cout << "剩余空间只有" << free\_bit\_num << "bit,不足以分配！" << endl;

}

else if (map\_PCB.count(name) == 1) {//已存在该进程

cout << "进程已存在！" << endl;

}

else {

PCB p;

p.length = plength;

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

int k = getFree();

int cylinder\_num = k / 8;//柱面号

int track\_num = k / 4;//磁道号

int sector\_num = k % 4;//扇区号

p.bit\_num.push\_back(k);

free\_bit\_num--;

bit\_map[k / 8][k % 8] = false;

cout << "分配 --- 柱面号：" << cylinder\_num << '\t' << "磁道号："

<< track\_num << '\t' << "扇区号：" << sector\_num << endl;

}

map\_PCB[name] = p;

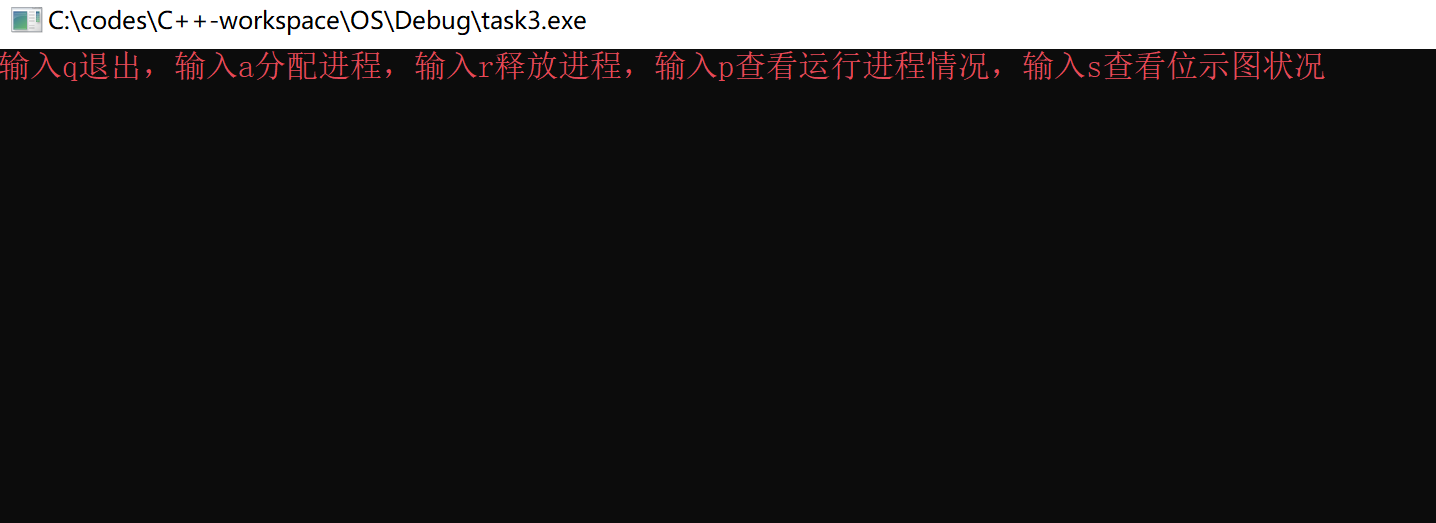
}

}

作用：根据用户输入分配进程并输出分配扇区的柱面号、磁道号、扇区号，若剩余空间不够，则提示“空间不足”，若已存在该进程，则提示已存在。

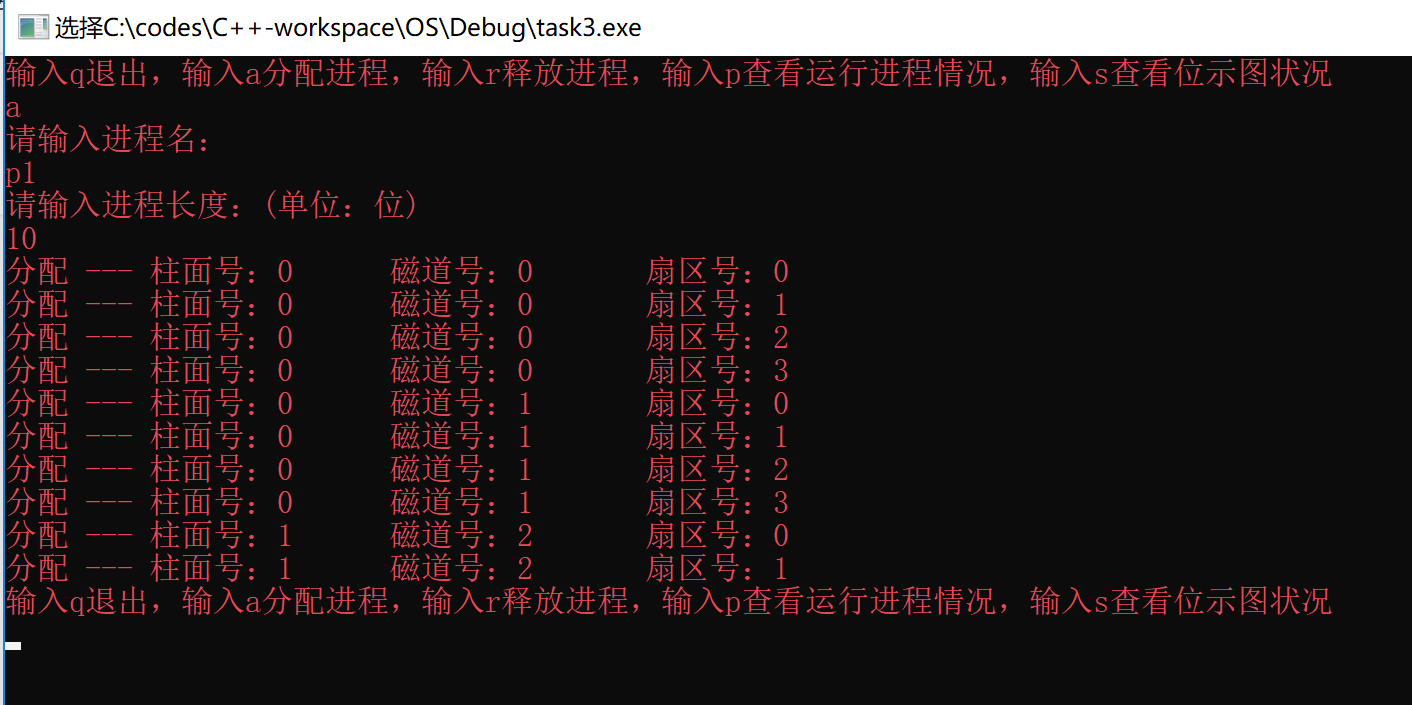
## 3.3 调试过程

### 3.3.1 初始界面



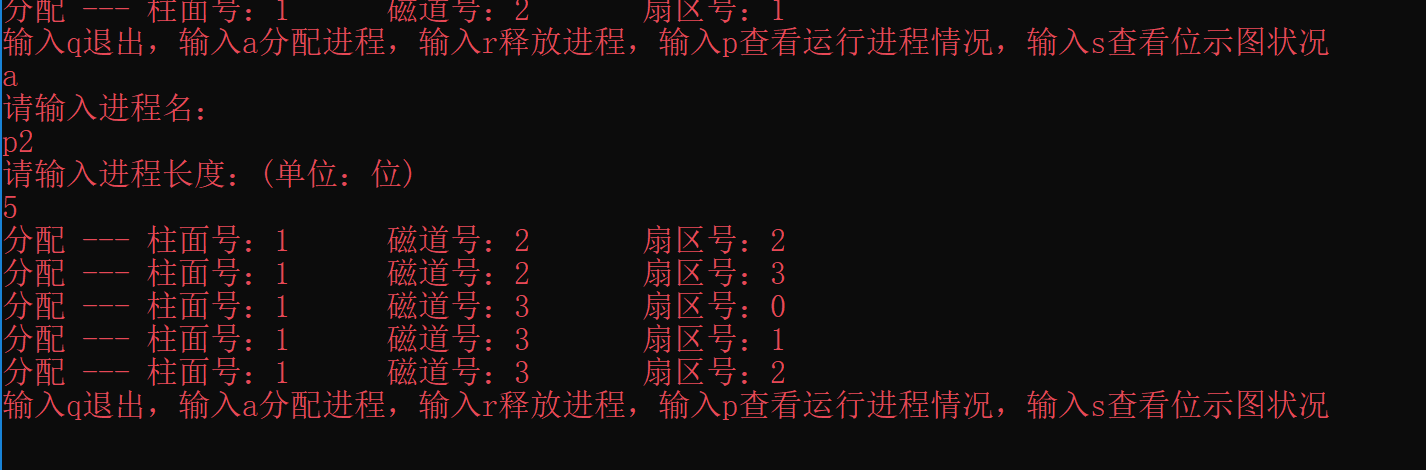
初始界面给出操作提醒。

### 3.3.2 分配长度为10的进程p1



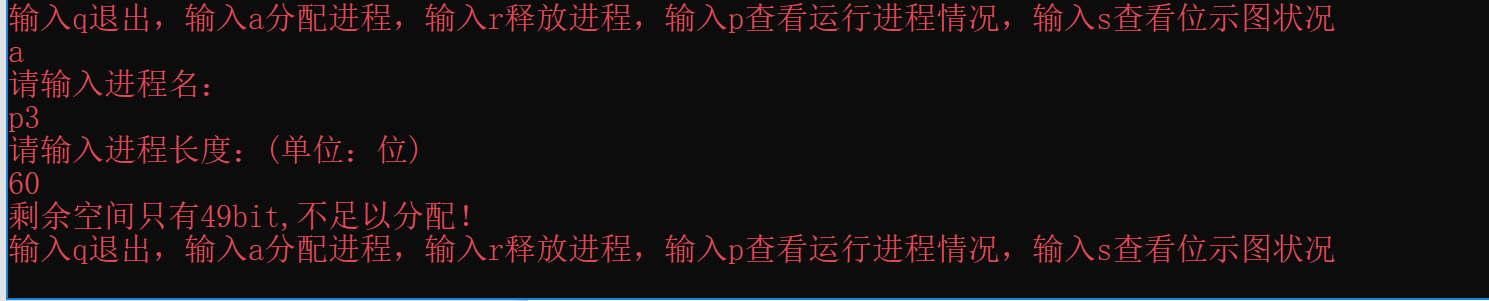
可以看到，本程序是位数的顺序寻找空闲块的，图中也给出了每个空闲块的柱面号、磁道号、扇区号。因为本实验要求8个扇区，每个扇区两个磁道，每个磁道4个扇区，对照图中所给的结果，易知分配是正确的。

### 3.3.3 分配长度为5的进程p2



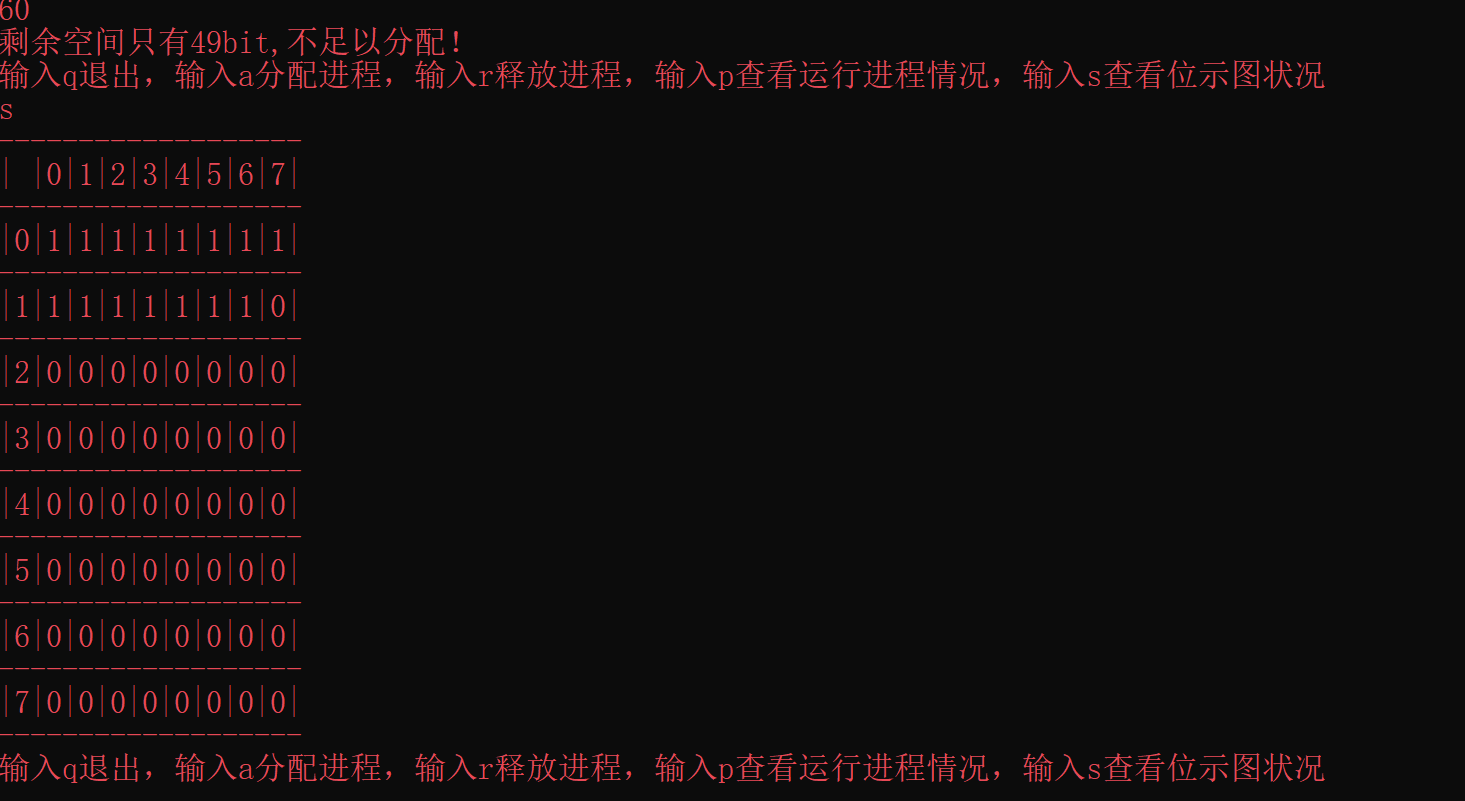
结果同上可分析，不再详述。

### 3.3.4 输入一个过大的进程



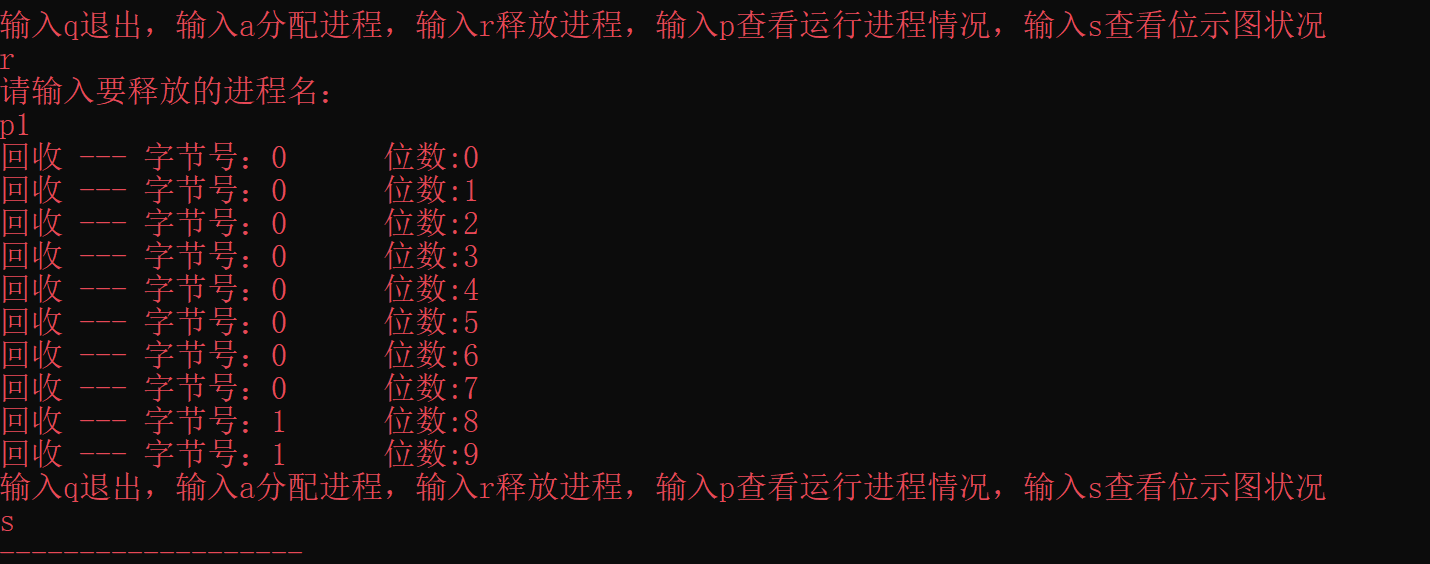
可以看见，提示分配失败，内存不足。

### 3.3.5 显示位示图



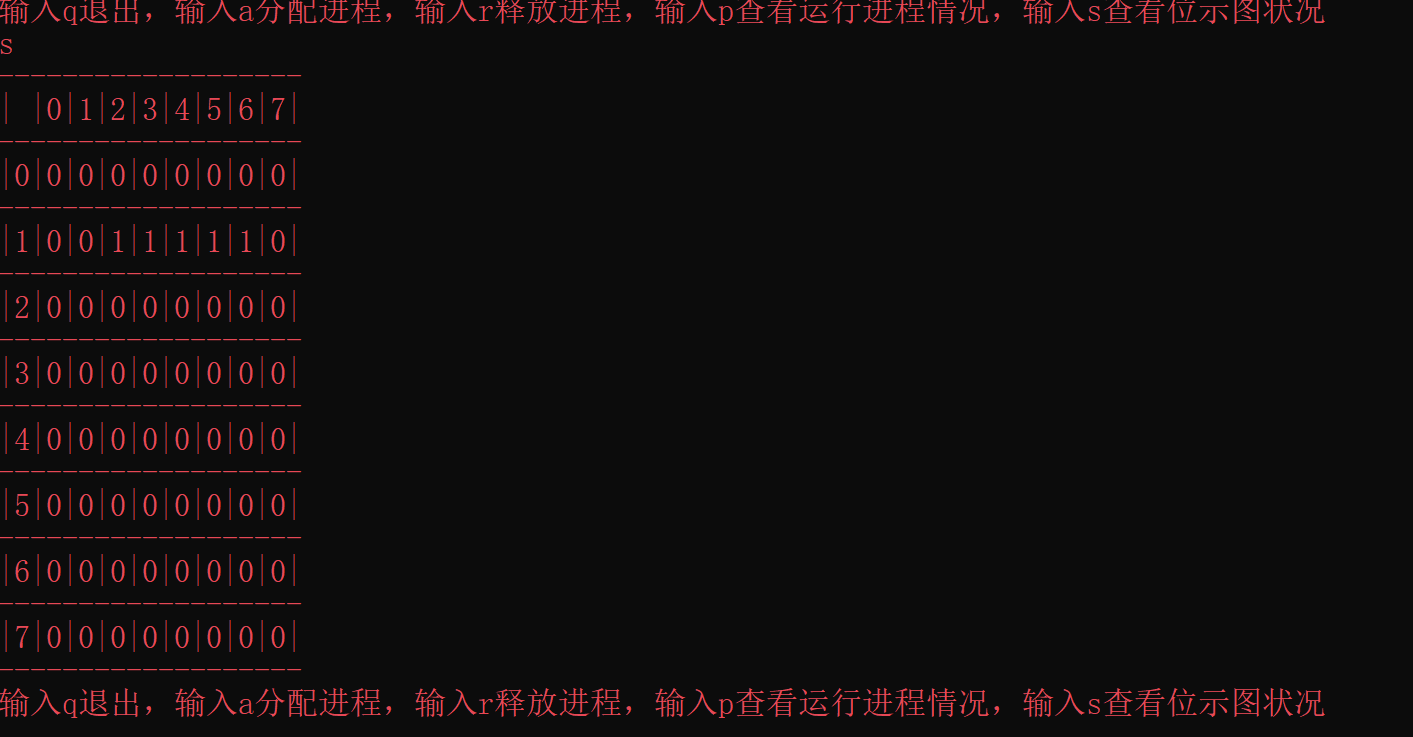
可以看到，第一行的8个和第二行的前7个都被占用，正好对应p1和p2占用的10bit和5bit。

### 3.3.6 回收p1并显示位示图



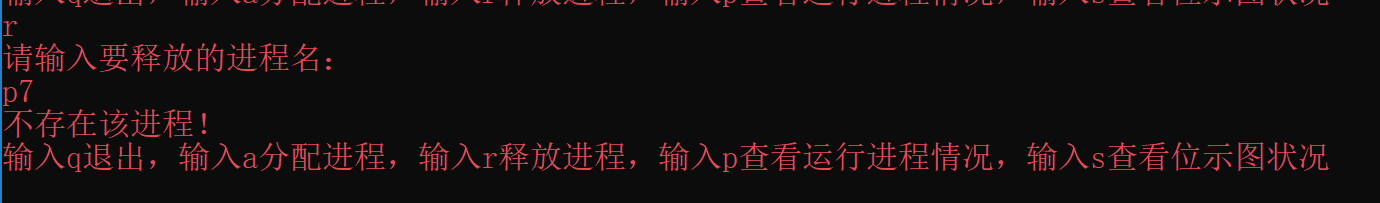
可以看到，回收的正好是之前给p1分配的前10个bit。

再显示位示图：



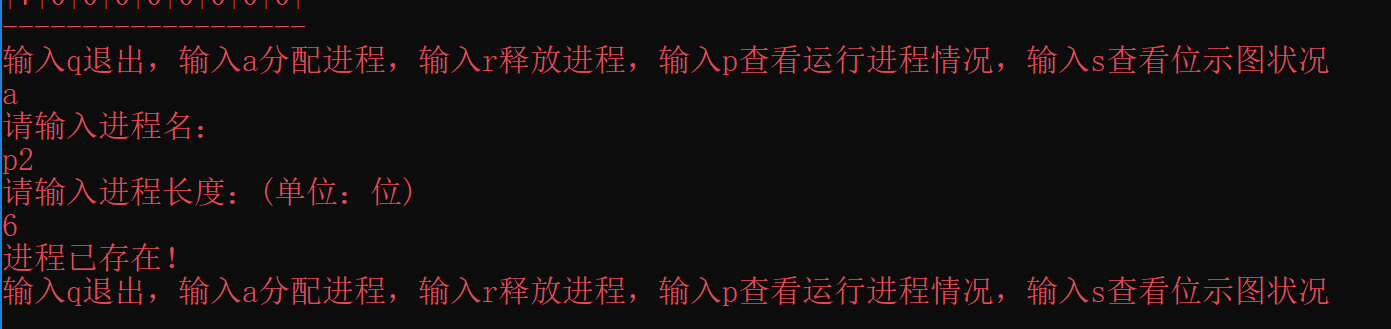
p1所占用的地方已经变成空闲。

### 3.3.7 回收一个不存在的进程



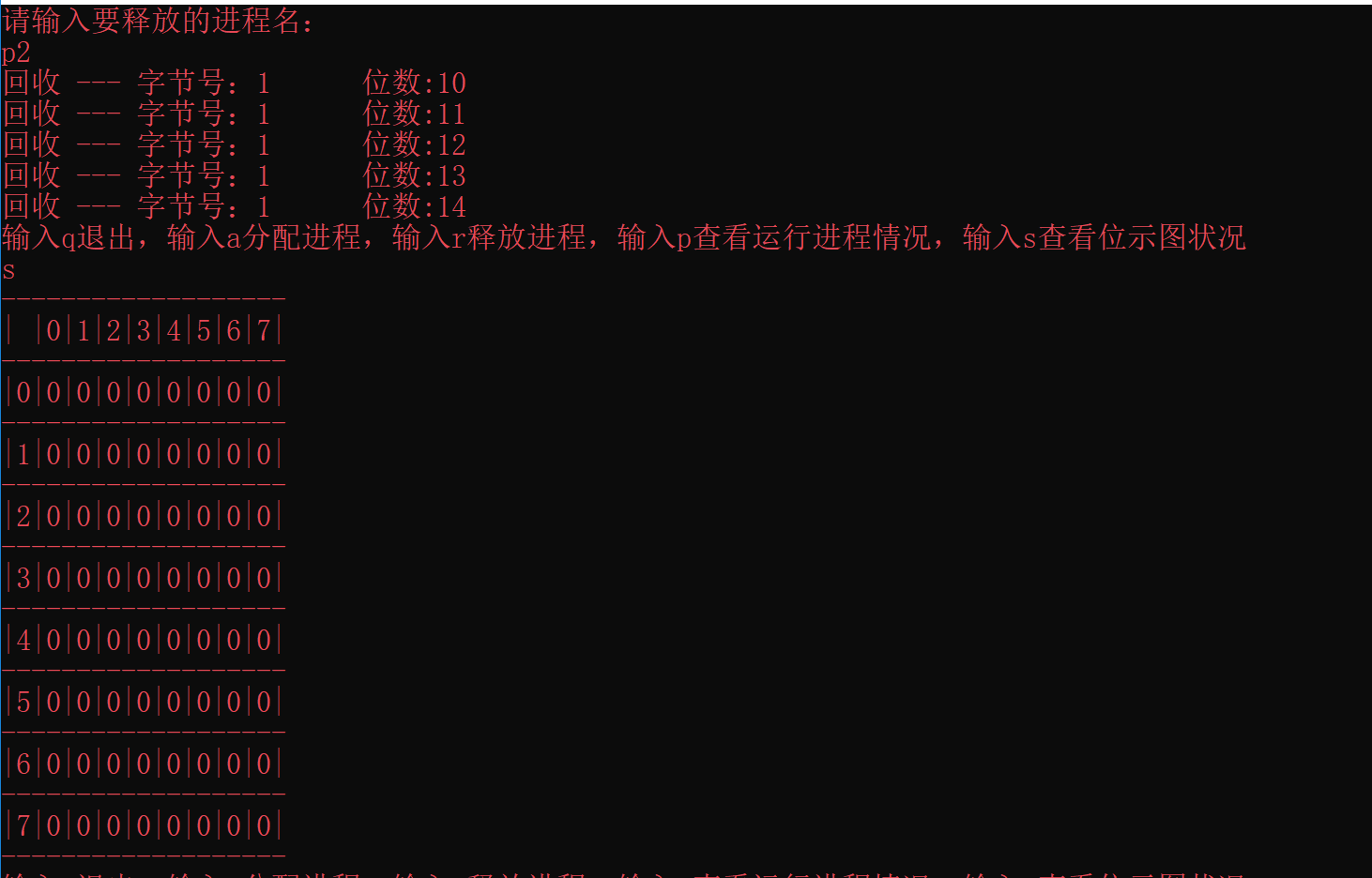
提示进程不存在。

### 3.3.8 创建已存在的进程



提示进程已存在，无法创建。

### 3.3.9 回收进程p2并显示位示图



回收p2后，位示图又全部恢复空闲状态。

至此，基本上可以说实验成功。

### 3.4 实验总结

刚开始做这个实验的时候，其实我们还没有学到文件系统这一章，刚开始我对位示图比较迷茫，书上的介绍也只有寥寥几句，我又去网上查找了资料，可结果也不尽如意。结合网络和书籍，我只能知道位示图用0表示空闲，1表示占用，且位示图中的每一个都对应一个存储单元，即扇区。至于书上位示图横向的0-15的编号以及纵向的数字编号是代表的什么我也不知道，虽然心中有一些猜测，但毕竟是猜测，拿不准，也不敢随便按猜想去做。后来，向老师请教之后，我总算明白并初步理解了位示图的用法。至于本实验的其他地方，只要知道位示图的使用的含义，其他地方还是比较简单的。

这次实验让我对文件系统有了进一步的了解，对位示图有了初步的掌握，也为我几周后学习这一章打下了一定的基础。

# 4 进程创建

## 4.1 实习内容及上机实验所用平台

### 4.1.1 实习内容

利用fork()系统调用创建进程。

编制一段程序，使用系统调用fork( )创建两个子进程，这样在此程序运行时，在系统中就有一个父进程和两个子进程在活动。每一个进程在屏幕上显示一个字符，其中父进程显示字符A，子进程分别显示字符 B和字符C。试观察、记录并分析屏幕上进程调度的情况。

### 4.1.2 上机所用平台

上机所用操作系统：Linux操作系统，使用工具：gcc。

## 4.2 数据结构及代码段分析

### 4.2.1 数据结构

本实验十分简单，无数据结构。

### 4.2.2 代码段分析

int main(){

pid\_t pid1,pid2;

pid1 = fork();

if( pid1 == -1 ){//child

printf("child1 failed\n");

}

else if( pid1 == 0){//child

write(STDOUT\_FILENO,"child1 write: B\n",sizeof("child1 write: B\n")-1);

}

else if ( pid1 > 0 ){//parent

pid2 = fork();

if(pid2 == -1){

printf("child2 failed\n");

}

else if(pid2 == 0){

write(STDOUT\_FILENO,"child2 write: C\n",sizeof("child2 write: C\n")-1);

}

else if(pid2 > 0){

write(STDOUT\_FILENO,"parent write: A\n",sizeof("parent write: A\n")-1);

}

}

waitpid(pid1,NULL,0);

waitpid(pid2,NULL,0);

return 0;

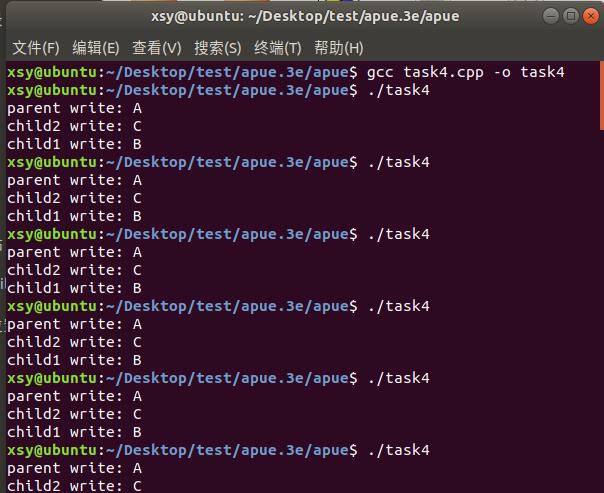
}

以上即本次实验所有代码，首先创建一个子进程，然后判断进程id——pid1是否为-1，若为-1则表明分配失败；若不为-1，再看pid1是否为0，若为0，表示这是第一个子进程，输出“child1 write: B”并换行；再看pid1是否大于0，若大于0，则为父进程，然后再创建第二个进程并获得其id——pid2，判断pid2是否为-1，若为-1，分配出错，若为0，则为第二个子进程，输出“child2 write: C”并换行，若大于0，则为父进程，输出“parent write: A”并换行。

最后，让父进程等待两个子进程创建成功后一起执行，模拟并发执行。

## 4.3 调试过程

本人做了几次尝试，但输出的顺序都是ACB，如下图：



## 4.4 实验总结

本次实验让我更加了解了linux环境下的fork()函数调用机制，更加了解进程的并发执行特征。

# 5 进程同步

## 5.1 实习内容及上机实验所用平台

### 5.1.1 实习内容

模拟实现同步机构，以避免发生进程执行时可能出现的与时间有关的错误。当进程并发执行时，如果对进程访问的共享变量不加限制，就会产生“与时间有关”的错误。为了防止这类错误，系统必须用同步机构来控制进程对共享变量的访问。一般说，同步机构是由若干条同步原语所组成。本实验要求学生模拟P、V操作同步机构的实现，模拟进程的并发执行，了解进程并发执行时同步机构的作用。主要是模拟P、V操作实现同步机构，且用P、V操作解决生产者—消费者问题。

### 5.1.2 上机实验所用平台

本次实验使用的操作系统是windows10操作系统，使用的集成开发环境是VisualStudio2019。

## 5.2 数据结构及代码段分析

### 5.2.1 数据结构

vector<PCB> runnable\_pool;//就绪池

vector<PCB> producer\_waiting\_pool;//等待池

vector<PCB> consumer\_waiting\_pool;

enum State{Running , Runnable ,Waiting ,Finished};

enum WaitReason{S1,S2,No};

enum ProInstruction { Produce, Ps1, Put, Vs2, GotoP0 };

enum ConInstruction { Ps2, Get, Vs1, Consume, GotoC0 };

enum Type{Producer,Consumer};

struct PCB

{

string name;

State state = Runnable;

WaitReason reason = No;//可以通过reason判断它是生产者还是消费者

int PC;//断点

Type type;

int product = 0;

};

为了完成本实验，本文使用了以上数据结构，其中——

State表示的是一个进程的状态，分为四个状态：运行、就绪、等待(即中断)、完成。

WaitReason表示的是中断的等待原因，S1指因为等待S1而中断，S2表示因等待S2而中断，而No表示没有中断。

ProInstruction主要是与生产者的五个指令所对应，而ConInstruction则与消费者的五个指令相对应。

Type这表示此进程是生产者还是消费者。

PCB代表一个进程控制块，其中的name则表示它所控制的进程的名字，state表示控制进程的状态，reason表示控制进程的等待原因，初始值为No，PC则记录了控制进程的中断位置，当它所控制的进程再次被唤醒并被执行时，则从中断处继续，而product表示生产者生产的产品或者消费者获得的产品，产品的范围为1-100，初值为0，当值为0时，表示没有产品。

### 5.2.2 代码段分析

#### 5.2.2.1 处理器的模拟函数handler

void handler() {

while (outputtime < stopTime) {

if (runnable\_pool.size() == 0) {

cout << "死锁！" << endl;

}

else {

int index = rand() % runnable\_pool.size();

PCB runningPCB = runnable\_pool[index];

runningPCB.state = Running;

erase(runningPCB.name, runnable\_pool);

if (runningPCB.type == Producer) {

ProInstruction instruction = PA[runningPCB.PC];

doProducer(runningPCB,instruction);

}

else if (runningPCB.type == Consumer) {

ConInstruction instruction = SA[runningPCB.PC];

doConsumer(runningPCB, instruction);

}

}

output();

}

}

此函数模拟了处理器的功能，由于不加制止的话本程序会一直运行，所以本文为此添加了终止条件，即outputtime >= stopTime，outputtime 表示的是执行指令的次数，由于每次执行了一条指令，就要输出一次，所以它也是输出的次数，而stopTime则表示执行总次数，执行到了这么多次指令就会停止。

从理论上分析，由于每次每个进程申请的资源都只有一个，所以是一定不会发生死锁的，不过本文还是决定给出判断条件：当就绪池为空时，则证明产生了死锁。然后随机从选出一个，执行其一条命令，执行后如果不中断，则其会被再次放入就绪池中。

#### 5.2.2.2 P操作

void P(int &s, PCB& p) {

s = s - 1;

if (s < 0)

{

p.state = Waiting;

if (&s == &s1) {

p.reason = S1;

}

else if (&s == &s2) {

p.reason = S2;

}

if (p.type == Producer) {

producer\_waiting\_pool.push\_back(p);

}

else if (p.type == Consumer) {

consumer\_waiting\_pool.push\_back(p);

}

}

else {

p.state = Runnable;

runnable\_pool.push\_back(p);

}

}

这个很简单，跟要求的一样，让参数s减一，并且如果它小于0，则将此进程放入中断池。

#### 5.2.2.3 V操作

void V(int &s, PCB& p) {

s = s + 1;

if (s <= 0)

{

if (&s == &s1) {//生产者等待s1

int index = rand() % producer\_waiting\_pool.size();

PCB weakP = producer\_waiting\_pool[index];

weakP.state = Runnable;

erase(weakP.name, producer\_waiting\_pool);

runnable\_pool.push\_back(weakP);

}

else if (&s == &s2) {//消费者等待s2

int index = rand() % consumer\_waiting\_pool.size();

PCB weakP = consumer\_waiting\_pool[index];

weakP.state = Runnable;

erase(weakP.name, consumer\_waiting\_pool);

runnable\_pool.push\_back(weakP);

}

}

p.state = Runnable;

runnable\_pool.push\_back(p);

}

此操作也很简单，和P操作相似，每次执行将s加一，若s<=0，则从中断池中随机唤醒一个进程。

#### 5.2.2.4 生产者和消费者执行指令模拟

void doProducer(PCB& p, ProInstruction instruction) {

p.PC = (p.PC + 1) % 5;

if (instruction == Produce) {

if (p.product == 0) {

p.product = rand() % 100 + 1;//1-100

p.state = Runnable;

runnable\_pool.push\_back(p);

cout << p.name << " has produced." << endl;

}

}

else if (instruction == Ps1) {

P(s1, p);

cout << p.name << " has done P" << endl;

}

else if (instruction == Put) {

if (products[in] == 0 && p.product != 0) {//如果该处无商品并且p生产了商品

products[in] = p.product;

p.product = 0;

in = (in + 1) % 10;

cout << p.name << " has done Put" << endl;

}

p.state = Runnable;

runnable\_pool.push\_back(p);

}

else if (instruction == Vs2) {

V(s2, p);

cout << p.name << " has done V" << endl;

}

else if (instruction == GotoP0) {

p.state = Finished;

cout << p.name << " has finished" << endl;

cout << p.name << " has goto 0" << endl;

p.state = Runnable;

runnable\_pool.push\_back(p);

}

}

void doConsumer(PCB& p, ConInstruction instruction) {

p.PC = (p.PC + 1) % 5;

if (instruction == Ps2) {

P(s2, p);

cout << p.name << " has done P" << endl;

}

else if (instruction == Get) {

if (products[out] != 0 && p.product == 0) {//如果该处有商品

p.product = products[out];

products[out] = 0;

out = (out + 1) % 10;

cout << p.name << " has Get product " << p.product << endl;

}

p.state = Runnable;

runnable\_pool.push\_back(p);

}

else if (instruction == Vs1) {

V(s1, p);

cout << p.name << " has done V" << endl;

}

else if (instruction == Consume) {

if (p.product != 0) {

p.product = 0;

}

cout << p.name << " has Consumed product " << p.product << endl;

p.state = Runnable;

runnable\_pool.push\_back(p);

}

else if (instruction == GotoC0) {

p.state = Finished;

cout << p.name << " has finished" << endl;

cout << p.name << " has goto 0" << endl;

p.state = Runnable;

runnable\_pool.push\_back(p);

}

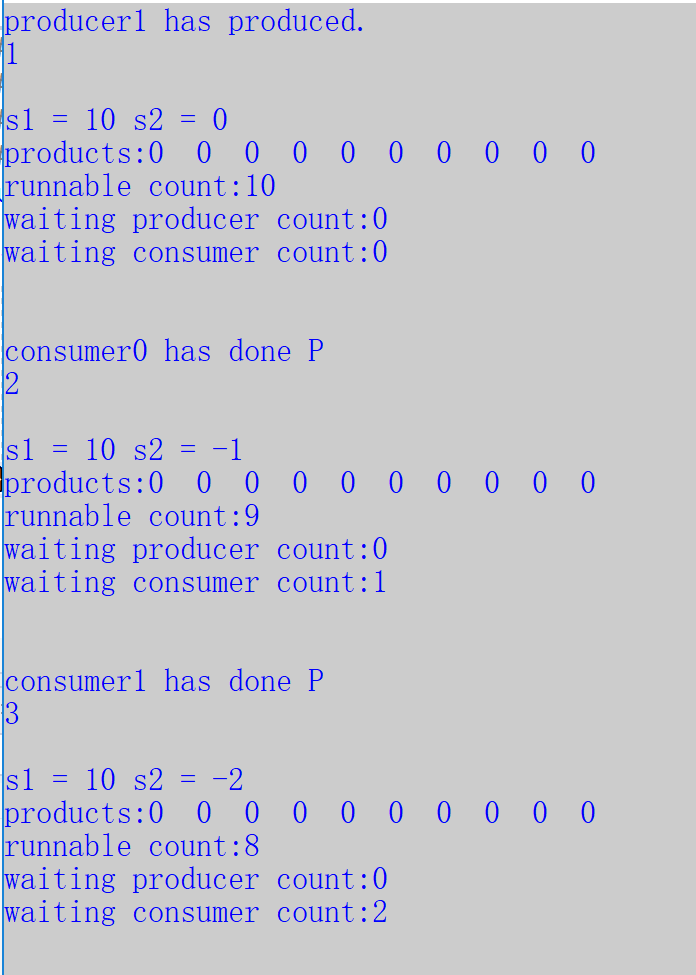
}

根据指令执行相应操作。

## 5.3 调试过程

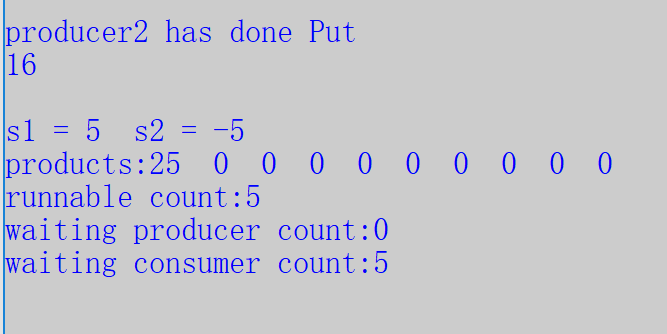
由于调试输出过长，所以本文只挑选部分有说明性的地方给出。

### 5.3.1 执行P操作前后对比



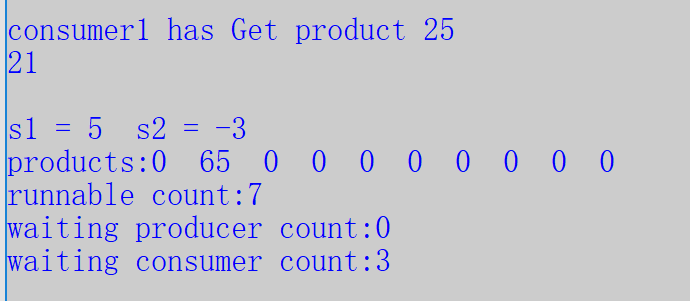
可以看到，连续执行两次P操作之后，s2分别变成-1，-2，而当s2为负数时，s2的绝对值即为等待s2的进程数目，而消费者的等待池中的数目即为等待s2的进程数，可以从下面的输出看到，消费者等待池中进程的数目的确是2个。

### 5.3.2 生产者放入产品



可以看到，producer2将它的产品放入产品缓冲区中，然后缓冲区中的第一个就变成了25，这就是producer2生产的产品。

### 5.3.3 消费者取出产品



可以看到，consumer1得到了产品“25”，然后产品缓冲区中的第一个“25”就变成0了，0表示没有。

## 5.4 实验总结

本次实验使我进一步了解了进程的同步调用机制，以及进程各个状态之间的转换问题。本次实验的控制程序也充当了操作系统的角色，让我对操作系统的职能、作用以及如何控制进程之间的调度问题有了更充分的认识。

# 6 参考文献

[1]郑鹏、曾平、金晶.计算机操作系统[M].第二版.湖北省武汉市：武汉大学出版社，2014年7月：P86-P86

[2]郑鹏、曾平、金晶.计算机操作系统[M].第二版.湖北省武汉市：武汉大学出版社，2014年7月：P114-P114

[3]郑鹏、曾平、金晶.计算机操作系统[M].第二版.湖北省武汉市：武汉大学出版社，2014年7月：P189-P189

[4]作者：超超boy，题目：linux中fork()函数详解，网址：<https://www.cnblogs.com/jycboy/p/l_fork.html> [2019.04.29]

作者：学习记录园，题目：linux中fork()函数详解，网址：

<https://www.cnblogs.com/dongguolei/p/8086346.html> [2019.04.29]

作者：AIDOU\_LBJ，题目：fork()函数，网址：

<https://blog.csdn.net/aidou_lbj/article/details/78012849> [2019.04.29]

作者：菜菜也逆袭，题目：linux如何创建子进程fork和exec函数组详细介绍，网址：

<https://blog.csdn.net/weixin_40921797/article/details/81948010> [2019.04.29]

[5]郑鹏、曾平、金晶.计算机操作系统[M].第二版.湖北省武汉市：武汉大学出版社，2014年7月：P56-P58

[6]郑鹏、曾平、金晶.计算机操作系统[M].第二版.湖北省武汉市：武汉大学出版社，2014年7月：P31-P73