西安工业大学
XI'AN TECHNOLOGICAL UNIVERSITY

# 操作系统课程设计报告

## 题目：银行家算法的设计与实现

院　（系）：计算机科学与工程学院

专　　业：

班　　级：

学　　生：翟晓岩

学　　号：

指导教师：

2010 年 12 月

# 关键路径的算法设计与实现

## 摘 要

Dijkstra 的银行家算法是最有代表性的避免死锁的算法，该算法由于能用于银行系统现金贷款的发放而得名。银行家算法是在确保当前系统安全的前提下推进的。对进程请求先进行安全性检查，来决定资源分配与否，从而确保系统的安全，有效的避免了死锁的发生。

该论文在理解和分析了银行家算法的核心思想以及状态的本质涵义的前提下，对算法的实现在总体上进行了设计，包括在对算法分模块设计，并对各个模块的算法思想通过流程图表示，分块编写代码，并进行测试，最后进行程序的测试，在设计思路上严格按照软件工程的思想执行，确保了设计和实现的可行，可信。代码实现采用 C 语言。

**关键词：**银行家算法；死锁；避免死锁；安全性序列

# 目 录

# 1 绪论

## 1.1 课题背景

在多道程序系统中，虽可以借助多个进程的并发执行来改善系统的资源利用率，提高系统吞吐量，但可能发生一种危险——死锁，即多个进程在运行过程中因争夺资源而造成的一种僵局，若无外力作用，将无法再向前推进。如此，寻求一种避免死锁的方法便显得有为重要。死锁的产生一般的原因有两点：竞争资源和进程间推进顺序非法。因此，我们只需在当前的有限资源下，找到一组合法的执行顺序，便能很好的避免死锁，我们称它为安全序列。而银行家算法起源于银行系统的发放贷款，和计算机操作系统的资源分配完全符合，因此可以借鉴该算法的思想，设计出一种有效的算法程序，解决该问题。

## 1.2 课题意义

（1）运用软件工程的方法指导设计和实现，即是对这学期刚刚学过的软件工程课的复习，又是一次实战演练，从而提高自己的分析问题，解决问题和动手能力；

（2）通过整个算法的设计与实现进一步加深了对算法的理解和多道程序下的计算机系统资源分配现状，为以后进一步的学习打下了良好的基础。

## 1.3 银行家算法

我们可以把操作系统看作是银行家，操作系统管理的资源相当于银行家管理的资金，进程向操作系统请求分配资源相当于用户向银行家贷款。

为保证资金的安全,银行家规定:

(1) 当一个顾客对资金的最大需求量不超过银行家现有的资金时就可接纳该顾客;

(2) 顾客可以分歧贷款,但贷款的总数不能超过最大需求量;

(3) 当银行家现有的资金不能满足顾客尚需的贷款数额时,对顾客的贷款可推迟支付,但总能使顾客在有限的时间里得到贷款;

(4) 当顾客得到所需的全部资金后,一定能在有限的时间里归还所有的资金.

操作系统按照银行家制定的规则为进程分配资源，当进程首次申请资源时，要测试该进程对资源的最大需求量，如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源，否则就推迟分配。当进程在执行中继续申请资源时，先测试该进程已占用的资源数与本次申请的资源数之和是否超过了该进程对资源的最大需求量。若超过则拒绝分配资源，若没有超过则再测试系统现存的资源能否满足该进程尚需的最大资源量，若能满足则按当前的申请量分配资源，否则也要推迟分配。

## 1.4 死锁

死锁是进程死锁的简称，是由 Dijkstra 于 1965 年研究银行家算法时首先提出来的。是指多个进程循环等待它方占有的资源而无限期地僵持下去的局面。很显然，如果没有外力的作用，那麼死锁涉及到的各个进程都将永远处于封锁状态。它是计算机操作系统乃至并发程序设计中最难处理的问题之一。实际上，死锁问题不仅在计算机系统中存在，在我们日常生活中它也广泛存在。

在计算机系统中，涉及软件，硬件资源都可能发生死锁。例如：系统中只有一台 CD-ROM 驱动器和一台打印机，某一个进程占有了 CD-ROM 驱动器，又申请打印机；另一进程占有了打印机，还申请 CD-ROM。结果，两个进程都被阻塞，永远也不能自行解除。

## 1.5 安全性序列

安全序列的的实际意义在于：系统每次进行资源分配后，如果对于系统中新的资源状况，存在一个安全序列，则至少存在一条确保系统不会进入死锁的路径。按照该序列，银行家可以实施一个有效的分配过程使得所有客户得到满足

银行家算法的核心在于安全序列的产生。安全序列正是一种安全的进程推进顺序

# 2　需求分析

## 2.1 问题描述

运用银行家算法，避免死锁的发生。在确保当前系统安全的前提下推进的。对进程请求先进行安全性检查，来决定资源分配与否，从而确保系统的安全，有效的避免了死锁的发生。

问题的关键在于安全性算法，即找安全性序列。

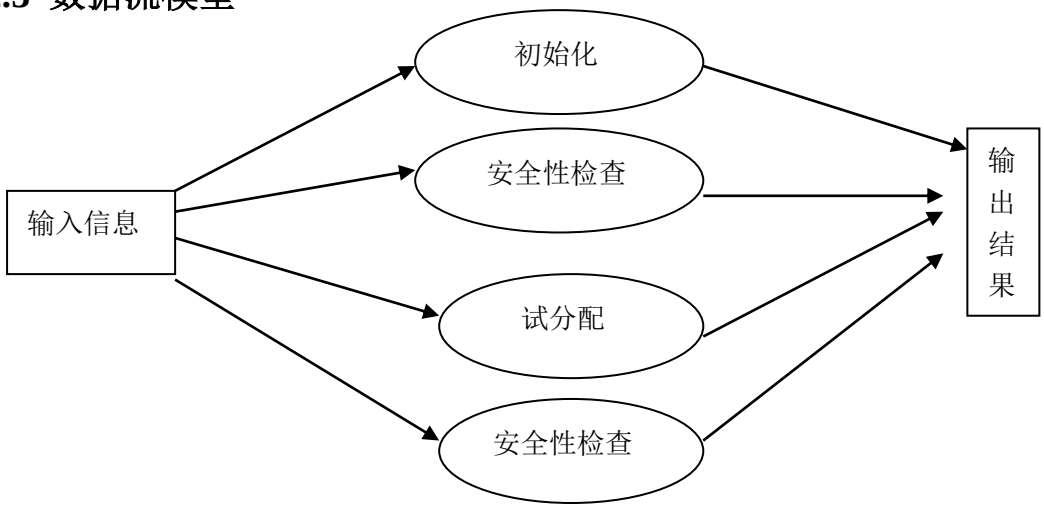
## 2.2 基本要求

（1）从键盘输入当前系统的资源信息，包括当前可用资源，每个进程对各类资源的最大需求量，每个进程当前已分配的各个资源量和每个进程尚需要的各个资源量，输出结果显示在 DOS 界面上；

（2）输入进程请求，按照设计好的安全性算法进行检查，得到结果并输出整个执行过程的相关信息和最终结果（主要包括资源分配表和安全序列）

（3）要求要有各种异常的处理，程序的可控制性和可连续性执行。包括对进程的存在有无检查，请求向量的不合法检查，试分配失败后的数据恢复和重新接受进程请求等。

## 2.3 数据流模型



# 3 概要设计

## 3.1 模块的划分

由于该算法规模较小，所以选用结构化的设计方法，将该系统划为五块，分别是：

⑴主模块，处在整个系统的最高层，负责组织调用其他模块。

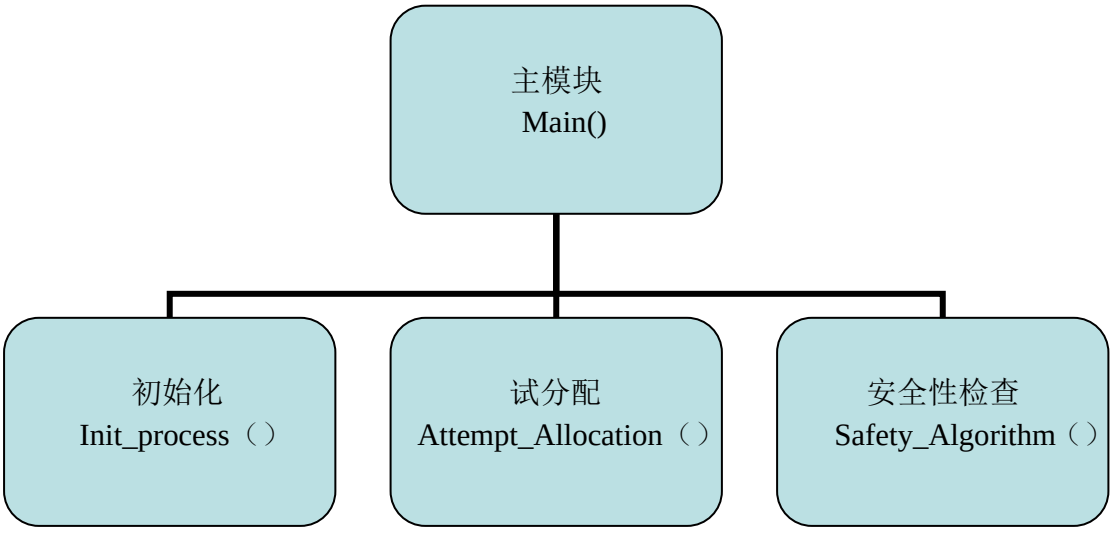
（2）初始化模块，负责从键盘读入系统资源和进程状态，并将系统初识资源分配状态打印。

（3）试分配模块，负责处理进程请求，和相应的数据结构的修改，已经特殊情况的处理。

（4）安全性检查，负责试分配后的安全性检查，以及系统不安全时的资源恢复。

## 3.2 模块调用关系

各模块之间的调用如下图示：



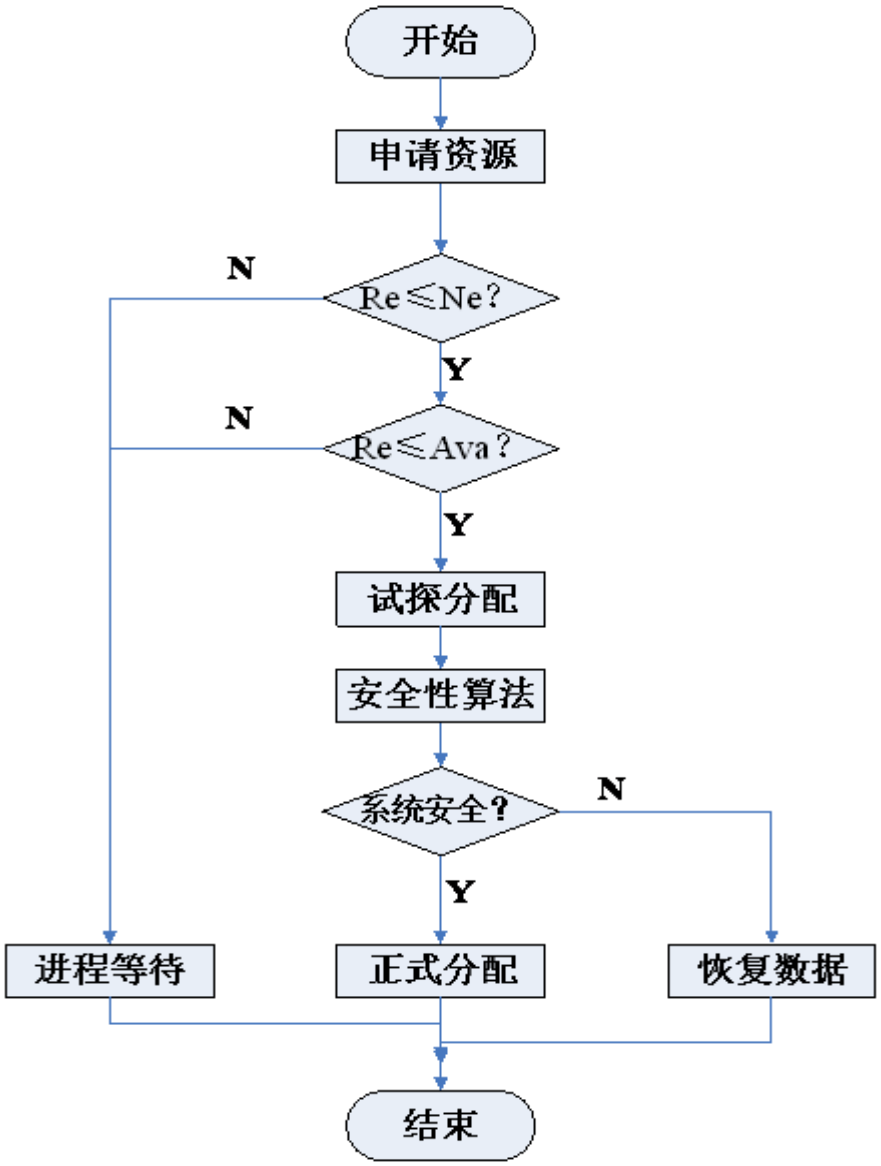

## 3.3 各模块之间的接口

（1） Flag1：试分配模块 Attempt_Allocation 与安全性检查 Safety_Algorithm 之间接口 Attempt_Allocation 通过检查 flag 的真假了判断是否执行。

（2） pro:一个地址， Safety_Algorithm 返回给主模块 main 的信息，不为 NULL 时表示试分配成功，否则系统转入相应异常处理。

（3） Flag2:Safety_Algorithm 与主模块之间的接口，为真则调用打印函数，输出最终结果，否则调用恢复函数，恢复之前系统状态

## 3.4 程序流程图



# 4 详细设计

## 4.1 数据结构选取分析

该算法中用到了较多的数据，基于程序的易实现和较好的结构，决定采用结构链表，以进程为单位（结点）。

## 4.2 数据结构设计

```
typedef struct my_process
{
  int num;       //进程标号
  int Max[M];                 // 表示某个进程对某类资源的最大需求
  int Allocation[M];          // 表示某个进程已分配到某类资源的个数
  int Need[M];                // 表示某个进程尚需要某类资源的个数
  struct my_process* next;  //指向下一个结点（进程）
}process;

  int Available[M]={0};// 其中每一个数组元素表示当前某类资源的可用数
                          目，初始化为系统所提供的资源的最大数目
  int Request[M]={0};  //请求向量
  int Record_work[N][M]={0};//存储当前 work[]的值，以便输出
  int Safety[N]={0};   //存储安全序列，以便后面排序
```

## 4.3 算法整体设计与调用

主函数 void main()主要分四大块：

（1）首先需要初始化 Init_process(process **head,int m,int* count)，存储系统当前状态信息；

（2）调用安全算法 Safety_Algorithm，检测当前系统安全状态，若安全则进行下一步，否则打印相关信息，程序退出。

（3） 调用试分配函数 Attempt_Allocation，进行试分配，若试分配成功，修改相关数据结构，打印当前系统资源分布图，转下一步，否则，打印提示信息，接收其他请求向量。

（4） 再次调用安全性算法，检查试分配以后的系统安全性，若安全打印安全性序列和当前系统资源分布图，并进入新一轮的执行。否则之前的试分配作废，恢复试分配之前的数据结构，输出相关提示信息，接收下一个进程请求

## 4.4 模块设计与时间复杂度分析

4.4.1 系统资源初始化函数 Init_process

(1)首先读入系统可用资源。

(2)用 malloc()函数动态创建进程并且输入相关资源 Max[M]，Allocation[M]，Need[M]。进程之间以链表组织，输入-1 结束初始化。

```
for(i=0;i<m;i++)
{
    scanf("%d",&node.Need[i]);   //初始化资源
}
Insret_Tail(head,node);              //插入链尾
(*count)++;                          //资源个数加 1
```

4.4.2 安全性算法 Safety_Algorithm

找安全序列，其中用到查找当前合法进程的函数 process* Reasonable（）

（1）先初始化相关数据结构

work=(int*)malloc(m*sizeof(int)); //当前系统可用资源

finish=(int*)malloc(n*sizeof(int)); //标记向量，初始为 false，通过安全检查后最为 true

（2） Reasonable(head,finish,work,m,n)函数，找到当前可执行的进程，执行 Alter_Data（）修改当前数据结构。

（3）while(count<n) 重复步骤（2）

4.4.3 接受进程请求，试分配 Attempt_Allocation(head,Request,Available,M);

（1）先判断标志 flag 的值，若为真，则执行 Attempt_Allocation 函数，该 flag 是由 Safety_Algorithm 返回的。

（2）进入 Safety_Algorithm 函数，首先输入进程号，并检查该进程是否存在，存在则输入它的请求向量，并进行检查。

request[i]<=p->Need[i]

request[i]<=avail[i]

若以上两个条件都为真，试分配成功，则执行 Alter_Data（）函数，修改相关数据结构，否则该进程被阻塞，系统转而接受其他请求

avail[i]=avail[i] - request[i];

p->Allocation[i]=p->Allocation[i] + request[i];

p->Need[i]=p->Need[i] - request[i];

将结果打印出来。

4.4.4 对试分配后的系统，进行安全性检查 Safety_Algorithm。

执行过程同 4.4.2 大致一样，唯一一点在于当找不到安全序列时，将本次试分配作废，恢复该次试分配之前的数据结构。

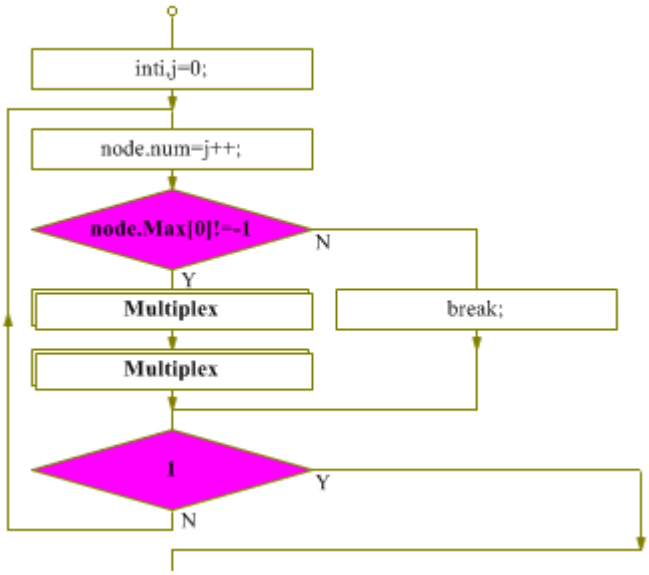
## 4.5 程序流图

4.5.1 系统以及进程资源初始化 Init_process 的程序流程图，如下图 4-1

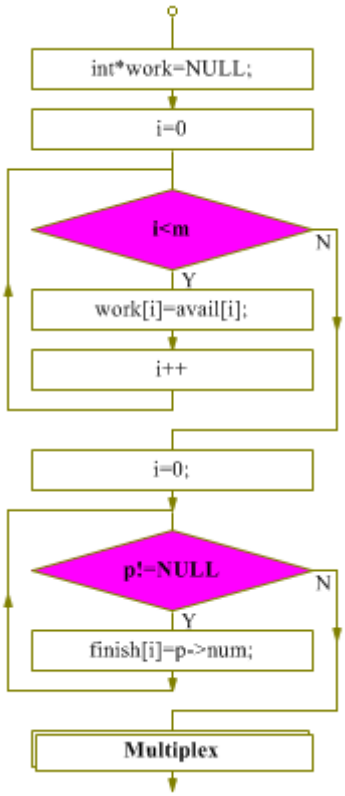
第一步，读入当前系统可用资源；

第二步，读入进程资源，建立进程链表，输入-1 结束初始化；

第三步， 打印当前系统资源分配表。




4-1




4-2

4.5.2 安全性算法 Safety_Algorithm 的程序流程图，如上图 4-2;

第一步，初识化 work,finish[];

第二步，判断异常情况；

第三步，Reasonable(head,finish,work,m,n)函数，找到当前可执行的进程

第四步，执行安全检查，并显示检查后结果，返回给 flag。

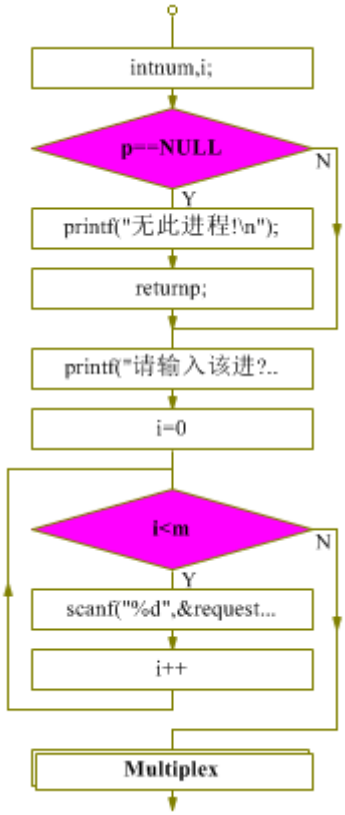
4.5.3 接受进程请求，试分配 Attempt_Allocation 的程序流程图，如下图 4-3

第一步，p 指针指向弧的结点；

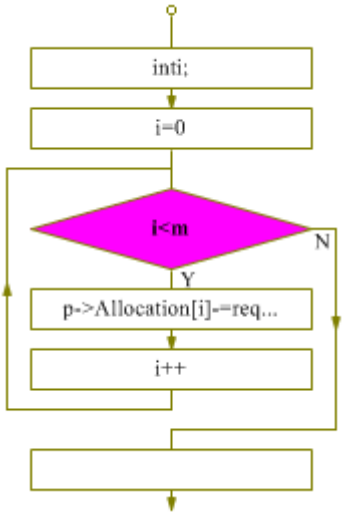
第二步，p 指向邻接表的首节点；

第三步，p 指针下移，直到为空，

4.5.4 对试分配后的系统，进行安全性检查 Safety_Algorithm 的程序流程图，和 4.5.2 大致一样，唯一一点在于当找不到安全序列时，将本次试分配作废，恢复该次试分配之前的数据结构。如下图 4-4




4-3




4-4 回收资源

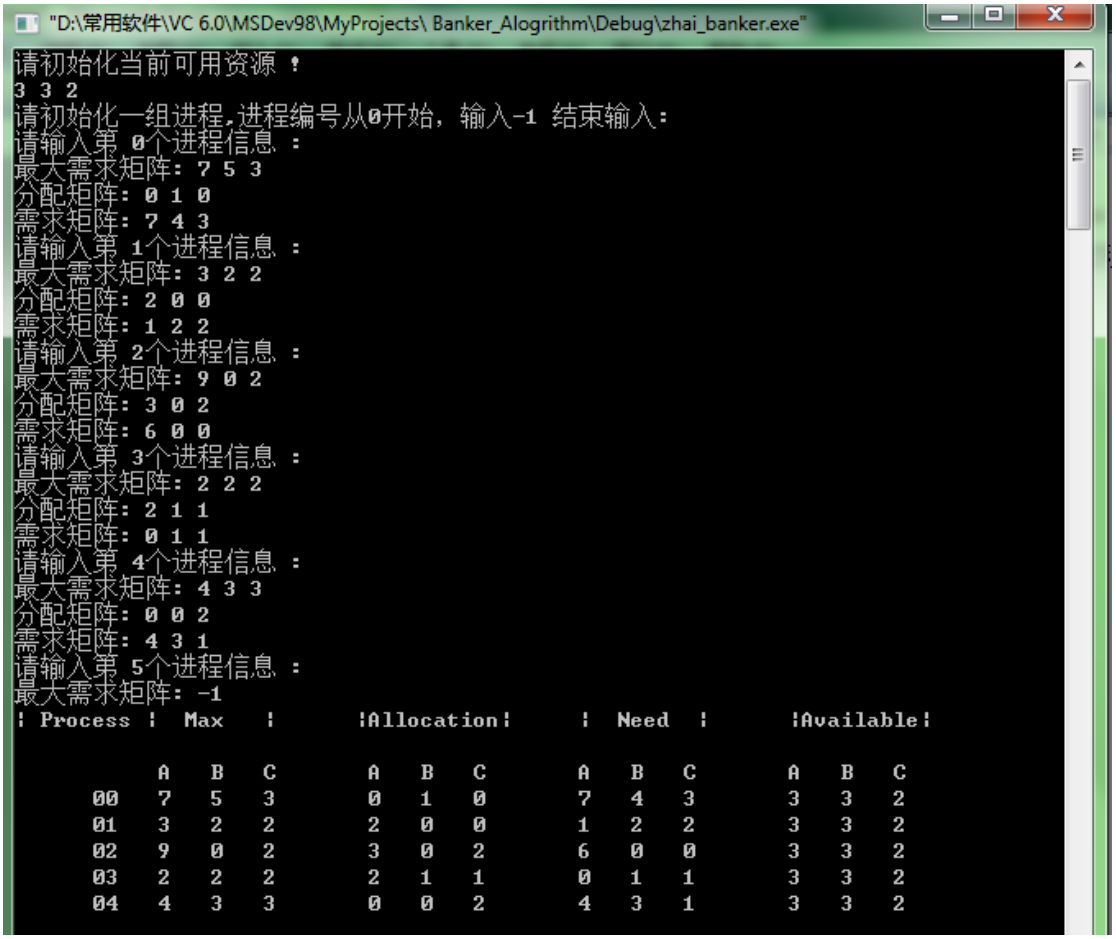
# 5　程序分析测试

## 5.1 分模块分析与测试

函数的书写分模块进行，每完成一个模块进行调试、测试直到该函数运行无误。

5.1.1 初始化系统资源模块 Init_process(process **head,int m,int* count)的测试

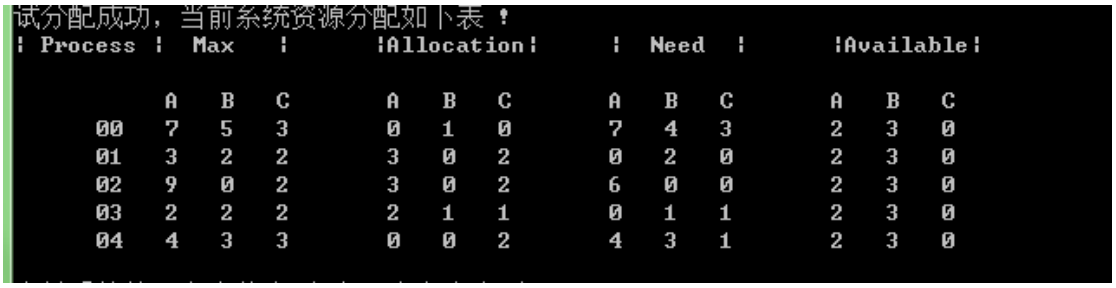
按提示输入，以-1 结束整个初始化过程，并打印结果。




上图的效果起初不理想，经过一些调整后，显示才比较理想。

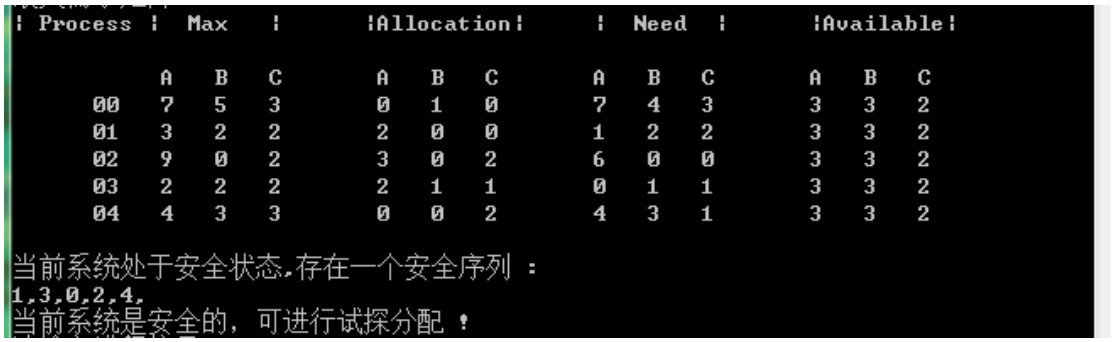
5.1.2 试分配模块 Attempt_Allocation 的测试

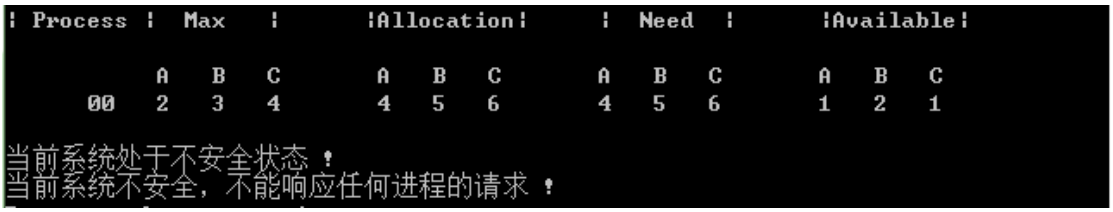
试分配模块，主要是在系统进过第一次安全检查后，对系统资源的一次尝试性分配，试分配完成后，相关的数据结构被修改。



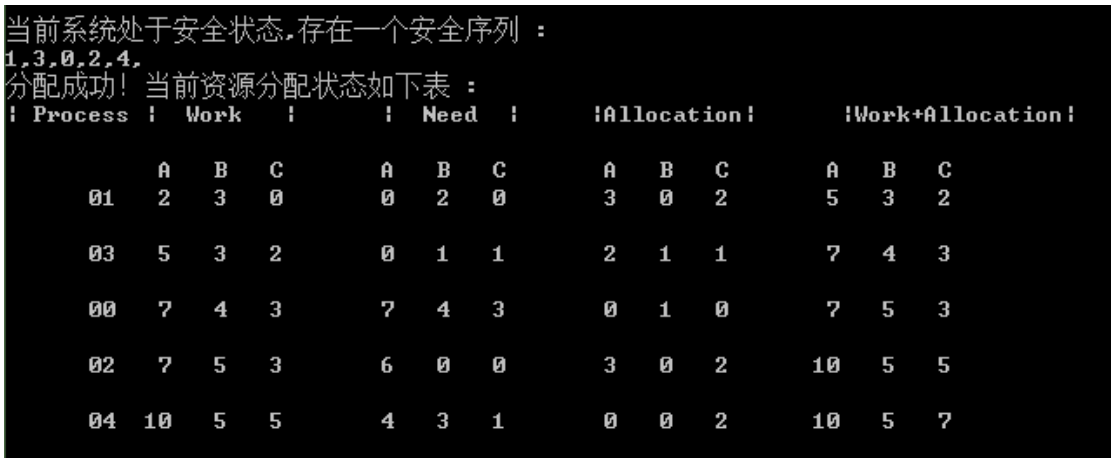

5.1.3 安全模块 Safety_Algorithm 的调试

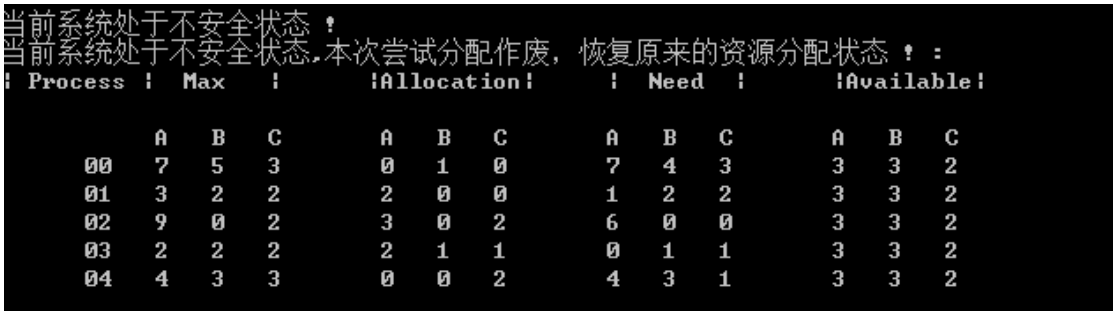
（1）试分配前的安全算法，结果如果输出一个安全性序列，并且经过人工检查该安全性序列，确实有效，则该模块正确；如果系统不安全，打印出相关信息，返回上一层。效果如下图示：




（2）试分配后的安全算法，结果如果输出一个安全性序列，并且经过人工检查该安全性序列，确实有效，则该模块正确；否则，之前的试分配作废，恢复试分配前的资源状态。




## 5.2 集成测试

各模块测试通过后，集成在一起测试，系统初始资源和模块测试时保持一致，以下是一组测设用例以及结果，基本包括了该算法的所有情况。鉴于整个过程的截图较大，所以省略，部分过程可参看上面模块测试截图。

（1）p6: 结果;无此进程

（2）P1: Request(2,0,2) 结果;系统不能满足

（3）P1: Request(1,0,2) 结果;两次安全性检查都通过，并打印出最终结果

（4）P0: Request(0,2,0) 结果;试分配后，系统不安全，试分配作废

（5）P0: Request(0,1,0) 结果; 两次安全性检查都通过，并打印出最终结果

# 6 小结

（1） 在银行家算法的课程设计中，首先对该算法的思想进行分析，大致需要几个模块。

（2） 首先，在网上收集了一些关于银行家算法的资料，包括它的起源，以及在实际中多个领域的应用，加深了对它的理解。之后，确定自己设计的算法分四大模块。首先需要初识化系统资源，其次，安全性检查，再者，试分配，最后是试分配后的安全性检查。

（3） 在程序测试中出现了很多问题，譬如，死循环，逻辑关系的设计不当，还有显示效果不理想等等。但通过查阅书本，对算法细节重新建立正确的认识后，在通过单步调试后，最终解决，界面也几经尝试，才达到理想的界面风格。

（4） 在集成测试中，由于之前的模块测试做的比较扎实，所以相对只是一些细节上的问题，很快也达到了预期的效果。

# 参考文献

[1] 严蔚敏 吴伟民，.数据结构(C 语言版)，1999，清华大学出版社；

[2] 汤小丹 梁红兵 哲凤屏 汤子赢， 计算机操作系统（第三版）西安电子科技大学出版社；

# 附录（源代码）

```
//银行家算法
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define M 3 //  系统资源的种类
#define N 10 //  进程上限
#define D12 %5d%5d%5d%5d%5d%5d%5d%5d%5d%5d%5d%5d%5d    //输入输出格式定义
typedef struct my_process
{
   int num;            //进程标号
   int Max[M];                    //  表示某个进程对某类资源的最大需求
   int Allocation[M];             //  表示某个进程已分配到某类资源的个数
   int Need[M];                   //  表示某个进程尚需要某类资源的个数
   struct my_process* next;
}process;

void Insret_Tail(process **head,process node)       //尾插法建立进程链表
{
     process* p=(process*)malloc(sizeof(process));
     process* last=NULL;
     memcpy(p,&node,sizeof(process));         //动态创建进程结点
     p->next=NULL;
     if(NULL==*head)
     {
          *head=p;
     }
     else          //找表尾
     {
          last=*head;
          while(last->next!=NULL)
```

```
        {
            last=last->next;
        }
        last->next=p;                                   //插入链尾
    }

}
void Init_process(process **head,int m,int* count)        //初始化系统资源
{
    int i,j=0;
    process node;
    printf("请初始化一组进程,进程编号从 0 开始，输入-1  结束输入:\n");
    do
    {
        node.num=j++;
        printf("请输入第  %d 个进程信息  :\n",node.num);
        printf("最大需求矩阵: ");
        scanf("%d",&node.Max[0]);
        if(node.Max[0]!=-1)                    //输入-1  结束输入
        {
            for(i=1;i<m;i++)
            {
                scanf("%d",&node.Max[i]);
            }

            printf("分配矩阵: ");
            for(i=0;i<m;i++)
            {
                scanf("%d",&node.Allocation[i]);
            }
            printf("需求矩阵: ");
            for(i=0;i<m;i++)
            {
                scanf("%d",&node.Need[i]);
```

```
            }
            Insret_Tail(head,node);                    //插入链尾
            (*count)++;                                //进程数加 1
        }
        else
        {
            break;
        }
    }
    while(1);
}

void Print(process *head,int *avail,int m)          //打印初识系统资源
{
    process* p=NULL;
    int i,j;
    char ch;
    p=head;
    if(NULL==p)
    {
        printf("当前无进程 !\n");
        exit(0);
    }
    else
    {
        printf("| Process |    Max     |          |Allocation|         |    Need    |
|Available|\n\n");
        printf("\t");
        for(i=0;i<4;i++)
        {
            ch='A';
            for(j=0;j<m;j++)
            {
                printf("%4c",ch++);
```

```
            }
            printf("\t");
        }
        puts("");
            while(p!=NULL)
            {
                printf("%8.2d",p->num);
                for(j=0;j<m;j++)
                {
                    printf("%4d",p->Max[j]);
                }
                printf("\t");
                for(j=0;j<m;j++)
                {
                    printf("%4d",p->Allocation[j]);
                }
                printf("\t");
                for(j=0;j<m;j++)
                {
                    printf("%4d",p->Need[j]);
                }
                printf("\t");
                for(j=0;j<m;j++)
                {
                    printf("%4d",avail[j]);
                }
                printf("\n");
                p=p->next;
            }
            puts("");
        }
}
```

process* Location(process* head,int pro_num) //进程定位函数，找到当前请求进程在进程链表中的位置，以便后面对其操作

```
{
    process *p=NULL;
    p=head;
    if(NULL==p)
    {
        printf("error !\n");                    //异常，当前链表为空
        return p;
    }
    else
    {
        while(p!=NULL)
        {
            if(p->num==pro_num)
            {
                break;
            }
            else
            {
                p=p->next;
            }
        }
        if(NULL==p)                             //无此进程，输入错误
        {
            printf("无此进程 !\n");
            return p;
        }
        else
        {
            return p;
        }
    }
}
process* Attempt_Allocation(process* head,int *request,int *avail,int m)
{   //试分配
```

```
int num,i;
process* p=NULL;
printf("请输入进程编号: \n");
scanf("%d",&num);
p=Location(head,num);
if(p==NULL)
{
    printf("无此进程!\n");
    return p;
}
printf("请输入该进程的请求向量: \n");
for(i=0;i<m;i++)
{
    scanf("%d",&request[i]);
}

for(i=0;i<m;i++)      //请求的合法性检验
{
    if(request[i]<=p->Need[i])
    {
        ;
    }
    else
    {
        printf("该请求系统不能满足 !\n");
        return NULL;
    }
}
for(i=0;i<m;i++)
{
    if(request[i]<=avail[i])
    {
        ;
```

```
            }
            else
            {
                printf("该请求系统不能满足 !\n");
                return NULL;
            }
        }
        for(i=0;i<m;i++)                        //试分配，修改相关数据结构
        {
            avail[i]=avail[i] - request[i];
            p->Allocation[i]=p->Allocation[i] + request[i];
            p->Need[i]=p->Need[i] - request[i];
        }
    return p;
    }

    process* Reasonable(process* head,int*finish,int*work,int m,int n)        //找当前可执行的进程
    {
        int i=0,j=0,count=0;
        process* p=NULL;
        while(1)
        {
            if(finish[i]!=-1)   //表示该进程未执行安全性检查
            {
                p=Location(head,finish[i]);          //定位该进程
                if(p!=NULL)
                {
                    for(j=0;j<m;j++)
                    {
                        if(p->Need[j]>work[j])
                        {

                            break;
```

```
                    }
                    else
                    {
                        continue;
                    }
                }
                if(j==m)
                {
                    return p;
                }
                else
                {
                    i++;      //当前进程检查没有通过，则进行下一个进程的检查
                }
            }
        }
        else
        {
            i++;    //当前进程已经检查过，则进行下一个进程的检查
        }
        if(i==n)
        {
            return NULL;   //遍历所有进程都未找到，则跳出返回 NULL
        }
    }
}
void Alter_Data(process* p,int* work,int* finish,int record[][M],int m)   //修改相关数据结构
{
    int i;
    for(i=0;i<m;i++)
    {
        record[p->num][i]=work[i];
        work[i]=work[i]+p->Allocation[i];
```

```
    }
    finish[p->num]=-1;     //表示该进程已通过安全性检查
}
int Safety_Algorithm(process* head,int *avail,int *safety,int Record[][M],int m,int
n)//安全性算法
{
    int *work=NULL;
    int *finish=NULL;
    process *p=NULL;
    process *pro=NULL;
    int i,count=0;

    work=(int*)malloc(m*sizeof(int));       //当前系统可供给进程的各个资源数目
    finish=(int*)malloc(n*sizeof(int));     //标志数组
    //safety=(int*)malloc(n*sizeof(int));
    p=head;
    for(i=0;i<m;i++)
    {
        work[i]=avail[i];
    }
    i=0;
    while(p!=NULL)
    {

        finish[i]=p->num;
        p=p->next;
        i++;
    }
    i=0;
    while(count<n)
    {
        pro=Reasonable(head,finish,work,m,n);
        if(pro!=NULL)                           //Need[i,j]<=work[j],即当前系统
```

可以满足该进程

```
        {
            Alter_Data(pro,work,finish,Record,m);
            count++;
            safety[i]=pro->num;    //
            i++;
        }
        else
        {
            printf("当前系统处于不安全状态 !\n");
        //  remark=1;
            break;
        }

    }
    if(count==n)
    {
        printf("当前系统处于安全状态,存在一个安全序列 :\n");
        for(i=0;i<n;i++)
        {
            printf("%d,",safety[i]);        //打印安全序列
        }
        puts("");
    }

    free(finish);
    free(work);
    finish=NULL;
    work=NULL;
    if(count==n)
    {
        return 1;     //找到安全序列则返回 1
    }
    else
```

```
    {
        return 0;//未找到安全序列则返回 0
    }
}
void Return_Source(process* p,int *request,int *avail,int m)  //若试分配失败，则恢复试分配前的资源状态
{
    int i;
    {
        for(i=0;i<m;i++)
        {
            p->Allocation[i]-=request[i];
            p->Need[i]+=request[i];
            avail[i]+=request[i];
        }
    }
}
void Print_After_Safety(process *head,int *safety,int work[][M],int m,int n)  //打印试分配后的系统资源状态
{
    process* p=NULL;
    int i,j;
    char ch;

    p=head;
    if(NULL==p)
    {
        exit(0);
    }
    else
    {
        printf("| Process |  Work  |  |  Need  |  |Allocation| |Work+Allocation|  |  Finish  |\n\n");
        printf("\t");
```

```
for(i=0;i<4;i++)
{
    ch='A';
    for(j=0;j<m;j++)
    {
        printf("%4c",ch++);
    }
    printf("\t");
}
puts("");

for(i=0;i<n;i++)
    {
        p=head;
        while(p!=NULL)
        {
            if(p->num==safety[i])
            {   printf("%8.2d",p->num);

                for(j=0;j<m;j++)
                {
                    printf("%4d",work[p->num][j]);
                }
                printf("\t");
                for(j=0;j<m;j++)
                {
                    printf("%4d",p->Need[j]);
                }
                printf("\t");
                for(j=0;j<m;j++)
                {
                    printf("%4d",p->Allocation[j]);
                }
                printf("\t");
```

```
                    for(j=0;j<m;j++)
                    {
                        printf("%4d",work[p->num][j]+p->Allocation[j]);
                    }
                    printf("\n");
                    break;
                }
                else
                {
                    p=p->next;
                }
            }
            puts("");

        }
    }
}
void main()
{
    int i,flag=0,count=0;
    char ch;
    int *p=NULL;        //进程链头指针数组
    int Available[M]={0};            // 其中每一个数组元素表示当前某类资源的可用数目，初始化为系统所提供的资源的最大数目
    int Request[M]={0};
    int Record_work[N][M]={0};
    int Safety[N]={0};    //存储安全序列，以便后面排序
    process *head=NULL;
    process *pro=NULL;
    p=&count;
    printf("请初始化当前可用资源 !\n");
    for(i=0;i<M;i++)
    {
        scanf("%d",&Available[i]);
```

```
    }
    Init_process(&head,M,p);//初识化系统资源
    Print(head,Available,M);       //打印初始化资源状况
    do
    {
        flag=Safety_Algorithm(head,Available,Safety,Record_work,M,count); //安全性检验
        if(1==flag)
        {
            printf("当前系统是安全的，可进行试探分配 !\n");
            do
            {
                pro=Attempt_Allocation(head,Request,Available,M);          //通过则试分配
                if(NULL!=pro)
                {
                    printf("试分配成功，当前系统资源分配如下表 !\n");
                    Print(head,Available,M);
                    break;
                }
                else                                  // 否则，退到上一级
                {
                    printf("当前请求系统不能满足 ！您可以选择重新输入请求向量(Y/y)，或者退出(N/n)\n\n");
                    printf("您是否要继续操作  (Y/N):\n");
                    getchar();
                    scanf("%c",&ch);
                    if(ch=='N'|| ch=='n')
                    {
                        exit(0);
                    }

                }
            }while(ch=='Y'||ch=='y');
```

```
        }
        else            //未通过安全性算法
        {
            printf("当前系统不安全，不能响应任何进程的请求 !\n");
            exit(0);
        }

        flag=Safety_Algorithm(head,Available,Safety,Record_work,M,count);
        if(1==flag)
        {
            printf("分配成功！当前资源分配状态如下表 :\n");
            Print_After_Safety(head,Safety,Record_work,M,count);
            printf("您是否还要继续操作 (Y(y)/N(y)\n)");
            getchar();
            scanf("%c",&ch);
        }
        else
        {
            printf("当前系统处于不安全状态,本次尝试分配作废，恢复原来的资
源分配状态 ！:\n");
            Return_Source(pro,Request,Available,M);
            Print(head,Available,M);
            printf("您是否还要继续操作 (Y(y)/N(y)\n)");
            getchar();
            scanf("%c",&ch);
        }
    }while(ch=='Y'|| ch=='y');
    printf("您已经正常退出!\n\n");
}
```