操作系统课程设计

银行家算法

1．目的和要求

在熟练掌握死锁发生原理和解决死锁问题的基础上，利用一种程序设计语言模拟实现利用银行家算法实现死锁避免，一方面加深对原理的理解，另一方面提高学生通过编程根据已有原理解决实际问题的能力，为学生将来进行系统软件开发和针对实际问题提出高效的软件解决方案打下基础。

2．实验内容

模拟实现银行家算法实现死锁避免。要求：初始数据（如系统在T0时刻的资源分配情况、每一种资源的总数量）从文本文件读入，文件中给出最大需求矩阵Cliam、分配矩阵Allocation，在程序中求得需求矩阵Requset和可利用资源向量Available。

3. 程序逻辑伪代码

3.1 数据结构

Resource $= \mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available $= \mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim $= \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation $= \mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

3.2 逻辑伪代码

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) Global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     <error>;                          /* total request > claim*/
else if (request [*] > available [*])
     <suspend process>;
else  {                                      /* simulate alloc */
     <define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*]>;
}
if (safe (newstate))
     <carry out allocation>;
else {
     <restore original state>;
     <suspend process>;
}
```

(b) Resource alloc algorithm

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
           claim [k,*] - alloc [k,*]<= currentavail;
        if (found) {                 /* simulate execution of Pk */
           currentavail = currentavail + alloc [k,*];
           rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) Test for safety algorithm (banker's algorithm)

4. 程序实现

4.1 文本数据的导入

本例中准备了h1和h2两个文本文件。其中依次存放了如下信息。

Resource vector R: 9 3 6
P2 : 2
Request: 1 0 1

Claim matrix C:	Allocation matrix A:
3 2 2	1 0 0
6 1 3	5 1 1
3 1 4	2 1 1
4 2 2	0 0 2

导入程序：见附录 函数input(); 其中,由输入的数据计算出了Avaliable vector V

4.2 数据预处理

1.判断线程P2的request是否超出了原定的Cliam，若超出则违反条件挂起线程
2.判断当前的Available是否满足request，若不是就挂起线程。
3.若可以分配则尝试分配，并用函数safe();检验当前状态是否为安全态。
4.若非安全状态则回滚到上一状态，挂起线程。

代码片段：

```
if(safe()){
   printf("\nThe request is allowed.\tThere is a new state.\n");
 }
 else{  //rollback
    for(int i=0;i<M;i++){
     alloc[p][i]-=request[i];
     available[i]+=request[i];
   }
   printf("\nROLLBACK:\tProcess %d is suspend\n",p+1);
 }
```

5.结果分析

5.1 情形一：

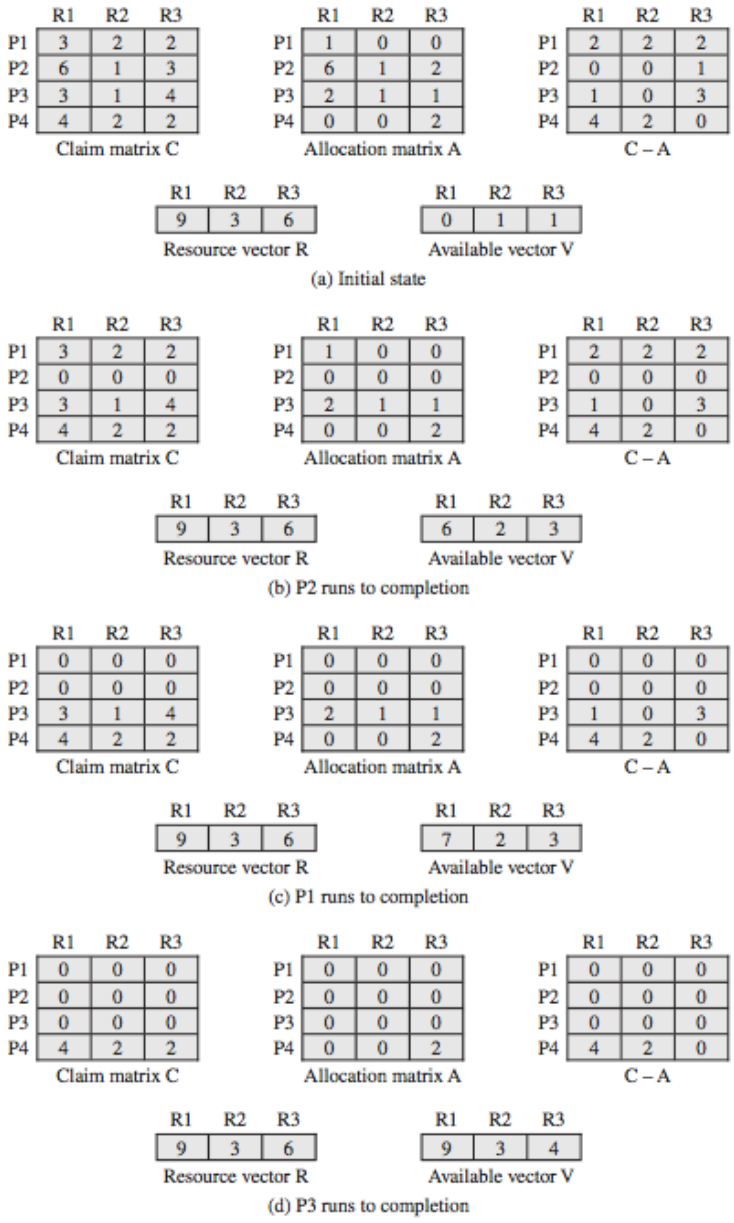


Figure 6.7 Determination of a Safe State

此时的输入数据为h1，得到如上图的状态，已知该状态为安全状态。以下是程序结果：

```
sgd:bank mac$ ./a.out <h1

the Available matrix :
0 1 1

the request matrix :
1 0 1

the Alloc matrix :
1 0 0
6 1 2
2 1 1
0 0 2


the Claim matrix :
3 2 2
6 1 3
3 1 4
4 2 2

is safe : 1

The request is allowed. There is a new state.
```

此时的输入数据为h1，得到如上

5.2 情形二：

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

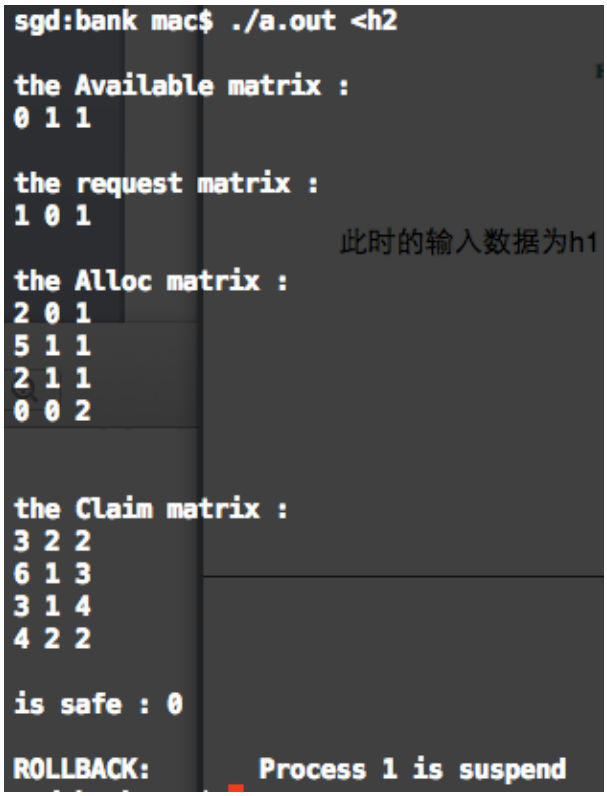
R1	R2	R3
0	1	1

Available vector V

(b) P1 requests one unit each of R1 and R3

Figure 6.8 Determination of an Unsafe State

此时的输入数据为h2，得到如上图的状态，已知该状态为非安全状态。以下是程序结果：



可变分区内存管理

1．目的和要求

在熟练掌握计算机分区存储管理方式的原理的基础上，利用一种程序设计语言模拟实现操作系统的可变分区存储管理的功能，一方面加深对原理的理解，另一方面提高学生通过编程根据已有原理解决实际问题的能力，为学生将来进行系统软件开发和针对实际问题提出高效的软件解决方案打下基础。

2．实验内容

设计合理的数据结构来描述存储空间：对于未分配出去的部分，可以用空闲分区队列来描述，对于已经分配出去的部分，由装入内存的作业占据，可以将作业组织成链表或数组实现分区存储管理的内存分配功能，要求选择至少两种适应算法（如首次适应算法，循环首次适应算法，最佳适应算法，最坏适应算法）。
实现分区存储管理的内存回收算法：要求能够正确处理回收分区与空闲分区的四种邻接关系当碎片产生时，能够进行碎片的拼接。

3. 程序主要模块

3.0 数据结构基础

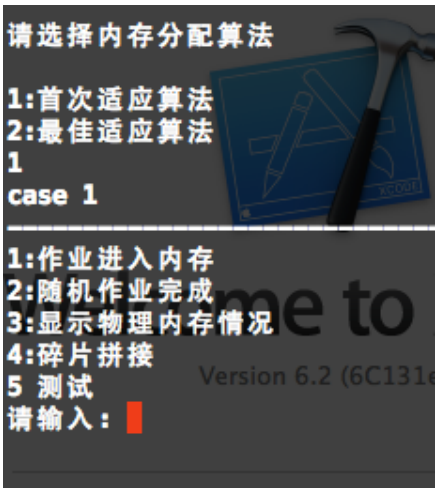
本程序一共维护两个单项链表，以使用内存作业链表 USED 和未使用内存链表 UNUSED。选择动态链表好处是方便的插入与删除操作。定义了如下数据结构

```
typedef struct Node
{
    int start;              ————————————-内存块的初始地址
    int big;                ————————————-内存块的大小
    int end;                ————————————-内存块的终止地址（包含）
    struct Node *next;      ————————————-指向下一个内存块

}Node;
```

每加入一个作业就开辟一个新内存块，并对两个链表所有的节点做相应维护

3.1 菜单模块



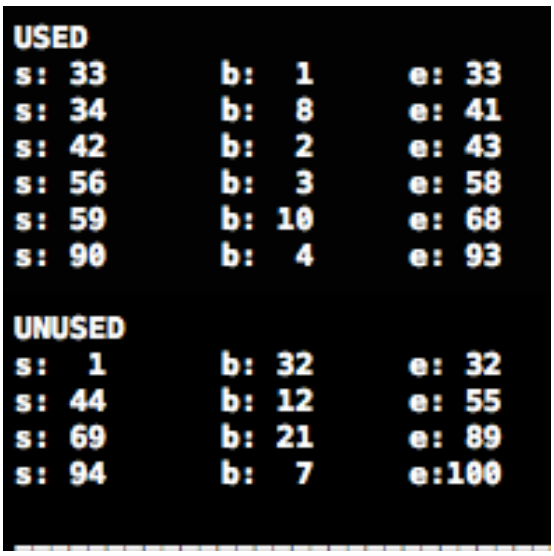
本模块由两个 switch语句实习了两级选择菜单，程序会执行相应定义的功能。对于内存分配算法的选择，在此使用了函数指针的小技巧。

3.2 内存分配模块

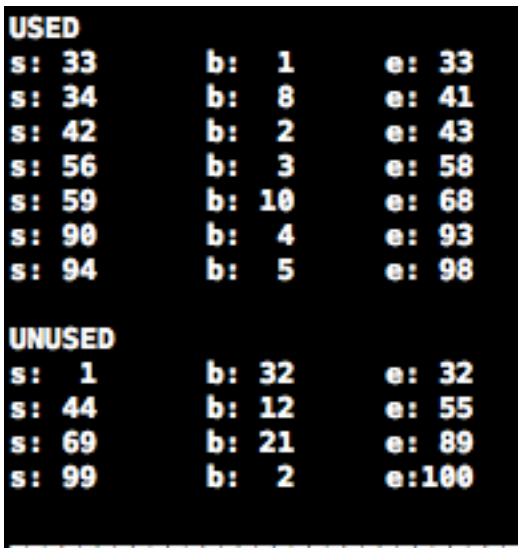
此模块共实现了两个内存分配算法

```
//首次适应函数  ////////////////////////
void join_first();

//最佳适应函数  ////////////////////////
void join_best();
```



分配前状态



分配后状态

上述两幅运行结果图，说明最新分配的工作内存 大小为5，选择了最优化的第四块UNUSEDN内存，我们的最佳适应函数是行之有效的

3.3 作业释放模块

1. 从USED链表中随机选择一个节点释放

2. 将释放的节点加入UNUSED链表中

3. 使用mk_clean();函数对未分配链表做内存整合操作

```
//对未分配链 unuse_head做合并清理的工作
void mk_claen(){
   Node* tp=unused_head;
   while(tp->next){
      //清理
      if(tp->next->big==0){
         Node* fre_p=tp->next;
         tp->next=fre_p->next;
         free(fre_p);
      }
      if(tp->next==NULL){

      }else{
        tp=tp->next;
      }
   }
   tp=unused_head;
   while(tp->next){
      //合并
      if( (tp->end+1)==tp->next->start){
```

```
            Node* fre_p=tp->next;
            tp->big+=fre_p->big;
            tp->end=fre_p->end;
            tp->next=fre_p->next;
            free(fre_p);
        }
        if(tp->next==NULL){

        }else{
            tp=tp->next;
        }
    }
}
```

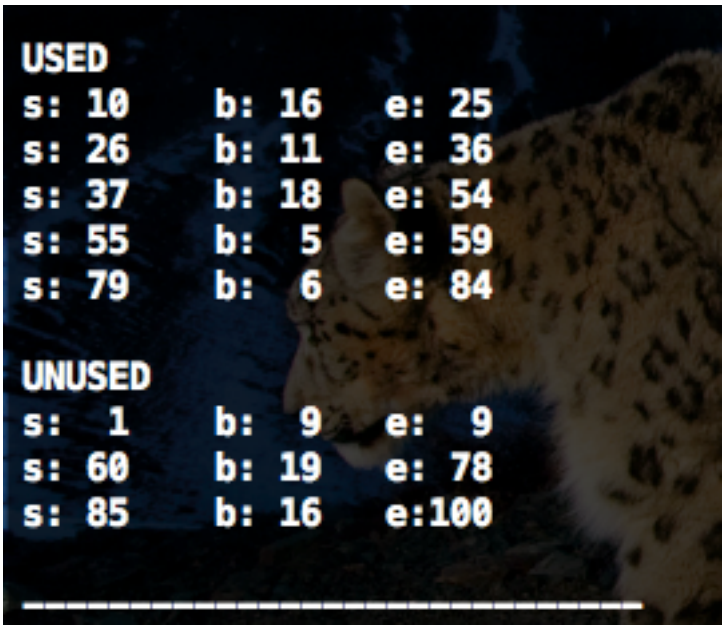
3.4 碎片拼接模块

使用connected();函数,将所有已分配空间整合成连续的内存空间，对于未分配内存则留下一个单独的较大内存块。

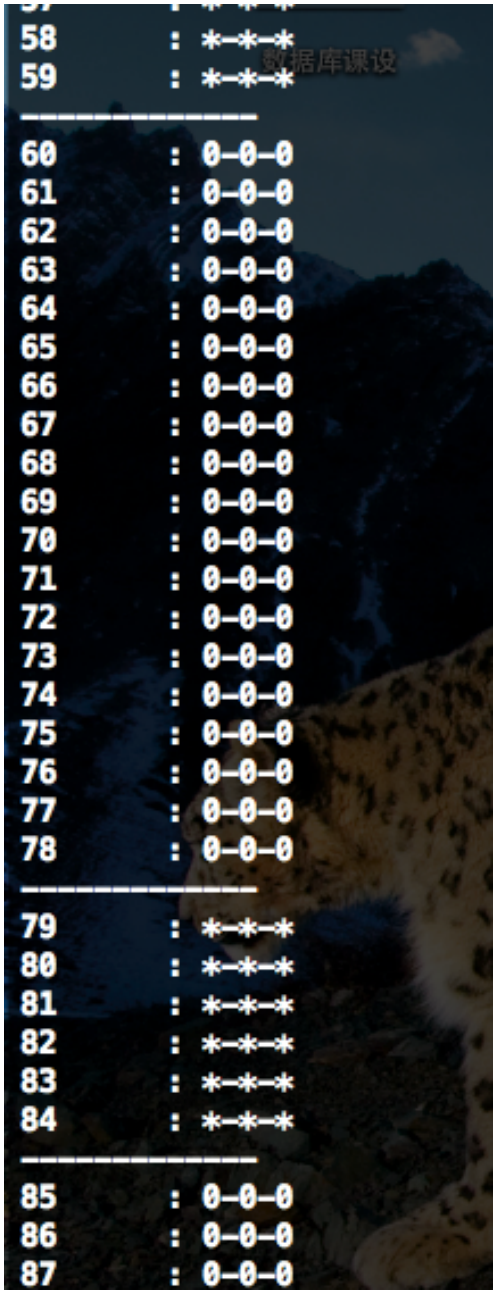
3.5 内存显示模块

两中显示模式：图形化显示模式和字符型显示模式

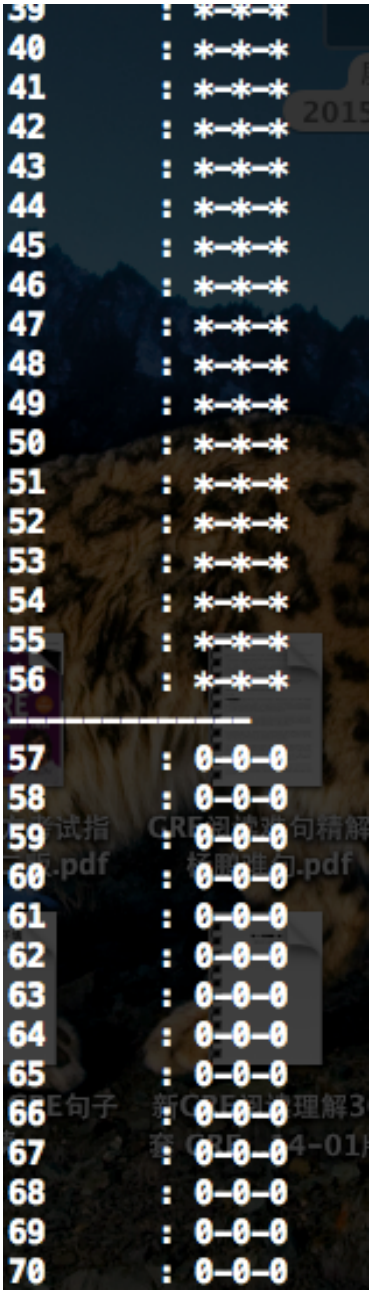
第二种更为直接，一下列举几幅结果图：



两支 链表结果输出



碎片未清理前



碎片清理后

附录

银行家算法:

```
/*
* banker algorithm
* author :ssd227
* date: 2015.7.23
*/
#include<string.h>
#include<stdio.h>

#define N 4
#define M 3

//global variable
//*******************
int resource[M];
int available[M];
int request[M];
int p;
int claim[N][M];
int alloc[N][M];
int need[N][M];

//*******************
void input();
int safe();

//*******************
int main(){
  input();

  p--;
  int err_flag_1=0;
  for(int i=0;i<M;i++){
    if((alloc[p][i]+request[i])>claim[p][i]){
      err_flag_1=1;
      break;
    }
  }
  int err_flag_2=0;
  for(int i=0;i<M;i++){
    if(request[i]>available[i]){
      err_flag_2=1;
      break;
    }
  }


  /*  for(int i=0;i<3;i++)
    printf("%d ",available[i]);

  printf("\n");
  for(int i=0;i<4;i++){
```

```
  for(int j=0;j<3;j++){
    // printf("%d ",alloc[i][j]);
    printf("%d ",claim[i][j]);
  }
  printf("\n");
}
// printf("\nis safe : %d",safe());
*/

if(err_flag_1){
  printf("ERROR:What you need is more than what you claim.");
  printf("That is ridiculous!!\n");
  return 1;
}
else if(err_flag_2){
  printf("Your request is not enough now.\n");
  printf("Process %d is suspend\n",p+1);
  return 1;
}
else{    //pudate a new state
  for(int i=0;i<M;i++){
    alloc[p][i]+=request[i];
    available[i]-=request[i];
  }
}

//here is a big introduction ////////////////////
  printf("\nthe Available matrix :\n");
for(int i=0;i<3;i++)
  printf("%d ",available[i]);

  printf("\n\nthe request matrix :\n");
  for(int i=0;i<3;i++)
      printf("%d ",request[i]);

printf("\n\nthe Alloc matrix :\n");
for(int i=0;i<4;i++){
  for(int j=0;j<3;j++){
    printf("%d ",alloc[i][j]);
    //printf("%d ",claim[i][j]);
  }
  printf("\n");
}

  printf("\n\nthe Claim matrix :\n");
  for(int i=0;i<4;i++){
      for(int j=0;j<3;j++){
          printf("%d ",claim[i][j]);
      }
      printf("\n");
  }
printf("\nis safe : %d\n",safe());
////////////////////////////////////////////////


if(safe()){
```

```
    printf("\nThe request is allowed.\tThere is a new state.\n");
  }
  else{  //rollback
     for(int i=0;i<M;i++){
      alloc[p][i]-=request[i];
      available[i]+=request[i];
    }
    printf("\nROLLBACK:\tProcess %d is suspend\n",p+1);
  }

  return 0;
}//End Main


//  judge the current state  ////////////////
int safe(){
  int currentavail[M];
  memcpy(&currentavail[0],&available[0],sizeof(int)*M);

  int ProcessIsDown[N]={1,1,1,1};
  int possible =1;
  while (possible){
    int flag_found=0;
    int pk;
    for(int i=0;i<N;i++){
      if(ProcessIsDown[i]){
            int flag=1;
            for(int j=0;j<M;j++){
              if((claim[i][j]-alloc[i][j])>currentavail[j]){
                flag=0;
                break;
              }
            }
            if(flag==1){
              flag_found=1;
              pk=i;
              ProcessIsDown[pk]=0;
              break;
            }
      }
    }//End : outer for
    if(flag_found){
      for(int j=0;j<M;j++){
            currentavail[j]+=alloc[pk][j];
      }
    }
    else{
      possible=0;
    }
  }//End :while

  int is_null=0;  //or is safe
  for(int i=0;i<N;i++){
    if(ProcessIsDown[i]){
      is_null++;
    }
```

```
  }

  return is_null==0? 1:0;

}//End :fun safe


//  scanf the nessary data ////////////////////
void input(){

  for(int i=0;i<M;i++){
    scanf("%d",&resource[i]);
  }
  scanf("%d",&p);
  for(int i=0;i<M;i++){
    scanf("%d",&request[i]);
  }
  for(int i=0;i<N;i++){
    for(int j=0;j<M;j++){
      scanf("%d",&claim[i][j]);
      // printf("%d",claim[i][j]);
    }
    //printf("\n");
  }
  for(int i=0;i<N;i++){
    for(int j=0;j<M;j++){
      scanf("%d",&alloc[i][j]);
      //printf("%d",alloc[i][j]);
    }
    //printf("\n");
  }

  //compute the available
  for(int j=0;j<M;j++){
    available[j]=resource[j];
    for(int i=0;i<N;i++){
      available[j]-=alloc[i][j];
    }
  }

}
```

可变分区内存管理

```
//
//  main.c
//  memory
//
//  Created by ssd227 on 15-7-24.
//  Copyright (c) 2015年 ssd227. All rights reserved.
//
#include<stdlib.h>
#include <stdio.h>
#include <time.h>
```

```
typedef struct Node
{
    int start;
    int big;
    int end;
    struct Node *next;

}Node;

//*** global variable  *****
Node* used_head;
Node* unused_head;

//*** fun delcration   *****
void initial();
void join_first();
void join_best();
int creater_random(int a,int b);
void mk_claen();

void work_comp02();
void show();
void insert(Node* T,Node* one);
void connected();
void test();

// START AT HERE////////////////////////////
//*** main  ***************
int main(int argc, const char * argv[]) {

    initial();
    printf("\n");
    printf("请选择内存分配算法\n\n");
    printf("1:首次适应算法\n");
    printf("2:最佳适应算法\n");
    //function pointer;
    void (*join)();

    char input_1;
    input_1=getchar();
    //putchar(input_1);

    switch (input_1) {
        case '1':
            join=join_first;
            printf("case 1");
            break;

        case '2':
            join=join_best;
```

```
        printf("case 2\n");
        break;

    default:
        printf("what you input is wrong\n");
        return 1;
        break;
}


//the core operation
while(1){
    printf("\n----------------------------\n");
    printf("1:作业进入内存\n");
    printf("2:随机作业完成\n");
    printf("3:显示物理内存情况\n");
    printf("4:碎片拼接\n");
    printf("5 测试\n");
    printf("请输入:\t");
    getchar();

    char input_2;
    input_2=getchar();
    //putchar(input_2);

    switch (input_2) {
        case '1':
            join();
            break;

        case '2':
            work_comp02();
            break;

        case '3':
            show();
            break;

        case '4':
            connected();
            break;

        case '5':
            test();
            break;

        default:
            printf("what you input is wrong\n");
            break;
    }
```

```
    }


    return 0;
}

//alternative choice of good result ////////////////////
void test(){
    printf("\nUSED\n");
    Node* p=used_head;
    while (p->next) {
        p=p->next;
        printf("s:%3d \t b:%3d \t e:%3d\n",p->start,p->big,p->end);
    }
    printf("\nUNUSED\n");
    p=unused_head;
    while (p->next) {
        p=p->next;
        printf("s:%3d \t b:%3d \t e:%3d\n",p->start,p->big,p->end);
    }
}

//
void initial(){

    used_head=(Node*)malloc(sizeof(Node));
    unused_head=(Node*)malloc(sizeof(Node));

    unused_head->start=-20;
    unused_head->end=-10;
    unused_head->next=NULL;
    used_head->next=NULL;


    //in the begining ,the used is null and the unused is 1-100
    Node* p=(Node*)malloc(sizeof(Node));
    p->start=1;
    p->end=100;
    p->big=100;
    p->next=NULL;
    unused_head->next=p;

}

//首次适应函数   ////////////////////
void join_first(){
    //this is the new memory we need for our new process
    Node* new_one=(Node*)malloc(sizeof(Node));
    new_one->big=creater_random(1, 20);
    new_one->next=NULL;
```

```
    Node* temp=unused_head;
    int suc_fl=0;
    while(temp->next){
        temp=temp->next; //转移到下一个节点

        //successful
        if(temp->big>=new_one->big){
            new_one->start=temp->start;
            new_one->end=new_one->start+new_one->big-1;

            temp->start+=new_one->big;
            temp->big-=new_one->big;

            //将newone 插入到已使用链中 used_head
            insert(used_head,new_one);

            printf("内存分配成功\n");
            suc_fl=1;
            mk_claen();
            break;
        }
    }
    if(suc_fl==0){
        printf("内存不足，分配不成功\n");
    }
}


//最佳适应函数  ////////////////////
void join_best(){
    //this is the new memory we need for our new process
    Node* new_one=(Node*)malloc(sizeof(Node));
    new_one->big=creater_random(1, 20);
    new_one->next=NULL;

    Node* tp=unused_head;
    int min=10000;
    Node* temp=NULL; //指向最优分配节点

    while(tp->next){
        tp=tp->next; //转移到下一个节点
        int remine=tp->big-new_one->big;
        if(remine>=0 && remine<min){
            min=remine;
            temp=tp;
        }
    }
    if(temp){
        new_one->start=temp->start;
```

```
        new_one->end=new_one->start+new_one->big-1;

        temp->start+=new_one->big;
        temp->big-=new_one->big;

        //将newone 插入到已使用链中 used_head
        insert(used_head,new_one);

        printf("内存分配成功\n");
        mk_claen();
    }
    else{
        printf("内存不足，分配不成功\n");
    }
}


//随机作业完成函数 version 2   ////////////////////
void work_comp02(){
    Node* tp=used_head;
    int relise_flag=0;
    Node* reli_p=NULL;

    if(tp->next==NULL){
        printf("没有作业可以完成并回收\n\n");
        mk_claen();
    }
    else{
        //随机找到一个作业，从使用链中剪出
        while (relise_flag==0) {
            if(creater_random(0, 100)<30){
                if(tp->next==NULL){
                    ;
                }
                else{
                    reli_p=tp->next;
                    tp->next=reli_p->next;
                    relise_flag=1;
                }
            }
            if (tp->next==NULL) {
                tp=used_head;
            }else{
                tp=tp->next;
            }
        }
        //将剪出作业插入未使用链
        insert(unused_head,reli_p);
        //合并整理未使用链
        mk_claen();
```

```
        printf("作业内存回收完毕\n");
    }
      mk_claen();
}

//打印内存分配情况 1-100  *-*-*已分配 0-0-0未分配 /////////////////
void show(){
    Node* tp=unused_head;
    if (tp->next->start>1) {
        for(int i=1;i<tp->next->start;i++){
            printf("%d\t: *-*-*\n",i);
        }
    }
    while(tp->next){
        tp=tp->next;
        printf("-------------\n");
        for (int i=tp->start; i<=tp->end; i++) {
            printf("%d\t: 0-0-0\n",i);
        }
        if (tp->next){
            printf("-------------\n");
            for(int i=tp->end+1;i<tp->next->start;i++){
                printf("%d\t: *-*-*\n",i);
             }
        }
    }//end while

    if(tp->end<100){
         printf("-------------\n");
        for (int i=tp->end+1; i<=100; i++) {
            printf("%d\t: *-*-*\n",i);
        }
    }

}//end fun show()

//random number creater a-b
int creater_random(int a,int b){

    srand((int)time(NULL));    //每次执行种子不同，生成不同的随机数

    return a+rand()%(b-a+1);
}

//队列插入元素
void insert(Node* T,Node* one){

    Node* temp_used=T;
    int insert_flag=0;

    while(temp_used->next){
```

```
        if(one->start < (temp_used->next->start) ){
            one->next=temp_used->next;
            temp_used->next=one;
            insert_flag=1;
            break;
        }
        temp_used=temp_used->next;
    }
    if(insert_flag==0){  //上述插入不成功，就放队列最后
        temp_used->next=one;
    }
}

//对未分配链 unuse_head做合并清理的工作
void mk_claen(){
    Node* tp=unused_head;
    while(tp->next){

        //清理
        if(tp->next->big==0){
            Node* fre_p=tp->next;
            tp->next=fre_p->next;
            free(fre_p);
        }
        if(tp->next==NULL){

        }else{
          tp=tp->next;
        }

    }
    tp=unused_head;
    while(tp->next){
        //合并
        if( (tp->end+1)==tp->next->start){
            Node* fre_p=tp->next;
            tp->big+=fre_p->big;
            tp->end=fre_p->end;
            tp->next=fre_p->next;
            free(fre_p);
        }
        if(tp->next==NULL){

        }else{
            tp=tp->next;
        }
    }
}

//碎片拼接，整合剩余空间  //////////////////
```

```
void connected(){
   //紧凑已使用队列
   Node* tp=used_head->next;
   int content=0;

   if(tp==NULL){
      printf("使用队列为空,无需碎片拼接\n");
   }
   else{
      tp->start=1;
      tp->end=tp->big;
      content+=tp->big;

      while(tp->next){
         tp->next->start=tp->end+1;
         tp=tp->next;              //tp point to the current Node
         tp->end=tp->start+tp->big-1;
         content+=tp->big;
      }

      Node* temp=(Node*)malloc(sizeof(Node));
      temp->start=tp->end+1;
      temp->big=100-content;
      temp->end=temp->start+temp->big-1;
      temp->next=NULL;

      //归并未使用队列,free all current Node,and add a new one
      tp=unused_head;
      while(tp->next){
         Node* fre_p=tp->next;
         tp->next=fre_p->next;
         free(fre_p);
      }
      unused_head->next=temp;

      printf("碎片拼接成功\n");
   }

}
```