操作系统课程设计报告

时间：2010-12-20～2010-12-31

地点：信息技术实验中心

计算机科学与技术专业

2008 级 2 班 15 号

杨　烨

2010-12-31

目　　录

一、课程设计的目的和意义.....2
二、进程调度算法模拟.....2
1、设计目的.....2
2、设计要求.....2
3、时间片轮转算法模拟.....3
实现思想：.....3
（1）流程图.....4
（2）程序代码.....4
（3）运行结果.....6
4、先来先服务算法模拟.....7
算法思想.....7
（1）流程图.....8
（2）程序代码.....8
（3）运行结果.....12
三、主存空间的回收与分配.....13
1、设计目的.....13
2、设计要求.....13
3、模拟算法的实现.....15
（1）流程图.....15
（2）程序代码.....15
（3）运行结果.....30
四、模拟 DOS 文件的建立和使用.....30
1 设计目的.....30
2 设计要求.....30
3、模拟算法实现.....33
（1）流程图.....33
（2）程序代码.....33
（3）运行结果.....38
五、磁盘调度算法模拟.....38
1.设计目的.....38
2.实验原理.....39
3．设计要求.....39
4、模拟算法的实现.....40
（1）各算法流程图.....40
（2）程序代码.....41
（3）运行结果.....47
六、总结.....47

一、课程设计的目的和意义

本次操作系统课程设计的主要任务是进行系统级的程序设计。本课程设计是操作系统原理课程的延伸。通过该课程设计，使学生更好地掌握操作系统各部分结构、实现机理和各种典型算法，加深对操作系统的设计和实现思路的理解，培养学生的系统设计和动手能力，学会分析和编写程序。课程设计的实施将使学生在以下几个方面有所收获：

（1）加深对操作系统原理的理解，提高综合运用所学知识的能力；

（2）培养学生自主查阅参考资料的习惯，增强独立思考和解决问题的能力；

（3）通过课程设计，培养严谨的科学态度和协作精神。

二、进程调度算法模拟

1、设计目的

（1）要求学生设计并实现模拟进程调度的算法：时间片轮转及先来先服务。

（2）理解进程控制块的结构。

（3）理解进程运行的并发性。

（4）掌握进程调度算法。

2、设计要求

在多道程序运行环境下，进程数目一般多于处理机数目，使得进程要通过竞争来使用处理机。这就要求系统能按某种算法，动态地把处理机分配给就绪队列中的一个进程，使之运行，分配处理机的任务是由进程调度程序完成的。一个进程被创建后，系统为了便于对进程进行管理，将系统中的所有进程按其状态，将其组织成不同的进程队列。于是系统中有运行进程队列、就绪队列和各种事件的进程等待队列。进程调度的功能就是从就绪队列中挑选一个进程到处理机上运行。进程调度的算法有多种，常用的有优先级调度算法、先来先服务算法、时间片轮转算法。

进程是程序在处理机上的执行过程。进程存在的标识是进程控制块（PCB），进程控制块结构如下：

```
typedef struct node
{
   char name[10];                  /* 进程标识符 */
   int prio;                        /* 进程优先数 */
   int round;                       /* 进程时间轮转时间片 */
   int cputime;                     /* 进程占用 CPU 时间*/
   int needtime;                    /* 进程到完成还需要的时间*/
   int count;                       /* 计数器*/
   char state;                      /* 进程的状态*/
   struct node *next                 /*链指针*/
}PCB;
```

系统创建一个进程，就是由系统为某个程序设置一个 PCB，用于对该进程进行控制和管理，进程任务完成，由系统收回其 PCB，该进程便消亡。每个进程可以有三个状态：运行状态、就绪状态和完成状态。

用 C 语言、C++或者 Java 语言编写一个程序实现进程调度的算法，模拟进程调度的过程，加深对进程控制块概念和进程调度算法的理解。

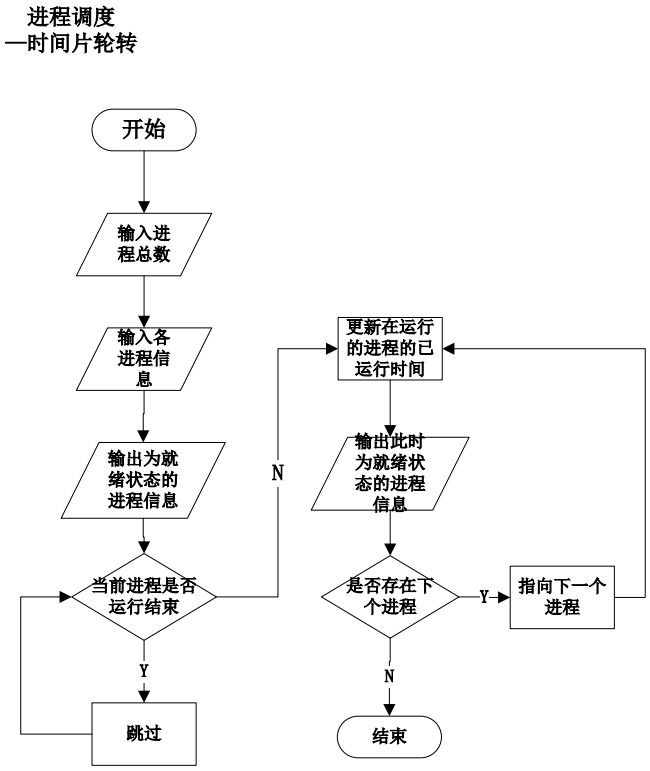
本任务要求完成时间片轮转及先来先服务两个算法。

3、时间片轮转算法模拟

实现思想：

每次调度时，系统吧处理机分配给队列首进程让器执行一个时间片，当执行的时间片用完时，由一个计时器发出时钟中断请求，调度根据这个请求停止该进程的运行将其送到就绪队列的末尾，再把处理机分给就绪队列中新的队首进程，同时让它执行一个时间片。

（1）流程图



（2）程序代码

```
#include <iostream.h>
#include <cstdlib>
using namespace std;
typedef struct PNode
{
    // PCB
    struct PNode *next; // 定义指向下一个节点的指针
```

```
char  name[10];      // 定义进程名，并分配空间
```

```
    int  All_Time;      // 定义总运行时间
    int  Runed_Time;    // 定义已运行时间
    char  state;        // 定义进程状态  Ready / End
}
* Proc; // 指向该 PCB 的指针
int ProcNum; // 总进程个数
void InitPCB(Proc &H)// 初始化就绪队列
{
    cout<<"请输入总进程个数: ";
    cin>>ProcNum; // 进程总个数
    int Num=ProcNum;
    H=(Proc)malloc(sizeof(PNode)); // 建立头节点
    H->next=NULL;
    Proc p=H;  //定义一个指针
    cout<<"总进程个数为 "<<ProcNum<<" 个,请依次输入相应信息\n\n";
while (Num--)
{
        p=p->next=(Proc)malloc(sizeof(PNode));
        cout<<"进程名 总运行时间 已运行时间 :";
        cin>>p->name>>p->All_Time>>p->Runed_Time;
        p->state='R';
        p->next=NULL;
}
    p->next=H->next;
}
void DispInfo(Proc H) //输出运行中的进程信息
{
    Proc p=H->next;
    do
        {
        if (p->state != 'E')    //如果该进程的状态不是 End 的话
        {
            cout<<"进程名:"<<p->name<<"\t 总运行时间:"<<p->All_Time
                <<"\t 已运行时间:"<<p->Runed_Time
                <<"\t 状态:"<<p->state<<endl;
            p=p->next;
        }
        else p=p->next;
        }
    while (p != H->next); // 整个进程链条始终完整，只是状态位有差异
}
void SJP_Simulator(Proc &H) // 时间片轮转法
{
```

```
        cout<<endl<<"-------------------START--------------------\n";
        int flag=ProcNum; // 记录剩余进程数
        int round=0; // 记录轮转数
        Proc p=H->next;
        while (p->All_Time > p->Runed_Time)
       {
                // 即未结束的进程
                round++;
                cout<<endl<<"Round "<<round<<"--正在运行 "<<p->name<<" 进程"<<endl;
                p->Runed_Time++;    // 更改正在运行的进程的已运行时间
                DispInfo(H);        // 输出此时为就绪状态的进程的信息
                if (p->All_Time == p->Runed_Time) {    // 并判断该进程是否结束
                p->state='E';
                flag--;
                cout<<p->name<<" 进程已运行结束,进程被删除!\n";
        }
         p=p->next;
         while (flag && p->All_Time == p->Runed_Time)
         p=p->next; // 跳过先前已结束的进程
}
cout<<endl<<"--------------------END---------------------\n";
}
void main()
{
        Proc H;
        InitPCB(H); // 数据初始化
        DispInfo(H); // 输出此刻的进程状态
        SJP_Simulator(H); // 时间片轮转法
        system("pause");
}
```

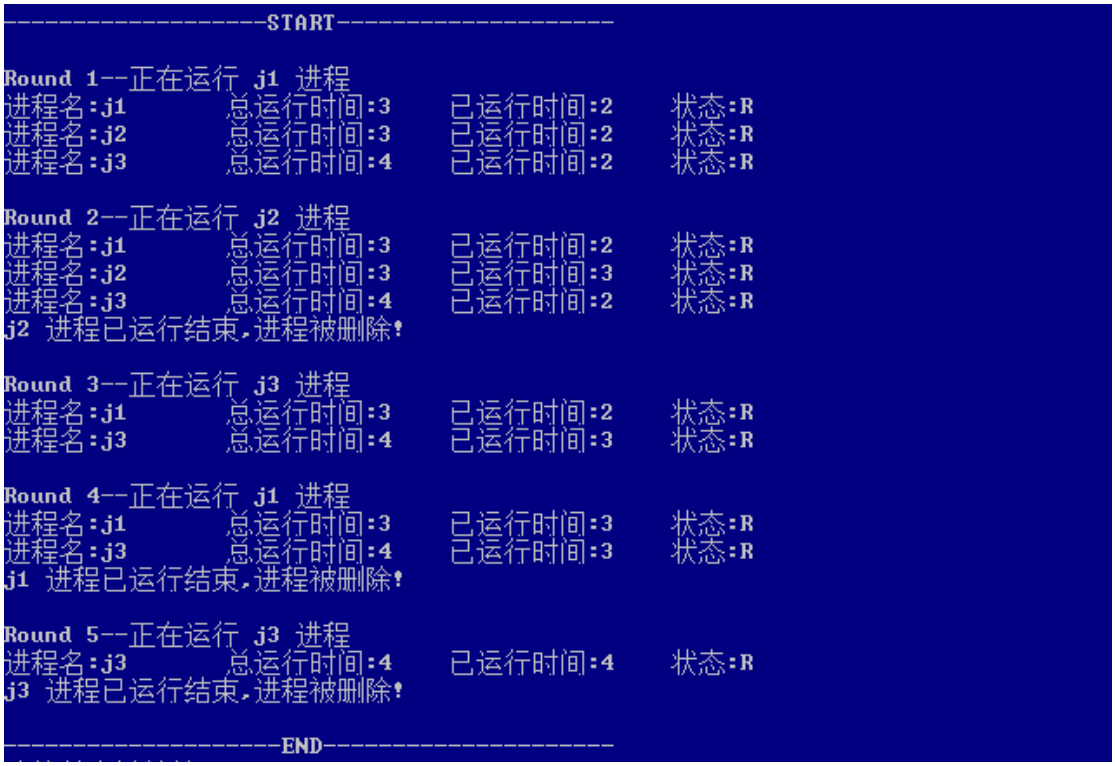
（3）运行结果

输入相关进程信息：

```
请输入总进程个数: 3
总进程个数为 3 个,请依次输入相应信息

进程名 总运行时间 已运行时间 :j1 3 1
进程名 总运行时间 已运行时间 :j2 3 2
进程名 总运行时间 已运行时间 :j3 4 2
进程名:j1       总运行时间:3    已运行时间:1    状态:R
进程名:j2       总运行时间:3    已运行时间:2    状态:R
进程名:j3       总运行时间:4    已运行时间:2    状态:R
```

输出相关运行结果：



4、先来先服务算法模拟

算法思想

按照进程的某种顺序进行排序，然后按照这个顺序进行调度。例如：可以按照作业提交时间或进程变为就绪状态的先后次序来分派处理器，让排在后面的进程占用处理器，知道该进程执行完或者由于某种原因被阻塞才让出处理器。

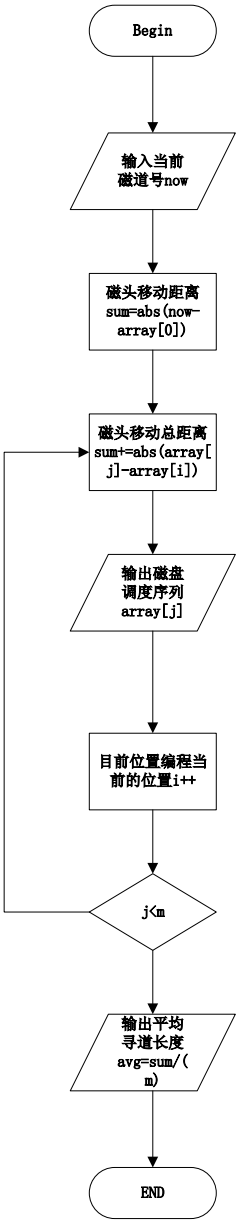
在该调度策略中还规定，当有一个事件发生（如一个 I/O 操作完成）时，会有一些进程被唤醒，这些被唤醒的进程并不能立即恢复执行，而是要等到当前运行的进程出让处理器后才可以被调度执行。采用此算法存在以下几个特点：

周转时间：对进程 i 来说，假设 Tei 是进程的完成时间，Tsi 是进程的提交时间，那么进程 i 的周转时间 Ti=进程完成时间-进程提交时间；

进程平均周转时间：T=1/nETi；

进程带权周转时间=进程等待时间+进程运行时间。

（1）流程图



（2）程序代码

```
#include "stdio.h"
#include <stdlib.h>
#include <conio.h>
#define getpch(type) (type*)malloc(sizeof(type))
#define NULL 0
struct pcb { /* 定义进程控制块 PCB */
char name[10];
char state;
```

```
int  super;
```

```
int ntime;
int rtime;
struct pcb* link;
}*ready=NULL,*p;
typedef struct pcb PCB;
void sort() /* 建立对进程进行优先级排列函数*/
{
PCB *first, *second;
int insert=0;
if((ready==NULL)||((p->super)>(ready->super))) /*优先级最大者,插入队首*/
{
p->link=ready;
ready=p;
}
else /* 进程比较优先级,插入适当的位置中*/
{
first=ready;
second=first->link;
while(second!=NULL)
{
if((p->super)>(second->super)) /*若插入进程比当前进程优先数大,*/
{ /*插入到当前进程前面*/
p->link=second;
first->link=p;
second=NULL;
insert=1;
}
else /* 插入进程优先数最低,则插入到队尾*/
{
first=first->link;
second=second->link;
}
}
if(insert==0) first->link=p;
}
}
void input() /* 建立进程控制块函数*/
{
int i,num;
printf("\n 请输入进程数：");
scanf("%d",&num);
for(i=0;i<num;i++)
{
```

```
printf("\n 进程号 No.%d:\n",i+1);
p=getpch(PCB);
printf("\n 输入进程名:");
scanf("%s",p->name);
printf("\n 输入进程优先数:");
scanf("%d",&p->super);
printf("\n 输入进程运行时间:");
scanf("%d",&p->ntime);
printf("\n");
p->rtime=0;p->state='w';
p->link=NULL;
sort(); /* 调用 sort 函数*/
}
}
int space()
{
int l=0; PCB* pr=ready;
while(pr!=NULL)
{
l++;
pr=pr->link;
}
return(l);
}
void disp(PCB * pr) /*建立进程显示函数,用于显示当前进程*/
{
printf("\n qname \t state \t super \t ndtime \t runtime \n");
printf("|%s\t",pr->name);
printf("|%c\t",pr->state);
printf("|%d\t",pr->super);
printf("|%d\t",pr->ntime);
printf("|%d\t",pr->rtime);
printf("\n");
}
void check() /* 建立进程查看函数 */
{
PCB* pr;
printf("\n **** 当前正在运行的进程是:%s",p->name); /*显示当前运行进程*/
disp(p);
pr=ready;
printf("\n ****当前就绪队列状态为:\n"); /*显示就绪队列状态*/
while(pr!=NULL)
{
```

```
disp(pr);
pr=pr->link;
}
}
void destroy() /*建立进程撤消函数(进程运行结束,撤消进程)*/
{
printf("\n 进程 [%s] 已完成.\n",p->name);
free(p);
}
void running() /* 建立进程就绪函数(进程运行时间到,置就绪状态*/
{
(p->rtime)++;
if(p->rtime==p->ntime)
destroy(); /* 调用 destroy 函数*/
else
{
(p->super)--;
p->state='w';
sort(); /*调用 sort 函数*/
}
}
int
main() /*主函数*/
{
int len,h=0;
char ch;
input();
len=space();
while((len!=0)&&(ready!=NULL))
{
ch=getchar();
h++;
printf("\n The execute number:%d \n",h);
p=ready;
ready=p->link;
p->link=NULL;
p->state='R';
check();
running();
printf("\n 按任一键继续.....");
ch=getchar();
}
printf("\n\n 进程已经完成.\n");
```

```
ch=getchar();
}
```

（3）运行结果

输入相关进程信息：

```
请输入进程数：2
进程号No.1:
输入进程名:j1
输入进程优先数:1
输入进程运行时间:2

进程号No.2:
输入进程名:j2
输入进程优先数:2
输入进程运行时间:2
```

输出相关运行结果：

```
The execute number:1

**** 当前正在运行的进程是:j2
qname    state    super    ndtime           runtime
|j2      |R       |2       |2       |0

****当前就绪队列状态为:

qname    state    super    ndtime           runtime
|j1      |w       |1       |2       |0

按任一键继续......


The execute number:2

**** 当前正在运行的进程是:j1
qname    state    super    ndtime           runtime
|j1      |R       |1       |2       |0

****当前就绪队列状态为:

qname    state    super    ndtime           runtime
|j2      |w       |1       |2       |1

按任一键继续......
```

三、主存空间的回收与分配

1、设计目的

主存是中央处理器能直接存取指令和数据的存储器，能否合理地利用主存，在很大程度上将影响到整个计算机系统的性能。主存分配是指在多道作业和多进程环境下，如何共享主存空间。主存回收是指当作业执行完毕或进程运行结束后将主存空间归还给系统。主存分配与回收的实现是与主存储器的管理方式有关。本次设计主要是为了帮助学生深入理解主存空间的分配与回收的几种算法。

（1）掌握最先适应分配算法

（2）掌握最优适应分配算法

（3）掌握最坏适应分配算法

2、设计要求

用户提出内存空间请求，系统根据申请者要求，按照最先适应分配算法的分配策略分析内存空间的使用情况，找出能满足请求的空闲区，分给申请者，当程序执行完毕时，系统要收回它所占用的内存空间。建立空闲数据文件，空闲区数据文件包括若干行，每行有两个字段：起始地址、内存块大小（均为整数），各字段以逗号隔开。下面是一个空闲区数据文件的示例：

0， 10

10， 08

18， 10

28， 06

34， 10

44， 09

读取空闲区数据文件，建立空闲区表并在屏幕上显示空闲内存状态，空闲区表记录了可供分配的空闲内存的起始地址和大小，用标志位指出该分区是否是未分配的空闲区。

接收用户的内存申请，格式为：作业名、申请空间的大小。

按照内存分配算法中的一种方法选择一个空闲区，分割并分配，修改空闲区表，填写内存已分配区表（起始地址、长度、标志位），其中标志位的一个作用是指出该区域分配给哪个作业。

进程结束后回收内存。

空闲区表的结构如下：

```
typedef struct node{
int start;
int length;
char tag[20];
}job;
```

本次设计要求完成如下算法：

（1）设计一个内存分配回收的程序使用最先适应分配算法

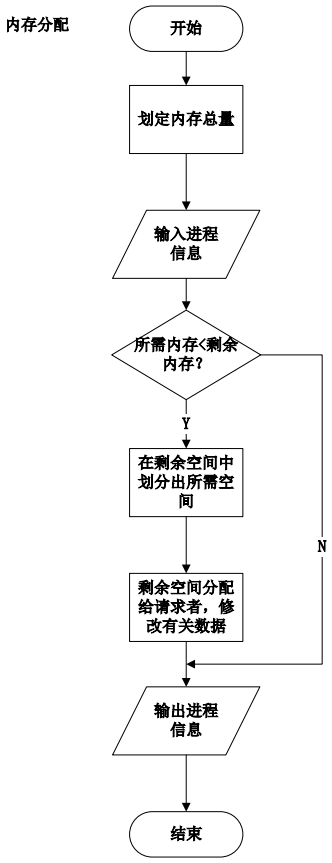
（2）设计一个内存分配回收的程序使用最优适应分配算法

（3）设计一个内存分配回收的程序使用最坏适应分配算法

用户提出内存空间请求，系统根据申请者要求，选择上述算法的一种分配策略分析内存空间的使用情况，找出合适的空闲区，分给申请者，当进程执行完毕时，系统收回它所占用的内存空间。

3、模拟算法的实现

（1）流程图



（2）程序代码

```
#include<stdio.h>
#include<malloc.h>
#define LEN    sizeof(struct area)
#define LEN_POINTER_LIST sizeof(struct AreaPointer_list)
struct area{
        int start;    //分区的其始地址
        int length; //分区的长度
        int job;      //若被作业占用值为作业号，若空闲值为 0
        struct area * next;
};
//分区指针的链表
//当把空闲分区链表和占用分区链表按照地址先后顺序合并
//以显示整个内存情况的时候使用
struct AreaPointer_list{
struct area * data;
```

```
struct AreaPointer_list * next;
```

```
};
struct area * idle;                                    //全局变量，空闲分区链表头指针
struct area * used;                                    //全局变量，占用分区链表头指针
struct AreaPointer_list * whole = NULL;  //全局变量，分区指针链表头指针
//p(previcious) n(next)指出在链表中的何处插入新生成的元素
//p==NULL  在链表头插入，返回头指针
//p!=NULL  在链表中或链表尾插入，返回当前插入的元素的指针
struct area * insert(int s,int l,int j,struct area * p,struct area * n)
{
    struct area * current = (struct area * )malloc(LEN);
    current->start = s;
    current->length = l;
    current->job = j;
    if(p == NULL){//在链表头插入
        current->next = n;
        return current;
    }
    else{
        if(p->next == NULL){//在链表尾插入
            current->next = NULL;
            p->next = current;
        }
        else{//在链表中插入
            current->next = p->next;
            p->next = current;
        }
        return current;
    }
}
//此模块居于次要地位，只被使用一次
//打印分区链表
void print(struct area * head)
{
    if(head == NULL){
        printf("Area list is null...\n");
    }
    else{
        while(head != NULL)
        {
            if(head->job == 0)
                printf("begin:%dK\tlength:%dK\t 空闲\t\t|\n",head->start,head->length);
            else
                printf("begin:%dK\tlength:%dK\tuse:Job%d\t|\n",head->start,head->length,hea
d->job);
```

```
                head = head->next;
            }
        }
    }
    void file_print(struct area * head,FILE * file)
    {
        if(head == NULL){
            fprintf(file,"Area list is null...\n");
        }
        else{
            while(head != NULL)
            {
                if(head->job == 0)
                    fprintf(file,"begin:%dK\tlength:%dK\t 空闲\t\t|\n",head->start,head->length);
                else
                    fprintf(file,"begin:%dK\tlength:%dK\tuse:Job%d\t|\n",head->start,head->length,
    head->job);
                head = head->next;
            }
        }
    }
    //打印分区链表
    //释放分区链表空间
    void free_AreaList(struct area * head)
    {
        struct area * temp;
        while(head != NULL){
            temp = head;
            head = head->next;
            free(temp);
        }
    }
    //释放分区链表空间
    //在分区链表中搜索插入位置
    //flag==0 表明分区链表按起始地址从小到大排列
    //flag==1 表明分区链表按分区长度从小到大排列
    //输入参数 element 不能为 NULL
    struct area * search_pos(struct area * element,struct area * head,int flag)
    {
        struct area * p = NULL;
        while(head != NULL){
            if(flag == 0)
            {
                if(element->start < head->start)
```

```
                    break;
            }
            else
            {
                if(element->length < head->length)
                    break;
            }
            p = head;
            head = head->next;
        }
        return p;
}
//返回值 p==NULL 表明插入位置在链表头
//返回值 p!=NULL 表明插入位置在 p 之后
//进行分区链表的实际插入工作
//flag==0 表明分区链表按起始地址从小到大排列
//flag==1 表明分区链表按分区长度从小到大排列
//输入参数 element->next 要为 NULL
struct area * insert_list(struct area * element,struct area * list,int flag)
{
    if(list == NULL)
        list = element;
    else{
        struct area * pos = search_pos(element,list,flag);
        if(pos == NULL){
            element->next = list;
            list = element;
        }
        else{
            element->next = pos->next;
            pos->next = element;
        }
    }
    return list;
}//返回插入元素之后新链表的头指针
//进行查询空闲分区链表动态分配分区的实际工作，算法步骤：
//1。查询空闲分区链表中是否有长度大于或等于申请长度的分区，若没有返回 FALSE
//2。若查找到符合条件的分区，把它从空闲链表中取出
//3。根据请求把取出的空闲分区分块，把新的占用分区和剩余空闲分区分别插入链表
//注意：插入占用分区链表按照固定的地址先后顺序，插入空闲分区链表的方式要根据 flag 的值
int memory_alloc(int length,int job,int flag)
{
    struct area * used_element;
```

```
struct area * free_element;
struct area * head = idle;
struct area * head_temp = used;
struct area * p = NULL;
//检测输入的作业号是否存在
while(head_temp != NULL)
{
    if(head_temp->job == job)
        return 2;
    head_temp = head_temp->next;
}
//在空闲分区链表中查找
while(head != NULL)
{
    if(head->length >= length)
        break;
    p = head;
    head = head->next;
}
if(head != NULL)
{
    //从空闲区链表中取出
    if(p == NULL)//链表中的第一个分区符合条件
    {
        idle = idle->next;
    }
    else
    {
        p->next = head->next;
    }
    head->next = NULL;
}
else return 0;
//生成新的占用区链表元素并插入占用区链表
used_element = (struct area * )malloc(LEN);
used_element->start = head->start;
used_element->length = length;
used_element->job = job;
used_element->next = NULL;
used = insert_list(used_element,used,0);
//若空闲分区分块后有剩余，生成新的空闲区链表元素并插入空闲区链表
if(head->length > length){
    free_element = (struct area * )malloc(LEN);
```

```
            free_element->start = head->start + length;
            free_element->length = head->length - length;
            free_element->job = 0;
            free_element->next = NULL;
            idle = insert_list(free_element,idle,flag);
        }
        //释放空间
        free(head);
        return 1;
}
//进行查询占用分区链表动态释放分区的实际工作，算法步骤：
//1。根据作业号查询到占用分区链表中要释放的分区，若没有返回 FALSE
//2。若查找到要释放的分区，把它从空闲链表中取出
//3。根据取出的分区的数据建立新的空闲分区
//4。在空闲分区链表中查询是否有和新空闲分区相邻的空闲分区，有则合并
//5。根据 flag 的取值按照特定方式插入空闲分区链表
int memory_free(int job,int flag)
{
        struct area * used_element;
        struct area * free_element;
        struct area * head = used;
        struct area * p = NULL;
        struct area * previcious1 = NULL;
        struct area * current1 = NULL;
        struct area * previcious2 = NULL;
        struct area * current2 = NULL;
        //根据作业号在占用分区链表中查找
        while(head != NULL)
        {
            if(head->job == job)
                break;
            p = head;
            head = head->next;
        }
        if(head != NULL)
        {
            //从占用区链表中取出
            if(p == NULL)//链表中的第一个分区符合条件
            {
                used = used->next;
            }
            else
            {
```

```
            p->next = head->next;
        }
        head->next = NULL;
    }
    else return 0;
    //建立新的空闲分区
    used_element = head;
    free_element = (struct area * )malloc(LEN);
    free_element->start = used_element->start;
    free_element->length = used_element->length;
    free_element->job = 0;
    free_element->next = NULL;
    //从空闲区链表查找和新的空闲分区相邻分区
    head = idle;
    p = NULL;
    while(head != NULL)
    {
        if(head->start + head->length == used_element->start)
        {
            previcious1 = p;
            current1 = head;
        }
        if(used_element->start + used_element->length == head->start)
        {
            previcious2 = p;
            current2 = head;
        }
        p = head;
        head = head->next;
    }
    //合并相邻空闲分区
    if( current1 != NULL )
    {
        //把和新分区相邻的分区从空闲分区链表中取出
        if( previcious1 == NULL )
            idle = idle->next;
        else
            previcious1->next = current1->next;
        current1->next = NULL;
        //修改新空闲分区的相关数据
        free_element->start = current1->start;
        free_element->length = free_element->length + current1->length;
    }
```

```
        if( current2 != NULL )
        {
            //把和新分区相邻的分区从空闲分区链表中取出
            if( previcious2 == NULL )
                idle = idle->next;
            else
                previcious2->next = current2->next;
            current2->next = NULL;
            //修改新空闲分区的相关数据
            free_element->length = free_element->length + current2->length;
        }
        //根据 flag 的取值按照特定方式插入空闲分区链表
        idle = insert_list(free_element,idle,flag);
        //释放空间
        free(used_element);
        return 1;
}
//和分区指针链表相关的操作，用来合并空闲分区链表和占用分区链表，保存链表元素的指针
struct AreaPointer_list * search_position(int s,struct AreaPointer_list * head)
{
        struct AreaPointer_list * p = NULL;
        while(head != NULL){
            if(s < (head->data)->start)
                break;
            p = head;
            head = head->next;
        }
        return p;
}
struct AreaPointer_list * emerge(struct area * idle_temp,struct area * used_temp)
{
        struct AreaPointer_list * previcious;
        struct AreaPointer_list * temp;

        if(used_temp != NULL)
        {
            whole = (struct AreaPointer_list *)malloc(LEN_POINTER_LIST);
            whole->data = used_temp;
            whole->next = NULL;
            previcious = whole;
            used_temp = used_temp->next;
            while(used_temp != NULL){
                temp = (struct AreaPointer_list *)malloc(LEN_POINTER_LIST);
```