

**计算机与信息 学院实验报告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验课程： | 操作系统实验 | | | | |
| 实验编号： | 实验四 | | | | |
| 实验名称： | 动态分区分配 | | | | |
| 实验人员： | 学号 | 18111207248 | | | |
| 姓名 | 吴钰 | | | |
| 班级 | 18级计算机科学与技术创新班 | | | |
| 实验日期： | 2020.5.20 | | | | |
| 实验室： |  | | | | |
|  |  | | | | |
| 实验评价： |  | | | | |
| 实验成绩： | |  | 评价日期： |  |
|  | 指导教师： | |  | | |

# 一、实验目的

# 熟悉内存的分配与回收过程；理解在不同的管理方式下，如何实现内存空间的分配与回收；通过实验，掌握动态分区分配方式中的数据结构、分配算法、动态分区存储管理方式及其实现过程。

# 二、实验要求

（1）问题描述

编写一个内存动态分区分配模拟程序，用于模拟内存的分配和回收的完整过程，并根据模拟情况，

讨论几种算法的优劣。

（2）基本要求

分别实现四种分配算法：首次适应、循环首次适应、最佳适应和最坏适应；

每次分配和回收后把空闲分区的变化情况以及各进程的申请、释放情况尽量能以图形方式显

示；

尽量能设计一个实用、友好的可视化用户界面，显示内存区域的动态变化情况、分配与回收的

过程。

# 三、设计

【定义所有抽象数据类型，自定义函数的功能详细描述（伪代码表示），以及主程序的流程图。】

#include<iostream>

#include<algorithm>

#include<queue>

using namespace std;

#define MAXNUMBER 100

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*各个变量的含义\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int ProcessNum; //进程个数

int PartitionNum; //内存中空闲分区的个数

int FreePartition[MAXNUMBER]; //空闲分区对应的内存

int begin[MAXNUMBER];

char ProcessName[MAXNUMBER];//进程名字

int ProcessNeed[MAXNUMBER]; //进程大小

int Processarrive[MAXNUMBER];//进程到达时间

int Processend[MAXNUMBER];//进程结束时间

char NameProcessToPartition[MAXNUMBER][MAXNUMBER];//各个进程所在分区位置

int choose;//算法选择

int LeftFreePartition[MAXNUMBER];//剩下空闲内存

int LeftProcessNeed[MAXNUMBER];//剩下进程需要空间

int LeftProcessarrive[MAXNUMBER];//剩下进程到达时间

int LeftProcessend[MAXNUMBER];//剩下进程结束时间

typedef struct{ //进程结构体

char ProcessName;

int ProcessNeed;

int Processarrive;

int Processend;

int flag;//判断是否分配到空间

}Process;

typedef struct lis{//空闲分区表

int partitionSize;//空闲区大小

int id; //空闲区编号

int begin; //起始地址

lis \*next;

}sortNeed;

bool cmp1(sortNeed a,sortNeed b) //最佳适应算法和最坏适应算法用到排序

{

return a.partitionSize<b.partitionSize; //如果是return a<b 的话就是升序排序；

}

bool cmp2(sortNeed a,sortNeed b) //最佳适应算法和最坏适应算法用到排序

{

return a.partitionSize>b.partitionSize; //如果是return a>b 的话就是降序排序；

}

int num1,num2;//分别记录空闲分区和已分配区域的个数

sortNeed Empty[MAXNUMBER];

sortNeed Full[MAXNUMBER];

sortNeed \*Emptylist=NULL;

sortNeed \*Fulllist=NULL;

Process run[MAXNUMBER];

bool operator<(Process a,Process b) //重载自定义类型优先级比较

{

return a.Processarrive>b.Processarrive;

}

void Enter();//输入分区数和大小、资源数和大小

void initial();//初始化

void display();//显示分区结果

void FirstFit();//首次适应算法FF

void NextFit();//循环首次适应算法NF

void BestFit();//最佳适应算法BF

void WorstFit();//最坏适应算法WF

void choose\_Algorithm();//选择算法

priority\_queue<Process>PCB;

int main()

{

Enter();

choose\_Algorithm();

return 0;

}

void Enter(){

int i;

cout<<"请输入分区个数\n";

cin>>PartitionNum;

cout<<"请输入进程个数\n";

cin>>ProcessNum;

cout<<"请输入每个分区的大小和起始地址\n";

for (i=0;i<PartitionNum;i++){//对应的内存

cin>>FreePartition[i];

cin>>begin[i];

}

for (i=0;i<ProcessNum;i++){//初始化名称

ProcessName[i]=i+65;

}

cout<<"请输入每个进程的进程名 大小 到达时间 结束时间\n";

for(i=0;i<ProcessNum;i++){

cin>>ProcessName[i];//进程名

cin>>ProcessNeed[i];//进程大小

cin>>Processarrive[i];//进程到达时间

cin>>Processend[i];//进程结束时间

}

//按照到达时间的顺序加入优先队列

cout<<"读取数据如下："<<endl;

cout<<"进程名称: "<<"\t\t";

for (i=0;i<ProcessNum;i++){

cout<<ProcessName[i]<<"\t";

}

cout<<endl;

cout<<"进程大小： "<<"\t\t";

for (i=0;i<ProcessNum;i++){

cout<<ProcessNeed[i]<<"\t";

}

cout<<endl;

cout<<"进程到达时间： "<<"\t\t";

for (i=0;i<ProcessNum;i++){

cout<<Processarrive[i]<<"\t";

}

cout<<endl;

cout<<"进程结束时间： "<<"\t\t";

for (i=0;i<ProcessNum;i++){

cout<<Processend[i]<<"\t";

}

cout<<endl;

cout<<"分区名称: "<<"\t\t";

for (i=0;i<PartitionNum;i++){

cout<<"P"<<i+1<<"\t";

}

cout<<endl<<"内存大小: "<<"\t\t";

for (i=0;i<PartitionNum;i++){

cout<<FreePartition[i]<<"\t";

}

cout<<endl<<"----------------------"<<endl;

}

void FirstFit()

{

int count=0;//统计进程的完成个数

int t=0;//统计正在运行的进程个数

int x=PartitionNum;

sortNeed \*p;

sortNeed \*q;

cout<<"FirstFit"<<endl;

initial();

int i,j;

for(i=0;count!=ProcessNum;i++)//不考虑空间不足的情况

{

cout<<"在"<<i<<"时刻"<<" ";

while(i==PCB.top().Processarrive&&PCB.empty()!=1)

{

run[t++]=PCB.top();

PCB.pop();

cout<<"进程"<<run[t-1].ProcessName<<"到达"<<" ";

//某个进程到达

}

for(j=0;j<t;j++)

{

if(run[j].Processend==i) //回收进程已被分配空间

{

int sum=0;

count++;

cout<<"进程"<<run[j].ProcessName<<"完成"<<" ";

p=Emptylist;

q=Fulllist;

while(q->id+65!=run[j].ProcessName)

{

q=q->next;

}

while(1)

{

if(q->begin==p->begin+p->partitionSize)

{

p->partitionSize=p->partitionSize+q->partitionSize;

break;

}else if(q->begin+q->partitionSize==p->begin)

{

p->begin=q->begin;

p->partitionSize=p->partitionSize+q->partitionSize;

break;

}else if(p->next==NULL)

{

x++;

sortNeed \*e;

if((e = (sortNeed \*)malloc(sizeof(sortNeed)))==NULL) //更新已分配表

{

perror("malloc");

exit(1);

}

e->id=x-1;

e->begin=q->begin;

e->next=NULL;

e->partitionSize=q->partitionSize;

p->next=e;

break;

}

p=p->next;

}

sortNeed \*e=Fulllist; //已分配区域删除这个模块

if(e==q)

{

Fulllist=Fulllist->next;

}else

{

while(e->next!=q)

{

e=e->next;

}

e->next=q->next;

}

e=Emptylist;

while(e!=NULL)

{

p=Emptylist;

while(p!=NULL)

{

if(e->begin+e->partitionSize==p->begin)

{

e->partitionSize=e->partitionSize+p->partitionSize;

q=Emptylist;

while(q->next!=p&&q->next!=NULL&&p!=Emptylist)

{

q=q->next;

}

if(p==Emptylist)

Emptylist=Emptylist->next;

q->next=p->next;

break;

}

p=p->next;

}

e=e->next;

}

}

if(run[j].flag==0&&(run[j].Processarrive<run[j].Processend)) //看空闲分区表中是否有足够空间分配给进程

{

p=Emptylist;

while(p!=NULL)

{

if(p->partitionSize>=run[j].ProcessNeed)

{

if((q = (sortNeed \*)malloc(sizeof(sortNeed)))==NULL) //更新已分配表

{

perror("malloc");

exit(1);

}

q->id=run[j].ProcessName-65;

q->partitionSize=run[j].ProcessNeed;

q->begin=p->begin;

q->next=Fulllist;

Fulllist=q;

p->begin =p->begin+run[j].ProcessNeed;

p->partitionSize=p->partitionSize-run[j].ProcessNeed;

run[j].flag=1;

cout<<"进程"<<run[j].ProcessName<<"得到P"<<p->id<<"的空间"<<" ";

break;

}

p=p->next;

}

}

}

cout<<endl;

cout<<"此时刻所有空闲分区的首地址和大小"<<endl;

p=Emptylist;

while(p!=NULL)

{

cout<<"P"<<p->id<<" "<<p->begin<<" "<<p->partitionSize<<endl;

p=p->next;

}

cout<<"此时刻所有已分配分区的首地址和大小"<<endl;

q=Fulllist;

while(q!=NULL)

{

char c=q->id+65;

cout<<c<<" "<<q->begin<<" "<<q->partitionSize<<endl;

q=q->next;

}

cout<<endl;

}

}

void NextFit()

{

cout<<"NextFit"<<endl;

int count=0;//统计进程的完成个数

int t=0;//统计正在运行的进程个数

int x=PartitionNum;

sortNeed \*p;

sortNeed \*q;

initial();

int i,j;

sortNeed \*r=Emptylist; //标记位置，从上一次这里开始查找

for(i=0;count!=ProcessNum;i++)//不考虑空间不足的情况

{

cout<<"在"<<i<<"时刻"<<" ";

while(i==PCB.top().Processarrive&&PCB.empty()!=1)

{

run[t++]=PCB.top();

PCB.pop();

cout<<"进程"<<run[t-1].ProcessName<<"到达"<<" ";

//某个进程到达

}

for(j=0;j<t;j++)

{

if(run[j].Processend==i) //回收进程已被分配空间

{

int sum=0;

count++;

cout<<"进程"<<run[j].ProcessName<<"完成"<<" ";

p=Emptylist;

q=Fulllist;

while(q->id+65!=run[j].ProcessName)

{

q=q->next;

}

while(1)

{

if(q->begin==p->begin+p->partitionSize)

{

p->partitionSize=p->partitionSize+q->partitionSize;

break;

}else if(q->begin+q->partitionSize==p->begin)

{

p->begin=q->begin;

p->partitionSize=p->partitionSize+q->partitionSize;

break;

}else if(p->next==NULL)

{

x++;

sortNeed \*e;

if((e = (sortNeed \*)malloc(sizeof(sortNeed)))==NULL) //更新已分配表

{

perror("malloc");

exit(1);

}

e->id=x-1;

e->begin=q->begin;

e->next=NULL;

e->partitionSize=q->partitionSize;

p->next=e;

break;

}

p=p->next;

}

sortNeed \*e=Fulllist; //已分配区域删除这个模块

if(e==q)

{

Fulllist=Fulllist->next;

}else

{

while(e->next!=q)

{

e=e->next;

}

e->next=q->next;

}

e=Emptylist;

while(e!=NULL)

{

p=Emptylist;

while(p!=NULL)

{

if(e->begin+e->partitionSize==p->begin)

{

e->partitionSize=e->partitionSize+p->partitionSize;

q=Emptylist;

while(q->next!=p&&q->next!=NULL&&p!=Emptylist)

{

q=q->next;

}

if(p==Emptylist)

Emptylist=Emptylist->next;

q->next=p->next;

break;

}

p=p->next;

}

e=e->next;

}

}

if(run[j].flag==0&&(run[j].Processarrive<run[j].Processend)) //看空闲分区表中是否有足够空间分配给进程

{

p=r;

do

{

if(p->partitionSize>=run[j].ProcessNeed)

{

if((q = (sortNeed \*)malloc(sizeof(sortNeed)))==NULL) //更新已分配表

{

perror("malloc");

exit(1);

}

q->id=run[j].ProcessName-65;

q->partitionSize=run[j].ProcessNeed;

q->begin=p->begin;

q->next=Fulllist;

Fulllist=q;

p->begin =p->begin+run[j].ProcessNeed;

p->partitionSize=p->partitionSize-run[j].ProcessNeed;

run[j].flag=1;

cout<<"进程"<<run[j].ProcessName<<"得到P"<<p->id<<"的空间"<<" ";

break;

}

if(p->next!=NULL)

p=p->next;

else

p->next=Emptylist;

}while(p!=r);

if(p->next!=NULL)

r=p->next;

else

r=Emptylist;

}

}

cout<<endl;

cout<<"此时刻所有空闲分区的首地址和大小"<<endl;

p=Emptylist;

while(p!=NULL)

{

cout<<"P"<<p->id<<" "<<p->begin<<" "<<p->partitionSize<<endl;

p=p->next;

}

cout<<"此时刻所有已分配分区的首地址和大小"<<endl;

q=Fulllist;

while(q!=NULL)

{

char c=q->id+65;

cout<<c<<" "<<q->begin<<" "<<q->partitionSize<<endl;

q=q->next;

}

cout<<endl;

}

}

void BestFit()

{

cout<<"BestFit"<<endl;

initial();

sort(Empty,Empty+num1,cmp1);//按进程大小升序排序

int count=0;//统计进程的完成个数

int t=0;//统计正在运行的进程个数

int i,j,l,k;

for(i=0;count!=ProcessNum;i++)//不考虑空间不足的情况

{

cout<<"在"<<i<<"时刻"<<" ";

while(i==PCB.top().Processarrive&&PCB.empty()!=1)

{

run[t++]=PCB.top();

PCB.pop();

cout<<"进程"<<run[t-1].ProcessName<<"到达"<<" ";

//某个进程到达

}

for(j=0;j<t;j++)

{

if(run[j].Processend==i) //回收进程已被分配空间

{

count++;

cout<<"进程"<<run[j].ProcessName<<"完成"<<" ";

for(k=0;k<num2;k++)

{

if(Full[k].id+65==run[j].ProcessName)

break;

}

for(l=0;l<num1;l++)

{

if(Full[k].begin==Empty[l].begin+Empty[l].partitionSize)

{

Empty[l].partitionSize=Empty[l].partitionSize+Full[k].partitionSize;

sort(Empty,Empty+num1,cmp1);

break;

}else if(Full[k].begin+Full[k].partitionSize==Empty[l].begin)

{

Empty[l].begin=Full[k].begin;

Empty[l].partitionSize=Empty[l].partitionSize+Full[k].partitionSize;

sort(Empty,Empty+num1,cmp1);

break;

}else if(l==num1-1)

{

num1++;

Empty[num1-1].partitionSize=Full[k].partitionSize;

Empty[num1-1].begin=Full[k].begin;

Empty[num1-1].next=NULL;

Empty[num1-1].id=num1-1;

sort(Empty,Empty+num1,cmp1);

break;

}

}

Full[k]=Full[num2-1];

num2--;

for(k=0;k<num1;k++)

{

for(l=0;l<num1;l++)

{

if(Empty[k].begin+Empty[k].partitionSize==Empty[l].begin)

{

Empty[k].partitionSize=Empty[k].partitionSize+Empty[l].partitionSize;

Empty[l]=Empty[num1-1];

num1--;

break;

}

}

}

}

if(run[j].flag==0&&(run[j].Processarrive<run[j].Processend)) //看空闲分区表中是否有足够空间分配给进程

{

run[j].flag=1;

for(l=0;l<num1;l++)

{

if(Empty[l].partitionSize>=run[j].ProcessNeed)

{

Full[num2].next=NULL;

Full[num2].id=run[j].ProcessName-65;

Full[num2].begin=Empty[l].begin;

Full[num2].partitionSize=run[j].ProcessNeed;

num2++;

Empty[l].begin=Empty[l].begin+run[j].ProcessNeed;

Empty[l].partitionSize=Empty[l].partitionSize-run[j].ProcessNeed;

cout<<"进程"<<run[j].ProcessName<<"得到P"<<Empty[l].id<<"的空间"<<" ";

sort(Empty,Empty+num1,cmp1);

break;

}

}

}

}

cout<<endl;

cout<<"此时刻所有空闲分区的首地址和大小"<<endl;

for(l=0;l<num1;l++)

{

cout<<"P"<<Empty[l].id<<" "<<Empty[l].begin<<" "<<Empty[l].partitionSize<<endl;

}

cout<<"此时刻所有已分配分区的首地址和大小"<<endl;

for(l=0;l<num2;l++)

{

char c=Full[l].id+65;

cout<<c<<" "<<Full[l].begin<<" "<<Full[l].partitionSize<<endl;

}

cout<<endl;

}

}

void WorstFit()

{

cout<<"WorstFit"<<endl;

initial();

sort(Empty,Empty+num1,cmp2);//按进程大小升序排序

int count=0;//统计进程的完成个数

int t=0;//统计正在运行的进程个数

int i,j,l,k;

for(i=0;count!=ProcessNum;i++)//不考虑空间不足的情况

{

cout<<"在"<<i<<"时刻"<<" ";

while(i==PCB.top().Processarrive&&PCB.empty()!=1)

{

run[t++]=PCB.top();

PCB.pop();

cout<<"进程"<<run[t-1].ProcessName<<"到达"<<" ";

//某个进程到达

}

for(j=0;j<t;j++)

{

if(run[j].Processend==i) //回收进程已被分配空间

{

count++;

cout<<"进程"<<run[j].ProcessName<<"完成"<<" ";

for(k=0;k<num2;k++)

{

if(Full[k].id+65==run[j].ProcessName)

break;

}

for(l=0;l<num1;l++)

{

if(Full[k].begin==Empty[l].begin+Empty[l].partitionSize)

{

Empty[l].partitionSize=Empty[l].partitionSize+Full[k].partitionSize;

sort(Empty,Empty+num1,cmp2);

break;

}else if(Full[k].begin+Full[k].partitionSize==Empty[l].begin)

{

Empty[l].begin=Full[k].begin;

Empty[l].partitionSize=Empty[l].partitionSize+Full[k].partitionSize;

sort(Empty,Empty+num1,cmp2);

break;

}else if(l==num1-1)

{

num1++;

Empty[num1-1].partitionSize=Full[k].partitionSize;

Empty[num1-1].begin=Full[k].begin;

Empty[num1-1].next=NULL;

Empty[num1-1].id=num1-1;

sort(Empty,Empty+num1,cmp2);

break;

}

}

Full[k]=Full[num2-1];

num2--;

for(k=0;k<num1;k++)

{

for(l=0;l<num1;l++)

{

if(Empty[k].begin+Empty[k].partitionSize==Empty[l].begin)

{

Empty[k].partitionSize=Empty[k].partitionSize+Empty[l].partitionSize;

Empty[l]=Empty[num1-1];

num1--;

break;

}

}

}

}

if(run[j].flag==0&&(run[j].Processarrive<run[j].Processend)) //看空闲分区表中是否有足够空间分配给进程

{

run[j].flag=1;

if(Empty[0].partitionSize>=run[j].ProcessNeed)

{

Full[num2].next=NULL;

Full[num2].id=run[j].ProcessName-65;

Full[num2].begin=Empty[0].begin;

Full[num2].partitionSize=run[j].ProcessNeed;

num2++;

Empty[0].begin=Empty[0].begin+run[j].ProcessNeed;

Empty[0].partitionSize=Empty[0].partitionSize-run[j].ProcessNeed;

cout<<"进程"<<run[j].ProcessName<<"得到P"<<Empty[0].id<<"的空间"<<" ";

sort(Empty,Empty+num1,cmp2);

break;

}

}

}

cout<<endl;

cout<<"此时刻所有空闲分区的首地址和大小"<<endl;

for(l=0;l<num1;l++)

{

cout<<"P"<<Empty[l].id<<" "<<Empty[l].begin<<" "<<Empty[l].partitionSize<<endl;

}

cout<<"此时刻所有已分配分区的首地址和大小"<<endl;

for(l=0;l<num2;l++)

{

char c=Full[l].id+65;

cout<<c<<" "<<Full[l].begin<<" "<<Full[l].partitionSize<<endl;

}

cout<<endl;

}

}

void choose\_Algorithm()

{

cout<<"请输入正确的选择“1-首次适应算法 2-循环首次适应算法 3-最佳适应算法 4-最坏适应算法 0-退出”"<<endl;

cin>>choose;

cout<<endl;

if (choose==1)

{

FirstFit();

choose\_Algorithm();

}

else if(choose==2)

{

NextFit();

choose\_Algorithm();

}

else if(choose==3){

BestFit();

choose\_Algorithm();

}

else if(choose==4){

WorstFit();

choose\_Algorithm();

}

else if(choose==0){

exit(0);

}

else

{

cout<<"请输入正确的选择1-首次适应算法 2-循环首次适应算法 3-最佳适应算法 4-最坏适应算法WF 0-退出"<<endl;

cout<<"------------------------------------------------------"<<endl;

choose\_Algorithm(); //递归

}

}

void initial(){ //初始化函数

int i,j;

Emptylist=NULL;

Fulllist=NULL;

for (i=0;i<ProcessNum;i++){

for (j=0;j<PartitionNum;j++){

NameProcessToPartition[i][j] =NULL;

}

}

for(i=PartitionNum-1;i>=0;i--)

{

sortNeed \*p;

if((p = (sortNeed \*)malloc(sizeof(sortNeed)))==NULL)

{

perror("malloc");

exit(1);

}

p->id =i;

p->begin =begin[i];

p->partitionSize=FreePartition[i];

p->next=Emptylist;

Emptylist=p;

}

for(i=0;i<PartitionNum;i++)

{

Empty[i].id=i;

Empty[i].begin=begin[i];

Empty[i].next=NULL;

Empty[i].partitionSize=FreePartition[i];

}

num1=PartitionNum;

num2=0;

for(i=0;i<ProcessNum;i++)

{

Process t;

t.ProcessName=ProcessName[i];

t.ProcessNeed=ProcessNeed[i];

t.Processarrive=Processarrive[i];

t.Processend=Processend[i];

t.flag=0;

PCB.push(t);

}

}

# 四、调试分析和思考

【(1) 调试过程中遇到的问题是如何解决的以及对设计与实现中关键点的回顾讨论和分析；(2) 算法的时空分析(包括基本操作和其他算法的时间复杂度和空间复杂度的分析)和改进设想；(3) 经验和体会等。】

1.首次适应算法（first Fit,FF）

FF算法是以空闲链的首地址递增顺序组织起来，当提出分配需求时，遍历组织好的空白链，找到第一个空间大于等于分配需求的空白分配块分配。若遍历一遍都未找到满足需求的空白块，则分配失败。

优点：该算法倾向于优先利用地址部分的空闲块，从而保留了高地址部分的空闲块，则高地址部分就有可能留有大容量的内存块，为大需求的作业创造了条件

缺点：该算法每次都是从低地址找起，导致其低地址留下了许多无法使用的外部碎片，降低了后续查找的效率。

2.循环首次适用算法（next fit ,NF）

NF算法在FF算法的基础上，针对其查找效率降低的缺点进行改进，不改变空白链的组织方式，只改变你查找方式，每次查找的起始位置是从上次查找位置的下一个位置开始，而不是从头查找，当循环查找一遍之后仍未找到满足需求的空白内存块时分配失败！这样就避免了对外部碎片的查找浪费。

优点：使内存分配在内存中更加均匀，相对于首次适应算法来说查找效率更高。

缺点：由于分配均匀，使得内存中缺乏大的空闲内存块，当后续出现大内存需求的作业时无法满足。

3.最佳适应算法（best fit ,BF）

“最佳”指的是大小合适，最接近。空白链以容量大小的顺序组织，每次遍历空白链查找第一个能满足需求的空白块进行分配，这样就一定程度减少了外部碎片的大小。也避免了“大材小用”

优点：每次分割剩余的空间总是最小的，减少了外部碎片产生的大小。

缺点：空间利用率来看，NF算法确实是物尽其用，但是每次分割留下的都是难以利用的外部碎片，又降低了查找效率

4.最坏适应算法（worst fit,WF）

最坏适应算法和最佳使用算法的空白块选择策略刚好相反，它在扫描整个空表链时，总是挑选一个最大空闲块，从中分割需求的内存块，实际上，这样的算法未必是最坏的

优点：可使剩下的空闲区不至于太小，产生外部碎片的可能性更小，对中小作业有利。查找效率高，只找最大的，若最大不满足，就直接失败。

缺点：可能会导致空白存储链中缺少大容量的空白内存块，当大容量作业进入时无法满足。

参数G: 其中由于在分配过程中会产生各种容量极小，无法利用，但其存在又会增加查找效率。于是系统设定一个定值参数G,当某次分配后该内存块剩余的内存容量小于参数G，为避免产生外部碎片降低查找效率，于是将该内存块剩余所有空间全部分配给该进程。

内部碎片：内部碎片是分配给进程，但进程未使用到的空闲内存；但不会再分配给其他进程

外部碎片：内存分配后剩余的空闲内存块，由于容量太小而无法使用，成为外部碎片

# 五、测试数据与结果

【列出你的测试结果，明确输入和输出数据。测试数据应该完整和严格，可以直接贴结果图。】









