**操作系统实验**

**实验报告**

题目：[实验四 动态分区分配算法]

**专业：[软件工程]**

**年级：[2017级]**

**姓名：[陈思翔]**

**学号：[1725121003]**

目录

[一、需求分析 4](#_Toc27926826)

[1、程序设计的任务和目的 4](#_Toc27926827)

[2、输入的形式和输入值的范围 4](#_Toc27926828)

[3、输出的形式 4](#_Toc27926829)

[4、程序所能达到的功能 4](#_Toc27926830)

[5、测试数据，包括正确的输入及其输出结果和含有错误的输入及其输出结果 4](#_Toc27926831)

[正确用例 4](#_Toc27926832)

[错误用例（输入值过大，分区大小无法处理分配，循环首次适应分区（CycleFirstPartition）算法会进入死循环） 5](#_Toc27926833)

[二、概要设计 6](#_Toc27926834)

[1、抽象数据类型的定义 6](#_Toc27926835)

[2、主程序的流程 6](#_Toc27926836)

[3、各程序模块之间的层次(调用)关系 6](#_Toc27926837)

[三、详细设计 7](#_Toc27926838)

[1、FirstPartition算法 8](#_Toc27926839)

[2、CycleFirstPartition算法 8](#_Toc27926840)

[3、BestPartition算法 9](#_Toc27926841)

[4、WorstPartition算法 10](#_Toc27926842)

[四、调试分析 11](#_Toc27926843)

[调试过程中遇到的问题以及解决方法，设计与实现的回顾讨论和分析 11](#_Toc27926844)

[算法的性能分析(包括基本操作和其它算法的时间复杂度和空间复杂度的分析)及其改进设想 11](#_Toc27926845)

[性能分析 11](#_Toc27926846)

[改进设想 12](#_Toc27926847)

[五、用户使用说明 12](#_Toc27926848)

[使用说明 12](#_Toc27926849)

[六、测试结果 12](#_Toc27926850)

[测试结果，包括输入和输出 12](#_Toc27926851)

[七、附录 15](#_Toc27926852)

[带注释的源程序 15](#_Toc27926853)

# 一、需求分析

## 1、程序设计的任务和目的

通过这次实验，加深对动态分区分配算法的理解，进一步掌握首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法的实现方法。

## 2、输入的形式和输入值的范围

空闲分区个数n，空闲分区大小P1, … ,Pn，进程个数m，进程需要的分区大小S1, … ,Sm，算法选择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。

## 3、输出的形式

最终内存空闲分区的分配情况。

## 4、程序所能达到的功能

模拟四种动态分区分配算法：首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法的工作过程。假设内存中空闲分区个数为n，空闲分区大小分别为P1, … ,Pn，在动态分区分配过程中需要分配的进程个数为m（m≤n），它们需要的分区大小分别为S1, … ,Sm，分别利用四种动态分区分配算法将m个进程放入n个空闲分区，给出进程在空闲分区中的分配情况。

## 5、测试数据，包括正确的输入及其输出结果和含有错误的输入及其输出结果

### 正确用例

#### 输入

请输入空闲分区个数PartitionNum：9  
请输入空闲分区1大小FreePartition[1]：16  
请输入空闲分区2大小FreePartition[2]：16  
请输入空闲分区3大小FreePartition[3]：8  
请输入空闲分区4大小FreePartition[4]：32  
请输入空闲分区5大小FreePartition[5]：64  
请输入空闲分区6大小FreePartition[6]：32  
请输入空闲分区7大小FreePartition[7]：8  
请输入空闲分区8大小FreePartition[8]：16  
请输入空闲分区9大小FreePartition[9]：64  
请输入进程个数ProcessNum：6  
请输入进程1需要的分区大小progress[1].ProcessNeed：7  
请输入进程2需要的分区大小progress[2].ProcessNeed：18  
请输入进程3需要的分区大小progress[3].ProcessNeed：9  
请输入进程4需要的分区大小progress[4].ProcessNeed：20  
请输入进程5需要的分区大小progress[5].ProcessNeed：35  
请输入进程6需要的分区大小progress[6].ProcessNeed：8  
  
  
请选择想要先使用的算法（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：1

#### 输出

您选择的是1-首次适应分区算法（FirstPartition）  
  
进程分配情况如下表：  
 Partition 1 Partition 2 Partition 3 Partition 4 Partition 5 Partition 6 Partition 7 Partition 8 Partition 9  
Assigned 16 8 0 18 55 0 0 0 0  
Available 0 8 8 14 9 32 8 16 64  
 Process 1 Process 2 Process 3 Process 4 Process 5 Process 6  
ProcessNum 1 4 1 5 5 2

### 错误用例（输入值过大，分区大小无法处理分配，循环首次适应分区（CycleFirstPartition）算法会进入死循环）

#### 输入

请输入空闲分区个数PartitionNum：3  
请输入空闲分区1大小FreePartition[1]：32  
请输入空闲分区2大小FreePartition[2]：64  
请输入空闲分区3大小FreePartition[3]：128  
请输入进程个数ProcessNum：3  
请输入进程1需要的分区大小progress[1].ProcessNeed：50  
请输入进程2需要的分区大小progress[2].ProcessNeed：100  
请输入进程3需要的分区大小progress[3].ProcessNeed：150

#### 输出

您选择的是2-循环首次适应分区算法（CycleFirstPartition）  
  
Process finished with exit code 1

# 二、概要设计

## 1、抽象数据类型的定义

int FreePartition[MaxNumber];//空闲分区大小  
int PartitionNum;//空闲分区个数  
int ProcessNum;//进程个数  
int isAlgorithm;//算法选择  
  
//定义进程的数据结构  
typedef struct {  
 int FirstPartition;//首次适应分区  
 int CycleFirstPartition;//循环首次适应分区  
 int BestPartition;//最佳适应分区  
 int WorstPartition;//最坏适应分区  
 int ProcessNeed;//进程需要的分区大小  
} Progress;

## 2、主程序的流程

int main() {  
 PartitionAlgorithm partitionAlgorithm{};  
  
 partitionAlgorithm.Input();  
  
 return 0;  
}

## 3、各程序模块之间的层次(调用)关系

int main() {  
 virtualMemoryPageReplacementAlgorithm virtualMemoryPageReplacementAlgorithm{};  
  
 virtualMemoryPageReplacementAlgorithm.Input();  
  
 return 0;  
}  
  
//输入函数调用InputAlgorithm函数选择输入函数  
void Input() {  
 ···  
 InputAlgorithm();  
}  
  
//调用IsAlgorithm函数进行算法存储确认  
void InputAlgorithm() {  
 ···  
 IsAlgorithm();  
}  
  
//算法确认后按照确认结果调用不同算法函数，若确认失败，要求重新输入  
void IsAlgorithm() {  
 ···  
 switch (isAlgorithm) {  
 case 1:  
 AlgorithmFirstPartition();  
 case 2:  
 AlgorithmCycleFirstPartitionS();  
 case 3:  
 AlgorithmBestPartition();  
 case 4:  
 AlgorithmWorstPartition();  
 default:  
 InputAlgorithm();  
 }  
}  
  
//算法函数仅以FirstPartition为例，在完成算法处理后调用Print函数输出本算法运算结果，并调用NextAlgorithm函数询问下一算法使用  
void AlgorithmFirstPartition() {  
 ···  
  
 //验证方法：验证空闲分区是否足够分配  
 Test(progressCopy);  
  
 Print(progressCopy, freePartition);  
  
 NextAlgorithm();  
}  
  
//输出最终内存空闲分区的分配情况  
void Print(Progress pg[MaxNumber], int fp[MaxNumber]) {  
 ···  
}  
  
//询问后续，若有则重新调用算法确认函数循环流程，若触发终止条件结束进程  
void NextAlgorithm() {  
 ···  
 IsAlgorithm();  
}

# 三、详细设计

**实现程序模块的具体算法**

## 1、FirstPartition算法

//调用FirstPartition算法进行调度计算  
void AlgorithmFirstPartition() {  
 //复制进程  
 Progress progressCopy[MaxNumber];  
 for (int i = 1; i <= ProcessNum; i++) {  
 progressCopy[i] = progress[i];  
 }  
  
 //复制空闲分区大小  
 int freePartition[MaxNumber];//空闲分区大小  
 for (int i = 1; i <= PartitionNum; i++) {  
 freePartition[i] = FreePartition[i];  
 }  
  
 //进程分配  
 for (int i = 1; i <= ProcessNum; i++) {  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[j] >= progressCopy[i].ProcessNeed) {  
 freePartition[j] -= progressCopy[i].ProcessNeed;  
 progressCopy[i].FirstPartition = j;  
 break;  
 }  
 }  
 }  
  
 ···  
}

## 2、CycleFirstPartition算法

//调用CycleFirstPartition算法进行调度计算  
void AlgorithmCycleFirstPartitionS() {  
 //复制进程，如第一个算法，不赘述  
 ···  
  
 //复制空闲分区大小，如第一个算法，不赘述  
 ···  
  
 //进程分配  
 int i = 1;  
 while (i <= ProcessNum) {  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[j] >= progressCopy[i].ProcessNeed) {  
 freePartition[j] -= progressCopy[i].ProcessNeed;  
 progressCopy[i].CycleFirstPartition = j;  
 i++;  
 if (i > ProcessNum) {  
 break;  
 }  
 }  
 }  
 if (i > ProcessNum) {  
 break;  
 }  
 }  
  
 ···  
}

## 3、BestPartition算法

//调用BestPartition算法进行调度计算  
void AlgorithmBestPartition() {  
 //复制进程，如第一个算法，不赘述  
 ···  
  
 //复制空闲分区大小，如第一个算法，不赘述  
 ···  
  
 //进程分配吗，为此算法设计了辅助算法，用于排序分区大小，得到从小到大分区对应的index  
 for (int i = 1; i <= ProcessNum; i++) {  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[AscendingBubbleSort(freePartition, j)] >= progressCopy[i].ProcessNeed) {  
 progressCopy[i].BestPartition = AscendingBubbleSort(freePartition, j);  
 freePartition[AscendingBubbleSort(freePartition, j)] -= progressCopy[i].ProcessNeed;  
 break;  
 }  
 }  
 }  
 ···  
}  
  
//辅助算法：升序冒泡排序算法，将分区剩余大小从小到大冒泡排序，返回该index排序的值  
int AscendingBubbleSort(const int freePartition[MaxNumber], int index) {  
 //复制空闲分区  
 int freePartitionCopy[MaxNumber];//空闲分区大小  
 for (int i = 1; i <= PartitionNum; i++) {  
 freePartitionCopy[i] = freePartition[i];  
 }  
  
 //将分区剩余大小从小到大冒泡排序  
 for (int i = 1; i <= PartitionNum; i++) {  
 for (int j = 1; j <= PartitionNum - i; j++) {  
 if (freePartitionCopy[j] > freePartitionCopy[j + 1]) {  
 int temp = freePartitionCopy[j];  
 freePartitionCopy[j] = freePartitionCopy[j + 1];  
 freePartitionCopy[j + 1] = temp;  
 }  
 }  
 }  
  
 //查找该最短服务时间对应的进程号  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[j] == freePartitionCopy[index]) {  
 return j;  
 }  
 }  
  
 return 0;  
}

## 4、WorstPartition算法

//调用WorstPartition算法进行调度计算  
void AlgorithmWorstPartition() {  
 //复制进程，如第一个算法，不赘述  
 ···  
  
 //复制空闲分区大小，如第一个算法，不赘述  
 ···  
  
 //进程分配为此算法设计了辅助算法，用于排序分区大小，得到从大到小分区对应的index队列  
 for (int i = 1; i <= ProcessNum; i++) {  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[DescendingBubbleSort(freePartition, j)] >= progressCopy[i].ProcessNeed) {  
 progressCopy[i].WorstPartition = DescendingBubbleSort(freePartition, j);  
 freePartition[DescendingBubbleSort(freePartition, j)] -= progressCopy[i].ProcessNeed;  
 break;  
 }  
 }  
 }  
 ···  
}  
  
//辅助算法：降序冒泡排序算法，将分区剩余大小从大到小冒泡排序，返回该index排序的值  
int DescendingBubbleSort(const int freePartition[MaxNumber], int index) {  
 //复制空闲分区  
 int freePartitionCopy[MaxNumber];//空闲分区大小  
 for (int i = 1; i <= PartitionNum; i++) {  
 freePartitionCopy[i] = freePartition[i];  
 }  
  
 //将分区剩余大小从大到小冒泡排序  
 for (int i = 1; i <= PartitionNum; i++) {  
 for (int j = 1; j <= PartitionNum - i; j++) {  
 if (freePartitionCopy[j] < freePartitionCopy[j + 1]) {  
 int temp = freePartitionCopy[j];  
 freePartitionCopy[j] = freePartitionCopy[j + 1];  
 freePartitionCopy[j + 1] = temp;  
 }  
 }  
 }  
  
 //查找该最短服务时间对应的进程号  
 for (int j = 1; j <= PartitionNum; j++) {  
 if (freePartition[j] == freePartitionCopy[index]) {  
 return j;  
 }  
 }  
  
 return 0;  
}

# 四、调试分析

## 调试过程中遇到的问题以及解决方法，设计与实现的回顾讨论和分析

|  |  |  |
| --- | --- | --- |
| 问题 | 描述 | 解决方法 |
| 脏读问题 | 为实现同一次数据输入，测试各算法运算结果的功能，各算法不能改变输入数组的存储位置 | 每个算法伊始，copy一分分区及进程序列进行运算 |

## 算法的性能分析(包括基本操作和其它算法的时间复杂度和空间复杂度的分析)及其改进设想

### 性能分析

|  |  |  |
| --- | --- | --- |
| 算法 | 时间复杂度 | 空间复杂度 |
| FirstPartition算法 | T(n) = O(n2) | S(n) = O(n) |
| CycleFirstPartition算法 | T(n) = O(n2) | S(n) = O(n) |
| BestPartition算法 | T(n) = O(n2) | S(n) = O(n) |
| AscendingBubbleSort升序辅助算法 | T(n) = O(n2) | S(n) = O(n) |
| WorstPartition算法 | T(n) = O(n2) | S(n) = O(n) |
| DescendingBubbleSort降序辅助算法 | T(n) = O(n2) | S(n) = O(n) |
| Test算法验证空闲分区是否足够分配 | T(n) = O(n2) | S(n) = O(n) |
| NextAlgorithm算法 | T(n) = O(1) | S(n) = O(1) |

### 改进设想

简化辅助算法

# 五、用户使用说明

## 使用说明

* 控制台会提示要求用户进行输入，按提示输入内容即可
* 不要输入超过分区大小的进程，分区大小无法处理分配，循环首次适应分区（CycleFirstPartition）算法会进入死循环
* 选择算法输入完成后将会输出动态分区分配过程及模拟列表
* 可根据需要选择是否使用其他算法进行动态分区分配或退出

# 六、测试结果

## 测试结果，包括输入和输出

请输入空闲分区个数PartitionNum：9  
请输入空闲分区1大小FreePartition[1]：16  
请输入空闲分区2大小FreePartition[2]：16  
请输入空闲分区3大小FreePartition[3]：8  
请输入空闲分区4大小FreePartition[4]：32  
请输入空闲分区5大小FreePartition[5]：64  
请输入空闲分区6大小FreePartition[6]：32  
请输入空闲分区7大小FreePartition[7]：8  
请输入空闲分区8大小FreePartition[8]：16  
请输入空闲分区9大小FreePartition[9]：64  
请输入进程个数ProcessNum：6  
请输入进程1需要的分区大小progress[1].ProcessNeed：7  
请输入进程2需要的分区大小progress[2].ProcessNeed：18  
请输入进程3需要的分区大小progress[3].ProcessNeed：9  
请输入进程4需要的分区大小progress[4].ProcessNeed：20  
请输入进程5需要的分区大小progress[5].ProcessNeed：35  
请输入进程6需要的分区大小progress[6].ProcessNeed：8  
  
  
请选择想要先使用的算法（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：1  
  
您选择的是1-首次适应分区算法（FirstPartition）  
  
进程分配情况如下表：  
 Partition 1 Partition 2 Partition 3 Partition 4 Partition 5 Partition 6 Partition 7 Partition 8 Partition 9  
Assigned 16 8 0 18 55 0 0 0 0  
Available 0 8 8 14 9 32 8 16 64  
 Process 1 Process 2 Process 3 Process 4 Process 5 Process 6  
ProcessNum 1 4 1 5 5 2  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：2  
  
您选择的是2-循环首次适应分区算法（CycleFirstPartition）  
  
进程分配情况如下表：  
 Partition 1 Partition 2 Partition 3 Partition 4 Partition 5 Partition 6 Partition 7 Partition 8 Partition 9  
Assigned 15 0 0 18 9 20 0 0 35  
Available 1 16 8 14 55 12 8 16 29  
 Process 1 Process 2 Process 3 Process 4 Process 5 Process 6  
ProcessNum 1 4 5 6 9 1  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：3  
  
您选择的是3-最佳适应分区算法（BestPartition）  
  
进程分配情况如下表：  
 Partition 1 Partition 2 Partition 3 Partition 4 Partition 5 Partition 6 Partition 7 Partition 8 Partition 9  
Assigned 0 0 7 27 35 20 8 0 0  
Available 16 16 1 5 29 12 0 16 64  
 Process 1 Process 2 Process 3 Process 4 Process 5 Process 6  
ProcessNum 3 4 4 6 5 7  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：4  
  
您选择的是4-最坏适应分区算法（WorstPartition）  
  
进程分配情况如下表：  
 Partition 1 Partition 2 Partition 3 Partition 4 Partition 5 Partition 6 Partition 7 Partition 8 Partition 9  
Assigned 0 0 0 8 36 0 0 0 53  
Available 16 16 8 24 28 32 8 16 11  
 Process 1 Process 2 Process 3 Process 4 Process 5 Process 6  
ProcessNum 5 9 5 5 9 4  
  
请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：0  
  
Process finished with exit code 0

# 七、附录

## 带注释的源程序

#include <iostream>   
#include <iomanip>   
   
using namespace std;   
#define MaxNumber 100   
   
class PartitionAlgorithm {   
public:   
   
 int FreePartition[MaxNumber];//空闲分区大小   
 int PartitionNum;//空闲分区个数   
 int ProcessNum;//进程个数   
 int isAlgorithm;//算法选择   
   
 //定义进程的数据结构   
 typedef struct {   
 int FirstPartition;//首次适应分区   
 int CycleFirstPartition;//循环首次适应分区   
 int BestPartition;//最佳适应分区   
 int WorstPartition;//最坏适应分区   
 int ProcessNeed;//进程需要的分区大小   
 } Progress;   
   
 Progress progress[MaxNumber];   
   
 //输入空闲分区数、空闲的分区大小、进程数、进程需要的分区大小   
 void Input() {   
 cout << "请输入空闲分区个数PartitionNum：";   
 cin >> PartitionNum;   
   
 for (int i = 1; i <= PartitionNum; i++) {   
 cout << "请输入空闲分区" << i << "大小" << "FreePartition[" << i << "]：";   
 cin >> FreePartition[i];   
 }   
   
 cout << "请输入进程个数ProcessNum：";   
 cin >> ProcessNum;   
   
 for (int i = 1; i <= ProcessNum; i++) {   
 cout << "请输入进程" << i << "需要的分区大小" << "progress[" << i << "].ProcessNeed：";   
 cin >> progress[i].ProcessNeed;   
 }   
   
 cout << endl;   
   
 InputAlgorithm();   
 }   
   
 //获取算法选择输入   
 void InputAlgorithm() {   
 cout << endl   
 << "请选择想要先使用的算法（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：";   
 cin >> isAlgorithm;   
 IsAlgorithm();   
 }   
   
 //算法存储确认   
 void IsAlgorithm() {   
 switch (isAlgorithm) {   
 case 1:   
 cout << endl << "您选择的是1-首次适应分区算法（FirstPartition）" << endl;   
 AlgorithmFirstPartition();   
 break;   
 case 2:   
 cout << endl << "您选择的是2-循环首次适应分区算法（CycleFirstPartition）" << endl;   
 AlgorithmCycleFirstPartitionS();   
 break;   
 case 3:   
 cout << endl << "您选择的是3-最佳适应分区算法（BestPartition）" << endl;   
 AlgorithmBestPartition();   
 break;   
 case 4:   
 cout << endl << "您选择的是4-最坏适应分区算法（WorstPartition）" << endl;   
 AlgorithmWorstPartition();   
 break;   
 default:   
 cout << "算法值：" << isAlgorithm   
 << "有误,请重新输入正确的算法类型（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））"   
 << endl;   
 InputAlgorithm();   
 }   
 }   
   
 //输出空闲分区剩余情况、各进程分配情况   
 void Print(Progress pg[MaxNumber], int fp[MaxNumber]) {   
   
 cout << endl << left << setw(15) << "进程分配情况如下表：" << endl;   
   
 //分区状况   
 cout << left << setw(15) << "";   
 for (int i = 1; i <= PartitionNum; i++) {   
 cout << right << setw(10) << "Partition" << setw(2) << i;   
 }   
 cout << endl << left << setw(15) << "Assigned";   
 for (int i = 1; i <= PartitionNum; i++) {   
 cout << right << setw(12) << FreePartition[i] - fp[i];   
 }   
 cout << endl << left << setw(15) << "Available";   
 for (int i = 1; i <= PartitionNum; i++) {   
 cout << right << setw(12) << fp[i];   
 }   
 cout << endl;   
   
 //进程分配状况   
 cout << left << setw(15) << "";   
 for (int i = 1; i <= ProcessNum; i++) {   
 cout << right << setw(10) << "Process" << setw(2) << i;   
 }   
 cout << endl << left << setw(15) << "ProcessNum";   
 for (int i = 1; i <= ProcessNum; i++) {   
 switch (isAlgorithm) {   
 case 1:   
 cout << right << setw(12) << pg[i].FirstPartition;   
 break;   
 case 2:   
 cout << right << setw(12) << pg[i].CycleFirstPartition;   
 break;   
 case 3:   
 cout << right << setw(12) << pg[i].BestPartition;   
 break;   
 case 4:   
 cout << right << setw(12) << pg[i].WorstPartition;   
 break;   
 }   
 }   
 cout << endl;   
   
 }   
   
 //调用FirstPartition算法进行调度计算   
 void AlgorithmFirstPartition() {   
 //复制进程   
 Progress progressCopy[MaxNumber];   
 for (int i = 1; i <= ProcessNum; i++) {   
 progressCopy[i] = progress[i];   
 }   
   
 //复制空闲分区大小   
 int freePartition[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartition[i] = FreePartition[i];   
 }   
   
 //进程分配   
 for (int i = 1; i <= ProcessNum; i++) {   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[j] >= progressCopy[i].ProcessNeed) {   
 freePartition[j] -= progressCopy[i].ProcessNeed;   
 progressCopy[i].FirstPartition = j;   
 break;   
 }   
 }   
 }   
   
 Test(progressCopy);   
   
 Print(progressCopy, freePartition);   
   
 NextAlgorithm();   
 }   
   
 //调用CycleFirstPartition算法进行调度计算   
 void AlgorithmCycleFirstPartitionS() {   
 //复制进程   
 Progress progressCopy[MaxNumber];   
 for (int i = 1; i <= ProcessNum; i++) {   
 progressCopy[i] = progress[i];   
 }   
   
 //复制空闲分区大小   
 int freePartition[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartition[i] = FreePartition[i];   
 }   
   
 //进程分配   
 int i = 1;   
 while (i <= ProcessNum) {   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[j] >= progressCopy[i].ProcessNeed) {   
 freePartition[j] -= progressCopy[i].ProcessNeed;   
 progressCopy[i].CycleFirstPartition = j;   
 i++;   
 if (i > ProcessNum) {   
 break;   
 }   
 }   
 }   
 if (i > ProcessNum) {   
 break;   
 }   
 }   
   
 Test(progressCopy);   
   
 Print(progressCopy, freePartition);   
   
 NextAlgorithm();   
 }   
   
 //调用BestPartition算法进行调度计算   
 void AlgorithmBestPartition() {   
 //复制进程   
 Progress progressCopy[MaxNumber];   
 for (int i = 1; i <= ProcessNum; i++) {   
 progressCopy[i] = progress[i];   
 }   
   
 //复制空闲分区大小   
 int freePartition[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartition[i] = FreePartition[i];   
 }   
   
 //进程分配   
 for (int i = 1; i <= ProcessNum; i++) {   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[AscendingBubbleSort(freePartition, j)] >= progressCopy[i].ProcessNeed) {   
 progressCopy[i].BestPartition = AscendingBubbleSort(freePartition, j);   
 freePartition[AscendingBubbleSort(freePartition, j)] -= progressCopy[i].ProcessNeed;   
 break;   
 }   
 }   
 }   
   
 Test(progressCopy);   
   
 Print(progressCopy, freePartition);   
   
 NextAlgorithm();   
   
   
 }   
   
 //辅助算法：升序冒泡排序算法，将分区剩余大小从小到大冒泡排序，返回该index排序的值   
 int AscendingBubbleSort(const int freePartition[MaxNumber], int index) {   
 //复制空闲分区   
 int freePartitionCopy[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartitionCopy[i] = freePartition[i];   
 }   
   
 //将分区剩余大小从小到大冒泡排序   
 for (int i = 1; i <= PartitionNum; i++) {   
 for (int j = 1; j <= PartitionNum - i; j++) {   
 if (freePartitionCopy[j] > freePartitionCopy[j + 1]) {   
 int temp = freePartitionCopy[j];   
 freePartitionCopy[j] = freePartitionCopy[j + 1];   
 freePartitionCopy[j + 1] = temp;   
 }   
 }   
 }   
   
 //查找该最短服务时间对应的进程号   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[j] == freePartitionCopy[index]) {   
 return j;   
 }   
 }   
   
 return 0;   
 }   
   
 //调用WorstPartition算法进行调度计算   
 void AlgorithmWorstPartition() {   
 //复制进程   
 Progress progressCopy[MaxNumber];   
 for (int i = 1; i <= ProcessNum; i++) {   
 progressCopy[i] = progress[i];   
 }   
   
 //复制空闲分区大小   
 int freePartition[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartition[i] = FreePartition[i];   
 }   
   
 //进程分配   
 for (int i = 1; i <= ProcessNum; i++) {   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[DescendingBubbleSort(freePartition, j)] >= progressCopy[i].ProcessNeed) {   
 progressCopy[i].WorstPartition = DescendingBubbleSort(freePartition, j);   
 freePartition[DescendingBubbleSort(freePartition, j)] -= progressCopy[i].ProcessNeed;   
 break;   
 }   
 }   
 }   
   
 Test(progressCopy);   
   
 Print(progressCopy, freePartition);   
   
 NextAlgorithm();   
 }   
   
 //辅助算法：降序冒泡排序算法，将分区剩余大小从大到小冒泡排序，返回该index排序的值   
 int DescendingBubbleSort(const int freePartition[MaxNumber], int index) {   
 //复制空闲分区   
 int freePartitionCopy[MaxNumber];//空闲分区大小   
 for (int i = 1; i <= PartitionNum; i++) {   
 freePartitionCopy[i] = freePartition[i];   
 }   
   
 //将分区剩余大小从大到小冒泡排序   
 for (int i = 1; i <= PartitionNum; i++) {   
 for (int j = 1; j <= PartitionNum - i; j++) {   
 if (freePartitionCopy[j] < freePartitionCopy[j + 1]) {   
 int temp = freePartitionCopy[j];   
 freePartitionCopy[j] = freePartitionCopy[j + 1];   
 freePartitionCopy[j + 1] = temp;   
 }   
 }   
 }   
   
 //查找该最短服务时间对应的进程号   
 for (int j = 1; j <= PartitionNum; j++) {   
 if (freePartition[j] == freePartitionCopy[index]) {   
 return j;   
 }   
 }   
   
 return 0;   
 }   
   
 //验证方法：验证空闲分区是否足够分配   
 void Test(Progress pg[MaxNumber]) {   
 switch (isAlgorithm) {   
 case 1:   
 for (int i = 1; i <= ProcessNum; i++) {   
 if (pg[i].FirstPartition < 0) {   
 cout << "空闲分区不足本算法分配" << endl;   
 exit(0);   
 }   
 }   
 break;   
 case 2:   
 for (int i = 1; i <= ProcessNum; i++) {   
 if (pg[i].CycleFirstPartition < 0) {   
 cout << "空闲分区不足本算法分配" << endl;   
 exit(0);   
 }   
 }   
 break;   
 case 3:   
 for (int i = 1; i <= ProcessNum; i++) {   
 if (pg[i].BestPartition < 0) {   
 cout << "空闲分区不足本算法分配" << endl;   
 exit(0);   
 }   
 }   
 break;   
 case 4:   
 for (int i = 1; i <= ProcessNum; i++) {   
 if (pg[i].WorstPartition < 0) {   
 cout << "空闲分区不足本算法分配" << endl;   
 exit(0);   
 }   
 }   
 break;   
 }   
 }   
   
 //询问是否还要进行其余算法   
 void NextAlgorithm() {   
 cout << endl   
 << "请问是否还要进行其余算法，若是，请输入（1-4值）；若否，请输入任意字符（1-首次适应分区算法（FirstPartition），2-循环首次适应分区算法（CycleFirstPartition），3-最佳适应分区算法（BestPartition），4-最坏适应分区算法（WorstPartition））：";   
 cin >> isAlgorithm;   
 if (isAlgorithm != 1 && isAlgorithm != 2 && isAlgorithm != 3 && isAlgorithm != 4) {   
 return;   
 }   
 IsAlgorithm();   
 }   
};   
   
int main() {   
 PartitionAlgorithm partitionAlgorithm{};   
   
 partitionAlgorithm.Input();   
   
 return 0;   
}