

操作系统

实 验 三 报 告

学院： 计算机科学学院

专业： 网络工程1701

姓名： 胡玉琛41709040102

姓名： 何一鸣41709040107

1. 1. **实验题目**

动态分区分配算法

1. **实验目的**

动态分区分配是根据进程的实际需要，动态地为之分配内存空间，而在分配时，须按照一定的分配算法，从空闲分区表或空闲分区链中选出一分区分配给该作业。在本实验中运用了四种分配算法，分别是首次适应算法、循环首次适应算法、最坏适应算法、最佳适应算法。

1. **实验环境**

操作系统：Windows10家庭和学生版

环境：Microsoft Visual Studio 2013

1. **算法思想**
2. 数据结构
3. 分区开始地址startdress
4. 分区大小size
5. 分区状态state
6. 功能介绍
7. 首次适应算法

在首次适应算法中，是从已经建立好的数组顺序中查找，直到找到第一个大小能满足要求的空闲分区为止，然后再按照作业大小，从该分区中划出一块内存空间分配给请求者，余下的空间另开辟一块新的地址，大小为原来的大小减去作业大小，若查找结束都不能找到一个满足的分区，则此次内存分配失败。

1. 循环首次适应算法

该算法是由首次适应算法演变而成，在为进程分配内存空间时，不再是每次都从第一个空间开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到第一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业，为实现本算法，设置一个全局变量f，来控制循环查找，当f%N==0时，f=0；若查找结束都不能找到一个满足要求的分区，则此次内存分配失败。

1. 最坏适应算法

最坏适应分配算法时是每次为作业分配内存时，扫描整个数组，总能把满足条件的，又是最大的空闲分区分配给作业。

1. 最佳适应算法

该算法总是把既能满足要求，又是最小的空闲分区分配给作业。为了加速查找，该算法要求将所有的空闲区按其大小排序后，以递增顺序形成一个空白链。这样每次找到的第一个满足要求的空闲区，必然是最优的。孤立地看，该算法似乎是最优的，但事实上并不一定。因为每次分配后剩余的空间一定是最小的，在存储器中将留下许多难以利用的小空闲区。同时每次分配后必须重新排序，这也带来了一定的开销。特点：每次分配给文件的都是最合适该文件大小的分区。缺点：内存中留下许多难以利用的小的空闲区。

**五、实验内容**

1.输入分区数和大小、资源数和大小

void Enter(){

int i;

cout<<"Enter PartitionNum: ";

cin>>PartitionNum;

cout<<"Enter FreePartition: ";

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

cin>>FreePartition[i];

}

cout<<"Enter ProcessNum: ";

cin>>ProcessNum;

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

ProcessName[i]=i+65;

}

cout<<"Enter ProcessNeed: ";

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

cin>>ProcessNeed[i];

}

}

//初始化供4个算法使用

void initial(){

int i,j;

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

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

NameProcessToPartition[i][j] =NULL;

LeftFreePartition[j] = FreePartition[j];

}

}

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

LeftProcessNeed[i] = ProcessNeed[i];

}

}

//显示分区结果

void display(){

int i;

cout<<"ProcessName: "<<"\t\t";

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

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

}

cout<<endl;

cout<<"ProcessNeed： "<<"\t\t";

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

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

}

cout<<endl;

cout<<"PartitionNum: "<<"\t\t";

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

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

}

cout<<endl<<"FreePartition: "<<"\t\t";

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

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

}

cout<<endl<<"LeftFreePartition: "<<"\t";

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

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

}

cout<<endl<<"Partition of process:"<<endl;

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

for (int j = 0;j<ProcessNum;j++){

if (NameProcessToPartition[j][i]!=NULL){

cout<<NameProcessToPartition[j][i]<<": P"<<i+1<<endl;

}

}

}

}

2.首次适应算法FF

void FirstFit(){

cout<<"FF"<<endl;

initial();

int i,j;

for (i=0;i<ProcessNum;i++){ //遍历进程

for (j=0;j<PartitionNum;j++){ //每次从分区的首地址开始查找

//当系统内存分区足够大的时候，即分配给进程资源

if (LeftProcessNeed[i] <= LeftFreePartition[j] && LeftFreePartition!=0){

LeftFreePartition[j] -= LeftProcessNeed[i]; //扣除分配给进程的资源

LeftProcessNeed[i] = 0; //当内存分区足够才执行，当前进程大小置0

NameProcessToPartition[i][j] = ProcessName[i]; //存储各个进程所在的分区位置

break; //一个进程分区完后，立即进行下一个进程的判断

}

}

}

display();

}

3.循环首次适应算法NF

void NextFit(){

cout<<"NF"<<endl;

initial();

int i,nextPoint = 0;

bool isWhile;

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

isWhile = true;

while(isWhile){ //每次都从当前分区的下一个分区开始查找

if (LeftFreePartition[nextPoint] >= LeftProcessNeed[i]){

LeftFreePartition[nextPoint] -= LeftProcessNeed[i];

LeftProcessNeed[i] = 0;

NameProcessToPartition[i][nextPoint] = ProcessName[i];

nextPoint++;

if (nextPoint > PartitionNum - 1){

nextPoint = 0; //当j遍历到分区末尾的时候，返回首位置

}

isWhile = false;

}

else{

nextPoint++;

if (nextPoint > PartitionNum - 1)

nextPoint = 0; //当j遍历到分区末尾的时候，返回首位置

}

}

}

display();

}

4.最佳适应算法BF

void BestFit(){

//对分区大小进行排序,每次分配完一个进程的内存大小后，重新排序

cout<<"BF"<<endl;

initial();

int i,j,s,t;

sortNeed best[MAXNUMBER];

sortNeed temp;

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

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

best[i].id = i;

}

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

//冒泡排序(每次分配完一个进程后，都需要重新排序)

for (s=0;s < PartitionNum - 1;s++){

for (t = s;t < PartitionNum - 1;t++){

if (best[s].partitionSize > best[t].partitionSize){

temp = best[s];

best[s] = best[t];

best[t] = temp;

}

}

}

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

if (LeftProcessNeed[i] <= best[j].partitionSize){

best[j].partitionSize -= LeftProcessNeed[i];

LeftProcessNeed[i] = 0;

NameProcessToPartition[i][best[j].id] = ProcessName[i];

break;

}

}

LeftFreePartition[best[j].id] = best[j].partitionSize;

}

display();

}

5.最坏适应算法WF

void WorstFit(){

cout<<"WF"<<endl;

initial();

int i,j,s,t;

sortNeed Worst[MAXNUMBER];

sortNeed temp;

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

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

Worst[i].id = i;

}

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

for (s=0;s<PartitionNum - 1;s++){

for (t=s;t<PartitionNum - 1;t++){

if (Worst[s].partitionSize < Worst[t].partitionSize){

temp = Worst[s];

Worst[s] = Worst[t];

Worst[t] = temp;

}

}

}

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

if (LeftProcessNeed[i] <= Worst[j].partitionSize){

Worst[j].partitionSize -= LeftProcessNeed[i];

LeftProcessNeed[j] = 0;

NameProcessToPartition[i][Worst[j].id] = ProcessName[i];

break;

}

else{

cout<<"Fail to Partition "<<ProcessName[i]<<endl;

break;

}

}

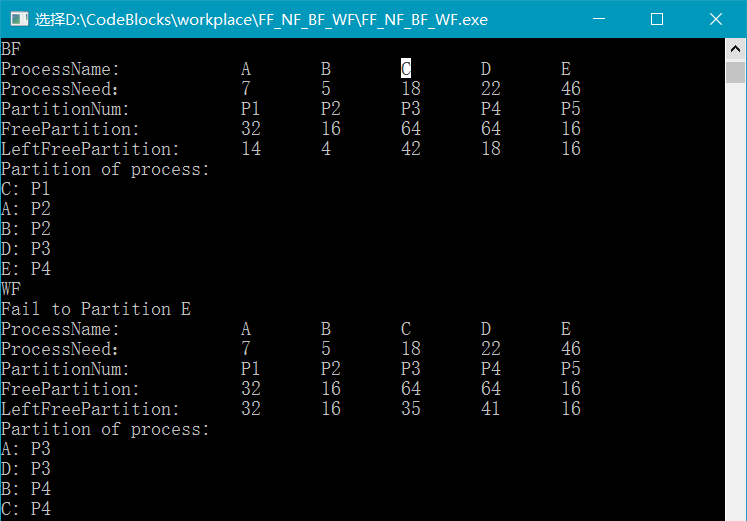
LeftFreePartition[Worst[j].id] = Worst[j].partitionSize;

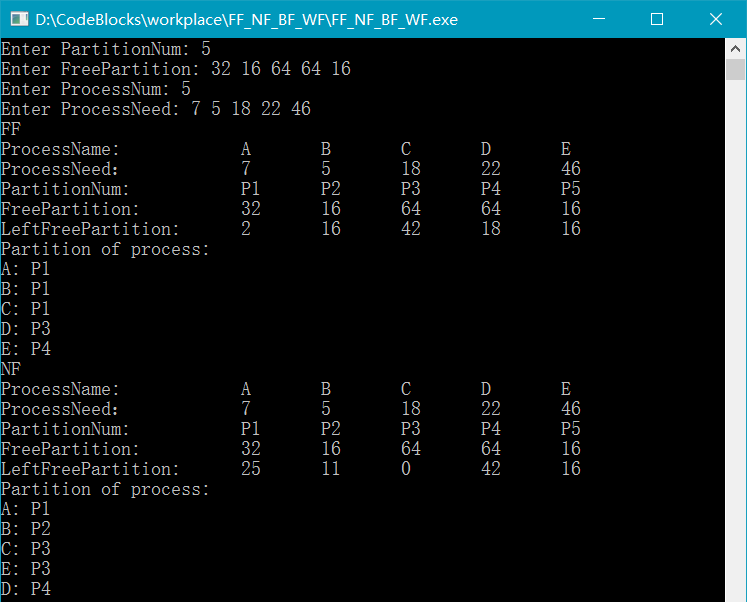
}

display();

}

1. **实验结果**

****

****

1. **附录**

源程序：

#include <iostream>

#include <fstream>

#include <iomanip>

using namespace std;

#define MAXNUMBER 100

static int PartitionNum; //内存中空闲分区的个数

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

static int ProcessNum; //进程个数

static char ProcessName[MAXNUMBER];//进程名字

static int ProcessNeed[MAXNUMBER]; //进程大小

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

static int LeftFreePartition[MAXNUMBER];

static int LeftProcessNeed[MAXNUMBER];

typedef struct{

int partitionSize;

int id;

}sortNeed;

void Enter();//输入分区数和大小、资源数和大小

void initial();//初始化供4个算法使用

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

int main(){

Enter();

FirstFit();

NextFit();

BestFit();

WorstFit();

return 0;

}

//输入分区数和大小、资源数和大小

void Enter(){

int i;

cout<<"Enter PartitionNum: ";

cin>>PartitionNum;

cout<<"Enter FreePartition: ";

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

cin>>FreePartition[i];

}

cout<<"Enter ProcessNum: ";

cin>>ProcessNum;

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

ProcessName[i]=i+65;

}

cout<<"Enter ProcessNeed: ";

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

cin>>ProcessNeed[i];

}

}

//初始化供4个算法使用

void initial(){

int i,j;

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

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

NameProcessToPartition[i][j] =NULL;

LeftFreePartition[j] = FreePartition[j];

}

}

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

LeftProcessNeed[i] = ProcessNeed[i];

}

}

//显示分区结果

void display(){

int i;

cout<<"ProcessName: "<<"\t\t";

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

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

}

cout<<endl;

cout<<"ProcessNeed： "<<"\t\t";

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

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

}

cout<<endl;

cout<<"PartitionNum: "<<"\t\t";

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

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

}

cout<<endl<<"FreePartition: "<<"\t\t";

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

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

}

cout<<endl<<"LeftFreePartition: "<<"\t";

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

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

}

cout<<endl<<"Partition of process:"<<endl;

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

for (int j = 0;j<ProcessNum;j++){

if (NameProcessToPartition[j][i]!=NULL){

cout<<NameProcessToPartition[j][i]<<": P"<<i+1<<endl;

}

}

}

}

//首次适应算法FF

void FirstFit(){

cout<<"FF"<<endl;

initial();

int i,j;

for (i=0;i<ProcessNum;i++){ //遍历进程

for (j=0;j<PartitionNum;j++){ //每次从分区的首地址开始查找

//当系统内存分区足够大的时候，即分配给进程资源

if (LeftProcessNeed[i] <= LeftFreePartition[j] && LeftFreePartition!=0){

LeftFreePartition[j] -= LeftProcessNeed[i]; //扣除分配给进程的资源

LeftProcessNeed[i] = 0; //当内存分区足够才执行，当前进程大小置0

NameProcessToPartition[i][j] = ProcessName[i]; //存储各个进程所在的分区位置

break; //一个进程分区完后，立即进行下一个进程的判断

}

}

}

display();

}

//循环首次适应算法NF

void NextFit(){

cout<<"NF"<<endl;

initial();

int i,nextPoint = 0;

bool isWhile;

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

isWhile = true;

while(isWhile){ //每次都从当前分区的下一个分区开始查找

if (LeftFreePartition[nextPoint] >= LeftProcessNeed[i]){

LeftFreePartition[nextPoint] -= LeftProcessNeed[i];

LeftProcessNeed[i] = 0;

NameProcessToPartition[i][nextPoint] = ProcessName[i];

nextPoint++;

if (nextPoint > PartitionNum - 1){

nextPoint = 0; //当j遍历到分区末尾的时候，返回首位置

}

isWhile = false;

}

else{

nextPoint++;

if (nextPoint > PartitionNum - 1)

nextPoint = 0; //当j遍历到分区末尾的时候，返回首位置

}

}

}

display();

}

//最佳适应算法BF

void BestFit(){

//对分区大小进行排序,每次分配完一个进程的内存大小后，重新排序

cout<<"BF"<<endl;

initial();

int i,j,s,t;

sortNeed best[MAXNUMBER];

sortNeed temp;

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

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

best[i].id = i;

}

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

//冒泡排序(每次分配完一个进程后，都需要重新排序)

for (s=0;s < PartitionNum - 1;s++){

for (t = s;t < PartitionNum - 1;t++){

if (best[s].partitionSize > best[t].partitionSize){

temp = best[s];

best[s] = best[t];

best[t] = temp;

}

}

}

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

if (LeftProcessNeed[i] <= best[j].partitionSize){

best[j].partitionSize -= LeftProcessNeed[i];

LeftProcessNeed[i] = 0;

NameProcessToPartition[i][best[j].id] = ProcessName[i];

break;

}

}

LeftFreePartition[best[j].id] = best[j].partitionSize;

}

display();

}

//最坏适应算法WF

void WorstFit(){

cout<<"WF"<<endl;

initial();

int i,j,s,t;

sortNeed Worst[MAXNUMBER];

sortNeed temp;

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

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

Worst[i].id = i;

}

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

for (s=0;s<PartitionNum - 1;s++){

for (t=s;t<PartitionNum - 1;t++){

if (Worst[s].partitionSize < Worst[t].partitionSize){

temp = Worst[s];

Worst[s] = Worst[t];

Worst[t] = temp;

}

}

}

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

if (LeftProcessNeed[i] <= Worst[j].partitionSize){

Worst[j].partitionSize -= LeftProcessNeed[i];

LeftProcessNeed[j] = 0;

NameProcessToPartition[i][Worst[j].id] = ProcessName[i];

break;

}

else{

cout<<"Fail to Partition "<<ProcessName[i]<<endl;

break;

}

}

LeftFreePartition[Worst[j].id] = Worst[j].partitionSize;

}

display();

## }