实验四 内存分配算法

姓名：叶倩琳

学号：201706061330

完成日期：2019/12/12

# 一、实验目的

# 编写程序，实现内存的分配算法（最佳适配和邻近适配）。

# 二、实验内容

1、在所提供的程序基础之上实现最佳适配和邻近适配算法。

2、增加测试案例，验证程序的正确性

-删除掉一个分配的进程（进程3）

-重新分配一个进程，假设进程6大小为100

3、利用VC++6.0实现上述程序设计和调试操作。

4、通过阅读和分析实验程序，熟悉内存分配算法。

# 实验步骤

1. **动态分区的数据结构**

动态分区的数据结构有两种：

1. 空闲区表形式

空闲分区表为每个尚未分配的分区设置一个表项，包括分区的序号、大小、始址和状态。为了更加直观，我运用了OSgraphics图形化软件包显示程序结果。

 2）空闲区链形式

为了实现对空闲分区的分配和链接，在每个分区的起始部分，设置一些 用于控制分区分配的信息，为了检索方便也设置了控制分区分配的信息，以 及用于链接其它分区的前、后向指针，从而可将所有的分区链接成一个双向 链表。我按照如下空闲链结构，设计出程序的主要数据结构。

1. **前向指针 N+2 0**
2. 0
3. **后向指针 N+2 0**
4. 0
5. N个字可用

# 图3-1-1 空闲链结构

1. **数据结构**

enum status { FREE, BUSY };

typedef struct freespace{

int num; //分区号

int size; //分区大小

int address; //分区首地址

status state; //分区状态，FREE和BUSY

};

typedef struct node{

freespace data;

node \*head;

node \*next;

}\*Linklist;

Linklist first,last;

node \*latest;//指针记录最新一次的地址分配

1. **首次适应算法FF（First Fit）**

【思想】

从空闲分区表的第一个表目起查找该表，把最先能够满足要求的空闲区分配给作业。为适应这种算法，空闲分区表中的空闲分区要按地址由低到高进行排序。

【算法】

若空闲区大小=申请空间大小，则分配，且只要更新节点的状态、大小。

若空闲区大小>申请空间大小，则分配，且要更新节点的前后指针信息以及数据信息。

1. **最佳适应算法BF（Best Fit）**

【思想】

从全部空闲区中找出能满足作业要求的、且大小最小的空闲分区，这种方法能使碎片尽量小。该算法保留大的空闲区，但造成许多小的空闲区。

【算法】

通过while循环找到最佳适配（min\_space最小）的空闲位置后判断若空闲区大小=申请空间大小，则分配，且只要更新节点的状态、大小；若空闲区大小>申请空间大小，则分配，且要更新节点的前后指针信息以及其他信息。

1. **循环首次适应算法/邻近算法NF(Next Fit)**

【思想】

把空闲分区表中的空闲分区按地址递增构成一个循环链。在分配内存空间时，从上次找到的空闲区的下一个空闲区开始查找，直到找到第一个能满足要求的的空闲区为止，并从中划出一块与请求大小相等的内存空间分配给作业。该算法使得内存中的空闲区分布得比较均匀。

【算法】

算法思想和FF算法类似，唯一不同是开始指针位置是上一次访问结束的指针位置。

1. **可变分区回收算法**

【思想】

当一个作业运行完毕释放内存时，系统根据释放区的首地址，从空闲区说明表中找到相应的插入点，分四种情况考虑：

A)当回收区既不与F1领接，又不与F2领接时，应为回收区单独建立一项新表目，填写回收区的起址和大小，插入到空闲区说明表的适当位置。

B)当回收区只与插入点的前一个分区F1相领接时，应将回收区与插入点的前一个分区合并，只需修改F1分区表目的大小即可。

C)当回收区只与插入点的后一个分区F2相领接时，将把两个空闲区合并，修改F2分区的表目大小，并把回收区的起址作为新空闲区的起址。

D)当回收区与插入点的前、后两个分区（F1和F2）都相领接时，合并三个分区，用F1表目的起址作为新空闲区的起址，修改其大小为三块分区之和，最后取消F2的表目。

【算法】

先通过指针找出用户输入的分区号的位置，修改该分区状态，并根据其前、后分区位置情况更新链表指针。

# 四、主要程序结构（附注释）

**（1）数据结构**

#define MAXSIZE 1024 //内存空间初始大小1024

enum status { FREE, BUSY };

typedef struct freespace{

int num; //分区号

int size; //分区大小

int address; //分区首地址

status state; //分区状态，FREE和BUSY

};

typedef struct node{

freespace data;

node \*head;

node \*next;

}\*Linklist;

Linklist first,last;

node \*latest;//指针记录最新一次的地址分配

**（2）初始化链表**

void initial(){

first = new node;

last = new node;

first->head = NULL;

first->next = last;

last->head = first;

last->next = NULL;

last->data.address = 0;

last->data.size = MAXSIZE;

last->data.num = 0;

last->data.state = FREE;

latest = first->next;

}

**（3）菜单**

void menu(){

cout << "--------------------内存分配系统--------------------" << endl;

cout << " 1.首次适配算法 " << endl;

cout << " 2.最佳适配算法 " << endl;

cout << " 3.查看主存分配情况 " << endl;

cout << " 4.回收主存 " << endl;

cout << " 5.exit " << endl;

cout << endl << "Please input the number you choose:";

}

**（4）打印主存分配情况**

void display(){

cout << "--------------------主存分配情况--------------------" << endl;

node \*p = first->next;

while (p){

cout << "分区号：";

if (p->data.num == 0){

cout << "空闲区 start address：" << p->data.address << " end address：" <<

p->data.address+p->data.size <<" Partition size：" << p->data.size << "KB" << " state:FREE" << endl;

}

else{

cout << p->data.num << " start address：" << p->data.address << " end address：" <<

p->data.address + p->data.size << " Partition size：" << p->data.size << "KB" << " state:";

if (p->data.state == FREE)

cout << "FREE" << endl;

else if (p->data.state == BUSY)

cout << "BUSY" << endl;

}

p = p->next;

}

cout << endl;

}

**（5）首次适配**

int firstAlloc(){

int num, size;

cout << "Please input a process number and allocated main memory size（KB）：" << endl;;

cin >> num >> size;

Linklist list = new node;

list->data.num = num;

list->data.size = size;

list->data.state = BUSY;

node \*p = first->next;

while (p){

//若空闲区大小>=申请空间大小，则分配更新

if (p->data.state == FREE&&p->data.size == size){

p->data.state = BUSY;

p->data.num = num;

display();

return 1;

}

if (p->data.state == FREE&&p->data.size > size){

list->head = p->head;

list->next = p;

list->data.address = p->data.address;

p->head->next = list;

p->head = list;

p->data.address = list->data.address + list->data.size;

p->data.size -= size;

display();

return 1;

}

latest = p;

p = p->next;

}

display();

return 0;

}

**（6）最佳适配**

int bestAlloc(){

int num, size;

cout << "Please input a process number and allocated main memory size（KB）：" << endl;;

cin >> num >> size;

int min\_space=MAXSIZE;

Linklist list = new node;

list->data.num = num;

list->data.size = size;

list->data.state = BUSY;

node \*p = first->next;

node \*q = NULL;

while (p){

if ((p->data.size > size || p->data.size == size)&&p->data.state==FREE){

if (p->data.size - size < min\_space){

q = p;

min\_space = p->data.size - size;

}

}

latest = p;

p = p->next;

}

if (q == NULL)

return 0;

else{

if (min\_space == 0){

q->data.num = num;

q->data.state = BUSY;

display();

return 1;

}

else{

list->head = q -> head;

list->next = q;

list->data.address = q->data.address;

q->head->next = list;

q->head = list;

q->data.address += size;

q->data.size -= size;

display();

return 1;

}

}

}

**（7）邻近适配**

int nearAlloc(){

int num, size;

cout << "Please input a process number and allocated main memory size（KB）：" << endl;;

cin >> num >> size;

Linklist list = new node;

list->data.num = num;

list->data.size = size;

list->data.state = BUSY;

node \*p = latest->next; //不同

while (p){

//若空闲区大小>=申请空间大小，则分配更新

if (p->data.state == FREE&&p->data.size == size){

p->data.state = BUSY;

p->data.num = num;

display();

return 1;

}

if (p->data.state == FREE&&p->data.size > size){

list->head = p->head;

list->next = p;

list->data.address = p->data.address;

p->head->next = list;

p->head = list;

p->data.address = list->data.address + list->data.size;

p->data.size -= size;

display();

return 1;

}

latest = p;

p = p->next;

}

display();

return 0;

}

**（8）回收碎片**

int recycle(){

int num;

cout << "Please input the process number you want to reclaim memory：" << endl;

cin >> num;

node \*p = first;

while (p){

if (p->data.num == num){

p->data.state = FREE;

p->data.num = 0;

//根据与前后空闲区update内存分布情况

if (p->head->data.state == FREE){

p->head->data.size += p->data.size;

p->head->next = p->next;

p->next->head = p->head;

}

if (p->next->data.state == FREE){

p->data.size += p->next->data.size;

p->next->next->head = p;

p->next = p->next->next;

}

break;

}

p = p->next;

}

display();

return 1;

}

# 五、实验结果

（1）进入主界面，查看内存最初状态

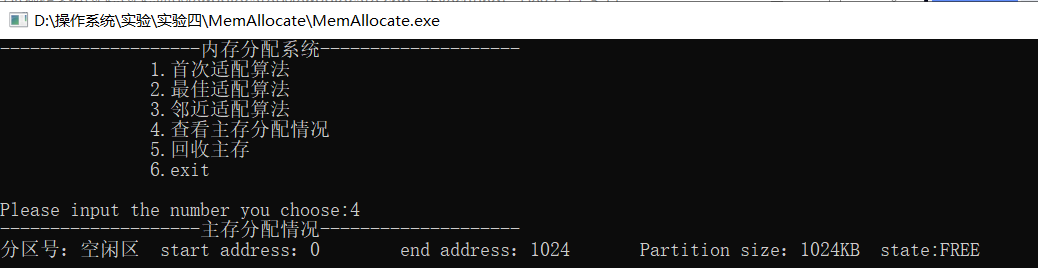


图5-1-1 内存最初分配情况

（2）采用FF分配进程

AddAJobInEnd("Job1",1,180,NULL);

AddAJobInEnd("Job2",2,360,NULL);

AddAJobInEnd("Job3",3,180,NULL);

AddAJobInEnd("Job4",4,60,NULL);

AddAJobInEnd("Job5",5,20,NULL);

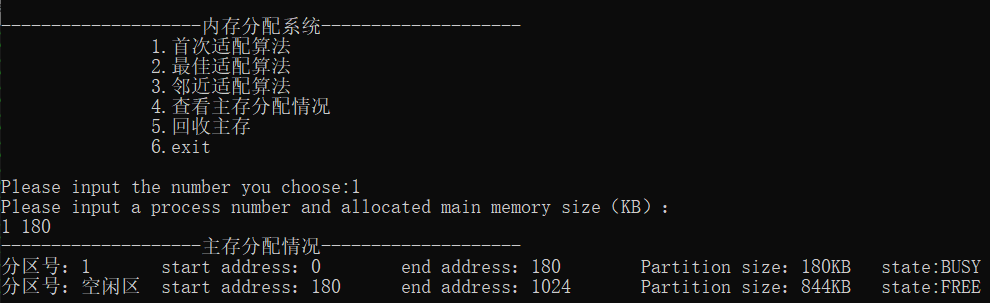


图5-2-1 给进程1分配内存180K

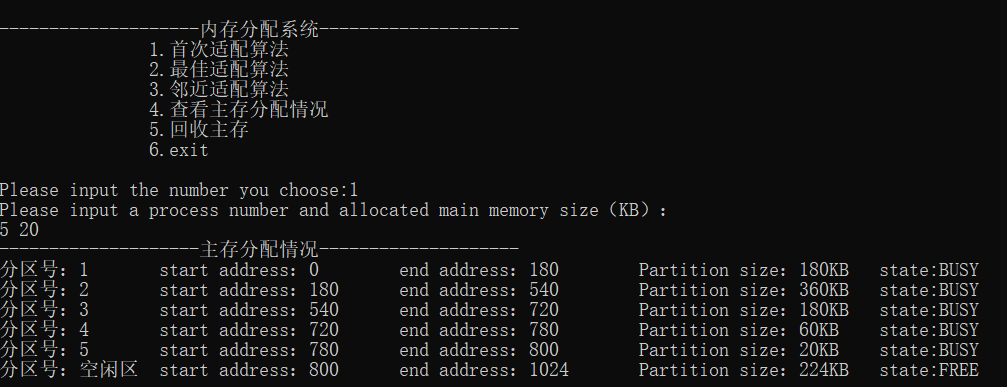


图5-2-2 FF分配结束后内存分配情况

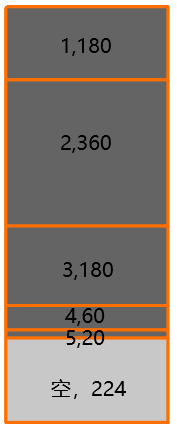


图5-2-3 FF分配结束后内存分配情况

1. 回收进程1和3

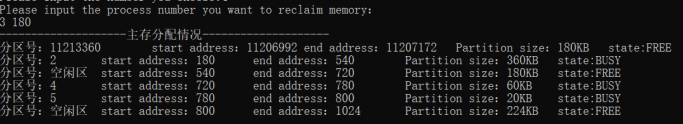


图5-3-1 回收进程1、3后内存分配情况

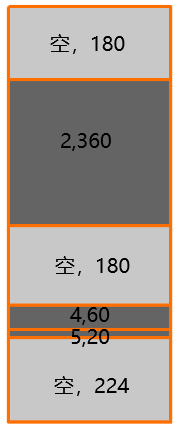


图5-3-2 回收进程1、3后内存分配情况

（4）采用BF分配进程

AddAJobInEnd("Job1",1,20,NULL);

AddAJobInEnd("Job2",6,170,NULL);

AddAJobInEnd("Job3",3,20,NULL);

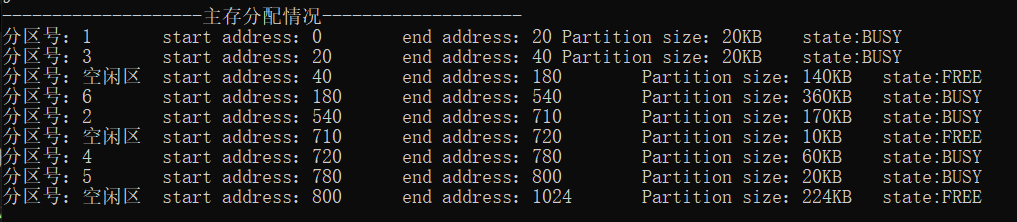


图5-4-1 BF分配完成后内存分配情况

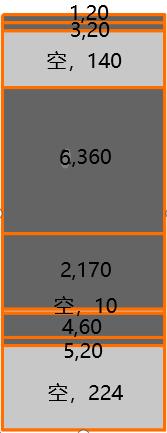


图5-4-2 BF分配完成后内存分配情况

（5）采用BF分配进程1内存10K

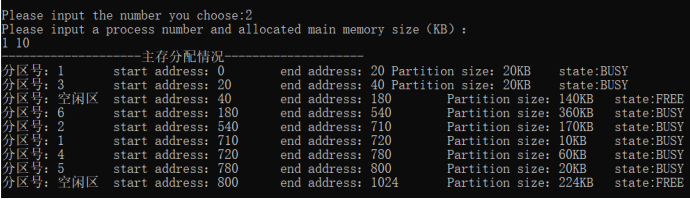


图5-5-1 BF分配进程1内存10K后内存分配情况

（6）采用NF分配进程9内存10K



图5-6-1 BF分配进程9内存10K后内存分配情况

（7）在（2）的情况下采用NF分配进程2、7、8

AddAJobInEnd("Job1",7,190,NULL);

AddAJobInEnd("Job2",2,170,NULL);

AddAJobInEnd("Job3",8,50,NULL);



图5-7-1 NF分配进程后内存分配情况

（8）回收进程2



图5-8-1 回收进程2后内存分配情况

# 六、实验中遇到的问题及解决方法

【问题】

在邻近适配算法中，我在原先数据结构的基础上增加了latest指针来指向最后一次分配的地址位置，后来发现在这种情况下，有的时候碎片回收算法会出错。

【解决】

经分析发现，在上述情况下，如果执行碎片回收算法，首先要判断释放的位置是否刚好位于latest指针所在的位置，若是，则回收的时候要在向上合并的过程中加上对latest指针值的更新，否则下次执行邻近适配算法时会出错。

# 七、实验体会

考虑到固定分区具有分区的数目事先生成，限制了系统中活动进程的数目、小作业不能有效地利用分区空间、存在内部碎片问题的缺点，所以本次实验设计实现了动态分区的分配算法，既可以获得较大的灵活性，又能提高内存的利用率。 其中，最佳适配算法是选择内存空闲块中最适合进程大小的块分配；邻近适配算法是从上一次分配的地址开始查找符合要求的块，所查找到的第一个满足要求的空闲块就分配给进程。在设计这些算法进行模拟时，运用了链表指针，由于逻辑不够严谨导致几次出错，我会在以后加强这方面的逻辑概念。

在实验过程中，我发现这几个算法的主要代码结构是类似的，只是针对具体的算法有所小变动，比如：NF算法比FF算法多增加了一个latest指针来指向最新修改的位置。

我发现实践与理解还是有很大差距的，只有多上机才能发现自己的很多问题，也只有通过不断纠正自身存在的问题，才会有一定的进步提高。