**郑州轻工业大学**

**《操作系统》课程****实验报告**

实验名称：动态分区存储管理算法

姓 名： 原彬贺

学 号： 542001020223

地 点： 实验楼302

指导老师： 吴庆岗

专业班级： 计算机科学与技术20-02班

实验成绩：

1. 实验目的：

编写程序，实现动态分区存储管理算法，模拟动态分区存储分配、回收和合并的过程，加深对动态分区存储管理的理解。

二、实验内容：

模拟实现至少两种动态分区存储管理算法（首次适应算法、最佳适应算法或最坏适应算法等）。每种算法包括动态分区存储分配、回收和合并的过程，熟悉并掌握各种算法的空闲区组织方式。实验内容如下：

1、分区的初始化——可以由用户输入初始分区的大小（初始化后只有一个空闲分区，起始地址为0，大小是用户输入的大小）。

2、分区的动态分配过程：由用户输入作业号和作业的大小，实现分区过程。

3、分区的回收：用户输入作业号，实现分区回收，同时，分区的合并也要体现出来（注意：不存在的作业号要给出错误提示！）。

4、分区的显示：任何时刻，可以查看当前内存的情况（起始地址是什么，大小多大的分区是空闲或者占用的，能够显示出来）。

实验要求如下：

（1）内存空间不足的情况，要有相应的显示；

（2）作业不能同名，但是删除后可以再用这个名字；

（3）作业空间回收时输入作业名，回收相应的空间，如果这个作业名不存在，也要有相应的提示。

三、实验代码

#include <stdio.h>

#include <stdlib.h>

#include <iostream>

#include <string>

#include <vector>

#include <list>

#include <algorithm>

#include <set>

using namespace std;

class work {

private:

int id;

int start;

int end;

int size;

int stat;

public:

work(int ID, int Size, int Start = 0, int Stat = 1) :id(ID),size(Size), start(Start), stat(Stat) {

this->end = Start + Size;

}

bool setstart(int Start)

{

if (stat == 1)

{

this->start = Start;

this->end = this->start + this->size;

}

else if (stat == 0)

{

this->start = Start;

this->size = this->end - this->start;

}

return true;

}

bool setid(int ID)

{

if (stat == 0)

{

id = ID;

stat = 1;

return true;

}

else

{

return false;

}

}

int setsize(int Size)

{

this->size = Size;

this->end = this->start + Size;

return size;

}

int setstat(int Stat)

{

stat = Stat;

return stat;

}

int getid()const

{

return id;

}

int getstat()const {

return stat;

}

int getstart()const

{

return start;

}

int getend()const

{

return end;

}

int getsize()const

{

return size;

}

bool operator==(const work& s)const

{

if (id == s.id && stat == s.stat)

{

return true;

}

else

return false;

}

};

class master {

protected:

int start;

int end;

int busy;

int free;

set<int> bit;

list<work> listworks;

list<work> freeworks;

list<work> busyworks;

public:

master(int End, int Start = 0) :start(Start), end(End) {

work s(0, End - Start, Start, 0);

free = End - Start;

listworks.push\_back(s);

freeworks.push\_back(s);

}

bool virtual addwork(const work s) = 0;

bool virtual delwork(const int num) = 0;

bool virtual insertlist(list<work>& free, work s) = 0;

void showfree() {

cout << "内存大小为 ：" << this->end - this->start << " 内存已使用 ：" << this->end - this->free << " 内存剩余" << this->free << endl;

cout << " worklist" << endl;

cout << "stat 为 1 为使用，stat 为 0 为空闲" << endl;

cout << "ID" << " " << "start" << " " << "end" << " " << "size"<<" " << "stat" << endl;

for (auto& itr : listworks)

{

cout << itr.getid() << " " << itr.getstart() << " " << itr.getend() << " " << itr.getsize() << " " << itr.getstat() << endl;

}

cout << " freelist" << endl;

cout << "ID" << " " << "start" << " " << "end" << " " << "size" << " " << "stat" << endl;

for (auto& itr : freeworks)

{

cout << itr.getid() << " " << itr.getstart() << " " << itr.getend() << " " << itr.getsize() << " " << itr.getstat() << endl;

}

}

};

class firstmaster :public master {

public:

firstmaster(int End, int Start = 0) :master(End, Start) {

}

bool virtual insertlist(list<work>& free, work s)

{

int w = 0;

for (auto itr = free.begin(); itr != free.end(); itr++)

{

w++;

if (itr->getstart() > s.getstart())

{

//itr--;

//cout << itr->getid() << " s "<<w<<endl;

//cout << "wancheng" << endl;

//break;

free.insert(itr, s);

return true;

}

}

free.insert(free.end(), s);

return true;

}

bool addwork(work s)

{

if (bit.count(s.getid()) == 1)

{

cout << "该id已使用，请换其他的ID" << endl;

return false;

}

bit.insert(s.getid());

if (s.getsize() <= free)

{

auto itr = freeworks.begin();

for (; itr != freeworks.end(); itr++)

{

if ((\*itr).getsize() > s.getsize())

{

auto litr = find(listworks.begin(), listworks.end(), (\*itr));

s.setstart((\*itr).getstart());

itr->setstart(s.getend());

litr->setstart(s.getend());

listworks.insert(litr, s);

free -= s.getsize();

return true;

}

else if ((\*itr).getsize() == s.getsize())

{

auto litr = find(listworks.begin(), listworks.end(), (\*itr));

(\*litr).setid(s.getid());

litr->setstat(1);

freeworks.erase(itr);

free -= s.getsize();

return true;

}

}

if (itr == freeworks.end())

{

bit.erase(s.getid());

return false;

}

}

else

{

bit.erase(s.getid());

cout << "内存不足"<< endl;

return false;

}

}

bool virtual delwork(const int num)

{

work tem(num, 0);

auto itr = find(listworks.begin(), listworks.end(), tem);

if (itr != listworks.end())

{

free += itr->getsize();

if (itr == listworks.begin())

{

auto itrr = itr;

itrr++;

if (itrr == listworks.end()) //内存只有一个空间块使用

{

(\*itr).setstat(0);

freeworks.push\_back(\*itr);

}

else {

if ((\*itrr).getstat() == 0)

{

itr->setsize(itr->getsize() + itrr->getsize());

listworks.erase(itrr);

auto fitr = freeworks.begin();

fitr->setsize(itr->getsize());

}

else {

itr->setstat(0);

freeworks.insert(freeworks.begin(), \*itr);

}

}

}

else

{

auto itrl = itr;

itrl--;

auto itrr = itr;

itrr++;

if (itrl->getstat() == 0 && itrr != listworks.end() && itrr->getstat() == 0)

{

auto fitr = find(freeworks.begin(), freeworks.end(), \*itrl);

itrl->setsize(itrl->getsize() + itr->getsize() + itrr->getsize());

fitr->setsize(itrl->getsize());

itrr = listworks.erase(itr);

listworks.erase(itrr);

}

else if (itrl->getstat() == 0){

auto fitr = find(freeworks.begin(), freeworks.end(), \*itrl);

itrl->setsize(itrl->getsize() + itr->getsize());

fitr->setsize(itrl->getsize());

listworks.erase(itr);

}

else {

itr->setstat(0);

this->insertlist(this->freeworks, \*itr);

//fitr--;

//cout << fitr->getid() << endl;

//this->freeworks.insert(fitr, \*itr);

}

}

bit.erase(num);

return true;

}

else {

cout << "该任务不存在" << endl;

return false;

}

}

};

class goodmaster :public master {

public:

goodmaster(int End, int Start = 0) :master(End, Start) {

}

bool virtual insertlist(list<work>& free, work s)

{

for (auto itr = free.begin(); itr != free.end(); itr++)

{

if (itr->getsize()> s.getsize())

{

//itr--;

//cout << itr->getid() << " s "<<w<<endl;

//cout << "wancheng" << endl;

//break;

//itr--;

free.insert(itr, s);

return true;

}

}

free.insert(free.end(), s);

return true;

}

bool addwork(work s)

{

if (bit.count(s.getid()) == 1)

{

cout << "该id已使用，请换其他的ID" << endl;

return false;

}

bit.insert(s.getid());

if (s.getsize() <= free)

{

auto itr = freeworks.begin();

for (; itr != freeworks.end(); itr++)

{

if ((\*itr).getsize() > s.getsize())

{

auto litr = find(listworks.begin(), listworks.end(), (\*itr));

s.setstart((\*itr).getstart());

litr->setstart(s.getend());

//to do inset free

freeworks.erase(itr);

this->insertlist(this->freeworks, \*litr);

listworks.insert(litr, s);

free -= s.getsize();

return true;

}

else if ((\*itr).getsize() == s.getsize())

{

auto litr = find(listworks.begin(), listworks.end(), (\*itr));

(\*litr).setid(s.getid());

litr->setstat(1);

freeworks.erase(itr);

free -= s.getsize();

return true;

}

}

if (itr == freeworks.end())

{

bit.erase(s.getid());

return false;

}

}

else

{

bit.erase(s.getid());

cout << "内存不足" << endl;

return false;

}

}

bool virtual delwork(const int num)

{

work tem(num, 0);

auto itr = find(listworks.begin(), listworks.end(), tem);

if (itr != listworks.end())

{

free += itr->getsize();

if (itr == listworks.begin())

{

auto itrr = itr;

itrr++;

if (itrr == listworks.end()) //内存只有一个空间块使用

{

(\*itr).setstat(0);

freeworks.push\_back(\*itr);

}

else {

if ((\*itrr).getstat() == 0)

{

freeworks.erase(itrr);

itr->setsize(itr->getsize() + itrr->getsize());

listworks.erase(itrr);

this->insertlist(this->freeworks, \*itr);

}

else {

itr->setstat(0);

this->insertlist(this->freeworks, \*itr);

}

}

}

else //不是第一个元素

{

auto itrl = itr;

itrl--;

auto itrr = itr;

itrr++;

if (itrl->getstat() == 0 && itrr != listworks.end() && itrr->getstat() == 0)//前后都能合并

{

freeworks.remove(\*itrl);

freeworks.remove(\*itrr);

itrl->setsize(itrl->getsize() + itr->getsize() + itrr->getsize());

itrr = listworks.erase(itr);

listworks.erase(itrr);

this->insertlist(this->freeworks, \*itrl);

}

else if (itrl->getstat() == 0) { //只能合并一个元素

//auto fitr = find(freeworks.begin(), freeworks.end(), \*itrl);

freeworks.remove(\*itrl);

itrl->setsize(itrl->getsize() + itr->getsize());

this->insertlist(freeworks, \*itrl);

//fitr->setsize(itrl->getsize());

listworks.erase(itr);

}

else {//不能进行元素合并

itr->setstat(0);

this->insertlist(this->freeworks, \*itr);

//fitr--;

//cout << fitr->getid() << endl;

//this->freeworks.insert(fitr, \*itr);

}

}

bit.erase(num);

return true;

}

else {

cout << "该任务不存在" << endl;

return false;

}

}

};

void test(master\* master)

{

master->addwork(work(1, 10));

master->addwork(work(2, 10));

master->addwork(work(3, 10));

master->addwork(work(4, 10));

master->showfree();

master->delwork(2);

master->showfree();

master->addwork(work(2, 5));

master->showfree();

master->addwork(work(2, 4));

master->showfree();

master->addwork(work(0, 10));

master->showfree();

}

int main()

{

master\* master;

cout << "欢迎使用动态空间管理，请进行初始化操作" << endl;

int add, size;

//cout << "请输入开始内存地址" << endl;

//cin >> add;

add = 0;

cout << "请输入内存大小" << endl;

cin >> size;

while (1)

{

cout << "请输入要选择的分区类型 1.最佳适应分区 2.最先匹配分区" << endl;

int s;

cin >> s;

if (s == 1)

{

master = new goodmaster(size, add);

break;

}

if (s == 2)

{

master = new firstmaster(size, add);

break;

}

}

master->showfree();

//test(master);

while (1)

{

int select;

cout << "请选择操作" << endl;

cout << "1.添加任务" << endl;

cout << "2.删除任务" << endl;

cout << "3.查看内存" << endl;

cout << "0.退出" << endl;

cin >> select;

if (select == 0)

{

break;

}

else if (select == 1)

{

int id;

int size;

cout << "请输入任务ID" << endl;

cin >> id;

cout << "请输入任务大小" << endl;

cin >> size;

cout << "正在准备插入任务 "<<id << endl;

if (master->addwork(work(id, size)))

{

cout << "插入成功" << endl;

}

else

{

cout << "插入失败" << endl;

}

}

else if (select == 2)

{

int id;

int size;

cout << "请输入任务ID" << endl;

cin >> id;

cout << "正在准备删除任务 "<<id << endl;

if (master->delwork(id))

{

cout << "删除成功" << endl;

}

else

{

cout << "删除失败" << endl;

}

}

else if (select == 3)

{

master->showfree();

}

}

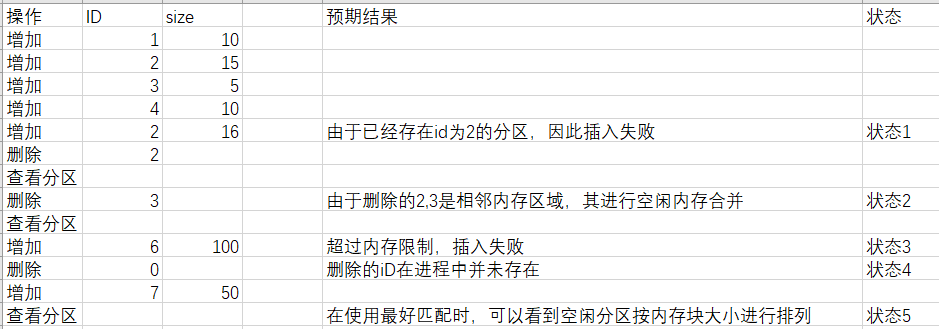
return 0;

}

四、实验结果

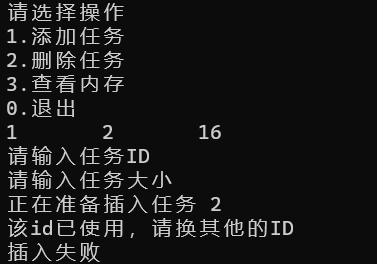
本系统维护两个列表，一个是内存链表，一个是空闲链表，在内存链表中，由于进行匹配是按照id和state进行匹配的，只有完全吻合时，才会进行匹配，因此，在内存链表中，允许两个不同状态的id相同，但不允许两个相同状态的id相同

操作数据



实验结果：

状态1：插入相同id

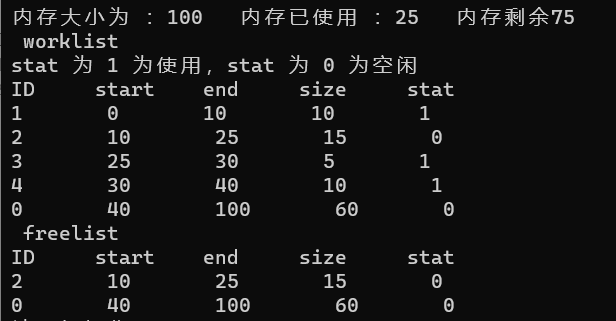


图表 1插入相同id

状态2：删除后空闲内存进行合并

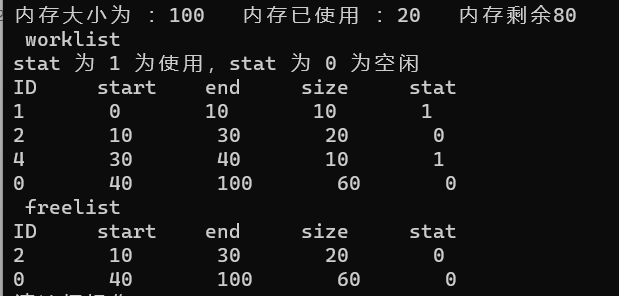
删除区块3前的空间状态，此时空闲块2与任务块3相连，此时删除块3后形成的空闲块3可以与空闲块2合并

删除前空闲块2大小为15



图表 2删除前空间状态

删除后，可以看到任务块3删除了，同时形成的空闲块3与空闲块2合并成空闲块2，此时空闲块2的大小为20



图表 3删除后空间状态

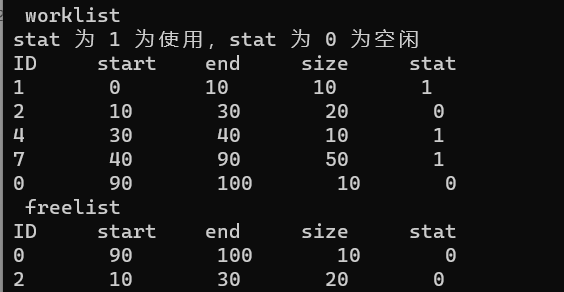
状态3：插入超过空闲内存大小的块

状态4删除id在任务块中不存在的块

状态5：经过增加删除操作后，内存形成了空闲块的分离，此时空闲块链表中维护的顺序变为按空闲块大小排列的顺序

最佳适应算法：

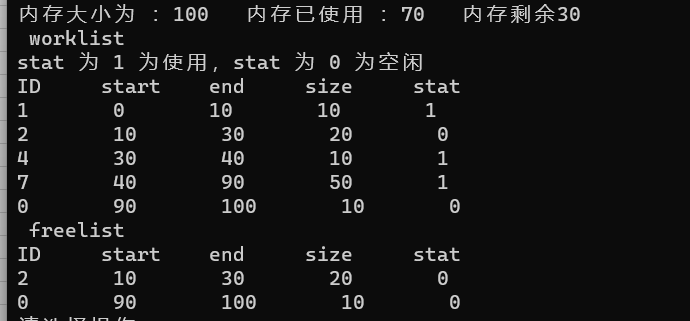
可以看到在freelist中，尽管id为0的空闲块开始内存在id为1的空闲块前面，但由于0的size比2的size小，因此空闲块2在空闲块0的后面



图表 4最佳适应算法中空闲块链表排序策略

首次适应算法

对于首次适应算法来说，由于空闲内存链表由起始地址进行排列，因此由于id为2的空闲块起始地址比id为0的空闲块起始位置小，因此空闲块2在空闲块0的前面



图表 5最先适应算法中空闲块链表排序策略

五、实验总结

在进行动态分区存储管理算法的实验过程中，我深刻地理解到了动态分区存储分配、回收和合并的过程。通过模拟这些过程，我不仅加深了对动态分区存储管理的理解，同时还具体实现了最佳适配和最先适配算法。

首先，在实验中，我了解到了动态分区存储管理的一般流程。这个流程包括了初始化、分配、回收和合并等步骤。其中，初始化是将整个存储空间划分为若干个固定大小的块；分配是根据需求将可用的块分配给不同的进程或线程；回收则是将不再使用的块回收并释放空间；合并则是将相邻的空闲块合并成更大的块以提高利用率。

其次，在实验中，我学会了如何实现动态分区存储管理算法。具体来说，我使用了一种基于链表的数据结构来管理每个进程或线程所占用的块。当需要分配新的块时，我会先检查当前空闲块的数量是否足够，如果不够则会向链表中添加一个新的节点来表示一块可用的空闲块；如果有足够的空闲块，则会从链表头部取出一个块并分配给需要的进程或线程。当需要回收块时，我会遍历整个链表，找到所有已经被标记为“已使用”的块并将其回收。最后，当需要合并块时，我会遍历整个链表，找到相邻的两个空闲块并将它们合并成一个更大的块。

在最佳适应算法中，我通过维护一个按size升序的空闲块链表，来进行快速寻找插入位置，而在最先适应算法中，我通过委会一个按start地址升序的空闲块链表来进行快速寻找插入位置，以此利用空间来换取时间效率。

总之，通过这次实验，我对动态分区存储管理算法有了更深入的理解，并且掌握了如何实现这种算法的方法和技巧。我相信这些知识和技能对我今后的工作和学习都会有很大的帮助。

|  |
| --- |
| **评语**（不要删除，指导老师填写评语，放置在实验报告后面单独成页） |
|  |