项目说明文档

操作系统课程设计

——内存管理系统

作 者 姓 名： 李德涛

学 号： 1852141

指 导 教 师： 王冬青

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 简介 1](#_Toc495668153)

[2 设计 2](#_Toc495668156)

[2.1 首次适应算法 2](#_Toc495668157)

[2.2 最佳适应算法 2](#_Toc495668158)

2.3 LRU算法..............................................................................................................2

[3 实现 3](#_Toc495668161)

3.1 mainMM.java的实现...........................................................................................3

3.2 Chain.java的实现.................................................................................................4

3.3 BestFit.java的实现...............................................................................................4

3.4 FirstFit.java的实现...............................................................................................5

3.5 Page.java的实现...................................................................................................6

3.6 LRU.java的实现...................................................................................................6

[4 测试 9](#_Toc495668186)

# 1 简介

开发环境：jdk-14.0.1\_windows-x64

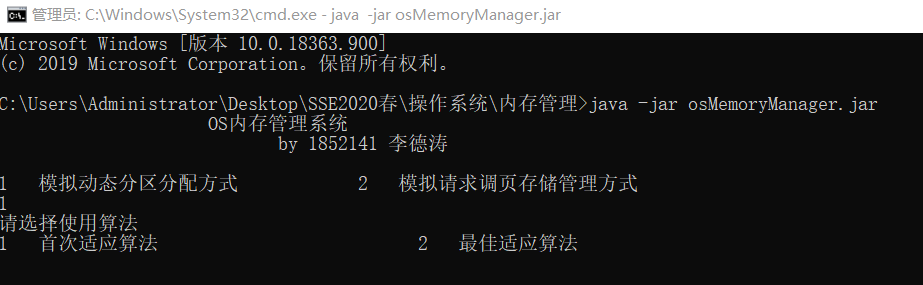
开发平台：Eclipse

开发语言：JAVA

项目名称：内存管理系统osMemoryManager

操作界面：





操作说明：

在安装有JAVA JDK包的情况下，使用cmd的java命令打开osMemoryManager.jar文件即可。

通过输入数字执行命令，1执行模拟动态分区分配方式，2执行模拟请求调页存储管理方式。模拟动态分区分配方式可选择使用首次适应算法或最佳适应算法执行。

# 2 设计

内存管理系统的核心算法包括以下三部分：

## 2.1 首次适应算法

该算法从空闲分区链首开始查找，直至找到一个能满足其大小要求的空闲分区为止。然后再按照作业的大小，从该分区中划出一块内存分配给请求者，余下的空闲分区仍留在空闲分区链中。 该算法倾向于使用内存中低地址部分的空闲区，在高地址部分的空闲区很少被利用，从而保留了高地址部分的大空闲区。显然为以后到达的大作业分配大的内存空间创造了条件。其缺点在于，低地址部分不断被划分，留下许多难以利用、很小的空闲区，而每次查找又都从低地址部分开始，会增加查找的开销。

2.2 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。每次分配给文件的都是最合适该文件大小的分区，缺点在于内存中留下许多难以利用的小的空闲区。

2.3 LRU算法

最近使用的页面数据会在未来一段时期内仍然被使用,已经很久没有使用的页面很有可能在未来较长的一段时间内仍然不会被使用。基于这个思想,LRU算法设计了一种缓存淘汰机制，每次从内存中找到最久未使用的数据然后置换出来，从而存入新的数据。它的主要衡量指标是使用的时间，附加指标是使用的次数。它的合理性在于优先筛选热点数据。

# 3 实现

源代码由dynamicSim包、pageSim包和主类mainMM.java组成：

3.1 mainMM.java的实现

该类为程序的入口，通过Scanner检测用户输入的指令并转到相应的类执行操作。

源码：

**public** **static** **void** main(String[] args)

{

System.***out***.println(" OS内存管理系统");

System.***out***.println(" by 1852141 李德涛");

System.***out***.println();

System.***out***.println("1 模拟动态分区分配方式 2 模拟请求调页存储管理方式");

Scanner sc = **new** Scanner(System.***in***);

**int** key = 0;

key = sc.nextInt();

**if**(key==2) {

LRU.*run*();

}

**else** {

System.***out***.println("请选择使用算法");

System.***out***.println("1 首次适应算法 2 最佳适应算法");

key = sc.nextInt();

**if**(key ==1) {

FirstFit.*run*();

}

**else** {

BestFit.*run*();

}

}

sc.close();

3.2 Chain.java的实现

该类是一个实体类，定义了动态分区分配方式中的分区链。成员包括分区链的起始地址、长度以及其一个用于表示该分区占用状态的标识。

带参数构造器：

**public** Chain(**int** address, **int** lenth, String flag)

{

**this**.address = address;

**this**.lenth = lenth;

**this**.flag = flag;

}

3.3 BestFit.java的实现

该类实现了最佳适应算法。mainMM通过run()运行时，先初始化空闲和已分配的分区链，然后每次调用distribute函数运行最佳适应算法完成作业。3.4中使用同样的方法，不再赘述。

源码：

**public** **static** **void** run()

{

//初始化空闲和已分配的分区链

List<Chain> undisChain = **new** LinkedList<>();

List<Chain> disChain = **new** LinkedList<>();

undisChain.add(**new** Chain(0, 640, "未分配"));

//完成以下作业

*distribute*("作业1", 130, **true**, undisChain, disChain);

*distribute*("作业2", 60, **true**, undisChain, disChain);

*distribute*("作业3", 100, **true**, undisChain, disChain);

*distribute*("作业2", 60, **false**, undisChain, disChain);

*distribute*("作业4", 200, **true**, undisChain, disChain);

*distribute*("作业3", 100, **false**, undisChain, disChain);

*distribute*("作业1", 130, **false**, undisChain, disChain);

*distribute*("作业5", 140, **true**, undisChain, disChain);

*distribute*("作业6", 60, **true**, undisChain, disChain);

*distribute*("作业7", 50, **true**, undisChain, disChain);

*distribute*("作业6", 60, **false**, undisChain, disChain);

}

BF算法：

Optional<Chain> min = undisChain.stream()

.filter((Chain chain) -> chain.getLenth() >= lenth)

.min((o1, o2) -> o1.getLenth() - o2.getLenth());

Chain chain = min.get();

**if** (chain.getLenth() > lenth)

{

disChain.add(**new** Chain(chain.getAddress(), lenth, num));

chain.setAddress(chain.getAddress() + lenth);

chain.setLenth(chain.getLenth() - lenth);

} **else** **if** (chain.getLenth() == lenth)

{

disChain.add(**new** Chain(chain.getAddress(), lenth, num));

undisChain.remove(chain);

}

3.4 FirstFit.java的实现

该类实现了首次适应算法。mainMM通过run()运行时，先初始化空闲和已分配的分区链，然后每次调用distribute函数运行首次使用算法完成作业。

FF算法：

**for** (**int** i = 0; i < undisChain.size(); i++)

{

Chain chain = undisChain.get(i);

**if** (chain.getLenth() > lenth)

{

disChain.add(**new** Chain(chain.getAddress(), lenth, num));

chain.setAddress(chain.getAddress() + lenth);

chain.setLenth(chain.getLenth() - lenth);

**break**;

} **else** **if** (chain.getLenth() == lenth)

{

disChain.add(**new** Chain(chain.getAddress(), lenth, num));

undisChain.remove(i);

**break**;

}

}

3.5 Page.java的实现

该类是一个实体类，定义了请求调页存储管理方式中的页面。成员包括页号、所在的内存块号、用于表示是否在内存中的布尔量、LRU算法所需的访问次数字段。

带参数构造器：

**public** Page(**int** pageNum)

{

**this**.pageNum = pageNum;

blockNum = -1;

flag = **false**;

visited = -1;

}

3.6 LRU.java的实现

该类使用LRU算法模拟了请求调页存储管理方式，包含三个成员：

//mainMM通过run()运行LRU算法

public static void run()

//定义了一个洗牌方法，用于生成320条指令的随机序列

public static <T> void swap(int[] arr, int i, int j)

public static <T> void shuffle(int[] arr)

随机序列的生成：

**public** **static** <T> **void** swap(**int**[] arr, **int** i, **int** j){

**int** temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

**public** **static** <T> **void** shuffle(**int**[] arr) {

**int** length = arr.length;

**for** ( **int** i = length; i > 0; i-- ){

**int** randInd = *rand*.nextInt(i);

*swap*(arr, randInd, i - 1);

}

}

从数组的最后一个位置（假设下标是n）开始向前扫描，然后随机生成一个0到n之间的随机数，假设该随机数是r1，然后将数组最后一个位置（下标n）与r1位置互换，之后开始扫面下一个数（下标为n-1），然后随机生成一个0到(n-1)之间的随机数，假设该随机数是r2，然后将数组倒数第二个位置（下标为n-1）与r2位置互换，然后继续扫面下一个数（下标为n-2），就这样一直迭代下去。在这个迭代过程中，可以保证扫面点左边的数字都是尚未确定位置的，而右边的数字都是已经安排好位置的。

生成随机序列后，初始化页面和内存，按顺序执行指令。访问指令在内存中时，显示其物理地址。当访问缺页且内存未满时，直接将页面调入，访问缺页且需要调页时，使用LRU算法决定需要替换的页面。访问缺页时需要记录最近一次访问时间。同时每次访问时记录lossCount和getCount，全部320条指令运行结束后计算缺页率

LRU源码：

pages[memory[org]].setFlag(**false**);

pages[memory[org]].setBlockNum(-1);

memory[org] = page.getPageNum();

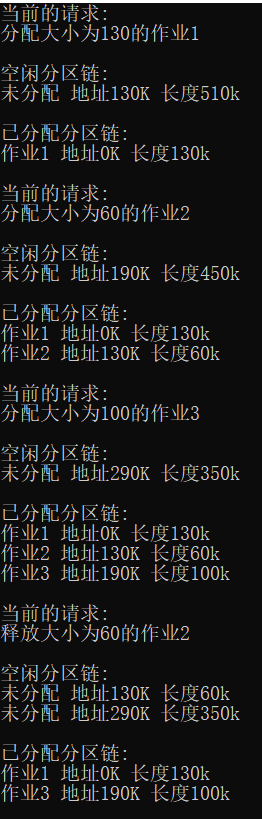
page.setBlockNum(org);

page.setFlag(**true**);

page.setVisited(getCount++);

# 4 测试

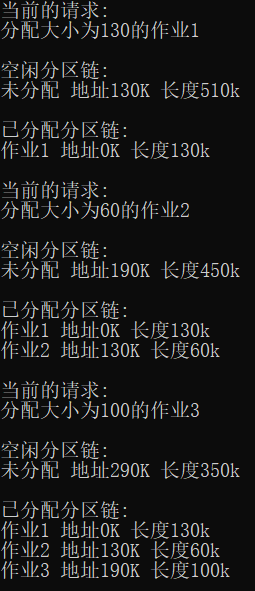
4.1 首次适应算法模拟动态分区分配方式

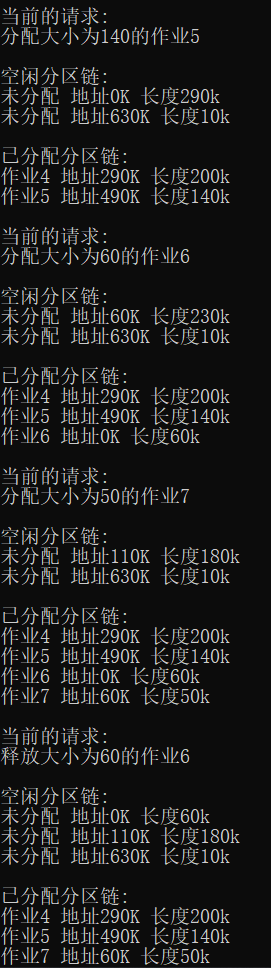


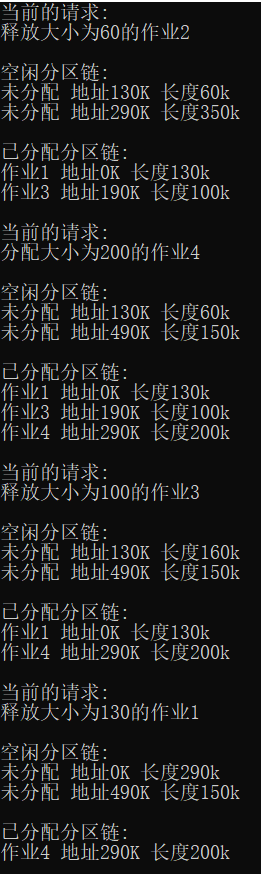
## IMG_256

## IMG_256

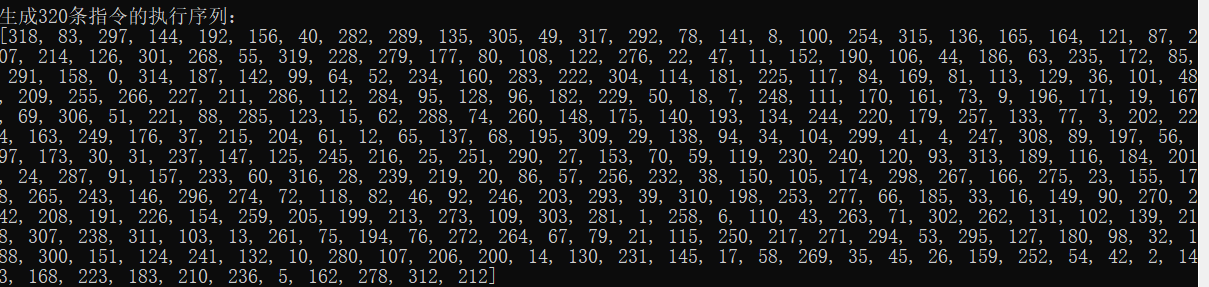
4.2 最佳适应算法模拟动态分区分配方式

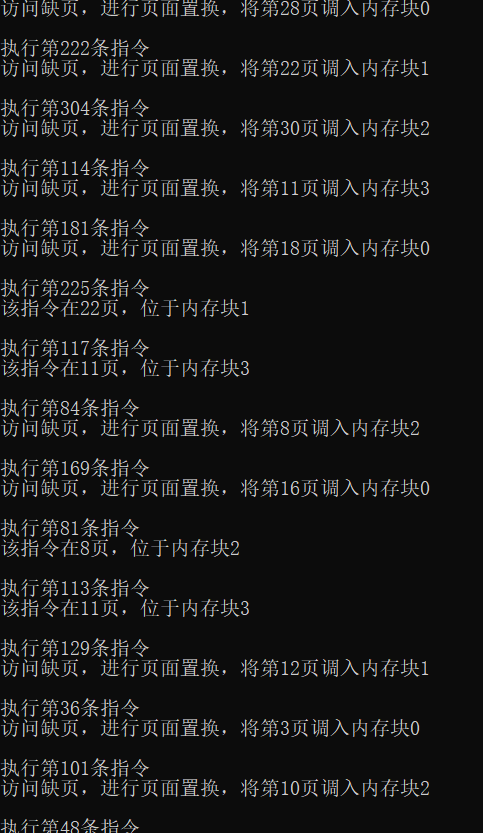
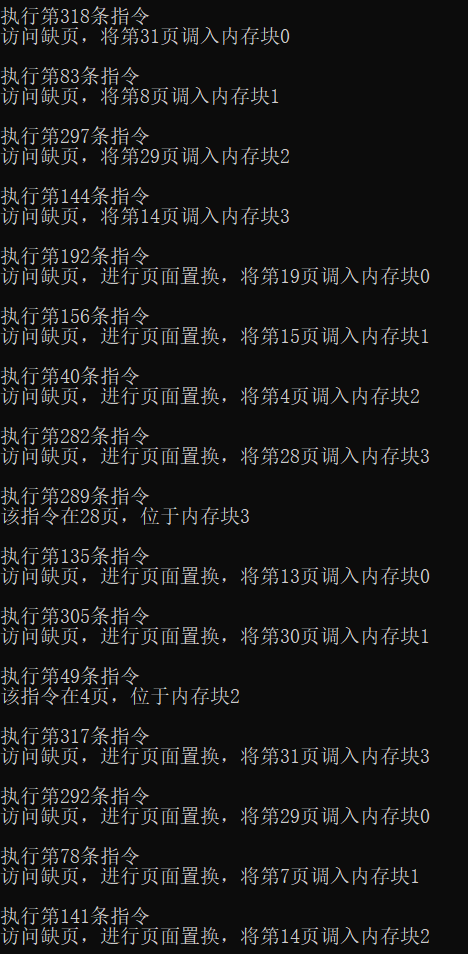






4.3 模拟请求调页存储管理方式





由于指令太多，只截取部分结果。使用上述的随机序列得到的缺页率较大，而使用项目要求文档中给出的可选序列缺页率较小，在0.5左右。

