操作系统内存管理项目：

计算机内存管理模拟系统

李万葶（1652724）

Contents

一、分析

项目目的

1、加深对动态分区存储管理方式及其实现过程的理解。

2、加深对请求调页系统的原理和实现过程的理解。

3、进一步掌握动态分区中的首次适应算法和最佳适应算法。

4、进一步掌握请求分区置换算法，本系统实现了FIFO和LRU两种算法。

5、通过设计图形界面对计算机内存管理过程进行模拟。

项目功能要求

1、动态分区分配方式的模拟：

要求：假设初始态下，可用内存空间为640K，并有下列请求序列，请分别用首次适应算法和最佳适应算法进行内存块的分配和回收，并显示出每次分配和回收后的空闲分区链的情况来。

给定任务列表：

作业1申请130K

作业2申请60K

作业3申请100k

作业2释放60K

作业4申请200K

作业3释放100K

作业1释放130K

作业5申请140K

作业6申请60K

作业7申请50K

作业6释放60K

2、请求调页存储管理方式模拟：

内容：假设每个页面可存放10条指令，分配给一个作业的内存块为4。模拟一个作业的执行过程，该作业有320条指令，即它的地址空间为32页，目前所有页还没有调入内存。

模拟过程：

●在模拟过程中，如果所访问指令在内存中，则显示其物理地址，并转到下一条指令；如果没有在内存中，则发生缺页，此时需要记录缺页次数，并将其调入内存。如果4个内存块中已装入作业，则需进行页面置换。

●所有320条指令执行完成后，计算并显示作业执行过程中发生的缺页率。

●置换算法可以选用FIFO或者LRU算法。

●作业中指令访问次序可以按照下面原则形成：50%的指令是顺序执行的，25%是均匀分布在前地址部分，25%是均匀分布在后地址部分。

题目分析

根据题目要求，系统需要模拟的计算机功能分为动态分区和请求分区两种，这两种从实现方法的角度又能各自分为两种，相当于系统需要执行四种算法，输出相应的结果，并通过相应的界面将模拟过程和结果进行呈现。

系统设计的基本思路，是用户可通过button按钮自由选择模拟内容，系统监听鼠标点击后跳转到相应的模拟页面。

二、设计

类结构设计

按照之前的分析，系统的类结构只设计了两种：一类是界面，另一类用来存放对象。由于整个系统需要四个模拟结果页面、选择“FF还是BF算法”的子页面、选择“FIFO还是LRU”算法的子页面和选择“动态模拟还是请求模拟”的主页面，而最终的模拟结果页面两两类似，可以只用两个页面类分别呈现动态分区和请求分区的结果，所以系统总共包含五个页面类，以及存储进程对象的Process类和存储块对象的Block类。

系统的类结构总体比较简单，类总共只有七个，它们以页面跳转为脉络，相互衔接方便用户点选。

成员操作设计

在本Java系统中，Process类与Block类相当于C语言中的结构体，充当自定义类型的作用。除此以外，其他的五个页面类具有许多类似的成员操作，这里将它们总体概述。

页面类的主要操作为绘制页面和执行算法。绘制页面即绘制系统为满足用户需要设置的标签、按钮等，按钮分别对应鼠标监听，所以五个页面都包含响应鼠标监听、跳转到下一页面的操作，以除了主页面外的所有页面都包含的“返回”键为例：

```
类中：JButton jb=null;

构造函数中：jb=new JButton("返回");

                jb.addActionListener(this);//加入事件监听

public void actionPerformed(ActionEvent e) {

                    if(e.getActionCommand()=="返回") {

                        this.dispose();

                    }

}
```

界面设计

模拟系统的界面设计较为简单，使用Java.swing中不同的布局方式将标签、按钮、文本框和表格等元素铺排在JFrame窗口。主页面和选择算法页面尺寸较小，只包涵两个选择按钮

和一个返回按钮，简单清晰，在动态分布模拟结果界面，系统使用两个表格呈现1～11步之后的内存情况，在请求分区模拟结果页面，系统使用两个文本域展示产生的320个随机数和对应的调用页面，用两个文本框呈现缺页数和缺页率。

由于页面总数较少，所有的页面图片将在后面的部分呈现。

算法设计

动态分布模拟算法包括FF和BF，由于这两种方法的回收方式是相同的，所以系统用三个函数来实现这两种算法，它们是BF分配、FF分配和Free回收函数。

请求分区模拟算法包括FIFO和LRU，函数设计分为随机数生成、初始化块、Exist、Space等。

部分核心代码片段将在下一部分给出。

三、实现

FF实现

```
    static void FF(int number, int size, int task){
        Process date = new Process();
        date.number = number;
        date.length = size;
        int i;
        for ( i = 0; i < list.size(); i++) {
            if (date.length <= ((Process)list.get(i)).length && ((Process)list.get(i)).flag
== 0) {
                break;
            }
        }
        if (i == list.size()) {
            System.out.println("没有足够的内存进行分配");
        }
        if (((Process)list.get(i)).length - date.length <= 4 && i != list.size() - 1 ) {
            ((Process)list.get(i)).number = date.number;
            ((Process)list.get(i)).flag = 1;
        }
        else {
            date.flag = 1;
            ((Process)list.get(i)).length -= date.length;
            date.startAddress = ((Process)list.get(i)).startAddress;
            ((Process)list.get(i)).startAddress += date.length;
```

```
            list.add(i, date);
        }
        disPlay(task);
    }
```

BF实现

```
    static void BF(int number, int size,int task){
        Process date = new Process();
        date.number = number;
        date.length = size;
        int m, j;
        Process target = new Process();
        for (m = 1; m < list.size()-1; m++){
            j = m;
            target.number = ((Process)list.get(m)).number;
            target.flag = ((Process)list.get(m)).flag;
            target.length = ((Process)list.get(m)).length;
            target.startAddress = ((Process)list.get(m)).startAddress;
            while(j>0 && ((Process)list.get(j-1)).length>target.length){

                ((Process)list.get(j)).number = ((Process)list.get(j-1)).number;
                ((Process)list.get(j)).flag = ((Process)list.get(j-1)).flag;
                ((Process)list.get(j)).length = ((Process)list.get(j-1)).length;
                ((Process)list.get(j)).startAddress =
((Process)list.get(j-1)).startAddress + ((Process)list.get(j)).length;
                j--;
            }
            ((Process)list.get(j)).number = target.number;
```

```
                ((Process)list.get(j)).length = target.length;
                ((Process)list.get(j)).flag = target.flag;
                ((Process)list.get(j)).startAddress = target.startAddress-
((Process)list.get(j+1)).length;


            }
            int i;
            for ( i = 0; i < list.size(); i++) {
                if (date.length <= ((Process)list.get(i)).length &&
((Process)list.get(i)).flag == 0) {
                    //当有适合的内存，且未被使用
                    break;
                }
            }
            if (i == list.size()) {
            }
            if (((Process)list.get(i)).length - date.length <= 4 && i != list.size() - 1 ) {
                ((Process)list.get(i)).number = date.number;
                ((Process)list.get(i)).flag = 1;
            }
            else {
                //分片分配内存
                date.flag = 1;
                ((Process)list.get(i)).length -= date.length;
                date.startAddress = ((Process)list.get(i)).startAddress;
                ((Process)list.get(i)).startAddress += date.length;
                list.add(i, date);
            }
```

```
            disPlay(task);
        }
```

FIFO实现

```
    void FIFO() {
            Rand();
            initiate();
            int exist, space, position;
            int curpage;
            for (int i = 0; i < 320; i++) {
                pc = temp[i];
                curpage = pc / 10;
                exist = Exist(curpage);
                if (exist == -1) {
                    space = Space();
                    if (space != -1) {
                        block[space].pagenum = curpage;
                        n = n + 1;
                    }
                    else {
                        position = Replace();
                        if (position >= block.length || block[position] ==
null) {
                            break;
                        }
                        else {
                            block[position].pagenum = curpage;
                            n++;
```

```
                                        block[position].accessed--;
                                }
                        }
                }
                for (int j = 0; j < Bsize; j++)
                        block[j].accessed++;
        }
    }
```

LRU实现

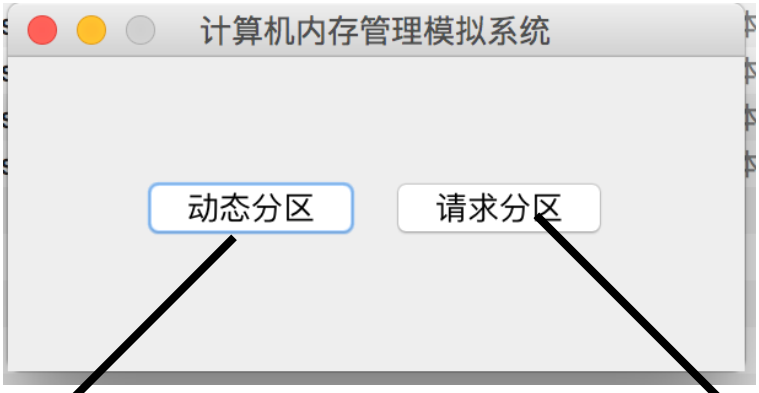
```
    void LRU() {
            Rand();
            initiate();
            int exist, space, position;
            int curpage;
            for (int i = 0; i < 320; i++) {
                    pc = temp[i];
                    curpage = pc / 10;
                    exist = Exist(curpage);
                    if (exist == -1) {
                            space = Space();
                            if (space != -1) {
                                    block[space].pagenum = curpage;
                                    n = n + 1;
                            }
                            else {
                                    position = Replace();
```

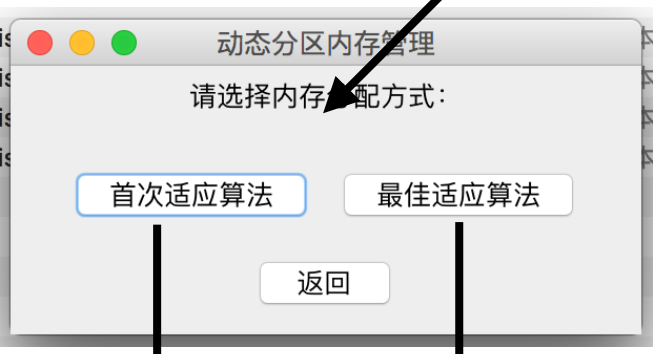
```
                                    if (position >= block.length || block[position] ==
null) break;
                                    else {
                                        block[position].pagenum = curpage;
                                        n++;
                                    }
                                }
                            }
                            else block[exist].accessed = -1;
                            for (int j = 0; j < 4; j++) {
                                block[j].accessed++;
                            }
                        }
                    }
```

经本机测试，程序运行状况正常，正确地模拟了计算机内存分配情况，符合题目要求。如果可执行程序打开异常，请老师耐心重新运行一次，java源码也可以在eclipse中编译运行，谢谢您！

四、测试

实现效果图


计算机内存管理模拟系统
动态分区
请求分区


动态分区内存管理
请选择内存分配方式：
首次适应算法
最佳适应算法
返回


请求分区内存管理
请选择内存分配方式：
FIFO算法
LRU算法
返回