项目说明文档

操作系统课程设计

——动态分区分配方式的模拟

作 者 姓 名：　罗吉皓

学　　号：　1652792

指 导 教 师：　王冬青老师

学院、 专业：　软件学院 软件工程

同济大学

Tongji University

目录

I. 分析

1.1 项目名称： 动态分区分配方式的模拟

1.2 项目需求

实现动态分区分配方式进行内存管理

1.3 项目功能描述

假设初始态下，可用内存空间为640K，并有下列请求序列

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*	

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。

1.4 项目需求分析

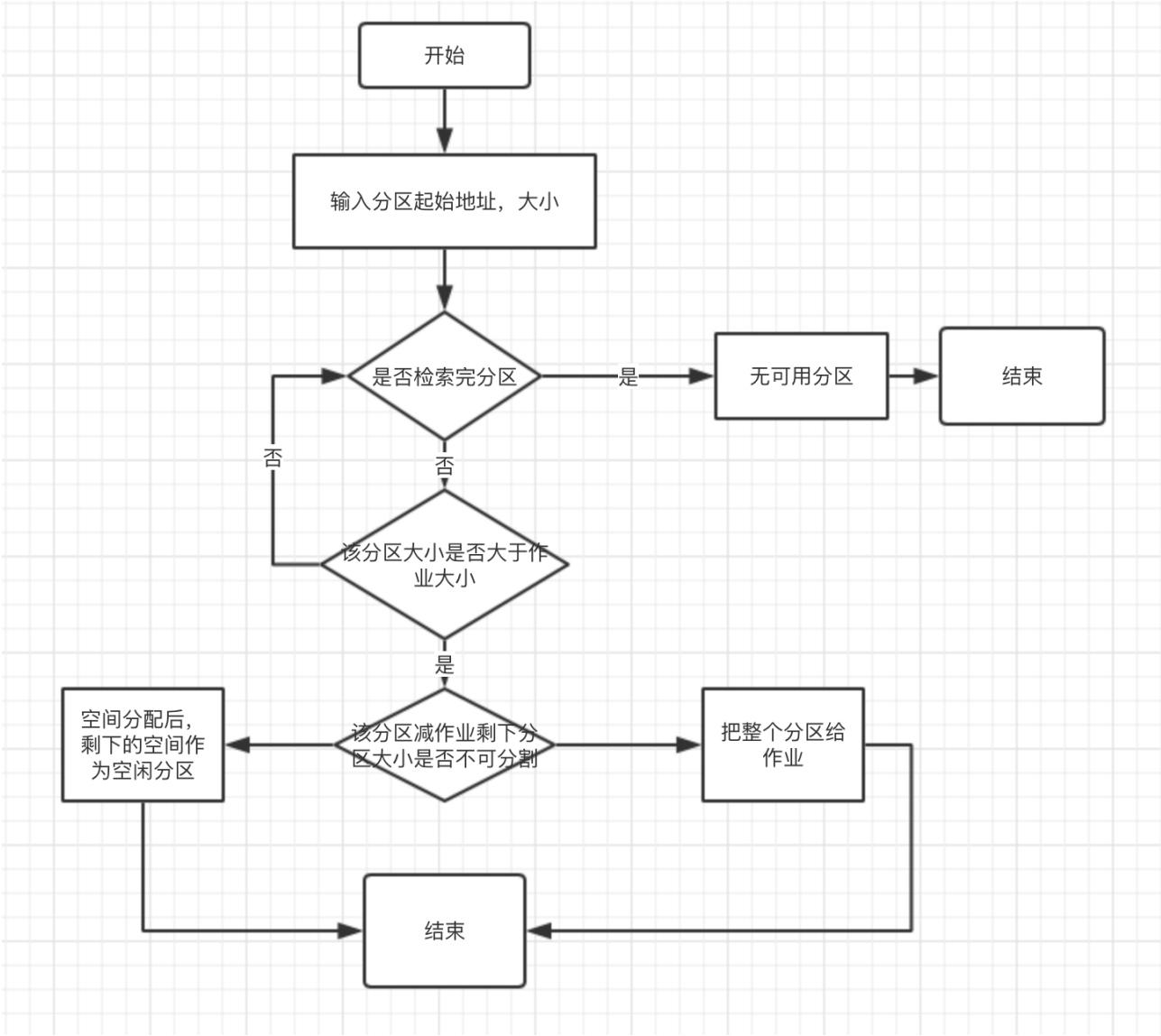
实现动态分区分配方式进行内存管理主体是首次适应算法和最佳适应算法的实现。以下是对于首次适应算法和最佳适应算法的定义：

> *首次适应算法（first-fit）：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，这种方法的目的在于减少查找时间。*
>
> *最佳适应算法（best-fit）：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。*

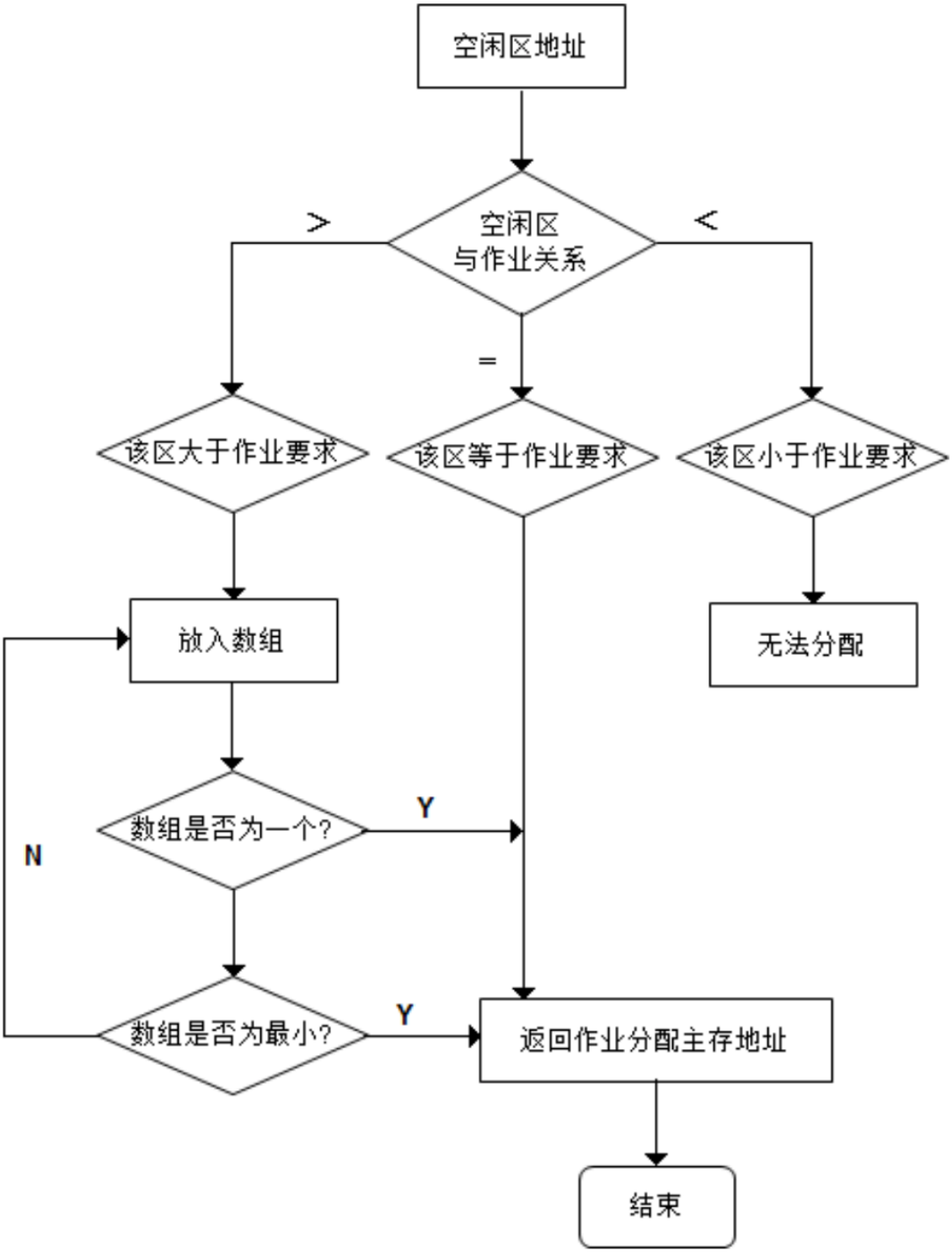
从定义中不难看出，首次适应算法要求我们从当前**最先搜索到的空闲区**作为优先分配的空间。一般来说，首次适应算法会对于空闲区进行单方向的遍历，知道全部搜索完毕没有空闲区的时候，才会回到起点，对于整体进行遍历。而最佳适应算法要求每次分配的空闲分区，是要**满足条件的最佳空闲分区**。这要求我们每次分配时，都要对于当前所有的空闲分区进行遍历，排序，最后进行分配工作。

相关流程图如下图所示：

首次适应算法


开始
输入分区起始地址，大小
是否检索完分区
是
无可用分区
结束
否
否
该分区大小是否大于作业大小
是
空间分配后，剩下的空间作为空闲分区
该分区减作业剩下分区大小是否不可分割
把整个分区给作业
结束

最佳适应算法


空闲区地址
空闲区
与作业关系
>
=
<
该区大于作业要求
该区等于作业要求
该区小于作业要求
放入数组
无法分配
数组是否为一个?
Y
N
数组是否为最小?
Y
返回作业分配主存地址
结束

1.5 内存回收

项目的另一个重点是对于释放的内存进行回收。在讲释放的作业所在的内存块改为空闲状态后，我们需要进行相关的判断：

*1.*新释放的内存块的前一块内存块是否为空，若为空，则将这两块内存块进行合并

*2.*新释放的内存块的后一块内存块是否为空，若为空，则将这两块内存块进行合并

*3.*释放的内存块的前一块和后一块内存块是否为空，若为空，则将这三块内存块进行合并

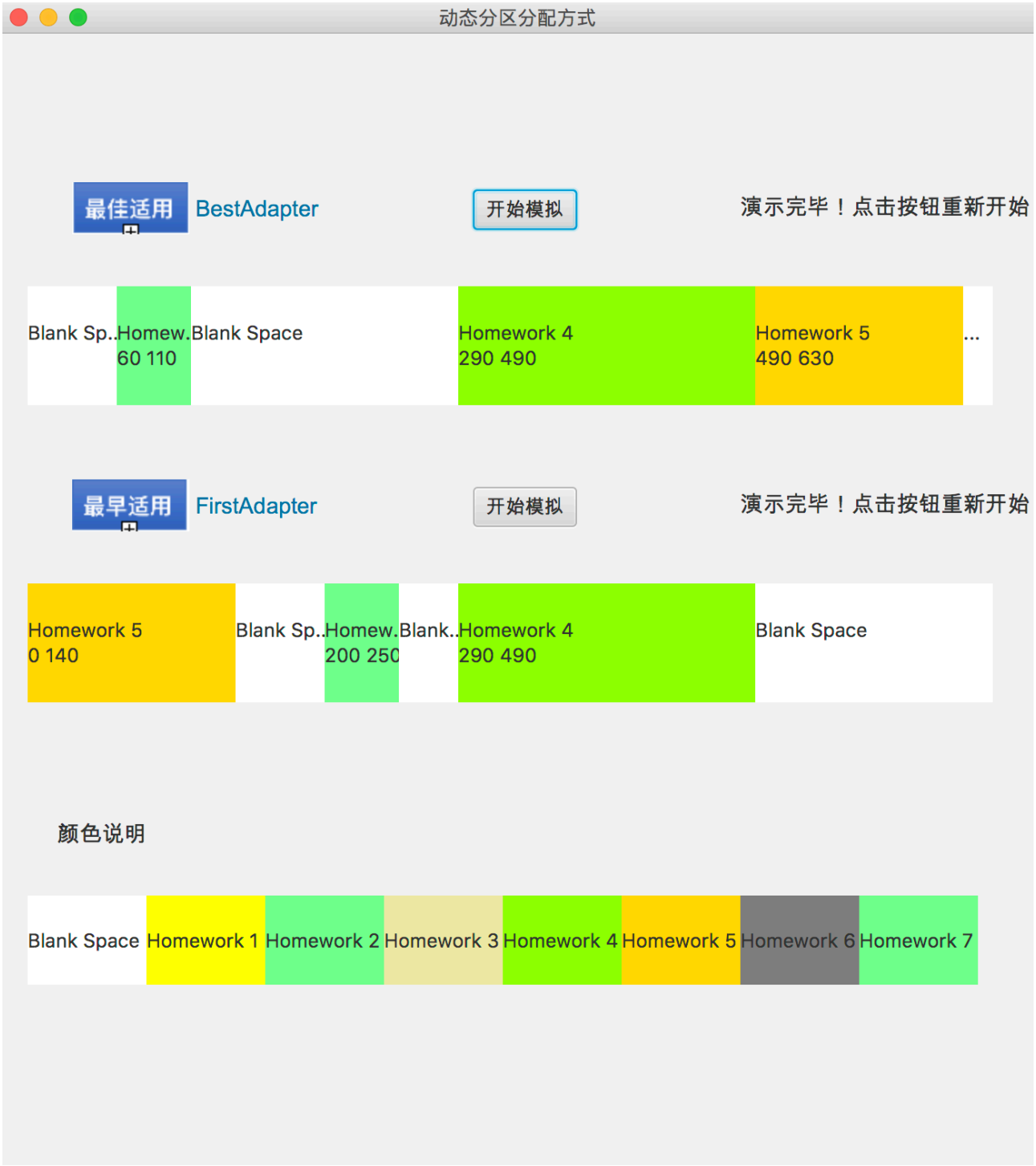
II. 设计

2.1 项目整体设计

整体界面中首次适应算法和最佳适应算法分开演示，通过点击按钮来进行下一条指令的操作。在系统中，给不同的作业分配了不同

的颜色，使得演示更为一目了然。在相应的文本框中，显示分区大小的起始以及终止位置。

实际界面如下图所示：


动态分区分配方式
最佳适用
BestAdapter
开始模拟
演示完毕！点击按钮重新开始
Blank Sp..
Homew.
60 110
Blank Space
Homework 4
290 490
Homework 5
490 630
...
最早适用
FirstAdapter
开始模拟
演示完毕！点击按钮重新开始
Homework 5
0 140
Blank Sp..
Homew.
200 250
Blank..
Homework 4
290 490
Blank Space
颜色说明
Blank Space
Homework 1
Homework 2
Homework 3
Homework 4
Homework 5
Homework 6
Homework 7

2.2 开发工具及环境

开发环境：

java version "1.8.0_144"

Java(TM) SE Runtime Environment (build 1.8.0_144-b01)

Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)

开发工具：

Intellij IDEA（Java）

2.3 前端设计

整个前端由Javafx制作，程序最开始启动的时候，跳出弹窗显示该程序的运行方式。点击确认后进入主程序，首次适应算法和最佳适应算法分开演示，通过点击按钮来进行下一条指令的操作。在系统中，给不同的作业分配了不同的颜色，使得演示更为一目了然。在相应的文本框中，显示分区大小的起始以及终止位置。

当所有的请求序列完成后，会显示“演示完毕！点击按钮重新开始”。再次点击按钮，会重新从第一条序列开始演示。

III 实现

3.1 整体实现

本项目主要采用MVC（Model-View-Controller）框架来实现。

由于没有链接数据库，Model主要实现首次适应算法和最佳适应算法。

首次适应算法和最佳适应算法各自生成自己的作业调度序列，传给前端，由按钮的触碰事件来出发相应指令。

3.2 控件

本程序中动画演示的效果由label制作。通过不断在原来的label上添加label，起到覆盖的作用，并且用不同的颜色等来实现不同作业的情况演示。

主界面的生成：

```
部分代码演示：
    Scene scene = new Scene(root, 700, 764);
    Image image = new
Image(getClass().getResourceAsStream("BestAdapter.png"),79,35,false,false);
    Label labelBestAdapter = new Label();
    image = new
Image(getClass().getResourceAsStream("FirstAdapter.png"),79,35,false,false);
    Label labelFirstAdapter = new Label();
    final Button strBestButton = new Button("开始模拟");
    final Button strFirstButton = new Button("开始模拟");

    label.setText("Blank Space\n    ");
    label.setBackground(new Background(new BackgroundFill(Color.WHITE, null, null)));

```

3.3 相关类的设计

程序中实现了memoryBlock类的设计，主要是为了储存相应的内存块，以及他们储存的位置，具体实现如下：

```java
public class MemoryBlock {
    private int startPosition;
    private int endPosition;
    private int curTag;
    private int curLength;

    public MemoryBlock() {}
    public MemoryBlock(int startPosition, int endPosition,int curTag) {
        this.startPosition = startPosition;
        this.endPosition = endPosition;
        this.curTag = curTag;
        this.curLength = endPosition-startPosition;
    }

    public int getStartPosition() {
        return startPosition;
    }
    public int getCurTag() {
        return curTag;
    }
    public void setCurTag(int curTag) {
        this.curTag = curTag;
    }
    public int getEndPosition() {
        return endPosition;
    }
    public int getCurLength() {
        return curLength;
    }
    public void setStartPosition(int startPosition) {
        this.startPosition = startPosition;
    }
    public void setEndPosition(int endPosition) {
        this.endPosition = endPosition;
    }
    public void setCurLength() {
        this.curLength = this.endPosition - this.startPosition;
    }
}

```

```
相关参数说明：
    private int startPosition;
    private int endPosition;
    private int curTag;
    private int curLength;

startPosition 与 endPosition 代表了内存块存储的起始和终止位置
curTag                       代表当前是作业编号，0代表空白块
curLength                    代表长度
```

3.4 Model实现

首次适应算法分配的实现：

```
    boolean inputFlag = false;
    int temp;
    for (temp = 0; temp < memoryBlocks.size(); temp++){
        if (memoryBlocks.get(temp).getCurTag() == 0 &&
memoryBlocks.get(temp).getCurLength()>numCommand[2]){
            inputFlag = true;
            break;
         }
     }
     if (!inputFlag) System.out.println("分配空间不足");
     else{
          int curStartPosition = memoryBlocks.get(temp).getStartPosition();
          int curEndPosition = curStartPosition + numCommand[2];
          MemoryBlock cMBlock = new
MemoryBlock(curStartPosition,curEndPosition,numCommand[0]);
          regionNum++;
          memoryBlocks.get(temp).setStartPosition(curEndPosition);
          memoryBlocks.get(temp).setCurLength();
          memoryBlocks.add(temp,cMBlock);
     }
```

最佳适应算法在最后的分配上与首次适应算法相似，在分配之前还有一个对于所有内存块的排序，找出最适合的内存块进行分配。

最佳适应算法的排序算法实现如下：

```
            String[] sCommand = new String[3];
            sCommand = firstAdapterCommand.get(i).toString().split(" ");
            int[] numCommand = new int[3];
            for (int j = 0; j < 3; j++) {
                numCommand[j] = Integer.parseInt(sCommand[j]);
            }
            ArrayList<MemoryBlock> emptyMemoryBlocks = new ArrayList<MemoryBlock>(50);
            for (int j=0;j<memoryBlocks.size();j++){
                if (memoryBlocks.get(j).getCurTag() == 0) {
                    emptyMemoryBlocks.add(memoryBlocks.get(j));
                }
            }


            for (int j=0;j<emptyMemoryBlocks.size();j++){
                for (int k = j+1; k<emptyMemoryBlocks.size();k++){
                    if (emptyMemoryBlocks.get(k).getCurLength()
<emptyMemoryBlocks.get(j).getCurLength()){
                        MemoryBlock tempMBlock = emptyMemoryBlocks.get(j);
                        emptyMemoryBlocks.set(j,emptyMemoryBlocks.get(k));
                        emptyMemoryBlocks.set(k,tempMBlock);

                    }
                }
            }

```

空间释放时，要对于相邻的内存块进行判断，判断具体见上文。实现如下：

```
if (temp>0 && temp<memoryBlocks.size()-1 && memoryBlocks.get(temp-1).getCurTag()==0 && memoryBlocks.get(temp+1).getCurTag()==0){//前中后合并
    int curStartPosition = memoryBlocks.get(temp-1).getStartPosition();
    int curEndPosition =  memoryBlocks.get(temp+1).getEndPosition();
    MemoryBlock cMBlock = new MemoryBlock(curStartPosition,curEndPosition,0);
    memoryBlocks.remove(temp+1);
    memoryBlocks.remove(temp);
    memoryBlocks.set(temp-1,cMBlock);
}
else if (temp>0 && memoryBlocks.get(temp-1).getCurTag()==0){//前中合并
    int curStartPosition = memoryBlocks.get(temp-1).getStartPosition();
    int curEndPosition =  memoryBlocks.get(temp).getEndPosition();
    MemoryBlock cMBlock = new MemoryBlock(curStartPosition,curEndPosition,0);
    memoryBlocks.set(temp-1,cMBlock);
    memoryBlocks.remove(temp);
}
else if (temp<memoryBlocks.size()-1 && memoryBlocks.get(temp+1).getCurTag()==0){//中后合并
    int curStartPosition = memoryBlocks.get(temp).getStartPosition();
    int curEndPosition =  memoryBlocks.get(temp+1).getEndPosition();
    MemoryBlock cMBlock = new MemoryBlock(curStartPosition,curEndPosition,0);
    memoryBlocks.set(temp,cMBlock);
    memoryBlocks.remove(temp+1);
}
else{//不合并
    int curStartPosition = memoryBlocks.get(temp).getStartPosition();
    int curEndPosition =  memoryBlocks.get(temp).getEndPosition();
    MemoryBlock cMBlock = new MemoryBlock(curStartPosition,curEndPosition,0);
    memoryBlocks.set(temp,cMBlock);
}
```

IV 总结

写完项目后，对于本系统中实现的两种算法进行一定的比较

首次适应算法（First Fit）：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。

特点： 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。

缺点：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。

最佳适应算法（Best Fit）：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。

V 结束语

本次项目中进一步加深对动态分区存储管理方式及其实现过程的理解。通过实现动态分区存储管理中首次适应算法以及最佳适应算法，体会并理解动态分区存储管理方式，受益匪浅。

VI 参考文献

1.操作系统课本

2.Github