廣東工業大學

《操作系统实验报告》

学　　院　计算机学院

专　　业　计算机科学与技术

年级班别　17 级 4 班

学　　号　3117004568

学生姓名　黄 钰 竣

辅导教师　申 建 芳

成　　绩

2019 年 12 月

实验一 进程调度

一、实验目的

用高级语言编写和调试一个进程调度程序，以加深对进程的概念及进程调度算法的理解。

二、实验内容

1. 设计一个有 N 个进程并发的进程调度程序。
2. 本实验采用了先来先服务、优先级优先以及时间片轮转三种调度算法。
3. 本实验采用的是抢占式调度，将处理机分配给一个进程后，如果作业在本时间片内未完成，则将会被重新放入就绪队列。在下一次调度时，将重新按照调度算法的规则选出应该分配处理机的作业。

三、实验运行截图

1. 主界面：

图 1.1 主界面图

2. 数据输入

此时可以填写进程的各个参数，只有进程的各个参数都填好了以后，才能正确“提

交”，否则将会产生提示信息。

图 1.2 未输入完整信息

图 1.3 未按格式输入“进入时间”信息

图 1.4 未按格式输入“需运行时间”

提交完进程后，进程在就绪队列中排队：

3. 选择调度算法；

控制面板显示单选按钮，有“先来先服务”、“优先级优先”、“时间片轮转”三种调度算法供选择。

图 1.5 选择调度算法

4. 运行：

点击“运行”，可见位于就绪队列队首的进程被选择调度开始运行：

图 1.6 运行时界面

5. 暂停执行

图 1.7 暂停时界面

6. 运行完成

进程运行完成后，将有弹窗显示，并计算出平均周转时间、平均带权周转时间

图 1.8 运行结束

7. 点击重置即可恢复到初始界面

图 1.9 点击重置后

四、关键代码

```
public void runSystem() {
        storeList.sortForFCFS(); // 对输入队列按到达时间排序，排序后就无需再比较所有元素
        int num = storeList.size();
        Time clock = storeList.getFirstTime();
        while (pcbFList.size() < num) {
            // 模拟作业按照时间到达
            for (int i = 0; i < storeList.size(); i++) {
                if (clock.compareTo(storeList.getFirstTime()) == 0) {
                    pcbWList.add(storeList.pop());
                } else {
                    // 因前面队列已经按到达时间排好序了，所以不做无用的比较
                    break;
                }
            }
            // 执行进程调度
            if (pcbWList.size() > 0) {
                // 按照各算法的要求对就绪队列排序
                switch (mode) {
                case FCFS:
                    pcbWList.sortForFCFS();
```

```
            break;
        case PRIORITY:
            pcbWList.sortForPriority();
            break;
        case SLICE_ROTATION:
            // 无需排序
            break;
        }
        ;
        // 重置引发的终止操作，此处防止抛出异常
        if (isStop) {
            break;
        }
        // 集中更新界面信息
        new Thread(new Runnable() {
            @Override
            public void run() {
                // TODO Auto-generated method stub
                changeTables(pcbWList.get(0));
                setClockOutput(clock);
            }
        }).start();
        // 暂停操作
        while (isPause) {
            try {
                Thread.sleep(1000);
            } catch (Exception e) {
                System.exit(0);// 退出程序
            }
        }
        // 重置引发的终止操作
        if (isStop) {
            break;
        }
        // 停留DELAY秒，便于观察
        try {
            Thread.sleep(DELAY);
        } catch (Exception e) {
            System.exit(0);// 退出程序
        }
        PCB pcb = pcbWList.pop();
        // 运行进程
        if (pcb.run(TIMESLICE) == Status.WAIT) {
            // 运行完一个时间片后，作业仍未完成
```

```
                pcbWList.add(pcb);
            } else {
                // 作业已完成
                pcb.setFinishedTime(clock);
                pcbFList.add(pcb);
            }
        }

        System.out.println("In " + clock.toString());
        System.out.println("存储队列: ");
        storeList.displayList();
        System.out.println("外存队列: ");
        pcbWList.displayList();
        System.out.println("完成队列: ");
        pcbFList.displayList();
        System.out.println();

        clock.increase(TIMESLICE);
    }
    // 运行完成
    stopLabel.setText("(结束)");
    button_4.setEnabled(false);
    for (int i = 0; i < 5; i++) {
        table_1.setValueAt("", 0, i);
    }
    pcbWList.updateTable(5);
    pcbFList.updateTable(5);
    }
```

实验二 作业调度

一、实验目的

用高级语言编写和调试一个或多个作业调度的模拟程序，了解作业调度在操作系统中的作用。

二、实验内容

模拟作业调度程序，分别采用短作业优先和高响应比优先作业调度算法。

三、实验运行截图

1. 主界面：

图 2.2 主界面图

2. 数据输入

此部分与实验一中的内容一致，为模块复用
此时可以填写进程的各个参数，只有进程的各个参数都填好了以后，才能正确“提交”，否则将会产生提示信息。

图 2.2 未输入完整信息

图 2.3 未按格式输入“进入时间”信息

图 2.4 未按格式输入“需运行时间”

提交完进程后，进程在就绪队列中排队：

3．选择调度算法；

控制面板显示单选按钮，有“短作业优先”和“响应比高者优先”两种调度算法供选择。

图 2.5 选择调度算法

4.运行：

点击“运行”，可见位于就绪队列队首的进程被选择调度开始运行：

图 2.6 运行时界面

5. 暂停执行

图 2.7 暂停时界面

6. 运行完成

进程运行完成后，将有弹窗显示，并计算出平均周转时间、平均带权周转时间

图 2.8 运行结束

7. 点击重置即可恢复到初始界面

图 2.9 点击重置后

四、关键代码

```
public void runSystem() {
        storeList.sortForFCFS(); // 对输入队列按到达时间排序，排序后就无需再比较所有元素
        int num = storeList.size();
        Time clock = storeList.getFirstTime();
        while (pcbFList.size() < num) {
            // 模拟作业按照时间到达
            for (int i = 0; i < storeList.size(); i++) {
                if (clock.compareTo(storeList.getFirstTime()) == 0) {
                    pcbWList.add(storeList.pop());
                } else {
                    // 因前面队列已经按到达时间排好序了，所以不做无用的比较
                    break;
                }
            }
            // 执行进程调度
            if (pcbWList.size() > 0) {
                // 按照各算法的要求对就绪队列排序
                switch (mode) {
                case SJF:
                    pcbWList.sortForSJF();;
```

```
    break;
case HRN:
    pcbWList.sortForHRN();;
    break;
}
;
// 重置引发的终止操作，此处防止抛出异常
if (isStop) {
    break;
}
// 集中更新界面信息
new Thread(new Runnable() {
    @Override
    public void run() {
        // TODO Auto-generated method stub
        changeTables(pcbWList.get(0));
        setClockOutput(clock);
    }
}).start();
// 暂停操作
while (isPause) {
    try {
        Thread.sleep(1000);
    } catch (Exception e) {
        System.exit(0);// 退出程序
    }
}
// 重置引发的终止操作
if (isStop) {
    break;
}
// 停留DELAY秒，便于观察
try {
    Thread.sleep(DELAY);
} catch (Exception e) {
    System.exit(0);// 退出程序
}
PCB pcb = pcbWList.pop();
// 运行进程
if (pcb.run(TIMESLICE) == Status.WAIT) {
    // 运行完一个时间片后，作业仍未完成
    pcbWList.add(pcb);
} else {
    // 作业已完成
```

```
                    pcb.setFinishedTime(clock);
                    pcbFList.add(pcb);
                    //计算响应比
                    for(PCB pcb1 : pcbWList.pcbList){
                        int waitedTime =
clock.compareTo(pcb1.getArrivalTime());
                        int needTime = pcb1.getNeedTime();

    pcb1.setHRN((waitedTime+needTime)*1.0/(needTime*1.0));
                    }
                }
            }

            System.out.println("In " + clock.toString());
            System.out.println("存储队列: ");
            storeList.displayList();
            System.out.println("外存队列: ");
            pcbWList.displayList();
            System.out.println("完成队列: ");
            pcbFList.displayList();
            System.out.println();

            clock.increase(TIMESLICE);
        }
        // 运行完成
        stopLabel.setText("(结束)");
        button_4.setEnabled(false);
        for (int i = 0; i < 5; i++) {
            table_1.setValueAt("", 0, i);
        }
        pcbWList.updateTable(5);
        pcbFList.updateTable(5);
        }
```

实验三 存储管理

一、实验目的

通过编写和调试存储管理的模拟程序以加深对存储管理方案的理解。熟悉虚存管理的各种页面淘汰算法。

二、实验内容

设计一个有空闲分区分配的存储管理方案，模拟实现分区的分配与回收过程。
本实验采用了首次适应算法、循环首次适应算法、最佳适应算法、最坏适应算法共四种算法。

本程序在每次为新的作业分配内存空间时，将按照动态分区分配算法找出分配分区，然后占用分区的起始部分。人为规定，分区允许的最小单位为 5（即如果过检测到分配后，分区的大小小于 5，则分配失败）

程序的内存回收部分，按照回收机制的 4 种情况，进行回收，将使用第一个分区的名字作为合并后的分区名

三、实验运行截图

1. 主界面：

图 3.1 主界面图

2. 数据输入

只有正确输入想要分配/回收的信息，程序才能显示出结果，如果过输入不正确会有弹窗提示。

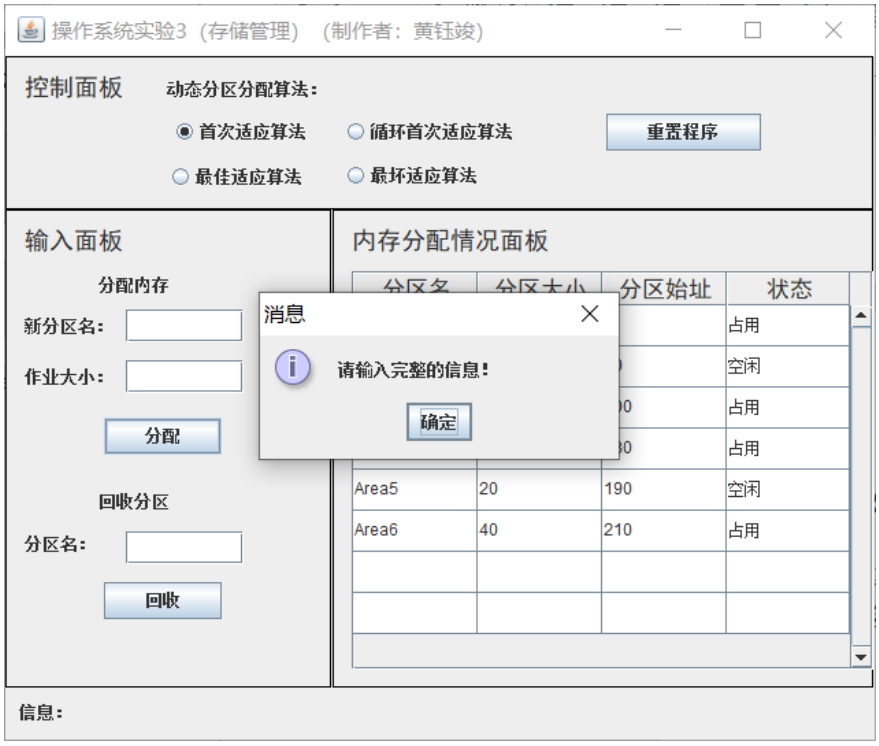

图 3.2 未输入完整信息

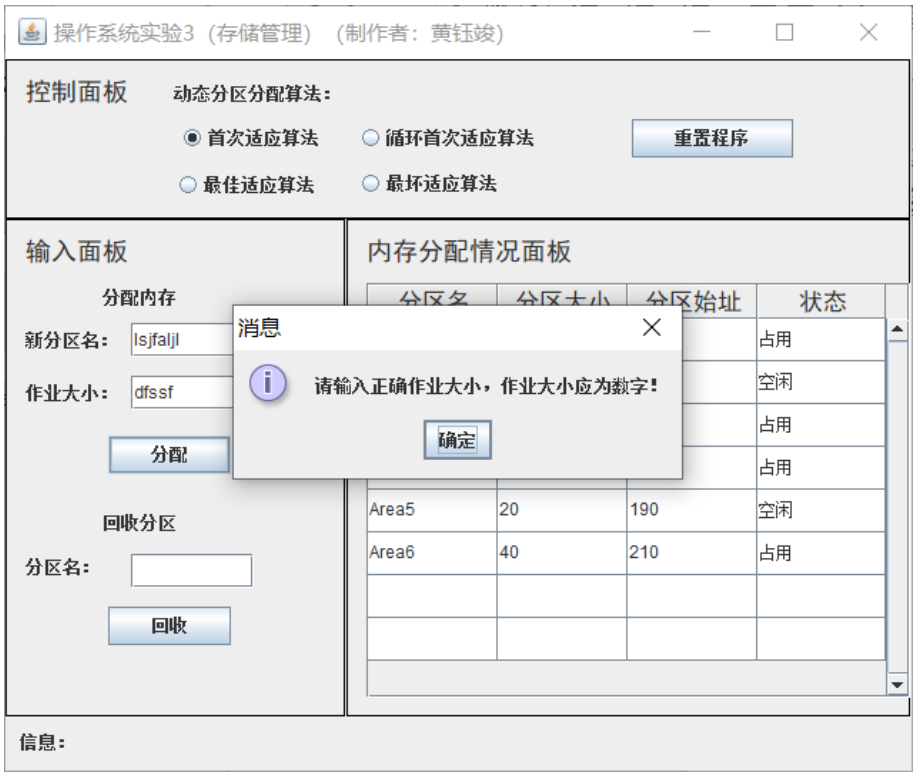

图 3.3 未按格式输入“作业大小”信息

图 3.4 输入了空闲分区链中不存在的分区名

3. 测试首次适应算法

按照已有分区设计作业名为“FFF”,大小为 5，程序运行结果如下图所示。程序结果符合算法“优先利用内存低地址部分的空闲区”的要求。

图 3.5 按照首次适应算法分区测试样例 1 结果图

在上一次运行的基础上，再分配一个分区名为“MMM”，大小为 5 的作业，运行结果如下图所示。再次验证了“优先分配低地址空闲空间”的要求。

图 3.6 按照首次适应算法分区测试样例 2 结果图

最后再分配一个分区名为“MAX”，大小为 1000 的作业，结果为：

图 3.7 按照首次适应算法分区测试样例 3 结果图

4. 测试循环首次适应算法

重置界面（即程序数据设置回初始值后）并选中“循环首次适应算法”后，输入测试分区名为“AAA”，大小为 5 的分区。

图 3.8 测试循环首次适应算法用例 1

在上一次的基础上，再输入分配用例（“BBB”,5）程序结果为下图所示，可见符合本算法要求的“从上一次找到的空闲分区的下一个空闲分区开始查找”。

图 3.9 测试循环首次适应算法用例 2

继续在上一次输入的基础上，再输入用例（“CCC”，5）程序执行结果如下图所示。可见，本程序符合算法的“循环”要求

图 3.10 测试循环首次适应算法用例 3

最后，再输入样例（“MAX”，1000），测试程序对于不可分配作业的运行结果。结果如下图所示，可见程序满足。

图 3.11 测试循环首次适应算法用例 4

5. 测试最佳适应算法

重置界面，并选择“最佳适应算法”后，输入测试用例（“AAA”，5），程序运行的结果如下图所示，可见程序符合算法“从空闲队列中总是优先分配能满足要求，且又是最小的空闲分区”的思想。

图 3.12 测试最佳适应算法用例 1

在上一次输入的基础上，再输入测试用例（“BBB”,6），再次验证程序执行结果。分配结果如下图所示，程序满足算法的分配要求。

图 3.13 测试最佳适应算法用例 2

最后，再测试程度对“不可满足作业”的分配情况，可见程序符合预期要求。

图 3.14 测试最佳适应算法用例 3

6. 测试最坏适应算法

重置界面，并选择“最坏适应算法”，输入测试用例（“AAA”，8），验证程序是否符合算法“总是优先挑选一个最大的空闲分区”的要求。程序运行结果如下图所示，可见程序符合要求。

图 3.15 测试最坏适应算法用例 1

在上一次输入的基础上，再次输入测试用例（“BBB”，10），以再次验证程序的正确性，程序运行结果如下图所示，可见程序符合要求。

图 3.16 测试最坏适应算法用例 2

最后，再测试程序对于“不可分配作业”的执行情况，输入用例（“MAX”，1000）运行结果如下图所示。程序结果符合预期。

图 3，17 测试最坏适应算法用例 3

7. 测试内存回收机制（四种情况）

回收前，分区链的情况如下图所示

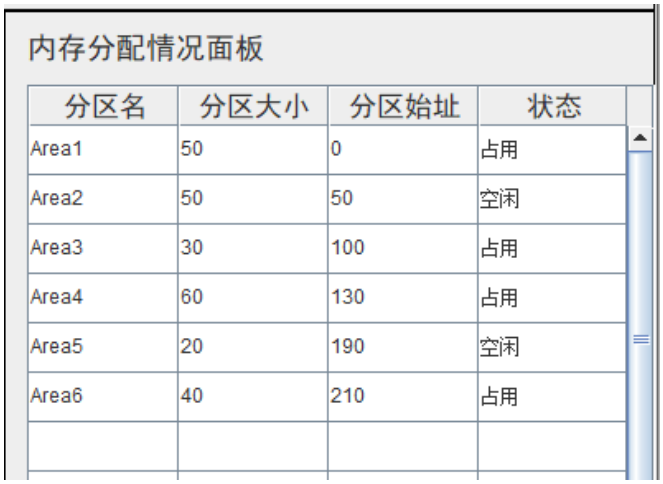

图 3.18 回收前分区链情况

对第一种情况，即“回收区的前一个分区为空闲分区”。若回收分区“Area3”正好符合本情况，回收后“Area3”应与“Area2”合并，合并后分区名为“Area2”，分区“Area3”移出分区链。设置回收用例为（“Area3”），程序执行结果如下图所示，回收结果符合预期。

图 3.19 测试第一种情况用例运行结果

对第二种情况，即“回收区的前一个和后一个分区均为空闲分区”，应将三个分区合并为一个。发现在测试完第一种情况后，“Area4”符合要求，设计回收测试用例为（“Area4”），程序运行结果应为，“Area2”、“Area4”、“Area4”合并为一个空闲分区“Area2”。

程序运行的结果如下图所示，可见符合预期。

图 3.20 测试第二种情况用例运行结果

对第三种情况，即“回收区的后一个分区为空闲分区”。发现在测试完第二种情况的分区链种，“Area1”符合要求，设计回收测试用例为（“Area1”），回收后，“Area1”、“Area2”应合并为一个分区“Area1”。

程序执行结果如下图所示，可见程序运行结果符合预期。

图 3.21 测试第三种情况用例运行结果图

对第四种情况，即“回收分区前后均无空闲分区”。设计的测试用例为，在“重置程序”后，使用“最佳适应算法”先分配一个作业为（“Test”，10），使其形成如下图所示的分区链情况。

图 3.22 构造第三种回收情况

分配后，“Area4”将满足第四种情况的要求，使用回收用例（“Area4”），回收后“Area4”的状态应改为“空闲”。程序执行结果如下图所示，可见程序符合要求。

”

四、关键代码

1. 分配内存空间部分代码

```
public boolean distribute(String newName, int size) {
        MyNode node = header;
        MyNode myParent;
        if (mode == FF) {
            // 使用首次适应算法
            while (node != null) {
                if (node.getData().getStatus().equals(Status.EMPTY)) {
                    // 若分区未被占用，则尝试分区
                    StoreArea area = node.getData().splitArea(newName,
size);
                    if (area != null) {
                        // 说明该分区可再细分，分区成功
                        myParent = node.parent;
                        if (myParent == null) {
                            // 说明为头节点
                            MyNode newNode = new MyNode(area);
                            newNode.insert(node);
                            header = newNode;
                            noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                            return true;
                        } else {
                            // 说明不是头节点
                            MyNode newNode = new MyNode(area);
                            myParent.insert(newNode);
                            noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                            return true;
                        }
                    }
                }
                node = node.next;
            }
        } else if (mode == NF) {
            // 使用最佳适应算法
            if (NFNode == null) {
                NFNode = header;
            }
            // 记下当前指针指向的对象，防止在陷入死循环
```

```
            MyNode label = NFNode;
            // 其余部分代码与首次适应算法相似,只是将node改为NFNode
            while (true) {
                if (NFNode.getData().getStatus().equals(Status.EMPTY)) {
                    // 若分区未被占用，则尝试分区
                    StoreArea area = NFNode.getData().splitArea(newName,
size);
                    if (area != null) {
                        // 说明该分区可再细分，分区成功
                        myParent = NFNode.parent;
                        if (myParent == null) {
                            // 说明为头节点
                            MyNode newNode = new MyNode(area);
                            newNode.insert(NFNode);
                            header = newNode;
                            noteLabel.setText(newName+" 成功从 "
+NFNode.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                            // 按照算法思想，将指针指向下一个空闲区
                            NFNode = NFNode.next;
                            return true;
                        } else {
                            // 说明不是头节点
                            MyNode newNode = new MyNode(area);
                            myParent.insert(newNode);
                            noteLabel.setText(newName+" 成功从 "
+NFNode.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                            // 按照算法思想，将指针指向下一个空闲区
                            NFNode = NFNode.next;
                            return true;
                        }
                    }
                }
                NFNode = NFNode.next;
                if (NFNode == null) {
                    // 说明遍历到链表尾
                    NFNode = header;
                }
                if (NFNode == label) {
                    // 说明已经将链表遍历了一次，仍找不到合适分区
                    noteLabel.setText(" 空闲分区链表中无合适的分区可供
"+newName+"分配使用");
                    return false;
                }
            }
```

```
        } else if (mode == BF) {
            // 采用最佳适应算法
            // 首先遍历记下满足要求，且又是最小的空闲分区
            MyNode temp = header;
            while (temp != null) {
                // 判断该节点是否满足要求，即大小合适且为空闲块
                if (temp.getData().isFit(size)) {
                    if (!temp.getData().isBiger(node.getData())) {
                        // 比已记录节点大，则记下当前节点
                        node = temp;
                    }
                }
                temp = temp.next;
            }
            // 再次确保最小块不被占用后，开始分配
            if (node.getData().getStatus().equals(Status.EMPTY)) {
                StoreArea area = node.getData().splitArea(newName, size);
                if (area != null) {
                    // 说明该分区可再细分，分区成功
                    myParent = node.parent;
                    if (myParent == null) {
                        // 说明为头节点
                        MyNode newNode = new MyNode(area);
                        newNode.insert(node);
                        header = newNode;
                        noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                        return true;
                    } else {
                        // 说明不是头节点
                        MyNode newNode = new MyNode(area);
                        myParent.insert(newNode);
                        noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                        return true;
                    }
                }
            }

        } else if (mode == WF) {
            // 采用最坏适应算法，代码与前面相似，只是判断条件不同
            // 首先遍历记下满足要求，且又是最大的空闲分区
            MyNode temp = header;
            while (temp != null) {
```

```
                // 判断该节点是否满足要求，即大小合适且为空闲块
                if (temp.getData().isFit(size) &&
temp.getData().getStatus().equals(Status.EMPTY)) {
                    //若第一个节点node不为空闲分区，则找到第一个空闲且合适的分
区，赋值给node
                    if(node.getData().getStatus().equals(Status.FULL)){
                        node=temp;
                        continue;
                    }
                    if (temp.getData().isBiger(node.getData())) {
                        // 比已记录节点大，则记下当前节点
                        node = temp;
                    }
                }
                temp = temp.next;
            }
            // 再次确保最小块不被占用后，开始分配
            if (node.getData().getStatus().equals(Status.EMPTY)) {
                StoreArea area = node.getData().splitArea(newName, size);
                if (area != null) {
                    // 说明该分区可再细分，分区成功
                    myParent = node.parent;
                    if (myParent == null) {
                        // 说明为头节点
                        MyNode newNode = new MyNode(area);
                        newNode.insert(node);
                        header = newNode;
                        noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                        return true;
                    } else {
                        // 说明不是头节点
                        MyNode newNode = new MyNode(area);
                        myParent.insert(newNode);
                        noteLabel.setText(newName+" 成功从 "
+node.getData().getName()+" 中分得了大小为"+size+"的内存空间");
                        return true;
                    }
                }
            }
        }
        return false;
    }
```

2. 回收内存空间部分代码

```
// 回收指定名字的作业
    public boolean recycle(String name) {
        // 先定位到指定的节点
        MyNode node = header;
        boolean isRecycled=false;
        while (node != null) {
            if (node.getData().getName().equals(name)) {
                break;
            }
            node = node.next;
        }
        // 若无法找到，则说明不存在该节点
        if (node == null) {
            noteLabel.setText("空闲分区链表中不存在"+name+"节点");
            return false;
        }
        // 若找到的节点为空闲分区，则返回false
        if (node.getData().getStatus().equals(Status.EMPTY)) {
            noteLabel.setText(name+"不是一个占用分区，不需要回收");
            return false;
        }
        // 已找到指定名字的节点，先记录下该节点的父节点和子节点
        MyNode myParent = node.parent;
        MyNode myNext = node.next;
        String message = "回收的分区有："+name;
        // 判断前面的节点是否可以合并
        if(myParent!=null){
            if (myParent.getData().getStatus().equals(Status.EMPTY)) {
                // 说明前一个分区可以合并，则合并
                StoreArea area = node.deleteNode(); // 删除了该节点
                myParent.getData().merge(area);
                node = myParent; // 方便后面合并下一个分区
                message = message +"、"+myParent.getData().getName();
                isRecycled=true;
            }
        }
        if(myNext!=null){
            if (myNext.getData().getStatus().equals(Status.EMPTY)) {
                // 说明后一个分区可以合并
                StoreArea area1 = myNext.deleteNode(); // 删除了子节点
                node.getData().merge(area1);
                node.getData().setStatus(Status.EMPTY);
```

```
                message = message + "、"+area1.getName();
                isRecycled=true;
            }
        }
        if(!isRecycled){
            node.getData().setStatus(Status.EMPTY);
        }
        noteLabel.setText(message);
        return true;
    }
```