操作系统第二次作业

李文玥 1750803

一、问题分析

选题：动态分区分配方式的模拟

任务：对作业进行分配与释放，动态管理内存，并显示每次分配和回收后的空闲分区链的情况。

作业	活动	大小
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

二、算法设计

1.首次适应法

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。

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。

2.最佳适应法

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。

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。

三、实现

1.开发/运行环境

开发语言：html、css、javascript

开发工具：VSCode、Chrome

2.核心算法实现

数据结构：使用链表来模拟内存的分配与释放。由于链表便于插入删除，这一特性使得任务内存的分配与释放变得灵活且快速。

类	类的说明
class Node	结点类，含有左指针、右指针、内存块地址起点、长度以及占用情况
class List	链表类，含有头指针和尾指针，作为总内存存在。

核心代码：

```
class Node{
    constructor(pos,len,sta){
        this.position=pos;
        this.length=len;
        this.statement=sta;
        this.pre=null;
        this.next=null;
    }
}
```

```
class List{
    constructor(){
        this.head=new Node;
        this.tail=new Node;
        this.head.next=this.tail;
        this.tail.pre=this.head;
        this.head.pre=null;
        this.tail.next=null;
        this.size=0;
    }
    findNode=(pos)=>{}//找到本位置的结点
    findPos=(pos)=>{}//配合释放内存，找到要释放的结点
    insert=(pos,node)=>{}//插入结点
    addBack=(node)=>{}//将结点压到链表最后
    clear=()=>{}//清空链表
    remove=(node)=>{}//删除结点
    firstFit=(value)=>{}//首次适应算法
    bestFit=(value)=>{}//最佳适应算法
    setFree=(i)=>{}//释放内存
    show=()=>{}//可视化展示目前内存分配状态
}
```

算法实现：

首次适应算法：遍历链表，直到查找到第一个空间足够的内存区间，分配相应的内存，更新内存情况；否则返回内存空间不足。

```
firstFit=(value)=>{
        //首次适应算法
        let temp=this.head.next;
        let i=0;
        for(i=1;i<=this.size;i++){
            if(temp.statement===true&&temp.length>=value){
                let pos=temp.position;
                let newtemp=new Node(pos,value,false);
                if(temp.length>value){
                    temp.position=pos + value;
                    temp.length=temp.length-value;
                    temp.pre.next=newtemp;
                    newtemp.pre=temp.pre;
                    newtemp.next=temp;
                    temp.pre=newtemp;
                    this.size+=1;
                    console.log("show:",this.size);
                }
                else if(temp.length===value){
                    temp.statement=false;
                }
                console.log("firstFit:",value);
                return;
            }
            temp=temp.next;
        }
        return alert("空间不足");
    }
```

最佳适应算法：遍历链表，直到查找到一个最小的空间足够的内存区间，分配相应的内存，更新内存情况；否则返回内存空间不足。

```
bestFit=(value)=>{
    //最佳适应算法
        let tempFit=new Node();
        tempFit=null;
        let temp=this.head.next;
        let i=0;
        for(i=1;i<=this.size;i++){
            if(temp.statement===true&&temp.length>=value){
                if(tempFit===null){
                    tempFit=temp;
                }
                else if(tempFit.length>temp.length){
                    tempFit=temp;
                }
            }
            temp=temp.next;
        }
        if(tempFit===null)
        return alert("空间不足");
        else{
            let pos=tempFit.position;
                let newtemp=new Node(pos,value,false);
                if(tempFit.length>value){
                    tempFit.position=pos+value;
                    tempFit.length=tempFit.length-value;
                    tempFit.pre.next=newtemp;
                    newtemp.pre=tempFit.pre;
                    newtemp.next=tempFit;
                    tempFit.pre=newtemp;
                    this.size+=1;
                    console.log("show:",this.size);
                }
                else if(tempFit.length===value){
                    tempFit.pre.next=newtemp;
                    newtemp.pre=tempFit.pre;
                    newtemp.next=tempFit.next;
                    tempFit.next.pre=newtemp;
                    this.remove(tempFit);
                }
                console.log("bestFit:",value);
                return;
        }
    }
```

内存释放算法：首先检测当前结点左结点是否为空闲，若是，则合并；检测右结点是否空闲，若是，则合并；设置当前结点状态为空闲。

```
setFree=(i)=>{
        console.log("要释放哪个结点：",i);
        let temp=this.findNode(i);
        let tempPre=temp.pre;
        let tempNext=temp.next;
        if(tempPre.statement===true){
            tempPre.length+=temp.length;
            tempPre.next=tempNext;
            tempNext.pre=tempPre;
            temp=tempPre;
            this.size-=1;
        }
        if(tempNext.statement===true){
            temp.length+=tempNext.length;
            temp.next=tempNext.next;
            tempNext.next.pre=temp;
            temp.statement=true;
            this.size-=1;
        }
        temp.statement=true;
        return;
    }
```

3.可视化实现

使用html、css、javascript+jquery实现。

核心内存状态可视化使用自定义函数 `show()` ，每次调用首先初始化可视化示图，再遍历链表，查看内存块状态，并进行可视化显示。

```
show=()=>{
        let i=0;
        for(i=1;i<64;i++){
            $("#memShow").children().eq(i).attr("class", "table-primary");
            $("#memShow").children().eq(i).text(" ");
        }
        console.log("show多少结点",this.size);
        let temp=this.head.next;
        let j=0;
        for(j=1;j<=this.size;j++){
            let pos=temp.position;
            let len=temp.length;
            let sta=temp.statement;
            if(sta){
                for(i=pos/10;i<=pos/10+len/10;i++){
                    $("#memShow").children().eq(i).attr("class", "table-info");//设置空闲颜色
                }
            }
            else if(sta===false){
                for(i=pos/10;i<=pos/10+len/10;i++){
                    $("#memShow").children().eq(i).attr("class", "table-warning");//设置占用颜色
                }

            }
            $("#memShow").children().eq(pos/10).text(pos);
            $("#memShow").children().eq(pos/10+len/10-1).text("|");//前后位置显示
            console.log("show",pos,len,sta);
            temp=temp.next;
        }
    }
```

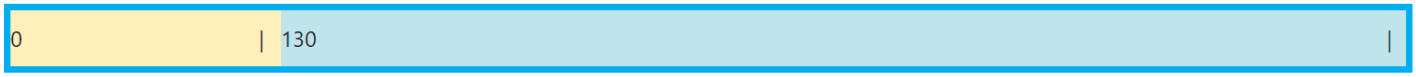
四、运行展示

0.初始界面

1.首次适应算法

首次适应算法 最佳适应算法 重置分配方式

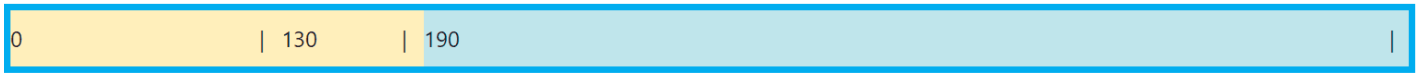
130 KB 分配



位置 Free!

首次适应算法 最佳适应算法 重置分配方式

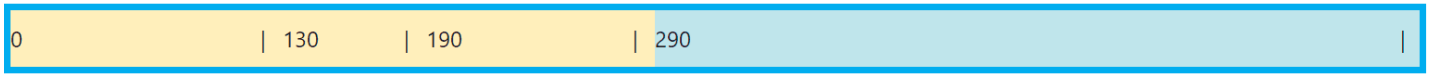
60 KB 分配



位置 Free!

首次适应算法 最佳适应算法 重置分配方式

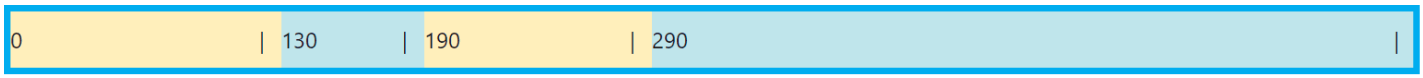
100 KB 分配



位置 Free!

首次适应算法 最佳适应算法 重置分配方式

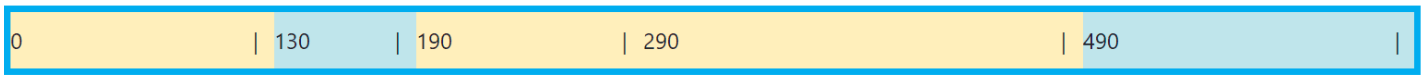
100 KB 分配



130 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

200 KB 分配



130 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

200 KB 分配

190 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

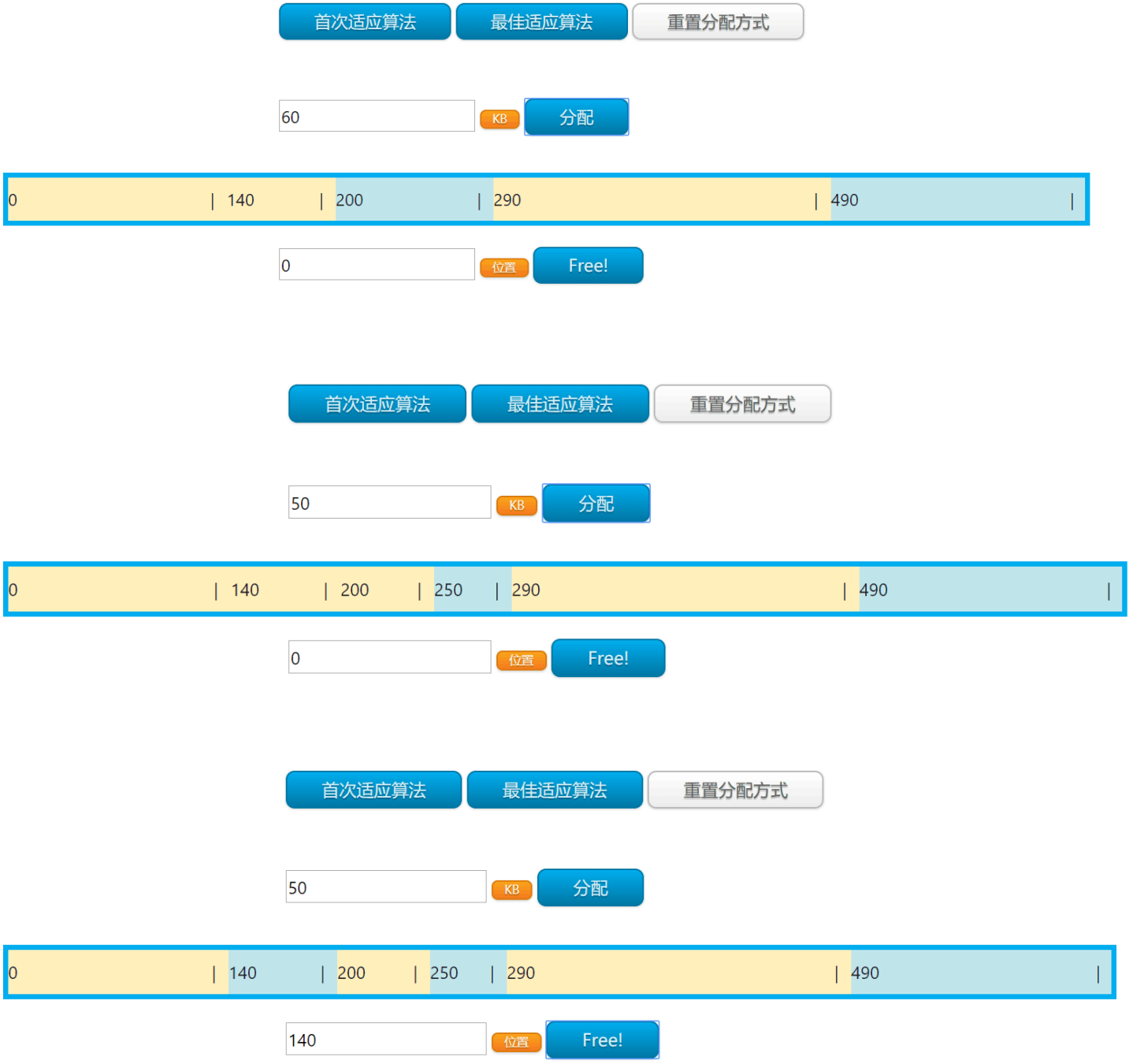
200 KB 分配

0 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

140 KB 分配

0 位置 Free!

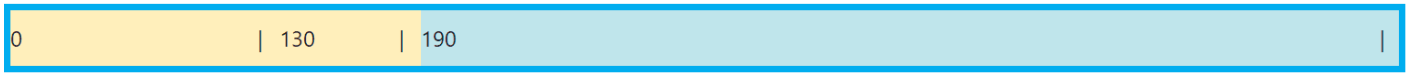

首次适应算法
最佳适应算法
重置分配方式
60
KB
分配
0 | 140 | 200 | 290 | 490 |
0
位置
Free!
首次适应算法
最佳适应算法
重置分配方式
50
KB
分配
0 | 140 | 200 | 250 | 290 | 490 |
0
位置
Free!
首次适应算法
最佳适应算法
重置分配方式
50
KB
分配
0 | 140 | 200 | 250 | 290 | 490 |
140
位置
Free!

2.最佳适应算法


首次适应算法
最佳适应算法
重置分配方式
130
KB
分配
0 | 130 |
位置
Free!

首次适应算法 最佳适应算法 重置分配方式

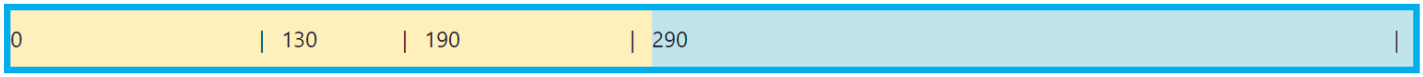
60 KB 分配



位置 Free!

首次适应算法 最佳适应算法 重置分配方式

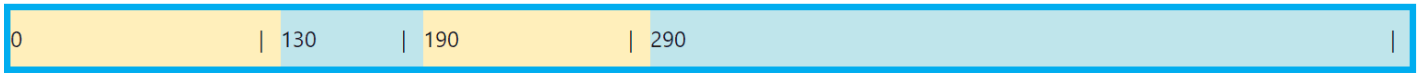
100 KB 分配



位置 Free!

首次适应算法 最佳适应算法 重置分配方式

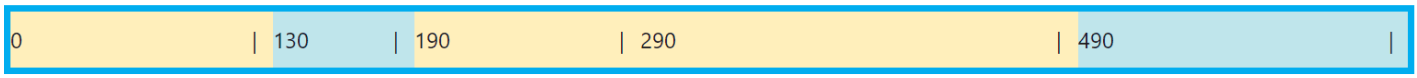
100 KB 分配



130 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

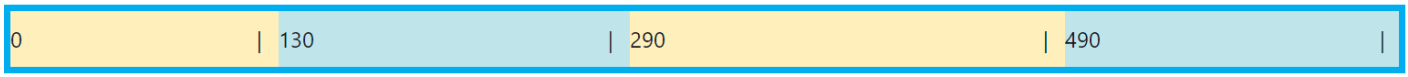
200 KB 分配



130 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

200 KB 分配



190 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

200 KB 分配

0 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

140 KB 分配

0 位置 Free!

首次适应算法 最佳适应算法 重置分配方式

60 KB 分配

0 位置 Free!

3.细节问题

未选择算法（每次只能选取一种算法执行到底，可以按重置分配方式按钮进行刷新、重新设置）

内存不足

不可释放（内存释放输入框输入了空闲结点位置或输入非结点Position值）