OS-MemoryManagement

Second assignment of OS course, implementing the memory management algorithm.

一、系统概述

1.1需求分析

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。

1.2系统功能

1.2.1内存调度

> 由于我觉得PPT要求中回收内存块还需要知道该内存块的大小是很愚蠢的，因此对项目功能进行了改动。有一个640k的内存块，用户必须先选择使用最优适应或最快适应，然后分配内存只需填入想要分配的内存块大小，系统就会自动分配内存块并安排编号，当用户想要收回内存块时，只需要填入内存块的编号就会删除对应内存块，无需记住内存块大小

1.3开发工具

> PyQt 进行开发

> PyCharm作为开发工具

> 运行在Windows系统上。

二、代码设计

2.1窗体类

```
class Example(QWidget):
    #默认属性和参数
    (mem_width,mem_height)=(150,1280)
    free_color=QColor(240, 255, 255, 100)
    busy_color=QColor(255,215,0,100)
    (start_x,start_y)=(0,0)
    ratio=2
```

```
    (MaxIndex,BlockIndex)=(640,1)
    #输入控件
    RequiredMemory = None
    RequiredIndex = None
    SubmitMemory=None
    SubmitIndex=None


    #已经被分配的内存块列表，paintEvent根据这个队伍扫描
    DeployList=None
    #由于被已经被分配的内存块分割产生的空闲块表
    FreeList=None

    def __init__(self):
        super().__init__()
        self.DeployList=[]
        self.FreeList=[]
        #初始化空闲表
        InitialFreeBlock=MemoryBlock()
        InitialFreeBlock.setProperty(addr=0,size=640,index=0,isDeploy=False)
        self.FreeList.append(InitialFreeBlock)
        self.initUI()

    def initUI(self):
        self.setGeometry(900, 900, 2400, 1500)
        self.setWindowTitle('MemoryManagement')
        self.start_x = (self.width() - self.mem_width) / 2
        self.start_y=self.height()*0.1
        self.setBasicNotes()
        self.createInputPanal()

        self.show()
```

窗体类继承于QWidget，类参数都是窗体的尺寸等参数，Deployment和FreeList记录了已分配内存块和空闲内存的内容，在构造函数中initUI负责UI创建，setBasicNotes()和createInputPanal负责标签和输入面板的创建。

2.2内存分配逻辑

2.2.1分配内存

```
    def allocateStrategy(self):
        scale=int(self.RequiredMemory.text())
        self.RequiredMemory.clear()
        if self.ratiobutton1.isChecked():
            print("strategy_1")
            block=MemoryBlock()
            for space in self.FreeList:
                print("space "+str(space.address))
                if space.size>=scale:
                   block.setProperty(addr=space.address,size=scale,index=1,isDeploy=True)
                    self.DeployList.append(block)
```

```
                    space.address=space.address+block.size
                    space.size=space.size-block.size
                    break
        elif self.ratiobutton2.isChecked():
            print("strategy_2")
            block=MemoryBlock()
            anchor,gap=-1,640
            for i in range(len(self.FreeList)):
                if self.FreeList[i].size-scale>=0 and self.FreeList[i].size-scale<gap:
                    anchor =i
                    gap=self.FreeList[i].size-scale
            if anchor!=-1:

 block.setProperty(addr=self.FreeList[anchor].address,size=scale,index=1,isDeploy=True)
                self.DeployList.append(block)
                self.FreeList[anchor].size=self.FreeList[anchor].size-scale
                self.FreeList[anchor].address=self.FreeList[anchor].address+scale
        else:
            QMessageBox.information(self, "提示","请先选择分配策略",QMessageBox.Yes |
QMessageBox.No)
        self.update()
```

系统根据按钮传输的值选择最优适配和最先适配，在最先适配中，寻找FreeList中的空闲块，找到第一个可以放入的块，修改该空闲块的起始位置和大小，然后在DeployList新加入一块；在最优适配中，遍历所有内存块，寻找可以放入新分配内存块的空闲块中内存最小的，然后修改大小和起始位置，并在Deployment中加入新的新的内存块

2.2.2 释放内存

```
    def recycleStrategy(self):
        recycler=int(self.RequiredIndex.text())
        self.RequiredIndex.clear()
        for i in range(len(self.DeployList)):
            if recycler==self.DeployList[i].index:
                self.FreeList.append(self.DeployList[i])
                while recycler in MemoryBlock.IndexPool:
                    MemoryBlock.IndexPool.remove(recycler)
                self.DeployList.pop(i)
                self.update()
                break
        cmpfun = operator.attrgetter('address')
        self.FreeList.sort(key=cmpfun)

        for j in range(len(self.FreeList)-1,0,-1):
            if self.FreeList[j].address==self.FreeList[j-1].address+self.FreeList[j-
1].size:
                print("find_one")
                self.FreeList[j-1].size=self.FreeList[j-1].size+self.FreeList[j].size
                self.FreeList.pop(j)
```

回收过程中在Deployment中找到要释放的内存，吧该对象放入FreeList，然后对FreeList执行对相邻空闲块的合并操作

2.2.3 绘图逻辑

```
    def paintEvent(self, e):
        #print("evoke")
        qp = QPainter()
        qp.begin(self)
        self.drawRectangles(qp,self.free_color)
        for task in self.DeployList:
            self.allocateMemory(qp,task)
        qp.end()
```

对于Deployment的每一项，通过其初始地址和内存大小，设置矩形大小，每一次DeployList发生变化，调用Update函数对视图进行更新。

2.3 函数和类

MemoryBlock类

内存块的抽象

函数名	作用
setProperty	设置内存的地址和大小

成员变量	作用
address	内存块的地址
size	内存块的大小

Example类

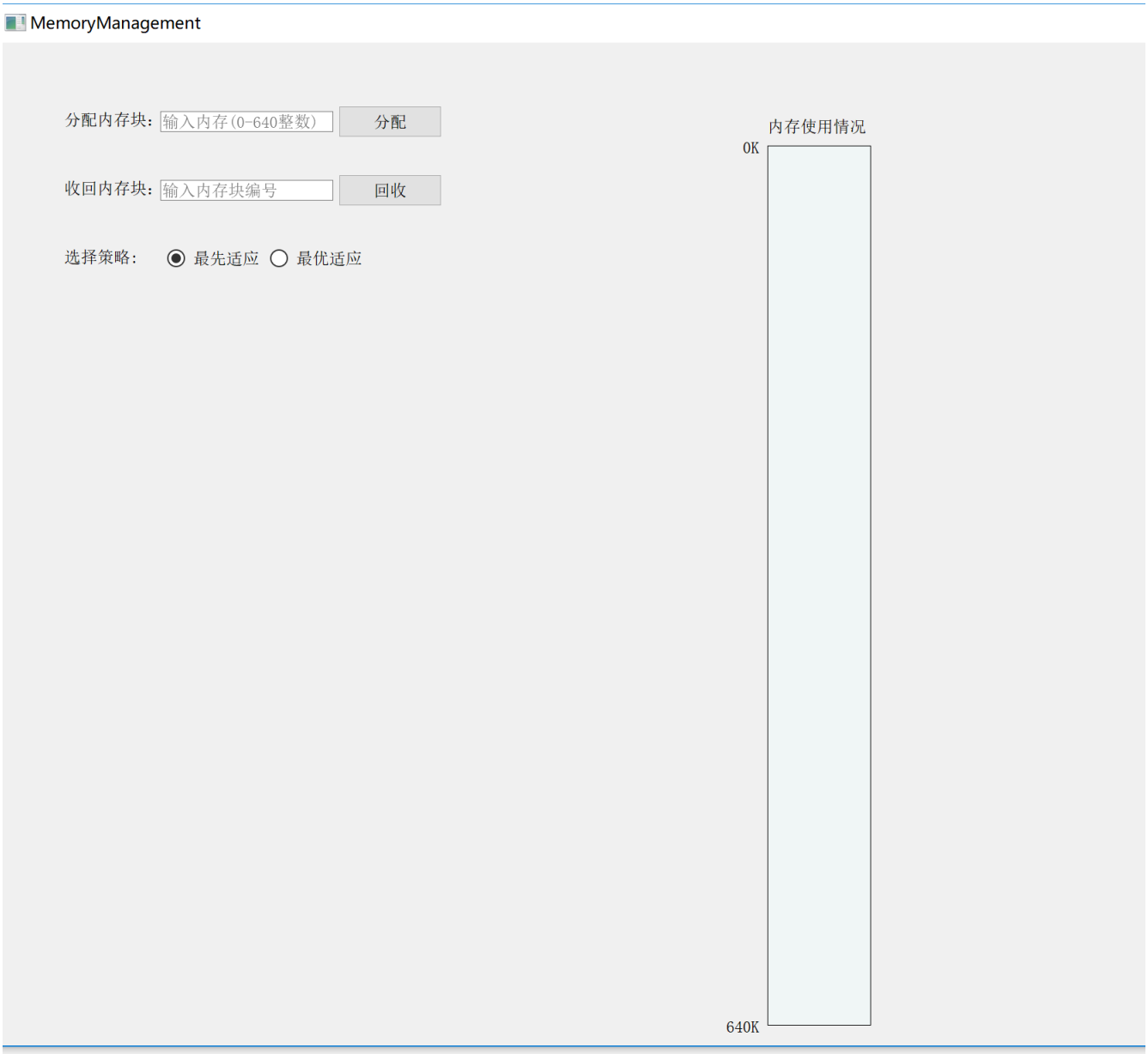
窗口主体类

函数名	作用
allocateStrategy	执行分配内存块策略
recycleStrategy	执行回收内存块策略
allocateMemory	分配内存
paintEvent	执行每次内存块视图的更新

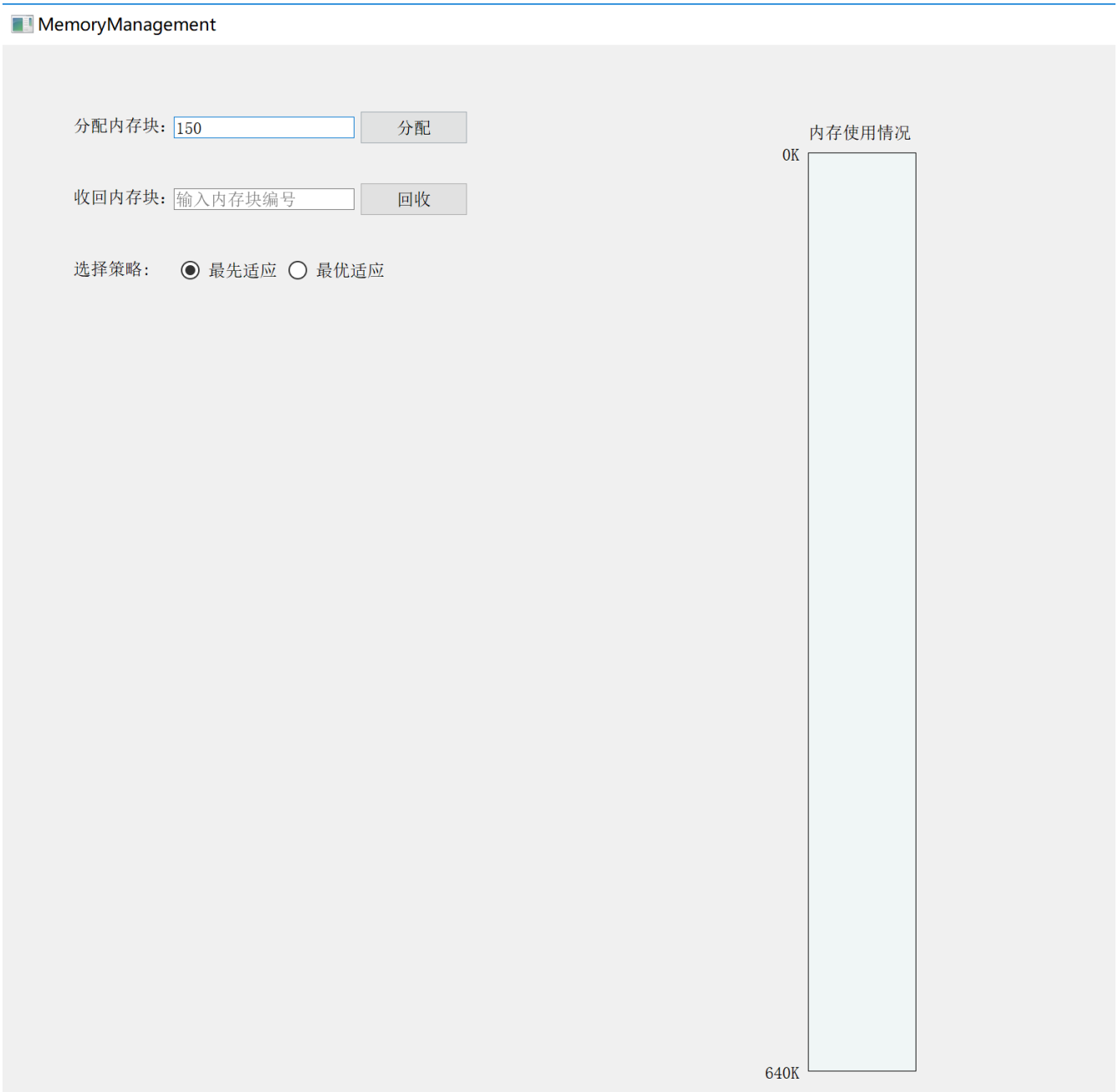
成员变量	作用
DeployList	记录已经被分配的内存块
RecycleList	保存每层内请求的监听器

三、运行效果

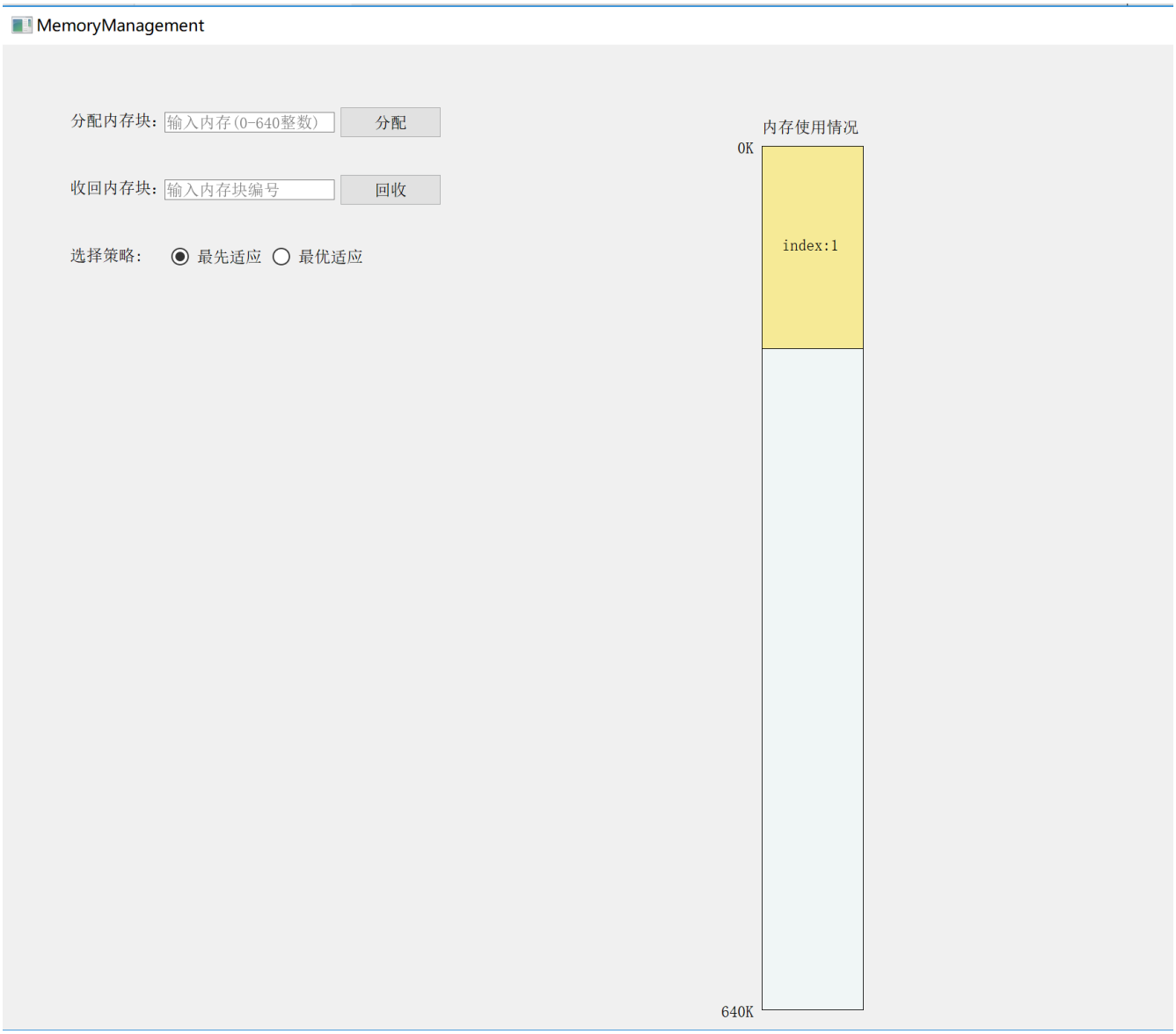
初始未分配内存



输入要分配的内存块大小


MemoryManagement
分配内存块：
150
分配
收回内存块：
输入内存块编号
回收
选择策略：
最先适应
最优适应
内存使用情况
0K
640K

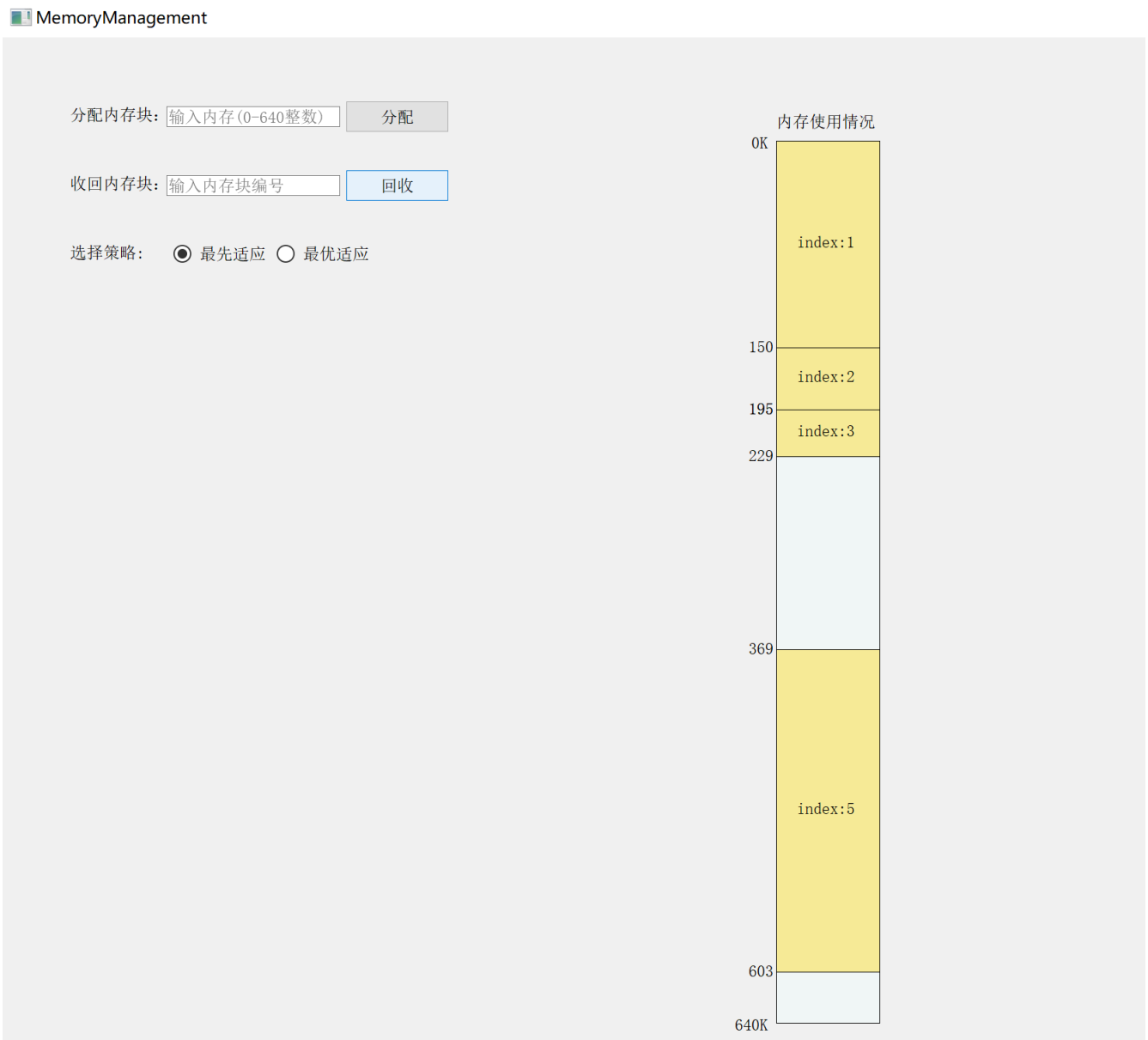
内存被分配


MemoryManagement
分配内存块：
输入内存(0-640整数)
分配
收回内存块：
输入内存块编号
回收
选择策略：
最先适应
最优适应
内存使用情况
0K
index:1
640K

输入要回收的内存编号


MemoryManagement
分配内存块：
输入内存(0-640整数)
分配
收回内存块：
4
回收
选择策略：
最先适应
最优适应
内存使用情况
0K
index:1
150
index:2
195
index:3
229
index:4
369
index:5
603
640K

内存被回收



四、分析

对于操作系统有了更深的理解，用python模拟内存分配，所有的请求是内存，我们模拟内存分配的问题，也就是模拟所有的内存资源如何分配给进程，内存应该分配到哪里的内容。 通过本程序，我第一次了解PyQt的使用，并加深了对Python面向对象的理解。 当然这个程序还是有些地方写得不够好的,设计时思路有一些混乱,没有很好地降低耦合性.