内存管理实验

一、实验目的

1、掌握对内存空间进行动态分区分配和回收的方法。

2、能够通过自主学习学习实验相关知识，并解决实验中遇到的具体问题。

二、实验结果及分析

功能模块：

```python
def allocate(flag, memory, Allocated_Memory, process):
    if flag == 1:
        temp = first_fit(memory, Allocated_Memory, process)
        if temp is None:
            print('分配失败')
        else:
            return temp
    elif flag == 2:
        temp = next_fit(memory, Allocated_Memory, process)
        if temp is None:
            print('分配失败')
        else:
            return temp
    elif flag == 3:
        temp = best_fit(memory, Allocated_Memory, process)
        if temp is None:
            print('分配失败')
        else:
            return temp
    else:
        temp = worst_fit(memory, Allocated_Memory, process)
        if temp is None:
            print('分配失败')
        else:
            return temp
```

内存分配

```python
# 定义内存回收算法
8 用法
def memory_recycle(memory, Allocated_Memory, process):
    # 参数：memory是一个空闲分区链表，process是一个进程对象
    # 返回值：如果回收成功，返回一个回收后的空闲分区链表，否则返回None
    current = Allocated_Memory.head  # 定义一个当前结点，从头结点开始查找
    while current is not None:
        if current.process == process:  # 如果当前分区的进程是要回收的进程
            temp = memory.head
            index = current.start + current.size
            while temp is not None:
                if temp.start == index:  # 找到与释放内存相邻的内存块
                    temp.start = current.start
                    temp.size = current.size + temp.size
                    Allocated_Memory.remove(current)
                    return memory
                temp = temp.next
            temp2 = Partition(current.start, current.size, 'free', None)  # 没找到相邻内存块就直接加入
            memory.append(temp2)
            Allocated_Memory.remove(current)
            return memory
        current = current.next  # 继续查找下一个分区
    return None  # 如果没有找到要回收的进程，回收失败
```

内存回收

```python
# 定义一个初始内存空间，共有5个分区，大小分别为130, 70, 140, 80, 20
memory = PartitionList()  # 创建一个空闲分区链表
memory.append(Partition(0, 130, "free", None))  # 添加第一个空闲分区
memory.append(Partition(130, 70, "free", None))  # 添加第二个空闲分区
memory.append(Partition(200, 140, "free", None))  # 添加第三个空闲分区
memory.append(Partition(340, 80, "free", None))  # 添加第四个空闲分区
memory.append(Partition(420, 20, "free", None))  # 添加第五个空闲分区

# 定义一个已被分配内存的链表
Allocated_Memory = PartitionList()

# 定义一些测试用的进程
p1 = Process("P1", 50)  # 进程P1，大小为50
p2 = Process("P2", 60)  # 进程P2，大小为60
p3 = Process("P3", 40)  # 进程P3，大小为40
p4 = Process("P4", 30)  # 进程P4，大小为30
p5 = Process("P5", 20)  # 进程P5，大小为20
```

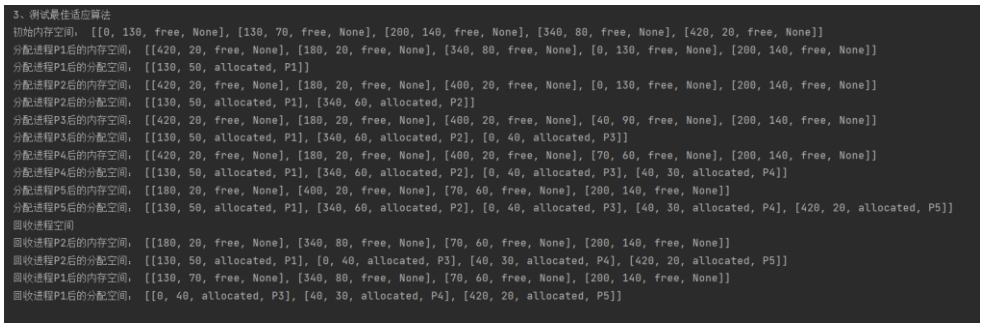
初始定义的内存块大小和进程大小

```
E:\progress\PY_progress\.condaPY\pythonProject\python.exe E:\progress\PY_progress\memory_allocation\allocation.py
1、测试首次适应算法
初始内存空间： [[0, 130, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P1后的内存空间： [[50, 80, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P1后的分配空间： [[0, 50, allocated, P1]]
分配进程P2后的内存空间： [[110, 20, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P2后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2]]
分配进程P3后的内存空间： [[110, 20, free, None], [170, 30, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P3后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3]]
分配进程P4后的内存空间： [[110, 20, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P4后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3], [170, 30, allocated, P4]]
分配进程P5后的内存空间： [[200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P5后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3], [170, 30, allocated, P4], [110, 20, allocated, P5]]
回收进程空间
回收进程P2后的内存空间： [[200, 140, free, None], [340, 80, free, None], [420, 20, free, None], [50, 60, free, None]]
回收进程P2后的分配空间： [[0, 50, allocated, P1], [130, 40, allocated, P3], [170, 30, allocated, P4], [110, 20, allocated, P5]]
回收进程P1后的内存空间： [[200, 140, free, None], [340, 80, free, None], [420, 20, free, None], [0, 110, free, None]]
回收进程P1后的分配空间： [[130, 40, allocated, P3], [170, 30, allocated, P4], [110, 20, allocated, P5]]
```

测试首次适应算法

```
2、测试循环首次适应算法
初始内存空间： [[0, 130, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P1后的内存空间： [[50, 80, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P1后的分配空间： [[0, 50, allocated, P1]]
分配进程P2后的内存空间： [[110, 20, free, None], [130, 70, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P2后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2]]
分配进程P3后的内存空间： [[110, 20, free, None], [170, 30, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P3后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3]]
分配进程P4后的内存空间： [[110, 20, free, None], [200, 140, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P4后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3], [170, 30, allocated, P4]]
分配进程P5后的内存空间： [[110, 20, free, None], [220, 120, free, None], [340, 80, free, None], [420, 20, free, None]]
分配进程P5后的分配空间： [[0, 50, allocated, P1], [50, 60, allocated, P2], [130, 40, allocated, P3], [170, 30, allocated, P4], [200, 20, allocated, P5]]
回收进程空间
回收进程P2后的内存空间： [[50, 80, free, None], [220, 120, free, None], [340, 80, free, None], [420, 20, free, None]]
回收进程P2后的分配空间： [[0, 50, allocated, P1], [130, 40, allocated, P3], [170, 30, allocated, P4], [200, 20, allocated, P5]]
回收进程P1后的内存空间： [[0, 130, free, None], [220, 120, free, None], [340, 80, free, None], [420, 20, free, None]]
回收进程P1后的分配空间： [[130, 40, allocated, P3], [170, 30, allocated, P4], [200, 20, allocated, P5]]
```

测试循环首次适应算法



测试最佳适应算法

测试最差适应算法

总共实现了 4 总内存分配算法，一种内存回收算法

三、实验环境、问题及解决方法

PyCharm 2023.1.3

1、如何用 Python 定义和操作内存分区的数据结构？你可以使用类（class）来封装分区的属性和方法，例如起始地址、长度、状态、ID 等。也可以使用列表（list）来存储空闲分区表和已分配分区表，方便进行插入和删除操作。

```
class Partition:
    def __init__(self, start, size, state, process):
        self.start = start  # 起始地址
        self.size = size  # 大小
        self.state = state  # 状态，"free"或"allocated"
        self.process = process  # 占用的进程，None表示空闲
        self.next = None
        self.prev = None

    def __str__(self):
        return f"[{self.start}, {self.size}, {self.state}, {self.process}]"


# 定义内存分区链表类
2 用法
class PartitionList:
    def __init__(self):
        self.head = None  # 头结点
        self.tail = None  # 尾结点
        self.length = 0  # 长度

    15 个用法 (9 个动态)
```

2、如何用 Python 实现不同的动态分区分配算法？可以根据算法的逻辑，遍历空闲分区表，找到合适的分区进行分配，如果分区大小大于作业需要的大小，

就要进行分割，如果分区大小等于作业需要的大小，就直接分配。

```python
# 定义内存分区链表类
2 用法
class PartitionList:
    def __init__(self):
        self.head = None  # 头结点
        self.tail = None  # 尾结点
        self.length = 0  # 长度

    15 个用法 (9 个动态)
    def append(self, partition):
        # 在链表尾部添加一个空闲分区
        if self.head is None:
            self.head = partition
            self.tail = partition
        else:
            self.tail.next = partition
            partition.prev = self.tail
            self.tail = partition
        self.length += 1

    7 个用法 (6 个动态)
    def remove(self, partition):
        # 从链表中删除一个空闲分区
        if self.head is None:
            raise ValueError("List is emp
```

3、如何用 Python 实现动态分区的回收和合并？可以根据作业的 ID，找到对应的分区，将其状态改为未分配，然后检查其前后是否有相邻的空闲分区，如果有，就进行合并，更新分区的起始地址、长度和 ID。

```python
# 定义内存回收算法
8 用法
def memory_recycle(memory, Allocated_Memory, process):
    # 参数：memory是一个空闲分区链表，process是一个进程对象
    # 返回值：如果回收成功，返回一个回收后的空闲分区链表，否则返回None
    current = Allocated_Memory.head  # 定义一个当前结点，从头结点开始查找
    while current is not None:
        if current.process == process:  # 如果当前分区的进程是要回收的进程
            temp = memory.head
            index = current.start + current.size
            while temp is not None:
                if temp.start == index:  # 找到与释放内存相邻的内存块
                    temp.start = current.start
                    temp.size = current.size + temp.size
                    Allocated_Memory.remove(current)
                    return memory
                temp = temp.next
            temp2 = Partition(current.start, current.size, 'free', None)  # 没找到相邻内存块就直接加入
            memory.append(temp2)
            Allocated_Memory.remove(current)
            return memory
        current = current.next  # 继续查找下一个分区
    return None  # 如果没有找到要回收的进程，回收失败
```

四、请了解目前国产操作系统的发展和应用情况，你觉得我们应该如何努力？

我们应该学好计算机专业的基础课程，如计算机组成原理、计算机操作系统、计算机网络、数据结构与算法、编译原理等，这些课程是计算机科学的核心知识，也是操作系统的基础。同时应该学好英语，因为很多新技术和资料都是英文的，英语能力可以帮助你获取更多的信息，也可以让你更好地参与到国际的交流和合作中。应该精通至少一门编程语言，如 C、Java、Python 等，编程语言是操作系统的实现工具，是展示自己技能的方式，可以通过编程语言来实现一些操作系统的功能，或者参与到一些开源操作系统的项目中。

最后，应该多阅读和实践，阅读一些经典的操作系统书籍和文章，了解操作系统的原理和设计，实践一些操作系统的实验和项目，提高自己的操作系统的理解和应用能力。

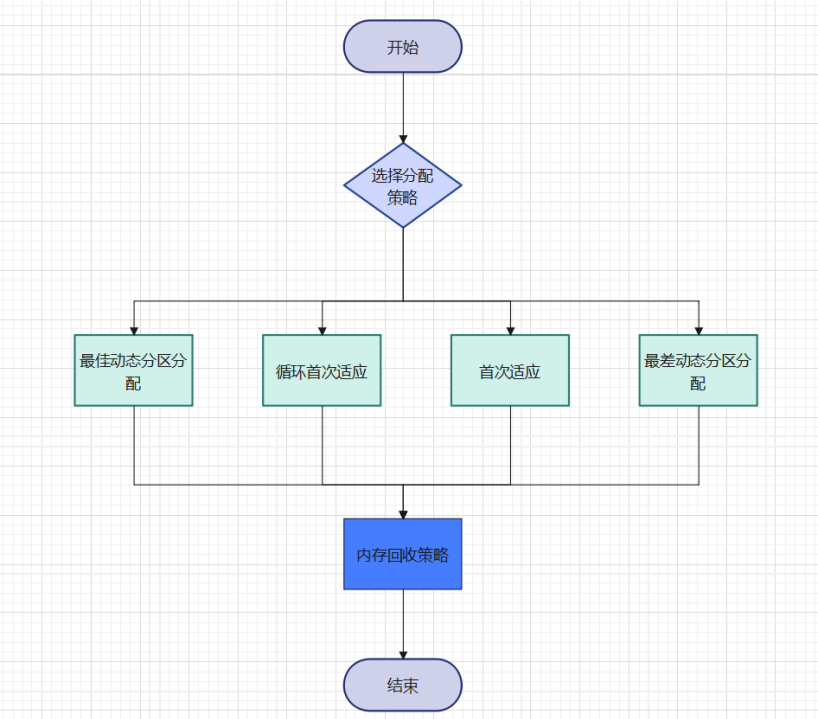
五、实验收获

1、通过编程实现不同的动态分区分配算法，加深了对操作系统内存管理的理解和掌握。

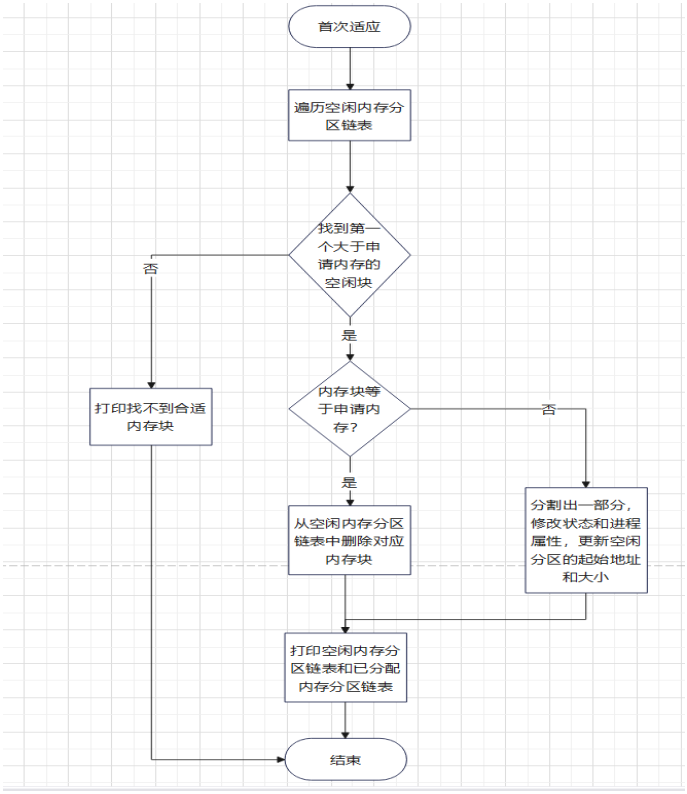
2、了解了动态分区分配算法的优缺点和适用场景，比如首次适应算法简单易实现，但容易产生外部碎片；最佳适应算法能够有效利用小空闲分区，但会增加查找时间和内部碎片；最差适应算法能够保留大空闲分区，但会造成大量的小碎片。

3、学习了如何用 Python 设计和操作数据结构，提高了编程能力和技巧。

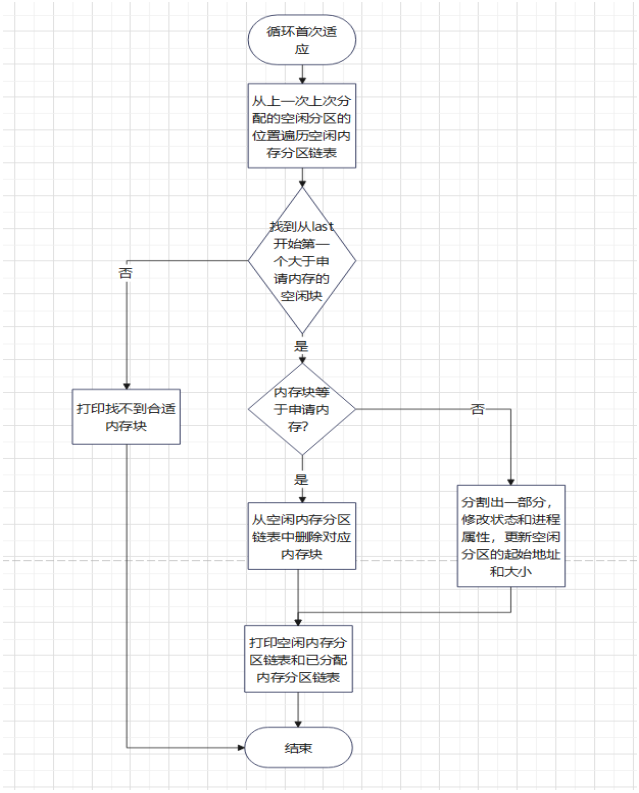
六、程序设计



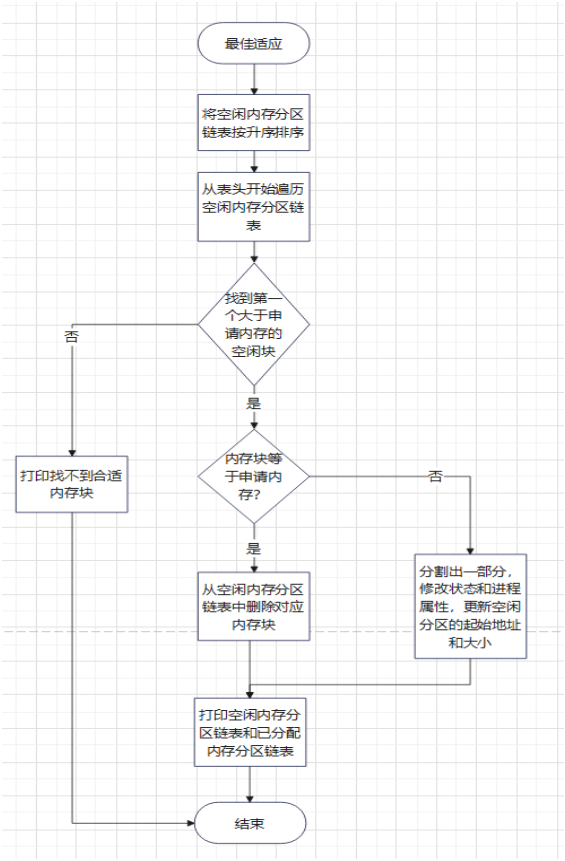
程序主体框架



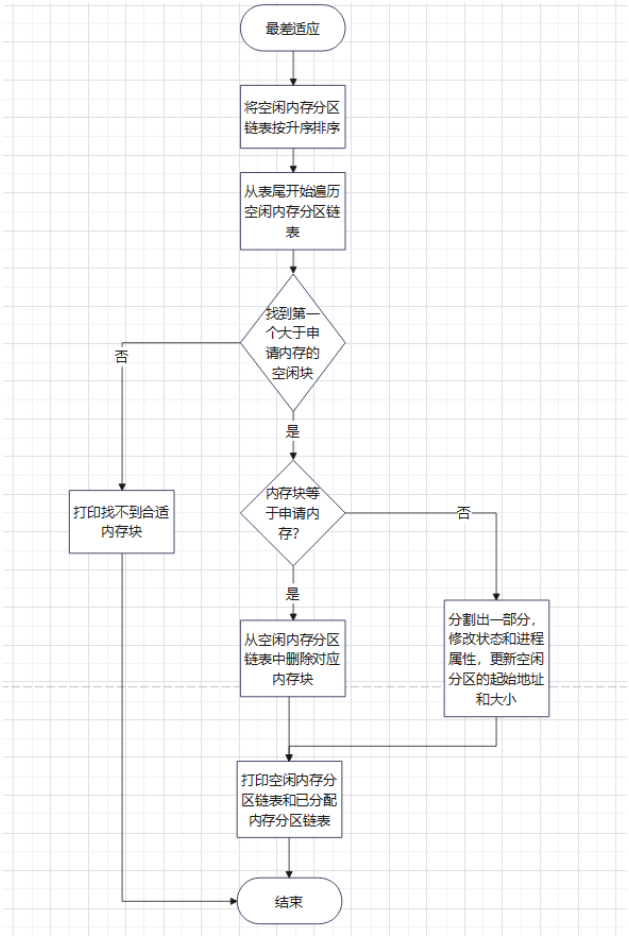
首次适应算法



循环首次适应算法


最佳适应
将空闲内存分区链表按升序排序
从表头开始遍历空闲内存分区链表
找到第一个大于申请内存的空闲块
否
是
打印找不到合适内存块
内存块等于申请内存?
否
是
从空闲内存分区链表中删除对应内存块
分割出一部分，修改状态和进程属性，更新空闲分区的起始地址和大小
打印空闲内存分区链表和已分配内存分区链表
结束

最佳适应算法



最差适应算法

七、参考资料

无