目录

[一、实验内容： 2](#_Toc167652225)

[二、实验目的： 3](#_Toc167652226)

[三、实验要求： 3](#_Toc167652227)

[四、实现方法 4](#_Toc167652228)

[1-基本思想： 4](#_Toc167652229)

[2-整体程序运行逻辑 5](#_Toc167652230)

[1-主函数 (main)： 5](#_Toc167652231)

[2-First\_fit 函数： 5](#_Toc167652232)

[3- Best\_fit函数： 6](#_Toc167652233)

[4- Worst-fit函数： 7](#_Toc167652234)

[3-相关数据结构 8](#_Toc167652235)

[1-已分配区表（used\_table） 8](#_Toc167652236)

[2. 空闲区表（free\_table） 8](#_Toc167652237)

[3. 作业队列（job） 8](#_Toc167652238)

[4-相关功能实现 9](#_Toc167652239)

[4.4.1初始化： 9](#_Toc167652240)

[4.4.2相关队列的展示 10](#_Toc167652241)

[4.4.3用户自定义作业数量以及内存分配方式 10](#_Toc167652242)

[4.4.4最先适应（First-fit）算法 11](#_Toc167652243)

[4.4.5最优适应（Best-fit）算法 12](#_Toc167652244)

[4.4.6最坏适应（Worst-fit）算法 14](#_Toc167652245)

[4.4.7内存回收： 16](#_Toc167652246)

[五、实验结果： 18](#_Toc167652247)

[5.1要求用户指定作业数量以及内存分配算法： 18](#_Toc167652248)

[5.2最先适应算法展示： 18](#_Toc167652249)

[5.2.1初始化相关变量和信息： 18](#_Toc167652250)

[5.2.2第一轮相关展示： 19](#_Toc167652251)

[5.2.3第二轮相关展示 20](#_Toc167652252)

[5.2.4其余三轮结果如下： 22](#_Toc167652253)

[5.2.5相关回收过程： 22](#_Toc167652254)

[5.3最优适应（Best-fit）算法展示： 25](#_Toc167652255)

[5.3.1初始化相关信息和变量 25](#_Toc167652256)

[5.3.2第一轮相关展示 26](#_Toc167652257)

[5.3.3后续轮相关展示： 27](#_Toc167652258)

[5.3.4相关回收过程： 29](#_Toc167652259)

[5.4最坏适应（Worst-fit）算法展示： 30](#_Toc167652260)

[5.4.1初始化相关信息和变量： 30](#_Toc167652261)

[5.4.2第一轮分配： 31](#_Toc167652262)

[5.4.3第一轮回收： 33](#_Toc167652263)

[5.4.4进行第二轮分配： 34](#_Toc167652264)

[5.4.5进行第二轮的回收 35](#_Toc167652265)

[5.4.6其余轮中一些重要的过程： 36](#_Toc167652266)

[5.4.7最终结果： 37](#_Toc167652267)

[六、总结： 37](#_Toc167652268)

[1-本实验中使用了三种算法： 37](#_Toc167652269)

[2-学习和收获： 37](#_Toc167652270)

# 一、实验内容：

可变分区调度算法有：最先适应分配算法，最优适应分配算法，最坏适应算法。用户提出内存空间的申请；系统根据申请者的要求，按照一定的分配策略分析内存空间的使用情况，找出能满足请求的空闲区，分给申请者；当程序执行完毕或主动归还内存资源时，系统要收回它所占用的内存空间或它归还的部分内存空间。

根据可变分区的主存分配思想，首先设计记录主存使用情况的数据表格，用来记录空闲区和作业占用的区域，即“已分配区表”和“空闲区表”。然后在数据表格上进行主存的分配，其主存分配算法选择一种算法。程序中可选择进行主存分配或主存回收，若是主存分配，输入作业名和所需主存空间大小；若是回收，输入回收作业的作业名，以循环进行主存分配和回收。

当要装入一个作业时，从空闲区表中查找标志为“未分配”的空闲区，从中找一个能容纳该作业的空闲区。如果找到的空闲区正好等于该作业的长度，则把该分区全部分配给该作业。这时应该把该空闲区登记栏中的标志改为“空”，同时在已分配区表中找到一个标志为“空”的栏目登记新装入作业所占用分区的起始地址、长度和作业名。如果找到的空闲区大于作业长度，则把空闲区分成两部分，一部分用来装入作业，另一部分仍然为空闲区，这时只要修改空闲区的长度，且把新装入的作业登记到已分配区表中。本实验中采用首次适应算法、循环首次适应算法、最坏适应算法中的一种为作业分配主存。

动态分区方式下回收主存空间时应该检查是否有与归还区相邻的空闲区域。若有，则应该合并成一个空闲区。一个归还区可能有上邻空闲区，也可能有下邻空闲区，或者既有上邻空闲区又有下邻空闲区，或者既无上邻空闲区也无下邻空闲区。

在实现回收时，首先将作业归还的区域在已分配表中找到，将该栏目的状态变为“空”，然后检查空闲区表中标志为“未分配”栏目，查找是否又相邻空闲区；最后合并空闲区，修改空闲区表。

# 二、实验目的：

1．了解可变分区管理方式下如何进行主存的分配；

2．了解可变分区管理方式下如何进行主存空间的回收。

3．了解可变分区的调度算法分配策略。

# 三、实验要求：

由于动态分区的大小是由作业需求量决定的，故分区的长度是预先不固定的，且分区的个数也随主存分配和回收变动。总之，所有分区情况随时可能发生变化，数据表格的设计必须和这个特点相适应。由于分区长度不同，因此设计的表格应该包括分区在主存中的起始地址和长度。由于分配时，空闲区有时会变成两个分区：空闲区和已分分区，回收主存分区时，可能会合并空闲区，这样如果整个主存采用一张表格记录已分分区和空闲区，就会使表格操作繁琐。主存分配时查找空闲区进行分配，然后填写已分配区表，主要操作在空闲区；某个作业执行完后，将该分区变成空闲区，并将其与相邻的空闲区合并，主要操作也在空闲区。由此可见，主存的分配与回收主要时对空闲区的操作。这样为了便于对主存空间的分配与回收，就建立两张分区表记录主存的使用情况：“已分配区表”记录作业占用分区，“空闲区表”记录空闲区。

这两张表的实现方法一般由两种：链表形式、顺序表形式。在本实验中，可以采用任意一种形式。如果采用顺序表形式，用数组模拟。由于顺序表的长度必须提前固定，所以无论是“已分配区表”还是“空闲区表”都必须事先确定长度。它们的长度必须是系统可能的最大项数，系统运行过程中才不会出错，因此在多数情况下，无论是“已分配表区”还是“空闲区表”都是空闲栏目。已分配区表中除了分区起始地址、长度外，也至少还有一项“标志”，如果是空闲栏目，内容为“空”，如果为某个作业占用分区的登记项，内容为该作业的作业名；空闲区表除了分区起始地址、长度外，也要有一项“标志”，如果是空闲栏目，内容为“空”，如果为某个空闲区的登记项，内容为“未分配”。在实际系统中，这两个表格的内容可能还要多，实验中仅仅使用上述必须的数据。为此，“已分配区表”和“空闲区表”在实验中有如下的结构定义。

已分配区表的定义：

#define n 10               //假定系统允许的最大作业数量为n

struct{

 float address;       //已分分区起始地址

 float length;        //已分分区长度，单位为字节

 int flag;            //已分配表区登记栏标志，用0表示空栏目，

}used\_table[n];          //已分配区表

空闲区表的定义：

#define m 10               //假定系统允许的空闲区表最大为m

struct{

 float address;       //空闲区起始地址

 float length;        //空闲区长度，单位为字节

 int flag;            //空闲区表登记栏目用0表示空栏目，1表示未分配

}free\_table[m];          //空闲区表

# 四、实现方法

## 1-基本思想：

* **可变分区分配的基本思想：**

依据作业即时需求动态分配内存，不预设固定区间。若存储空间不足，作业便进入等待状态，直到有足够内存释放。该策略提升了内存使用效率，但需妥善管理以避免外部碎片问题。

* **分配算法：**

**首次适应算法：**最先适应算法从空闲区表的头开始，依次查找能够容纳待分配作业的首个空闲分区。找到后，如果该空闲分区的大小正好等于作业大小，则直接分配；如果大于作业大小，则将空闲分区拆分成两部分，一部分分配给作业，另一部分仍作为空闲区。

**最优适应（Best-fit）算法：**最优适应算法在空闲区表中查找最小的、但能够容纳待分配作业的空闲分区。这样做的目的是尽量减少内存浪费，保留较大的空闲分区供后续使用。找到后，如果该空闲分区的大小正好等于作业大小，则直接分配；如果大于作业大小，则将空闲分区拆分成两部分。

**最坏适应（****Worst-fit）算法：**最坏适应算法在空闲区表中查找最大的空闲分区，以便使得分配后仍能保留较大的空闲分区。找到后，如果该空闲分区的大小正好等于作业大小，则直接分配；如果大于作业大小，则将空闲分区拆分成两部分。

* **回收分区的基本思想：**

当作业运行结束释放内存后，系统将根据该内存块的起始地址，在空闲分区链表中找到恰当的插入点以回收该内存空间。这一过程可能遇到以下四种情况：

1. 如果释放内存与插入点前一个空闲分区毗邻，它们将被合并为一个分区。

2. 如果释放内存与插入点后一个空闲分区毗邻，这两个分区也将合并。

3. 如果释放内存同时与插入点前后的空闲分区相邻，则这三者合并为单一大分区。

4. 如果释放内存与任何相邻分区都不接壤，它将形成一个新的独立空闲分区。

通过这一内存回收机制，操作系统能够有效地实现空闲内存的合理化管理，减少内存碎片问题，提高内存利用效率。

## 2-整体程序运行逻辑

### 1-主函数 (main)：

* + 初始化随机数种子。
  + 提示用户输入需要分配内存的作业数量，并读取用户输入。
  + 验证输入的作业数量是否在有效范围（1-10）内，如果无效则退出程序。
  + 调用 **First\_fit** 函数执行内存分配和回收。

### 2-First\_fit 函数：

1. **初始化**：
   * 调用 **Init\_table()** 初始化已分配区表和空闲区表。
   * 调用 **Init\_job(job\_count)** 初始化作业队列。
   * 调用 **Display()** 输出已分配区表和空闲区表。
   * 调用 **Display\_job()** 输出作业队列。
   * 初始化变量 **num** 为 **job\_count**，表示未分配作业的数量。
2. **内存分配和回收循环**：
   * 当 **num** 大于 0 时，执行分配和回收循环。

**分配阶段**：

* + 输出 "开始一轮分配"。
  + 对每一个未分配作业进行分配：
    - 查找 **job** 队列中 **flag** 为 0（未分配）的作业。
    - 输出找到的作业及其大小。
    - 调用 **Distribute(job[i].name, job[i].length, 0)** 进行分配。
    - 如果分配成功：
      * 输出 "成功分配"。
      * 将 **job[i].flag** 置为 1（已分配）。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。
    - 否则：
      * 输出 "该作业暂无空间分配"。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。

**回收阶段**：

* + 如果所有作业都已分配（即 **num == 0**），输出 "全部作业都分配了空间，程序结束" 并结束循环。
  + 否则：
    - 输出 "开始一轮回收"。
    - 对每一个已分配作业进行随机回收：
      * 查找 **job** 队列中 **flag** 为 1（已分配）的作业。
      * 生成随机数 **choice**（0 或 1）。
      * 输出随机数 **choice**。
      * 如果 **choice** 为 1：
        + 输出 "回收作业" 及其名称。
        + 调用 **Reclaim(job[i].name)** 进行回收。
        + 将 **job[i].flag** 置为 2（已完成）。
        + 输出 "作业已经完成"。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。
      * 否则：
        + 输出 "暂不回收作业" 及其名称。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。

**更新未分配作业计数**：

* + 遍历 **job** 队列，更新未分配作业的数量 **num**。

### 3- Best\_fit函数：

1. **初始化**：
   * 调用 **Init\_table()** 初始化已分配区表和空闲区表。
   * 调用 **Init\_job(job\_count)** 初始化作业队列。
   * 调用 **Display()** 输出已分配区表和空闲区表。
   * 调用 **Display\_job()** 输出作业队列。
   * 初始化变量 **num** 为 **job\_count**，表示未分配作业的数量。
2. **内存分配和回收循环**：
   * 当 **num** 大于 0 时，执行分配和回收循环。

**分配阶段**：

* + 输出 "开始一轮分配"。
  + 按空闲区长度升序排序 **free\_table**。
  + 对每一个未分配作业进行分配：
    - 查找 **job** 队列中 **flag** 为 0（未分配）的作业。
    - 输出找到的作业及其大小。
    - 调用 **Distribute(job[i].name, job[i].length, 1)** 进行分配。
    - 如果分配成功：
      * 输出 "成功分配"。
      * 将 **job[i].flag** 置为 1（已分配）。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。
    - 否则：
      * 输出 "该作业暂无空间分配"。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。

**回收阶段**：

* + 如果所有作业都已分配（即 **num == 0**），输出 "全部作业都分配了空间，程序结束" 并结束循环。
  + 否则：
    - 输出 "开始一轮回收"。
    - 对每一个已分配作业进行随机回收：
      * 查找 **job** 队列中 **flag** 为 1（已分配）的作业。
      * 生成随机数 **choice**（0 或 1）。
      * 输出随机数 **choice**。
      * 如果 **choice** 为 1：
        + 输出 "回收作业" 及其名称。
        + 调用 **Reclaim(job[i].name)** 进行回收。
        + 将 **job[i].flag** 置为 2（已完成）。
        + 输出 "作业已经完成"。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。
      * 否则：
        + 输出 "暂不回收作业" 及其名称。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。

**更新未分配作业计数**：

* + 遍历 **job** 队列，更新未分配作业的数量 **num**。

### 4- Worst-fit函数：

**1.初始化**：

* + 调用 **Init\_table()** 初始化已分配区表和空闲区表。
  + 调用 **Init\_job(job\_count)** 初始化作业队列。
  + 调用 **Display()** 输出已分配区表和空闲区表。
  + 调用 **Display\_job()** 输出作业队列。
  + 初始化变量 **num** 为 **job\_count**，表示未分配作业的数量。

**2.内存分配和回收循环**：

* + 当 **num** 大于 0 时，执行分配和回收循环。

**分配阶段**：

* + 输出 "开始一轮分配"。
  + 按空闲区长度降序排序 **free\_table**。
  + 对每一个未分配作业进行分配：
    - 查找 **job** 队列中 **flag** 为 0（未分配）的作业。
    - 输出找到的作业及其大小。
    - 调用 **Distribute(job[i].name, job[i].length, 2)** 进行分配。
    - 如果分配成功：
      * 输出 "成功分配"。
      * 将 **job[i].flag** 置为 1（已分配）。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。
    - 否则：
      * 输出 "该作业暂无空间分配"。
      * 调用 **Display()** 输出已分配区表和空闲区表。
      * 调用 **Display\_job()** 输出作业队列。

**回收阶段**：

* + 如果所有作业都已分配（即 **num == 0**），输出 "全部作业都分配了空间，程序结束" 并结束循环。
  + 否则：
    - 输出 "开始一轮回收"。
    - 对每一个已分配作业进行随机回收：
      * 查找 **job** 队列中 **flag** 为 1（已分配）的作业。
      * 生成随机数 **choice**（0 或 1）。
      * 输出随机数 **choice**。
      * 如果 **choice** 为 1：
        + 输出 "回收作业" 及其名称。
        + 调用 **Reclaim(job[i].name)** 进行回收。
        + 将 **job[i].flag** 置为 2（已完成）。
        + 输出 "作业已经完成"。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。
      * 否则：
        + 输出 "暂不回收作业" 及其名称。
        + 调用 **Display()** 输出已分配区表和空闲区表。
        + 调用 **Display\_job()** 输出作业队列。

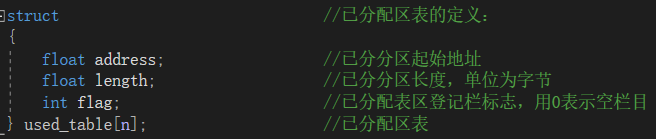
**更新未分配作业计数**：

* + 遍历 **job** 队列，更新未分配作业的数量 **num**。

## 3-相关数据结构

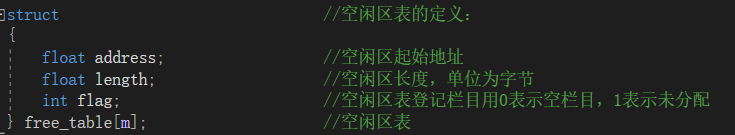
### 1-已分配区表（used\_table）

用于记录系统中已经分配出去的内存区块的信息，包括起始地址、长度和标志（作业名）。



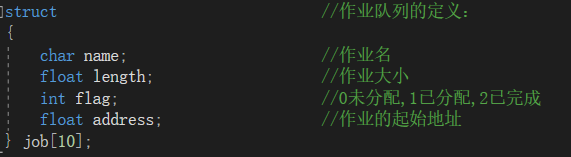
### 空闲区表（free\_table）

用于记录系统中尚未分配的空闲内存区块的信息，包括起始地址、长度和标志（是否空闲）。



### 作业队列（job）

用于记录需要分配内存的作业的信息，包括作业名、大小、状态和起始地址。



## 4-相关功能实现

### 4.4.1初始化：

* **定义全局变量：**

总内存大小：int L = 1024;

用户输入的作业数量。int job\_count;

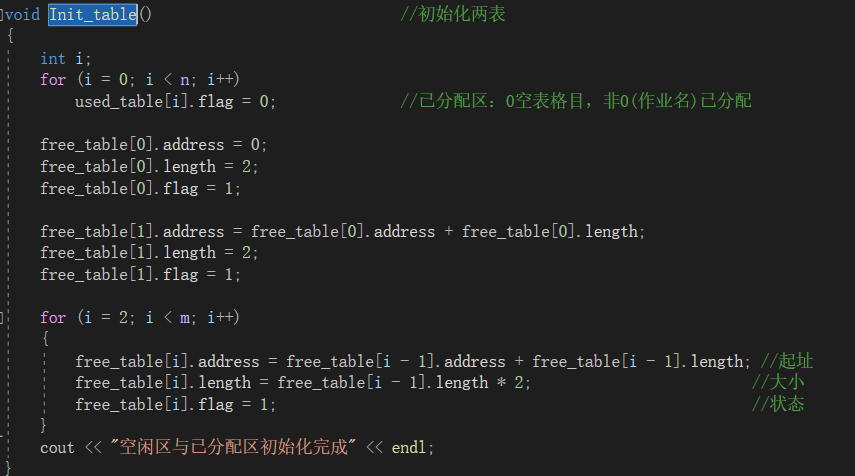


* **Init\_table：**

初始化 free\_table 和 used\_table，设定初始的空闲区和已分配区状态。

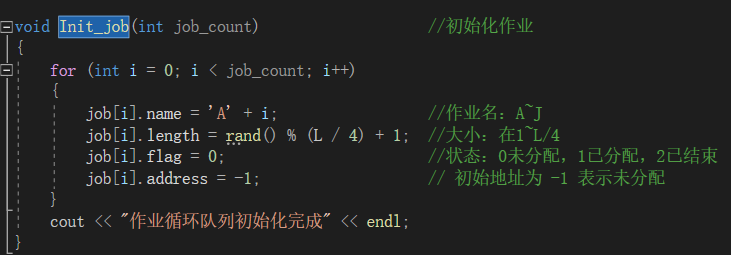
used\_table 初始化：将 used\_table 的 flag 设为0，表示所有栏目为空。

free\_table 初始化：按照地址递增的次序初始化空闲区。第一个和第二个空闲区大小为2，从第三个空闲区开始大小为前一个的两倍。



* **Init\_job：**

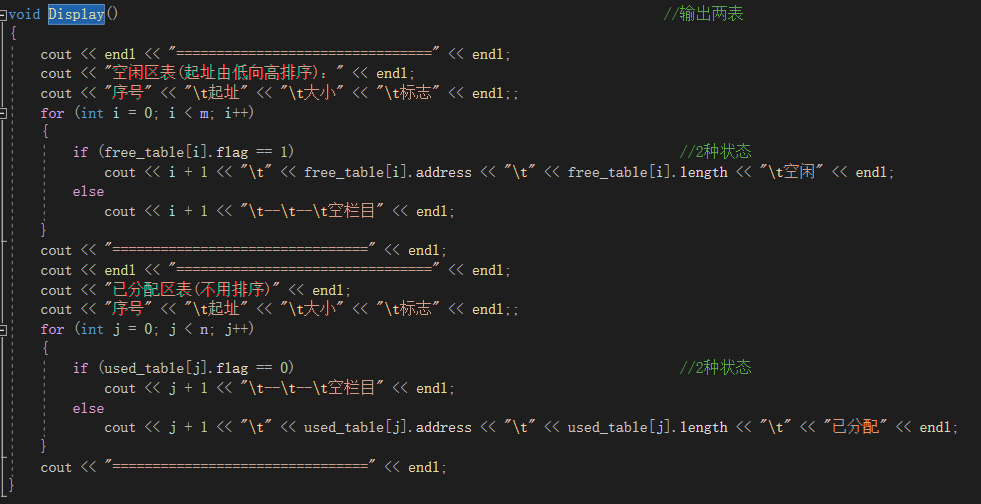
初始化作业队列的每个作业，设置作业名（A-J），随机生成作业大小（1到总内存的1/4），状态设为未分配（flag 为0），起始地址设为-1 表示未分配。



### 4.4.2相关队列的展示

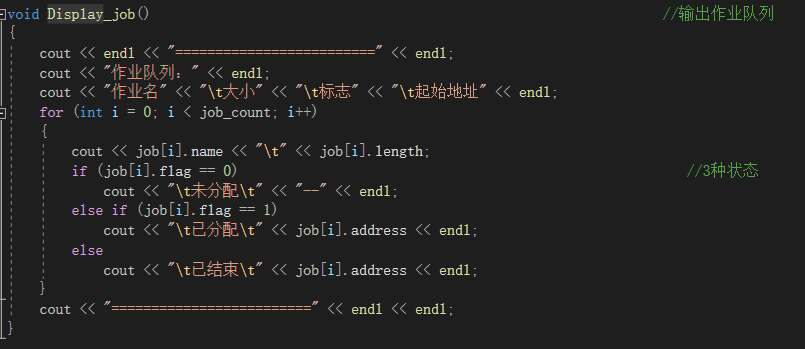
* **Display函数：**

输出空闲区表和已分配区表的当前状态，包括序号、起始地址、大小和标志（空闲或已分配）。



* **Display\_job函数：**

输出作业队列的当前状态，包括作业名、大小、状态（未分配、已分配、已结束）和起始地址。



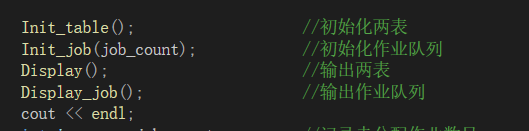
### 4.4.3用户自定义作业数量以及内存分配方式

要求用户输入需要分配内存的作业的数量以及要求用户指定内存分配的方式.

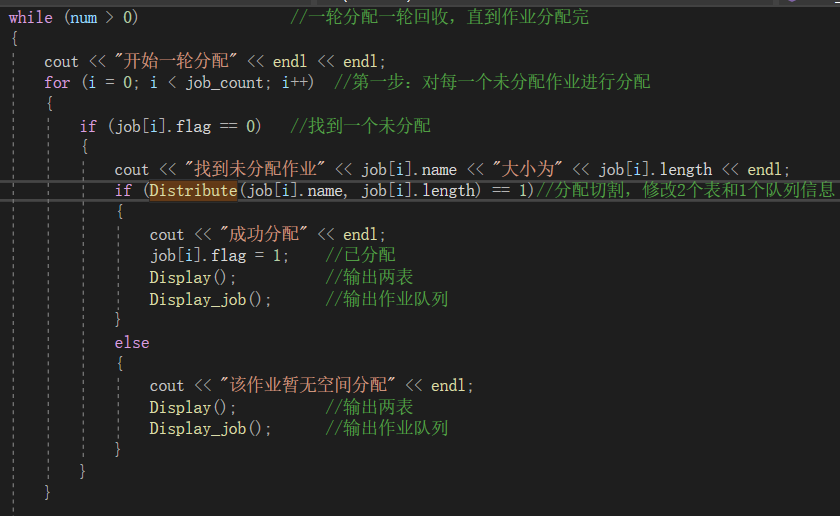


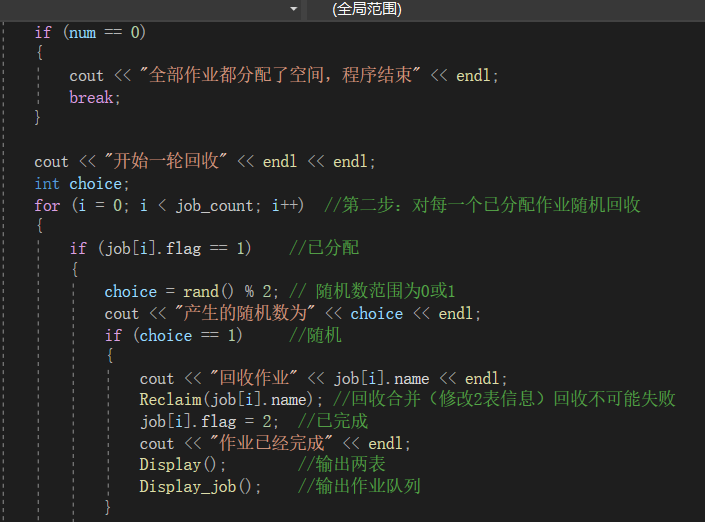
### 4.4.4最先适应（First-fit）算法

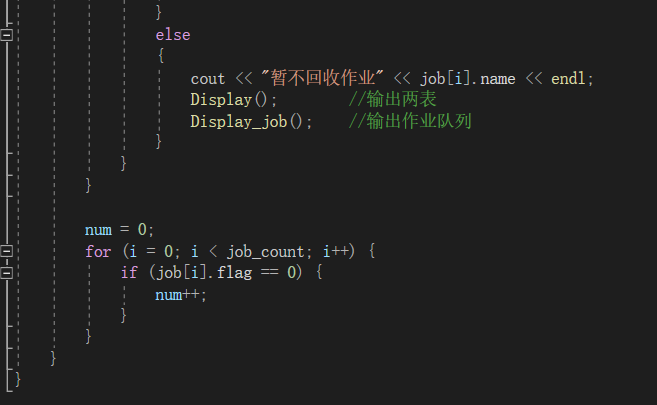
1. **初始化两表**：调用 **Init\_table()** 函数初始化已分配区表和空闲区表。
2. **初始化作业队列**：调用 **Init\_job(job\_count)** 函数初始化作业队列。
3. **显示初始状态**：调用 **Display()** 和 **Display\_job()** 函数输出初始的空闲区表和已分配区表以及作业队列。



1. **循环分配和回收**：
   * 分配作业：遍历作业队列，对每一个未分配的作业，调用 **Distribute()** 函数进行内存分配。
     + 如果分配成功，更新作业状态为已分配，并再次显示空闲区表和已分配区表。
     + 如果分配失败，显示当前的空闲区表和已分配区表。
   * 回收作业：遍历作业队列，对每一个已分配的作业，随机决定是否回收。
     + 如果决定回收，调用 **Reclaim()** 函数回收内存，更新作业状态为已完成，并再次显示空闲区表和已分配区表。
     + 如果不回收，显示当前的空闲区表和已分配区表。
   * 更新未分配作业的数量，继续下一轮分配和回收，直到所有作业分配完毕。

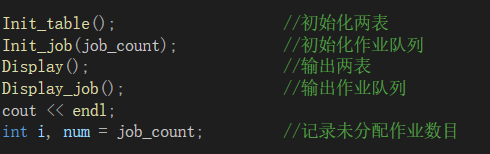




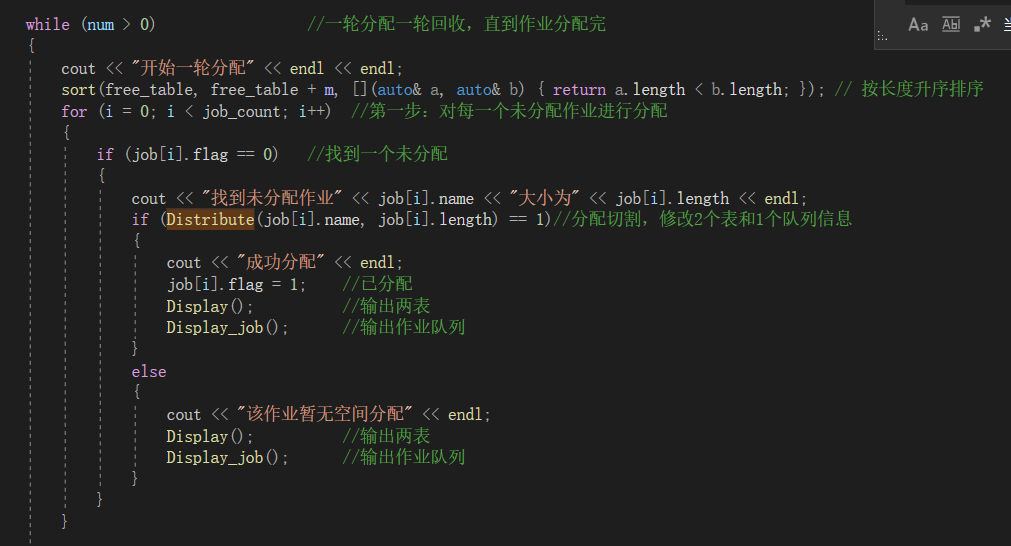


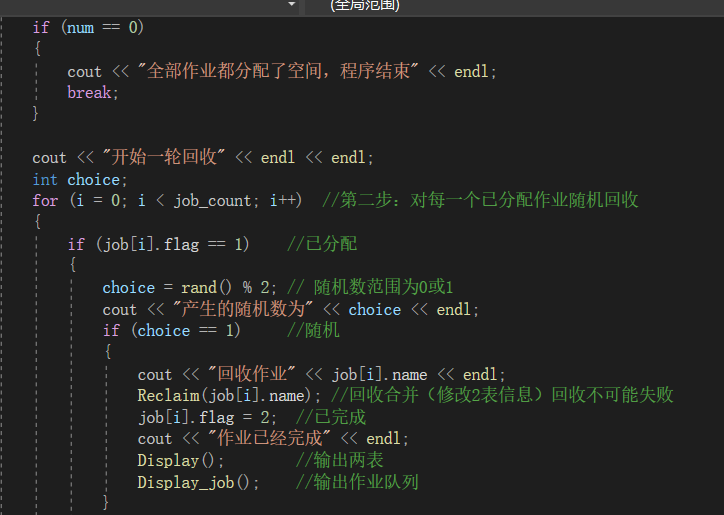
### 4.4.5最优适应（Best-fit）算法

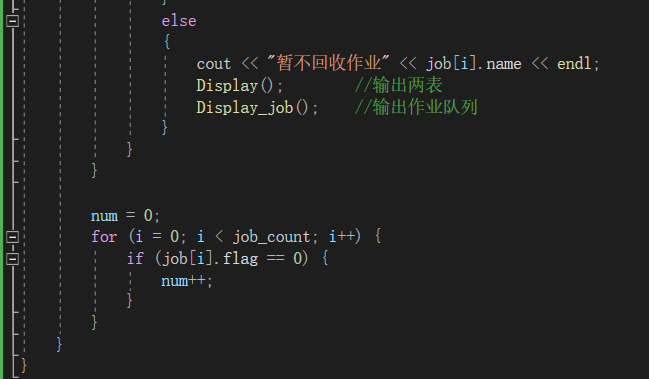
1. **初始化两表**：调用 **Init\_table()** 函数初始化已分配区表和空闲区表。
2. **初始化作业队列**：调用 **Init\_job(job\_count)** 函数初始化作业队列。
3. **显示初始状态**：调用 **Display()** 和 **Display\_job()** 函数输出初始的空闲区表和已分配区表以及作业队列。



1. **循环分配和回收**：
   * 分配作业：
     + 对空闲区表按照空闲区大小进行升序排序。
     + 遍历排序后的空闲区表，对每一个未分配的作业，调用 **Distribute()** 函数进行内存分配。
     + 如果分配成功，更新作业状态为已分配，并再次显示空闲区表和已分配区表。
     + 如果分配失败，显示当前的空闲区表和已分配区表。
   * 回收作业：遍历作业队列，对每一个已分配的作业，随机决定是否回收。
     + 如果决定回收，调用 **Reclaim()** 函数回收内存，更新作业状态为已完成，并再次显示空闲区表和已分配区表。
     + 如果不回收，显示当前的空闲区表和已分配区表。
   * 更新未分配作业的数量，继续下一轮分配和回收，直到所有作业分配完毕。

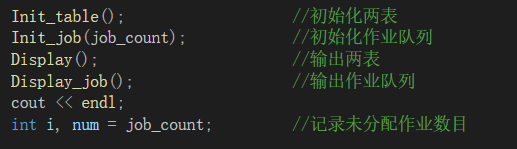




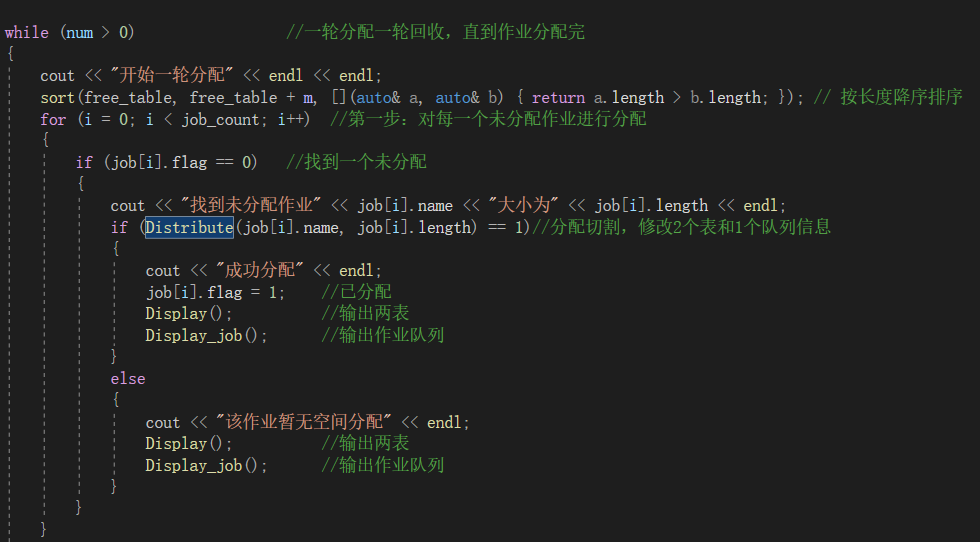


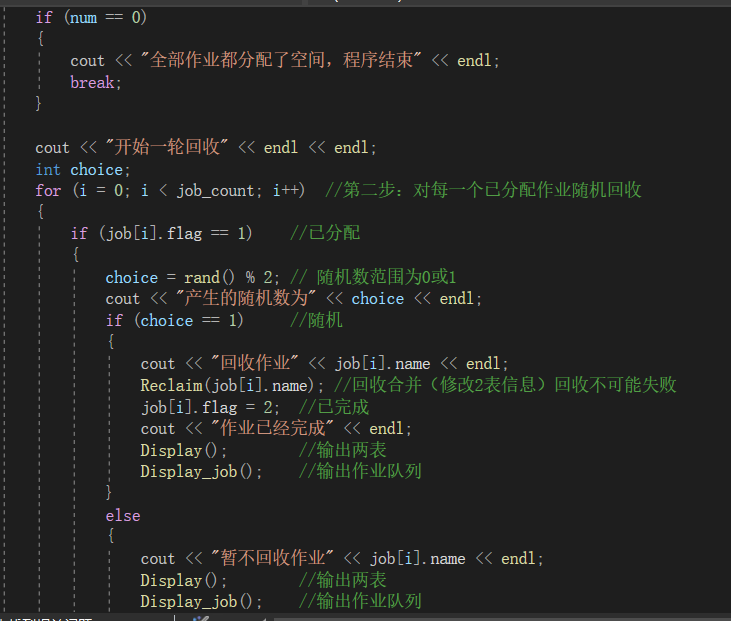
### 4.4.6最坏适应（Worst-fit）算法

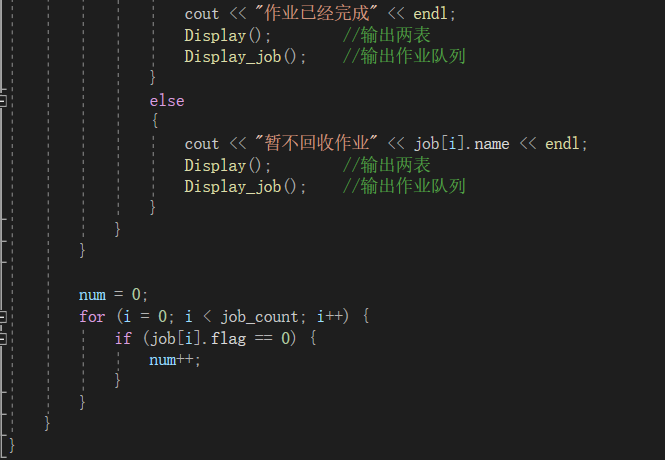
1. **初始化两表**：调用 **Init\_table()** 函数初始化已分配区表和空闲区表。
2. **初始化作业队列**：调用 **Init\_job(job\_count)** 函数初始化作业队列。
3. **显示初始状态**：调用 **Display()** 和 **Display\_job()** 函数输出初始的空闲区表和已分配区表以及作业队列。



1. **循环分配和回收**：
   * 分配作业：
     + 对空闲区表按照空闲区大小进行降序排序。
     + 遍历排序后的空闲区表，对每一个未分配的作业，调用 **Distribute()** 函数进行内存分配。
     + 如果分配成功，更新作业状态为已分配，并再次显示空闲区表和已分配区表。
     + 如果分配失败，显示当前的空闲区表和已分配区表。
   * 回收作业：遍历作业队列，对每一个已分配的作业，随机决定是否回收。
     + 如果决定回收，调用 **Reclaim()** 函数回收内存，更新作业状态为已完成，并再次显示空闲区表和已分配区表。
     + 如果不回收，显示当前的空闲区表和已分配区表。
   * 更新未分配作业的数量，继续下一轮分配和回收，直到所有作业分配完毕。







### 4.4.7内存回收：

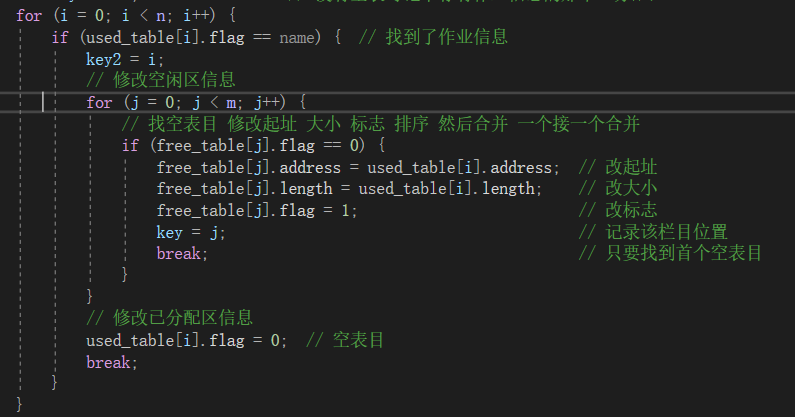
这部分功能的实现主要是通过**Reclaim函数**实现。

实现具体步骤如下：

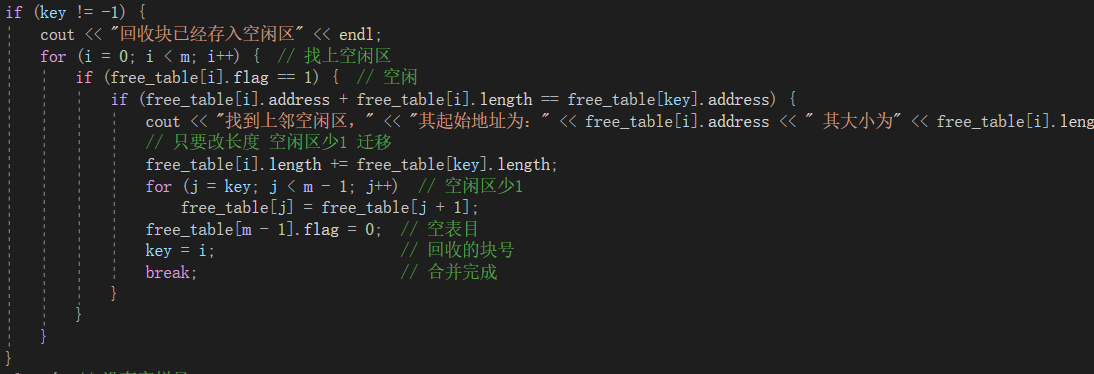
1. **初始化变量：**
   * 初始化 key 为 -1，用于记录空闲区表中空表目的位置。
   * 初始化 key2 为 -1，用于记录存有作业信息的那个已分配区表的位置。



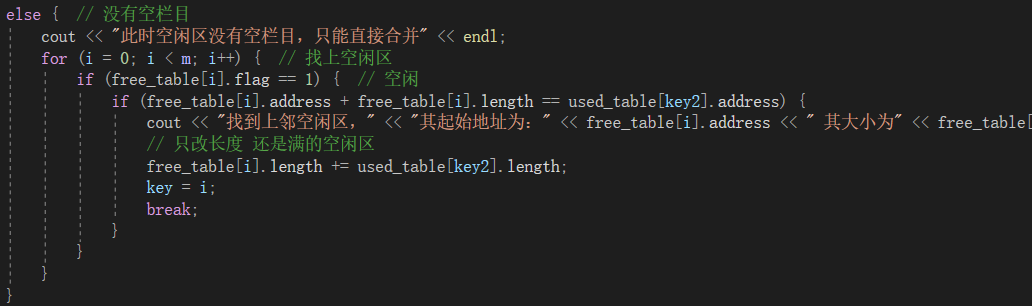
1. **查找已分配的作业信息并修改空闲区信息：**
   * 遍历 **used\_table**，查找 **flag** 为 **name** 的作业信息，记录位置 **key2**。
   * 遍历 **free\_table**，找到 **flag** 为 0 的空表目，更新该空闲区的 **address**、**length** 和 **flag**，记录位置 **key**，并将已分配区表中的对应项标记为空表目。



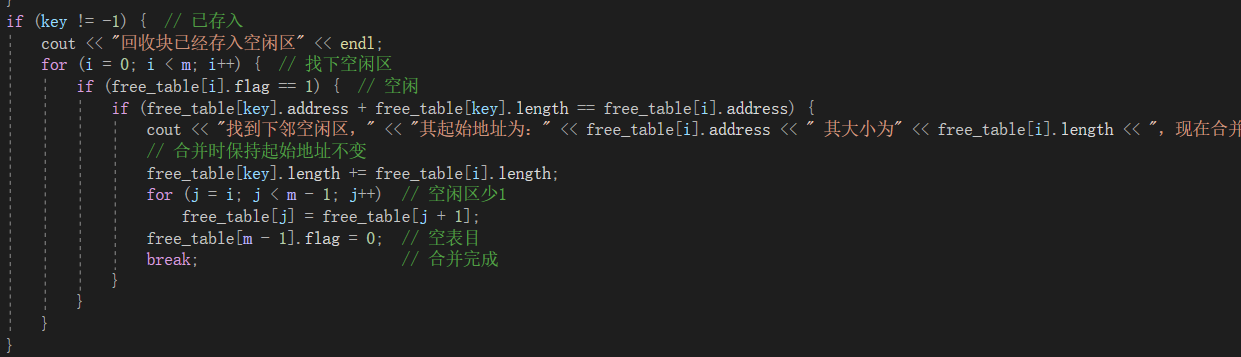
1. **合并上邻空闲区**：
   * 如果 **key** 不为 -1，表示找到了空表目，将回收块存入空闲区。
   * 遍历 **free\_table** 查找上邻空闲区，如果找到上邻空闲区，合并并更新长度，删除多余的空闲区表项（移位），并更新 **key**。



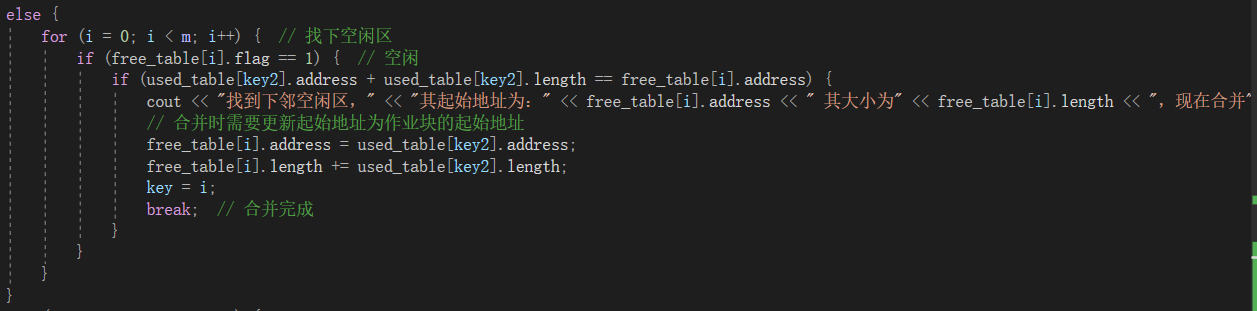
1. **没有空表目直接合并上邻空闲区**：
   * 如果 **key** 为 -1，表示没有空表目，直接合并。
   * 遍历 **free\_table** 查找上邻空闲区，如果找到上邻空闲区，合并并更新长度，并记录合并后的空闲区位置 **key**。



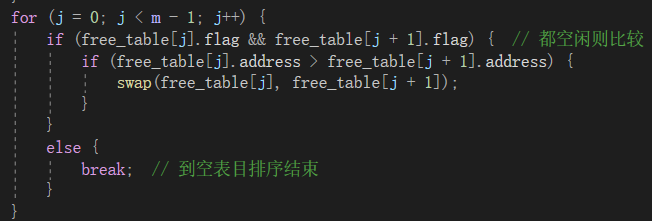
1. **合并下邻空闲区**：
   * 如果 **key** 不为 -1，表示已存入空闲区。
   * 遍历 **free\_table** 查找下邻空闲区，如果找到下邻空闲区，合并并保持起始地址不变，更新长度，删除多余的空闲区表项（移位）。



1. **没有空表目直接合并下邻空闲区**：
   * 如果 **key** 为 -1，表示没有空表目，直接合并。
   * 遍历 **free\_table** 查找下邻空闲区，如果找到下邻空闲区，合并并更新起始地址为作业块的起始地址，更新长度。

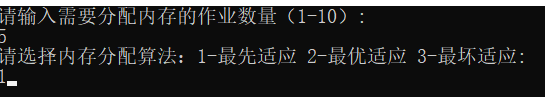


1. **排序空闲区表**：
   * 遍历 **free\_table**，对空闲区按地址排序，如果两个连续的空闲区都有效且前一个地址大于后一个地址，交换它们的位置。



# 五、实验结果：

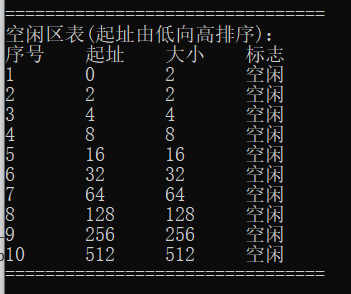
## 5.1要求用户指定作业数量以及内存分配算法：

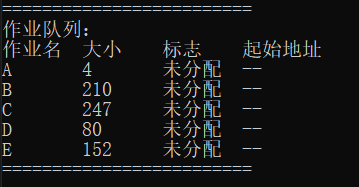


## 5.2最先适应算法展示：

### 5.2.1初始化相关变量和信息：

我们设置内存大小为 1024，空闲区表的大小分别为 2，2，4，8，16，32， 64，128，256，512，标志均为【空闲】，生成 5 个作业，作业名为从 A~E，随 机给它分配大小，标志均为【未分配】，已分配区表的 10 项均为【空栏目】，如 下所示：

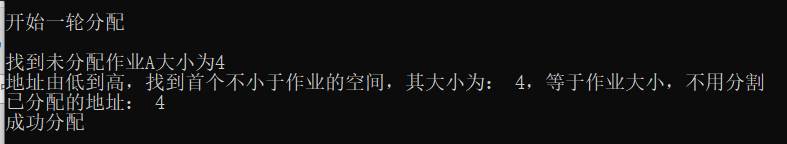
 



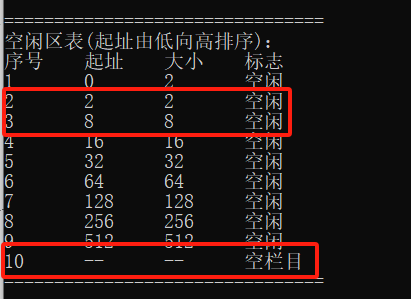
### 5.2.2第一轮相关展示：

* **分配空间**

找到第一个未分配作业 A，其大小为 4，找到首个不小于作业大小的空间，为序号为3的空间，其大小为 4,等于作业大小，不需要分割。



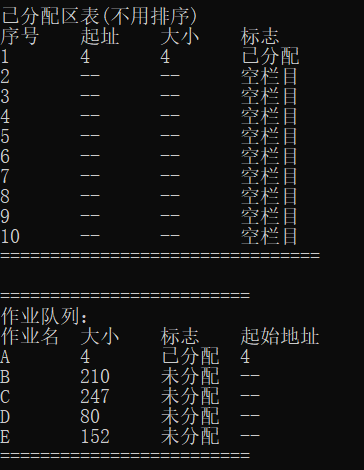
* **更新空闲区表：**



因为作业A恰好将第三个空闲区占满，因此现在就只有9个空闲区了

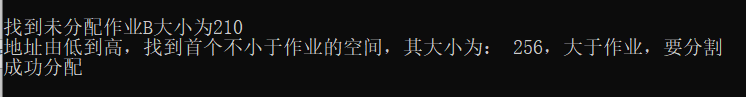
* **分配作业** A **后更新已分配区表和作业队列：**

已分配区表的起始地址等于占用空闲区的起始地址 4，大小即为作业 A 的大小4， 分配作业名为 A,修改标志为【已分配】。



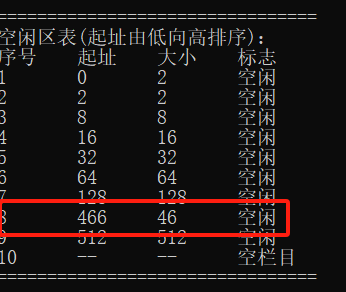
### 5.2.3第二轮相关展示

* **分配空间**



找到第二个未分配作业 B，其大小为 210，找到首个不小于作业大小的空间，为序号为8的空间，其大小为 256,大于作业大小,需要分割。

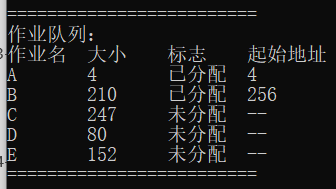
* **更新空闲区表：**



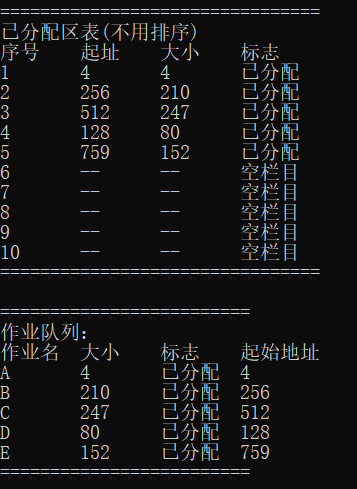
作业 B 找到的第一个不小于自己大小的空闲区为第8 个空闲区，起始地址=原来的起始地址（256）+A 的大小（210）=466，空闲区的大小=原来的大小（256）- A 的大小 （210）=46

* **分配作业** B **后更新已分配区表和作业队列：**

已分配区表的起始地址等于占用空闲区的起始地址 128，大小即为作业 B 的大小 108， 分配作业名为 B,修改标志为【已分配】。

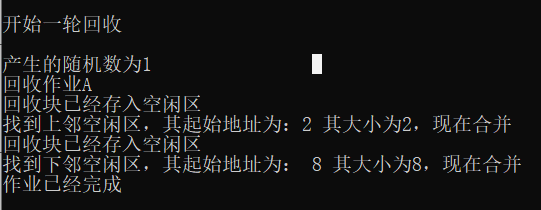
### 5.2.4其余三轮结果如下：



### 5.2.5相关回收过程：

* **第一轮回收：**

根据产生的随机数来判断是否回收作业，由于产生的随机数为 1，回收作业 A

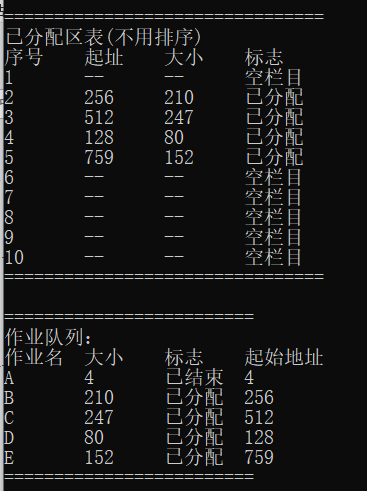


其中：此时找到的上邻空闲区，其起始地址为：2其大小为：2，2+2=4，正好是该作业块的起始地址，因此可以合并。

同理，找到下邻空闲区：该作业快的起始地址（4）+作业块的大小（4）=8，正好为3号空闲区的启示地址（8），因此可以正常的合并。

回收完成。

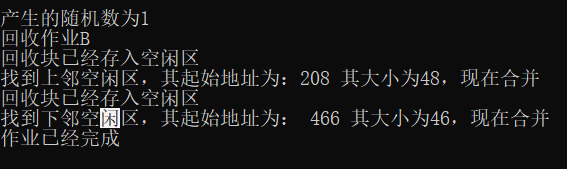
回收之后更新相关队列：

因为刚刚才分配的时候，恰好占据了一个分区，所以在回收之前就只有9个分区，然后经过了合并，成为了现在的8个空闲分区！

* **第二轮回收：**

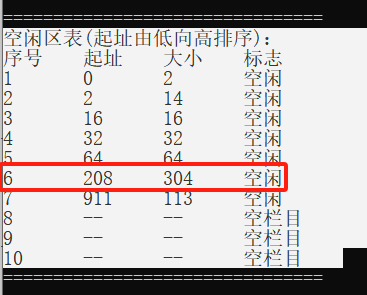
产生的随机数为1，因此需要回收作业B



此时找到的上邻空闲区，其起始地址为：208其大小为：48，208+48=156，正好是该作业块的起始地址，因此可以合并。

同理，找到下邻空闲区：该作业快的起始地址（256）+作业块的大小（210）=466，正好为3号空闲区的启示地址（466），因此可以正常的合并。

更新空闲区表：

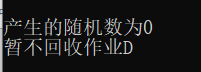


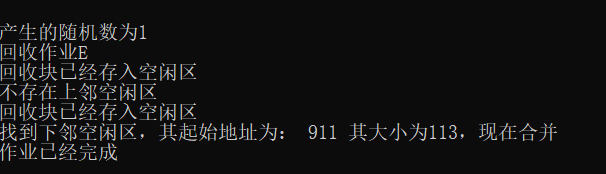
其中的起始地址为上邻区空闲区域的起始地址，大小为：48+210+46=304。

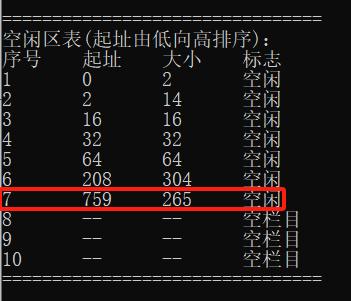
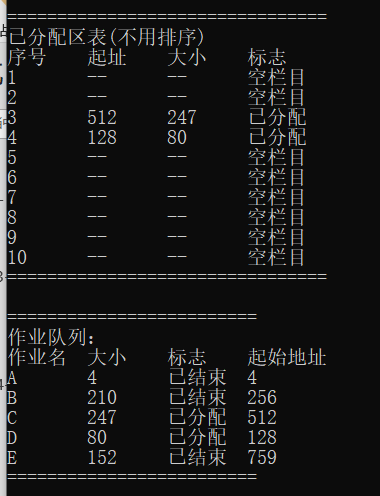
更新相关队列：

**其余轮的操作同理：**

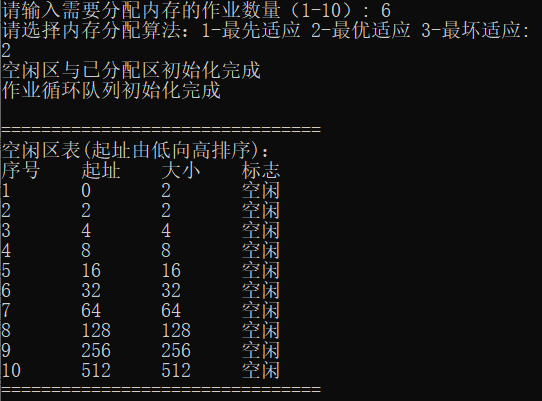


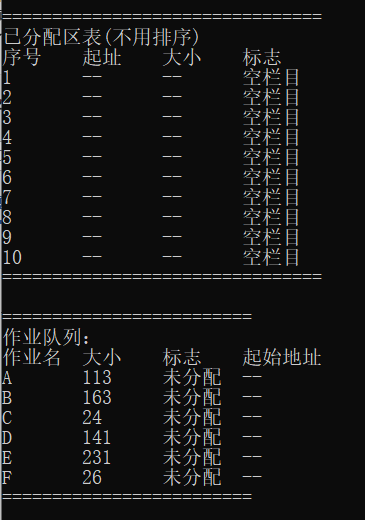
 

## 5.3最优适应（Best-fit）算法展示：

### 5.3.1初始化相关信息和变量

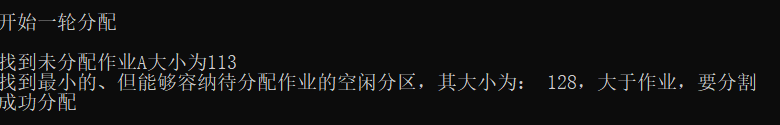
与如上的操作一致，这一次我输入的是对6个作业进行操作。编号由A-F





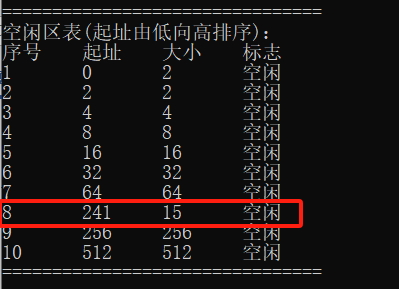
### 5.3.2第一轮相关展示

* **分配空间**



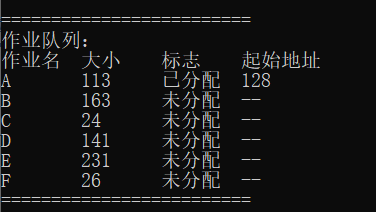
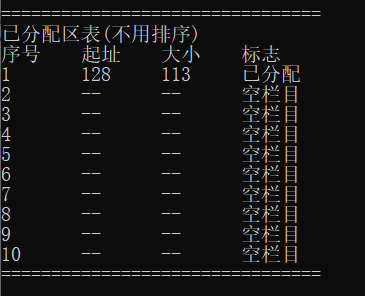
第一个分配的作业A的大小为113，找到最小的、但能够容纳待分配作业的空闲分区，其大小为： 128，大于作业，要分割

* **更新空闲区表：**

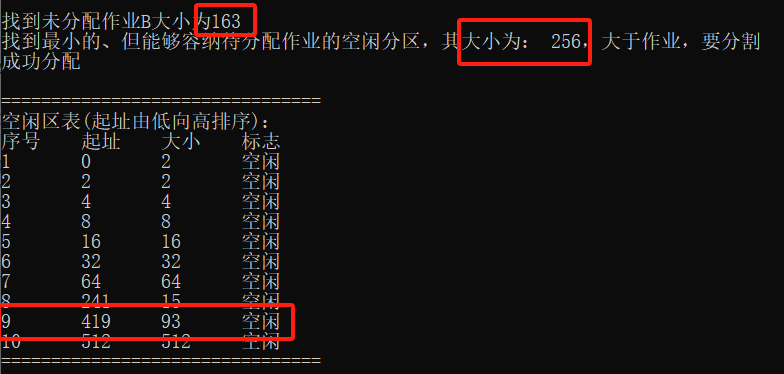


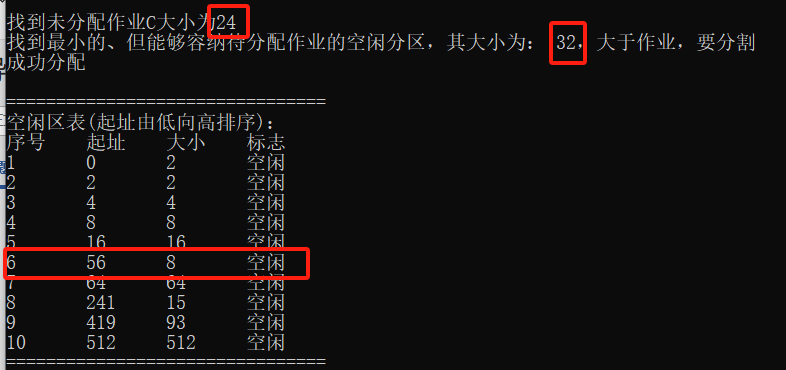
此时空闲区表的起始地址为,128+113=241，其大小为128-113=15。

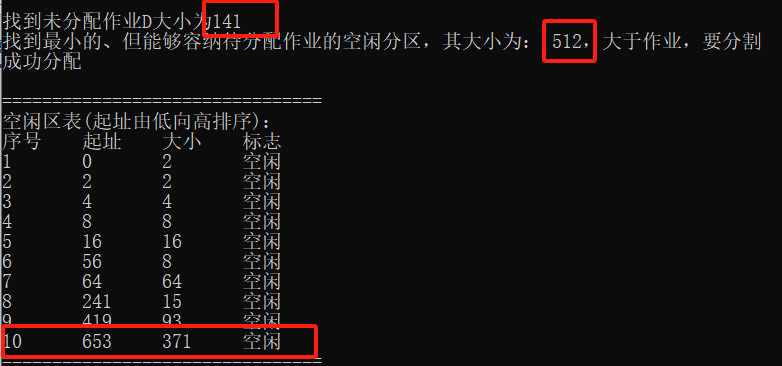
* **更新已分配区表和作业队列**

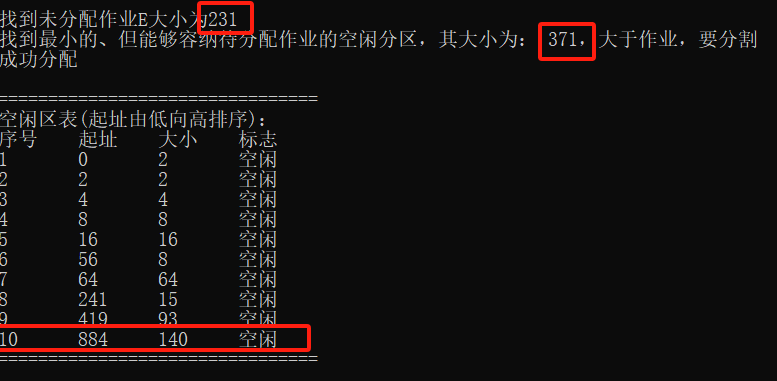


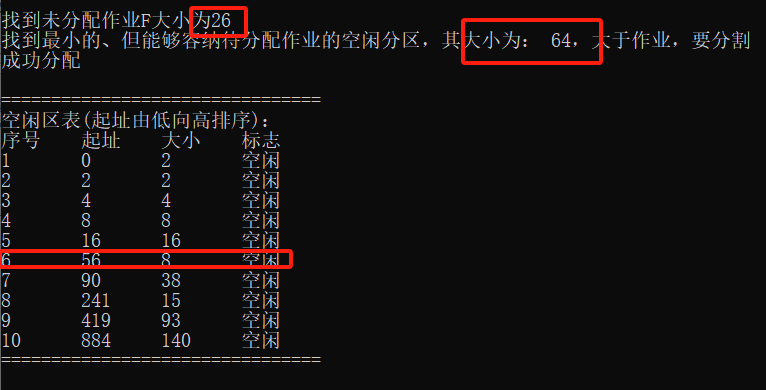
### 5.3.3后续轮相关展示：









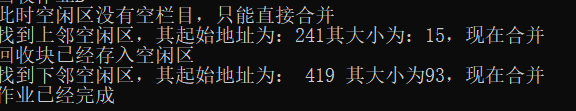


### 5.3.4相关回收过程：



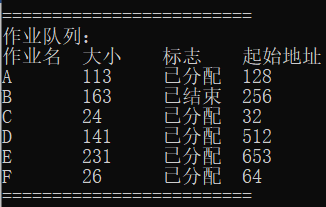
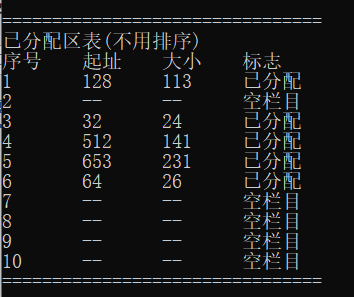
* **第二轮回收**

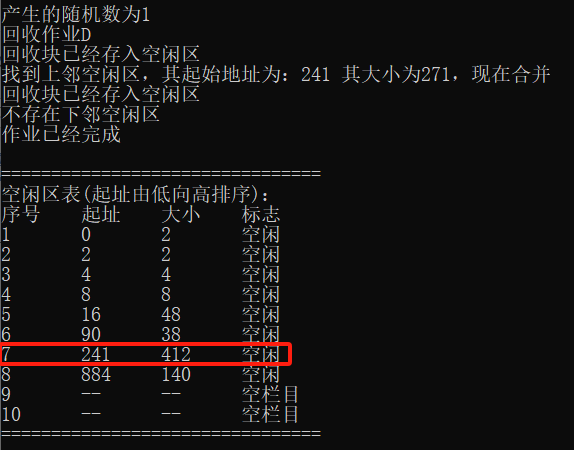
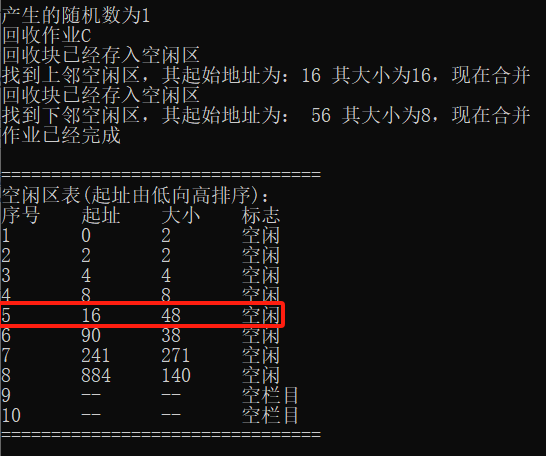
**回收作业B的内存**

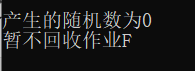
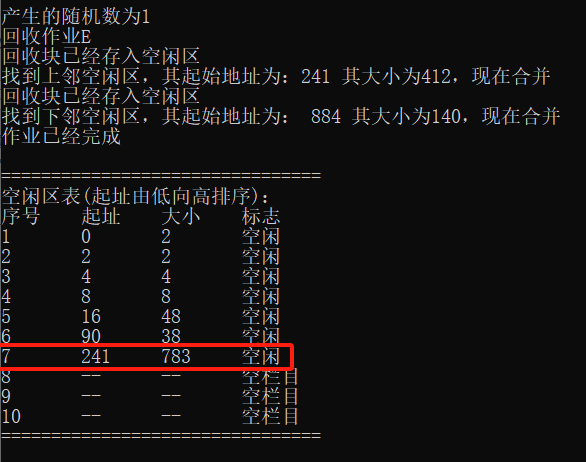




其中，合并之后的空闲区的起始地址为上领空闲区的起始地址241，大小为：15+163+93=271。



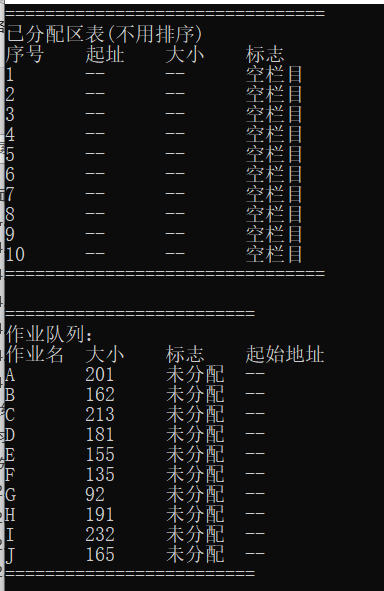




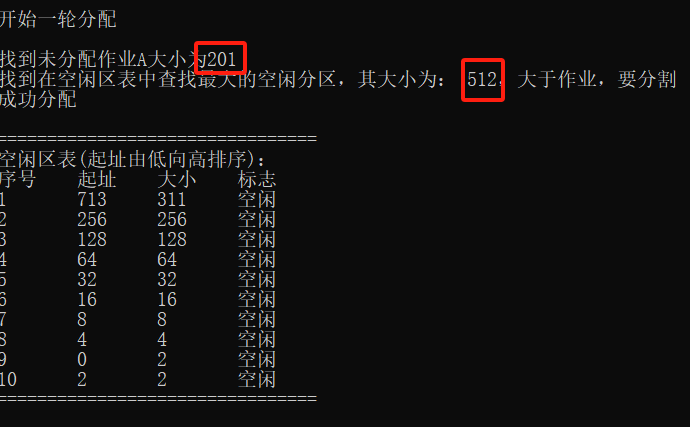
## 5.4最坏适应（Worst-fit）算法展示：

### 5.4.1初始化相关信息和变量：

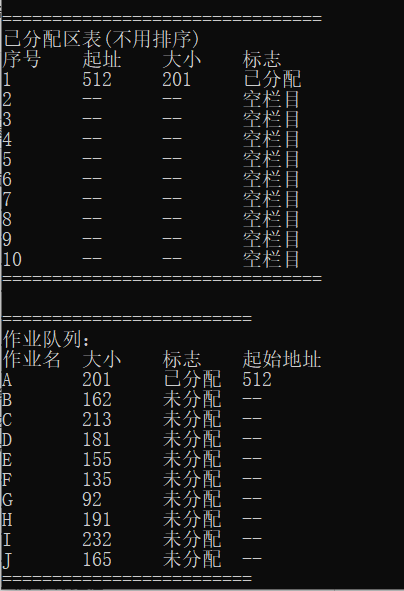
这一次我分配了10个作业，作业号为从A到J。具体信息如下图所示。



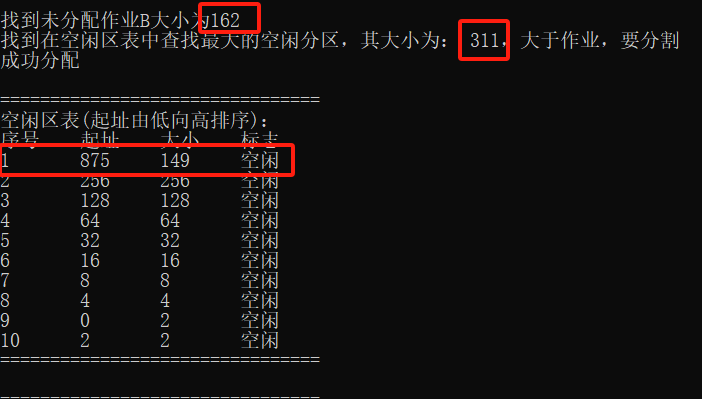
### 5.4.2第一轮分配：



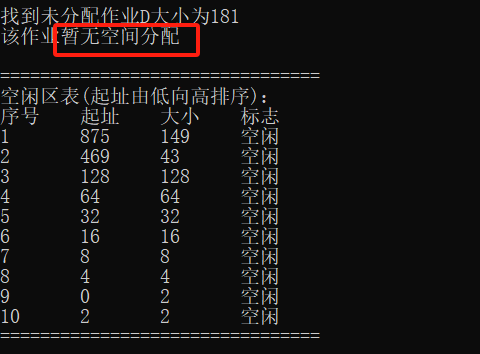
这里，选区选择了最大空闲分区，体现了最坏适应算法！



同理如下所示：



遇到没有足够空间的情况下，先暂时不分配内存给该作业。

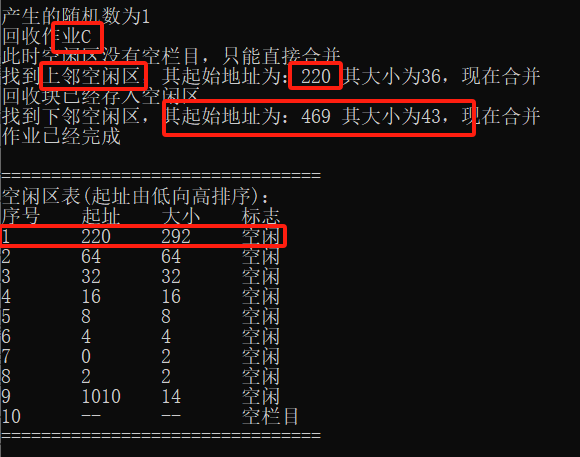


第一轮分配的结果如下：



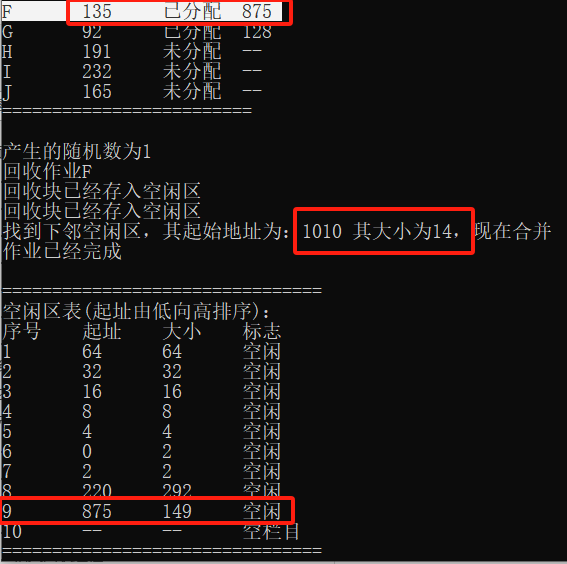
### 5.4.3第一轮回收：

作业A和作业B 都暂时不回收，对作业C进行回收，将其上下空闲邻区进行合并，之后得到如图所示的结果。

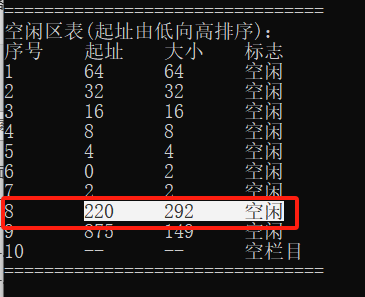


回收作业F：

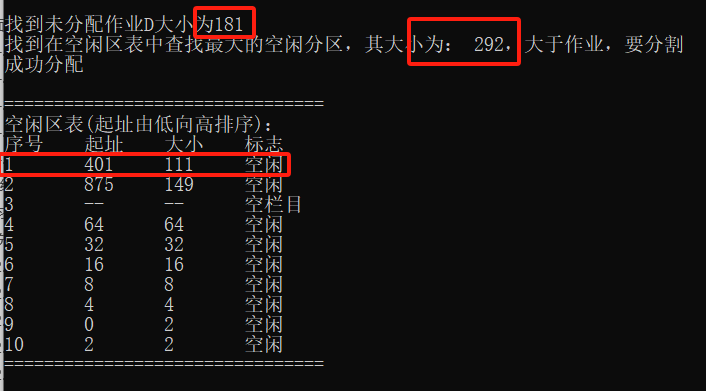
只和其下邻空闲区合并，其起始地址为该作业块的起始地址，大小为该作业块的大小（135）+空闲区的大小（14）=149



### 5.4.4进行第二轮分配：



在上一轮的内存回收的过程中，合并出了一块起始地址为220，大小为292的空闲区，因此可以将这部分空闲区的一部分划分给作业D。



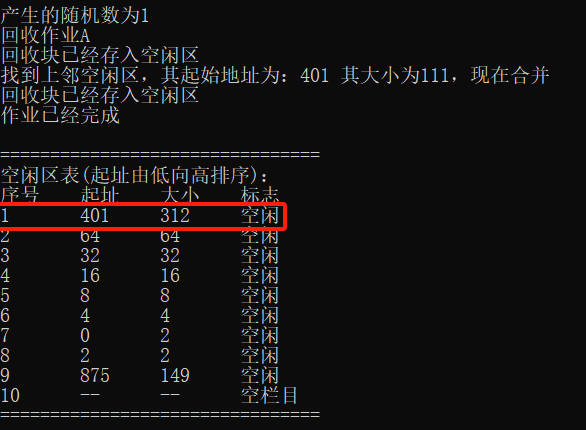
之后，进行新一轮的分配：

结果如下：

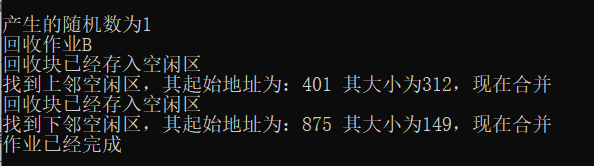
### 5.4.5进行第二轮的回收

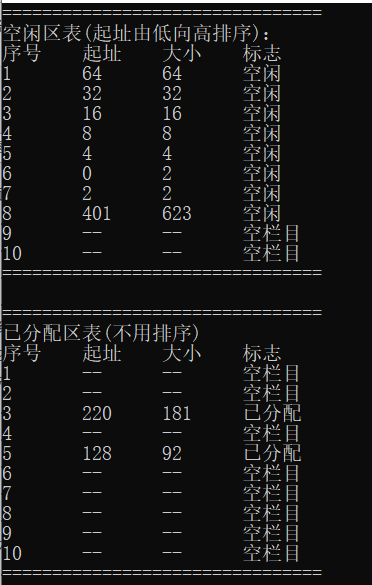
根据随机数，随机数为1，回收作业A



同理，最后回收的结果如图所示：

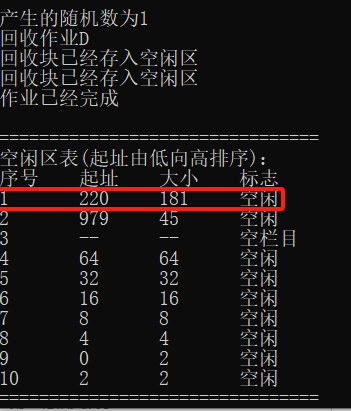
这一轮只回收了作业A和作业B



### 5.4.6其余轮中一些重要的过程：

在回收D的时候，发现并没有上下空闲邻区，但是有空栏目，那么将其设置为一个空闲邻区。



### 5.4.7最终结果：

经过了多轮的内存分配和回收，得到了如下结果：





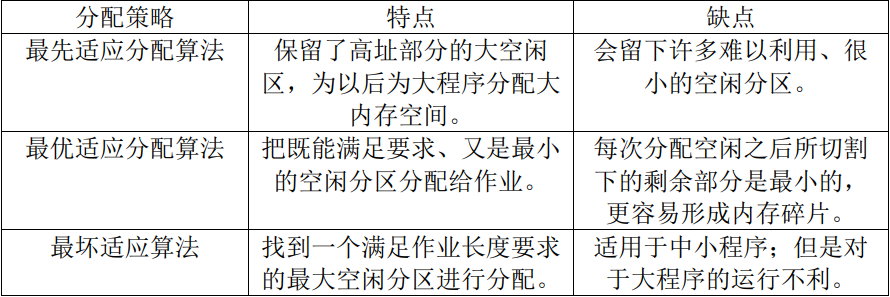
# 六、总结：

### 1-本实验中使用了三种算法：

最先适应分配算法，最优适应分配算法， 最坏适应算法。

这三种内存分配算法的关键在于对空闲区表的遍历和排序。最先适应算法不进行排序，按地址顺序遍历；最优适应算法按大小升序排序；最坏适应算法按大小降序排序。分配过程中的基本操作是一致的，都是寻找合适的空闲区并进行分配或拆分。

其中这三种算法的优缺点总结如下：



### 2-学习和收获：

通过实践三种内存分配策略，我对动态分区分配方法在主存分配及空间回收方面的应用有了深刻理解，掌握了动态分区分配中的调度算法及其具体实施方法。系统通过维护分区说明表来进行内存管理，其中包括可用空闲区分区表和已分配分区状态表。这种动态管理方式突显了其优越性：能够更加有效地分配内存空间。

在本次实验中，我采用顺序表数据结构，它进一步加深了我对顺序表插入、排序与删除等基本操作的理解与掌握。通过这种结构的应用，体现了顺序表在内存分配管理中的效率和实用性。