

**操作系统实验报告**

|  |  |
| --- | --- |
| **学 院：** | **计算机科学学院** |
| **专 业：** | **计算机科学与技术（创新实验班）** |
| **学 号：** | **42112255** |
| **姓 名：** | **何佳民** |
| **指导教师：** | **孙增国** |

**2023 年 11 月 28日**

**实验四 动态分区分配算法**

**一、实验目的**

通过这次实验，加深对动态分区分配算法的理解，进一步掌握首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法的实现方法。

**二、实验器材**

微机

**三、实验要求**

1）上机前认真复习动态分区分配算法，熟悉首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法的计算过程；

2）上机时独立编程、调试程序；

3）根据具体实验要求，完成好实验报告（包括实验的目的、内容、要求、源程序、实例运行结果截图、发现的问题以及解决方法）。

**四、实验内容**

**1.问题描述：**

设计程序模拟四种动态分区分配算法：首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法的工作过程。假设内存中空闲分区个数为n，空闲分区大小分别为P1, … ,Pn，在动态分区分配过程中需要分配的进程个数为m（m≤n），它们需要的分区大小分别为S1, … ,Sm，分别利用四种动态分区分配算法将m个进程放入n个空闲分区，给出进程在空闲分区中的分配情况。

**2.程序要求：**

1）利用首次适应算法、循环首次适应算法、最佳适应算法和最坏适应算法四种动态分区分配算法模拟分区分配过程。

2）模拟四种算法的分区分配过程，给出每种算法进程在空闲分区中的分配情况。

3）输入：空闲分区个数n，空闲分区大小P1, … ,Pn，进程个数m，进程需要的分区大小S1, … ,Sm，算法选择1-首次适应算法，2-循环首次适应算法，3-最佳适应算法，4-最坏适应算法。

4）输出：最终内存空闲分区的分配情况。

**3.问题分析：**

（1）分区分配算法：

① 首次适应算法(First Fit)

我们以空闲分区链为例来说明采用FF算法时的分配情况。FF算法要求空闲分区链以地址递增的次序链接。

-- 在分配内存时，从链首开始顺序查找，直至找到一个大小能满足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空间分配给请求者，余下的空闲分区仍留在空闲链中。

-- 若从链首直至链尾都不能找到一个能满足要求的分区，则此次内存分配失败，返回。

首次适应算法倾向于优先利用内存中低址部分的空闲分区，从而保留了高址部分的大空闲区。这给为以后到达的大作业分配大的内存空间创造了条件。其缺点是低址部分不断被划分，会留下许多难以利用的、很小的空闲分区，而每次查找又都是从低址部分开始，这无疑会增加查找可用空闲分区时的开销。

②循环首次适应算法(Next Fit)

该算法是由首次适应算法演变而成的。在为进程分配内存空间时，不再是每次都从链首开始查找，而是从上次找到的空闲分区的下一个空闲分区开始查找，直至找到一个能满足要求的空闲分区，从中划出一块与请求大小相等的内存空间分配给作业。

为实现该算法，应设置一起始查寻指针，用于指示下一次起始查寻的空闲分区，并采用循环查找方式，即如果最后一个(链尾)空闲分区的大小仍不能满足要求，则应返回到第一个空闲分区，比较其大小是否满足要求。找到后，应调整起始查寻指针。

该算法能使内存中的空闲分区分布得更均匀，从而减少了查找空闲分区时的开销，但这样会缺乏大的空闲分区。

1. 最佳适应算法(Best Fit)

所谓“最佳”是指每次为作业分配内存时，总是把能满足要求、又是最小的空闲分区分配给作业，避免“大材小用”。为了加速寻找，该算法要求将所有的空闲分区按其容量以从小到大的顺序形成一空闲分区链。这样，第一次找到的能满足要求的空闲区，必然是最佳的。

孤立地看，最佳适应算法似乎是最佳的，然而在宏观上却不一定。因为每次分配后所切割下来的剩余部分总是最小的，这样，在存储器中会留下许多难以利用的小空闲区。

1. 最坏适应算法(Worst Fit)

最坏适应分配算法要扫描整个空闲分区表或链表，总是挑选一个最大的空闲区分割给作业使用，其优点是可使剩下的空闲区不至于太小，产生碎片的几率最小，对中、小作业有利，同时最坏适应分配算法查找效率很高。该算法要求将所有的空闲分区按其容量以从大到小的顺序形成一空闲分区链，查找时只要看第一个分区能否满足作业要求。

但是该算法的缺点也是明显的，它会使存储器中缺乏大的空闲分区。最坏适应算法与前面所述的首次适应算法、循环首次适应算法、最佳适应算法一起，也称为顺序搜索法。

（2）分区分配操作

①分配内存：系统利用某种分配算法，从空闲分区链（表），中找到所需大小的分区。

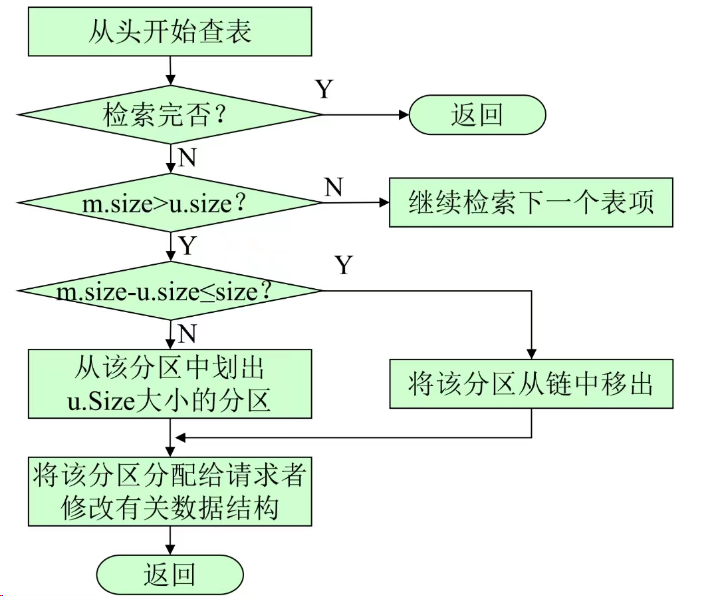


图 4-1 分配内存流程图

②回收内存:当进程运行完毕释放内存时，系统根据回收区2的首址，从空闲区链 (表) 中找到相应的插入点。

a) 回收区与插入点的前一个空闲分区F1相邻接，将回收区与插入点前的分区合并，不必为回收分区分配表项，修改前一分区F1的大小;

b) 回收区与插入点的后一空闲分区F2相邻接，此时可将两分区合并，形成新的空闲分区，回收区的首址作为新空闲区的首址，大小为两者之和；

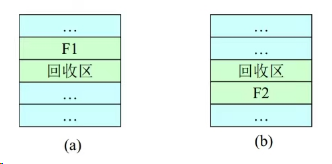


图 4-2 （a）（b）

**4.源程序：**

data.txt内容：  
9

16 16 8 32 64 32 8 16 64

6

A B C D E F

7 18 9 20 35 8

Python代码（Python版本3.9）：

# 动态分区分配算法

MAXNUMBER = 100

PartitionNum = 0 # 内存中空闲分区的个数

ProcessNum = 0 # 需要分配的进程个数

FreePartition = [0] \* MAXNUMBER # 空闲分区对应的内存

ProcessNeed = [0] \* MAXNUMBER # 需要分配的进程大小

LeftFreePartition = [0] \* MAXNUMBER

LeftProcessNeed = [0] \* MAXNUMBER

ProcessName = [''] \* MAXNUMBER

NameProcessToPartition = [[''] \* MAXNUMBER for \_ in range(MAXNUMBER)]

def read\_data\_function():

global PartitionNum, ProcessNum

with open("data.txt", "r") as read\_data:

PartitionNum = int(read\_data.readline().strip())

partition\_data = read\_data.readline().strip().split()

for i in range(PartitionNum):

FreePartition[i] = int(partition\_data[i])

ProcessNum = int(read\_data.readline().strip())

process\_name\_data = read\_data.readline().strip().split()

process\_need\_data = read\_data.readline().strip().split()

for i in range(ProcessNum):

ProcessName[i] = process\_name\_data[i]

ProcessNeed[i] = int(process\_need\_data[i])

def initial():

read\_data\_function()

for i in range(ProcessNum):

for j in range(PartitionNum):

NameProcessToPartition[i][j] = ''

LeftFreePartition[j] = FreePartition[j]

for i in range(ProcessNum):

LeftProcessNeed[i] = ProcessNeed[i]

def first\_fit():

print("\*\*\*\*\*\*\*\*\*\*\*首次适应算法\*\*\*\*\*\*\*\*\*\*\*")

initial()

for i in range(ProcessNum):

for j in range(PartitionNum):

if LeftProcessNeed[i] <= LeftFreePartition[j] and LeftFreePartition[j] != 0:

LeftFreePartition[j] -= LeftProcessNeed[i]

LeftProcessNeed[i] = 0

NameProcessToPartition[i][j] = ProcessName[i]

break

display()

def next\_fit():

print("\*\*\*\*\*\*\*\*\*\*\*循环首次适应算法\*\*\*\*\*\*\*\*\*\*\*")

initial()

next\_point = 0

for i in range(ProcessNum):

while True:

if LeftProcessNeed[i] <= LeftFreePartition[next\_point]:

LeftFreePartition[next\_point] -= LeftProcessNeed[i]

if LeftFreePartition[next\_point] == 0:

next\_point = (next\_point + 1) % PartitionNum

LeftProcessNeed[i] = 0

NameProcessToPartition[i][next\_point] = ProcessName[i]

break

else:

next\_point = (next\_point + 1) % PartitionNum

display()

def best\_fit():

print("\*\*\*\*\*\*\*\*\*\*\*最佳适应算法\*\*\*\*\*\*\*\*\*\*\*")

initial()

best = [{'partitionSize': FreePartition[i], 'id': i} for i in range(PartitionNum)]

for i in range(ProcessNum):

best.sort(key=lambda x: x['partitionSize'])

for j in range(PartitionNum):

if LeftProcessNeed[i] <= best[j]['partitionSize']:

best[j]['partitionSize'] -= LeftProcessNeed[i]

LeftProcessNeed[i] = 0

NameProcessToPartition[i][best[j]['id']] = ProcessName[i]

break

LeftFreePartition[best[j]['id']] = best[j]['partitionSize']

display()

def worst\_fit():

print("\*\*\*\*\*\*\*\*\*\*\*最坏适应算法\*\*\*\*\*\*\*\*\*\*\*")

initial()

worst = [{'partitionSize': FreePartition[i], 'id': i} for i in range(PartitionNum)]

for i in range(ProcessNum):

worst.sort(key=lambda x: x['partitionSize'], reverse=True)

for j in range(PartitionNum):

if LeftProcessNeed[i] <= worst[j]['partitionSize']:

worst[j]['partitionSize'] -= LeftProcessNeed[i]

LeftProcessNeed[i] = 0

NameProcessToPartition[i][worst[j]['id']] = ProcessName[i]

break

LeftFreePartition[worst[j]['id']] = worst[j]['partitionSize']

display()

def display():

print("需要分配内存的进程名：", end=' ')

for i in range(ProcessNum):

print(ProcessName[i], end=' ')

print("\n需要分配内存的进程分区大小：", end=' ')

for i in range(ProcessNum):

print(ProcessNeed[i], end=' ')

print("\n分配结果：")

print("分区序号:", end=' ')

for i in range(PartitionNum):

print("分区" + str(i + 1), end=' ')

print("\n分区大小:", end=' ')

for i in range(PartitionNum):

print(FreePartition[i], end=' ')

print("\n剩余大小:", end=' ')

for i in range(PartitionNum):

print(LeftFreePartition[i], end=' ')

print("\n分配进程情况:")

for i in range(PartitionNum):

for j in range(ProcessNum):

if NameProcessToPartition[j][i]:

print(NameProcessToPartition[j][i] + ": (分区" + str(i + 1) + ")")

print("\n\*\*\*\*\*\*\*\*结束\*\*\*\*\*\*\*\*\*\*")

def select\_algorithm(choose\_algorithm):

if choose\_algorithm == 1:

first\_fit()

elif choose\_algorithm == 2:

next\_fit()

elif choose\_algorithm == 3:

best\_fit()

elif choose\_algorithm == 4:

worst\_fit()

else:

print("请输入正确的序号：")

def main():

choose\_algorithm = 5

while choose\_algorithm:

print("请选择实现的算法：")

print("\*\*\*\*\* 1 - 首次适应算法 \*\*\*\*\*")

print("\*\*\*\*\* 2 - 循环首次适应算法 \*\*\*\*\*")

print("\*\*\*\*\* 3 - 最佳适应算法 \*\*\*\*\*")

print("\*\*\*\*\* 4 - 最坏适应算法 \*\*\*\*\*")

print("\*\*\*\*\* 0 - 结束 \*\*\*\*\*")

choose\_algorithm = int(input("chooseAlgorithm = "))

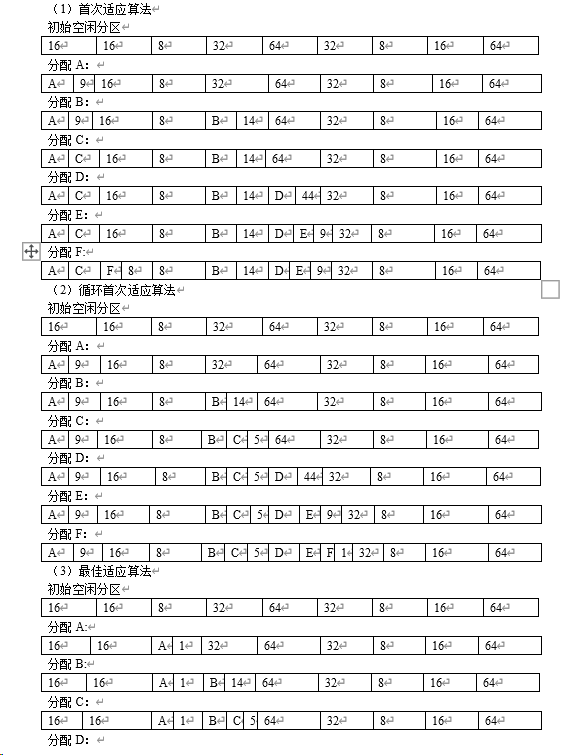
select\_algorithm(choose\_algorithm)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**5.运行结果与分析：**

为了检查算法运行结果是否正确，用作业例题来说明，如下图。

  
图 4-3 作业题目

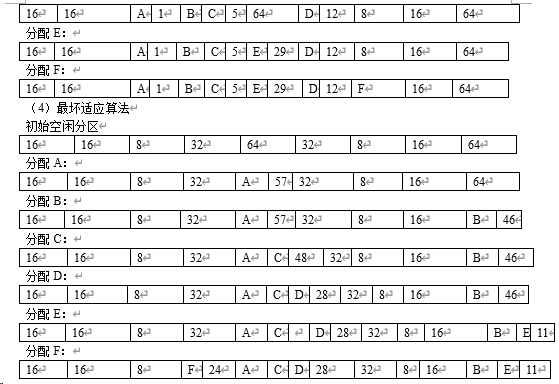


图 4-4 作业题目

算法运行结果：  
输出界面如下：

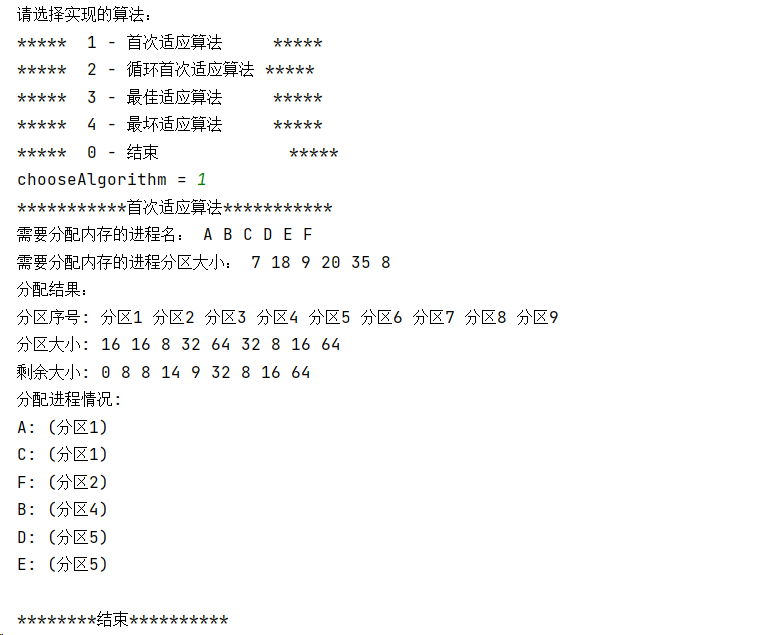


图 4-5 首次适应算法

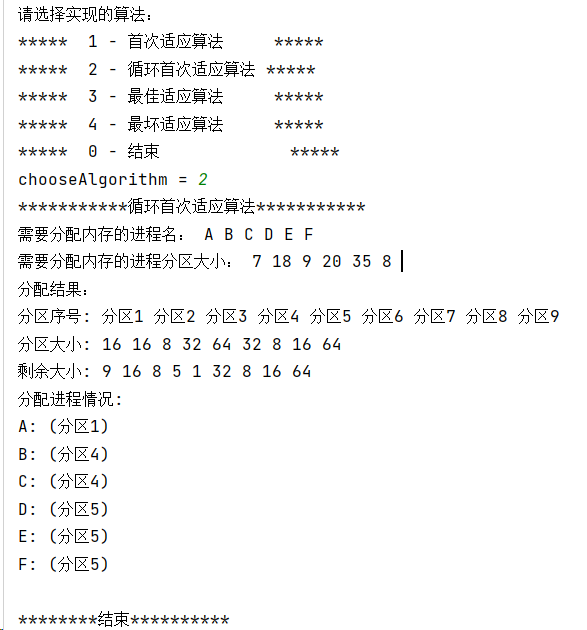


图 4-6 循环首次适应算法

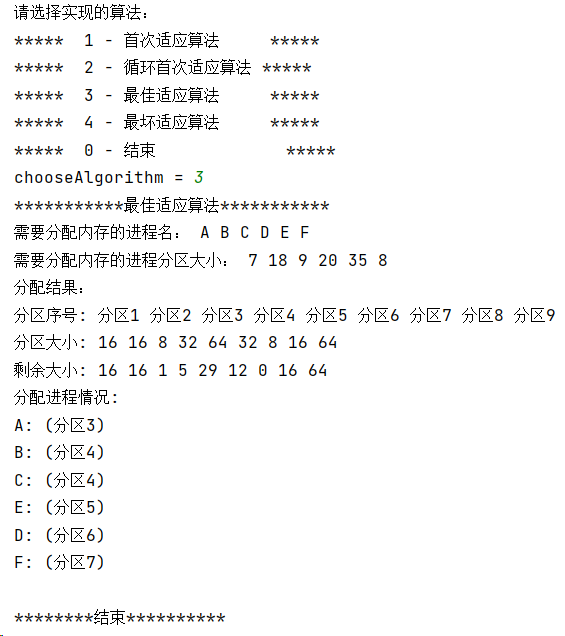


图 4-7 最佳适应算法

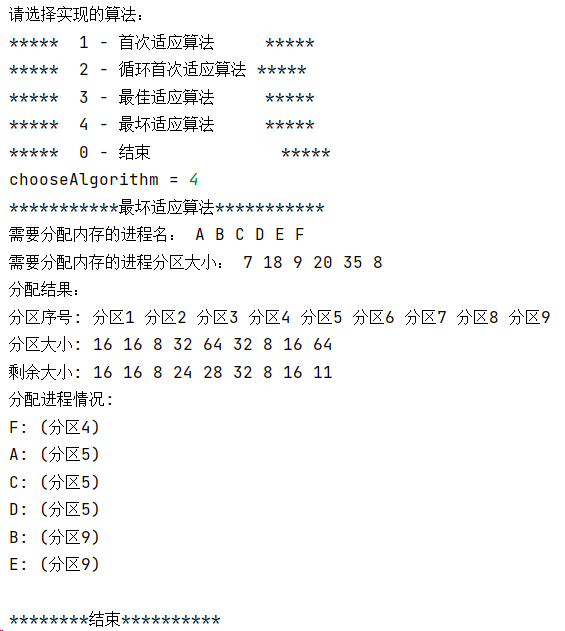


图 4-8 最坏适应算法

由图可知，四种算法运行结果都正确。

**五、实验总结**

1.

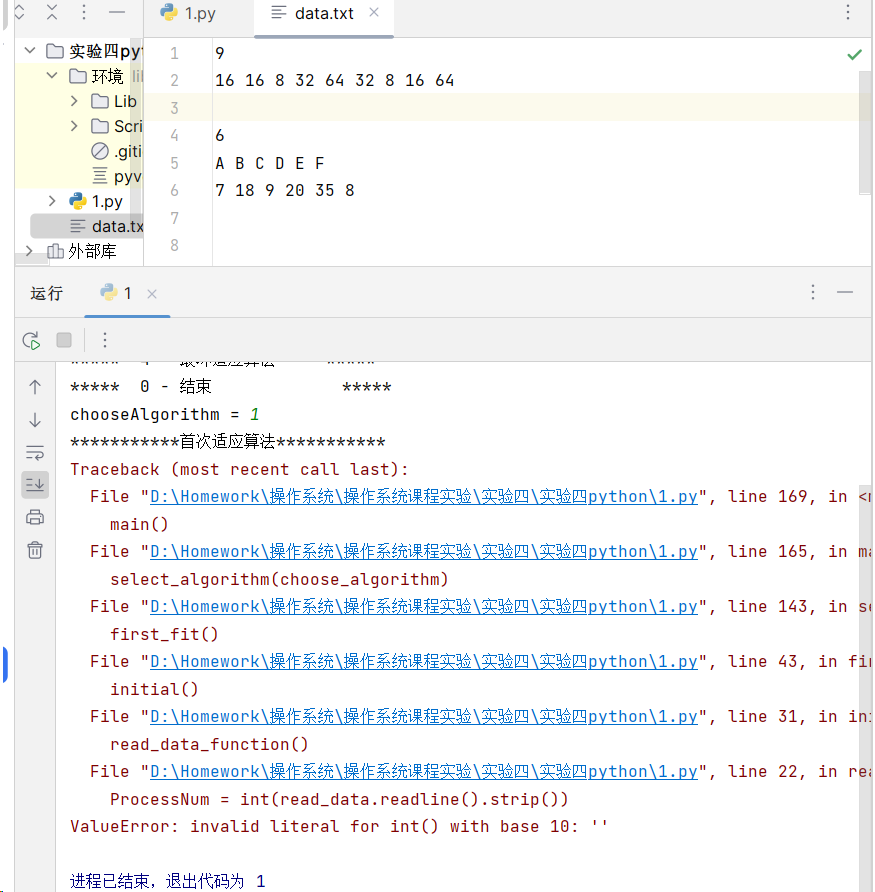


图 4-9 错误截图

解决办法：  
注意python读取数据时，每一行都要连续，不要出现空行，除非代码写明换行读取。

2.

数组索引越界，PartitionNum或ProcessNum的值超过MAXNUMBER，程序将试图访问数组的不存在的索引，从而导致错误。

解决办法:

在read\_data\_function函数中添加检查，确保读取的值不超过MAXNUMBER。

3.

在各种分配算法中，如果没有足够的空间为进程分配内存，可能不会有明确的提示或处理。

解决办法: 当进程无法分配内存时，程序应提供相应的提示或处理逻辑。

4.

在写循环首次使用算法时，出现错误，这是错误结果：

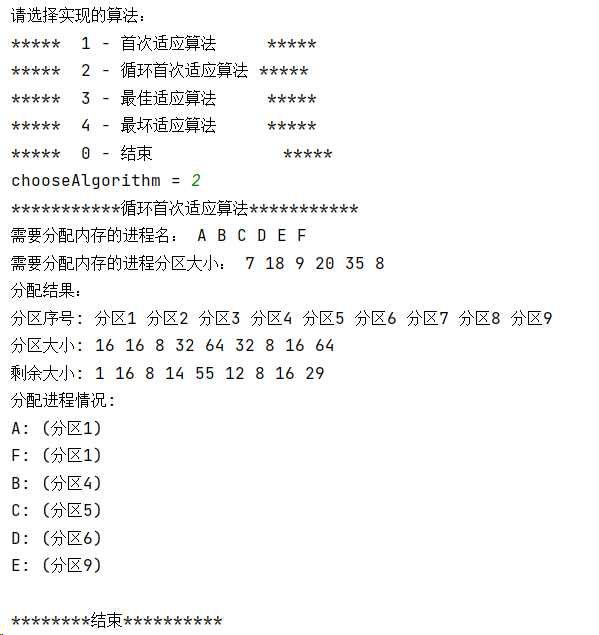


图 4-10 错误结果截图

解决办法：这是之前的next\_fit函数：  


图 4-11 代码截图

在实现循环首次适应算法（Next Fit）时，应当确保在分配完一个进程后，剩余的空闲内存仍然可以用于后续的分配。当前的 next\_fit 函数实现中，在分配完内存后，没有正确处理剩余空间，导致它可能无法被后续进程使用。为了解决这个问题，可以对 next\_fit 函数进行修改，确保在一个进程分配后，剩余的空闲内存仍然参与到后续的分配中。下面是修改后的 next\_fit 函数：

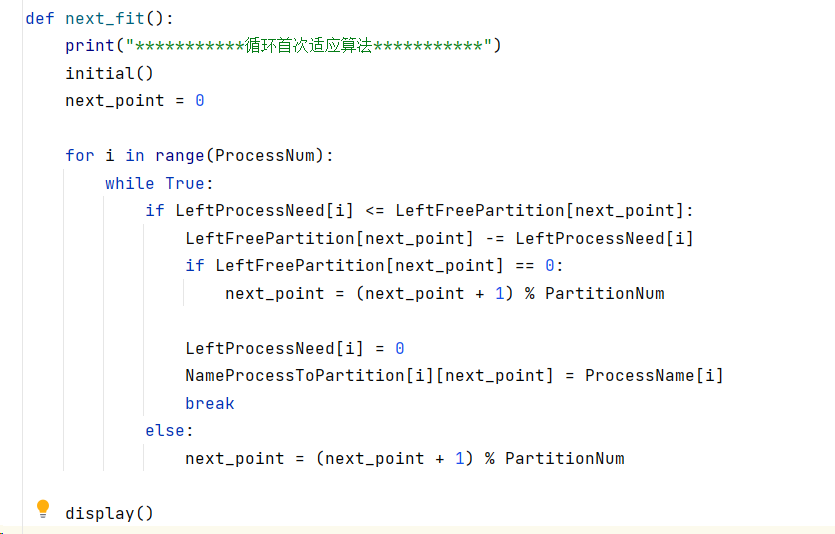


图 4-12 代码截图

在这个修改中，确保了即使分配后某个分区的剩余大小为0，指针next\_point也会移到下一个分区。这样，剩余的空间可以被后续的进程使用。同时，这也保证了算法会在所有分区中循环，而不是仅在未分配完的分区中循环。

正确结果：

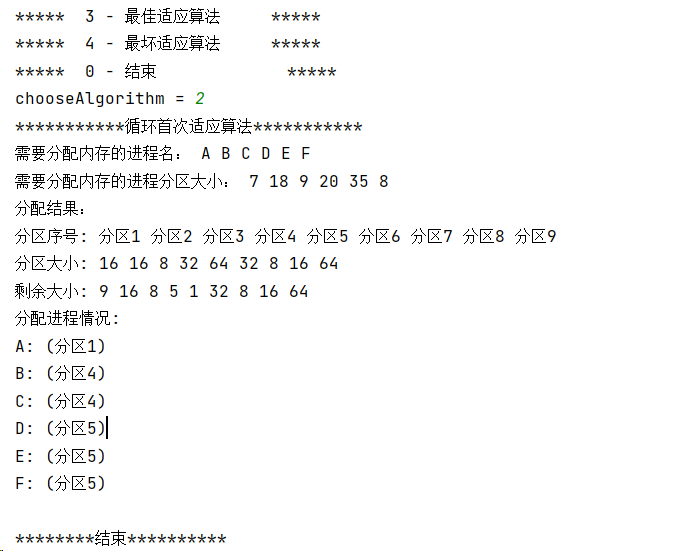


图 4-13 正确结果截图