实验三 存储管理问题

薛皓阳 *2020010647 xuehy20@mails.tsinghua.edu.cn*

动态分区存储管理

一、问题描述

分区存储管理有固定分区和动态分区两种方法。固定分区把内存划分为若干个固定大小 的连续分区，每个分区的边界固定；而动态分区并不预先将内存事先划分成分区，当程序需 要装入内存时系统从空闲的内存区中，采用不同的分配算法分配大小等于程序所需的内存空 间。它有效地克服了固定分区方式中，由于分区内部剩余内存空置造成浪费的问题。

请基于空闲内存分区链表的存储管理，设计一个动态分区存储管理程序，支持包括首次适配法、下次适配法、最佳适配法和最坏适配法在内的不同分区分配算法。

二、实现要求

1. 维护一个记录已分配内存分区和空闲内存分区的链表；
2. 设计申请、释放函数循环处理用户的请求；
3. 实现首次适配法、下次适配法、最佳适配法和最坏适配法四种分区分配算法；
4. 可视化展示内存使用情况。

三、实验环境

Windows操作系统，使用Python实现。

四、具体实现

1. 设计思路

本实验可以分为三个主要的任务：1. 实现空闲链表数据结构，并完成相关的维护算法，给出程序接口；2. 实现四种分配算法；3. 实现GUI界面，调用相关算法接口和数据结构接口，实现功能。下面针对这三个任务逐一分析。

1. 空闲链表

1. 链表节点表示一个连续的内存块，包含信息有：内存块号、是否空闲、起始位置、内存大小、下一节点。为方便具体实现相关算法这里加入一个新的属性：上一节点。
2. 初始状态，链表除表头外应该只有一个节点，号为0，空闲，起始位置为0，大小默认为1024KB，，下一节点为None，上一节点为head。
3. 运行中可能的操作有：申请内存、释放内存，对象均为某个被上层算法选中的节点。
    1. 申请内存时，传入需要的大小，进行移除判断，通过则将原有的一个空闲节点变为一个指定大小忙碌节点和剩下的空闲节点，返回忙碌节点。
    2. 释放内存时，首先判断该内存是否忙碌，忙碌则根据该节点前后节点空闲情况进行释放，保证释放后没有连续的空闲节点，返回释放后的空闲节点。
4. 考虑到链表节点包含较多信息，这里在`Class.py`中实现了`Memory`类和`process`类，链表节点为`Memory`类的实例，进程信息由`process`类的实例传入，上述算法均实现为`Memory`类的类内方法。

2. 分配算法

1. 首次分配：传入链表首节点`head`，逐一查找是否可分配，是则分配并返回tuple类型变量：(成功，分配后的内存块)，否则返回(失败，`None`)。
2. 下次分配：传入当前链表指针`cur_pointer`，逐一查找是否可分配，是则分配并返回tuple类型变量：(成功，分配后的内存块，当前指针)，否则返回(失败，`None`，`None`)。在调用时需要调用两次。如果第一次不成功可能则传入表头`head`再查找一次。
3. 最佳分配：传入链表首节点`head`，记录一个最佳分配对`tuple: best_pair`，包含(最佳匹配节点，最小内存差)。逐一遍历所有节点并更新最佳匹配对。最后返回tuple类型变量：(成功，分配后的内存块)，或(失败，`None`)。
4. 最差分配：传入链表首节点`head`，记录一个最差分配对`tuple: worst_pair`，包含(最差匹配节点，最大内存差)。逐一遍历所有节点并更新最差匹配对。最后返回tuple类型变量：(成功，分配后的内存块)，或(失败，`None`)。

3. GUI

应当包含的元素有：

- 一个表示内存的空白矩形框，展示当前内存分配情况
- 两个文本输入框`Size`和`Name`，用于表示接下来待分配进程的需要内存大小和进程名，应当对输入做合法性检查并提示。
- 两个按钮`Allocate`和`Free`，用于申请和释放内存， 鼠标按下按钮后可以触发相关函数实现GUI界面的变化，同时修改底层数据结构。
- 鼠标可以点击选择内存块。

2. 程序结构与核心代码分析

1. 空闲链表

在`Classes.py`中实现。

链表节点类`Memory`：

```
class Memory():
    def __init__(self, num=0, is_free=True, begin=0, size=1024, next=None, prev=None) -> None:
        self.mem_num    = num
        self.is_free    = is_free
        self.begin      = begin
        self.size       = size
        self.next       = next
        self.prev       = prev

    def allocate_mem(self, size):

    def free_mem(self):

    # 更新序号
    def update_num(self, is_allocate:bool):
```

申请内存函数：

当前节点为空闲节点`self`，先申请一个新节点并初始化，更新本节点的起始位置和大小、序号后，调整前后节点指针，最后返回新节点。

```
    def allocate_mem(self, size):
        ...
        ...

        # 申请新节点
        new_mem = Memory(self.mem_num, False, self.begin, size, self,
self.prev)
        if self.prev != None:
            self.prev.next = new_mem

        self.prev = new_mem
        self.begin = self.begin + size
        self.size  = self.size - size

        ...
        ...
```

释放内存函数：

由于释放后原有内存会变为空闲，需要考虑新的空闲内存与前后空闲内存的合并，因此考虑四种情况：前后皆空闲、仅前空闲、仅后空闲和前后皆不空闲，合并后保证没有连续的空闲节点。最终无论哪种情况都会返回空闲节点。

```
    def free_mem(self):
        # 左右皆空
        if self.prev != None and self.next != None and self.prev.is_free and
self.next.is_free:
            if self.next.next != None:
                self.next.next.update_num(is_allocate=False)
                self.next.next.update_num(is_allocate=False)
                self.next.next.prev = self.prev
            self.prev.next = self.next.next
            self.prev.size = self.prev.size + self.size + self.next.size
            del self.next
            temp = self.prev
            del self
            return temp
        # 左空
        elif self.prev.is_free:
            self.prev.size = self.prev.size + self.size
            self.prev.next = self.next
            if self.next != None:
                self.next.prev = self.prev
                self.next.update_num(is_allocate=False)
            temp = self.prev
            del self
            temp.is_free = True
            return temp
        # 右空
        elif self.next != None and self.next.is_free:
            self.next.update_num(is_allocate=False)
            self.size = self.size + self.next.size
            if self.next.next != None:
                self.next.next.prev = self
            temp = self.next
```

```
            self.next = self.next.next
            del temp
            self.is_free = True
            return self
        # 左右皆满（或无）
        else:
            self.is_free = True
            return self
```

进程类`process`：

```
class process():
    def __init__(self, need, num=None, arrive_time=None, require_time=None):
        self.num            = num
        self.need_mem       = need
        self.arrive_time    = arrive_time
        self.require_time   = require_time

    # 用于堆排序
    def __lt__(self, other):
        assert isinstance(other, process)
        return self.require_time < other.require_time # type:ignore
```

2. 分配算法

在`Algorithms.py`中实现

首次适配

逐一查找是否可分配。

```
def First_fit(head:Memory, p:process) -> typing.Tuple[bool, Memory or None]:
    cur = head
    assert isinstance(cur, Memory)
    while(cur != None):
        if cur.is_free and cur.size >= p.need_mem:
            return (True, cur.allocate_mem(p.need_mem))
        cur = cur.next
    return (False, None) # type: ignore
```

下次分配

与首次分配的区别在于传入指针为当前指针而非头指针；需要额外返回当前指针。

```
def Next_fit(cur_pointer:Memory, p:process) -> typing.Tuple[bool, Memory or None, Memory]:
    cur_pointer = cur_pointer.next # type: ignore
    assert isinstance(cur_pointer, Memory)
    while(cur_pointer != None):
        if cur_pointer.is_free and cur_pointer.size >= p.need_mem:
            return (True, cur_pointer.allocate_mem(p.need_mem), cur_pointer.prev)# type: ignore
        cur_pointer = cur_pointer.next # type: ignore
    return (False, None, None) # type: ignore
```

最佳分配

遍历所有节点找到最佳适配并返回

```
def Best_fit(head:Memory, p:process) -> typing.Tuple[bool, Memory or None]:
    cur = head
    best_pair = (None, 99999)
    assert isinstance(cur, Memory)
    while(cur != None):
        if cur.is_free and cur.size >= p.need_mem:
            if cur.size - p.need_mem < best_pair[1]:
                best_pair = (cur.prev.next, cur.size - p.need_mem) # type: ignore
        cur = cur.next
    return (True, best_pair[0].allocate_mem(p.need_mem)) if best_pair[0] != None else (False, None) # type: ignore
```

最差分配

区别在于找最坏而非最好

```
def Worst_fit(head:Memory, p:process) -> typing.Tuple[bool, Memory or None]:
    cur = head
    worst_pair = (None, -1)
    assert isinstance(cur, Memory)
    while(cur != None):
        if cur.is_free and cur.size >= p.need_mem:
            if cur.size - p.need_mem > worst_pair[1]:
                worst_pair = (cur.prev.next, cur.size - p.need_mem) # type: ignore
        cur = cur.next
    return (True, worst_pair[0].allocate_mem(p.need_mem)) if worst_pair[0] != None else (False, None) # type: ignore
```

3. GUI

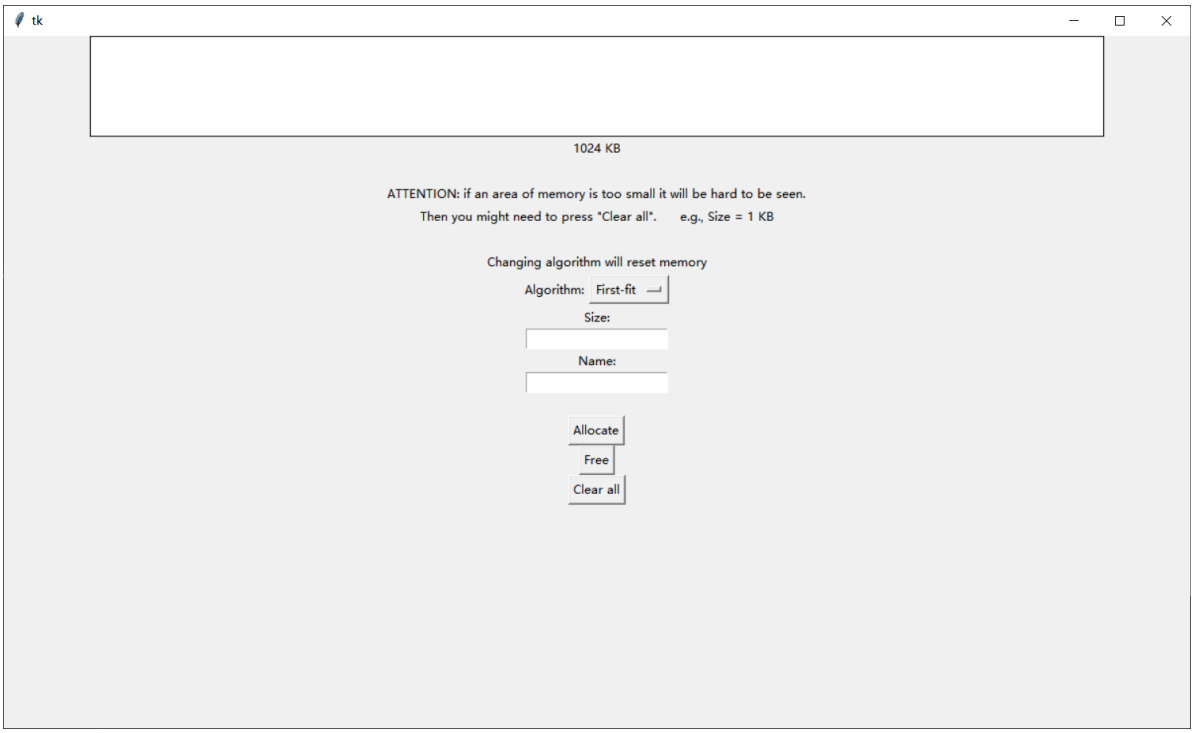
主要难度在于输入检查，这里使用了`RATIO`以表示矩形框放缩大小。

```
import tkinter as tk
from bridge import bridge

RATIO = 1
selected_rectangle = None
name_list = []
id_list = []
prev_algorithm = None


def allocate_rectangle():

def select_rectangle(rectangle_id):

def free_rectangle(mode=None):

def clear_all():

```

```
19  def initialization():
20
21  if __name__ == '__main__':
22      Bridge = bridge()
23      Bridge.main()
24      initialization()
```

3. 结果验证

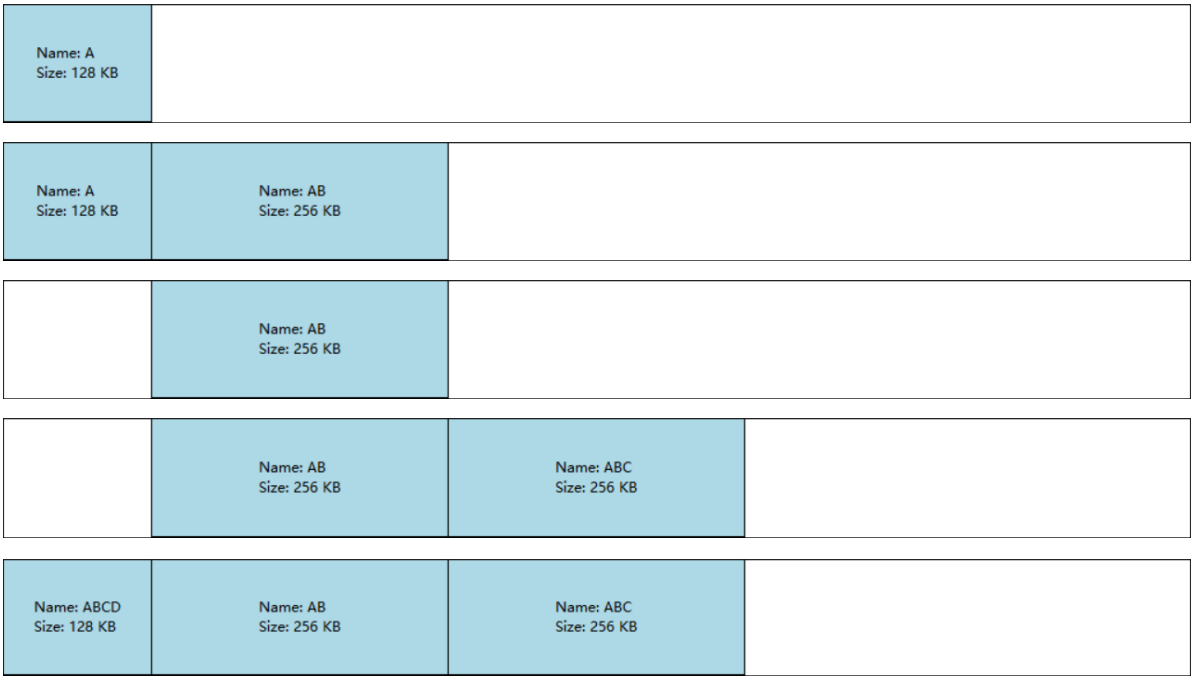
UI界面



首次匹配：

操作序列：

申请128K的A；申请256K的AB；释放A；申请256K的ABC（此时可以看到自动选择了后面的空闲分区）；申请128K的ABCD。



可以看到符合要求。

下次匹配

操作序列：

以下均申请大小为128KB的内存。

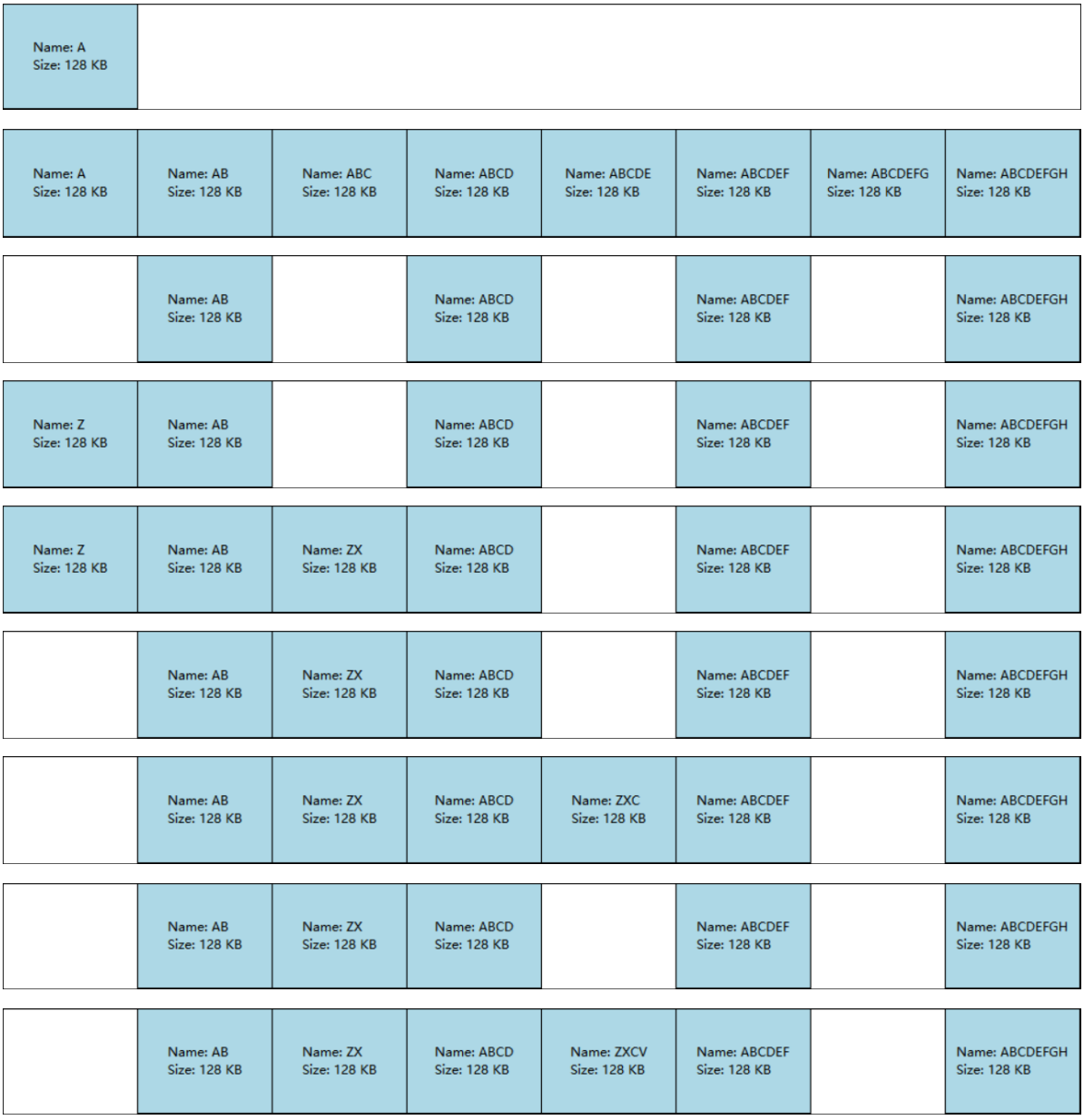
申请内存A、AB、...、ABCDEFGH；

释放奇数号内存块；

申请Z、ZX；释放Z；

申请ZXC（此时可以看到并没有从第一个开始分配内存，而是从下一个空闲分区开始）

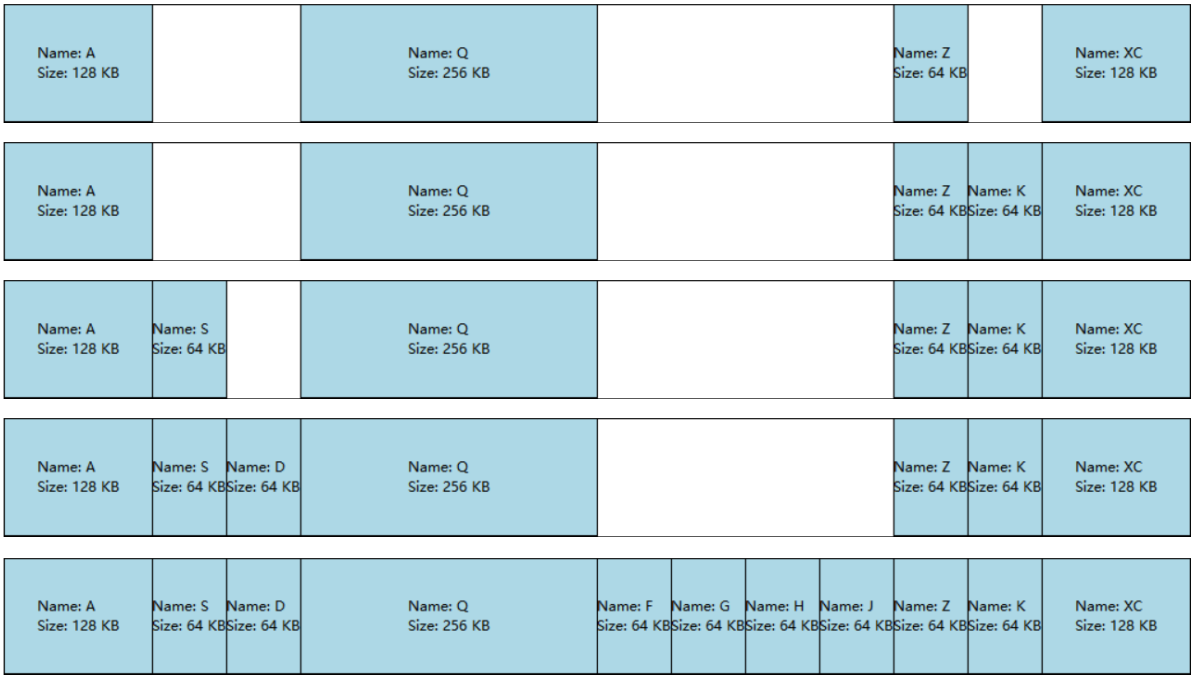
释放ZXC（指针指向前一个）；申请ZXCV（此时在ZXC被释放的内存处申请，符合逻辑）



最佳适配

首先通过合适的申请和释放产生如第一张图的分配结构，其中三个空闲分区分别为128K、256K、64K；

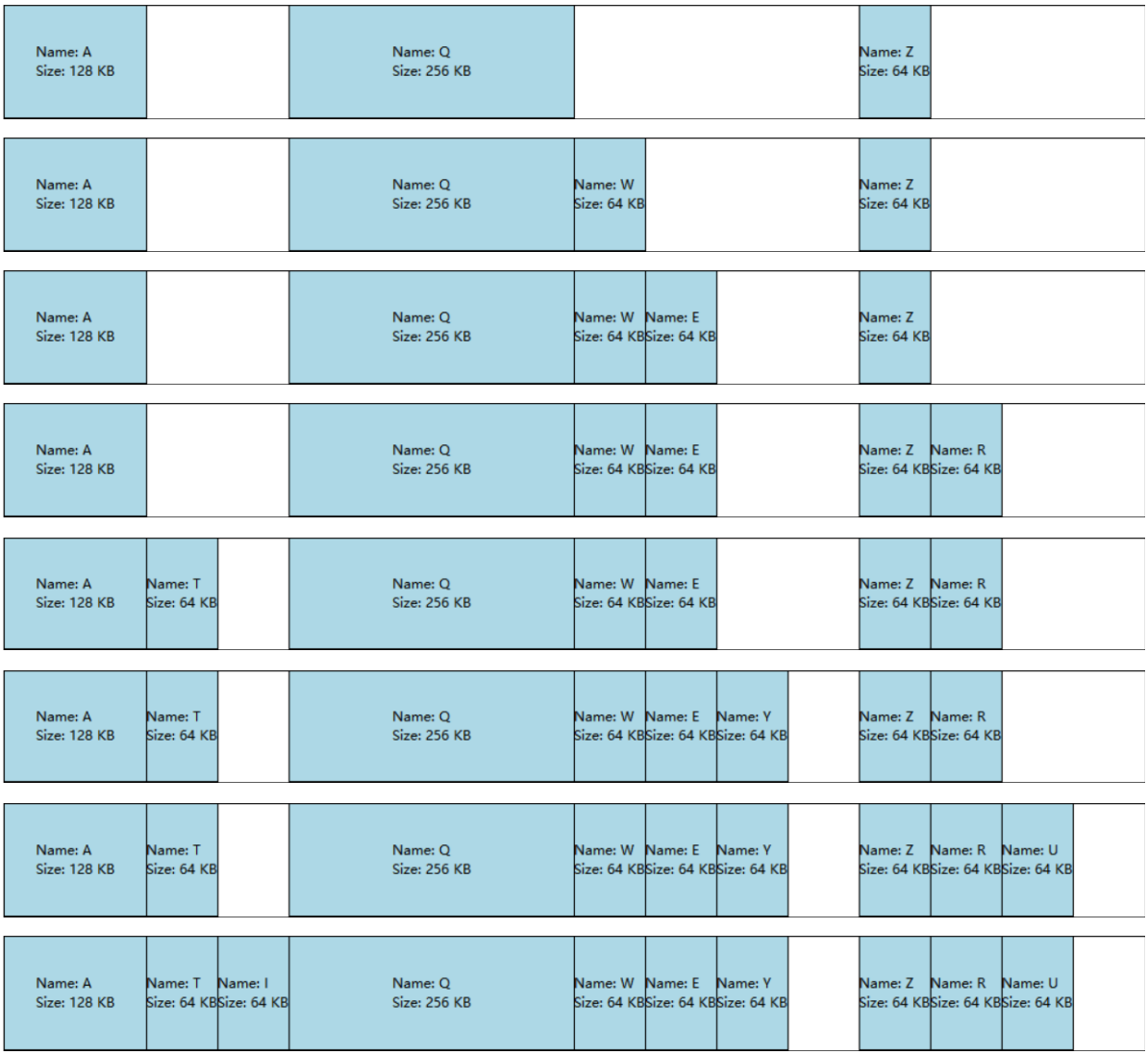
申请多个64K大小的内存块，可以看到优先填充小的空闲分区，大的空闲分区得以保留。



最差匹配

操作序列：

首先产生如第一张图的结构，空闲分区大小分别为128K、256K、192K。然后不断申请大小为64K的内存。可以看到优先分配较大的空闲分区。



Name: A Size: 128 KB	Name: T Size: 64 KB	Name: I Size: 64 KB	Name: Q Size: 256 KB	Name: W Size: 64 KB	Name: E Size: 64 KB	Name: Y Size: 64 KB	Name: O Size: 64 KB	Name: Z Size: 64 KB	Name: R Size: 64 KB	Name: U Size: 64 KB	
Name: A Size: 128 KB	Name: T Size: 64 KB	Name: I Size: 64 KB	Name: Q Size: 256 KB	Name: W Size: 64 KB	Name: E Size: 64 KB	Name: Y Size: 64 KB	Name: O Size: 64 KB	Name: Z Size: 64 KB	Name: R Size: 64 KB	Name: U Size: 64 KB	Name: P Size: 64 KB

4. 额外内容

额外实现了`CPU.py`，可以针对输入申请队列自动模拟内存的分配。输入队列的每一个申请单元包含：申请大小、进程序号、到达时间、占用时间。使用`generate_sequence.py`可以自动生成随机队列。运行`CPU.py`即可看到进程的申请、释放情况。由于时间限制，此内容未做GUI适配。

结果：

```
1 7 1 6 added, time = 1
2 126 2 19 added, time = 2
3 257 4 14 added, time = 4
1 7 1 0 deleted, time = 7
4 376 7 16 added, time = 7
3 257 4 0 deleted, time = 18
2 126 2 0 deleted, time = 21
4 376 7 0 deleted, time = 23
5 457 8 20 added, time = 23
6 45 10 19 added, time = 23
7 319 14 17 added, time = 23
7 319 14 0 deleted, time = 40
8 355 15 14 added, time = 40
6 45 10 0 deleted, time = 42
5 457 8 0 deleted, time = 43
9 329 16 4 added, time = 43
9 329 16 0 deleted, time = 47
10 414 19 18 added, time = 47
8 355 15 0 deleted, time = 54
10 414 19 0 deleted, time = 65
```

每行内容为：

进程号，申请大小，到达时间，剩余占用时间、申请或释放、当前时间

五、思考题

基于位图和空闲链表的存储管理各有什么优劣？如果使用基于位图的存储管理，有何额 外注意事项？

位图：

优点：

1. 空间利用率高，可以节省内存空间，区分一个内存分配单位是否占用只需要一个1bit标志位；
2. 修改内存分配状态快，效率高，只需修改位图中对应内存的标志位即可；
3. 占据内存空间固定，与进程数无关。

缺点：

1. 需要额外维护一个进程列表与位图中坐标相对应，这样才能在释放进程时修改对应进程占据内存分配单位的标志位。
2. 固定了最小分配单位，如果最小分配单位定义不合适可能：1. 分配单位过小，位图占用内存空间过大，且一个进程可能占据很多的分配单元，修改内存分配状态效率很低（尤其是在分配新内存时需要匹配连续的空闲分配单元，设置过小可能导致匹配性能很差）；2. 分配单位过大，则某些占用内存很少的进程可能不足以占据整个分配单元，导致产生内碎片（同固定分区）。

空闲链表：

优点：

1. 支持任意大小的进程
2. 链表包含了进程信息，因此释放进程时可以直接从表头开始查找到相应进程并释放，无需额外维护进程列表。
3. 占用的内存空间可以动态调整。

缺点：

1. 修改内存分配情况速度较慢，涉及多个节点之间指针的修改，同时需要新申请或释放内存以产生或删除节点；
2. 查找速度慢，需要遍历整个链表；
3. 如果有大量占据内存很小的进程，链表长度可能很长，导致查找、修改性能下降，同时占据内存空间较大。

使用基于位图的存储管理的注意事项：

1. 需要针对进程大小分布设计最小分配单元的大小，以避免过小或过大造成的性能影响。
2. 需要考虑数据的更新频率，选择合适的更新策略。

六、附录

文件结构

```
pack
|
├─codes
|     Algorithms.py                    # 实现四种分配算法
|     bridge.py                        # 作为链表系统和GUI系统的中间调度系统
|     Classes.py                       # 实现了内存块Memory类和进程process类
|     CPU.py                           # 实现了内存块具体的调度算法
|     generate_sequence.py             # 生成模拟时序上的内存申请队列
|     GUI.py                           # GUI界面，直接运行即可验证实验任务
|
└─report
    |  report.md
    |  report.pdf
    └─images

```