**操作系统实验三实验报告**

**动态分区存储管理**

学号：2019011008

班级：无92

姓名：刘雪枫

**目 录**

[一、 实验平台 1](#_Toc105868416)

[二、 目录结构 1](#_Toc105868417)

[三、 实验原理与算法设计 2](#_Toc105868418)

[（一） 首次适配 3](#_Toc105868419)

[（二） 下次适配 3](#_Toc105868420)

[（三） 最佳适配 3](#_Toc105868421)

[（四） 最坏适配 3](#_Toc105868422)

[（五） 内存释放 3](#_Toc105868423)

[四、 代码实现 3](#_Toc105868424)

[（一） 实现有序链表 3](#_Toc105868425)

[（二） 定义内存块 5](#_Toc105868426)

[（三） 定义内存管理策略 6](#_Toc105868427)

[I) 首次适配 12](#_Toc105868428)

[II) 下次适配 13](#_Toc105868429)

[III) 最佳适配 16](#_Toc105868430)

[IV) 最差适配 18](#_Toc105868431)

[（四） 用户指定分配策略 19](#_Toc105868432)

[五、 样例测试 20](#_Toc105868433)

[六、 内存使用可视化 23](#_Toc105868434)

[七、 实验心得 26](#_Toc105868435)

[八、 思考题 26](#_Toc105868436)

# 实验平台

本实验的所有代码均在 .NET 6.0 平台运行，使用C# 语言（C# 10）和XAML语言编写。其中，核心算法部分可以在Windows、Linux和MacOS系统下的 .NET 6.0运行时上运行，内存可视化部分使用WPF，仅支持Windows操作系统。实验所用Windows系统为Windows 11 21H2，Linux系统为Ubuntu 20.04 LTS。

# 目录结构

代码的目录结构如下：

.

└── DynamicPartitionedStorageManagement

├── DynamicPartitionedStorageManagement.sln

├── GUIEntrance

│   ├── App.xaml

│   ├── App.xaml.cs

│   ├── AssemblyInfo.cs

│   ├── GUIEntrance.csproj

│   ├── InitializeMemoryWindow.xaml

│   ├── InitializeMemoryWindow.xaml.cs

│   ├── MainWindow.xaml

│   ├── MainWindow.xaml.cs

│   ├── MainWindowViewModel.cs

│   └── RelayCommand.cs

├── MemoryManager

│   ├── BestFitAllocator.cs

│   ├── FirstFitAllocator.cs

│   ├── IMemoryManager.cs

│   ├── MemoryBlock.cs

│   ├── MemoryManager.cs

│   ├── MemoryManager.csproj

│   ├── MemoryManagerFactory.cs

│   ├── NextFitAllocator.cs

│   └── WorstFitAllocator.cs

├── UnitTest

│   ├── BestFitAllocatorTest.cs

│   ├── FirstFitAllocatorTest.cs

│   ├── NextFitAllocatorTest.cs

│   ├── UnitTest.csproj

│   ├── Usings.cs

│   └── WorstFitAllocatorTest.cs

└── Utils

├── SortedLinkedList.cs

└── Utils.csproj

其中，Utils和MemoryManager两个目录是核心算法，UnitTest是对算法进行测试的测试项目，GUIEntrance是内存可视化展示工具，其可执行文件为GUIEntrance.exe，附在额外的GUI目录中。

# 实验原理与算法设计

本实验设计了首次适配（first fit）、下次适配（next fit）、最佳适配（best fit）、最差适配（worst fit）四种动态分区管理算法。每种算法均提供申请内存、释放内存、查看空闲分区列表、查看已分配分区列表，几种接口。

## 首次适配

首次适配是当申请内存时，从空闲分区链表中从头开始查找，从找到的第一个大小不小于申请内存大小的分区中分配内存。

## 下次适配

下次适配是当申请内存时，从上一次分配的分区位置开始查找，从找到的第一个大小不小于申请内存大小的分区中分配内存。

## 最佳适配

最佳适配是在申请内存时，空闲链表中找到大小最接近申请内存大小的分区，并在该空闲分区中分配内存。

## 最坏适配

最佳适配是在申请内存时，空闲链表中找到大小最大的分区，并在该闲分区中分配内存。

## 内存释放

在用户归还内存时，先检查用户要归还的内存是否是自己已经分配出去的。如果是，则将该块内存放回空闲分区链表。如果这块分区和其他空闲分区正好是相连的，则将它们合并成一个空闲分区。

# 代码实现

## 实现有序链表

由于一些分配算法查找时可能需要按内存地址从小到大进行查找，因此空闲分区链表最好是有序的，即按内存从小到大排列。由于C# 并没有提供有序链表的实现，其只提供了链表（LinkedList<T>）和有序列表（SortedList<T>，为连续存储而非链式存储），而对于空闲分区管理需要频繁插入和删除的清醒，采用链式存储更加合理，因此需要自己手动实现有序链表SortedLinkedList<T>，提供插入、删除、查找等接口。其实现位于Utils/SortedLinkedList.cs下。提供的接口如下：

1. **namespace** Utils
2. {
3. **public** **class** SortedLinkedList<T>
4. : ICollection<T>, IEnumerable<T>, System.Collections.IEnumerable, IReadOnlyCollection<T>, System.Collections.ICollection
5. {
6. **public** SortedLinkedList(IComparer<T> comparer);
7. **public** SortedLinkedList(IEnumerable<T> collection, IComparer<T> comparer);
8. **public** **void** Clear();
9. **public** **void** Add(T item);
10. **public** LinkedListNode<T> AddAndGetNode(T item);
11. **public** **bool** Contains(T item);
12. **public** LinkedListNode<T>? LowerBound(T item);
13. **public** LinkedListNode<T>? UpperBound(T item);
14. **public** **bool** Remove(T item);
15. **public** **void** Remove(LinkedListNode<T> item);
16. **public** LinkedListNode<T>? First { **get**; }
17. **public** LinkedListNode<T>? Last { **get**; }
18. **public** **void** CopyTo(T[] array, **int** arrayIndex);
19. **void** System.Collections.ICollection.CopyTo(Array array, **int** index);
20. **public** IEnumerator<T> GetEnumerator();
21. System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator();
22. **public** **bool** IsReadOnly { **get**; }
23. **public** **bool** IsSynchronized { **get**; }
24. **public** **object** SyncRoot { **get**; }
25. **public** **int** Count { **get**; }
26. }
27. }

## 定义内存块

由于要进行内存块的管理，因此定义内存块数据结构。一个内存块包含两项数据：起始地址（Memory）和内存块的大小（Size）。由于不同的平台，内存地址可能不同，且为了减少unsafe块而避免使用指针类型byte\*，因此采用C# 的nuint类型来表示内存地址（nuint为平台相关整数类型，始终与内存地址大小相同）。而内存块大小为了简洁，则使用int（32位有符号整数）表示，则其最多能表示2G的内存块大小，足够本实验使用。内存块数据结构如下：

1. **namespace** MemoryManager
2. {
3. **public** **class** MemoryManager
4. {
5. **protected** **class** MemoryBlock
6. {
7. **public** MemoryBlock(nuint memory, **int** size);
8. **public** nuint Memory { **get**; **private** init; }
9. **public** **int** Size { **get**; **private** init; }
10. }
11. }
12. }

由于要将内存块按地址大小进行排序，因此定义内存块的比较器：

1. **namespace** MemoryManager
2. {
3. **public** partial **class** MemoryManager
4. {
5. **protected** **class** MemoryBlockComparer : IComparer<MemoryBlock>
6. {
7. **public** **int** Compare(MemoryBlock? x, MemoryBlock? y)
8. {
9. **if** (x **is** **null** || y **is** **null**)
10. {
11. **throw** **new** ArgumentException("Null Memory Block cannot compare!");
12. }
14. **if** (x.Memory < y.Memory)
15. {
16. **return** -1;
17. }
18. **else** **if** (x.Memory == y.Memory)
19. {
20. **return** 0;
21. }
22. **else**
23. {
24. **return** 1;
25. }
26. }
27. }
28. }
29. }

即，使用内存地址的大小作为内存块比较的判据。

由于外部要获取空闲和已分配内存块的信息，因此定义一个记录类型来记录内存块相关信息，信息也包括内存地址和大小：

1. **namespace** MemoryManager
2. {
3. **public** record **struct** MemoryBlockInfo(nuint Memory, **int** Size);
4. }

内存块的实现位于文件MemoryManager/ MemoryBlock.cs中。

## 定义内存管理策略

内存管理器需要实现分配内存、释放内存、获取空闲链表、获取已分配分区链表，四种操作。接口定义位于MemoryManager/ IMemoryManager.cs中：

1. **namespace** MemoryManager
2. {
3. **public** **interface** IMemoryManager
4. {
5. nuint? AllocateMemory(**int** size);
6. **bool** FreeMemory(nuint memory);
7. LinkedList<MemoryBlockInfo> GetFreeMemories();
8. LinkedList<MemoryBlockInfo> GetAllocatedMemories();
9. }
10. }

然后编写MemoryManager类来实现内存管理。完整代码位于MemoryManager.cs中。

该内存管理器用于管理用户需要托管的一整块内存。在后面的测试环节可以看到，如果需要托管多块内存，可以使用先托管整块，再从中申请掉一部分的方法来间接实现。因此，其初始化代码如下：

1. **namespace** MemoryManager
2. {
3. **class** MemoryManager
4. {
5. **public** MemoryManager(nuint memory, **int** size)
6. {
7. **if** (size < 0)
8. {
9. **throw** **new** ArgumentException("The size of memory must be non-negative!");
10. }
11. **if** (memory > 0 && nuint.MaxValue - memory + 1 < (nuint)size)
12. {
13. **throw** **new** ArgumentException("Memory address out of bound!");
14. }
15. **if** (size != 0)
16. {
17. freeBlocks.Add(**new** MemoryBlock(memory, size));
18. }
20. // Initialize members
21. // ...
22. }
23. }
24. }

首先，空闲分区链表使用有序链表（Utils.SortedLinkedList<T>）实现。而对于已分配分区列表，由于在释放内存时需要对内存是否是自己分配的进行查询，因此为了查询时更加方便，因此采用二叉搜索树（System.Collections.Generic.SortedDictionary<nuint, int>）实现。定义如下：

1. **namespace** MemoryManager
2. {
3. **public** **abstract** partial **class** MemoryManager : IMemoryManager
4. {
5. **protected** **readonly** SortedLinkedList<MemoryBlock> freeBlocks;
6. **private** **readonly** SortedDictionary<nuint, **int**> allocatedBlocks;
7. }
8. }

则查询空闲分区和已分配分区的代码如下：

1. **public** LinkedList<MemoryBlockInfo> GetFreeMemories()
2. {
3. var memoryInfoList = **new** LinkedList<MemoryBlockInfo>();
4. **foreach** (var memoryBlock **in** freeBlocks)
5. {
6. memoryInfoList.AddLast(**new** MemoryBlockInfo(memoryBlock.Memory, memoryBlock.Size));
7. }
8. **return** memoryInfoList;
9. }
11. **public** LinkedList<MemoryBlockInfo> GetAllocatedMemories()
12. {
13. var memoryInfoList = **new** LinkedList<MemoryBlockInfo>();
14. **foreach** (var memoryBlock **in** allocatedBlocks)
15. {
16. memoryInfoList.AddLast(**new** MemoryBlockInfo(memoryBlock.Key, memoryBlock.Value));
17. }
18. **return** memoryInfoList;
19. }

下面实现内存分配和释放算法。

对于内存分配，先使用不同的算法来分配内存块并从空闲分区列表中移除该部分内存（如果空闲分区恰好等于已经分配的内存，则移除分区，否则缩小空闲内存块），然后将内存块加入到已分配分区列表中。

内存分配过程代码如下：

1. **public** nuint? AllocateMemory(**int** size)
2. {
3. var memory = AllocateMemoryImpl(size);
4. **if** (memory.HasValue)
5. {
6. allocatedBlocks.Add(memory.Value, size);
7. }
8. **return** memory;
9. }
11. **protected** **abstract** nuint? AllocateMemoryImpl(**int** size);

其中，AllocateMemoryImpl即是不同的内存分配策略的具体实现，功能是分配内存并从空闲链表中移除。

对于内存释放策略，先判断该块内存是否是已经分配的，然后再释放该内存块。由于内存管理器维护了已分配内存块列表，因此用户只需要提供内存起始地址，而不需要提供内存块大小。具体代码如下：

1. **public** **bool** FreeMemory(nuint memory)
2. {
3. **if** (!IsAllocated(memory))
4. {
5. **return** **false**;
6. }
7. FreeMemoryBlock(**new** MemoryBlock(memory, allocatedBlocks[memory]));
8. allocatedBlocks.Remove(memory);
9. **return** **true**;
10. }

其中，IsAllocated用于判断是否是已分配的内存：

1. **private** **bool** IsAllocated(nuint memory)
2. {
3. **return** allocatedBlocks.ContainsKey(memory);
4. }

而FreeMemoryBlock用于释放内存块：

1. **protected** **virtual** **void** FreeMemoryBlock(MemoryBlock memoryBlock)
2. {
3. (var predecessor, var successor) = FindNearFreeBlocks(memoryBlock);
4. **if** (predecessor **is** not **null** || successor **is** not **null**)
5. {
6. nuint startAddress = memoryBlock.Memory;
7. **int** size = memoryBlock.Size;
8. **if** (predecessor **is** not **null**)
9. {
10. startAddress = predecessor.Value.Memory;
11. size += predecessor.Value.Size;
12. freeBlocks.Remove(predecessor);
13. }
14. **if** (successor **is** not **null**)
15. {
16. size += successor.Value.Size;
17. freeBlocks.Remove(successor);
18. }
19. memoryBlock = **new** MemoryBlock(startAddress, size);
20. }
21. freeBlocks.Add(memoryBlock);
22. }

其中，FindNearFreeBlocks函数是寻找是否存在与释放内存块相邻的空闲内存块。如果找到，则将其合并为新的内存块。FindNearFreeBlocks实现如下：

1. **protected** (LinkedListNode<MemoryBlock>? predecessor, LinkedListNode<MemoryBlock>? successor)
2. FindNearFreeBlocks(MemoryBlock memoryBlock)
3. {
4. LinkedListNode<MemoryBlock>? predecessor = **null**;
5. LinkedListNode<MemoryBlock>? successor = **null**;
7. nuint memory = memoryBlock.Memory;
8. **int** size = memoryBlock.Size;
9. **bool** isFirstBlock = IsFirstBlock(memoryBlock.Memory, memoryBlock.Size);
10. **bool** isLastBlock = IsLastBlock(memoryBlock.Memory, memoryBlock.Size);
12. var afterBlockNode = freeBlocks.LowerBound(memoryBlock);
13. LinkedListNode<MemoryBlock>? beforeBlockNode = **null**;
14. **if** (afterBlockNode **is** not **null**)
15. {
16. **if** (!isLastBlock && memory + (nuint)size == afterBlockNode.Value.Memory)
17. {
18. successor = afterBlockNode;
19. }
20. beforeBlockNode = afterBlockNode.Previous;
21. }
22. **else**
23. {
24. beforeBlockNode = freeBlocks.Last;
25. }
26. **if** (beforeBlockNode **is** not **null**)
27. {
28. **if** (!isFirstBlock && beforeBlockNode.Value.Memory + (nuint)beforeBlockNode.Value.Size == memory)
29. {
30. predecessor = beforeBlockNode;
31. }
32. }
34. **return** (predecessor, successor);
35. }

其中，IsFirstBlock和IsLastBlock是在判断是否是第一个和最后一个内存分区，如果是则不需要寻找前驱或后继内存块：

1. **private** **bool** IsFirstBlock(nuint memory, **int** size)
2. {
3. **return** size > 0 && size <= MemorySize && memory == StartAddress;
4. }
6. **private** **bool** IsLastBlock(nuint memory, **int** size)
7. {
8. **return** size > 0 && size <= MemorySize
9. && memory >= StartAddress && StartAddress + (nuint)MemorySize == memory + (nuint)size;
10. }

需要注意的是，上述FreeMemoryBlock对于大部分算法都是使用的，但是在后文将会看到，下次适配法由于需要维护上次分配内存块的位置，因此需要单独编写释放内存块函数。

下面实现各种算法的内存管理策略。

### 首次适配

对于首次适配法，只需要从头开始遍历空闲分区块，找到第一个可以容纳的分区即可。如果恰好等于申请内存数，则去掉该块；否则将内存块切割。其完整代码位于FirstFitAllocator.cs中，下面是核心算法代码：

1. **namespace** MemoryManager
2. {
3. **internal** **class** FirstFitAllocator : MemoryManager
4. {
5. **protected** **override** nuint? AllocateMemoryImpl(**int** size)
6. {
7. **if** (size <= 0)
8. {
9. **return** **null**;
10. }
12. **for** (var freeBlockNode = freeBlocks.First; freeBlockNode **is** not **null**; freeBlockNode = freeBlockNode.Next)
13. {
14. **if** (freeBlockNode.Value.Size >= size)
15. {
16. nuint ans = freeBlockNode.Value.Memory;
17. **if** (freeBlockNode.Value.Size == size)
18. {
19. freeBlocks.Remove(freeBlockNode);
20. }
21. **else**
22. {
23. freeBlockNode.ValueRef = **new** MemoryBlock(freeBlockNode.Value.Memory + (nuint)size, freeBlockNode.Value.Size - size);
24. }
25. **return** ans;
26. }
27. }
28. **return** **null**;
29. }
30. }
31. }

### 下次适配

下次适配的代码位于NextFitAllocator.cs中。下次适配相对来说比较复杂，需要维护一个上次分配的内存块的位置lastAllocatePosition。其初始化时将该位置设置为内存块首地址：

1. **namespace** MemoryManager
2. {
3. **internal** **class** NextFitAllocator : MemoryManager
4. {
5. **public** NextFitAllocator(nuint memory, **int** size) : **base**(memory, size)
6. {
7. lastAllocatePosition = freeBlocks.First;
8. }
9. **private** LinkedListNode<MemoryBlock>? lastAllocatePosition;
10. }
11. }

进行内存分配时，从记录上次的分区位置开始，找到合适的块之后记录本次分配的分区位置：

1. **namespace** MemoryManager
2. {
3. **internal** **class** NextFitAllocator : MemoryManager
4. {
5. **protected** **override** nuint? AllocateMemoryImpl(**int** size)
6. {
7. **if** (size <= 0)
8. {
9. **return** **null**;
10. }
12. **if** (lastAllocatePosition **is** **null**)
13. {
14. **return** **null**;
15. }
17. LinkedListNode<MemoryBlock> itr = lastAllocatePosition;
18. **do**
19. {
20. **if** (itr.Value.Size >= size)
21. {
22. nuint ans = itr.Value.Memory;
23. **if** (itr.Value.Size == size)
24. {
25. lastAllocatePosition = freeBlocks.Count == 1 ? **null** :
26. itr.Next ?? freeBlocks.First;
27. freeBlocks.Remove(itr);
28. }
29. **else**
30. {
31. itr.ValueRef = **new** MemoryBlock(itr.Value.Memory + (nuint)size, itr.Value.Size - size);
32. lastAllocatePosition = itr;
33. }
34. **return** ans;
35. }
37. itr = itr.Next ?? freeBlocks.First
38. ?? **throw** **new** Exception("This code shouldn't be reachable.");    // lastAllocatePosition Means freeBlocks isn't empty
40. } **while** (!ReferenceEquals(itr, lastAllocatePosition));
42. **return** **null**;
43. }
44. }
45. }

对于下次适配，释放内存块的代码需要重新编写。这是因为释放内存块时可能存在合并内存块的操作，这时如果lastAllocatePosition指向的内存块被合并，则需要更新lastAllocatePosition的值，指向新的内存块。代码如下：

1. **namespace** MemoryManager
2. {
3. **internal** **class** NextFitAllocator : MemoryManager
4. {
5. **protected** **override** **void** FreeMemoryBlock(MemoryBlock memoryBlock)
6. {
7. (var predecessor, var successor) = FindNearFreeBlocks(memoryBlock);
9. **if** (predecessor **is** not **null** || successor **is** not **null**)
10. {
11. nuint startAddress = memoryBlock.Memory;
12. **int** size = memoryBlock.Size;
13. **bool** updateLastPos = **false**;
15. **if** (predecessor **is** not **null**)
16. {
17. startAddress = predecessor.Value.Memory;
18. size += predecessor.Value.Size;
20. **if** (ReferenceEquals(predecessor, lastAllocatePosition))
21. {
22. updateLastPos = **true**;
23. }
25. freeBlocks.Remove(predecessor);
26. }
28. **if** (successor **is** not **null**)
29. {
30. size += successor.Value.Size;
32. **if** (ReferenceEquals(successor, lastAllocatePosition))
33. {
34. updateLastPos = **true**;
35. }
37. freeBlocks.Remove(successor);
38. }
40. **if** (updateLastPos)
41. {
42. lastAllocatePosition = freeBlocks.AddAndGetNode(**new** MemoryBlock(startAddress, size));
43. }
44. **else**
45. {
46. freeBlocks.Add(**new** MemoryBlock(startAddress, size));
47. }
48. }
49. **else**
50. {
51. freeBlocks.Add(memoryBlock);
52. lastAllocatePosition ??= freeBlocks.First;
53. }
54. }
55. }
56. }

### 最佳适配

最佳适配比较简单，只需要找到最接近且能够容纳下申请内存大小的块即可。完整代码位于BestFitAllocator.cs中。下面是核心算法代码：

1. **namespace** MemoryManager
2. {
3. **internal** **class** BestFitAllocator : MemoryManager
4. {
5. **protected** **override** nuint? AllocateMemoryImpl(**int** size)
6. {
7. **if** (size <= 0)
8. {
9. **return** **null**;
10. }
12. LinkedListNode<MemoryBlock>? result = **null**;
13. **for** (var freeBlockNode = freeBlocks.First; freeBlockNode **is** not **null**; freeBlockNode = freeBlockNode.Next)
14. {
15. **int** blockSize = freeBlockNode.Value.Size;
16. **if** (blockSize >= size && (result **is** **null** || blockSize < result.Value.Size))
17. {
18. result = freeBlockNode;
19. }
20. }
22. **if** (result **is** not **null**)
23. {
24. nuint ans = result.Value.Memory;
25. **if** (result.Value.Size == size)
26. {
27. freeBlocks.Remove(result);
28. }
29. **else**
30. {
31. result.ValueRef = **new** MemoryBlock(result.Value.Memory + (nuint)size, result.Value.Size - size);
32. }
33. **return** ans;
34. }
35. **return** **null**;
36. }
37. }
38. }

### 最差适配

最差适配也比较简单，只需要找到最大的内存块并确保其能容纳申请的内存大小即可。完整代码位于WorstFitAllocator.cs中。下面是核心算法代码：

1. **namespace** MemoryManager
2. {
3. **internal** **class** WorstFitAllocator : MemoryManager
4. {
5. **protected** **override** nuint? AllocateMemoryImpl(**int** size)
6. {
7. **if** (size <= 0)
8. {
9. **return** **null**;
10. }
12. LinkedListNode<MemoryBlock>? result = **null**;
13. **for** (var freeBlockNode = freeBlocks.First; freeBlockNode **is** not **null**; freeBlockNode = freeBlockNode.Next)
14. {
15. **int** blockSize = freeBlockNode.Value.Size;
16. **if** (blockSize >= size && (result **is** **null** || blockSize > result.Value.Size))
17. {
18. result = freeBlockNode;
19. }
20. }
22. **if** (result **is** not **null**)
23. {
24. nuint ans = result.Value.Memory;
25. **if** (result.Value.Size == size)
26. {
27. freeBlocks.Remove(result);
28. }
29. **else**
30. {
31. result.ValueRef = **new** MemoryBlock(result.Value.Memory + (nuint)size, result.Value.Size - size);
32. }
33. **return** ans;
34. }
35. **return** **null**;
36. }
37. }
38. }

## 用户指定分配策略

最后，向用户暴露统一的接口，让用户可以指定托管的内存首地址和大小，以及分配策略。完整代码位于MemoryManagerFactory.cs中。用户可以通过MemoryManagerFactory. CreateMemoryManager方法来获得内存管理器。

1. **namespace** MemoryManager
2. {
3. **public** **static** **class** MemoryManagerFactory
4. {
5. **public** **enum** AllocationStrategy
6. {
7. Default = 0,
8. FirstFit = 1,
9. NextFit = 2,
10. BestFit = 3,
11. WorstFit = 4,
12. }
14. **public** **static** MemoryManager CreateMemoryManager(nuint memory, **int** size, AllocationStrategy allocationStrategy = AllocationStrategy.Default)
15. {
16. **switch** (allocationStrategy)
17. {
18. **case** AllocationStrategy.Default:
19. **case** AllocationStrategy.FirstFit:
20. **return** **new** FirstFitAllocator(memory, size);
21. **case** AllocationStrategy.NextFit:
22. **return** **new** NextFitAllocator(memory, size);
23. **case** AllocationStrategy.BestFit:
24. **return** **new** BestFitAllocator(memory, size);
25. **case** AllocationStrategy.WorstFit:
26. **return** **new** WorstFitAllocator(memory, size);
27. }
28. **throw** **new** ArgumentException("Invalid allocation strategy!");
29. }
30. }
31. }

# 样例测试

编写适当样例对四种管理策略进行测试。

测试代码位于UnitTest项目中。由于代码很长，下面只举最差适配法的测试作为例子，其他的内存分配测试均类似。最差适配法的测试代码位于UnitTest/ WorstFitAllocatorTest.cs中。

首先对基本的分配、释放、获取空闲列表、是否会造成内存泄漏等功能进行测试，分别位于TestAlloction、TestFree、TestFailedAllocation、TestFailedFree、TestFreeList、TestMemoryLeak方法中。

然后使用具体的测试用例对其实现的正确性进行测试。首先让其管理起始地址为128，大小为1024的内存块：

1. var manager = MemoryManagerFactory.CreateMemoryManager(128u, 1024, MemoryManagerFactory.AllocationStrategy.WorstFit);

然后构造大小分别为32、32、256、16、64、16、128的七个空闲分区。方法是先申请内存将大小为1024的内存块分割成大小为32、64、32、128、256、16、16、64、64、16、16、64、128、128的内存块，再每隔一个内存块便释放一个内存块，即可实现多个空闲分区：

1. **int**[] sizes = { 32, 64, 32, 128, 256, 16, 16, 64, 64, 16, 16, 64, 128, 128 };
2. nuint[] memories = **new** nuint[sizes.Length];
3. **for** (**int** i = 0; i < sizes.Length; ++i)
4. {
5. var tmp = manager.AllocateMemory(sizes[i]);
6. Assert.IsNotNull(tmp);
7. memories[i] = tmp.Value;
8. }
10. **for** (**int** i = 0; i < sizes.Length; i += 2)
11. {
12. Assert.IsTrue(manager.FreeMemory(memories[i]));
13. }

然后依次申请大小为96、16、256、128、0、192、32的内存分区：

1. var memory1Val = manager.AllocateMemory(96) ?? 0;
2. // |32| 64 |32| 128 96 |160| 16 |16| 64 |64| 16 |16| 64 |128| 128
3. var memory2Val = manager.AllocateMemory(16) ?? 0;
4. // |32| 64 |32| 128 96 16 |144| 16 |16| 64 |64| 16 |16| 64 |128| 128
5. var memory3Val = manager.AllocateMemory(256) ?? 0;
6. Assert.IsTrue(memory3Val == 0);
7. // |32| 64 |32| 128 96 16 |144| 16 |16| 64 |64| 16 |16| 64 |128| 128
8. var memory4Val = manager.AllocateMemory(128) ?? 0;
9. // |32| 64 |32| 128 96 16 128 |16| 16 |16| 64 |64| 16 |16| 64 |128| 128
10. var memory5Val = manager.AllocateMemory(0) ?? 0;
11. // |32| 64 |32| 128 96 16 128 |16| 16 |16| 64 |64| 16 |16| 64 |128| 128
12. var memory6Val = manager.AllocateMemory(192) ?? 0;
13. Assert.IsTrue(memory6Val == 0);
14. // |32| 64 |32| 128 96 16 128 |16| 16 |16| 64 |64| 16 |16| 64 |128| 128
15. var memory7Val = manager.AllocateMemory(32) ?? 0;
16. // |32| 64 |32| 128 96 16 128 |16| 16 |16| 64 |64| 16 |16| 64 32 |96| 128

绿色注释记录了每次操作后内存的排布。其中，由 || 所包围的数字表示空闲分区大小，没有包围的表示已分配分区的大小。

可以看到，由于最初空闲分区大小分别为32、32、256、16、64、16、128，根据最差适配，不难得到，申请256、0、192的操作都将失败，剩余空闲分区为32、32、16、16、64、16、96。

然后释放掉之前申请的大小为128的内存块：

1. manager.FreeMemory(memory4Val);
2. // |32| 64 |32| 128 96 16 |144| 16 |16| 64 |64| 16 |16| 64 32 |96| 128

该内存块将与剩余的大小为16的空余内存块合并成大小为144的内存块（即起初的大小为256的内存块剩余的）。得到的空闲内存块应为：32、32、144、16、64、16、96。使用代码进行测试空闲列表是否正确：

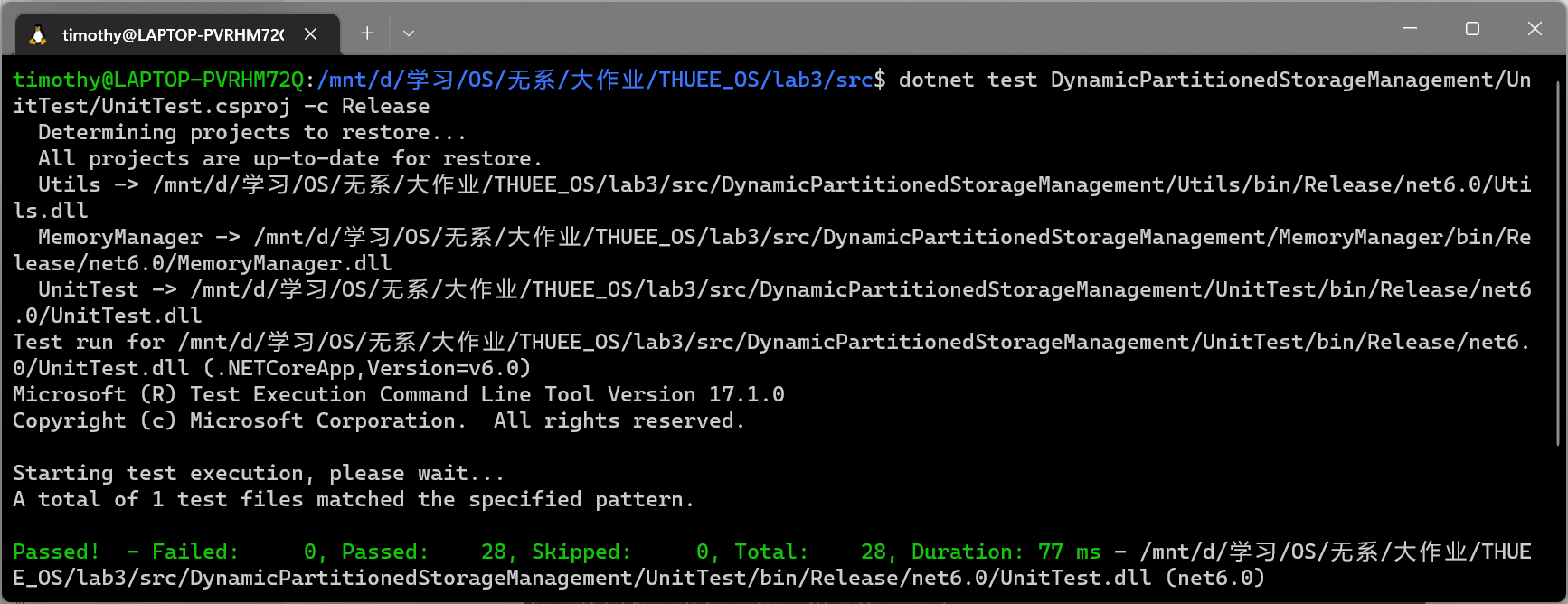
1. var freeList = manager.GetFreeMemories();
2. var freeArr = **new** **int**[] { 32, 32, 144, 16, 64, 16, 96 };
3. var q = **new** Queue<**int**>(freeArr);
4. Assert.IsTrue(freeList.Count == freeArr.Length);
5. **foreach** (var memoryBlockInfo **in** freeList)
6. {
7. Assert.IsTrue(memoryBlockInfo.Size == q.Dequeue());
8. }

以上便是最差适配的测试样例。其他算法的测试样例均相似，或是更加复杂，例如测试释放空闲块时在边界处能否正常合并空闲内存块，等等。

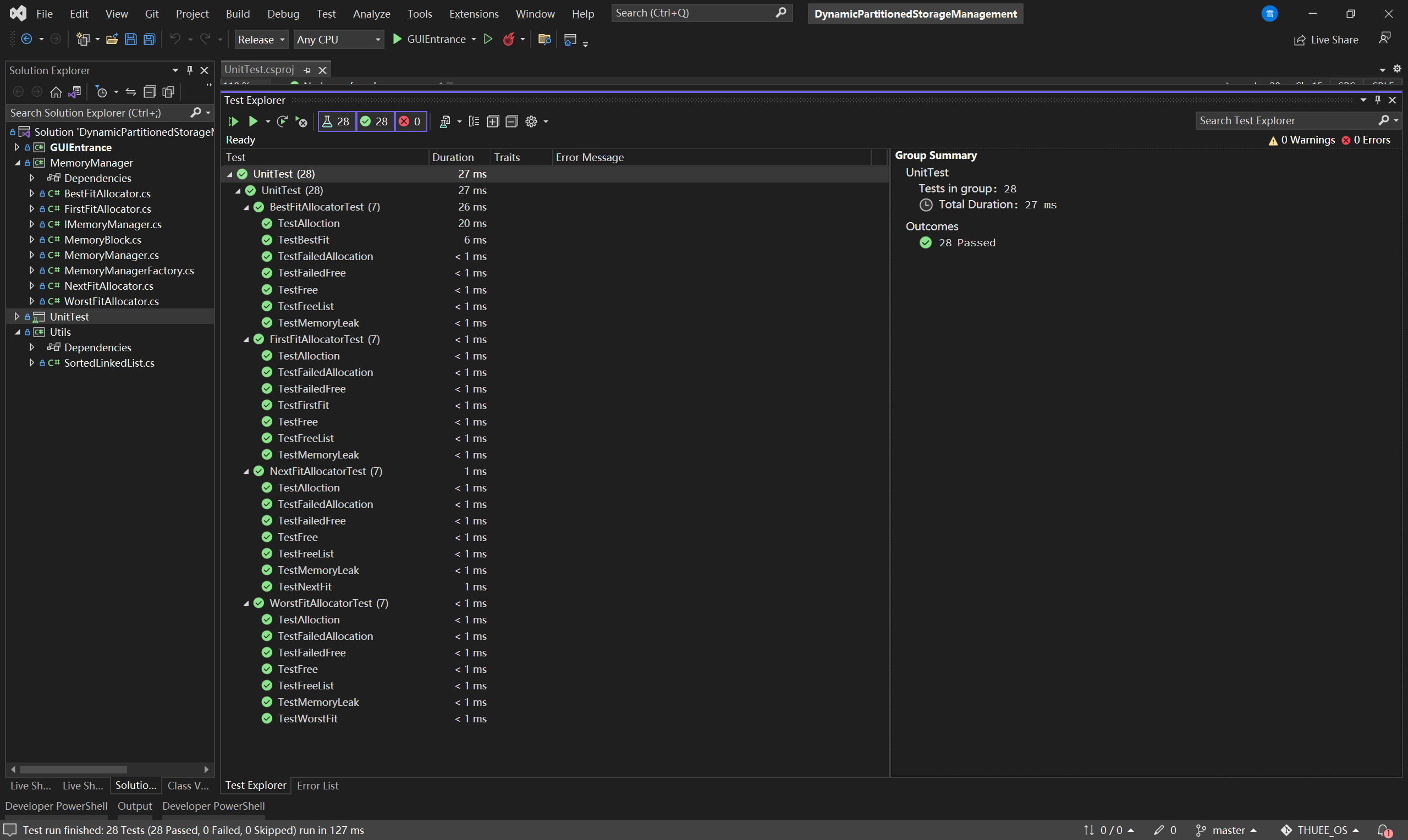
下面运行测试。在src目录下执行：

1. $ dotnet test DynamicPartitionedStorageManagement/UnitTest/UnitTest.csproj -c Release

可以看到，所有28个测试均在82ms内运行成功：

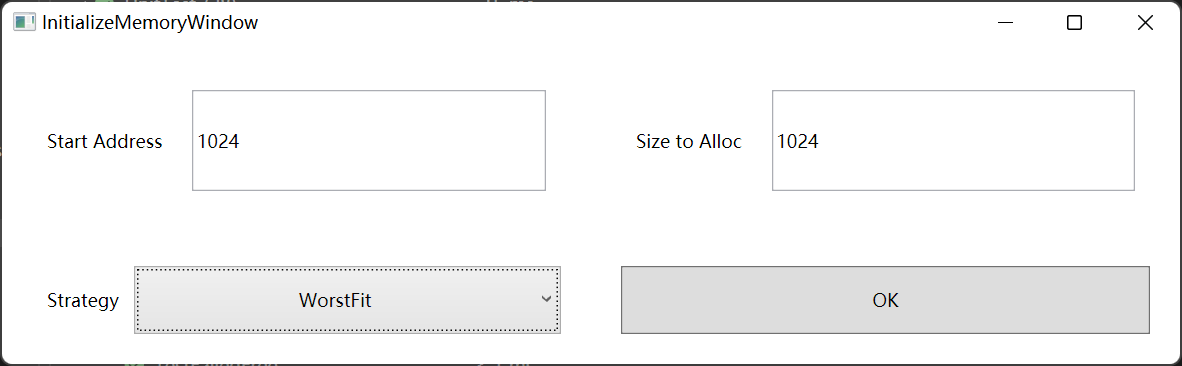


在Visual Studio可视化界面运行测试也可以看到所有测试运行成功：

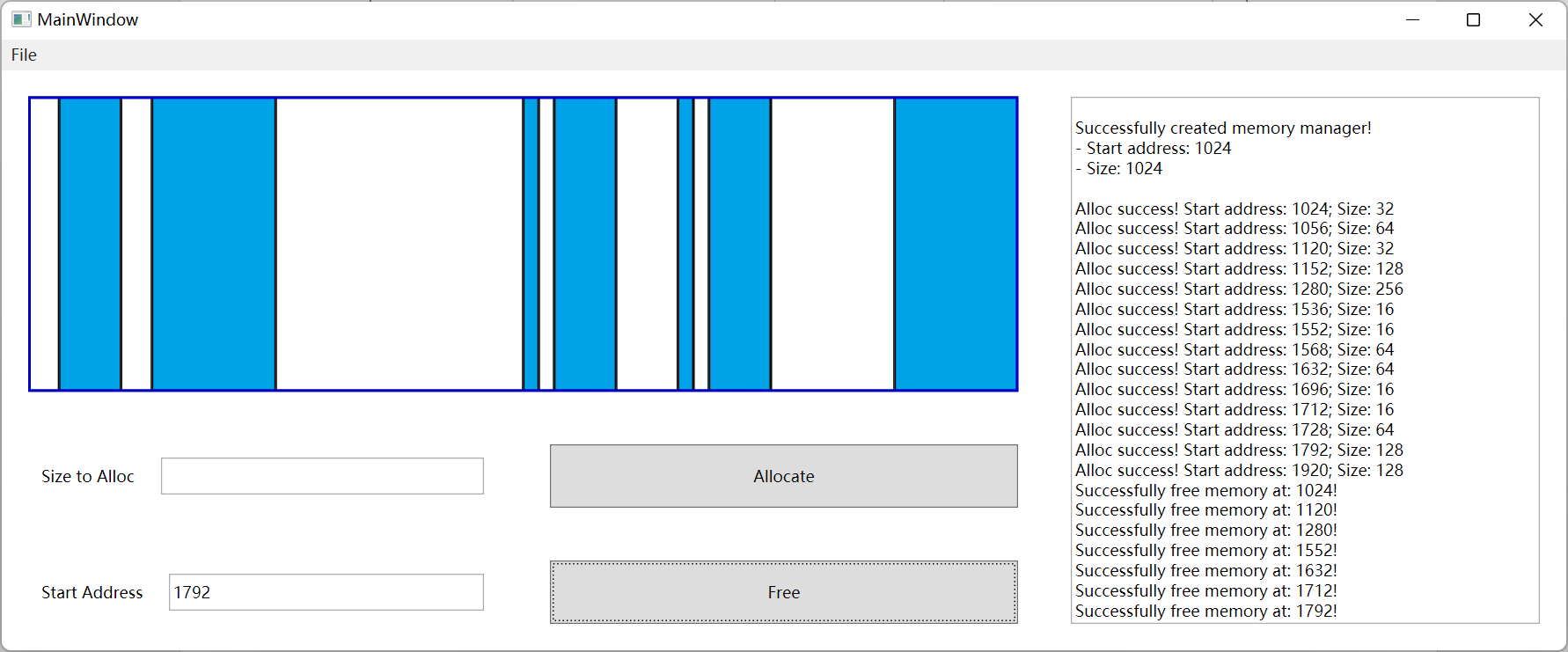


# 内存使用可视化

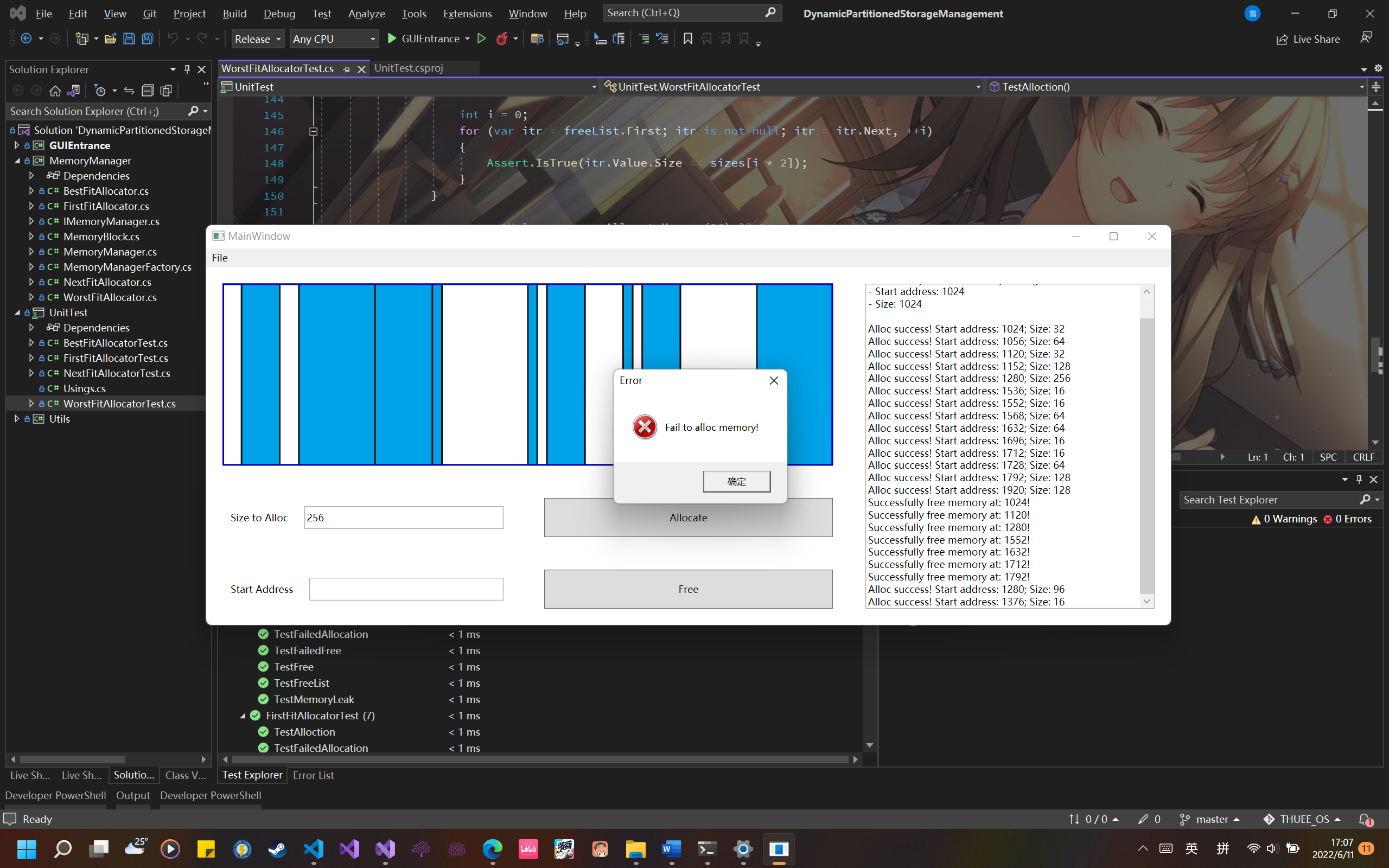
下面将使用图形界面观看内存分配情况。图形界面代码位于GUIEntrance目录内。可执行文件已经随本报告附在GUI目录内。运行GUIEntrance.exe，让其管理从1024开始的大小为1024的内存块，使用最差适配：



重新进行之前样例的测试，先分配一些空闲分区：



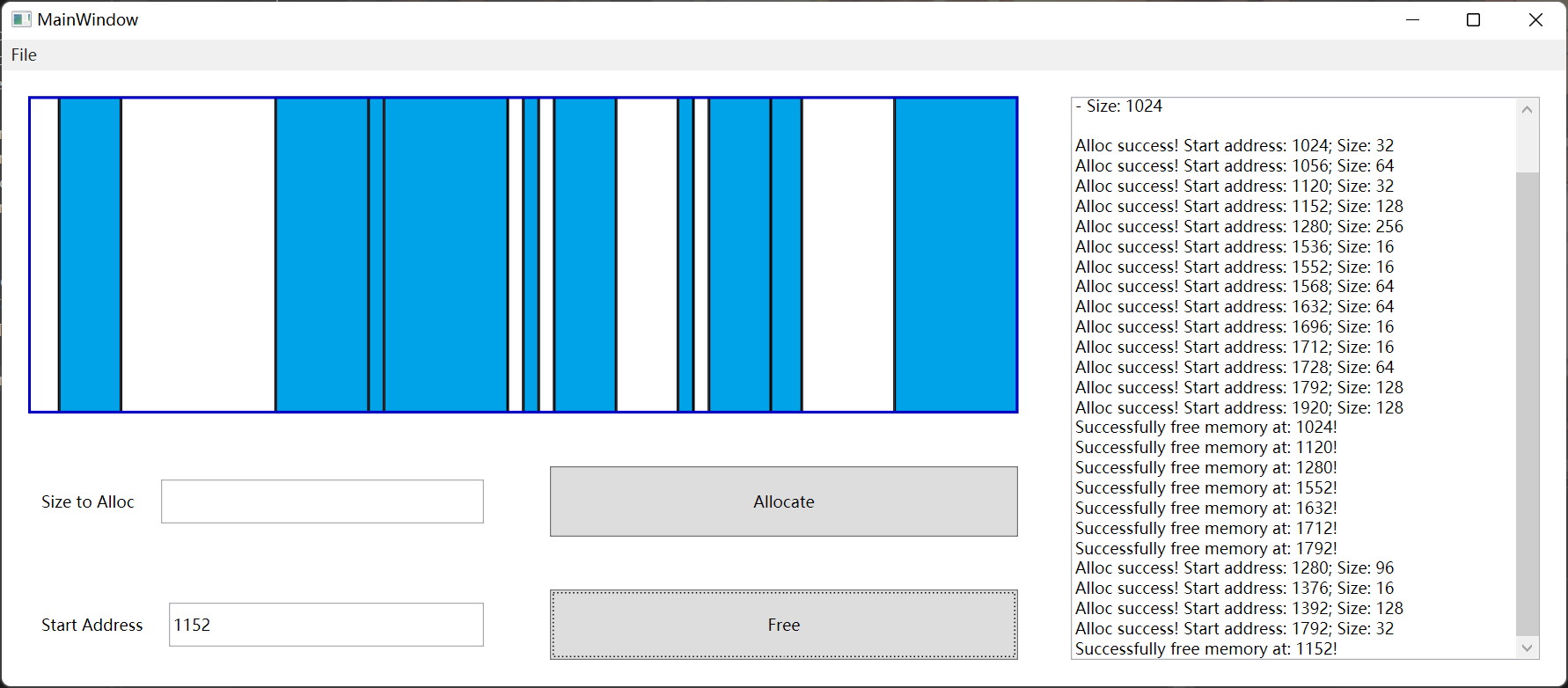
其中，白色代表空闲分区，蓝色代表已经分配的分区。然后对之前测试样例要申请的内存依次申请，并根据左方的可视化结果和右方的log信息。其中，在中途申请内存失败时会弹出消息框提示失败：



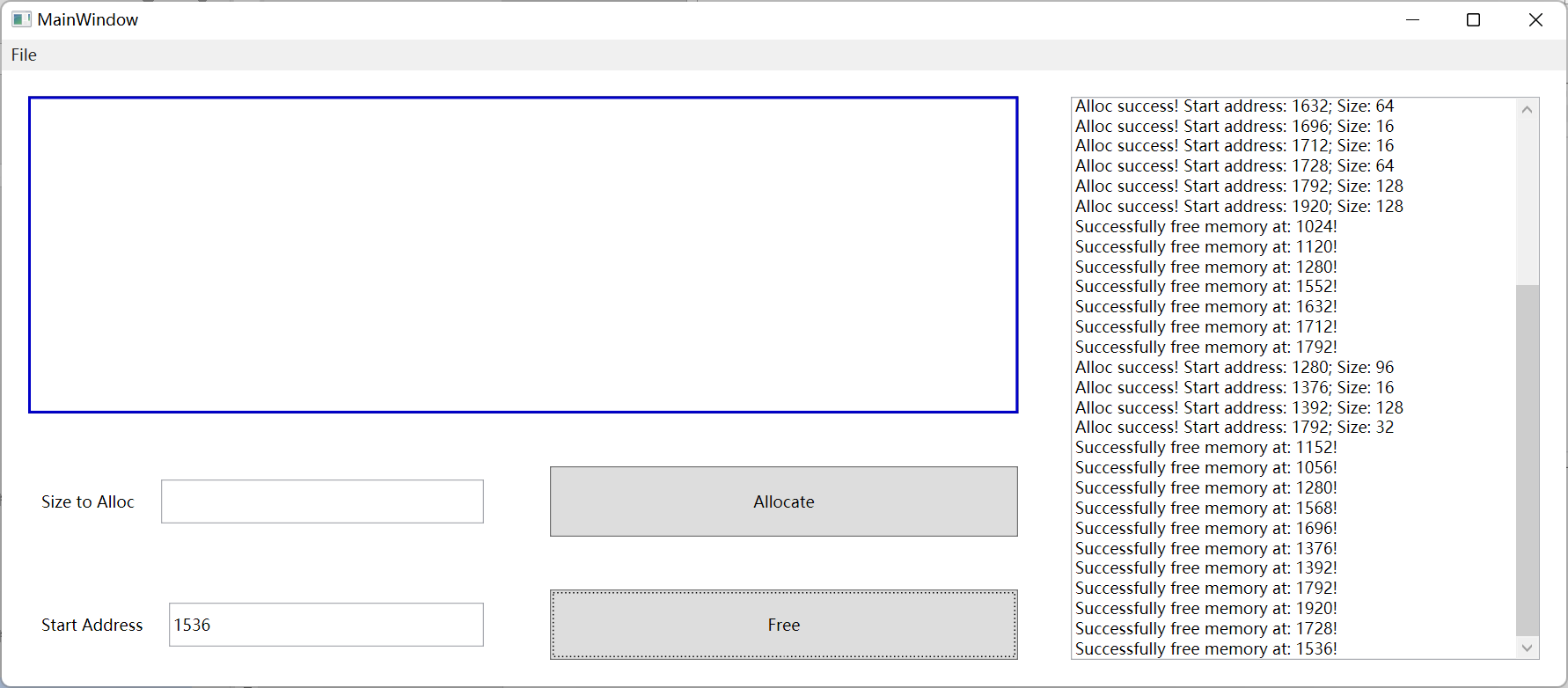
申请结束后，可以看到最终结果符合预期：



再释放位于1152的内存，可以看到空闲内存块成功合并：



释放所有内存后，空闲分区全部合并到一起：



在菜单中选择File目录，进入New目录项，或是直接按Ctrl+N快捷键，可以重新生成内存管理器。

可见，内存使用可视化展示功能是比较完善的。

# 实验心得

经过我的努力，我相对顺利地完成了本次实验。综合来看，我在这次实验中还是有很大收获的。

在本次实验中，我自己动手实现了四种动态内存分区管理算法，对这些算法都有了更深的理解，了解了操作系统如何对内存进行管理，同时也帮助我巩固了课上所学的知识。而且在编写代码的过程中，我也因为疏忽出现了一些bug，这些都在之后的测试中检查了出来，然后经过多次地断点调试等，找出了问题所在，并进行了修改。这也让我体会到相对完整地、全方位的测试样例的重要性。同时在debug的过程中，我也体会到了解决问题的快乐。此外，在这次实验中，我也熟悉了C# 语言、.NET类库、WPF中MVVM模式的使用，都让我受益匪浅。

最后，感谢老师在课上的讲授和助教的付出，没有老师和助教的讲解，我也难以顺利完成本次实验，在此对老师和助教表示衷心的感谢。

# 思考题

Q：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？

A：使用位图和空闲链表进行存储管理各有优劣。

1. 使用空闲链表存储，是把空闲分区连成链表进行存储。
   1. 当空闲分区数比较少的时候，空闲链表的结点数不多，占用的空间更小；
   2. 分配内存查找合适的内存分区时，需要对链表遍历查找，这时相比于位图的对所有内存地址从头进行遍历以查找连续多个0来说，链表会更快；
   3. 如果要直接定位到某一个内存地址，则需要沿链表依次遍历，速度较位图更慢；
   4. 当空闲分区数较多较杂的时候，链表项会很多，占用空间也会很大。
2. 使用位图存储，是把内存分割成一个个小的单元，使用一个个二进制位来表示内存是否分配。
   1. 首先，其占用空间是固定的。内存单元是否分配都要在位图中占用空间，因此位图所占内存可以静态分配；
   2. 由于占用内存固定，因此在空闲分区数较少的时候，位图占用更多的内存；空闲分区数很多的时候，位图相比之下更节省内存；
   3. 位图在查找某一个特定的地址的时候，由于只需要进行地址偏移操作，因此在定位内存的速度相对于空闲链表更快；
   4. 由于内存需要切割成一个个单元，分配需要以内存单元为单位，因此当申请的内存大小不是单元的整数倍的时候，会产生内存碎片，造成内存的浪费；
   5. 对于分配内存来说，由于查找一块适当大小的内存，需要进行连续多个0的查找，此算法相比于链表更复杂，也更加耗时。
3. 如果使用位图进行存储管理，则相比于链表来说，需要注意权衡切割的内存单元的大小。如果内存单元过大，则会产生较多的内存碎片；如果内存单元过小，则会增大位图的占用空间，也会增加位图的遍历搜索时间。