Лекция 5

Управление памятью в .NET

- Основные принципы управления памятью
- Выделение ресурсов в .NET
- Завершение (finalization)
- Слабые ссылки (weak references)
- Поколения (generations)
- Класс System.GC
- Сборка мусора в многопоточных приложениях

Основные этапы работы с памятью

- Выделение памяти под ресурс
- Инициализация памяти
- Использование памяти
- Очистка памяти/освобождение ресурса
- Повторное использование памяти

Проблемы управления памятью

- Память не бесконечна!
- Явное управление памятью отвлекает программиста от его основных задач
- Ошибки при работе с памятью возникают редко и потому труднонаходимы
- Проблема освобождения ресурса - система не знает, как освободить сетевой ресурс или снять блокировку в базе данных

Проблемы управления памятью (окончание)

- Необходимо различать уничтожение памяти (уничтожение объекта/уничтожение путей доступа) и утилизацию памяти (сборка мусора)
- Проблема отслеживания различных путей доступа к структуре (различные указатели на одно и то же место, передача формальным параметром) - не во всех языках это возможно
- Два полюса проблемы: **висячие ссылки** и **мусор**

Влияние управления памятью на языки программирования

Разработка практически всех языков программирования ориентирована на ту или иную методику управления памятью:

- Фортран: запрет рекурсивных вызовов подпрограмм $\Rightarrow$ не нужен стек точек возврата $\Rightarrow$ статическое управление памятью
- C: исключительное использование явного освобождения памяти
- Java, платформа .NET: сборка мусора

Примеры различных подходов

- Алгол 60: "первый идентификатор", с которым ассоциируется структура данных;
  - "первый идентификатор" разрушается при выходе из блока, так же, как и ассоциации, возникающие при вызове подпрограмм и возврате из них; после выхода из блока память может быть утилизирована.
- LISP: допускается множество ссылок на любой список, включая указатели на подсписки и т.п.
  - при недостатке памяти вызывается "сборщик мусора", помечающий все активные элементы и затем уничтожающий все остальные

Неявное управление памятью

- Тенденция развития современных языков программирования: предоставить программисту только неявные средства управления памятью, через использование возможностей языка.
  - Сходство с "проблемой языков высокого уровня"
  - Все равно программист не станет управлять временными переменными!
  - Трудности совмещения двух механизмов управления памятью (система/программист)

Фазы управления памятью

- Начальное распределение памяти
  - методы учета свободной памяти
- Утилизация памяти
  - перемещение указателя стека
  - сборка мусора
- Повторное использование и уплотнение
  - память либо сразу пригодна к повторному использованию, либо должна быть уплотнена для создания больших блоков свободной памяти

Статическое управление памятью

- Простейший способ распределения памяти
- Производится во время трансляции и не меняется во время исполнения
- Никакого управления памятью! (эффективно)
- Не подходит для рекурсивных вызовов, для структур данных, зависящих от внешней информации (динамические массивы) и т.п.
- Тем не менее, вполне достаточен для некоторых языков (Фортран, Кобол)

Стековое управление памятью

- Простейший метод распределения памяти времени выполнения
- Освобождение памяти в обратном порядке
- Задачи утилизации, уплотнения и повторного использования становятся тривиальными
- Используется для точек возврата из подпрограмм и локальных сред
- Подходит для языков со строго вложенной структурой (Pascal, Algol 60)

Управление кучей

- Куча - это блок памяти, части которого выделяются и освобождаются способом, не подчиняющимся какой-либо структуре
- Серьезные проблемы выделения, утилизации, уплотнения и повторного использования памяти
- Куча требуется в тех языках, где выделение и освобождение памяти требуется в произвольных местах программы

Начальное распределение и повторное использование

- Обычное решение - список свободного пространства. Например, в C-runtime heap для выделения памяти надо пройти по списку, найти свободный блок, разбить его на части и заменить указатели (метод первого подходящего, метод наиболее подходящего)
- Можно было бы воспользоваться механизмом, аналогичным стеку, но есть проблемы с повторным использованием

Различные подходы к утилизации памяти

- **Явный захват/возврат памяти**
  - Механизм malloc/free.
  - Проблема мусора и висячих ссылок
- **Счетчики ссылок**
  - В каждом элементе заводится место для счетчика ссылок на элемент.
  - Проблема циклического списка.
  - Стоимость работы (присваивание ссылок). Использование только для крупных структур.
- **Сборка мусора**
  - большая гибкость, но сложно определить, когда структура становится мусором

Сборка мусора

- Основная идея: допускается мусор, чтобы избежать висячих ссылок
- При исчерпании памяти мы выявляем мусор и возвращаем их в список свободного пространства
- Сбор мусора может быть дорогостоящим. Основной метод - "mark-and-compact" вместе со списком свободного пространства
- Затраты на сборку мусора обратно пропорциональны объему утилизируемой памяти! (т.к. основные затраты на маркировку)

Алгоритм выделения памяти в .NET

- Все ресурсы выделяются из управляемой кучи
- Стековый механизм выделения памяти (хранится указатель, назовем его NextObjPtr, который исходно указывает на начало кучи)
- При создании объекта к NextObjPtr просто прибавляется размер созданного объекта
- Если для создания объекта не хватает памяти, то производится сборка мусора

Алгоритм сборки мусора (нулевое приближение)

- Активные элементы определяются просмотром от корневых объектов (глобальные, статические, локальные объекты и регистры процессора). Список корневых объектов хранится в JIT-компиляторе и предоставляется сборщику мусора
- Достижимые объекты обходятся рекурсивно и помечаются как активные
- Активные элементы сдвигаются "вниз" путем копирования памяти
- Т.к. все указатели могли измениться, сборщик мусора исправляет все ссылки

Как все это работает?

```
class Application {
    public static int Main(String[] args) {

      // ArrayList object created in heap, myArray is now a root
      ArrayList myArray = new ArrayList();

      // Create 10000 objects in the heap
      for (int x = 0; x < 10000; x++) {
         myArray.Add(new Object());    // Object object created in heap
      }

      // Right now, myArray is a root (on the thread's stack). So,
      // myArray is reachable and the 10000 objects it points to are also
      // reachable.
      Console.WriteLine(a.Length);

      // After the last reference to myArray in the code, myArray is not a root.
      // Note that the method doesn't have to return, the JIT compiler knows
      // to make myArray not a root after the last reference to it in the code.

      // Since myArray is not a root, all 10001 objects are not reachable
      // and are considered garbage.  However, the objects are not
      // collected until a GC is performed.
   }
}
```

Проблемы сборки мусора

- Т.к. ссылки изменяются, на время работы сборщика мусора вся полезная работа прекращается - существенная потеря производительности!
- Невозможно использовать сборщик мусора для unmanaged C++ из-за приведения указателей от одного типа к другому
- В остальных языках для определения типов и взаимозависимостей используются метаданные

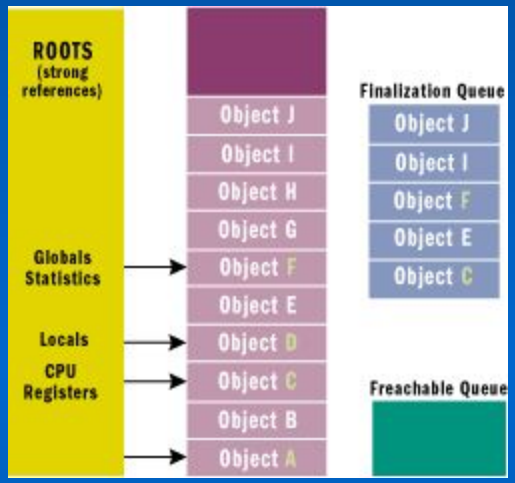
Завершители

- Позволяют выполнить необходимые операции перед тем, как объект будет освобожден сборщиком мусора
- **Завершители не являются деструкторами!** Например, при сборке объекта не вызывается завершитель базового класса - если это желательно, надо явно писать base.Finalize()
- Тем не менее, в C# можно писать ~MyObject() и компилятор сгенерирует соответствующий код в методе protected override void Finalize(), включая вызов base.Finalize() в конце метода

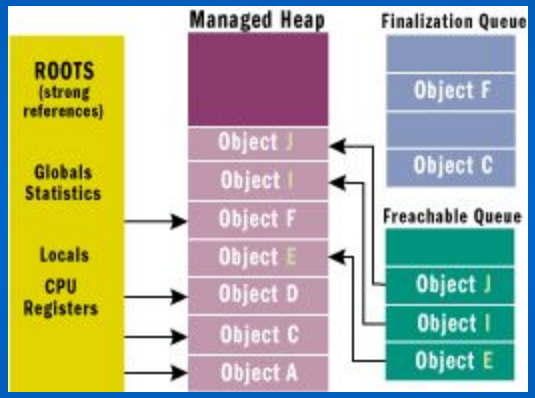
Проблемы завершителей

- Объекты с завершителями дольше создаются, требуют дополнительных затрат ресурсов и дольше остаются в системе (две сборки мусора)
- Завершение выполняется отдельным потоком и существенно понижает производительность (например, при освобождении массива)
- Не существует гарантированного порядка выполнения завершителей (obj1/obj2? obj2/obj1?)
- Поэтому надо создавать завершители только тогда, когда это абсолютно необходимо

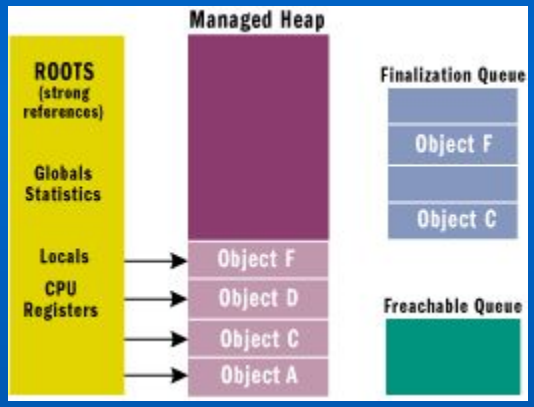
Finalization Internals?



Память до сборки мусора



Память после первой сборки мусора



Память после второй сборки мусора

Воскрешение объектов

```
class Application {
  static public Object ObjHolder; // defaults to null
  ...
}

public class BaseObj {
  protected override void Finalize() {
    Application.ObjHolder = this;
  }
```

Поведение воскрешенных объектов

- Объект уже был завершен; поведение малопредсказуемо
- Воскрешаются также все объекты, на которые указывал BaseObj
- При повторном освобождении метод Finalize уже не будет вызван - для этого надо вызвать метод `GC.ReRegisterForFinalize(this);` (однократно, иначе Finalize будет вызван несколько раз!)

Форсированная очистка объектов

- Методы Close/Dispose (Close для объектов, которые могут быть переиспользованы, Dispose для "одноразовых" объектов)
- Пример: System.IO.FileStream, использующий промежуточный буфер. Метод Finalize записывает данные и закрывает файл
- Можно явно вызвать метод Close (например, для того, чтобы дать доступ к этому файлу другим объектам)
- Но что тогда делает метод Finalize?

Метод SuppressFinalize

```
public class FileStream : Stream {

    public override void Close() {
        // Clean up this object: flush data and close file
        •••
        // There is no reason to Finalize this object now
        GC.SuppressFinalize(this);
    }

    protected override void Finalize() {
       Close(); // Clean up this object: flush data and close file
    }

    // Rest of FileStream methods go here
    •••
}
```

Завершители нельзя уравновешивать!

Повторные вызовы ReRegisterForFinalize нельзя уравновесить многократными SuppressFinalize, т.к. после вызова Finalize соответствующий флаг сбрасывается в исходное положение:

```
void method() {
    // The MyObj type has a Finalize method defined for it
    // Creating a MyObj places a reference to obj on the finalization table.
    MyObj obj = new MyObj();

    // Append another 2 references for obj onto the finalization table.
    GC.ReRegisterForFinalize(obj);    GC.ReRegisterForFinalize(obj);
    // There are now 3 references to obj on the finalization table.

    // Have the system ignore the first call to this object's Finalize method.
    GC.SuppressFinalize(obj);

    // Have the system ignore the first call to this object's Finalize method.
    GC.SuppressFinalize(obj);   // In effect, this line does absolutely nothing!

    obj = null;   // Remove the strong reference to the object.

    // Force the GC to collect the object.
    GC.Collect();

    // The first call to obj's Finalize method will be discarded but
    // two calls to Finalize are still performed.
}
```

Связанная проблема

```
FileStream fs = new FileStream("C:\\SomeFile.txt", FileMode.Open,
  FileAccess.Write, FileShare.Read)
StreamWriter sw = new StreamWriter (fs);
sw.Write ("Hello file");
sw.Close(); // correct usage
            // NOTE: StreamWriter.Close() closes the FileStream
            // No need to call fs.Close() ourselves
```

А как же метод Finalize? Как мы знаем, в нем порядок выполнения не определен!

Решение: класс StreamWriter вообще не имеет метода Finalize. Т.о., отсутствие закрытия файла означает потерю данных. Будем надеяться, что вы это заметите!

Слабые ссылки

- Ссылка на объект из корневого объекта называется сильной ссылкой (strong reference)
- Существует механизм слабых ссылок (weak references), который не мешает сборщику мусора освободить объект, но сохраняет ссылку на него
- Обычная схема использования:

```
WeakReference wr = new WeakReference(obj);
obj = null;
```

Пример использования слабых ссылок

```
Void Method() {
   Object o = new Object();    // Creates a strong reference to the
object.

   // Create a strong reference to a short WeakReference object.
   // The WeakReference object tracks the Object.
   WeakReference wr = new WeakReference(o);

   o = null;    // Remove the strong reference to the object

   o = wr.Target;
   if (o == null) {
      // A GC occurred and Object was reclaimed.
   } else {
      // a GC did not occur and we can successfully access the Object
      // using o
   }
}
```

Зачем нужны слабые ссылки?

- Для оптимизации: некоторые объекты легко создаются, но требуют много памяти (образ структуры каталогов на диске).
- Если пользователь начал пользоваться какой-то другой функциональностью и ему требуется память, то память можно освободить.
- Если же нагрузка на память невелика, то слабая ссылка доживет до возвращения пользователя и сможет быть преобразована обратно в сильную ссылку.

Два типа слабых ссылок

- Конструктор слабых ссылок имеет параметр trackRessurection (по умолчанию false)
- Слабые ссылки, не отслеживающие воскрешение, называются короткими слабыми ссылками;
- Отслеживающие воскрешение - длинные слабые ссылки (не рекомендуются к использованию)
- Если у объекта нет метода Finalize, то оба типа слабых ссылок ведут себя одинаково

Алгоритм сборки мусора (первое приближение)

- GC строит граф достижимых объектов (см. выше)
- GC просматривает таблицу коротких слабых ссылок; если в ней находится указатель на объект, не являющийся частью графа, то этот указатель обнуляется
- GC просматривает таблицу завершителей; если в ней есть указатель на объект, не являющийся частью графа, то он перемещается из очереди завершителей в очередь freacheable (тем самым объект вновь стал достижимым)
- GC просматривает таблицу длинных слабых ссылок; если если в ней находится указатель на объект, не являющийся частью графа, то указатель = null
- GC проводит сжатие памяти; со временем проснется нить, выполняющая метод Finalize; все объекты, прошедшие Finalize, ждут следующей сборки мусора

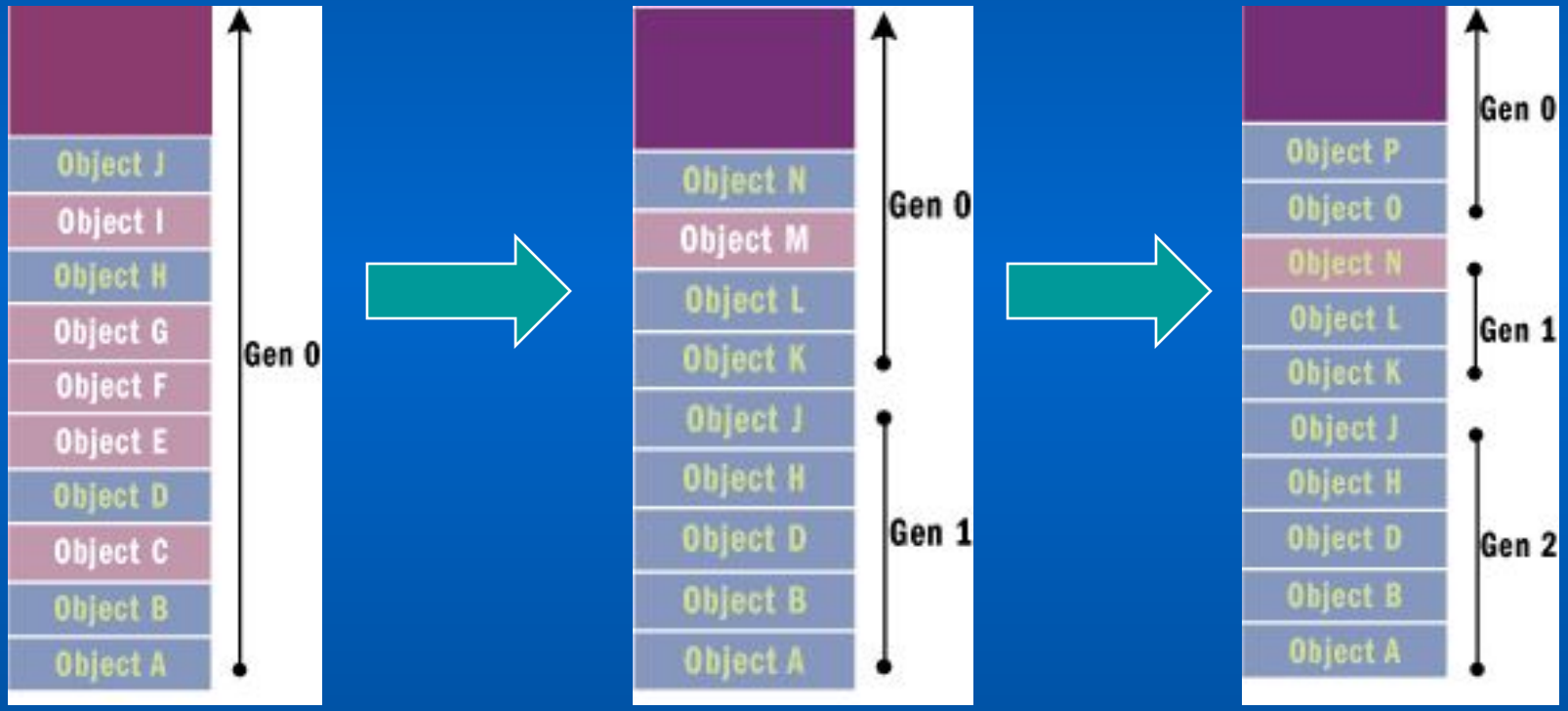
Поколения

Исследования показали, что для большинства программ верны следующие предположения:

- Чем моложе объект, тем меньше его ожидаемое время жизни
- Чем старше объект, тем больше его ожидаемое время жизни
- Молодые объекты зачастую сильно связаны друг с другом и обычно используются почти одновременно
- Сжатие части кучи обычно быстрее, чем сжатие всей кучи

Таким образом, можно оптимизировать алгоритм сборки мусора путем использования методики поколений

Пример поколений объектов



Оптимизация алгоритма сборки мусора

- Сборка мусора только в нулевом поколении
- Остановка обхода графа на старых объектах
- Выигрыш в скорости на локальности доступа между новыми объектами (все осядут в кэш)
- Тесты показывают, что выделение памяти в .NET быстрее, чем HeapAlloc из Win32
- Также, на 200Mhz Pentium полная сборка мусора в нулевом поколении - меньше 1 мс
- Цель Microsoft - добиться, чтобы GC отнимало не больше времени, чем обычный сбой памяти

Сборка мусора в многопоточных приложениях

- Все потоки должны приостанавливаться на время сборки мусора
- Fully Interruptible Code (т.к. CLR следит за всеми процессами, то все можно остановить)
- Hijacking the thread (подмена кода возврата на специальную функцию, приостанавливающую выполнение нити)
- Synchronization-free Allocations (на мульти-процессоных системах поколение 0 делится на несколько *арен*)
- Scalable Collections (на мультипроцессорных серверных системах куча делится на несколько секций, по числу процессоров)

Сборка крупных объектов

- Крупные объекты (больше 20 Кб) выделяются в отдельной куче.
- Завершение и освобождение крупных объектов происходит так же, как и для мелких, но сжатия никогда не происходит (из-за слишком высоких накладных расходов)
- Эта оптимизация прозрачна для приложения

Явное использование сборки мусора

- System.GC.Collect() инициирует немедленную сборку мусора (параметр - номер поколения, по умолчанию = GC.MaxGeneration)
- System.GC.RequestFinalizeOnShutdown() форсирует использование всех завершителей (это замедляет выход из приложения!)
- Методы GetGeneration для Object и WeakReference возвращают номер поколения

Литература к лекции

- Т. Пратт "Языки программирования: разработка и реализация", М.: Мир, 1979.
- J. Richter "Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework", Parts 1 & 2, MSDN Magazine, Nov. 2000/Dec. 2000
- Э. Гуннерсон "Введение в C#", СПб.: Питер, 2001. 304 с.