Сборщик мусора проверяет, есть ли в куче объекты, которые больше не используются приложением, то есть объекты, на которые больше не ссылаются указатели. Если да, можно освободить занятую ими память. (Если после сбора мусора в куче нет свободной памяти, то оператор new генерирует исключение OutOflVfemoryException.). Откуда сборщик знает, используется объект приложением или нет? Вопрос не из легких.

У каждого приложения есть набор корней (root). Корень - это адрес, по которому находится указатель на объект ссылочного типа. Этот указатель содержит ссылку на объект в управляемой куче или равен null. Например, статическое поле считается корнем, как и любой параметр метода или локальная переменная. Корнями могут быть только переменные ссылочного, а не значимого типа. Рассмотрим конкретный пример и начнем с определения класса.

internal sealed class SomeType

{

private TextWriter m\_TextWriter;

public SomeType(TextWriter tw)

{

m\_TextWriter = tw;

}

public void WriteBytes(Byte[] bytes)

{

for(Int32 x = 0; x< bytes.Length; x++)

{

m\_TextWriter.Write(bytes[x]);

}

}

}

При первом вызове метода WriteBytes JIT- компилятор преобразует код метода на промежуточном языке в машинные команды процессора. Допустим   
CLR работает на базе процессора х86, а метод WriteBytes компилируется в команды процессора, показанные на рис. 2.

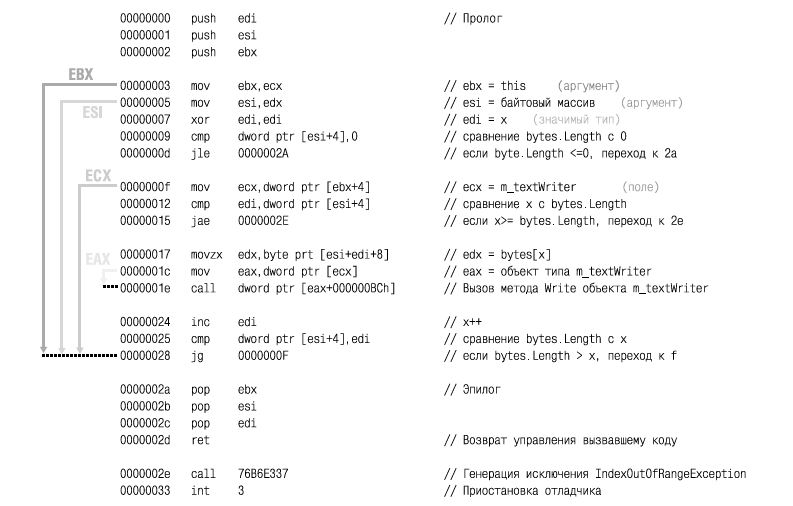


Рис.2 Машинный код, созданный JIТ - компилятором, с иерархией корней

При генерации машинного кода JIТ - компилятор также создает внутреннюю таблицу. Логически каждая строка таблицы указывает диапазон смещений байт машинных кодов процессора для этого метода, а также для каждого диапазона набор адресов памяти и регистры, содержащие корни. Для метода WriteBytes в этой таблице видно, что регистр ЕВХ сначала является корнем со смещение 0x00000003, регистр ESI - корнем со смещением 0x00000005, а регистр ЕСХ - корнем со смещением 0x0000000t. Все эти регистры перестают быть корнями в конце цикла (смещение 0x00000028). Также обратим внимание, что ЕАХ является корнем между 0x0000001с и 0x0000001 е. Регистр EDI служит для хранения величины типа Int32, представленной переменной х в исходном коде, а так как Int32 - значимый тип, то JIТ- компилятор не считает регистр EDI корнем.

Примечание. Описание регистров:

EBX. ECX. ЕАХ - регистры общего назначения. ESI, EDI – регистры указателей и индексов.

Метод WriteBytes довольно прост и все используемые в нем переменные могут быть зарегистрированы. Более сложный метод может использовать все имеющиеся регистры процессора, а некоторые корни будут располагаться относительно стекового фрейма метода. Также надо учитывать, что в архитектуре x86 CLR передает первые два аргумента методу через регистры ЕСХ и EDX. Для методов экземпляров классов в качестве первого аргумента выступает указатель this, всегда передаваемый в регистре ЕСХ. Именно поэтому известно, что в случае метода WriteBytes указатель this передается в регистре ЕСХ и сохраняется в регистре ЕВХ сразу после пролога метода, а также то, что аргумент bytes передастся в регистре EDX и сохраняется в регистре ESI после пролога. Если бы сбор мусора начался во время исполнения кода со смещением 0x00000017 в методе WriteBytes, сборщик мусора знал бы, что аргументы, на которые ссылается регистр ЕВХ (аргумент this), ESI (аргумент bytes) и ЕСХ (поле m\_TextWriter), - это корни, а объекты, соответствующие им в куче, нельзя считать мусором. Кроме того, сборщик может пройти по стеку вызовов потока и определить корни всех вызывающих методов, изучив внутреннюю таблицу каждого из них. Для поучения набора корней, хранимых в статических полях, сборщик просматривает все объекты-типы. Начиная работу, сборщик мусора предполагает, что все объекты в куче - мусор. Иначе говоря, он предполагает, что в стеке нет переменных, ссылающихся на объекты в куче, а также что на объекты в куче не ссылаются регистры процессора и статические поля. Затем сборщик переходит к этапу сбора мусора, называемому маркировка (marking). Он проходит по стеку потока и проверяет все корни. Если окажется, что корень ссылается на объект, в поле SyncBlockJndex этого объекта будет включен бит - именно так объект маркируется. Например, сборщик может найти локальную переменную, указывающую на объект в куче. На рис.3 показана куча с несколькими объектами, в которой корни приложения напрямую ссылаются на объекты А и С, В и F. Все эти объекты маркируются. При маркировке объекта D сборщик мусора замечает, что в этом объекте есть поле, ссылающееся на объект Н. Поэтому объект Н также помечается. Затем сборщик мусора продолжает рекурсивный просмотр всех достижимых объектов.

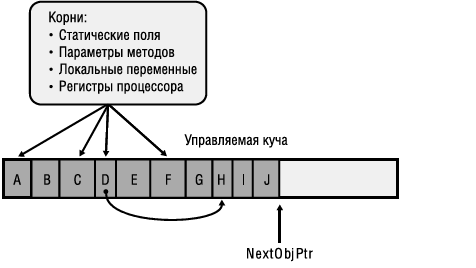


Рис 3. Управляемая куча перед сбором мусора

После маркировки корня и объекта, на который ссылается его поле, сборщик мусора проверяет следующий корень и продолжает маркировать объекты. При попытке пометить объект, уже ранее помеченный, сборщик останавливается. Это нужно по двум причинам. Во-первых, заметно повышается быстродействие, так как сборщик проходит набор объектов не больше одного раза, а во-вторых, исключается возможность бесконечных циклов, возникающих из-за замкнутых связных списков объектов. После проверки всех корней куча содержит набор маркированных и немаркированных объектов. Маркированные объекты, в отличие от немаркированных, достижимы из кода приложения. Недостижимые объекты считаются мусором, а занимаемая ими память становится доступной для освобождения. Затем сборщик мусора переходит к следующему этапу сбора мусора, называемому сжатие (compact phase). Теперь он проходит кучу линейно в поисках непрерывных блоков немаркированных объектов, то есть мусора. Небольшие блоки сборщик не трогает, а в больших непрерывных он перемешает вниз все «немусорные» объекты, сжимая таким образом кучу. Естественно, перемещение объектов в памяти делает все переменные и регистры процессора, содержащие указатели на объекты, недействительными. Поэтому сборщик мусора должен снова проверить и обновить все корни приложения, чтобы все значения корней указывали на новые адреса объектов в памяти. Кроме того, если объект содержит поле, указывающее на другой перемещенный объект, сборщик должен исправить эти поля. После сжатия памяти в указатель NextObjPtr управляемой кучи заносится первый адрес за последним объектом, не являющимся мусором. На рис. 4 показана куча после сбора мусора.

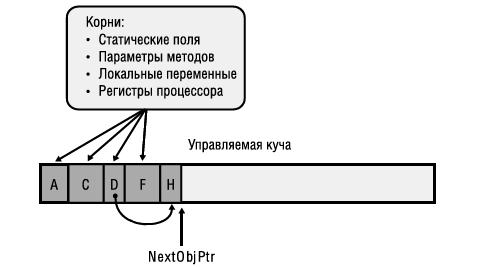


Рис. 4 Управляемая куча после сбора мусора

Примечание: Если сбор мусора — такая замечательная вещь, то почему его нет в С++.   
Дело в том. что сборщику мусора необходима возможность определять корни приложения и находить все указатели на объекты. Проблема с неуправляемым С++ в том, что он допускает приведение указателей одного типа к другому, поэтому нельзя узнать, на что ссылается указатель. Управляемая куча в CLR всегда знает настоящий тип объекта и при помощи метаданных способна определить, какие члены объекта ссылаются на другие объекты.