Bartok Memory Management

Contents

List of Figures

List of Tables

Revision History

Date	Author	Comments
11-15-2004	Daniel Frampton	Initial Revision

1 Overview

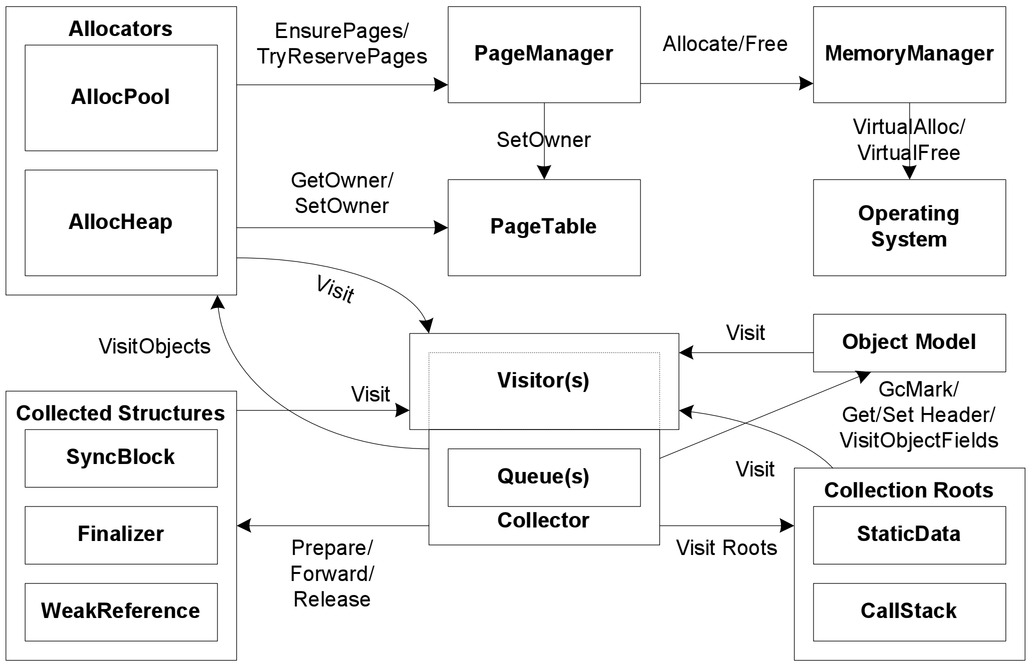

Figure 1: Components of the Bartok memory management system.

2 Object Model

The basic layout of each object is shown in figure 2.

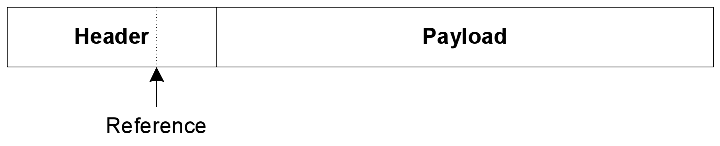

Figure 2: The layout of an Object.

2.1 Header

Each object includes header data that provides a mechanism for storing per object information such as type and GC state. The makeup of the header data is shown in Figure 3.

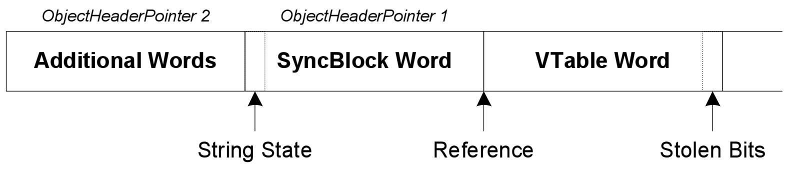

Figure 3: The layout of the object header.

2.1.1 VTable

See class: System.VTable

The *vtable* field of an object points an object of type VTable. This includes all type and method information and is used for all virtual method calls and dynamic type checks.

The VTable is also used as a parameter to all allocations, and is queried by the collector to find object sizes and locate reference fields within objects. Finding pointer fields is achieved through a pointer tracking word, see Section 5.7 for more detail.

The VTable indicates if an object requires 8-byte alignment. Any object that requires 8-byte alignment is guaranteed to have the allocator align the start of the object payload.

During collection, the low order bits of the *vtable* field can be used to indicate some GC state. This requires all reads of the *vtable* field to apply a bit mask.

2.1.2 Sync Block Index

See class: System.SyncBlock

Each object has room in the header for a sync block index for reasons as detailed in Section .

2.1.3 String State

See class: System.Globalization.CompareInfo

In order to facilitate efficient string comparison operations, two bits in the header are stolen. The string state indicates the type of characters that are used in the string.

2.1.4 Additional Words

See class: System.Object

Bartok supports the inclusion of an arbitrary number of additional header words. These can be used for:

- a reference count for each heap object;
- a profile word to allow per-object runtime profiling; or
- a header based queue to allow enqueuing objects without allocating a buffer.

The number of words is determined by a build time stage control *ObjectHeader-Pointers*. Note that the base value of this is 1 to include the word used by the Sync Block index.

2.2 Scalars

Scalars, both reference types and boxed value types have their fields laid out after the *vtable* field. This is shown in Figure 4.

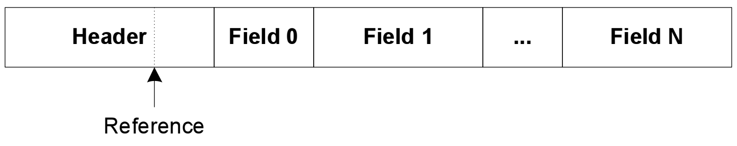

Figure 4: The layout of a Scalar.

See Section 5.7 for details on how pointers are located within the object at runtime

2.3 Vectors

Vectors are one-dimensional arrays. As this is the common case it is optimized within the runtime system. Note that vectors are exposed to users of the runtime as arrays.

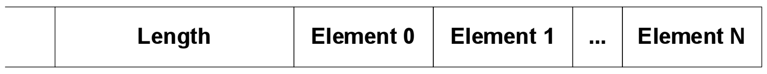

Figure 5: The layout of a Vector.

The layout of a vector is shown in Figure 5. A vector contains a single integer field, *length*. After this elements are laid out from index 0 and up.

See Section 5.7 for details on how pointers are located within the vector at runtime

2.4 Arrays

Arrays contain information regarding the number of dimensions (*rank*), the absolute size of the array (*length*), and the base and length of each dimension (*baseN lengthN*). This is shown in Figure 6.

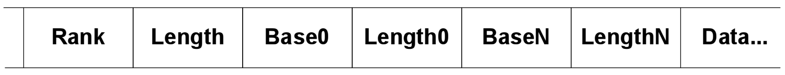

Figure 6: The layout of an Array.

See Section 5.7 for details on how pointers are located within the array at runtime

2.5 Strings

Strings are laid out identically to a vector of characters (See Section 2.3).

3 Page Management

The following sections outline how Bartok manages the explicit management of memory at the page level.

3.1 Page Table

See class: System.GCs.PageTable

The Page Table manages an array of byte entries for every page in the system. This table is manipulated by the allocator, garbage collector, and indirectly through the Page Manager. Each page has a byte value corresponding to a value on Table 1.

Value	Meaning
`0x00 - 0x03`	Reserved for specific interpretation by the collector, such as to distinguish generations.
`0x04 - 0x07`	Also reserved for specific interpretation by the collector, but in copying collectors used to represent *zombie* pages.
`0x08`	Allocated Unused Dirty page
`0x09`	Allocated Unused Clean page
`0x0a`	Unallocated page
`0x0b`	Allocated NonGC page
`0x0c - 0x0f`	Allocated System page

Table 1: Possible Page Table values

Among other things these tags are used to determine the size of the managed heap, create the filter for the generational write barrier, and assist visiting objects in the heap.

3.2 Page Manager

See class: System.GCs.PageManager

TODO: What about MarkUnusedPages, should that really be called from anywhere?

The page manager is the interface that classes within Bartok use to allocate and free memory pages. The most important methods within this class are:

EnsurePages This method either returns a chunk of contiguous pages of the requested size or crashes with an out of memory error.

TryReservePages This method attempts to return a specific chunk of pages. If the requested pages are not available then this method simply returns nothing.

ClearPages This method returns pages back to the page manager to allow it to reconsider them for allocation. The page manager may also release the pages back to the operating system.

The page manager also includes interfaces to allow the various subsystems of Bartok to indicate memory regions that are occupied by stack, static data, and system pages.

When the page manager is forced to go to the operating system to allocate new pages, it allocates a large chunk, returns the number of pages as requested to the caller, and marks the rest of the pages as allocated but unused.

3.3 Memory Manager

See class: System.GCs.MemoryManager

The memory manager is the interface from within Bartok to the virtual memory subsystem of the host operating system. The memory manager exposes a simple interface that includes methods to Allocate and Free chunks of virtual memory.

Memory is allocated and freed from the operating system in large chunks to minimize the effect of the call overhead.

The memory manager is also queried to provide information on the total memory available and the amount available for the managed heap.

4 Collected Structures

Several data structures in the Bartok demand collector cooperation in order to keep them in sync with the object graph. In order to facilitate this process each collector must provide at which the following three phases can be executed.

Initiation Reference fields are still valid so that object based structures can be traversed, and references can be reported to the collector to be considered a root for the collection.[1]

Resurrection During this phase the collector must allow a caller to query if object references point to live objects, and also add new objects to the closure.

Forwarding During this phase the collector must allow a caller to report new references that are either zeroed or forwarded depending on if the referent object is live. Note that at this point no further objects can be resurrected.

4.1 Sync Blocks

See class: System.SyncBlock

There are several operations that can be performed dynamically on objects at runtime. These may require associating additional information with certain objects. The currently identified operations are:

- Locking on an object using the `Monitor` classes `Enter` and `Exit` methods as exposed by the C# `lock` syntax;
- Finding the hash-code for an object using `GetHashCode`; and
- A safe index to identify a `Delegate` that has been passed to native code.[2]

[1]Note that once a reference is reported to the collector, then it is no longer guaranteed to be a valid reference.

[2]Once a SyncBlock index has been passed into native code, the SyncBlock is deemed to be immortal and is considered a strong reference to the object for purposes of collection.

As the operations are relatively infrequent, it was considered wasteful to pre-allocate space in every object to support these operations. Instead, a single field is allocated for each object to optionally hold a SyncBlock index into.

Each individual SyncBlock as shown in Figure 7 contains the following fields.

TODO: Figure

Figure 7: A Sync Block.

Object A reference to the object that the SyncBlock is allocated to. This is only valid when the SyncBlock is not in the *Free* status.

Next Free Block The next free index. This is only valid when the SyncBlock is in the *Free* status.

Monitor A reference to the monitor instance that has been created for the object to allow synchronization.

Status The status for this SyncBlock. One of *Free, Dying, InUse, Immortal.*

When the system is initialized, a SyncBlock table is allocated, and an initial entry containing 10 SyncBlocks is created.

The SyncBlock manager keeps two different structures that it uses to allocate SyncBlock indices. Firstly it manages a free-list that is linked through all SyncBlocks that have been allocated and subsequently reclaimed. Secondly, it maintains the highest SyncBlock index that has ever been allocated. When a new SyncBlock is requested and the free-list is empty, then the tables are extended from this highest index.

When the capacity of an array is reached, a new array twice the size of the previous array is allocated, and the SyncBlock table is extended. Indices into all tables are maintained globally. This means that if you are the third entry (index 2) in the second table, and the first table has length 10, then your SyncBlock index is 12.

4.1.1 Collector Processing

Sync blocks are processed during collection as follows:

Initiation Firstly, copy references to SyncBlock data structures into an unmanaged region of memory so that it can perform work on the tables later in the collection. Secondly, report any references to objects in sync blocks marked as immortal to the collector.

Resurrection No processing required.

Forwarding Using the shadow data structures, update references to any objects that are still alive, and recycle any sync blocks that refer to unreachable objects.

In the future it would be possible to coalesce several adjacent SyncBlock tables into a single table at collection time. This would slightly reduce the work required to look up an entry in the tables.

4.2 Finalization

See class: System.Finalizer

Objects are allowed to define a method to be called when the object becomes unreachable. When an object with such a method is allocated Bartok adds it to a *candidate* queue. When a candidate object becomes unreachable it is resurrected and placed on a *ready* queue to have the finalize method executed.

If a finalizable object is registered for finalization multiple times, then the finalizer is run multiple times. It is possible to remove a finalizable object from the candidate queue by calling a suppress method.

The life cycle of a finalizable object is therefore:

1. Object registered as a candidate
2. Object becomes unreachable (detected during a collection).
3. Object is resurrected and becomes ready.
4. Object has the finalize method run by the finalization thread.
5. Object is unreachable and collected at a subsequent collection. [3]

[3]It is possible for the finalize method to resurrect the object, changing the life cycle.

4.2.1 Collector Processing

Finalization processing occurs during collection as follows:

Initiation No processing required.

Resurrection All candidates are processed, and if a candidate has become unreachable. Once all candidates have been checked,[4] any candidate that became unreachable is reported to the collector to be resurrected.

Forwarding No processing required.

4.3 Weak References

See class: System.WeakReference

Weak references are references to objects that are not supposed to keep the target object alive. Clearly the collector does still need to process these references to either null them out if the target object dies, or update the reference if the target object is moved.

Weak references to finalizable objects can be set to either be nulled when the object is finalized, or follow it through finalization.

TODO: Figure

Figure 8: Weak Reference Chain.

To create a weak reference a programmer creates a WeakReference object. Weak reference objects are implemented by creating a linked list and inserting new weak references into the list as they are allocated. This can be seen in Figure 8. This list is then traversed during collection to perform the appropriate processing.

[4]Processing proceeds this way to deal with the following case: objects A and B are both candidates and become unreachable, and A has a reference to B, according to the spec both A and B's finalize methods must be run. That is, A being resurrected can not stop B having its finalize method executed.

4.3.1 Collector Processing

Weak references are processed during collection as follows:

Initiation No processing required.

Resurrection At the start of the resurrection phase, all weak references that are not set to track resurrection are reported to the collector and either nulled or included to be forwarded.

Forwarding All weak references that are set to track resurrection are reported to the collector and either nulled or included to be forwarded.

5 Compiler Support

5.1 Allocation Sequence

See class: Bartok.Convert.ConvertMethod

At allocation sites within the program, the code generator must translate MSIL allocation methods into calls to the selected allocator (AllocHeap or AllocPool).

To allow more efficient allocation, the code generator should be able to inline some allocation calls.

5.2 Write Barrier

See class: Bartok.Opt.Ir.IrWriteBarrier

The runtime is required to inject a write barrier sequence at appropriate writes during code generation.[5]

5.3 Finalizable Objects

See class: System.Finalizer
See class: Bartok.Convert.ToMir

The runtime is required to inject a finalization registration call for any object that is allocated that requires finalization.

5.4 Thread Local Data

See class: System.Threading.Thread
See class: System.GC.AllocHeap
See class: System.GC.AllocPool

In order for efficient allocation, each thread holds local allocation state from which it can allocate objects without requiring any synchronization. This state is exposed

[5]This is only required for some collectors.

through two structs (one for free list allocation, and one for bump pointer allocation) and included by the runtime in THREAD objects.

5.5 Static Data Maps

TODO: I don't know how the bitmap is generated.

5.6 Stack Maps

See class: SYSTEM.GCS.CALLSTACK

TODO: I don't know how the maps are generated.

5.7 Object Maps

See class: SYSTEM.GC.UTIL
See class: BARTOK.CONVERT.POINTERTRACK

Depending on the type of the object different methods are used to locate references. For more information on object layout see Section 2. The VTABLE for each type contains a *PointerTracking* field. The low order 4 bits of this field indicate the method of pointer tracking used for the object. The upper 28 bits are used differently depending on the tracking method:

Sparse Object `[0x1]` If there are 7 or fewer pointers each with an offset less than or equal to 15 words, then the offset of each is stored using 4 bits of the *PointerTracking* field. This is the preferred method of storing pointer information.

Dense Object `[0x3]` If all of an object's pointer fields are located in the first 28 words of the object, then the top 28 bits of the *PointerTracking* field are used as a bitmap for where the fields are located.

Other Object `[0x0]` In the case none of the optimized methods can be employed, the high 28 bits of the field contain a reference to a list of word offsets. The first element of the list is the length, and each subsequent element holds an offset of a pointer field.

Pointer Vector `[0x5]` If a vector contains elements that are pointers each is processed in a loop.

Other Vector `[0x7]` If a vector contains struct elements (which may contain references) each is processed according to the struct's VTable in a loop.

Pointer Array `[0x9]` If an array contains elements that are pointers each is processed in a loop.

Other Array `[0xb]` If an array contains struct elements (which may contain references) each is processed according to the struct's VTable in a loop.

String `[0xd]` Strings contain no references so they do not need to be processed.

6 Allocation

6.1 Static Allocation

Statically allocated data is compiled into the executable and loaded into memory when the application is run. This include type information, the executable code, and the location of static fields for each class.

6.2 Bootstrap Allocation

See class: System.GCs.BootstrapMemory
See class: System.GCs.AllocPool

Before all the necessary classes to allow normal allocation have been initialized, any data that is required must be allocated into bootstrap memory. Bootstrap memory is a special case of bump pointer allocation and uses a subset of the code from AllocPool.

Once the system is fully booted, bootstrap memory is closed and further allocations into it are not permitted. Objects within bootstrap memory are not automatically traced by the collector, which means that any class that creates bootstrap data that may point into the managed heap must manually reveal that data to the collector.

6.3 Bump Pointer Allocation

See class: System.GCs.AllocPool

6.4 Free List Allocation

See class: System.GCs.AllocHeap

pages

blocks

cells

global page

global free page

local page

freelist

freeing

allocation

7 Collection

7.1 Root Set

For the collection to proceed a set of *root* objects must be identified. The transitive closure of the object graph starting from these objects is considered the live set, and all unreachable objects may be collected.

7.1.1 Static Data

See class: System.GCs.StaticData

The area of static data that contains the static fields of classes needs to be processed for any references into the managed heap. This is achieved by walking through the static data and referring to a bitmap that indicates where pointer words are located. See Section 5.5 for details of how the bitmap is created.

7.1.2 Bootstrap Memory

See class: System.GCs.BootstrapMemory

Bootstrap memory may contain objects that contain references into the managed heap. Each user of bootstrap memory that allocates such objects is queried to add these references into the root set.

7.1.3 Call Stack

See class: System.GCs.CallStack

All live references within the stack that point into the heap are considered roots for a collection. The compiler is required to generate stack maps that locate such references. At this stage to simplify the process of generating the maps all stack references are considered to be potentially interior references.

Some references in the call stacks may be *pinning* referents. These references are presented to the collector separately and are guaranteed not to be moved during the collection.

7.2 Visitors

See class: System.GC.PtrVisitor
See class: System.GC.ObjectVisitor

Where a part of the system is used for processing values in multiple contexts, the visitor pattern is used. Bartok currently uses two types of visitors:

Pointer This is used in many places within Bartok to report memory locations that contain references to the collector, from reporting root references to visiting fields within individual objects.

Object This is primarily used to facilitate a walk of all objects in the heap. For example this is used during the sweep phase of the Mark Sweep collector.

7.3 Write Barriers

Write barriers are code sequences that are inserted during compilation to notify the collector of events of significant for collection.

7.3.1 Generational Barrier

See class: System.GC.WriteBarrierSSB

The role of a generational barrier is to record pointers that refer to an object in a younger generation. It is then possible to use these pointers as roots for a collection of the younger generation without considering any other objects in the older spaces.

In Bartok this is achieved by looking up and comparing the values that are stored in the page table (see Section 3.1) for the source and target objects. If the source object is in an older generation than the target then the location is stored in a sequential store buffer.

The sequential store buffer (SSB) is created in a fixed number of preallocated pages (128). If the buffer becomes full a collection is triggered. To support the removal of duplicates as required by the Sliding collector, the buffer also contains a Uniquify method that removes duplicates through sorting.

7.4 Semispace Collector

See class: SYSTEM.GCS.SEMISPACECOLLECTOR

TODO: Provide more details of Cheney Scan?

TODO: Provide information on pinned and large object processing.

The basic concept of a Semispace collector is to trace through objects in a region of memory (the *from* space), copy live objects into another region of memory (the *to* space), and then reclaim the original region en masse.

The Semispace Collector in Bartok is a generational collector. The following description for how collection is performed is the same for both a nursery and full heap collection in all respects but the selection of the region of *zombie* pages.

The collection begins by marking all pages in the from space as *zombie* pages. References locations are processed as shown in Table 2 starting with the root references.

```
VISIT($*location$)
    $object \leftarrow *location$
    IF ($object$ = null)
        RETURN
    IF (($object.vtable$ & 0x1) = 0x0)
        $object.vtable \leftarrow$ (COPY($object$) & 0x1)
    $*location \leftarrow (object.vtable$ & $\neg$0x1)
```

Table 2: Semispace location visitor.

This visitor is not enough to compute the transitive closure. To do this the Semispace collector uses a Cheney scan. This works by maintaining a cursor that moves through objects copied to the to space one by one, visiting each reference location. When the cursor moves past the last object, then all live objects have been copied and the collection is complete. At this point all of the zombie pages can be returned to the operating system.

The forward-only version of the visitor, as discussed in Section 4, is very similar to the normal visitor, but instead of copying and replacing the reference when the *vtable* field does not have the low order bit set, it zeroes the location and returns.

7.5 Sliding Collector

See class: System.GCs.SlidingCollector

7.6 Adaptive Copying Collector

See class: System.GCs.AdaptiveCopyingCollector

The adaptive copying collector dynamically switches between using the Sliding and Semispace collectors. This avoids the additional cost of the Sliding collector most of the time, but allows the use of a more space efficient collector when memory usage is high.

The heuristic for determining to use the sliding collector is if the from space is greater than half of the system's available physical memory.

7.7 Mark Sweep Collector

See class: System.GCs.MarkSweepCollector