

Garbage Collection

Tomáš Matoušek
&
Ladislav Prošek

tmd.havit.cz/teaching/csharp.htm

Garbage Collection In General

- two phases
 - garbage detection
 - garbage reclamation
- several methods for each phase
 - a technique for reclamation depends on the one used for detection
 - phases may also interleave
- roots
 - variables where to start detection from
 - static and global fields
 - locals and arguments on stacks
 - CPU registers
- reachability
 - an object is reachable (living) iff it is reachable from any of the roots via (strong) references
 - an unreachable object is a garbage

Basic Algorithms

- reference counting
- mark-sweep collection
- mark-compact collection
- copying collection
- implicit non-copying collection
- ...

Mark-Sweep Collection

- detection – marking
 - DFS starting from the root set
 - marks each visited object
 - a bit of the object is altered or some data structure is filled
- reclamation – sweeping
 - heap is traversed and all garbage linked to a free list
- pros
 - no operations during computations
- cons
 - heap fragmentation, bad locality of reference
 - overhead proportional to size of a heap
 - including both live and garbage objects

Mark-Compact Collection

- detection – marking
- reclamation – compacting
 - living objects are moved to be contiguous
 - linear scan over the heap
 - several phases
 - determination of new positions of the living objects
 - objects copying
 - references update
- pros
 - very fast allocation
 - incrementing a pointer + comparison to a size of the heap
 - remedy fragmentation and locality of reference
 - overhead proportional to living data size
- cons
 - several walks over living objects

Copying Collection

- scavenging
 - integrates garbage detection with reclamation
- Cheney's algorithm for scavenging
 - objects visited by BFS copied to separate area
 - a queue for BFS is made of fields of copied objects
 - references are updated when visiting next objects
 - forwarding pointer
 - copied objects are marked with pointer to the new position
 - if an object is visited for the next time no copying is done
- semispace collectors
 - heap consists of two contiguous areas
 - one for allocations the other for the collection

Copying Collection (cont.)

- pros
 - overhead proportional to living data size (one walk only)
 - fast allocation
 - the more memory is available the faster the program is
- cons
 - needs a lot of additional memory
 - two times larger than can be used for allocation
 - for the case there is no garbage at the time of the collection

Improvements

- generational collection
- large objects heap
- incremental garbage detection
- ...

Generational Collection

- experience
 - younger objects have shorter lifetime
 - older objects survive many collections
- principles
 - segregation of objects into several areas by age
 - younger generations collected more frequently
 - mark-compact or copying collection can be used here
 - survivals copied to elder generation
 - where mark-sweep collection can be used
 - new objects allocated from the youngest generation
- improves efficiency and locality
 - only references to the generations being collected are followed
 - only generations being collected are scanned for living objects

Intergenerational References

- problem
 - tracking of references from older to younger generations
 - collector doesn't inspect all objects in elder generations
- write barrier
 - each op. storing a reference followed by additional instructions
 - marks parts of the heap which should be scanned during collection for intergenerational references
 - several techniques (e.g. card marking)

Card Marking

- card table
 - heap divided into cards (e.g. 128 bytes wide)
 - a bitmap of the heap (one bit per card)
 - a bit is set if the card contains reference to younger generation
 - marked cards are searched for intergenerational references
- problem
 - a card may start in the middle of an object
 - How to find an object within a given card?
- brick table
 - heap divided into bricks (e.g. 2K wide)
 - for each brick the table contains a relative position of any object located in the brick
 - if there is no object in the brick it contains a brick index where it is
 - used for searching for arbitrary object within the specified range

Large Objects Heap

- objects larger than some threshold are allocated here
- no copying
- mark-sweep collection is used for this heap

.NET Framework

- generations
 - generation 0 (a.k.a. ephemeral)
 - the youngest (objects are allocated here)
 - generation 1
 - generation 2
 - the rest of the heap + large object heap
- thresholds
 - each generation has defined a maximal size (dynamically changes)
 - all generations surpassing that thresholds are marked-compacted
 - garbage in elder collections is ignored
 - garbage in large object heap is always marked-swept
- write barrier via card marking
 - detects references from elder to younger generations

Roots: Registers and Local Variables

- JIT creates a table for each method
 - maps ranges of byte offsets in native code to sets of roots
 - trade-off between accuracy and size of the table
- stack trace
 - GC examines all tables of methods on the stacks
- debugging
 - it would be confusing if variables were collected during debugging
 - DebuggableAttribute used on assembly
 - JIT makes variable range as long as is its scope

Special Features

- pinned objects
 - objects which cannot be moved
 - on the stack only
- managed pointers and typed references
 - can point inside objects (interior pointers)
 - on the stack only
 - updated as the target objects are moving
- finalization
- weak references

Finalization

- C# destructor is

```
protected override void Finalize()  
{  
    try { /* code */ } finally { base.Finalize(); }  
}
```

- executed by a single finalization thread
 - a special thread of the EE devoted to call finalizers
- the finalizer is called when
 - object's generation is being collected and object is a garbage
 - because it is full
 - because GC.Collect() has been called
 - AppDomain is being unloaded
 - AppDomain.IsFinalizingForUnload() returns true than
 - CLR is being shut down
 - timeout for finalizers
 - Environment.HasShutdownStarted is true than

Finalization Background

- finalization set
 - objects having finalizer
 - objects added by `newobj` instruction before the constructor is called
- f-reachable queue
 - belongs to the root set
 - a queue of objects waiting for finalization (i.e. execution of a finalizer)
 - processed by finalization thread
 - populated during collection
 - added objects survives collection (promoted to elder generation)
- resurrection
 - implicit (by GC)
 - object is moved from finalization list to f-reachable queue
 - explicit (by user code)
 - object makes itself reachable during finalization
- routines
 - `GC.SuppressFinalization()`
 - removes object from finalization set (actually, sets a bit in syncblock)
 - `GC.ReRegisterForFinalization()`
 - adds object to finalization set (must have finalizer)

Finalization Guideline

- use for classes maintaining unmanaged resources
- don't use if not necessary
- make finalized objects as small as possible
- don't use reference fields in finalized objects if possible
- don't access reference fields of finalized objects in the finalizer
- make finalizer as fast as possible
- check whether unmanaged resources has been allocated
 - finalizer is called even if constructor throws an exception
- use along with the dispose pattern

Deterministic Clean-up

- dispose pattern
 - interface IDisposable
- can be used along with finalization
 - if we don't want to wait until finalizer is eventually called
 - a user may clean up the object explicitly

Deterministic Clean-up Example

```
// thread-safe wrapper of some unmanaged handle:
public sealed class OSHandle : IDisposable
{
    private int disposed;
    private IntPtr handle;
    public OSHandle(IntPtr h) { handle = h; disposed = 0; }

    public void Dispose() {GC.SuppressFinalize(this); CleanUp(true);}
    public void Close() { Dispose(); } // redundant (alias)
    ~OSHandle() { CleanUp(false); }

    void CleanUp(bool disposing)
    {
        // the object has already been cleaned up:
        if (Interlocked.CompareExchange(ref disposed, 1, 0) == 1) return;

        // clean-up:
        if (disposing) { /* safe to access reference fields here */ }
    }
}
```

Weak References

- managed references
 - strong (default)
 - weak (`System.WeakReference`)
 - short – nulled after garbage detection before finalization list scan
 - long – nulled after finalization list scan (tracks resurrection)
- to access an object a strong reference is needed
 - weak reference can be converted to a strong one
 - `WeakReference.Target` property
- weak references registered in internal EE tables
 - short weak references table
 - long weak references table
 - scanned before (S) and after (L) finalization list scan

Advanced Issues

- suspending threads before the collection
 - thread is hijacked if it is in an unmanaged code
- multiprocessors
 - concurrent collections
 - synchronization-free allocations
 - ...

Further Information

- Paul R. Wilson:
Uniprocessor Garbage Collection Techniques, 1992
- Richard Jones, Rafael Lins:
Garbage Collection – Algorithms for Automatic Dynamic Memory Management,
John Wiley & Sons, 1996
- Jeffrey Richter:
Applied Microsoft .NET Framework Programming,
Microsoft Press, 2002
- David Stutz:
Shared Source CLI Essentials,
O'Reilly, 2003