Memory Management

Mingsheng Hong
CS 215, Spring 2008

Motivation

- Recall unmanaged code
  - eg C:

```
{
  double* A = malloc(sizeof(double)*M*N);
  for(int i = 0; i < M*N; i++) {
    A[i] = i;
  }
}
```

  - What's wrong?
    - memory leak: forgot to call `free(A);`
    - common problem in C

Motivation

- Solution: no explicit malloc/free
  - eg. in Java/C#

```
{
  double[] A = new double[M*N];
  for(int i = 0; i < M*N; i++) {
    A[i] = i;
  }
}
```

  - No leak: memory is "lost" but freed later
- A Garbage Collector tries to free memory
  - keeps track of used information somehow

System.GC

- Can control the behavior of GC
  - not recommend in general
  - sometimes useful to give hints
- Some methods:
  - Collect
  - Add/RemoveMemoryPressure
  - ReRegisterFor/SuppressFinalize
  - WaitForPendingFinalizers

Finalizers

- `protected virtual void Finalize()`
- Finalizers are called *nondeterministically*
  - During the garbage collection process
  - When the *GC.Collect* method is called
  - When the CLR is shutting down
  - Can be suppressed by `GC.SuppressFinalize`
  - Current thread can wait with `GC.WaitForPendingFinalizers`

Destructors

- Finalizers cannot be overridden
  - Instead, write a *class destructor* `~Classname()`
- Destructors are called bottom-up

```
public class A {
    ~A() { Console.WriteLine("A d'tor"); }
}
public class B : A {
    ~B() { Console.WriteLine("B d'tor"); }
}
```

- `A a = new B(); //program then shuts down`

Finalizers are Expensive

- Finalization in GC:
  - When object with Finalize method created
    - add to Finalization Queue
  - First GC: moved to Freachable queue
  - After finalizer is called, memory released in a future GC
    - One single thread to call finalizers
- At least 2 GC cycles needed

IDisposable

- `IDispose` declares `void Dispose()`
  - Can be explicitly invoked
-
```
public class A: IDisposable {
    public A() { // Allocate resources }
    public void Dispose() { // Release resources }
}
```
-
```
A disposableobject=null;
try { disposableobject=new A(); }
finally {
    disposableobject.Dispose();
    disposableobject=null;
}
```

The using Statement

- `using` calls `Dispose` automatically

```
using(A disposableobject= new A()) {
    // use object
}
```

- Can declare multiple objects

```
using(A a1 = new A(), a2 = new A())
using(B b1 = new B()) {
    // use objects
}
```

Weak References

- Sometimes want to keep references but not cause the GC to wait
  - `A a = new A();`
`WeakReference wr = new WeakReference(a);`
  - Now `a` can be collected
    - `wr.Target` is `null` if referenced after `a` collected
- Usage
  - Large objects
    - infrequently accessed
    - nice to have but can be regenerated
  - Handles to strings in a string table

Object Pinning

- Can require that an object not move
  - could hurt GC performance
  - useful for unsafe operation
  - in fact, needed to make pointers work
- syntax:
  - `fixed(…) { … }`
  - will not move objects in the declaration in the block

Soundness and Completeness

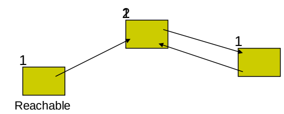
- For any program analysis
  - Sound?
    - are the operations always correct?
    - usually an absolute requirement
  - Complete?
    - does the analysis capture all possible instances?
- For Garbage Collection
  - sound = does it ever delete current memory?
  - complete = does it delete all unused memory?

Reference Counting

- Keep count of references
  - on assignment, increment (and decrement)
  - when removing variables, decrement
    - eg. local variables being removed from stack
  - at ref count 0, reclaim object space
- Advantage: incremental (don't stop)
- Is this safe?
  - Yes: not reference means not reachable

Reference Counting

- Disadvantages
  - can't detect cycles
  - constant cost, even when lots of space
    - optimize the common case!

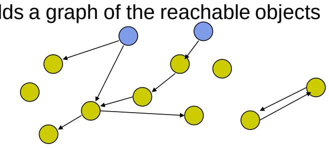


Reachability Graph

- Instead of counting references
  - keep track of some top-level objects
  - and trace out the reachable objects
  - only clean up heap when out of space
    - much better for low-memory programs
- Two major types of algorithm
  - Mark and Sweep
  - Copy Collectors

Reachability Graph

- Top-level objects (roots)
  - managed by CLR
  - local variables on stack
  - registers pointing to objects
- Garbage collector starts top-level
  - builds a graph of the reachable objects



Mark and Sweep

- Two-pass algorithm
  - First pass: walk the graph and mark all objects
    - everything starts unmarked
  - Second pass: sweep the heap, remove unmarked
    - not reachable implies garbage
- Soundness?
  - Yes: any object not marked is not reachable
- Completeness?
  - Yes, since any object unreachable is not marked
  - But only complete *eventually*

Copy Collectors

- Instead of just marking as we trace
  - copy each reachable object to new part of heap
  - needs to have enough space to do this
  - no need for second pass
- Advantages
  - one pass
  - compaction
- Disadvantages
  - higher memory requirements

Compacting Copy Collector

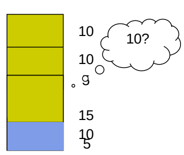
- Move live objects to bottom of heap
  - leaves more free space on top
  - contiguous allocation allows faster access
    - cache works better with locality
- Must then modify references
  - recall: references are really pointers
  - must update location in each object

Compacting Copy Collector

- Another possible collector:
  - divide memory into two halves
  - fill up one half before doing any collection
  - on full:
    - walk the graph and copy to other side
    - work from new side
- Need twice memory of other collectors
- But don't need to find space in old side
  - contiguous allocation is easy

Fragmentation

- Common problem in memory schemes
- Enough memory but not enough contiguous
  - consider allocator in OS



Heap Allocation Algorithms

- best-fit
  - search the heap for the closest fit
  - takes time
  - causes external fragmentation (as we saw)
- first-fit
  - choose the first fit found
  - starts from beginning of heap
- next-fit
  - first-fit with a pointer to last place searched

C# Memory management

- Related to next-fit, copy-collector
  - keep a NextObjPointer to next free space
  - use it for new objects until no more space
- Keep knowledge of Root objects
  - global and static object pointers
  - all thread stack local variables
  - registers pointing to objects
  - maintained by JIT compiler and runtime
    - eg. JIT keeps a table of roots

Generations

- Current .NET uses 3 generations:
  - 0 – recently created objects: yet to survive GC
  - 1 – survived 1 GC pass
  - 2 – survived more than 1 GC pass
- During compaction, promote generations
  - eg. Gen 1 reachable object goes to Gen 2
- Assumption: longer lived implies live longer
  - Valid for many applications

C# Garbage Collectors

- Workstation GC
  - Concurrent GC
  - Non-concurrent GC
- Server GC
  - Optimize for throughput, not response time