Memory Management

Hussam Abu-Libdeh
CS 2026, Spring 2009

Administrative Announcement

- Assignment #3 due Friday at 11:59PM
  - Office hour today after class
  - Questions ?

Motivation

- Recall unmanaged code
  - e.g. C:

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

  - What can go wrong?
    - Memory leak: forgetting to call `free(A);`
    - Common problem in C/C++

Motivation

- Solution: no explicit malloc/free
  - e.g. 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 data somehow

System.GC

- Can control the behavior of the Garbage Collector (GC)
  - Not recommended in general
  - Sometimes useful to give hints
- Some methods:
  - `Collect`
  - `Add/RemoveMemoryPressure`
  - `ReRegisterFor/SuppressFinalize`
  - `WaitForPendingFinalizers`

Finalizers

- `protected virtual void Finalize()`
- Finalizers are called nondeterministically
  - You don't know exactly when will be called
  - Before an object is freed up by garbage collector
  - When `GC.Collect` method is called
  - When CLR is shutting down
  - Can be suppressed by `GC.SuppressFinalize`

Destructors

- Called when object is being destroyed
- Used to free up resources tied to it
- Destructors are called “bottom-up”

```
public class A {
   ~A() { Console.WriteLine(“Destroy A”); }
}
public class B : A {
   ~B() { Console.WriteLine(“Destroy B”); }
}
```

- B's destructor is called first then A's

Destructors and Finalizers

- Destructors are just syntax sugar for finalizers
- `~MyClass() { … }` gets translated to

```
protected override void Finalize() {
   try { // do work }
   finally { base.Finalize(); }
}
```

- Use destructors instead of finalizers

IDisposable

- Used to free **unmanaged** resources
  - Files, streams, handles ..etc
- `IDisposable` declares `void Dispose()`
  - Can be explicitly invoked
  - Often used in the `finally` block of a `try/catch`
- ```
  public class A : IDisposable {
     public A() { .. }
     public void Dispose() { // free resources }
  }
  ```
  ```

The `using` Statement

- The `using` block declares variables and calls `Dispose()` on it automatically

```
using (Font font1 = new Font("Arial", 10.0f)) {
    byte charset = font1.GdiCharSet;
}
```

- Compilers translate this to

```
{
    Font font1 = new Font("Arial", 10.0f);
    try {
        byte charset = font1.GdiCharSet;
    } finally {
        if (font1 != null)
            ((Idisposable)font1).Dispose();
    }
}
```

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` has been collected

C# Garbage Collectors

- Different types based on the machine and environment
- Workstation GC
  - Concurrent GC
  - Non-concurrent GC
- Server GC
  - Optimized for throughput, not response time

Soundness and Completeness

- For any program analysis
  - Sound?
    - Are the operations always correct
  - 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?

Some intuition on how a Garbage Collector works

Reference Counting

- Keep count of references of an object
  - Increment ref count on assignment
  - Decrement ref count when out of scope
- When ref count reaches zero
  - Reclaim memory
- Why can this work ?
  - Because when an object has no references, it can not be used

Reference Counting

- Advantages
  - Simple
  - Incremental
- Disadvantages
  - Frequent updates & Requires extra storage for each object
  - Can not detect cycles

Reachability Graph

- Instead of counting references
  - Keep track of some top-level objects
  - Trace out the reachability of objects
  - Only cleanup heap when out of space
    - Much better for low-memory programs
- Two major types of algorithms
  - Mark and Sweep
  - Copy Collectors

Reachability Graph

- Top-level objects (root)
  - Managed by CLR
  - Local variables on stack
  - Registers pointing to objects
- Garbage collector starts top-level
  - Builds a graph of 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?
- Completeness?

Copy Collectors

- Instead of just marking as we trace
  - Copy each reachable object to a new part of heap
  - Need enough space to do this
  - No need for second pass
- Advantage
  - One pass
- Disadvantage
  - Higher memory requirement

Compacting Copy Collectors

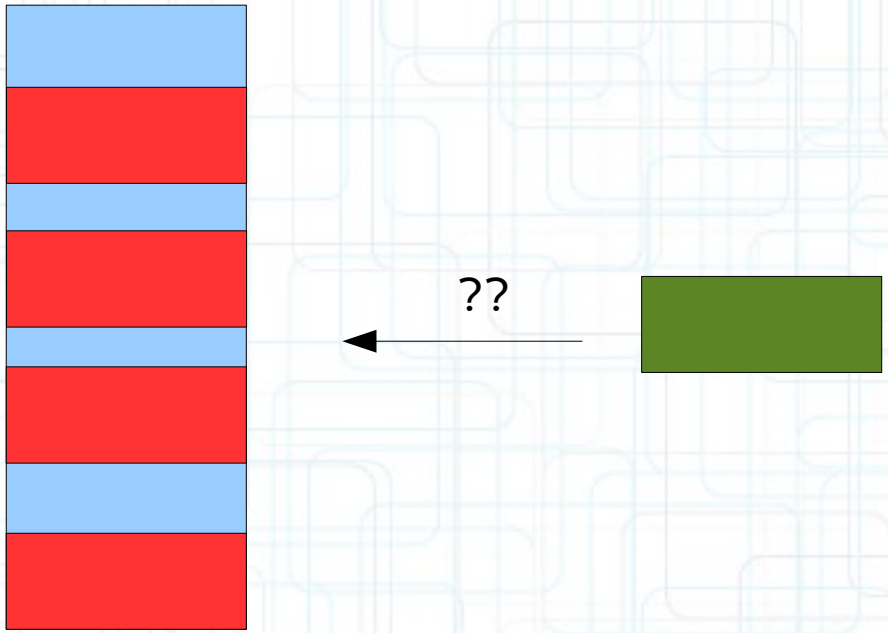
- Move live objects to bottom of heap
  - Leaves more free space on top
  - Contiguous allocation allows faster access
    - Cache works better with locality
- Must then modify references
  - Recall: references are really pointers
  - Must update location in each object

Compacting Copy Collectors

- 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 do not need to find space in old side
  - Contiguous allocation is easy

Fragmentation

- Common problem in memory schemes
- Enough memory but not enough contiguous



Heap Allocation Algorithms

- Best-fit
  - Search the heap for the closest fit
  - Takes time
- First-fit
  - Choose the first fit found starting from beginning of heap
- Next-fit
  - Just like first fit, but starts searching from last place found