Lecture 15

Garbage Collection

I. Introduction to GC
   -- Reference Counting
   -- Basic Trace-Based GC

II. Copying Collectors

III. Break Up GC in Time (Incremental)

IV. Break Up GC in Space (Partial)

Readings: Ch. 7.4 - 7.7.4
I. What is Garbage?
Two Approaches to Garbage Collection

What is not reachable, cannot be found! Needs to find the complement of reachable objects. Cannot collect a single object until all reachable objects are found! **Stop-the-world garbage collection!**

Catch the transition from reachable to unreachable. **Reference counting**
When is an Object not Reachable?

- **Mutator (the program)**
  - New / malloc: (creates objects)
  - Store: q = p; p->o1, q->o2
    - +: q->o1
    - -: If q is the only ptr to o2, o2 loses reachability

More?

- Load
  - Procedure calls
    - on entry: + formal args -> actual params
    - on exit: + actual arg -> returned object

More?

- **Important property**
  - once an object becomes unreachable, stays unreachable!
Reference Counting

- Free objects as they transition from “reachable” to “unreachable”
- Keep a count of pointers to each object
- Zero reference -> not reachable
  - When the reference count of an object = 0
    - delete object
    - subtract reference counts of objects it points to
    - recurse if necessary
- Not reachable -> zero reference?

answer?

- Cost
  - overhead for each statement that changes ref. counts
Why is Trace-Based GC Hard?

• Reasons
  – Requires complementing the reachability set - that’s a large set
  – Interacts with resource management: memory
Trace-based GC

• **Reachable objects**
  – Root set: (directly accessible by prog. without deref'ing pointers)
    • objects on the stack, globals, static field members
  – + objects reached transitively from ptrs in the root set.

• **Complication due to compiler optimizations**
  – Registers may hold pointers
  – Optimizations (e.g. strength reduction, common subexpressions) may generate pointers to the middle of an object
  – Solutions
    • ensure that a “base pointer” is available in the root set
    • compiler writes out information to decipher registers and compiler-generated variables (may restrict the program points where GC is allowed)
Baker’s Algorithm

- **Data structures**
  - Free: a list of free space
  - Unreached: a list of allocated objects, not Reached, not Scanned
  - Unscanned: a work list: Reached, but not Scanned
  - Scanned: a list of scanned objects: Reached and Scanned

- **Algorithm**
  - Scanned = ∅
  - Move objects in root set from Unreached to Unscanned
  - While Unscanned ≠ ∅
    - move object o from Unscanned to Scanned
    - scan o, move newly reached objects from Unreached to Unscanned
  - Free = Free ∪ Unreached
  - Unreached = Scanned
Trace-Based GC: Memory Life-Cycle

Mutator runs

- free \(\rightarrow\) unreached

GC Tracing

Repeat until unscanned = \(\emptyset\)

- free \(\rightarrow\) unreached
- reached
- scanned \(\rightarrow\) unscanned

Objects scanned for new reachable objects

GC Done tracing

- free \(\rightarrow\) unreached
- unscanned
- scanned

found to be reached
When Should We GC?
Frequency of GC

• **How many objects?**
  – Language dependent, for example, Java:
    • all non-primitive objects are allocated on the heap
    • all elements in an array are individually allocated
    • “Escape” analysis is useful
      -- object escapes if it is visible to caller
      -- allocate object on the stack if it does not escape

• **How long do objects live?**
  – Objects die young

• **Cost of reachability analysis: depends on reachable objects**
  – Less frequent: faster overall, requires more memory
### Performance Metric

<table>
<thead>
<tr>
<th></th>
<th>Reference Counting</th>
<th>Trace-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space reclaimed</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Overall execution time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space usage</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Pause time</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Data locality</td>
<td>✓✓</td>
<td></td>
</tr>
</tbody>
</table>
II. Copying Collector

- **To improve data locality**
  - place all live objects in contiguous locations

- **Memory separated into 2 (semi-)spaces: From and To**

  - Allocate objects in one
  - When (nearly) full, invoke GC, which copies reachable objects to the other space.
  - Swap the roles of semi-spaces and repeat
Trace-Based GC: Memory Life-Cycle

<table>
<thead>
<tr>
<th>Mutator runs</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>free</td>
</tr>
<tr>
<td></td>
<td>reached</td>
</tr>
<tr>
<td></td>
<td>unreachable</td>
</tr>
</tbody>
</table>

GC

Tracing

Repeat until unscanned = ∅

found to be reached

reached

unreached

objects scanned

for new reachable objects

GC

Done tracing

unreached

unreached

scanned

free
Copying Collector Algorithm

- UnScanned = Free = Start of To space
- Copy root set of objects space after Free, update Free;
- While UnScanned ≠ Free
  - scan o, object at UnScanned
  - copy all newly reached objects to space after Free, update Free
  - update pointers in o
  - update UnScanned
III. Incremental GC

- Break up GC to reduce pause time: interleave GC with mutator
  - Trace reachability in multiple rounds: GC-mutator-GC-...
  - Collect identified garbage in the last round

Kinds of Objects (not memory placement)

After the first GC round

R: Reachable objects before the first round of GC
New: Objects created since the first round of GC
Lost: Objects that lose reachability since the first round of GC

Ideal = (R \cup New) - Lost
Ideal \subseteq \text{Answer} \subseteq (R \cup New)

Forward progress guaranteed
Effects of Mutation

\[ \text{Ideal} = (R \cup \text{New}) - \text{Lost} \subseteq \text{Answer} \subseteq (R \cup \text{New}) \]

- **Ideal**: Very expensive
- **Conservative Incremental GC**: May misclassify some unreachable as reachable
  - should not include objects unreachable before GC starts
  - guarantees that garbage will be eliminated in the next round
- **Forward progress guaranteed**
Algorithm Proposal 1

• **Initial condition**
  – Scanned, Unscanned lists from before

• **To resume GC**
  – Find root sets
  – Place newly reached objects in “unscanned list”
  – Continue to trace reachability without redoing “scanned” objects

• **Did we find all reachable objects?**
Error: A reachable object classified as unreachable

- **When GC runs again:** A previously unreached, but reachable, object (C) is pointed to only in scanned objects (A)

- **How it can happen:**
  - Before the mutator runs
    - p in an unscanned or unreached object (B) points to an unreached object in C.

  - When the mutator runs
    - p copied to a scanned object (A)
    - p is overwritten in the unscanned/unreached set (B)
Solution

- Intercept \( p \) in any of the three steps
- Treat pointee of \( p \) as “unscanned”

**How it can happen:**

- Before the mutator runs
  - \( p \) in an unscanned or unreached object (B) points to an unreached object in C.
  
  Read Barrier:
  remember loads of pointers from B objects pointing at C objects

- When the mutator runs
  - \( p \) copied to a scanned object (A)
  
  Write Barrier:
  remember stores of pointers into A objects pointing at C objects

  - \( p \) is overwritten in the unscanned/unreached set (B)
  
  Overwrite Barrier:
  remember values overwritten in B objects pointing to C objects
Efficiency of Different Barriers

- **Most efficient: Write barrier**
  - less instances than read barrier
  - includes less unreachable objects than over-write barriers
IV. Partial GC: Incremental in Space

- Reduces pause time by collecting a subset of garbage (in target area):
  - **Stable set**
  - **Target set**

- **Algorithm**
  - New “root set”
    - original root set + pointers from Stable to Target set
  - Change program to intercept all writes to Stable set

- **Never misclassify reachable as unreachable**
- **May misclassify unreachable as reachable**
Generational GC

• **Observation: objects die young**
  - 80-98% die within a few million instructions or before 1 MB has been allocated

• **Generational GC: collect newly allocated objects more often**

• **i-th generation**
  - Stable set: Partitions # > i
  - Target set: Partitions # <= i
  - New root set
    = original root set + all pointers from the stable set to the target set
  - Ignore pointers from target back to stable
Generational Garbage Collection

Partitions

1 is full, GC 1, 1 is full, GC 1, 1 is full, GC 1, 2 is full, GC 2
Generational GC

• **Algorithm**
  – Always allocates in partition 1
    • Good locality for newly created objects
  – Copy to ith generation only when 1,…, (i-1) fills up
  – GC of mature objects takes longer
    • Size of target set increases
    • Eventually a full GC is performed

• **Effectiveness**
  – Objects die young:
    GC time is spent on partitions that are mostly garbage

• **Correctness and precision**
  – Conservative: Never misclassify reachable as unreachable
  – May misclassify unreachable as reachable
    • when pointers in earlier generations are overwritten
    • eventually collect all garbage as generations get larger
Conclusions

• **Reference counting:**
  – Cannot reclaim circular data structures
  – Expensive

• **Trace-based GC:**
  find all reachable objects, complement to get unreachable
  – 4 states: free, unreached, unscanned, scanned
  – break up reachability analysis
    • in time (incremental)
    • in space (partial: generational)
General Lessons

• **Understanding the program behavior**
  – is key to improve the efficiency of garbage collection

• **GC addresses a universal problem: memory management**
  – Time is spent on GC research saves a lot of time for developers!

• **The importance of compilers + runtime systems!**