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?

• Ideal: Eliminate all dead objects

• In practice: Unreachable objects
When is an Object not Reachable?

- **Mutator (the program)**
  - New / malloc: (creates objects)
  - Store p in a pointer variable or field in an object

- Load
  - Procedure calls

- **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?

• 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 = Ø
  – Move objects in root set from Unreached to Unscanned
  – While Unscanned ≠ Ø
    • 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

<table>
<thead>
<tr>
<th>Mutator runs</th>
<th>free → new → unreached</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC Tracing</td>
<td>free → unreached</td>
</tr>
<tr>
<td>Repeat until</td>
<td>reached → scanned</td>
</tr>
<tr>
<td>unscanned =</td>
<td>found to be reached</td>
</tr>
<tr>
<td>∅</td>
<td></td>
</tr>
<tr>
<td></td>
<td>objects scanned</td>
</tr>
<tr>
<td></td>
<td>for new reachable objects</td>
</tr>
<tr>
<td>GC Done tracing</td>
<td>free → unreached</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>scanned</td>
</tr>
</tbody>
</table>
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
<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>Program 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

Mutator runs
- new
- unreached
- free

GC Tracing
- traced
- reached
- scanned
- unscanned
- free
- unreached

Repeat until unscanned = ∅
- found to be reached
- objects scanned for new reachable objects

Done tracing
- free
- unreachable
- scanned
Copying Collector (Continued)

- 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

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

\[
(R \cup \text{New}) - \text{Lost} \subseteq \text{Answer} \subseteq (R \cup \text{New})
\]
Effects of Mutation

- Reachable set changes as mutator runs
  - $R$: set of reachable objects before the mutator runs
  - Ideal: set of reachable objects at the end of the GC cycle
  - New: set of newly created objects
  - Lost: set of objects that become unreachable in the interim
  - Ideal $= (R \cup \text{New}) - \text{Lost}$

- 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

$$\text{Ideal} = (R \cup \text{New}) - \text{Lost} \subseteq \text{Answer} \subseteq (R \cup \text{New})$$
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?
Missed Reachable Objects

- All reaching pointers are found in “scanned objects”
- Requires the occurrence of a 3-step sequence in the mutator:

0. after a stage of GC

1. Load p = ptr from B to C

2. Store p in A

3. Store new pointer in B, overwriting value p
Solution

- Intercept p in any of the three-step sequence
- Treat pointee of p as “unscanned”

0. after a stage of GC

1. Load p = ptr from B to C
   Read barrier: remember all loads of pointers from B → C

2. Store p in A
   Write barrier: remember all stores of pointers A → C

3. Store new pointer in B, overwriting value p
   Overwrite barrier: remember all overwrites of pointer B → C
Efficiency of Different Barriers

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

- Reduces pause time by collecting only objects in the target area:

  - **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

- **ith generation**
  - new root set
    = original root set + all pointers from generations \( j > i \)
- **When 1st generation fills up**
  - GC copies reachable objects into 2nd generation, and so on.
Properties

• Never misclassify reachable as unreachable

• Misclassify unreachable as reachable
  – when pointers in earlier generations are overwritten
  – eventually collect all garbage as generations get larger

• Effective: time spent on objects that are mostly garbage

• GC of mature objects takes longer
  – Size of target set increases
  – Eventually a full GC is performed
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)