Sustainable Memory Use

Allocation & (Implicit) Deallocation (mostly in Java)
Preliminaries

Today’s lecture

Automatic memory management
→ Procedure-local memory management
→ “Garbage Collection”
  → Reference Counting
  → Copying Collectors
  → Short-Pause Collectors
→ How you should program defensively now that you know

Second part: Java memory model, allocation, & recycling
→ How it works
→ How it affects your code’s runtime
→ How you should program defensively now that you know
Where do objects live in Java?

**COMP 140 & COMP 215**

- Students encouraged to ignore the issue of where objects, variables, and methods live.
- The implementation (Python or Java) takes care of these details.
- Fundamentally, abstraction is a good thing — *right up to the point where it causes problems*.
- At some point in your Java career, performance will matter.
  - COMP 412 labs
  - At that point, you need to pay attention to details.
- Today’s lecture is about details.
Where do objects live?

The Java System maps Java World onto Processor Resources

- Processor has finite resources
- Java suggests that you have “enough” resources
- Mapping “enough” onto “what’s there” is the job of the Java compiler and runtime (JVM)

Knowing how that mapping works can help you understand the behavior of your programs, and suggest ways to improve the program’s behavior.
In the example, what needs storage?

- The two classes (Point & C)
- Point’s local members (x, y, & draw)
- C’s local members (s, t, & m)
- m’s local variables (a, b, & p)

A classic example
In the example, what needs storage?
- The two classes (Point & C)
- Point’s local members (x, y, & draw)
- C’s local members (s, t, & m)
- m’s local variables (a, b, & p)

Memory in the Java runtime is divided, broadly speaking, into a **Heap** and a collection of **Stacks**
- One heap per program (large)
- One stack per thread (smaller)
Conceptually, Java memory is laid out along these lines

- When running code creates a variable, it goes into the thread’s stack
- When running code creates a class or an object (e.g., with a `new`), it goes into the heap
- Code lives off to the left (might consider it part of the heap)

So, can a program run out of heap space? (too many news)
⇒ Yes. Emphatically yes

What happens?
⇒ The runtime system tries to recycle space on the heap
Sustainable Memory Management

When the heap runs out of space, the system copes

• Scours the heap looking for objects that are no longer of interest
  – Technical term is "live"
  – An object is considered live iff it can be reached from the running code

• Start from all the names in the running code
  – Variables are on the stack\(^1\)
  – Global names such as declared or imported classes
  – Each object on the stack has a declaration which reveals its structure
  – You can imagine chasing down chains of references to find all live objects\(^2\)
    \[\rightarrow\text{That’s how it was done for a long time…}\]

\(^1\) Locals of the current method are on the stack. Locals of the method that called it are below the current method on the stack. Locals of the method that called that method are below …, and so on. That’s why the runtime uses a stack

Garbage Collectors

Reference-Counters

Stop-the-World

Mark-and-Sweep
- Basic
- Baker’s

Mark-and-Compact
- Basic
- Cheney’s

Trace-Based

Short-Pause

Incremental
- Generational
- Train

Partial
Reference Counting

<table>
<thead>
<tr>
<th>Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple implementation</td>
</tr>
<tr>
<td>Conservative</td>
</tr>
<tr>
<td>Easy to adapt to real-time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fragmentation</td>
</tr>
<tr>
<td>Self-referencing garbage</td>
</tr>
<tr>
<td>Locality effects</td>
</tr>
<tr>
<td>Overhead per pointer modification</td>
</tr>
<tr>
<td>Memory overhead</td>
</tr>
</tbody>
</table>
Reference Counting

Pros
- Simple implementation
- Conservative
- Easy to adapt to real-time

Cons
- Fragmentation
- Self-referencing garbage
- Locality effects
- Overhead per pointer modification
- Memory overhead
Reference Counting

Pros
Simple implementation
Conservative
Easy to adapt to real-time

Cons
Fragmentation
Self-referencing garbage
Locality effects
Overhead per pointer modification
Memory overhead
Reference Counting

**Pros**
- Simple implementation
- Conservative
- Easy to adapt to real-time

**Cons**
- Fragmentation
- Self-referencing garbage
- Locality effects
- Overhead per pointer modification
- Memory overhead
Taxonomy

Garbage Collectors

Reference-Counters

Stop-the-World

Mark-and-Sweep

Basic

Baker’s

Mark-and-Compact

Basic

Cheney’s

Trace-Based

Short-Pause

Incremental

Partial

Generational

Train
Mark and Sweep

1. **Free** = not holding an object; available for allocation.
2. **Unreached** = Holds an object, but has not yet been reached from the root set.
3. **Unscanned** = Reached from the root set, but its references not yet followed.
4. **Scanned** = Reached and references followed.
Marking

1. Assume all objects in Unreached state.
2. Start with the root set. Put them in state Unscanned.
3. **while Unscanned objects remain do**
   - examine one of these objects;
   - make its state be Scanned;
   - add all referenced objects to Unscanned
     - if they have not been there;

end;
Sweeping

• Place all objects still in the Unreached state into the Free state.
• Place all objects in Scanned state into the Unreached state.
  – To prepare for the next mark-and-sweep.

• Handles garbage cycles properly
Mark and Sweep

• **Problem**: Takes time proportional to the heap size.
  – Because you must visit all objects to see if they are Unreached.
• Baker’s algorithm keeps a list of all allocated chunks of memory, as well as the Free list.
• **Key change**: In the sweep, look only at the list of allocated chunks.
• Those that are not marked as Scanned are garbage and are moved to the Free list.
• Those in the Scanned state are put in the Unreached state.
  – For the next collection.
Taxonomy

Garbage Collectors

Reference-Counters

Stop-the-World

- Mark-and-Sweep
- Basic
- Baker’s

Trace-Based

- Mark-and-Compact
- Basic
- Cheney’s

Short-Pause

- Incremental
- Partial

Generational

Train
Issue: Why Compact?

- **Compact** = move reachable objects to contiguous memory.
- **Locality** --- fewer pages or cache-lines needed to hold the active data.
- **Fragmentation** --- available space must be managed so there is space to store large objects.
Basic Mark-and-Compact

1. Mark reachable objects as before.
2. Maintain a table (hash?) from reached chunks to new locations for the objects in those chunks.
   - Scan chunks from low end of heap.
   - Maintain pointer *free* that counts how much space is used by reached objects so far.
3. Move all reached objects to their new locations, and also retarget all references in those objects to the new locations.
   - Use the table of new locations.
4. Retarget root references.
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Example: Mark-and-Compact
Garbage Collectors

Reference-Counters
- Mark-and-Sweep
  - Basic
  - Baker’s

Trace-Based
- Stop-the-World
  - Mark-and-Compact
    - Basic
    - Cheney’s
- Short-Pause
  - Incremental
  - Partial
  - Generational
  - Train
Garbage Collection via Copying

Copying Collectors

• A **copying collector** divides the heap into two or more pools
• New objects are allocated in the current pool
• When the current pool is full, execution pauses and the collector:
  – copies all live objects from the current pool to the empty pool
  – swaps the designations current and empty

Unreachable objects are not copied, so the new pool has free space
Garbage Collection via Copying

Copying Collectors

• A **copying collector** divides the heap into two or more pools
• New objects are allocated in the current pool
• When the current pool is full, execution pauses and the collector:
  – copies all live objects from the current pool to the empty pool
  – swaps the designations current and empty

Unreachable objects are not copied, so the new pool has free space

Objects left in the empty pool are discarded en masse

### Before

<table>
<thead>
<tr>
<th>Current Pool</th>
<th>Empty Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 4 5 3</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

### After

<table>
<thead>
<tr>
<th>Current Pool</th>
<th>Empty Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td></td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
Cheney’s Copying Allocator

A shotgun approach to GC.
2 heaps: Allocate space in one, copy to second when first is full, then swap roles.
Maintain table of new locations.
As soon as an object is reached, give it the next free chunk in the second heap.
As you scan objects, adjust their references to point to second heap.
Advantages:
  O(live) complexity
  No unreachable garbage
  Very fast allocation
Issues:
  Only ½ memory can be used
  “Stop the world” to perform garbage collection
  Objects have to be “movable”
The Object Life-Cycle

• “Most objects die young.”
  – But those that survive one GC are likely to survive many.

• Tailor GC to spend more time on regions of the heap where objects have just been created.
  – Gives a better ratio of reclaimed space per unit time.
Garbage Collectors

Reference-Counters

Stop-the-World
- Mark-and-Sweep
  - Basic
  - Baker’s

Mark-and-Compact
- Basic
- Cheney’s

Trace-Based

Short-Pause
- Incremental
- Partial
  - Generational
  - Train
Generational Garbage Collection

• Divide the heap into partitions \( P_0, P_1, \ldots \)
  – Each partition holds older objects than the one before it.
• Create new objects in \( P_0 \), until it fills up.
• Garbage collect \( P_0 \) only, and move the reachable objects to \( P_1 \).
• When \( P_1 \) fills, garbage collect \( P_0 \) and \( P_1 \), and put the reachable objects in \( P_2 \).
• In general: When \( P_i \) fills, collect \( P_0, P_1, \ldots, P_i \) and put the reachable objects in \( P(i+1) \).
Summary

• Reference counting: conservative, small pause, fragmented, cyclic garbage
• Mark & Sweep: no cyclic garbage but long pause, O(heap)
• Mark & Compact: no fragmentation, O(heap), long pause, needs movable objects
• Copying (Cheney’s): fast allocation, needs movable objects, wastes ½ heap, O(live), long pause
• Generational: short(er) pause, needs write barrier, can be used for real time
• Concurrent: needs thread synchronization
• Incremental: run in parallel with execution, needs synchronization, can be used for real time
• Hybrid: copying for the youngest generation, mark&compact for older ones
Implications for Programming

If you want performance, pay attention to garbage

• Collector locates live objects by walking out from variables
  – When you are done with an object, set the variable to NULL
  – Leaving the reference to the heap object will keep it live

• Storage can “leak”, or become un-recyclable
  – Leave a pointer to a large data structure on the stack, or in a global, ...
  – or forgotten in another object, that happens to be live
  – Leads to extra collections and, eventually, an out of memory error

This is the takeaway message!
Implications for Programming

If performance really matters, pay attention to size of the pool

– Java uses a variant of a generational copying collector
– All \textbf{new} objects are allocated into Eden
– Eden is copied, when full, into one of Stable\textsubscript{0} or Stable\textsubscript{1}
– When Stable is too full, it is added to the Long Term Pool
Does this stuff matter?

Performance of One Student’s Java Code, COMP 412, Lab 1

- Standard
- Big Heap
- Small Heap

way too many collections

major collection (2x in speed)