Managed runtimes & garbage collection

Managed runtimes
Advantages?
• Reliability
• Security
• Portability
• Performance?

Disadvantages?
• Performance? Esp. memory overhead, compilation overhead

Portability (& performance)
Java source → javac → Java bytecode → ?? → Native code

Portability (& performance)
Java source → javac → Java bytecode → Interpreter and/or JVM's JIT compiler(s) → Native code

Adaptive optimization
Speculative optimizations (e.g., to deal w/class loading)
Memory and type safety

```java
int[] a = new int[64];
...
a[82] = ...;
```

C: undefined behavior \(\rightarrow\) major source of security exploits (how?)
Java: throws ArrayIndexOutOfBoundsException
How? Instrumentation added by JIT compiler (or proved unnecessary)

Memory and type safety

```java
MyType a = new MyType();
...
*((void*)a + 16) = 42;
```

C++: Undefined behavior
Java: Pointer arithmetic isn't part of the language

Memory and type safety

```java
SomeType a = ...;
b = (IncompatibleType) a;
b.f = ...;
```

C: undefined behavior \(\rightarrow\) potential security exploit
Java: throws ClassCastException
How? Instrumentation added by JIT compiler (or proves unnecessary)
Memory and type safety

```cpp
MyType a = new MyType(); /* or: malloc(sizeof(MyType)) */
... delete a; /* or: free(a) */
... a.f = ...;
```

C++: Undefined behavior

Java: Garbage collection \(\rightarrow\) no explicit freeing

Memory and type safety

```cpp
MyType a = new MyType();
... delete a;
... a.f = ...;
```

Java: Garbage collection \(\rightarrow\) no explicit freeing

Memory and type safety

```cpp
MyType a = new MyType();
... delete a;
... a.f = ...;
```

Java: Garbage collection \(\rightarrow\) no explicit freeing

Memory and type safety

```cpp
MyType a = new MyType();
... delete a;
... delete a;
```

C++: Undefined behavior

Java: Garbage collection \(\rightarrow\) no explicit freeing

Memory and type safety

```cpp
MyType a = new MyType();
... delete a;
... delete a;
```

Java: Garbage collection \(\rightarrow\) no explicit freeing

MyType a = new MyType();
... o.f = null; /* Last pointer to a is lost */
Memory and type safety

MyType a = new MyType();
...
o.f = null; /* Last pointer to a is lost */

C++: Memory leak
Java: Garbage collection

How? Knows all reference types and all "roots"; approximates liveness via transitive reachability

Why garbage collection?
(And how does GC know what's garbage?)

Programmers bad at explicit freeing
See prior slides for common mistakes
Requires global reasoning
GC goes hand-in-hand with memory & type safety

GC over-approximates liveness via reachability

Memory Leaks in Deployed Systems

- Memory leaks are a real problem
  - Managed languages do not eliminate them

Memory Leaks in Deployed Systems

- Memory leaks are a real problem
  - Managed languages do not eliminate them
Memory leaks are a real problem
- Managed languages do not eliminate them

- Slow & crash real programs
- Unacceptable for some applications

Fixing leaks is hard
- Leaks take time to materialize
- Failure far from cause
GC basics

Two types (duals of each other):

- Reference counting
  - Work proportional to dead objects
  - Memory freed immediately
  - Cycles are problematic

- Tracing
  - Work proportional to live objects
  - Freeing postponed
  - Can be concurrent

How does tracing GC work?

When does it happen?

- Stop-the-world: safe points inserted by VM
- Concurrent
- Incremental

How many GC threads?

- Single-threaded
- Parallel

Reachability

- Compiler produces a stack-map at GC safe-points and Type Information Blocks
- Type Information Blocks: identify reference fields in objects for each type (class) in the program, a map

- Tracing collector (semispace, marksweep)
  - Marks the objects reachable from the roots live, and then performs a transitive closure over them
Reachability

- Tracing collector (semispace, marksweep)
  - Marks the objects reachable from the roots live, and then performs a transitive closure over them
  - All unmarked objects are dead, and can be reclaimed

Example GC algorithms

Semispaces

- Fast bump pointer allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live
Semispace

- Fast bump pointer allocation
- Requires copying collection
- Cannot incrementally reclaim memory, must free en masse
- Reserves 1/2 the heap to copy in to, in case all objects are live

Mark phase:
- copies object when collector first encounters it
- installs forwarding pointers
- performs transitive closure, updating pointers as it goes
**Semispace**

- **Mark phase:**
  - copies object when collector first encounters it
  - installs forwarding pointers
  - performs transitive closure, updating pointers as it goes
  - reclaims "from space" en masse
  - start allocating again into "to space"

**Marksweep**

- Free-lists organized by size
  - blocks of same size, or
  - individual objects of same size
  - Most objects are small < 128 bytes
Marksweep

- Allocation
  - Grab a free object off the free list

- No more memory of the right size triggers a collection

- Mark phase - find the live objects
- Sweep phase - put free ones on the free list

Mark phase
- Transitive closure marking all the live objects

Sweep phase
- sweep the memory for free objects populating free list
Marksweep

- Mark phase
  - Transitive closure marking all the live objects
- Sweep phase
  - sweep the memory for free objects populating free list

Heap Organization

What objects should we put where?
- Generational hypothesis
  - young objects die more quickly than older ones (Lieberman & Hewitt'83, Ungar'84)
  - most pointers are from younger to older objects (Appel'89, Zorn'90)
- Organize the heap in to young and old, collect young objects preferentially

Generational Heap Organization

- Divide the heap in to two spaces: young and old
- Allocate in to the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
  - if space n + n fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up,
  - collect both spaces
  - Generalizing to m generations
    - if space n < m fills up, collect n through n-1

<table>
<thead>
<tr>
<th>Young</th>
<th>to space</th>
<th>Old</th>
<th>from space</th>
</tr>
</thead>
</table>

Generalization to m generations:
- If space n < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
  - if space n < m fills up, collect n through n-1

Young                               Old

Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
  - if space n < m fills up, collect n through n-1

Young                               Old

Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
  - if space n < m fills up, collect n through n-1

Young                               Old

Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up
  - collect both spaces
  - Generalizing to m generations
  - if space n < m fills up, collect n through n-1

Young                               Old
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate in the young space
- When the young space fills up,
  - collect it, copying into the old space
- When the old space fills up,
  - collect both spaces - ignore remembered sets
  - Generalizing to m generations
    - if space n < m fills up, collect 1 through n-1

Generational Write Barrier

Unidirectional barrier:
- record only older to younger pointers
- no need to record younger to older pointers, since we never collect the old space independently
  - most pointers are from younger to older objects [Appel'89, Zorn'90]
- track the barrier between young objects and old spaces

How are scalar and array objects represented?

<table>
<thead>
<tr>
<th>TIB</th>
<th>GC</th>
<th>Locking</th>
<th>Hashing</th>
<th>field0</th>
<th>field1</th>
<th>field2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIB</td>
<td>GC</td>
<td>Locking</td>
<td>Hashing</td>
<td>length</td>
<td>element0</td>
<td>element1</td>
</tr>
</tbody>
</table>

Header bits specify: owner thread & depth
Biased locking avoids synchronization

How does locking work?

synchronized (o) {
  ...
}

8/16/2013
How does hashing work?

o.hashCode();

If calls Object.hashCode()

• Identity hash code
• Needs to always return same hash code
• Dedicated bits
• Based on address? What if object moves?

Other components

• Classloading
• Threads and synchronization
• Exception handling
• Weak references & finalizers
• Other questions about how language VMs work?