Managed runtimes & garbage collection

CSE 6341
Some slides by Kathryn McKinley
Managed runtimes

Advantages?

Disadvantages?
Managed runtimes

Advantages?
• Reliability
• Security
• Portability
• Performance?

Disadvantages?
• Performance? Esp. memory overhead, compilation overhead
Portability (& performance)

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

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

Java source

javac

Java bytecode

Interpreter and/or JVM’s JIT compiler(s)

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

Native code
Memory and type safety

int[] a = new int[64];
...

a[82] = ...;
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

MyType a = new MyType();
...
*((void*)a + 16) = 42;
Memory and type safety

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

C++: Undefined behavior

Java: Pointer arithmetic isn’t part of the language
Memory and type safety

SomeType a = ...;

b = (IncompatibleType) a;
b.f = ...;
Memory and type safety

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

MyType a = new MyType(); /* or: malloc(sizeof(MyType)) */
...  
delete a; /* or: free(a) */
...
...  
a.f = ...;
Memory and type safety

MyType a = new MyType();
...
delete a;
...
a.f = ...;
Memory and type safety

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

C++: Undefined behavior

Java: Garbage collection \(\rightarrow\) no explicit freeing
Memory and type safety

MyType a = new MyType();
...
delete a;
...
delete a;
Memory and type safety

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

C++: Undefined behavior

Java: Garbage collection $\rightarrow$ no explicit freeing
Memory and type safety

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 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 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

![Diagram showing memory entities as live, dead, and reachable]
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
  - Slow & crash real programs
Memory Leaks in Deployed Systems

- Memory leaks are a real problem
  - Managed languages do not eliminate them
  - Slow & crash real programs
  - Unacceptable for some applications
Memory Leaks in Deployed Systems

- 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?

Roots
- Local variables: registers & stack locations
- Static variables

Transitive closure

Memory & type safety ensure GC knows the roots and references exactly
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
- GC safe points: new(), method entry, method exit, & back-edges (thread switch points)
- Stack-map: enumerate global variables, stack variables, live registers -- This code is hard to get right! Why?
- Type Information Blocks: identify reference fields in objects
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 i (class) in the program, a map

\[
\text{TIB}_i \begin{bmatrix} 0 & 2 & 3 \end{bmatrix}
\]

\[
\begin{aligned}
globals & \quad \text{stack} & \quad \text{registers} \\
\{A, B, C\} & \quad \{\ldots\} & \quad \text{heap} \\
\end{aligned}
\]

\[
\begin{align*}
\text{r0} &= \text{obj} \\
\text{PC} \rightarrow \text{p.f} &= \text{obj} \\
\text{...}
\end{align*}
\]
Reachability

- 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

```
... r0 = obj
PC -> p.f = obj
... mark
```
Reachability

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

```
... 
mark

globals  stack  registers

A
B
C

r0 = obj
PC -> p.f = obj
... 
```

heap
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
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
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
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
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
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
Semispase

- **Mark phase:**
  - copies object when collector first encounters it
  - installs *forwarding pointers*
Semispace

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

![Diagram of Semispace](image)
Semispace

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

![Diagram of Semispace](image)
Semispase

- **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
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”
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”
Semispace

- Notice:
  - Fast allocation
  - Locality of contemporaneously allocated objects
  - Locality of objects connected by pointers
  - Wasted space

![Diagram of heap memory management]
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
Marksweep

- **Allocation**
  - Grab a free object off the free list

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>128</td>
</tr>
</tbody>
</table>

free lists

heap
Marksweep

- Allocation
  - Grab a free object off the free list
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
Marksweep

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

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
</table>

free lists

heap
Marksweep

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

| 4  | 8  | 12 | 16 | ...
|----|----|----|----|----
|    |    |    |    | ... |
|    |    |    |    |    |
| 128|    |    |    |    |

free lists

heap
Marksweep

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

```plaintext
| 4  | 8  | 12 | 16 | ...
|----|----|----|----|----
|    |    |    |    | 128
```

free lists
Marksweep

- Mark phase
  - Transitive closure marking all the live objects
- Sweep phase
  - sweep the memory for free objects populating free list
  - can be made incremental by organizing the heap in blocks and sweeping one block at a time on demand

<table>
<thead>
<tr>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>...</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>free lists</td>
<td></td>
<td>heap</td>
<td></td>
</tr>
</tbody>
</table>

...
Marksweep

- Space efficiency
- Incremental object reclamation
- Relatively slower allocation time
- Poor locality of contemporaneously allocated objects

**free lists**

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>
```

heap
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 into young and old, collect young objects preferentially

![Diagram showing young and old objects]

- Young to space
- Old from space
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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  to space  Old

Young  from space
Generational Heap Organization

- Divide the heap into 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 < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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 $n-1$

```
          ⚫⚫⚫⚫⚫
 to space  to space  from space
 Young     Old
```
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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
```

to space  to space  from space
Generational Heap Organization

• Divide the heap into two spaces: young and old
• Allocate into 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 $n-1$
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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

![Diagram showing young and old spaces](image)
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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
Generational Heap Organization

- Divide the heap into 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 < m fills up, collect n through n-1
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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></th>
<th>Young</th>
<th>Old</th>
</tr>
</thead>
<tbody>
<tr>
<td>to space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>to space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>from space</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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
```

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

Generational Heap Organization

- Divide the heap into 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 $n < m$ fills up, collect $n$ through $n-1$

```
Young                               Old
```

Diagram

- Arrows indicate movement from young to old and back.
- Young space on the left, Old space on the right.
Generational Heap Organization

- Divide the heap into two spaces: young and old
- Allocate into 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 n through n-1
Generational Heap Organization

- Divide the heap into 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 - ignore remembered sets
  - Generalizing to m generations
    - if space n < m fills up, collect 1 through n-1

![Diagram showing young and old spaces with arrows indicating data movement]
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
Generational Write Barrier

unidirectional boundary barrier

```plaintext
// original program
p.f = o;

// compiler support for incremental collection
if (p > barrier && o < barrier) {
    remset_nursery = remset_nursery U &p.f;
}
p.f = o;
```
Generational Write Barrier

Unidirectional

- 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]
  - most mutations are to young objects [Stefanovic et al.'99]
How are scalar and array objects represented?

- TIB
- GC
- Locking
- Hashing
- field0
- field1
- field2
- objRef
- TIB
- GC
- Locking
- Hashing
- length
- element0
- element1
- objRef
How does locking work?

synchronized (o) {
    ...
}

Header bits specify: owner thread & depth

Biased locking avoids synchronization
How does hashing work?

```java
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?