Runtime System

COMP 524: Programming Languages
Srinivas Krishnan
April 12, 2011
What is the Runtime System (RTS)?

Language runtime **environment**.

➡ **OS view**: RTS is part of the user program.

➡ But RTS was **not programmed by the language user**.

➡ The RTS is everything not part of the OS and not explicitly provided by the user (i.e., the program or 3rd party)
What is the Runtime System (RTS)?

➡ **Language runtime environment.**

➡ **OS view:** RTS is part of the user program.

➡ RTS was not programmed by the language user.

➡ The RTS is everything not part of the OS and not explicitly provided by the user (i.e., the program or 3rd party).

**Examples:** memory allocator, **garbage collector**, support for runtime casts, exception handling infrastructure, **just-in-time (JIT) compiler**, support for closure and anonymous functions, lazy evaluation, dynamic type checking, byte code verifier, OS abstraction layers (if any), class-loading and plugin support (if any), multi-threading support, remote procedure calls (e.g., Java RMI), …

➡ The RTS is everything not part of the OS and not explicitly provided by the user (i.e., the program or 3rd party).
What is the Runtime System (RTS)?

**RTS:**
the infrastructure required to (transparently) realize higher-level language abstractions at runtime.

provided by the user (i.e., the program or 3rd party)
Our Focus

We’ll discuss three RTS components.

➡ Garbage collection.

➡ Just-in-Time Compilation.

➡ Security issues.
Heap Management

Allocation and deallocation of objects on the heap.

- **Arbitrary** object lifetime.

- Traditional language design:
  - Code, static, and runtime stack managed by compiler / interpreter.
Garbage

Memory reclamation.

- An object is “garbage” if it is not going to be used again.
- Memory holding garbage must be reclaimed in long-running programs.

Classic imperative approach: explicit heap management.

- malloc/free, new/delete, etc.
- Problems: dangling pointers, memory leaks…
- Experience suggests that programmers, on average, are not very good at correctly identifying garbage.
Garbage Collection

**Automatic heap management.**
- The RTS should manage memory, not the programmer.
- First developed for **Lisp** in 1958
- Merits hotly contested until ‘90ies.

**Widespread use.**
- Essential in **functional languages**
  - e.g., Haskell, ML.
- Key feature of **scripting languages**
  - e.g., Python, Perl.
- **Increasingly popular** modern imperative
Reachable Objects

Root Set
The set of objects that are immediately available to a program without following any pointers/references.

Object graph.
- Allocated objects form a graph.
  - Vertices: objects.
  - Edges: references/pointers.
- Any non-garbage object must be reachable.
Garbage Collection: Techniques

Detecting garbage.
» When is an object no longer being referenced?
» False positives: program crash.
» False negatives: memory leak.

Garbage collection techniques.
» Reference counting.
» Mark-and-sweep collection.
» Store-and-copy.
Reference Counting

**Indirect reachability.**

- Each object has an **associated reference counter**.
- Object graph: how many incoming edges?

**Maintained invariant.**

- Counter is incremented when a new reference is acquired.
- Counter is decremented when a reference is removed.
- **If** an object is reachable, **then** its associated reference counter is positive.

**Widespread use.**

- Easy to implement in C (but error-prone).
Reference Counting Example

str1 = “foo”
Reference Counting Example

After object allocation: reference counter is initially one.

str1 = "foo"
Reference Counting Example

str2 = str1
Reference Counting Example

Adding a new reference increments the counter.

str2 = str1
Each object has an associated reference counter

Keeps reference counter up to date, and deallocates objects when the counter is zero

\[
\begin{align*}
\text{str1} &= \text{None} \\
\text{str2} &= \text{None}
\end{align*}
\]
Reference Counting Example

No remaining references: it is now safe to deallocate the object.

str1 = None
str2 = None
Reference Counting: Problems

Efficiency.

➡ *Increases number* of (slow) *writes*.
➡ With multithreading, it may require (even slower) *atomic updates*.

Accuracy.

➡ *Disjoint union types*: what if one variant contains a reference, and another doesn’t?
   ‣ Reference counting must track variant tags.
➡ In a weakly typed language such as C?
   ‣ Cannot reliably tell *pointers* from *integers* apart.
➡ Cannot detect *circular* garbage.
Cycles in the Object Graph

```
stooges
Stack

Heap
2 "larry" → 1 "moe"

Heap
1 "curly"
```

```
stooges
Stack

Heap
1 "larry" → 1 "moe"

Heap
1 "curly"
```
Memory leak: not reachable, but will not be deallocated.
Mark and Sweep GC

Direct reachability.

- Instead of using a counter to track possible incoming paths, actually discover all paths at runtime by traversing the object graph.

- Anything not visited must be garbage.

- Every objects carries an “in-use” flag.

Algorithm concept.

- Mark every object in the heap as unreachable by clearing all “in-use” flag.

- Starting from the root set, traverse all references.

- Mark every visited object as reachable by setting its flag.

- Reclaim all unused objects (“sweep”).

- Run when memory is “low”.

Mark and Sweep: Challenges

“Stop the world” GC.

➡ What if object graph is changed during traversal?
➡ Simple solution: program execution is halted during GC.
  ‣ Can cause noticeable pauses.
Mark and Sweep: Challenges

“Stop the world” GC.

- What if object graph is changed during traversal?
- Simple solution: program execution is halted during GC.
  - Can cause noticeable pauses.

**Concurrent Garbage Collector:**
GC and program can run concurrently (i.e., any interleaving is acceptable).

**Incremental Garbage Collector:**
GC does not process whole object graph at once. Instead, it is invoked more frequently.
Mark and Sweep: Challenges

Identifying objects.

➡ How to identify objects in the heap?

› Must carry size/type tags, or have uniform size.

› Alternative: allocate objects of equal size/type from specific address ranges.

› Sometimes called “Big Bag of Pages” (BIBOP).

➡ How to discern arbitrary values from pointers?

› Could have a number that “points” to a garbage object.

› Could have a number that “points” outside of heap bounds.
Mark and Sweep: Challenges

Identifying objects.

- How to identify objects in the heap?
  - Must carry size/type tags, or have uniform size.
  - Alternative: allocate objects of equal size/type from specific address ranges. Sometimes called "Big Bag of Pages" (BIBOP).

- How to discern arbitrary values from pointers?
  - Could have a number that "points" to a garbage object.
  - Could have a number that "points" outside of heap bounds.

**Precise Garbage Collector:**
GC can unambiguously determine whether a given value is a pointer/reference.

**Conservative Garbage Collector:**
works without discerning pointers/reference from other values with certainty.
Mark and Sweep: Challenges

Memory requirements.

- GC algorithm runs when memory is scarce.
- Graph traversal requires memory itself!
  - Proportional to the longest path in the object graph.
  - Reserves are wasteful…

Tradeoff.

- Implementation complexity vs. efficiency.
- Could use incremental GC to reduce problem.
- Specialized stack-less techniques exist.
Mark & Sweep vs. Ref. Counting

Reference counting.

➡ Occurs continuously: no pauses.
  ‣ But: overheads are incurred continuously, too.
➡ Leaks circular structures.
➡ Relatively easy to implement.

Mark & Sweep.

➡ Difficult to implement efficiently.
  ‣ Esp. avoiding “stop the world”.
➡ Pauses, but otherwise fast execution and allocation.
➡ With precise GC, no leaking of unreachable objects.
Copying Garbage Collection

Partitioned heap.
- Two **arenas**: live objects arena and free space.
- Allocate from live object area until full.
- Then mark&sweep to find all live objects.
- Copy all live objects to free space.
  - **Fast consecutive allocation**.
- Switch roles: formerly live arena is now free.

simply identifying and freeing garbage doesn't solve fragmentation
Copying Garbage Collection

Live Arena

Free Arena
Copying Garbage Collection

Live Arena

Free Arena

Copy GC

Free Arena

Live Arena
Garbage doesn’t need to be explicitly reclaimed.
Copying Garbage Collection

Live Arena

Free Arena

Copy GC

Free Arena

Very fast allocation: no searching for available space.

Live Arena
Copying Garbage Collection

**Limitation**: Half of the heap is unused.
Generational GC

Generational Hypothesis.

➡ In many programs there is high “infant mortality.”
➡ Most objects are short-lived: they become garbage quickly after allocation.
➡ Thus, “older” objects are less likely to become garbage.

Arenas for different “ages”.

➡ Multiple allocation arenas.
➡ The “generation 0 arena” (the “nursery”) is used for new allocations.
➡ “Survivors” are copied to the next arena.
➡ Which is also gc’ed at some point, at which generation 1 objects move to the generation 2 arena, etc.
Generational GC

Generational Hyothesis.

Objects that are unlikely to be garbage are only examined infrequently: **reduced GC runtime**.

**New objects can be allocated very cheaply** from the nursery (simply increment the “end of last object” pointer).

Modern high-performance VMs often use this approach (e.g., Java Virtual Machine).

Which is also gc’ed at some point, at which generation 1 objects move to the generation 2 arena, etc.
GC vs. Manual Deallocation

Efficiency.

- Correct manual heap management is more efficient than naive GC.

- But software development cost considerations strongly favor GC.

- GC can be faster than manual management due to reduced allocation costs (copying GC).

Finalizers and non-memory resources.

- Languages such as Java use finalizers to free non-memory resources (such as file handles) when an object is freed.

- Problem: may run out of non-memory resources before GC kicks in.
Just-in-Time Compilation (JIT)

**Static compilation.**
- Compile time vs. run time.
- Compiler produces machine code once; resulting program is executed many times.

**Pure interpretation.**
- Interpreter evaluates syntax tree directly.
- Slow.

**Bytecode interpretation.**
- Source compiled to bytecode.
- Bytecode interpreted by VM.
- Still slower than statically compiled programs.
Just-in-Time Compilation (JIT)

**JIT**: compile byte code at run time to **speed up** overall program execution.

Compile time vs. run time:
- Compiler produces machine code **once**; resulting program is executed many times.

**Pure interpretation.**
- Interpreter evaluates syntax tree directly.
- **Slow**.

**Bytecode interpretation.**
- Source compiled to bytecode.
- **Bytecode interpreted by VM.**
- Still **slower** than statically compiled programs.
Just-in-Time Compilation (JIT)

Static compilation.
- Compile time vs. run time.
- Compiler produces machine code once; resulting program is executed many times.

Pure interpretation.
- Interpreter evaluates syntax tree directly.
- Slow.

Sometimes referred to as ahead-of-time compilation (AOT).
- Bytecode interpreted by VM.
- Still slower than statically compiled programs.
Idea and Limitations

“Write once, run anywhere.”

➡ Combine efficiency of compilation with flexibility of interpretation.

➡ “Late binding of machine code.”

➡ Java: web applets, mobile phones, embedded systems…

Overheads.

➡ Startup delay.

› After a program starts, parts must be compiled before output is produced, which can result in a noticeable delay.

› Hide by running interpreter and JIT compiler in parallel.

› Avoid compiling whole program at once.
JIT Overhead

Piecewise compilation.

- Program is compiled on demand in small chunks.
- Subroutine at a time, maybe even only parts of a subroutine.

Tradeoff.

- Compilation takes considerable time…
- …but compiled code is faster.
- Thus: compiled code must be executed many times to make tradeoff beneficial.

Threshold.

- Practical JIT systems trigger compilation only for code fragments that are executed more often than some threshold (e.g., 100 times).
- Intuition: focus on the common paths.
  - avoid initialization code and rare error paths
  - optimize main work loops
JIT Overhead

Piecewise compilation.

The exact threshold depends on the efficiency of the byte code interpreter and the JIT compilation speed and must be determined experimentally.

• Compilation takes considerable time...

• ...but compiled code is faster.

• Thus: compiled code must be executed many times to make tradeoff beneficial.

Threshold.

• Practical JIT systems trigger compilation only for code fragments that are executed more often than some threshold (e.g., 100 times).

• Intuition: focus on the common paths.

  ‣ avoid initialization code and rare error paths
Optimization vs. JIT Compilation

Simplicity wins.
- Only simple transformations.
- No “big picture” optimization.
- Fast, non-optimal algorithms instead of slower, provably better algorithms.

Goal: Lower Total Runtime

Fast Machine Code

Many Advanced Optimizations

Slower JIT C.

Increased Total Runtime

Higher JIT Threshold
Optimizations

Source Program

AOT Compiler

Byte Code

Input

VM + JIT

Output
The “heavy lifting”: intra-procedural analysis, common sub-expression analysis, dead code eliminations, flow analysis, polymorphism, etc.
**Simple transformations:**

basic byte code blocks to equivalent machine code.

Diagram:

1. Source Program
2. AOT Compiler
3. Byte Code
4. VM + JIT
5. Output

Input flows to VM + JIT, which produces the output.
JIT Advantages

Trace collection.
► Record execution statistics during interpretation.
► Can (re-)optimize at run time.

JIT can outperform AOT.
► **Additional information** available at run time.
  ‣ **Specific types** (instead of interfaces), accurate branch prediction.
► Can be used to **generate specialized code**.
  ‣ E.g., suppress error checking that is not needed for a **particular data set**.
► Additional **inlining** possibilities.
JIT Advantages

Trace collection.

- Record execution statistics during interpretation.
- Can (re-)optimize at run time.

JIT can outperform AOT.

- Additional information available at run time.

- Specific types (instead of interfaces), accurate branch prediction.
- Can be used to generate specialized code.
  - E.g., suppress error checking that is not needed for a particular data set.
- Additional inlining possibilities.

Tradeoff: long-running vs. short-running processes

Example: Java VM has a server mode that spends more time on aggressive optimizations.

branch prediction.

Tradeoff:
JIT and Prototype-Based Languages

Challenges.

- Java: JIT on class methods.
- What if there are no classes?

Tracing JIT.

- Derive “implicit” classes based on source code location where object was created (i.e., where the prototype was assigned).
- Most prototypes are not changed during run time.
- Must re-JIT an object if either
  - the object’s prototype is changed, or
  - a new prototype is assigned.
Binary Translation / Binary Rewriting

Compiling machine code to machine code.

- Either AOT or JIT.
- Basically a **compiler without source code**.

Uses.

- Debugging, logging (add invariant checking, etc.).
- Virtualization e.g. VMware ESX Server, Early IBM Servers
- **Performance analysis**.
- Adding security hooks.
  - Or **exploits**…
- **Legacy system emulation**.
  - E.g.: Apple’s **Rosetta**.
Security Issues

Untrusted code.

- Third party code that might be malicious.
- Often downloaded automatically via Internet.
  
  ✔ Embedded Javascript, Java applets, Flash, etc.
  
  ✔ Browser plugins.

Byte code validation.

- Proving arbitrary properties of arbitrary source code is impossible.
  
  ✔ Halting problem…

- Idea: allow only “known good” byte code.
  
  ✔ Be conservative.

Alternative.

- Code signing: attestation by trusted third party “this is ok.”
Security Issues

Untrusted code.

➡ Third party code that might be malicious.
➡ Often downloaded automatically via Internet.
› Embedded Javascript, Java applets, Flash, etc.
› Browser plugins.

Byte code validation.

➡ Proving arbitrary properties of arbitrary source code is impossible.
› Halting problem…

➡ Idea: allow only “known good” byte code.
› Be conservative.

Alternative.

➡ Code signing: attestation by trusted third party “this is ok.”

Java Track Record:
Many bugs and thus security vulnerabilities over the years.

Embedded javascript, Java applets, Flash, etc.
Security Issues

Untrusted code.

➡ Third party code that might be malicious.
➡ Often downloaded automatically via Internet.

‣ Embedded Javascript, Java applets, Flash, etc.

‣ Browser plugins.

Byte code validation.

➡ Proving *arbitrary properties* of arbitrary source code is impossible.

› Halting problem…

➡ Idea: allow only “known good” byte code.

› Be conservative.

Alternative.

➡ **Code signing**: attestation by trusted third party “this is ok.”

Example: **Microsoft-certified** Windows device drivers.