Modern Runtime Systems
(Java Virtual Machine and Microsoft Common Language Runtime)

James Riely

CTI Silver Anniversary Seminar Series
About me

- Six years at DePaul
- Teaching (SE/CSC):
  - Design patterns and OO development
  - Programming languages
  - Automata theory, semantics, formal security
- Research (National Science Foundation):
  - Programming language based access control
  - Aspect-oriented security policies
Modern Runtime Systems

- Classic runtime systems
- Modern challenges
- Modern solutions
  - Intermediate languages, interpreters and JITs
  - Class loaders
  - Type safety and bytecode verification
  - Security managers and stack inspection
  - Garbage collection

Foundational perspective (not a how-to lecture)
Classic Computer

- What’s our model of computation?
- Who is John von Neumann?
- What’s memory? A register?
- What’s a stack? A heap?
- What’s an opcode?
- What’s a process? A thread?
- What’s an operating system? A system call?
A Picture!
void f() {
    int i;
    int* ip = new int(42);
}

Stack / Heap
Classic Interpreter

Source Code

Pure Interpreter

Execution Results
Classic Compiler

COMPILER

lexical analyzer → syntax analyzer → semantic analyzer → code generator

file1.obj
Intermediate language code

file2.obj
Intermediate language code

LINKER

Execution Results

external libraries

file1.cpp

file2.cpp
Compilers vs Interpreters

- **Perform**ed by CPU (run-time)
- **Perform**ed by Compiler (off-line)

- **Perform**ed by CPU (run-time)
- **Perform**ed by Interpreter (run-time)

Diagram:
- Decode Program into Machine Language
  - All statements executed?
    - yes
      - Fetch next statement
      - Execute Statement
    - no
      - Repeat
- All statements executed?
  - yes
    - Return control to Operating System
  - no
    - Decode next program statement into Machine Language
      - Fetch next statement
      - Execute the Statement
Portability

- Windows  Linux MacOS
- PPC x86 i64
- Java C++ C# Basic
Compiler Interpreter Compiler
Local Optimization

- Can we inline second?

class Test {
    void first () { ... second() ... }
    void second () { ... }
}

Local Optimization

- Can we inline second?

class Test {
    void first () { ... second() ... }
    void second () { ... }
}

Class Test2 extends Test {
    void second () { ... }
}
Local Optimization

class Test {
    void f(Object x) { ... return x.toString(); } }  

- Since we never return to f, can we overwrite its stack frame (tail call elimination)
  - Some SecurityManagers make use of the call chain. Tail call elimination is only allowable (roughly) between classes having the same class loader.

- Local optimization is very difficult
JIT Phases

Figure 1: An overview of the JIT compiler
StarJIT for Java/CLI

Figure 1: StarJIT compiler architecture
Basic Problems

- No single authority has approved the code
  - Code not known until runtime!
- Plugins everywhere
  - Device drivers
  - Macros
  - Extensions
  - Scripts (including vector graphics and animations)
- Mutual distrust
Classic Computer Guarantees

- What’s a principal? Access control policy?
- What does a classic computer guarantee?
  - What’s the security model?
Modern Requirements

- Process separation good, but not portable
- Inefficient on some OS (e.g., Windows)
- Need protection within single address space
Modern Requirements

- Runtime protects underlying OS/hardware
- Security policy
  - Security manager
- Memory safety (AKA type safety)
- Classloader
  - Assigns code identity
  - Verifies bytecode
- Garbage Collector
CLR and JRE from 1000 ft

(JRE also enforces code type safety, but this is off by default for code loaded locally)
Java 1.1 Security

- Code loaded locally unchecked
- Code loaded from network run in “sandbox”
Policy Example

- As an example the sandbox policy may prohibit:
  - Reading writing deleting renaming local files / directories
  - Network connections
  - Creating new windows
  - Changing system settings
  - Loading dynamic libraries
  - Creating a new class loader or security manager
  - Creating classes that already exist

- Some typical policies:
  - No network communication to machine other than applet source
  - No send on network after read of local file
Java 1.2

- Much more expressive policies
- General notion of principal for both local and remote code
Stack inspection

- Buggy local code can be a problem
  - Local code that sends data across the network using two parameters (address/data)

- To prevent such obvious difficulties the Security Manager insists that all code on the call stack be trusted.
Stack Inspection Methods

- Backwards
- Forwards (Security Passing Style)
Shortcomings of Stack Inspection

- What about return values?
- What about data stored on the heap?
- History sensitive access control more general
  - Still a research topic
Type Safety

- Is it safe?
  int* ip = 0xAA50;
  print *ip;
- Is it safe?
  int[10] ia;
  print ia[-2058];
- Is it safe?
  int* ip = new int(42);
  delete ip;
  some_system_call();
  print *ip;
- Is it safe?
  int* f() {
    int i;
    return &i;
  }
The way to safety

- Bytecode uses abstract memory model
  - No notion of “memory address”
  - No explicit deallocation
  - No pointers into the stack

- Invariants enforced
  - Intermediate language definition – eg, no deallocation
  - IL verifier – eg, private/public respected
  - Interpreter/JIT – eg, array index in bounds

- Restricted access to system resources
  - check with security manager on every call
Theorems one would like

- Java is type safe.
  - The run-time types of local variables are compatible to the declared types.
- The Java compiler is correct.
  - The program semantics is preserved by the compilation.
- The Java bytecode verifier is correct.
  - Bytecode which is accepted by the verifier does not violate run-time checks (no type errors, no under- or overflow, no access to private fields, etc.)
- The Java compiler generates verifiable bytecode.
  - Bytecode from a legal Java program is accepted by the verifier.
What is type safe?

The program satisfies certain structural constraints (JVM Specification §4.8.2):

- The program counter is always a valid code index.
- No dangling pointers.
- No overflow nor underflow of the operand stack.
- Local variables are initialized before they are loaded.
- Primitive operations have enough operands.
- The operands are of the required type.
- Existing fields of objects are accessed only.
- The values of the fields of objects are compatible with the declared types.
- Arguments of method invocations are compatible with the declared types.
- No type violations at run-time.

Remark: No limits on memory and time consumption.
Bytecode Verification

WWW Browser

Internet

Server

JVM

Bytecode Verification
Bytecode Verification

javac.java

void main() {
    int i = 7;
    ...
}

Bytecode Programs

7: i
const_1
8: i
store 4

type safe
programs

accepted
by the
Verifier

Compiler
Bytecode Verification (Reality)

Java Programs

Compiler

Bytecode Programs

7: iconst_1
8: istore 4

type safe programs
accepted by the Verifier
How to make a bad bytecode

- Create X.java with a *public* method m
- Create Test.java that uses X.m
- Compile both
- Edit X.java to make method m *private*
- Recompile X (but not Test)
- Run Test! (Use java –Xverify:all)
public class Test1 {
    int test(boolean b) {
        int i;
        try {
            if (b) return 1;
            i = 2;
        } finally {
            if (b) i = 3;
        }
        return i;
    }
}

java version "1.3.0"
sun> javac Test1.java
sun> java Test1
java.lang.VerifyError: Register 2 contains wrong type
The Bytecode and analysis

int m(boolean b) {
    int i;
    try {
        if (b)
            return 1;
    }
    i = 2;
} finally {
    if (b)
        i = 3;
}
return i;

// 2 mod. by S

int m(boolean b) {
    int i;
    try {
        ifeq A (int) {1:int}
        iconst_1 () {1:int}
        istore_3 (int) {1:int}
        jsr S () {1:int,3:int}
        iload_3 () {1:int,3:int}
        ireturn (int) {1:int,3:int}
        A: iconst_2 () {1:int}
        istore_2 (int) {1:int}
        jsr S () {1:int,2:int}
        goto C () {1:int} // 2 mod. by S
        S: astore 4 (ra(S)) {1:int}
        iload_1 () {1:int,4:ra(S)}
        ifeq B (int) {1:int,4:ra(S)}
        iconst_3 () {1:int,4:ra(S)}
        istore_2 (int) {1:int,4:ra(S)}
        B: ret 4 () {1:int,4:ra(S)}
        C: iload_2 () {1:int}
        ireturn // 2 contains wrong type
Garbage Collection

Before

- Live Object (a)
- Dead Object (b)
- Live Object (c)
- Dead Object (d)
- Live Object (e)
- Dead Object (f)

Free Space

After

- Live Object (a)
- Live Object (c)
- Live Object (e)

Free Space
Garbage collection example

class List {
    field hd : Integer; field tl : List; field sz : Integer;
    method cons (hd : Integer) : List {
        return new List { hd=hd; tl=this; sz=this.sz+1; }
    }
    method snoc (last : Integer) : List {
        if (this.sz == 0) { return this.cons (last); }
        else { return this.tl.snoc (last).cons (this.hd); }
    }
    method reverse () : List {
        if (this.sz <= 1) { return this; }
        else { return this.tl.reverse ().snoc (this.hd); }
    }
    ...
}

object empty : List { hd=0; tl=empty; sz=0; }

thread Main {
    let x : List = empty.cons (10).cons (20).cons (30);
    let y : List = x.reverse ();
    // what can be gc‘d here?
    stdout.print ("x = " + $x + ", y = " + $y);
}
Garbage Collection

- The 1.3 JDK includes three different garbage collection strategies.
- The 1.4.1 JDK includes six, and over a dozen command-line options for configuring and tuning garbage collection.
- How do they differ? Why do we need so many?
Evaluating GCs

- **Pause time.** Does the collector stop the world to perform collection? How long?
- **Pause predictability.** Can garbage collection pauses be scheduled by user program?
- **CPU usage.** How much time overhead?
- **Memory footprint.** How much space overhead?
- **Virtual memory interaction.** Locality of reference?
- **Cache interaction.** Locality of reference?
- **Compiler and runtime support?** Does it interfere with compile-time optimizations?
Strategies

- Basic algorithms
  - Reference counting
  - Mark-sweep
  - Mark-compact
  - Copying
- Concurrent vs. stop-the-world
- Incremental vs. non-incremental
- Generational
Reference Counting

- Every copy of a reference must modify the reference count
- Predictable performance
- Used for ANSI C++ string class
- Does not handle recursive datatypes (lists)
Mark Sweep

- Easy to implement
- Every allocated object visited in sweep phase
Copying

- Heap divided into two equally sized semi-spaces, one of which contains active data and the other is unused.
- When the active space fills up, the world is stopped and live objects are copied from the active space into the inactive space.
- The roles of the spaces are then flipped, with the old inactive space becoming the new active space.
- Only visits live objects.
- Good for short-lived objects.
- Bad for long lived objects.
Generational Model

Simplified Model

- Generation 2 Objects
- Generation 1 Objects
- Generation 0 Objects
  - Committed (Ready to Use)
  - Uncommitted (Reserved)

Allocated Space

Free Space

Any generation can have some unreachable objects in it

Allocation Pointer
Generational model

- Level 0 is copying collector
- Object allocation cheaper than C!
- Little cost for objects that die young
- Expensive for old objects to reference new
  - Immutable programming is good
Thanks!
Implementing gc

- For each object on the heap, we need to know which fields are pointers, and which fields are base types. Why? How can we do this?
- For each frame on the stack, we need to know which fields are pointers. Why? How?
- For copying gc, where can we put the forwarding pointer?
- Note that this requires some help from the compiler. What can we do if the compiler won't play ball? (For example, can we gc C programs?)