Case Study: JVM

Virtual Machines

What is a machine?
- does something (...useful)
- programmable
- concrete (hardware)

Reality is somewhat fuzzy!
Is a Pentium-II a machine?

Hardware and software are logically equivalent (A. Tanenbaum)

Virtual Machine, Intermediate Language

- Pascal P-Code (1975)
  - stack-based processor
  - strong type machine language
  - compiler: one front end, many back ends
  - UCSD AppleJ implementation, PDP 11, Z80
- Modula M-Code (1980)
  - high code density
  - Lilith as microprogrammed virtual processor
  - Write Once – Run Everywhere
  - interpreters, JIT compilers, Hot Spot Compiler
- Microsoft .NET (2000)
  - language interoperability

JVM: Type System

- Primitive types
  - byte
  - short
  - int
  - long
  - float
  - double
  - char
  - reference
  - boolean mapped to int

- Object types
  - classes
  - interfaces
  - arrays

- Single class inheritance
- Multiple interface implementation

- Arrays
  - anonymous types
  - subclasses of java.lang.Object

JVM: Java Byte-Code

Memory access
- load / store
- iload / istore
- const
- getfield / putfield
- getstatic / putstatic

Operations
- iadd / isub / imul / idiv
- ishrts

Conversions
- f2i / f2f / f2d / ...;
- dup / dup2 / dup_x1 / ...

Control
- ifeq / ifne / iflt / ...;
- if_icmpeq / if_acmpeq
- invokestatic
- invokevirtual
- invokeinterface
- athrow
- ireturn

Allocation
- new / newarray

Casting
- checkcast / instanceof
**JVM: Java Byte-Code Example**

*bipush*

**Operation** Push byte

**Format**

```
bipush
byte
```

**Forms**

*bipush* = 16 (0x10)

**Operand Stack**

```
... => ..., value
```

**Description**

The immediate byte is sign-extended to an int value. That value is pushed onto the operand stack.

---

**JVM: Machine Organization**

**Virtual Processor**

- stack machine
- no registers
- typed instructions
- no memory addresses, only symbolic names

**Runtime Data Areas**

- pc register
- stack
  - locals
  - parameters
  - return values
- heap
- method area
  - code
- runtime constant pool
- native method stack

---

**JVM: Execution Example**

![Diagram](system-software-ws-04-05-338.png)

```
iload 5
iload 6
iadd
istore 4

iload 5
iload 6
iadd
v5 = v5 + v6
v6
v5
locals

 Time

operand stack

v4
v5
v6
```

**JVM: Reflection**

Load and manipulate *unknown* classes at runtime.

- java.lang.Class
  - getFields
  - getMethods
  - getConstructors
- java.lang.reflect.Field
  - setObject
  - getFloat
- java.lang.reflect.Method
  - getModifiers
  - invoke
- java.lang.reflectConstructor

---

**JVM: Reflection – Example**

```java
import java.lang.reflect.*;

public class ReflectionExample {
    public static void main(String[] args) {
        try {
            Class c = Class.forName(args[0]);
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++) {
                System.out.println(m[i].toString());
            }
        } catch (Throwable e) {
            System.err.println(e);
        }
    }
}
```

---

**JVM: Java Weaknesses**

Transitive closure of java.lang.Object contains

- 1.1  47
- 1.2  178
- 1.3  180
- 1.4  248
- 5 (1.5)  280
- classpath 0.03 299

```
Class Object {
   public String toString()
}
Class String {
   public String toUpperCase(Locale loc)
   public final class Locale implements Serializable, Cloneable {
     ....
   }
}
```
**JVM: Java Weaknesses**

Class static initialization
- T is a class and an instance of T is created
  
  ```java
  T tmp = new T();
  ```
- T is a class and a static method of T is invoked
  
  ```java
  T.staticMethod();
  ```
- A nonconstant static field of T is used or assigned
  (field is not static, not final, and not initialized with compile-time constant)
  
  ```java
  T.someField = 42;
  ```

Problem
- circular dependencies in static initialization code

---

**JVM: Memory Model**

- The JVM specs define a *memory model*:
  - defines the relationship between variables and the underlying memory
  - meant to guarantee the same behavior on every JVM
- The compiler is allowed to reorder operation unless synchronized or volatile is specified.

---

**JVM: Reordering**

- read and writes to ordinary variables can be reordered.

```java
public class Reordering {
  int x = 0, y = 0;
  public void writer() {
    x = 1;
    y = 2;
  } //
  public void reader() {
    int r1 = y;
    int r2 = x;
  }
}
```

---

**JVM: Double Checked Lock**

Singleton

```java
public class SomeClass {
  private Resource resource = null;
  public Resource synchronized getResource() {
    if (resource == null) {
      resource = new Resource();
    }
    return resource;
  }
}
```
JVM: Double Checked Lock

Double checked locking

public class SomeClass {
    private Resource resource = null;
    public Resource getResource()
    {
        synchronized
        {
            if (resource == null)
            {
                resource = new Resource();
            }
        }
        return resource;
    }
}

JVM: Immutable Objects are not Immutable

- Immutable objects:
  - all types are primitives or references to immutable objects
  - all fields are final
- Example (simplified): java.lang.String
  - contains
    - an array of characters
    - the length
    - an offset
  - example: s = "abcd", length = 2, offset = 2, string = "cd"

String s1 = "/usr/tmp";
String s2 = s1.substring(4); //should contain "/tmp"

- Sequence: s2 is instantiated, the fields are initialized (to 0),
  the array is copied, the fields are written by the constructor.
- What happens if instructions are reordered?

JVM: Reordering Volatile and Nonvolatile Stores

- volatile reads and writes are totally ordered among threads
- but not among normal variables
- example

```java
volatile boolean initialized = false;
SomeObject o = null;
```

Thread 1

```java
o = new SomeObject;
initialized = true;
```

Thread 2

```java
while (!initialized) {
    sleep();
}
o.field = 42;
```

Java JVM: Execution

- Interpreted (e.g., Sun JVM)
  - bytecode instructions are interpreted sequentially
  - the VM emulates the Java Virtual Machine
  - slower
  - quick startup
- Just-in-time compilers (e.g., Sun JVM, IBM JikesVM)
  - bytecode is compiled to native code at load time (or later)
  - code can be optimized at compile time or later
  - quicker
  - slow startup
- Ahead-of time compilers (e.g., GCJ)
  - bytecode is compiled to native code offline
  - quick startup
  - quick execution
  - static compilation
JVM: Class File Format

```java
class HelloWorld {
    public static void printHello() {
        System.out.println("hello, world");
    }
}
```

JVM: Class File (Constant Pool)

```
1. String hello, world
2. Class HelloWorld
3. Class java/io/PrintStream
4. Class java/lang/Object
5. Class java/lang/System
6. Methodref HelloWorld.<init>()
7. Methodref java/lang/Object.<init>()
8. Fieldref java/io/PrintStream java/lang/System
9. Methodref HelloWorld
10. Methodref HelloWorld
11. Methodref HelloWorld
12. Methodref java/io/PrintStream.println(java/lang/String)
13. NameAndType <init> ()V
14. NameAndType println()
15. NameAndType printHello ()
```

JVM: Class File (Code)

```java
public static void main(String[] args) {
    HelloWorld myHello = new HelloWorld();
    myHello.printHello();
}
```

JVM: Compilation – Pattern Expansion

- Each byte code is translated according to fix patterns
  - easy
  - limited knowledge
- Example (pseudocode)

```java
switch (o) {  
    case ICONST0: generate("push n"); PC++; break;
    case ILOAD0: generate("push off_n[F]"); PC++; break;
    case IADD: generate("pop -> R1");
    generate("add R1, R2 -> R1");
    generate("push R1");
    PC++;
    break;
    default:
        break;
}
```
**JVM: Compiler Comparison**

- Pattern expansion:
  - `push off4[FP]`
  - `push off5[FP]`
  - `pop EAX`
  - `add 0[SP], EAX`
  - `pop off6[FP]`

- Optimized:
  - `mov EAX, off4[FP]`
  - `add EAX, off5[FP]`
  - `mov off6[FP], EAX`

- IL code:
  - `iload_4`
  - `iload_5`
  - `iadd`
  - `istore_6`

- Instructions: 5
- Memory accesses: 6

---

**Linking (General)**

- A compiled program contains references to external code (libraries)
- After loading the code the system need to link the code to the library
  - identify the calls to external code
  - locate the callees (and load them if necessary)
  - patch the loaded code
- Two options:
  - the code contains a list of sites for each callee
  - the calls to external code are jumps to a procedure linkage table which is then patched (double indirection)

---

**Linking (General)**

- Bytecode interpreter
  - references to other objects are made through the JVM (e.g., invokevirtual, getfield, …)
- Native code (ahead of time compiler)
  - static linking
  - classic native linking
- JIT compiler
  - only some classes are compiled
  - calls could reference classes that are not yet loaded or compiled (delayed compilation)
  - code instrumentation

---

**JVM: Methods and Fields Resolution**

- method and fields are accessed through special VM functions (e.g., invokevirtual, getfield, …)
- the parameters of the special call defines the target
- the parameters are indexes in the constant pool
- the VM checks if the call is legal and if the target is present!
JVM: JIT – Linking and Instrumentation

- Use code instrumentation to detect first access of static fields and methods

```java
class A {
    ...
    ...
    . . . . B.x
}
```

```java
class B {
    int x;
}
```

```java
B.x
CheckClass(B);
IF ~B.initialized
THEN
    Initialize(B)
END;
```

Compilation and Linking Overview

Compilation and Linking Overview

Compilation and Linking Overview

Compilation and Linking Overview

Jaos (Java on Active Object System) is a Java virtual machine for the Bluebottle system

- Jaos (Interoperability Framework)

Jaos (Interoperability Framework)

Jaos (Interoperability Framework)

Jaos (Interoperability Framework)
Compiler generates "good" code... 
... that could be changed before reaching the JVM  
→ need for verification

Verification makes the VM simpler (less run-time checks):  
- no operand stack overflow  
- load / stores are valid  
- VM types are correct  
- no pointer forging  
- no violation of access restrictions  
- access objects as they are (type)  
- local variable initialized before load  
  ...  

Pass 1 (Loading):  
- class file version check  
- class file format check  
- class file complete

Pass 2 (Linking):  
- final classes are not subclassed  
- every class has a superclass (but Object)  
- constant pool references  
- constant pool names

Pass 3 (Linking):  
For each operation in code (independent of the path):  
- operation stack size is the same  
- accessed variable types are correct  
- method parameters are appropriate  
- field assignment with correct types  
- opcode arguments are appropriate

Pass 4 (RunTime):  
First time a type is referenced:  
- load types when referenced  
- check access visibility  
- class initialization  
First member access:  
- member exists  
- member type same as declared  
- current method has right to access member

Branch destination must exist  
Opcode must be legal  
Access only existing locals  
Code does not end in the middle of an instruction  
Types in byte-code must be respected  
Execution cannot fall of the end of the code  
Exception handler begin and end are sound

How to start a JVM?  
- External help needed!  
- Load core classes  
- Compile classes  
- Provide memory management  
- Provide threads

Solution:  
Implement Java on 3rd party system  
Linux  
Solaris  
Windows  
Bluebottle  
Java

All native methods in Active Oberon  
Use Bluebottle run-time structures  
- module (≡ class)  
- type descriptor  
- object (≡ object instance)  
- active object (≡ thread)

Bootstrap:  
- load core classes  
  - Object, String, System, Runtime, Threads, ...  
  - Exception  
- forward exception to java code  
- allocate java classes from Oberon

All native methods in Active Oberon  
Use Bluebottle run-time structures  
- module (≡ class)  
- type descriptor  
- object (≡ object instance)  
- active object (≡ thread)  
- Bootstrap
Bootstrapping: Jnode VM Example

- JVM written in Java
- small core in assembler
  - low-level functionalities that requires special assembler instructions
- some native methods inlined by the compiler
  - Unsafe
    - debug(String)
    - int AddressToInt(Address)
    - int getInt(Object, offset)
- Bootstrap
  - compile to Java classes
  - bootstrapper:
    - native compilation
    - code placement
    - structure allocation
    - make boot image
  - boot with GNU/GRUB

Bootstrapping: Oberon / Bluebottle

- Compile each module to machine code
  - system calls for
    - newrec (record)
    - newsys (block, no ptrs)
    - newarr (array)
  - linker / bootlinker patches syscalls with real procedures
- bootlinker is same as linker, but uses different memory addresses
  - simulates memory allocation
  - start address configurable
  - glue code to call all module bodies

Bootstrapping Compilers

- This compiler is written in language "C" and translates programs in "A" into programs in "B"
- language X is already available
- this compiler can be executed