The Java Language Implementation

Reading
- Chapter 13, sections 13.4 and 13.5
- Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches, pages 1–5.
Outline

• Java virtual machine overview
  – Loader and initialization
  – Linker and verifier
  – Bytecode interpreter
• JVM Method lookup
  – four different bytecodes
• Verifier analysis
• Method lookup optimizations (beyond Java)
• Java security
  – Buffer overflow
  – Java “sandbox”
  – Stack inspection
Java Implementation

• Compiler and Virtual Machine
  – Compiler produces bytecode
  – Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode

• Why this design?
  – Bytecode interpreter/compilers used before
    • Pascal “pcode”; Smalltalk compilers use bytecode
  – Minimize machine-dependent part of implementation
    • Do optimization on bytecode when possible
    • Keep bytecode interpreter simple
  – For Java, this gives portability
    • Transmit bytecode across network
Java Virtual Machine Architecture

Compile source code

A.java → Java Compiler → A.class

Java Virtual Machine

Loader → Verifier → Linker → Bytecode Interpreter

B.class

Network
JVM memory areas

- Java program has one or more threads
- Each thread has its own stack
- All threads share the same heap
Class loader

• Runtime system loads classes as needed
  – When class is referenced, loader searches for file of compiled bytecode instructions

• Default loading mechanism can be replaced
  – Define alternate ClassLoader object
    • Extend the abstract ClassLoader class and implementation
    • ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
  – Can obtain bytecodes from alternate source
    • VM restricts applet communication to site that supplied applet
Example issue in class loading and linking:

Static members and initialization

class ... {
    /* static variable with initial value */
    static int x = initial_value
    /* ---- static initialization block --- */
    static { /* code executed once, when loaded */ }  
}

• Initialization is important
  – Cannot initialize class fields until loaded
• Static block cannot raise an exception
  – Handler may not be installed at class loading time
JVM Linker and Verifier

• Linker
  – Adds compiled class or interface to runtime system
  – Creates static fields and initializes them
  – Resolves names
    • Checks symbolic names and replaces with direct references

• Verifier
  – Check bytecode of a class or interface before loaded
  – Throw VerifyError exception if error occurs
Verifier

• Bytecode may not come from standard compiler
  – Evil hacker may write dangerous bytecode
• Verifier checks correctness of bytecode
  – Every instruction must have a valid operation code
  – Every branch instruction must branch to the start of some other instruction, not middle of instruction
  – Every method must have a structurally correct signature
  – Every instruction obeys the Java type discipline

Last condition is complicated.
Bytecode interpreter

• Standard virtual machine interprets instructions
  – Perform run-time checks such as array bounds
  – Possible to compile bytecode class file to native code

• Java programs can call native methods
  – Typically functions written in C

• Multiple bytecodes for method lookup
  – `invokevirtual` - when class of object known
  – `invokeinterface` - when interface of object known
  – `invokestatic` - static methods
  – `invokespecial` - some special cases
Type Safety of JVM

• Run-time type checking
  – All casts are checked to make sure type safe
  – All array references are checked to make sure the array index is within the array bounds
  – References are tested to make sure they are not null before they are dereferenced

• Additional features
  – Automatic garbage collection
  – No pointer arithmetic

If program accesses memory, that memory is allocated to the program and declared with correct type
JVM uses stack machine

- **Java**
  
  ```java
  Class A extends Object {
      int i;
      void f(int val) { i = val + 1;}
  }
  ```

- **Bytecode**
  
  ```java
  Method void f(int)
  
  aload 0  ; object ref this
  iload 1   ; int val
  iconst 1
  iadd     ; add val +1
  putfield #4 <Field int i>
  return
  ```

  The JVM Activation Record refers to the constant pool.
Field and method access

• Instruction includes index into constant pool
  – Constant pool stores symbolic names
  – Store once, instead of each instruction, to save space

• First execution
  – Use symbolic name to find field or method

• Second execution
  – Use modified “quick” instruction to simplify search
Outline

• Java virtual machine overview
  – Loader and initialization
  – Linker and verifier
  – Bytecode interpreter

JVM Method lookup
  – four different bytecodes

• Verifier analysis

• Method lookup optimizations (beyond Java)

• Java security
  – Buffer overflow
  – Java “sandbox”
  – Stack inspection
Two cases in more detail

• Source code provides interface to object
  – Method lookup using Smalltalk-like search process
  – Cache last offset in case next lookup is same class

• Source code provides class or superclass
  – Method lookup uses Smalltalk-like search first time
    • Reason: run-time class loading; compiler doesn’t know representation of classes in different class files
  – Rewrite bytecode so that fixed offset on next lookup
invokeinterface <method-spec>

• Sample code
  ```java
  void add2(Incrementable x) { x.inc(); x.inc(); }
  ```

• Search for method
  – find class of the object operand (operand on stack)
    • must implement the interface named in <method-spec>
  – search the method table for this class
  – find method with the given name and signature

• Call the method
  – Usual function call with new activation record, etc.
Why is search necessary?

interface A {
    public void f();
}

interface B {
    public void g();
}

class C implements A, B {
    ...
}

Class C cannot have method f first and method g first
But if class instead of interface...

- Sample code
  ```java
  void deposit1(Account a) { a.deposit(1) ... }
  ```

- Class hierarchy
  ```java
class Account {
   public void deposit(int i);
}
class InterestAccount extends Account {
   ...
}
```

Single inheritance guarantees derived class vtable uses same order as base class vtable; remains true if class also implements many interfaces
invokevirtual <method-spec>

• Similar to invokeinterface, but class is known
• Search for method
  – search the method table of this class
  – find method with the given name and signature
• Can we use static type for efficiency?
  – Each execution of an instruction will be to object from subclass of statically-known class
  – Constant offset into vtable
    • like C++, but dynamic loading makes search useful first time
  – See next slide
Bytecode rewriting: invokevirtual

After search, rewrite bytecode to use fixed offset into the vtable. No search on second execution.
Bytecode rewriting: invokeinterface

Bytecode

invokeinterface

inv_int_quick

Constant pool

"A.foo()"

Cache address of method; check class on second use
Outline

• Java virtual machine overview
  – Loader and initialization
  – Linker and verifier
  – Bytecode interpreter

• JVM Method lookup
  – four different bytecodes

• Verifier analysis

• Method lookup optimizations (beyond Java)

Java security
  – Buffer overflow
  – Java “sandbox”
  – Stack inspection
Java Security

• Security
  – Prevent unauthorized use of computational resources

• Java security
  – Java code can read input from careless user or malicious attacker
  – Java code can be transmitted over network – code may be written by careless friend or malicious attacker

Java is designed to reduce many security risks
Java Security Mechanisms

• Sandboxing
  – Run program in restricted environment
    • Analogy: child’s sandbox with only safe toys
  – This term refers to
    • Features of loader, verifier, interpreter that restrict program
    • Java Security Manager, a special object that acts as access control “gatekeeper”

• Code signing
  – Use cryptography to establish origin of class file
    • This info can be used by security manager
Buffer Overflow Attack

- Most prevalent *general* security problem today
  - Large number of CERT advisories are related to buffer overflow vulnerabilities in OS, other code

- General network-based attack
  - Attacker sends carefully designed network msgs
  - Input causes privileged program (e.g., Sendmail) to do something it was not designed to do

- Does not work in Java
  - Illustrates what Java was designed to prevent
Sample C code to illustrate attack

```c
void f (char *str) {
    char buffer[16];
    ...
    strcpy(buffer,str);
}
void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    f(large_string);
}
```

• Function
  – Copies str into buffer until null character found
  – Could write past end of buffer, over function return addr

• Calling program
  – Writes 'A' over f activation record
  – Function f “returns” to location 0x4141414141
  – This causes segmentation fault

• Variations
  – Put meaningful address in string
  – Put code in string and jump to it !!

See: Smashing the stack for fun and profit
Java Sandbox

• Four complementary mechanisms
  – Class loader
    • Separate namespaces for separate class loaders
    • Associates protection domain with each class
  – Verifier and JVM run-time tests
    • NO unchecked casts or other type errors, NO array overflow
    • Preserves private, protected visibility levels
  – Security Manager
    • Called by library functions to decide if request is allowed
    • Uses protection domain associated with code, user policy
    • Coming up in a few slides: stack inspection
Security Manager

- Java library functions call security manager
- Security manager object answers at run time
  - Decide if calling code is allowed to do operation
  - Examine protection domain of calling class
    - Signer: organization that signed code before loading
    - Location: URL where the Java classes came from
  - Uses the system policy to decide access permission
## Sample SecurityManager methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkExec</td>
<td>Checks if the system commands can be executed.</td>
</tr>
<tr>
<td>checkRead</td>
<td>Checks if a file can be read from.</td>
</tr>
<tr>
<td>checkWrite</td>
<td>Checks if a file can be written to.</td>
</tr>
<tr>
<td>checkListen</td>
<td>Checks if a certain network port can be listened to for connections.</td>
</tr>
<tr>
<td>checkConnect</td>
<td>Checks if a network connection can be created.</td>
</tr>
<tr>
<td>checkCreateClassLoader</td>
<td>Check to prevent the installation of additional ClassLoaders.</td>
</tr>
</tbody>
</table>
Stack Inspection

- Permission depends on
  - Permission of calling method
  - Permission of all methods above it on stack
    - Up to method that is trusted and asserts this trust

- java.io.FileInputStream
- method h
- method g
- method f
Example: privileged printing

privPrint(f) = (* owned by system *)
{    
    checkPrivilege(PrintPriv);
    print(f);
}

foreignProg() = (* owned by Joe *)
{    
    ...; privPrint(file); ...;
}
Stack Inspection

• Stack frames are annotated with names of owners and any enabled privileges
• During inspection, stack frames are searched from most to least recent:
  – fail if a frame belonging to someone not authorized for privilege is encountered
  – succeed if activated privilege is found in frame
Stack Inspection

• Permission depends on
  – Permission of calling method
  – Permission of all methods above it on stack
    • Up to method that is trusted and asserts this trust

Many details omitted here

Stories: Netscape font / passwd bug; Shockwave plug-in
Outline

• Java virtual machine overview
  – Loader and initialization
  – Linker and verifier
  – Bytecode interpreter
• JVM Method lookup
  – four different bytecodes
• Verifier analysis
• Method lookup optimizations (beyond Java)
• Java security
  – Buffer overflow
  – Java “sandbox”
  – Stack inspection