Java Virtual Machine

part 1

One more architecture: the Java Virtual Machine

- JVM is a virtual machine that executes Java Byte Code (JBC)
  - start with Java source code (.java text file)
  - Java compiler (javac) creates JBC file with .class extension
    - standardized binary format
    - single program consists of one or more class files
    - multiple class files may be packaged together as a .jar file for program distribution

JVM execution environment

- JVM is represented by an executable program called java
  - emulates JVM instruction set
  - interpreter
  - JBC is stack based, so JVM uses stack architecture

Why a “virtual machine?”

- Idea didn’t originate with Java – idea comes from time-sharing systems (circa 1960s)
- VM advantages:
  - platform independence
  - transcends physical limits
  - ease of updates
  - security safeguards

Platform independence

- Java compiler is platform-independent: makes no assumptions about characteristics of underlying hardware
- JVM required to run Java byte code
- Works as a wrapper around a real machine’s architecture—so the JVM itself is extremely platform dependent

Java environment vs. “traditional” HLL environment

[Diagram]

- Source code (java or other HLL)
- Compiler/linker
- Java compiler
- Class file (.JBC)
- Machine language
- Java virtual machine
- Hardware platform
How it works

- Java compiler translates source code into JBC
- JVM acts as interpreter - translates specific byte codes into machine instructions specific to the harbor platform it’s running on
- Acts like giant switch/case structure: each bytecode instruction triggers jump to a specific block of code that implements the instruction in the architecture’s native machine language

JVM’s superpower: transcends physical limits

- No hardware costs (both $ and resource tradeoffs)
- Because of multithreading, can have (seemingly) unlimited processor power
- No backward compatibility issues
- Can be adapted to optimize hardware resources of specific platform
- Designed from scratch in mid-90s: several generations of engineering experience led to design superior to most physical chips

JVM and security issues

- A virtual machine can be (and JVM is) configured to run in secure environment
- VM can intervene if a program tries to do something it shouldn’t – can enforce stricter security policies than those of OS
- JVM bytecode is verifiable
  - most security flaws happen by accident
  - byte code is checked by both compiler and JVM
  - result: improved software quality & reliability

Downside of virtual machines

- Before Java, virtual machines relatively uncommon because of performance issues
- Takes about 1000 times longer to do an operation in software instead of hardware
  - hardware advances & compiler improvements mitigate this
  - in practical terms, speed difference is anywhere from 2x slower to more than about 6% slower
- VM doesn’t provide direct control over hardware available with native low-level language

Characteristics of JVM

- Not just a virtual machine; something like a virtual operating system
- Case in point: output statement printf()
  - Compiled C program calls the operating system’s write() function
  - Compiled Java program makes similar call, but to a JVM routine which then calls the “real” write() function

Java & threading

- Because Java is a virtual machine, it is free of some of the constraints of a real machine architecture
- Java threads exemplify this
  - separate processes running in parallel
  - simulates a multi-processor environment independent of actual platform
Characteristics of JVM

- Stack-based language & machine
- Each thread within a program has its own stack
- 32-bit word size
- Relatively small instruction set (about 200 instructions)

Characteristics of JVM: registers

- 4 registers (sort of):
  - program counter (PC)
  - optop: points to top of operand stack for currently-active method
  - frame: points to stack frame for current method
  - vars: points to start of local variables for current method
- Each program (or thread) has these, as well as its own stack

Characteristics of JVM: registers

- No general-purpose registers
  - means more memory fetches, detrimental to performance
  - tradeoff is high degree of portability
- Most instructions access stack

Characteristics of JVM: stack memory

- Each method call produces its own stack frame, which is pushed on the thread's stack; a return instruction pops the stack
- Stack frame includes:
  - local variables section
  - operand stack section

Local variables section of stack frame

- Consists of set of word-size slots, each of which holds a single variable; includes
  - parameters & locally-declared data, in order of declaration;
  - if method is non-static, first slot (slot 0) contains pointer to “this”

Operand stack section of stack frame

- Operand stack section is where method’s instructions operate - “the” stack referred to when talking about instructions operating on the stack
- Maximum depth of operand stack is determined at compile time
- Current stack depth is determined by number & type of operands on stack:
  - double and long values take up two slots
  - all other data types take one slot
JVM Method area

- Stores classes used by executing program; includes:
  - bytecode & access types of methods
  - values & access types of static variables
- PC points to this area – location of next instruction
- Method area also includes constant pool – storage for literal values used in program

JVM Heap

- Memory allocated for objects from this area
- Holds object’s instance values and pointer to object’s class in the Method area

JVM instruction set

- Instructions consist of one-byte opcode followed by 0 or more operands
- Instruction types include:
  - load/store of local variables & object fields
  - array
  - arithmetic and logical
  - type conversion
  - control
  - method call/return