High-Level Language VM – Outline

• Introduction
• Virtualizing conventional ISA Vs. HLL VM ISA
• Pascal P-code virtual machine
• OO HLL virtual machines
  – properties, architecture, terms
• Implementation of HLL virtual machine
  – class loading, security, GC, JNI
Introduction

• HLL PVM similar to a *conventional* PVM
  – V-ISA not designed for a real hardware processor

• Languages/compiler perspective of HLL VM
  – see figure 5.1

• Example
  – Pascal P-code
  – Java Virtual Machine
  – Common Language Infrastructure
Virtualizing Conventional ISA Vs. High-Level-Language VM ISA

• Drawbacks of virtualizing a conventional ISA
  – not developed for being virtualized!
  – operating system dependencies
  – issues with fixed-size address space, page-size
  – memory address formation
  – maintaining precise exceptions
  – instruction set features
  – instruction discovery during indirect jumps
  – self-modifying and self-referencing code
C-ISA Not for Being Virtualized

• Conventional ISA
  – after the fact solution for portability
  – no built-in ISA support for virtualization

• High-level language V-ISA
  – VM based portability is a primary design goal
  – generous use of *metadata*
  – metadata allows better type-safe code verification, interoperability, and performance
Operating System Dependencies

• Conventional ISA
  – most difficult to emulate
  – exact emulation may be impossible (different OS)

• High-level language V-ISA
  – find a least common denominator set of functions
  – programs interact with the library API
  – library interface is higher level than conventional OS interface
Memory Architecture

• Conventional ISA
  – fixed-size address spaces
  – specific addresses visible to user programs

• High-level language V-ISA
  – abstract memory model of indefinite size
  – memory *regions* allocated based on need
  – actual memory addresses are never visible
  – *out-of-memory* error reported if process requests more that is available of platform
Memory Address Formation

• Conventional ISA
  – unrestricted address computation
  – difficult to protect runtime from un-authorized guest program accesses

• High-level-language V-ISA
  – pointer arithmetic not permitted
  – memory access only through explicit memory pointers
  – static/dynamic type checking employed
Precise Exceptions

• Conventional ISA
  – many instructions trap, precise state needed
  – *global* flags enable/disable exceptions

• High-level language V-ISA
  – few instructions trap
  – test for exception encoded in the program
  – requirements for precise exceptions are relaxed
Instruction Set Features

• Conventional ISA
  – guest ISA registers > host registers is a problem
  – ISAs with condition codes are difficult to emulate

• High-level language V-ISA
  – stack-oriented
  – condition codes are avoided
Instruction Discovery

• Conventional ISA
  – indirect jumps to potentially arbitrary locations
  – variable-length instruction, embedded data, padding

• High-level-language V-ISA
  – restricted indirect jumps
  – no mixing of code and data
  – variable-length instructions permitted
Self-Modifying/Referencing Code

- Conventional ISA
  - pose problems for translated code
- High-level language V-ISA
  - self-modifying and self-referencing code not permitted
Pascal P-code

• Popularized the Pascal language
  – simplified porting of a Pascal compiler
• Introduced several concepts used in HLL VMs
  – stack-based instruction set
  – memory architecture is implementation independent
  – undefined stack and heap sizes
  – standard libraries used to interface with the OS
• Objective was compiler portability (not application portability)
Object Oriented HLL Virtual Machines

• Used in a networked computing environment

• Important features of HLL VMs
  – security and protection
    • protect remote resources, local files, VM runtime
  – robustness
    • OOP model provides component-based programming, strong type-checking, and garbage collection
  – networking
    • incremental loading, and small code-size
  – performance
    • easy code discovery allows entire method compilation
JVM Implementation

• A typical JVM implementation consists of
  – class loader subsystem
  – memory subsystem
  – emulation/execution engine

• see figure 6.1
Class Loader

• Functions
  – find the binary class
  – convert class data into implementation-dependent memory image
  – verify correctness and consistency of the loaded classes

• Security checks
  – checks class magic number
  – component sizes are as indicated in class file
  – checks number/types of arguments
  – verify integrity of the bytecode program
Java Native Interface (JNI)

• Allows java code and native code to interoperate
  – access legacy code, system calls from Java
  – access Java API from native functions

• see figure 5.13
  – each side compiles to its own binary format
  – different java and native stacks maintained
  – arguments can be passed; values/exceptions returned
Garbage Collector

• Provides implicit heap object space reclamation policy.
• Collects objects that have all their references removed or destroyed.
• Invoked at regular intervals, or when low on memory.
• see figure 6.5
  – root set point to objects in heap
  – objects not reachable from root set are garbage
Types of Collectors

• Reference count collectors
  – keep a count of the number of references to each object

• Tracing collectors
  – using the root set of references
Mark and Sweep Collector

• Basic tracing collector
• Garbage objects combined into a linked list
  – leads to fragmentation
  – segregated free-lists can be used
• Consolidation of free space can improve efficiency
  – compacting collectors (see figure 6.6)
  – copying collectors (see figure 6.8)
  – moving collectors need to update references (see figure 6.7)
Generational Collectors

• Reduce number of objects moved during each collection cycle.
• Exploit the bi-modal distribution of object lifetimes.
• Divide heap into two sub-heaps
  – *nursery*, for newly created objects
  – *tenured*, for older objects
• Collect a smaller portion of the heap each time.
Concurrent Collectors

• *Stop-the-world* collectors
  – time consuming, long pauses
  – unsuitable for real-time applications
• GC concurrently with application execution
  – partially collected heap may be unstable (see figure 6.9)
  – synchronization needed between the application (mutator) and the collector