Final Exam

- The take home final runs next week
  - Released early on Monday 15th
  - Due at midnight on Friday 19th
  - No extensions are available
- Clarification questions only during the week
  - I'll clear up ambiguity in the questions
  - I won't elaborate on or review course content
- Ask any questions on course content this week

METACIRCULARITY

Self-Hosting

- A compiler is just a regular computer program
  - Takes source files as inputs
  - Produces executable code as output
- A self-hosting compiler can compile itself
  - Compiler written in the target language
- Chicken-and-egg problem
  - Interrupt the recursion with a bootstrap compiler

Metacircularity

- Special case of a self-hosting runtime
  - The language's own features optimize the runtime
- Both VM and application are optimized together
  - VM data structures managed by the GC
  - VM code passes through the JIT
- VM can to be written in a higher-level language
  - Use Java synchronization rather than pthreads

AOT Compilation

- Metacircularity is simple with an AOT compiler
  - Write the VM in the target language
  - Compile the VM to machine code
- Great if an AOT compiler already exists
  - Not necessarily certain
  - Many established languages do have one
- May also transpile to a better supported language
  - Convert Java source into C

JIT Compilation

- Much of the VM implementation relies on the JIT
  - Compiles both the VM and the application
- The JIT is a metacircular JVM is written in Java
  - Which must be JIT compiled
- Same problem as self-hosted compilers
  - Solve in the same way
  - Create a small bootstrap VM
  - Bootstrap VM is used only to get the system running

Intrinsics

- Some required language features are unavailable
  - Low-level pointer access is the most obvious
- Remember JIT compiler intrinsics
  - JIT has a better implementation than the class file
  - Class file is ignored
- We can use intrinsics to implement new features
  - Intercept all calls to a given class or interface
  - JikesRVM uses the `org.vmmagic` package

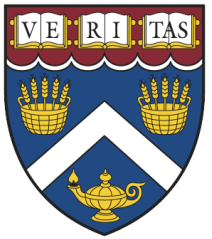
Magic

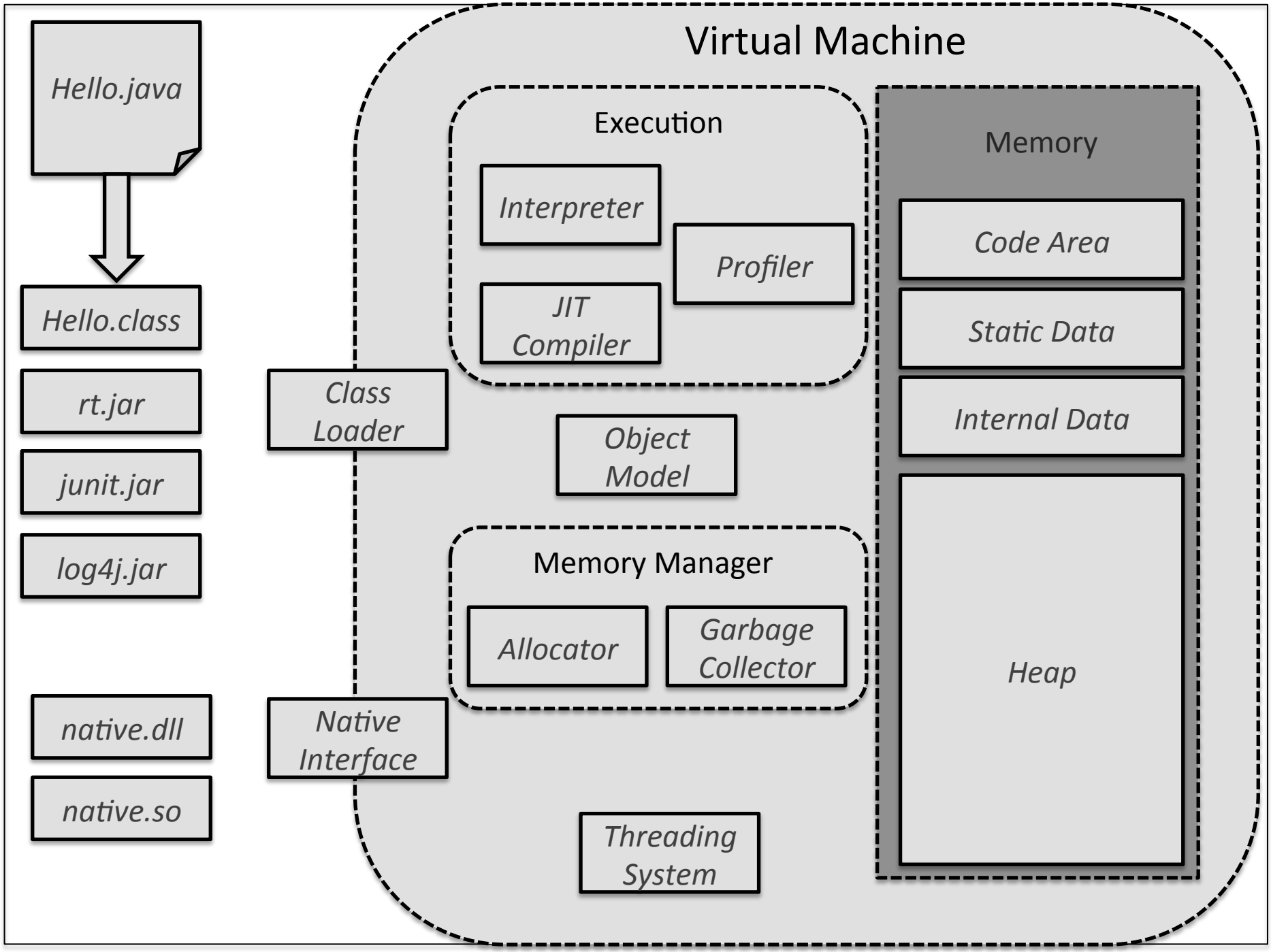
- Magic package split into two parts
- Pragma annotations
  - Tell the compiler to change the method
  - Remove safepoints, control compilation, etc
- Intrinsic annotation
  - Tell the compiler to replace the entire method

Magic Intrinsics

- `org.jikesrvm.runtime.Magic` has method stubs
  - Directly access registers
  - Manipulate stack frames
  - Cast arbitrary bits to Java types
  - Pointer arithmetic
  - Atomic operations such as CAS and memory fences
- Method stubs just throw runtime exceptions
  - Only there for the javac compiler
  - JIT compiler recognizes calls and rewrites

Review




Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Managed Languages

- Managed runtime environments are not new
  - Basic, Comal, Smalltalk ran using interpreters
  - Lisp, Self, used recognizable runtimes
- Popularity as a paradigm is now growing
  - Java, .NET in common server use
  - Ruby, Python initially scripting languages
  - JavaScript, ActionScript mainly in web browsers
  - Many embedded scripting languages

VM Implementations

- Java
  - Hotspot, JRocket, Maxine, Harmony, Jikes RVM
- .NET
  - CLR, Mono
- JavaScript
  - V8, WebKit, SpiderMonkey, Chakra
- Python
  - CPython, PyPy
- Ruby
  - MRI, YARV, Rubinus

Bytecode

- Simplified representation of the original program
- Most VMs use some form of bytecode
  - May be a custom internal representation
- Various cross-compilers exist
  - JRuby, XRuby, Jython target the JVM
  - IronRuby, IronPython target the CLR
- You read about JVM support for dynamic code

Bytecode Formats

- Targets an idealized system architecture
  - Doesn't correspond to an existing machine
  - Typically some form of stack-based computer
- Each Java method is represented by a stack frame
  - VM thread stack tracks method invocations
  - Each frame has an operand stack and local variables
- Values are either primitive or reference
  - Language defines various computational types

Method Dispatch

- Two ways in which methods are invoked
- Static dispatch finds methods by declared classes
  - `INVOKESTATIC`, `INVOKESPECIAL`
- Dynamic dispatch uses the runtime type
  - `INVOKEVIRTUAL`
- Dynamic dispatch allows for more flexibility
  - Slower to execute
  - Makes static analysis harder

Interpretation

- Interpretation is the simplest form of execution
  - Take each bytecode in turn
  - Simulate the state of the execution stack
- Switch interpreter is platform independent
  - Written once and compiled per platform
- Template interpreter is custom built at run time
  - Assembled from hand-optimized code
  - More complicated but higher performance

Class File Format

- Each class is defined by a `ClassFile` structure
  - Standard layout and required information
  - Class format versioned for backward compatibility
- Constant pool stores compile-time constants
  - Strings, class names, method references, etc
- Bytecode refers to constant pool entries
- Attributes supply optional or additional information
  - Method bytecode, debug information, etc

Class Loading

- Java allows new code to be added at runtime
  - Dynamic class loading
- Three types of class loader
  - Bootstrap class loader
  - System class loader
  - Context class loader
- Class loaders organized in a hierarchy
  - Delegate loading to their parent class loader

Class Loading Phases

- Loading
  - Get the bytes, check if class is already loaded
  - Parse the class file, check for compatibility
- Linking
  - Load required related classes
  - Verify the class file
  - Set up internal data structures
- Initialization
  - Assign default values to static fields
  - Execute any static initialization process

Verification

- Bytecode can come from untrusted sources
  - Need to ensure that the bytecode is valid
  - Can do it up-front or on demand
- Class level verification
  - `ClassFile` must have a valid structure
  - Class must fit into the established class hierarchy
- Method level verification
  - Simulated execution tracks all possible stack states
  - Verify bytecode types, jump targets, exception blocks, etc

Object Representations

- Data stored in the heap uses a standard layout
  - Data can be either an object or an array
  - Layout lets us easily distinguish between the two
- Some per-object metadata is required
  - Class pointer, GC data, hash code, lock information
- Metadata can be stored in a header or a handle
  - Headers require only one memory access per object
  - Handles make a number of algorithms simpler

Hotspot Object Header

- Two or three words
  - Class Pointer
  - Mark word
  - Optional array length

Hotspot Object Header

- Two or three words
  - Class Pointer
  - Mark word
  - Optional array length

State	Bitfields			Tag
Unlocked	Hash Code	Age	0	01
Thin Locked	Lock Information			00
Thick Locked	Monitor Address			10
GC Marked	GC Information			11
Biased Locked	Thread ID	Age	1	01

Dynamic Object Layout

- Not all languages are statically typed
  - Java tracks fields and methods at the class level
  - Ruby allows them to be added to a given object
- Class pointer is no longer sufficient
  - Each object may have a different layout
  - May need a per-object method or variable map
- Can default to class representation in most cases

Object Allocation

- Object layout is implementation dependent
  - Source of optimizations between VMs
- Allocation algorithm
  - Set aside heap space for the object
  - Populate the header or handle
  - Initialize all fields to their default values
  - Call the object's constructors
- Allocation may cause GC or throw OOME

Mixed Mode Execution

- Interpreter is convenient but slow
  - No startup compilation time
  - Simple to implement
- VM can mix interpreted and compiled code
  - May vary by method or code block
- Can defer compilation of certain methods
  - May also delegate compilation to background thread

Dynamic Profiling

- Compilation is expensive
  - We only want to compile important code
- Guided compilation based on profiling
  - Track hot methods by counting invocations
  - Track back branches to count loop iterations
  - Track branches on conditionals
- Monomorphic and polymorphic call sites
  - Determine set of possible methods called

Class Hierarchy Analysis

- Some optimizations use whole-program knowledge
  - Need to understand all classes in the system
  - Complicated by dynamic class loading
- Calculate class hierarchy at the current time
  - Inheritance relationships
  - Method overriding
  - Interface implementation
- May have to update when a new class is loaded

JIT Compilation

- Highly-optimizing compiler
  - Takes advantage of profiling and CHA information
- Optimizations transform between correct programs
  - Operate in multiple phases
- Compilation may be optimistic
  - Assume that profiling information is correct
  - Revert if it turns out to be wrong

JIT Optimizations

- Many possible optimizations
  - Constant folding
  - Escape analysis
  - Redundant load and store elimination
  - Loop unrolling
  - Register allocation
  - Type inference
  - Devirtualization
  - Inlining
  - Intrinsics

Compilation in Other Systems

- .NET JIT compiles all methods on first call
  - Optimizes only methods that are reachable
  - No dynamic profiling
- Standard Ruby VM compiles to bytecode only
  - Rubinius uses some JIT compilation
- Default CPython implementation has no JIT
  - PyPy project implements a JIT compiler

JIT in JavaScript

- Major competitive advantages between browsers
  - Good performance comparisons are hard to come by
- V8 doesn't bother with bytecode
  - Uses a fast (Full) compiler initially
  - Optimizes hot code with a optimizing compiler
- Webkit uses modified version of LLVM
  - Three-tier compilation strategy
- Chakra and SpiderMonkey use same approach as Java
  - Initial interpreter phase
  - Compile code after a certain threshold

Deoptimization

- Optimistic optimizations can be wrong
  - Wrong assumptions
  - Changes to the CHA graph
- Cooperative deoptimization
  - Code was designed to allow deoptimization
  - Extra checks for faulty assumptions
- Uncooperative deoptimization
  - Running code is paused by another thread
  - Deoptimization happens while the thread is suspended

Safepointing

- Sometimes we need the VM to be consistent
  - Deoptimization or OSR
  - Garbage collection
- Bring all threads to a safepoint
  - At bytecode boundaries
  - At a method exit or back branch
- Implement using a poison page algorithm

Memory Management

- Process to implicitly return memory to the heap
  - Liveness is approximated by reachability from roots
- Three major approaches to collecting
  - Reference Counting
  - Mark Sweep
  - Copying or Compacting
- Variations and hybrids also exist
  - Conservative collection
  - Generational collection

Reference Counting

- Maintain a count of pointers to a given object
  - Increment and decrement on pointer writes
  - Free the object when the count hits zero
- Easy to understand and implement
  - No additional runtime support needed
  - Extends regular malloc/free semantics
- Need to handle reference overflow
- Need more elaborate mechanism for cycles

Mark Sweep Collectors

- Mark phase identifies all live objects
  - Tricolor abstraction tracks object state
  - Abstraction implemented in various ways
- Sweep phase frees all objects that are not marked
  - Linear pass over the heap
- Holes in the memory space cause fragmentation
  - Allocator must be able to fill in the gaps

Copying Collectors

- Copying collectors move live objects
  - Reclaim all garbage objects en-masse
- Need to set aside a destination for the objects
  - Copy reserve must account for worst-case behavior
- Allocation can be very efficient
  - Fast path for a bump pointer is a few instructions
- Track copied objects using forwarding pointers

Compacting Collectors

- Copying collection without the copy reserve
  - Move objects within the same heap region
  - More space efficient, but more complicated
- Compaction uses reference chaining
  - Builds a linked list of reference locations per object
  - Linear scan determines new object locations
  - Follow linked list to write new object location

Generational Collectors

- Weak generational hypothesis
  - Roughly 90% of objects don't survive one GC cycle
- Strong generational hypothesis
  - Older objects likely to survive for a long time

- Generational collectors target newer objects
  - Partition the heap into age-based regions
  - Target most collections to the youngest region
  - Collect older regions only when necessary

Multi-region Collectors

- Don't need to limit ourselves to two regions
- We can partition the heap into many regions
  - Train algorithm
  - Garbage first algorithm
  - Scoped memory management
- Collecting smaller regions gives better pause times
  - Lower throughput
  - More bookkeeping overhead

Concurrency and Incrementality

- Stop the world collections pause all threads
  - Simple to reason about
  - Lead to long pauses, particularly on large heaps
- Concurrent collection uses a background thread
  - Need to account for mutator behavior
- Incremental collection happens a bit at a time
  - May also interleave with the mutator

Concurrent Marking

- Major performance gain from concurrency
  - Mark live objects while the mutator runs
  - Mutator can change the object graph
- Snapshot at the beginning algorithm
  - Marks all objects that were reachable at a given point
  - Also marks new objects created since then
- Incremental update algorithm
  - Marks objects alive at the end of the phase
  - Requires a remark pass

GC in Other VMs

- Python uses reference counting primarily
  - Fallback generational collector for cycles
- Ruby and Chakra use non-copying generational GC
  - Generations are tracked logically by object
- Rubinius has a three-generation copying collector
- .NET uses a three-generational compacting GC
  - Promote objects to older generations
  - Compact the oldest generation
  - Separate workstation and server versions

GC in JavaScript

- V8 and Chakra have separate scalar regions
  - Don't need to scan for references
- V8 uses generational GC
  - Occasional compaction in the mature region
  - Incremental marking, parallel lazy sweeping
- SpiderMonkey recently added precise GC
  - Now uses a generational collector

Exception Handling

- Checked and Unchecked handled similarly
  - Only difference is at compile and verify time
- Exceptions are thrown explicitly or implicitly
- Methods declare handler block
  - Include PC range, exception type and jump target
  - Uncaught exceptions terminate the thread

Native Code

- Most managed runtimes have a native interface
  - Interface is bi-directional
  - Defines how native calls Java and vice versa
- Java code declares methods as native
  - Linker finds the appropriate method body
  - Makes the method call through JNI
- Native code can embed a full Java VM

JNIEnv

- Native code accesses VM through the JNIEnv
- Single interface that gives access to the VM
  - Defines pointers to interface functions
  - Allows implementation to be changed
- The VM can track all calls from native to managed
  - Add tracking or bookkeeping
  - Manage safepoints

Object References

- Native code needs to access data on the Java heap
  - Native not aware of moving GC
  - Can't move objects with direct native references
- Indirect access through handles
  - Handles deal with moving objects
  - Introduces overhead when accessing managed data
- Can pin or copy arrays
  - Pinning has a negative effect on GC
  - Copying introduces overhead

Debugging and Profiling

- Debugging and profiling tools split into two parts
  - External heavyweight tool
  - Small JVMTI agent
- Agent communicates with internal VM systems
  - Sends relevant data to external tool
- Agent interface allows widespread VM modification
  - Trigger events
  - Bytecode instrumentation
  - Modify internal systems

Threading

- Threads used by both application and VM
  - Application threads use the Java Memory Model
- VM can manage threads in two ways
  - Green threads are scheduled by the VM
  - Native threads scheduled by the operating system
- Two types of threads in the system
  - Daemon and non-daemon

Mutual Exclusion

- Java uses monitors to guard critical code
  - Only one thread can own a monitor at a given time
- Performance optimized for no-contention case
  - Contention always hurts performance
- Various mechanisms to optimize locks
  - Thin/thick locking
  - Biased locking

Cooperation

- Java also uses monitors for thread communication
  - Wait/Notify semantics
- Monitor maintains two sets
  - Entry set for threads that want the monitor
  - Wait set for threads waiting to be notified
- Various VM and OS primitives help cooperation
  - Atomic instructions
  - Volatile variables

VM Threads

- Several internal VM threads
  - Scheduler, Signal dispatcher, VMThread
- Background JIT compilation thread
  - Can largely run in parallel
- Parallel and concurrent GC threads
  - Concurrent marking
  - Parallel copying

Other Topics

- Weak, soft and phantom references
- Finalization
- Ahead of time compilation
- Benchmarking
- Metacircularity

- Reading topics
  - `INVOKEDYNAMIC`
  - Conservative GC
  - Facebook on Android