In the Last Week

- Some good questions around Assignment 4
  - Structure of the line number table
  - Winding the PC back to the invoke instruction
  - How to clear the stack
  - How to rethrow exceptions
  - Using Exception in your code

Facebook Blog Post

- Engineers at Facebook built a new Android app
  - Re-implemented much of their application
- Used a new implementation pattern
  - Reduced method size
  - Greatly increased the number of methods
- Successive problems on old Android versions
  - Overflowing a buffer that tracks methods

Marching Ever Onwards

- Implemented a series of fixes
  - Couldn't break the application into smaller modules
  - Added code directly to the system class loader
  - Realized that the same problem exists at runtime
  - Tried and failed to bring down method count
  - Used JNI to increase the buffer size
  - Realized that vendors had changed the platform
  - Added heuristics to guess where the buffer was
  - Tested across 70 different models of phone

Discussion Boards

- Range of opinions on the problem and solution
  - Should they have hit the limit in the first place?
  - Was it worth targeting the new app to old phones?
  - Could the app be split into services?
  - It would have helped to know what else they tried
- General sympathy for the position they were in
  - Tight release deadline
  - Having to support old versions of the platform
  - Pragmatism when trading off business requirements

Issues With the Solution

- Was there a reason for the smaller buffer size?
  - Was it determined based on the hardware capacity?
  - Could scanning the larger buffer tax the GC?
- Effects on Google
  - Violating the spirit of the interface
  - Later Dalvik patch had to support the hack
- Why was the buffer fixed size to begin with?
  - The dangers of leaky abstractions
- Inherent fragility of the approach

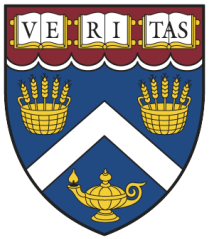
What They Did Right

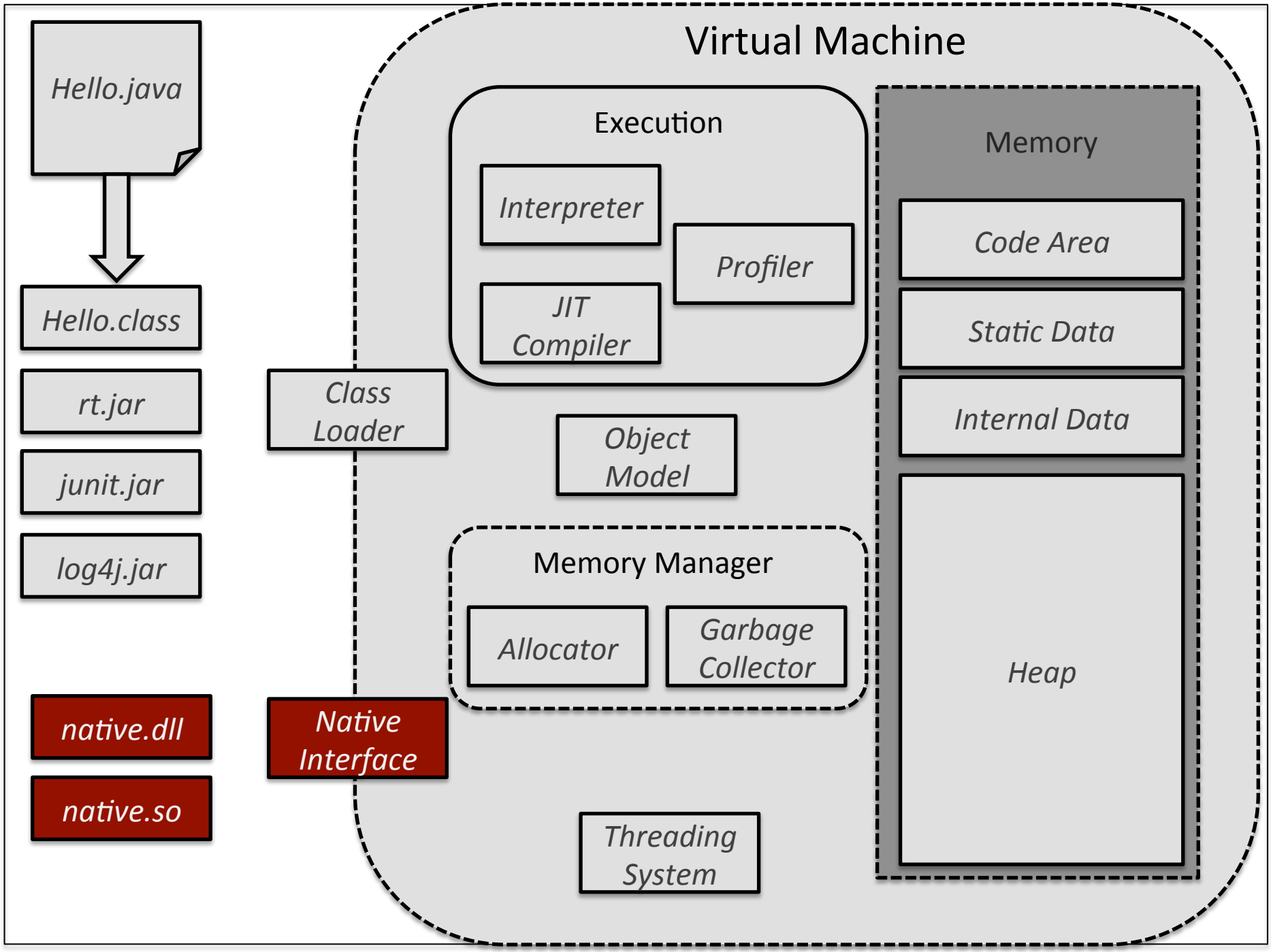
- Lots of testing
  - Gave them some confidence in their approach
- Knew the environment enough to make it work
  - Understood the problem and potential solutions
  - Dived into the source to convince themselves
- Shipped the app

Take-Away Lessons

- Know your platform ahead of time
  - Earlier knowledge of Dalvik internals could help
- Understand the business requirements
  - A temporary fix can be useful at times
  - Getting to market is important
- Manual testing has its place
- If you hack, don't blog about it…

Native Code II



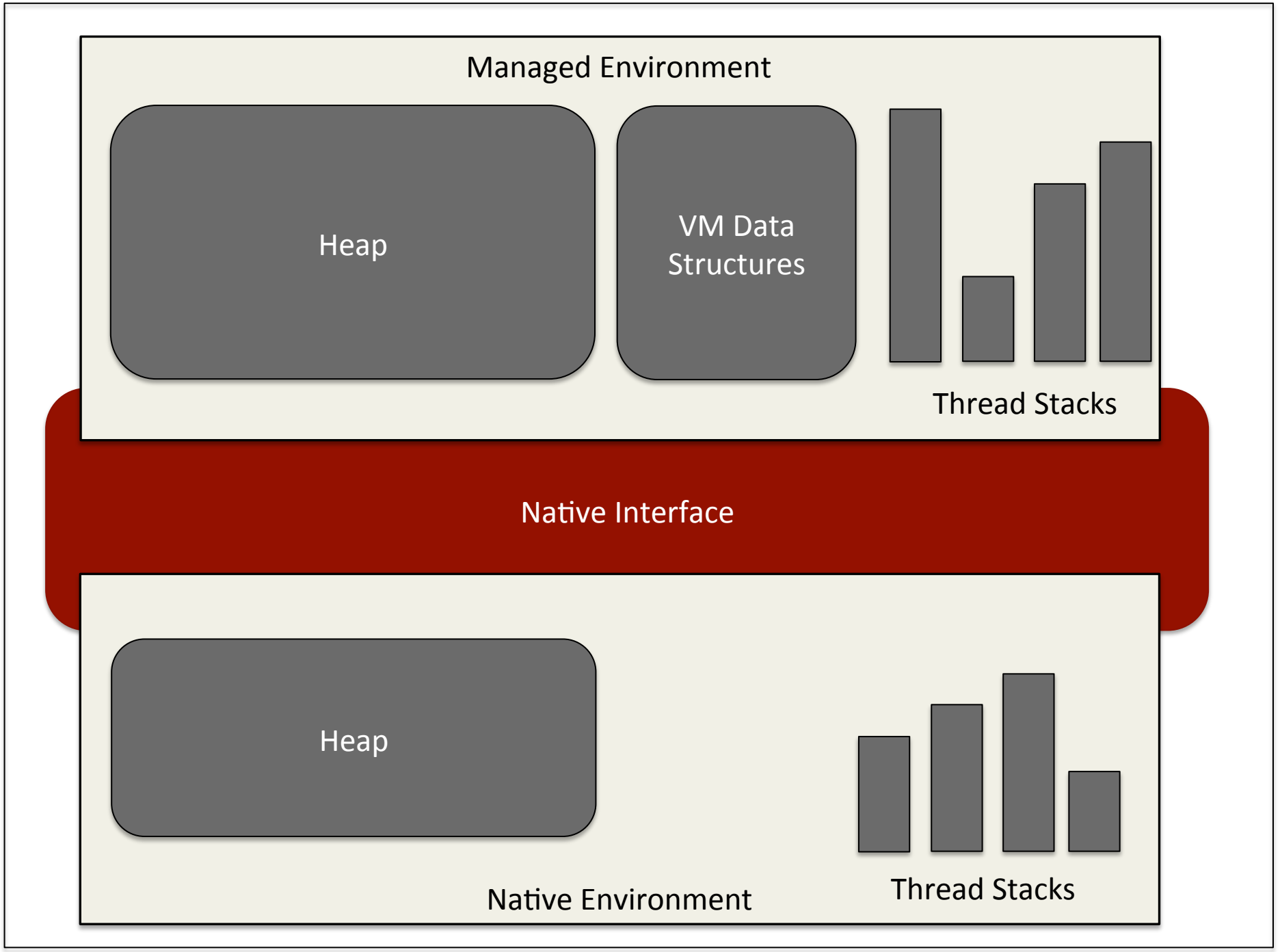

Hello.java
Hello.class
rt.jar
junit.jar
log4j.jar
native.dll
native.so
Virtual Machine
Execution
Interpreter
Profiler
JIT Compiler
Class Loader
Object Model
Memory Manager
Allocator
Garbage Collector
Native Interface
Threading System
Memory
Code Area
Static Data
Internal Data
Heap

Why Do We Need Native Code

- Need to integrate with existing code
  - Source and build tools may not be available
  - Modifying existing code may be impractical
- Need performance for critical section
  - May want to hand-optimize in assembly code
- Want to do things not allowed by the VM
  - Take advantage of custom platform features

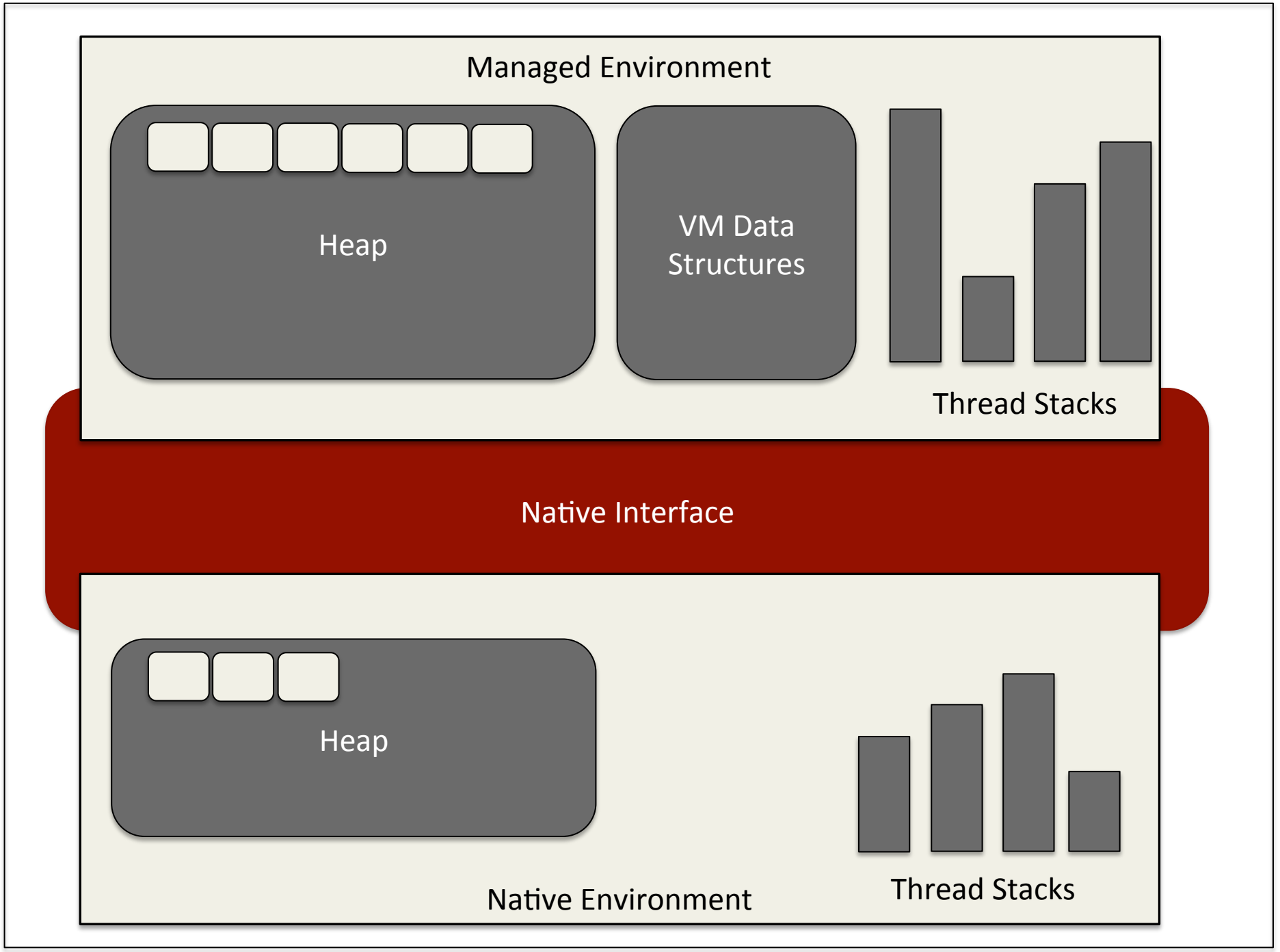
Foreign Function Interface

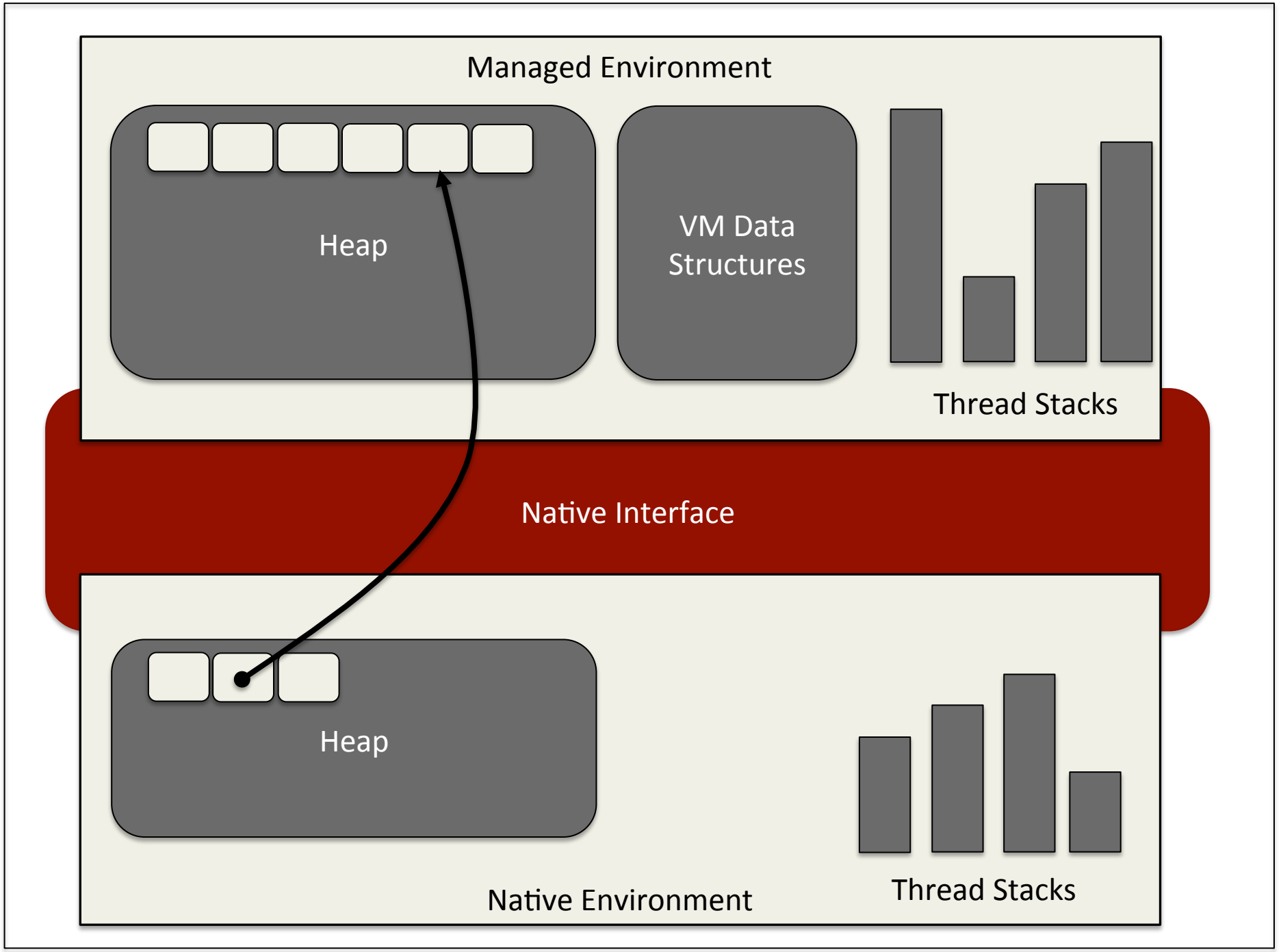
- Integrates managed code with native
  - Imposes a structure on native code
  - Ensures that native code respects VM conventions
- Java Native Interface defines boundary
  - Strict control of interactions between systems
- JNIEnv object provides access to VM
  - Double indirection lets us change implementation

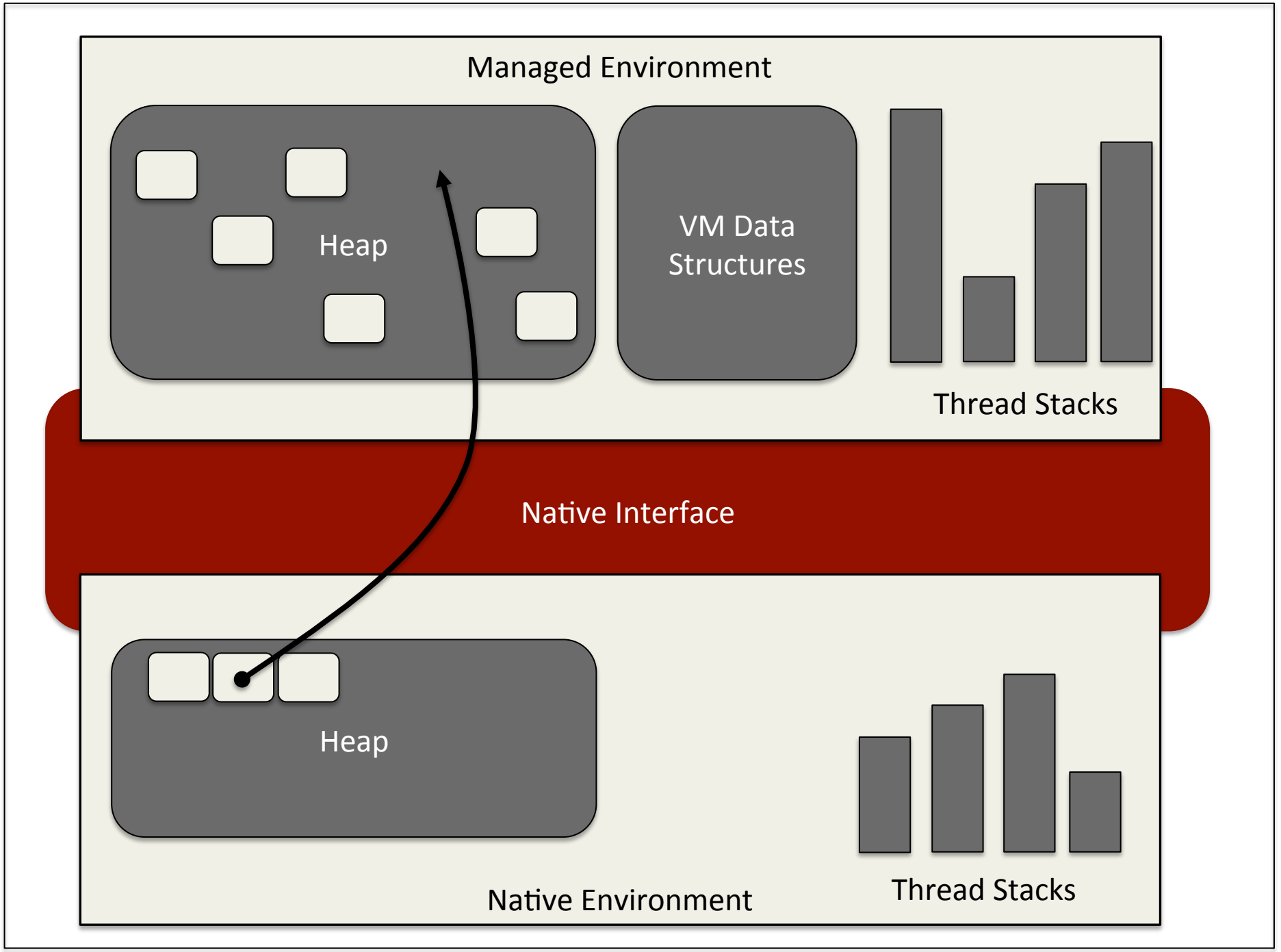

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

Memory Access

- Native code uses raw pointers
  - Prevents moving garbage collection
  - Assumes object layout

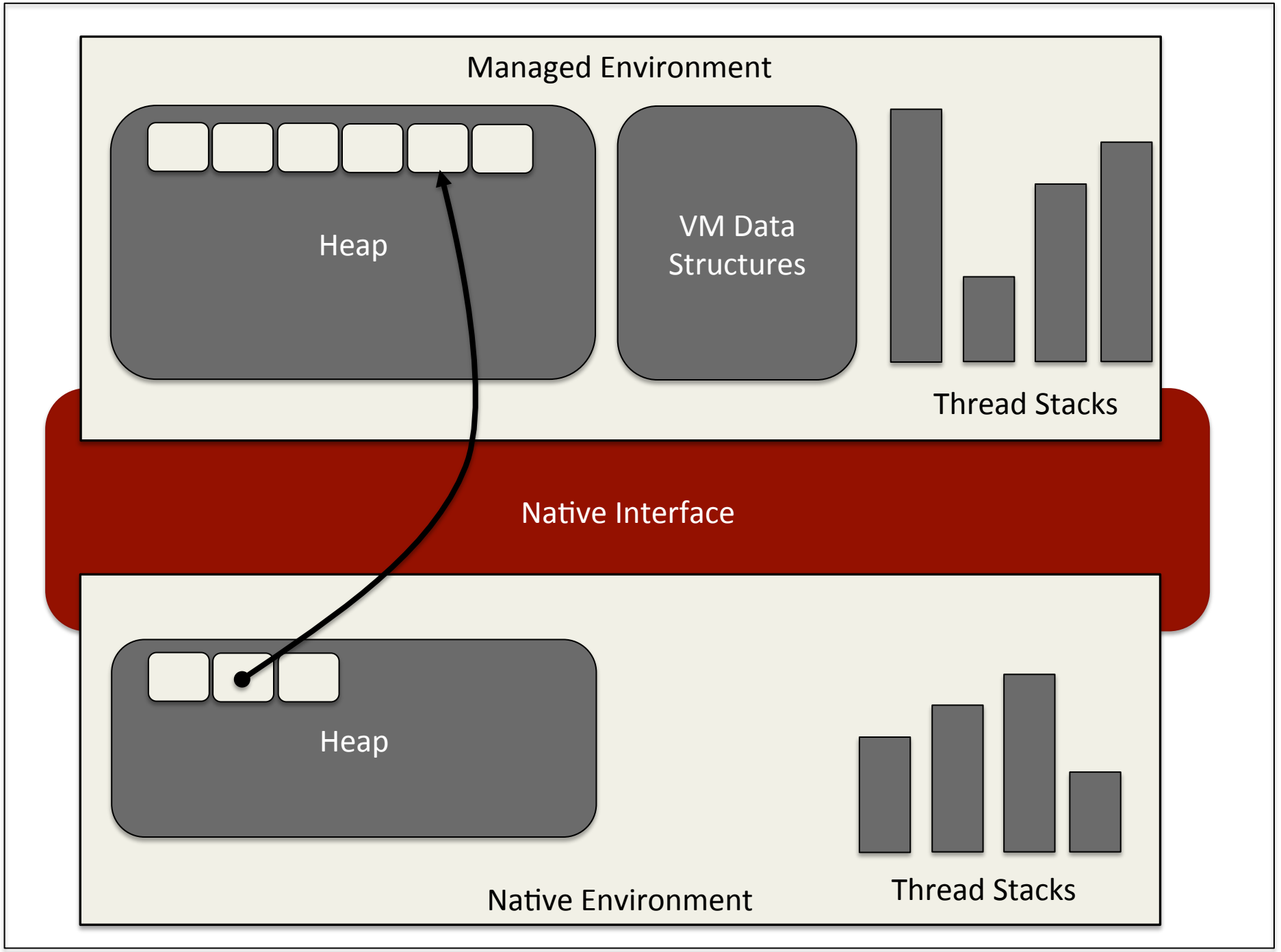

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

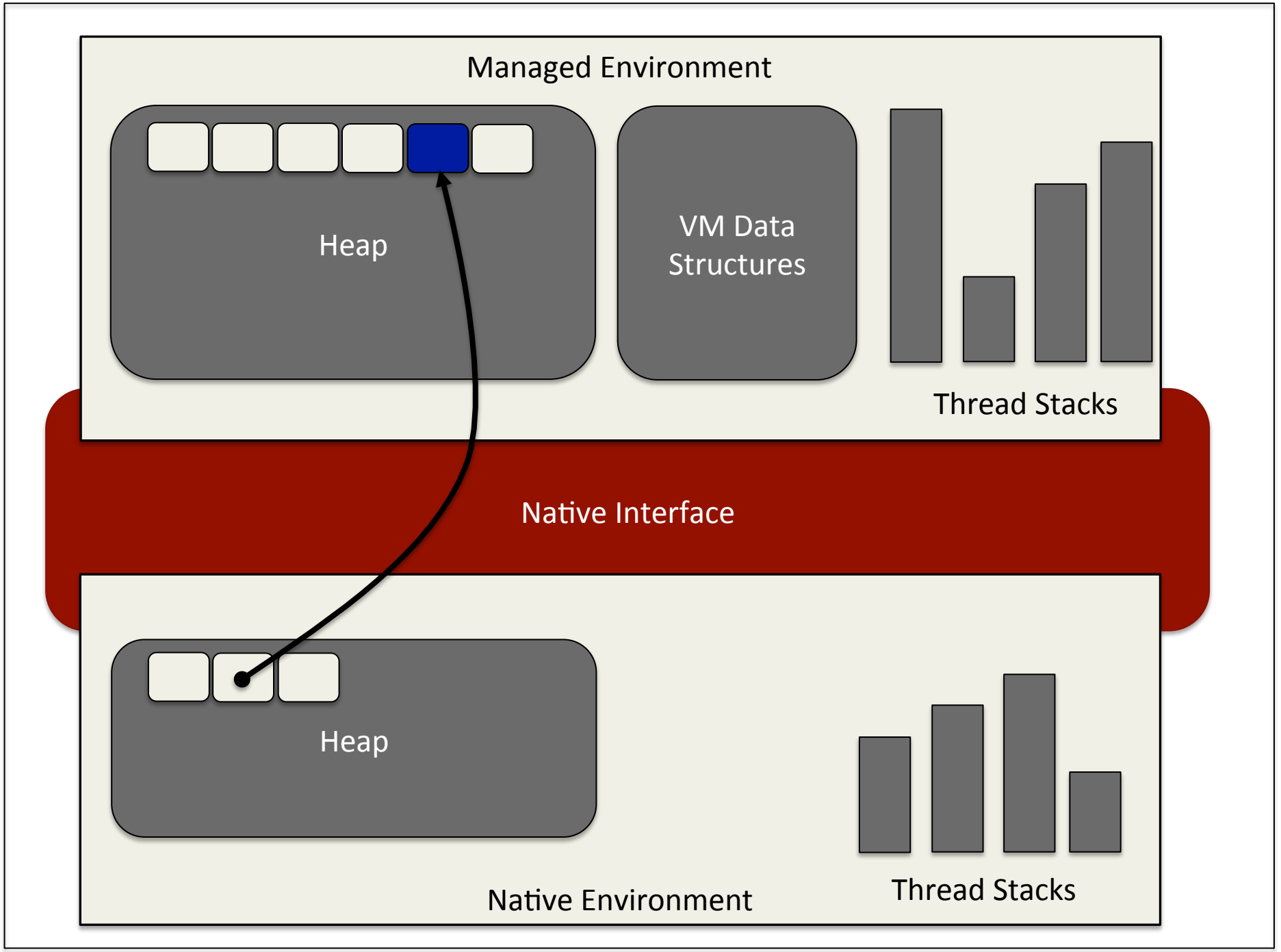

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Thread Stacks
Native Environment

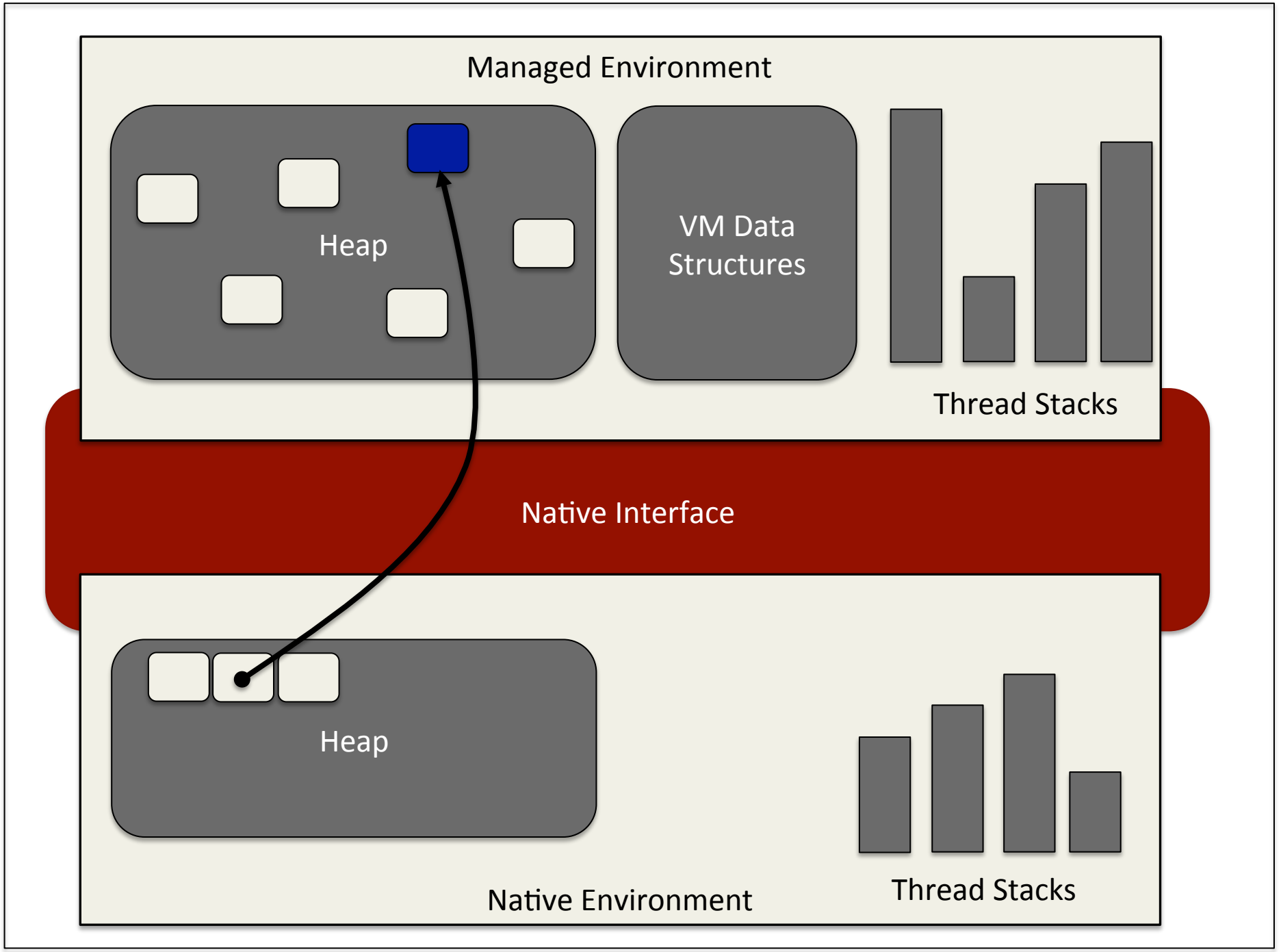

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

Memory Access

- Native code uses raw pointers
  - Prevents moving garbage collection
  - Assumes object layout
- One possible solution is pinning
  - Designate an object as unmovable
  - Restricts implementation of the VM

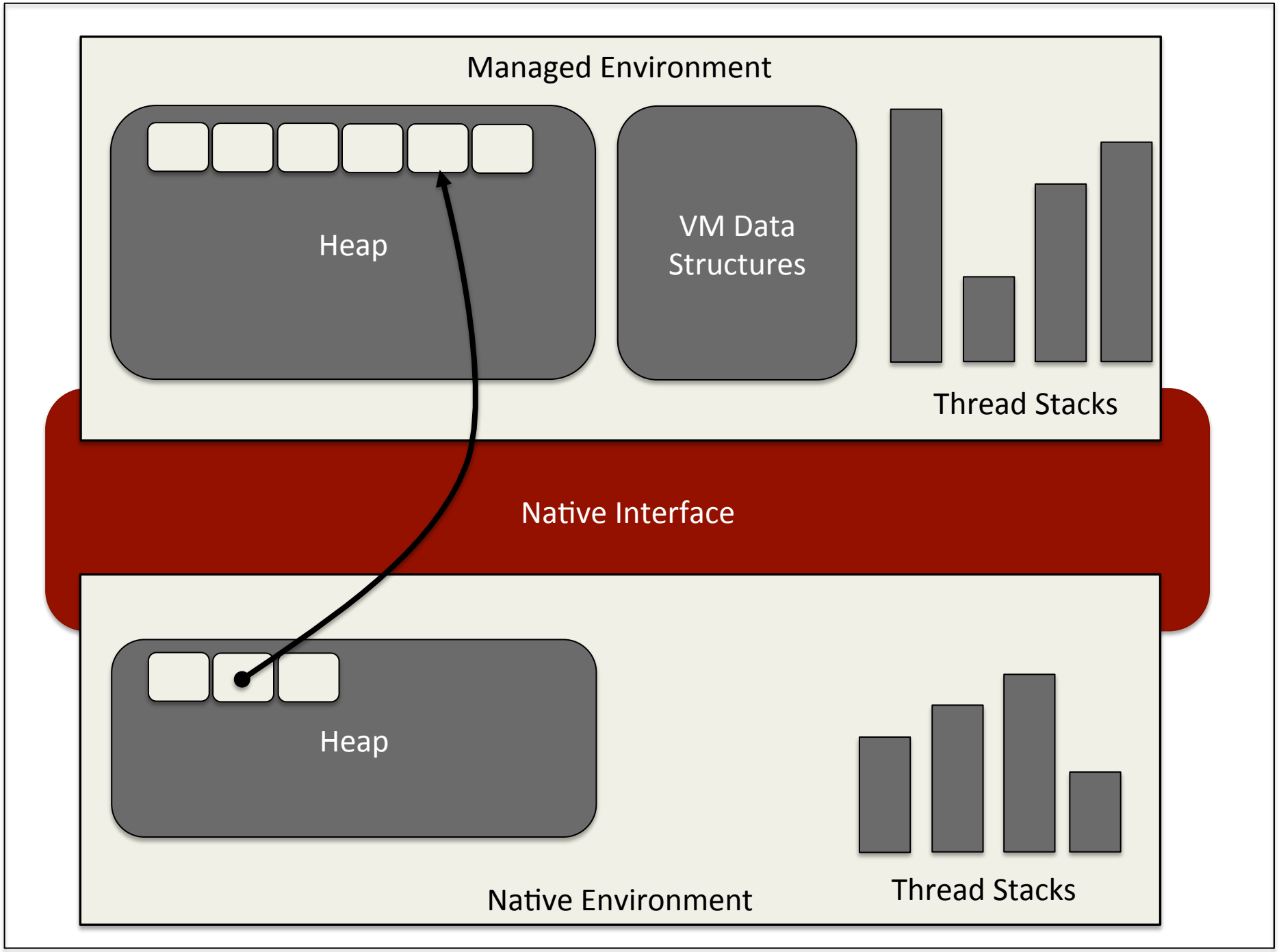

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Thread Stacks
Native Environment

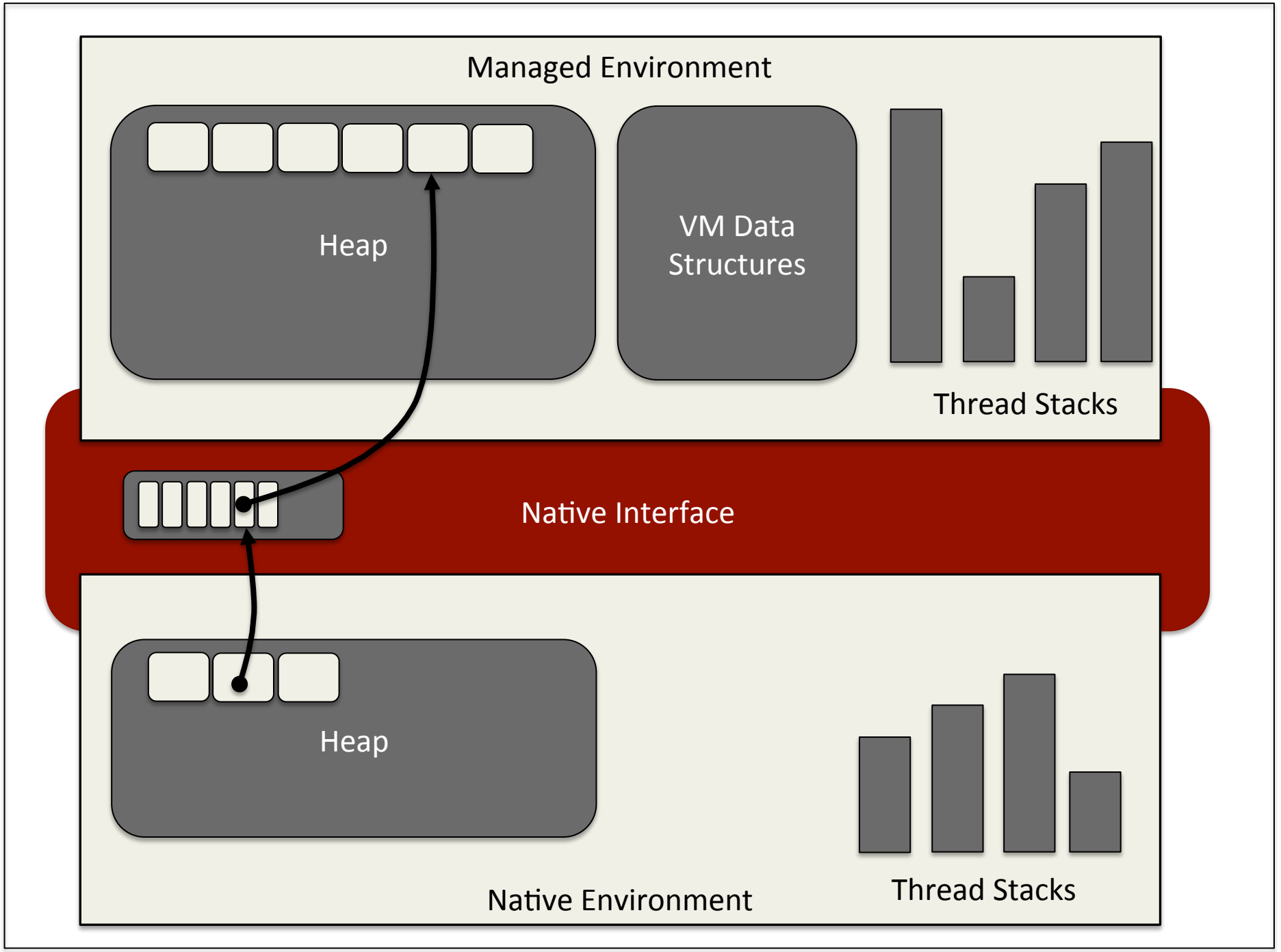

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

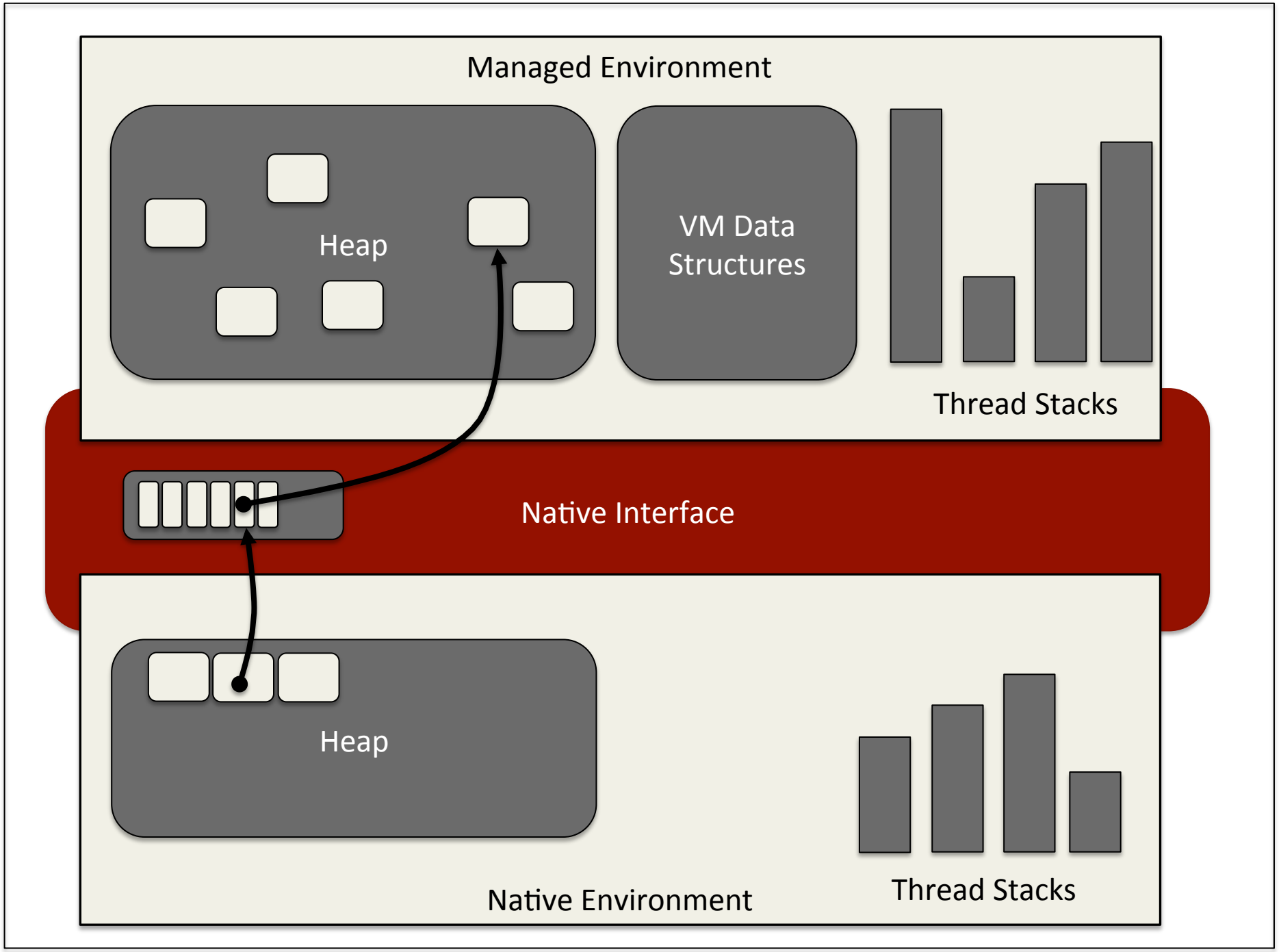

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

Memory Access

- Native code uses raw pointers
  - Prevents moving garbage collection
  - Assumes object layout
- One possible solution is pinning
  - Designate an object as unmovable
  - Restricts implementation of the VM
- Another approach is to use handles


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Thread Stacks
Native Environment


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

Handles

- We've talked about handles before
  - Objects can be moved without updating pointers
  - All accesses involve an extra indirection
- In this case the trade-off is better
  - Expect overhead when going from native to VM
  - Keeps object model out of native code
  - Lets us move objects without modifying native

Reserving Objects

- Native needs to tell the VMs what object it uses
  - Don't want a handle for every object
  - Part of the native interface definition
- Objects can be reserved in two ways
  - Local references scoped to the native method
  - Global references need to be explicitly freed
- Global created by promoting local reference

Local References

- Reserved for the execution of a single method
  - Automatically released on method exit
- Local references are thread specific
  - Native code can't share between threads
  - Could lead to dangling reference
- Local references can be released manually
  - Prevent locking of large objects
  - Avoid locking large numbers of objects

Global Reference

- Created by upgrading a local reference
- Native code takes responsibility for the reference
  - VM will never assume the reference is released
  - Native must explicitly release
- Can be shared between threads
- Potential source of memory leaks

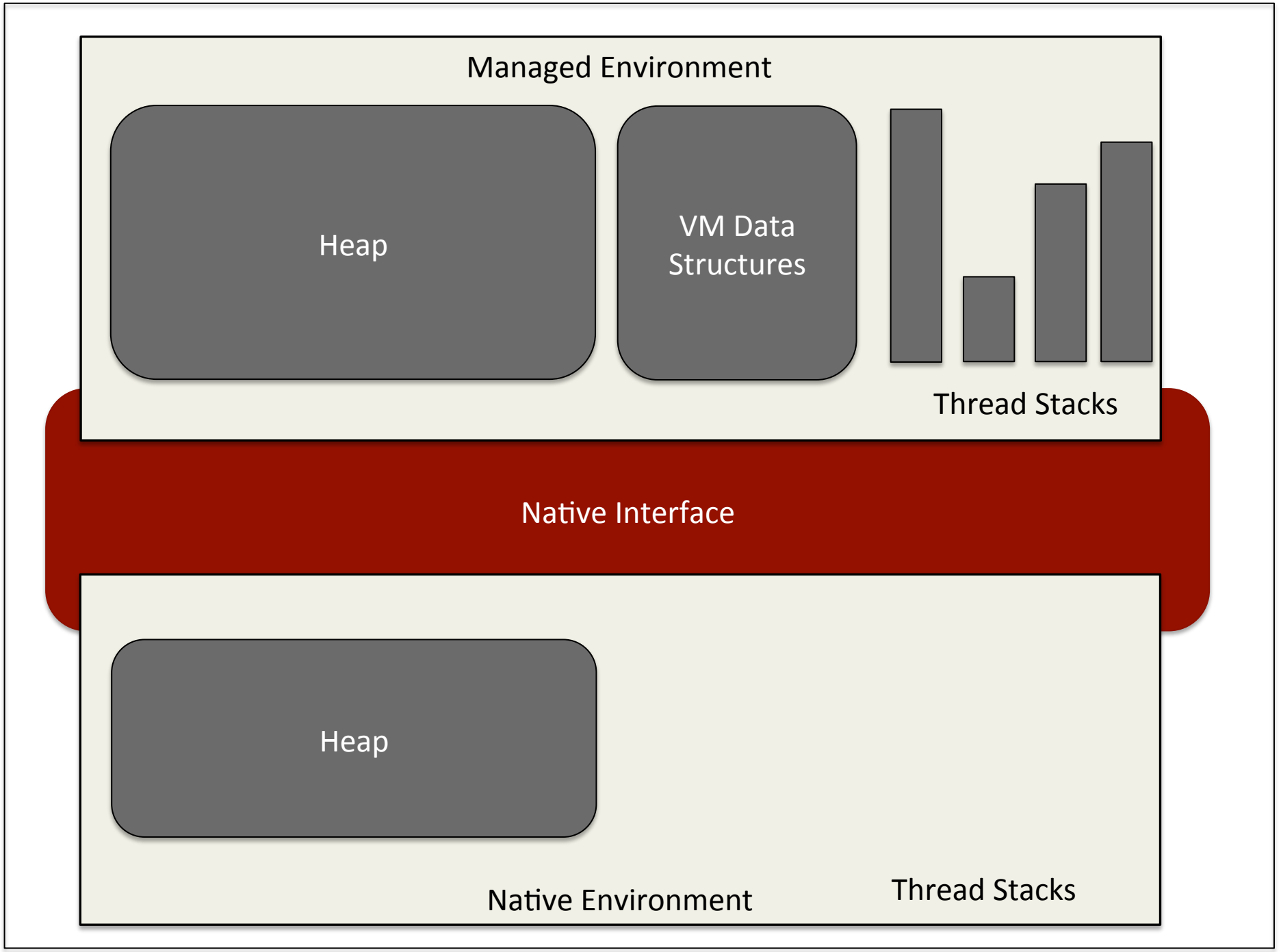
Handle Registry

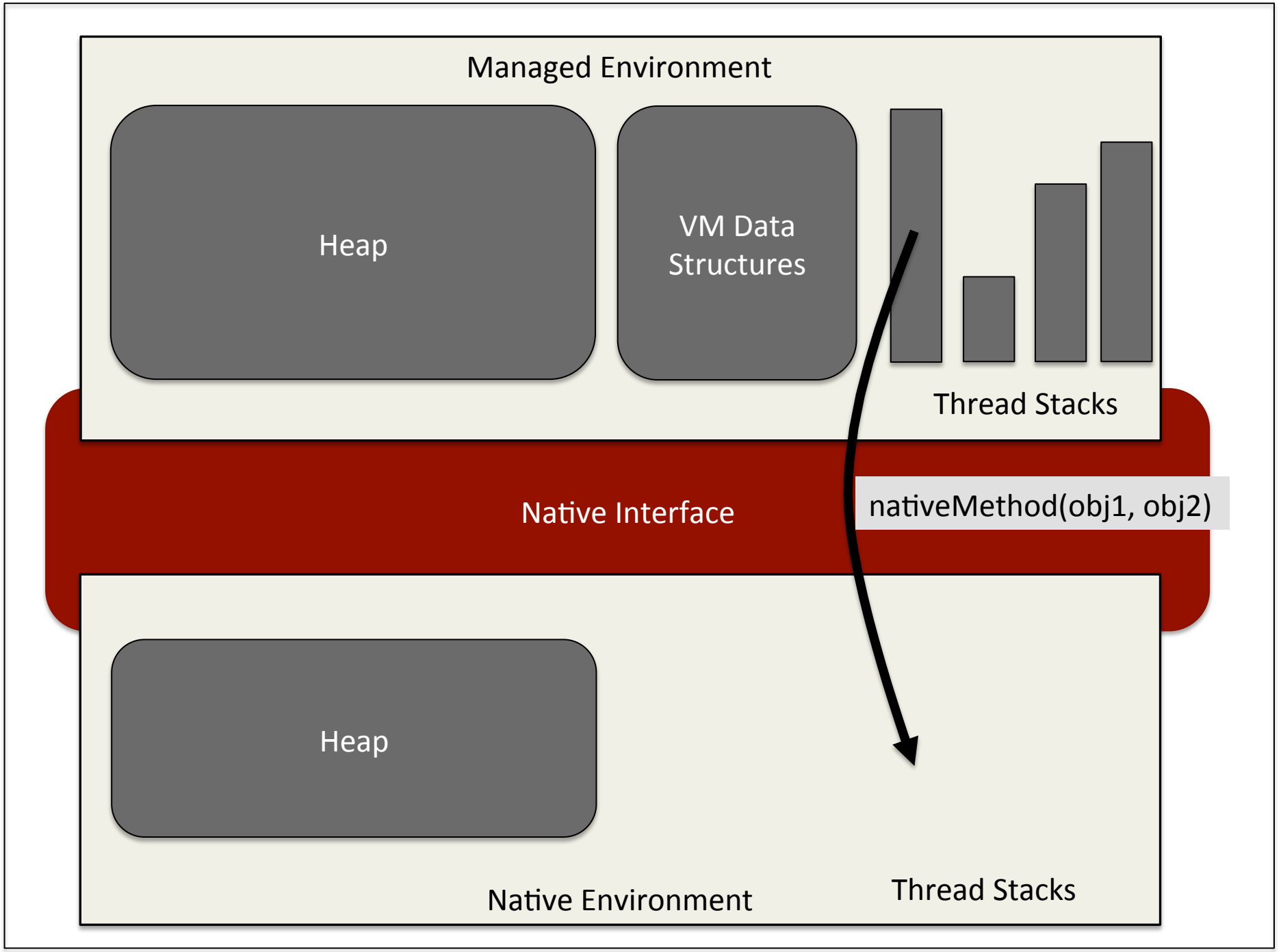
- Mechanism to keep track of references
  - Both local and global
  - Native code holds key index into the registry
  - Registry maps to Java object
- Separate mechanisms for local and global entries
  - Global uses explicit add and remove semantics
  - Local entries are added and removed automatically

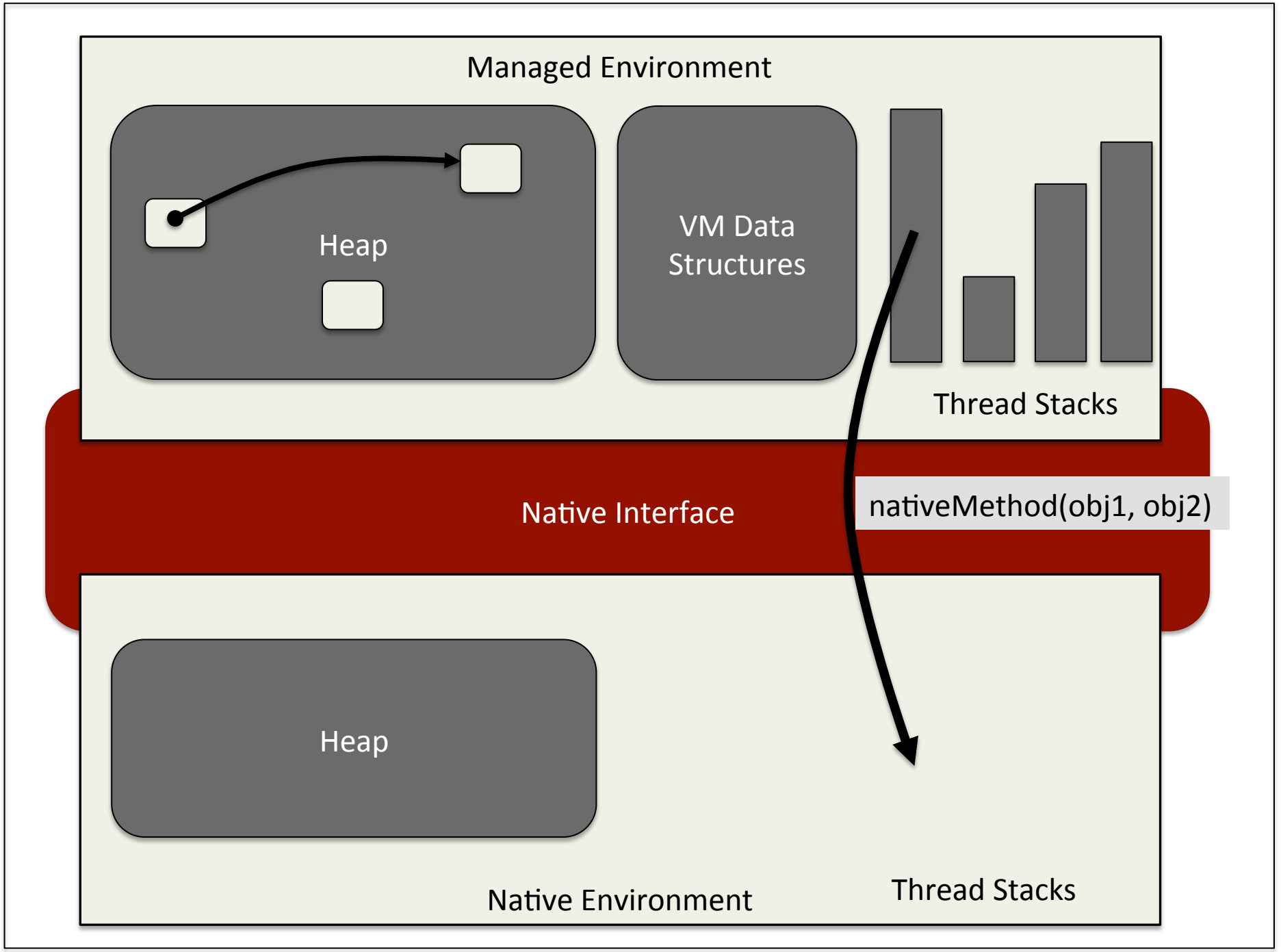
Local Registry

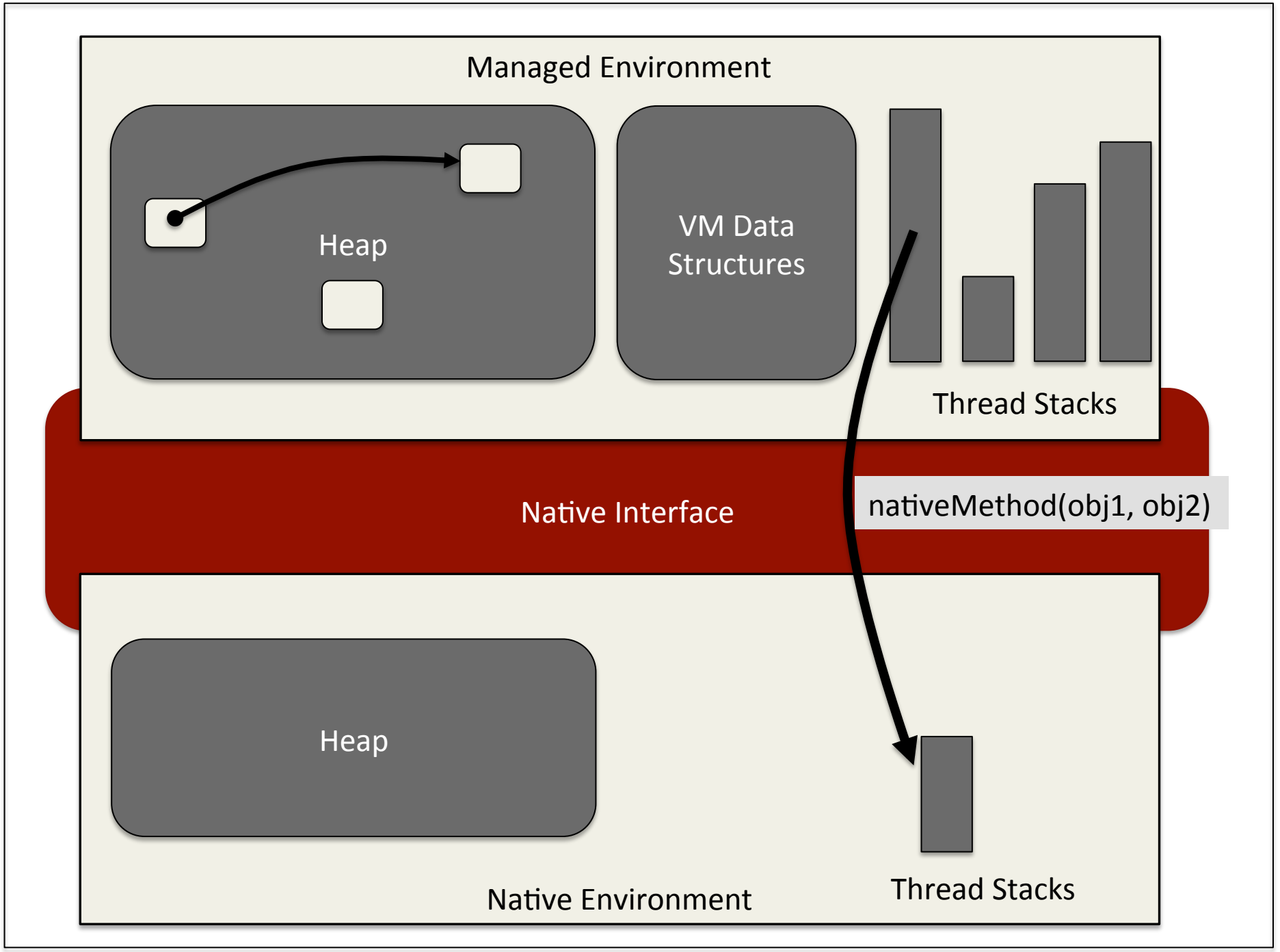
- Local references created in two ways
  - Values passed as parameters to native code
  - Return values from VM-level methods
- Explicitly retrieving a reference uses the second

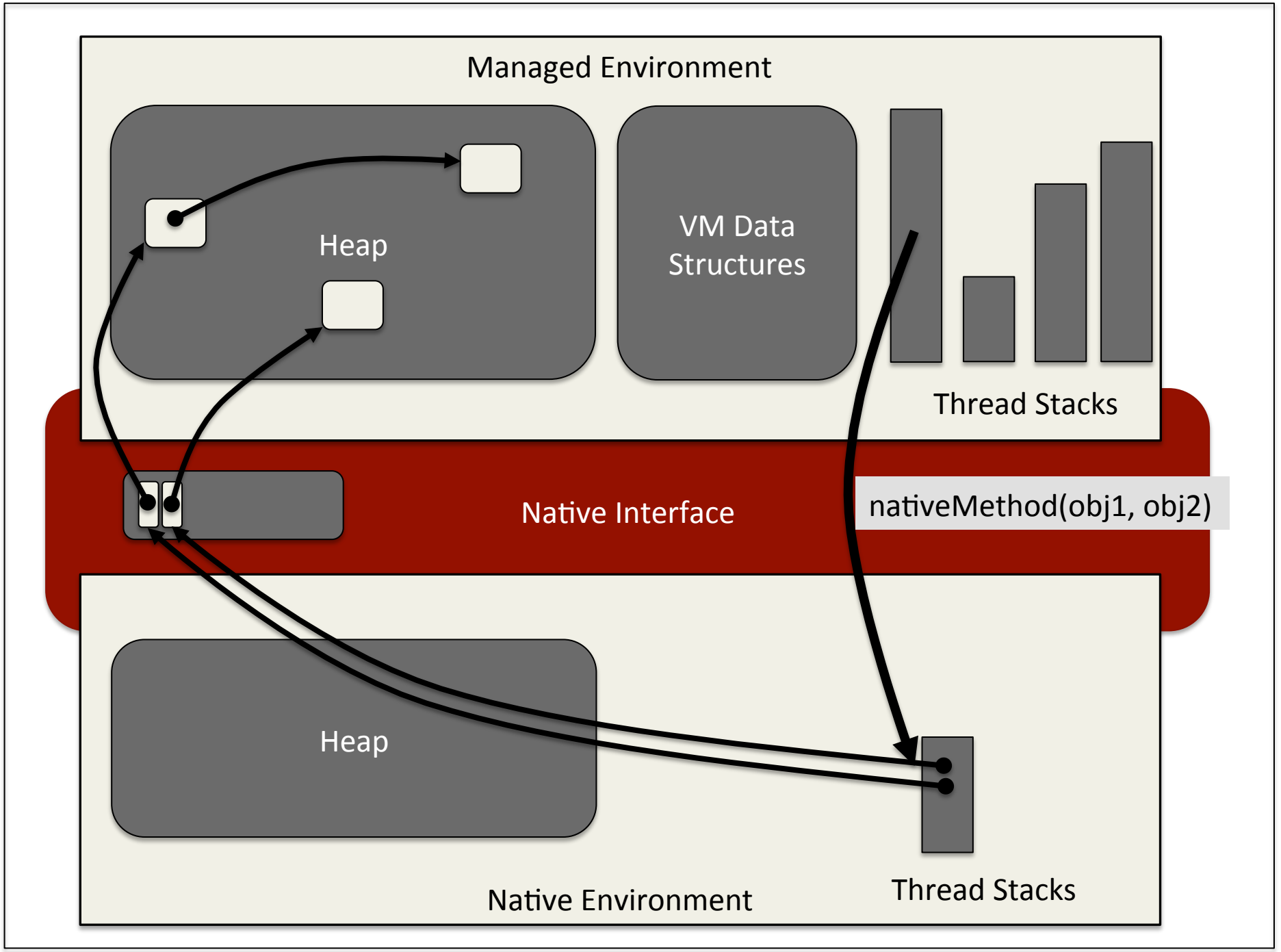
- References added to registry automatically
  - JNI tracks every transition from VM to native
- References deleted when native method returns

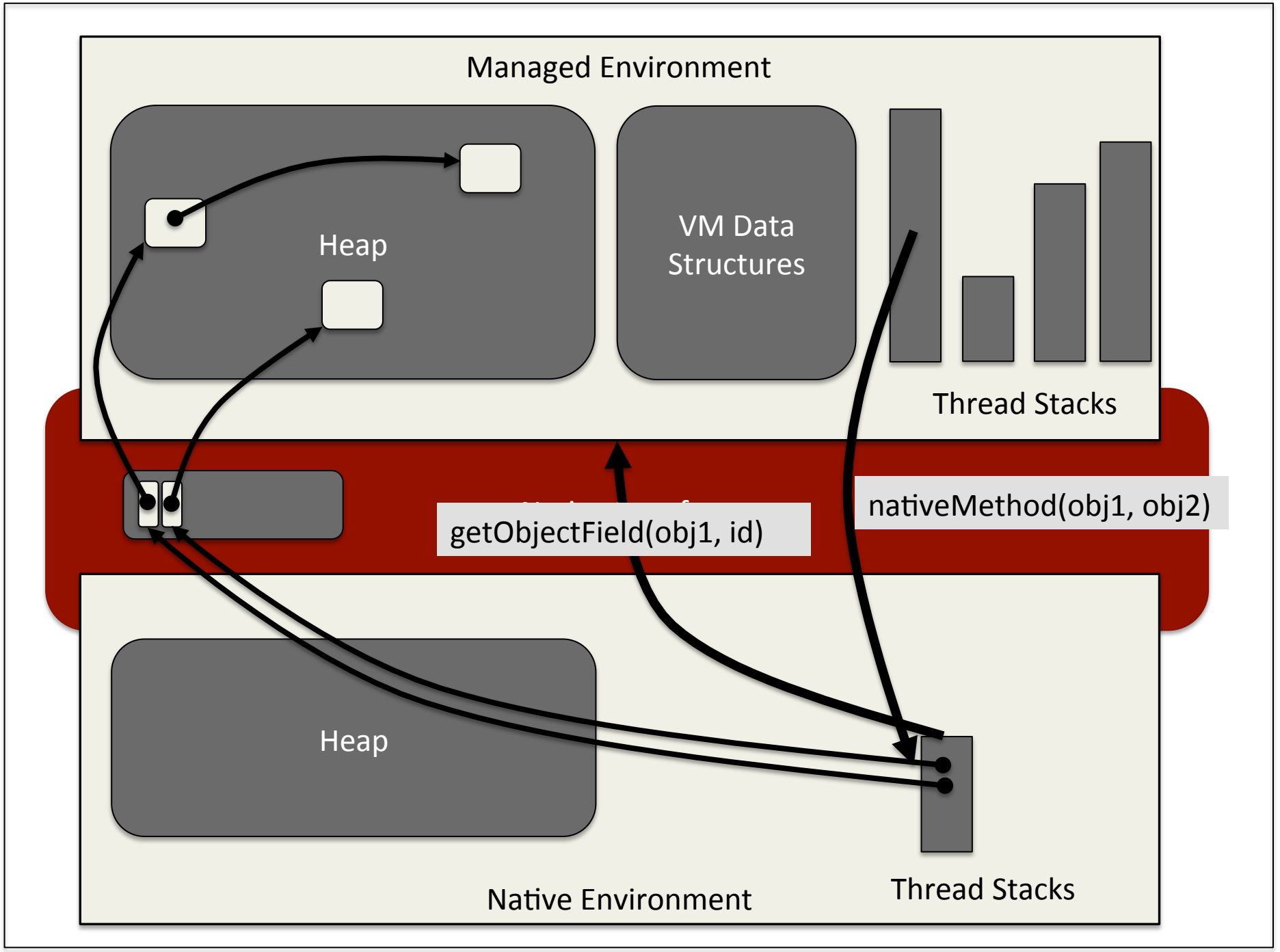

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

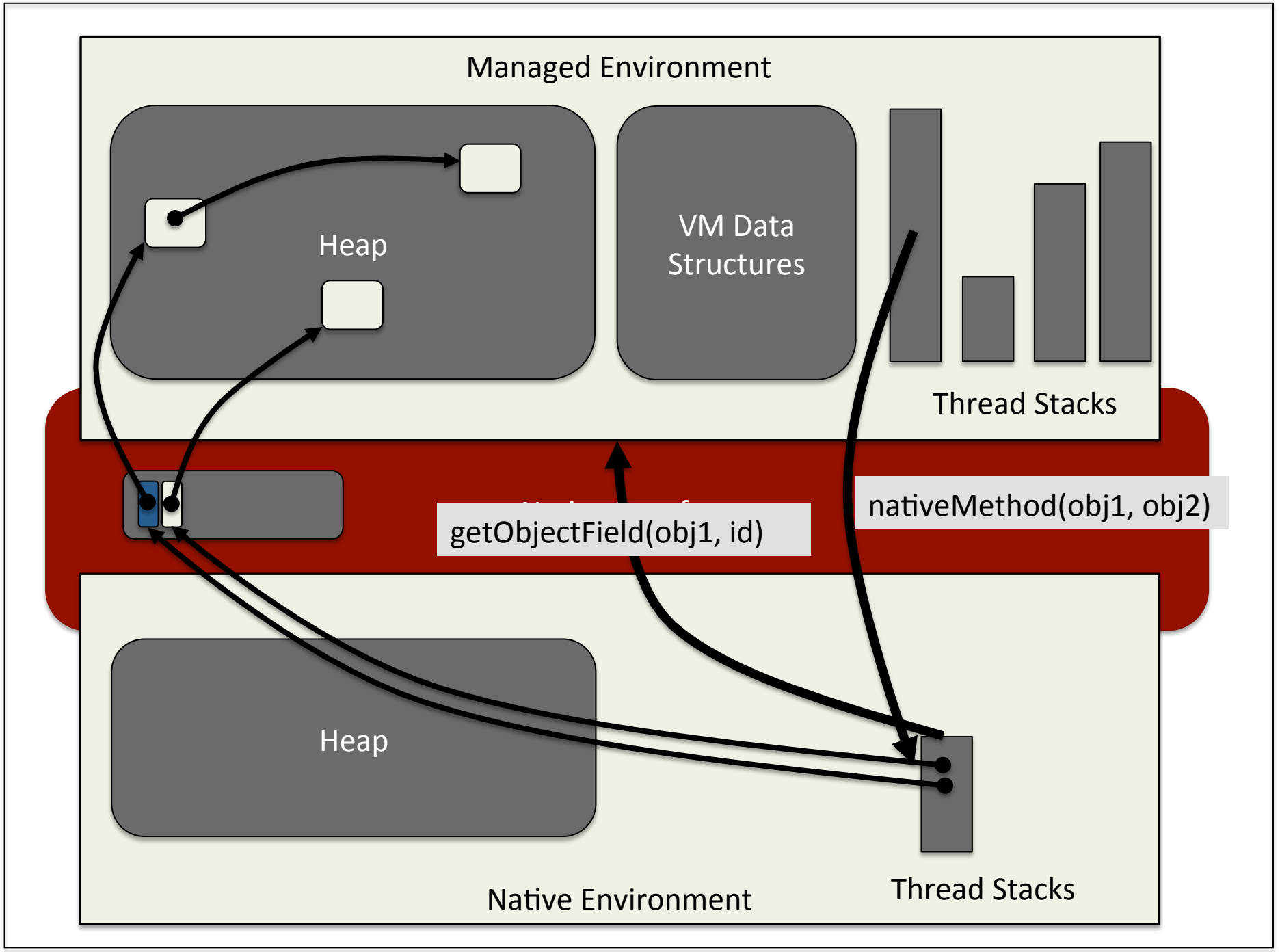

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
nativeMethod(obj1, obj2)
Heap
Native Environment
Thread Stacks

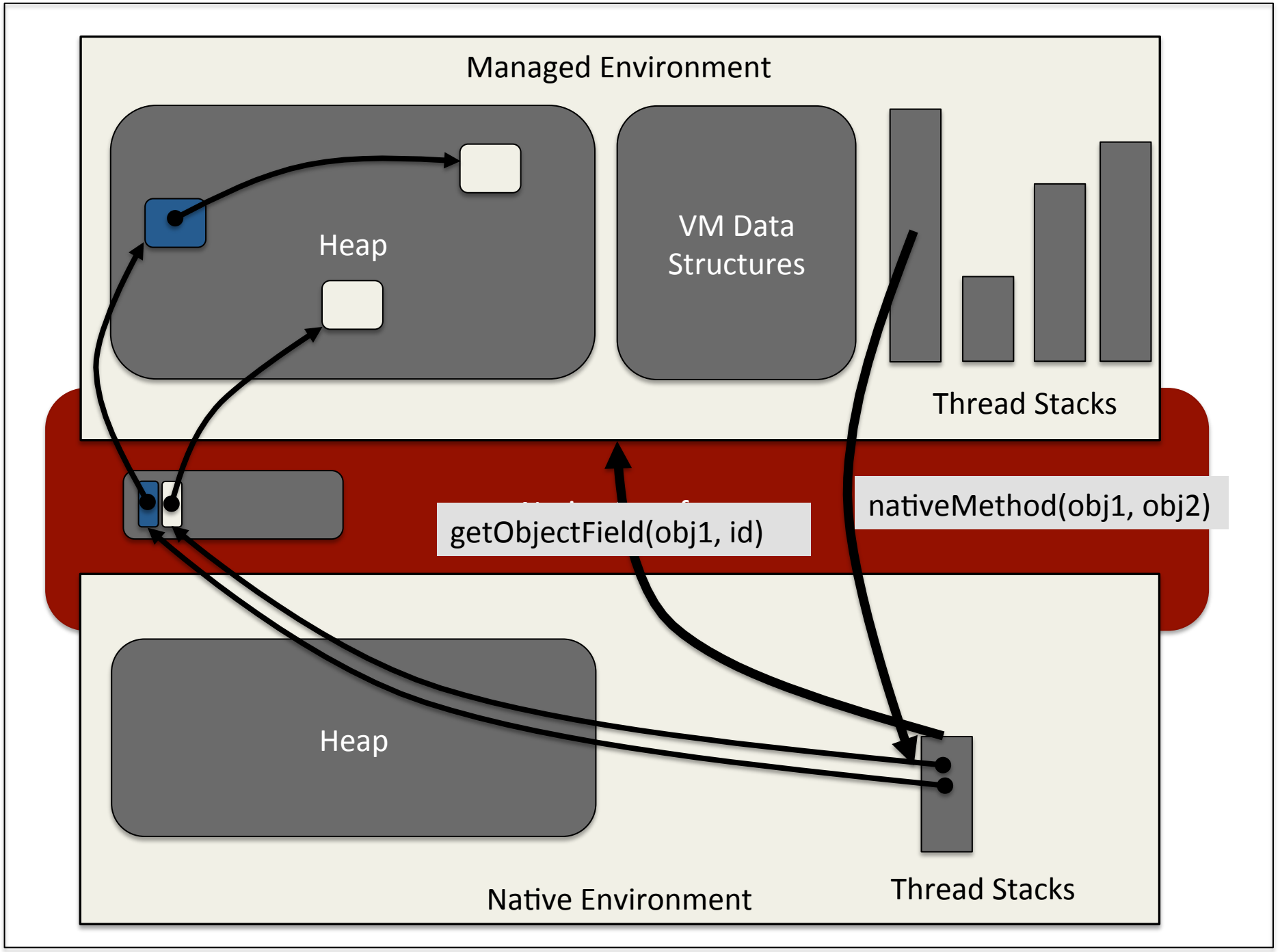

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
nativeMethod(obj1, obj2)
Heap
Native Environment
Thread Stacks

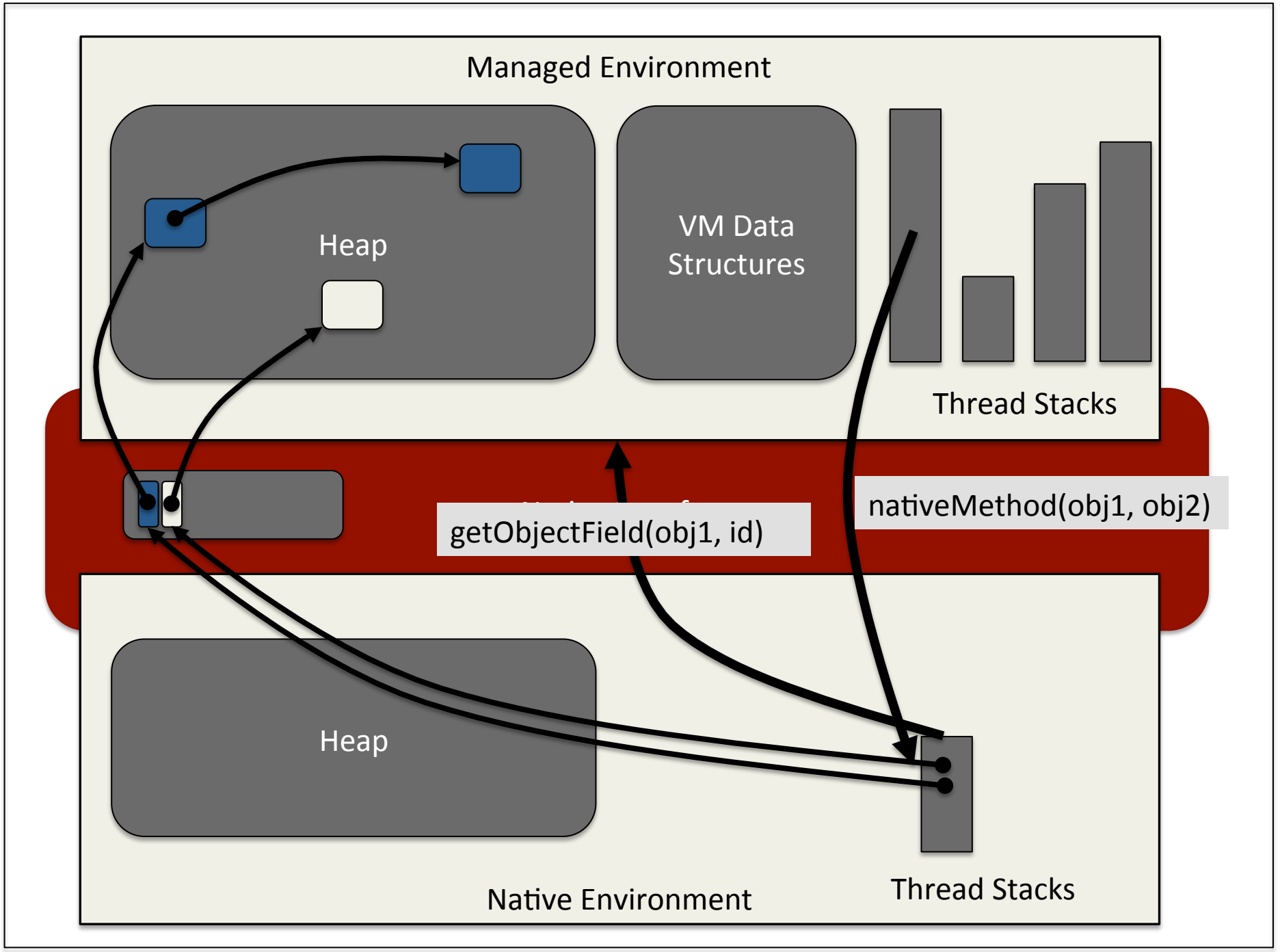

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
nativeMethod(obj1, obj2)
Heap
Thread Stacks
Native Environment

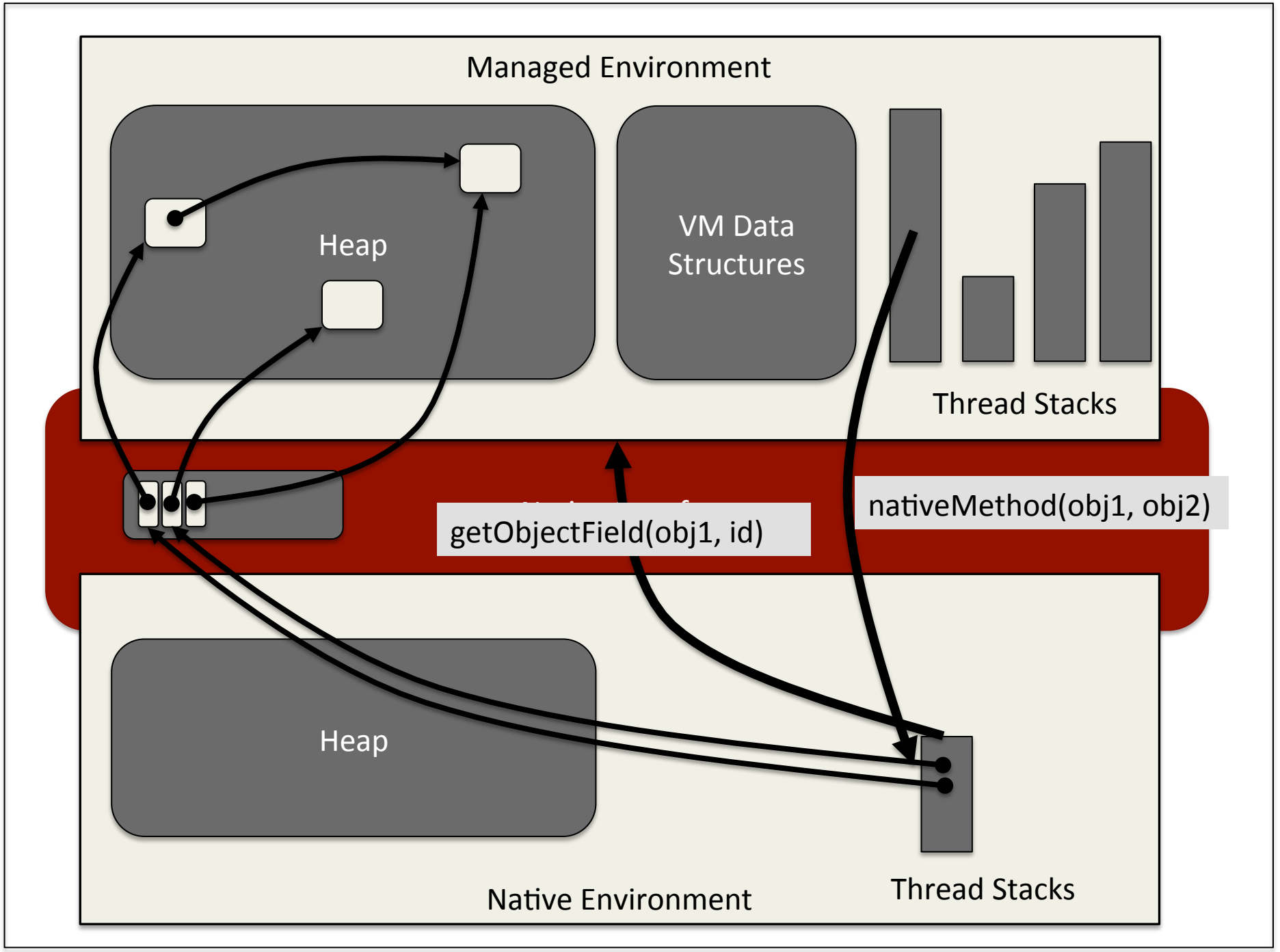

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
nativeMethod(obj1, obj2)
Heap
Native Environment
Thread Stacks

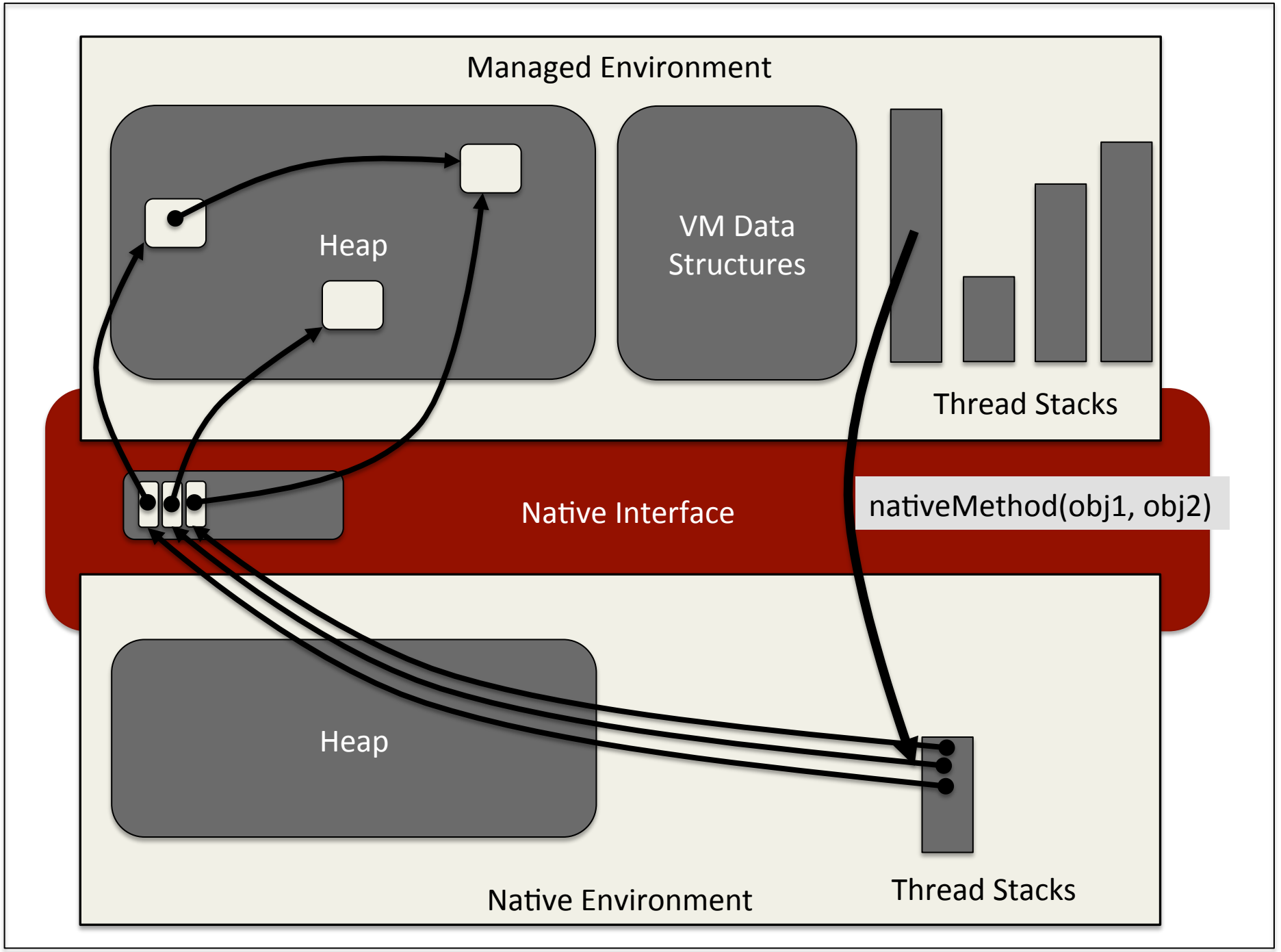

Managed Environment
Heap
VM Data Structures
Thread Stacks
nativeMethod(obj1, obj2)
getObjectField(obj1, id)
Heap
Native Environment
Thread Stacks

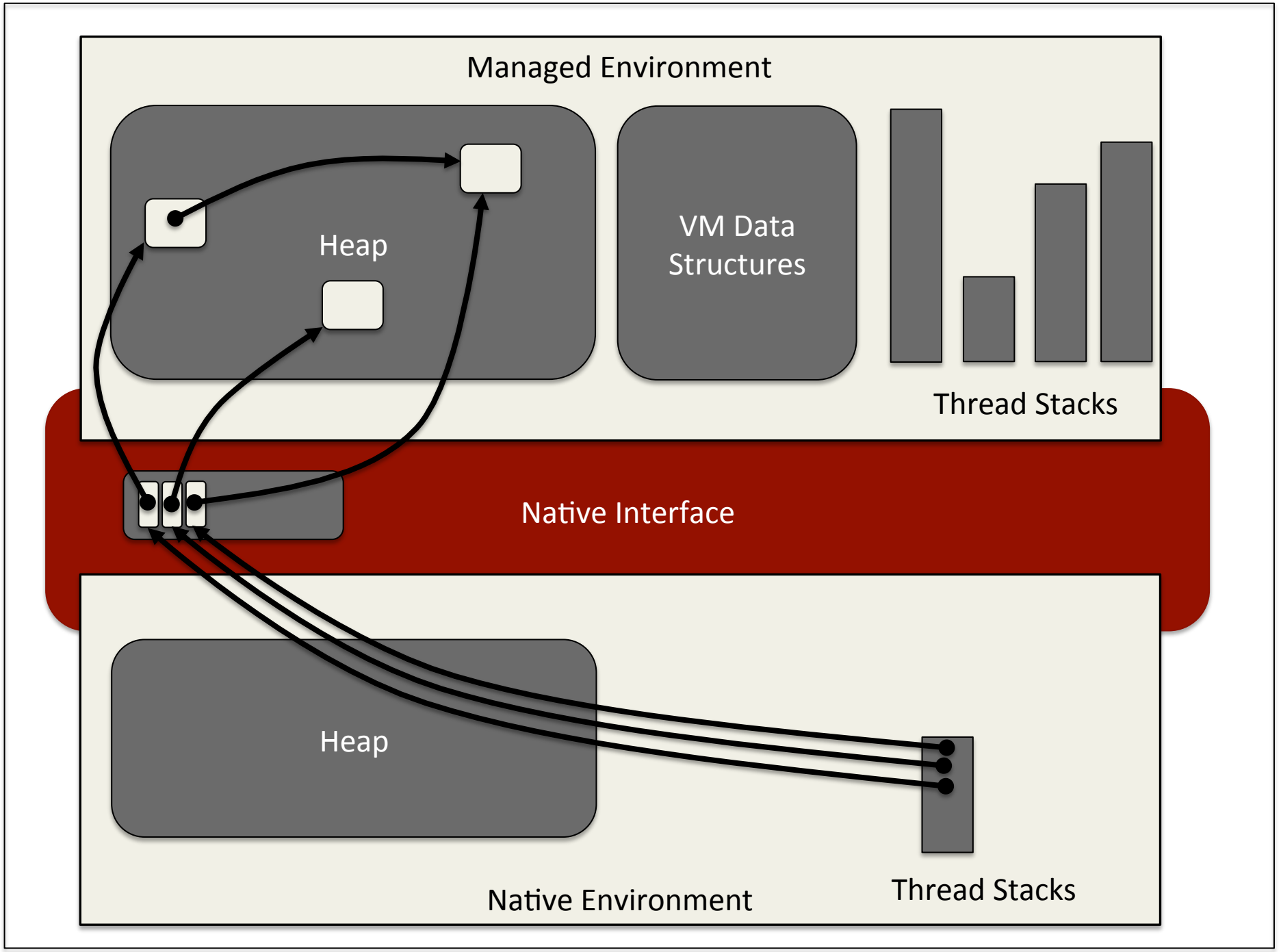

Managed Environment
Heap
VM Data Structures
Thread Stacks
nativeMethod(obj1, obj2)
getObjectField(obj1, id)
Heap
Native Environment
Thread Stacks

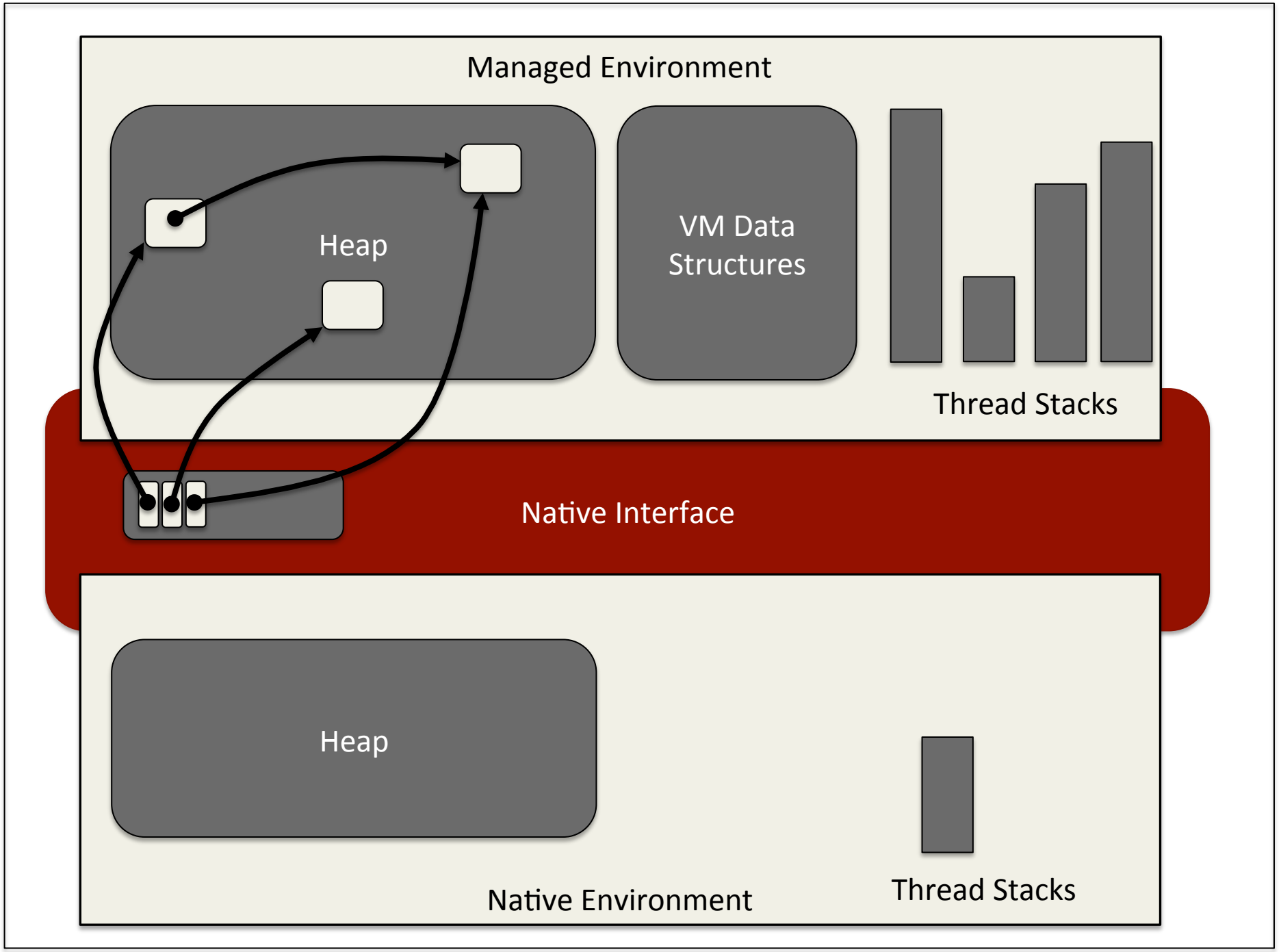

Managed Environment
Heap
VM Data Structures
Thread Stacks
nativeMethod(obj1, obj2)
getObjectField(obj1, id)
Heap
Native Environment
Thread Stacks

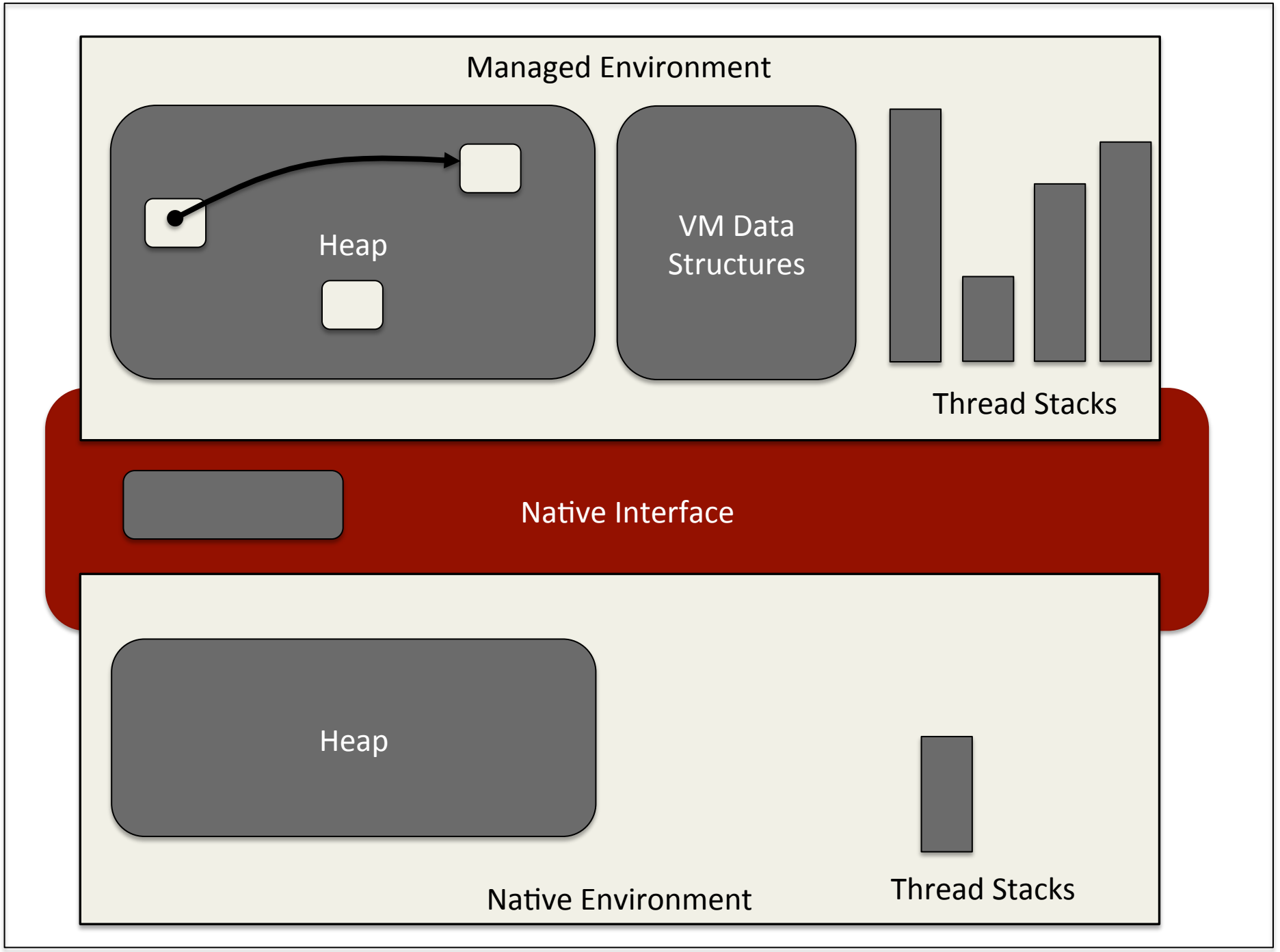

Managed Environment
Heap
VM Data Structures
Thread Stacks
nativeMethod(obj1, obj2)
getObjectField(obj1, id)
Heap
Native Environment
Thread Stacks


Managed Environment
Heap
VM Data Structures
Thread Stacks
nativeMethod(obj1, obj2)
getObjectField(obj1, id)
Heap
Native Environment
Thread Stacks


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
nativeMethod(obj1, obj2)
Heap
Native Environment
Thread Stacks


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Thread Stacks
Native Environment


Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

Local Registry Implementation

- Various implementation options
  - Hash map, linked list, etc
- Some similar characteristics to scoped memory
  - Maintain a stack of registry contexts
- Handle space is a limited resource
  - Native code can exhaust available references
  - May compact table by merging multiple references
  - Use a reference count to track reference use

Primitive Arrays

- Every managed object access uses a JNI call
  - Get and set methods track references
  - Allows objects to move behind the scenes
- This adds a significant overhead
- Overhead is particularly bad for primitive arrays
  - No handles to track on get and set
  - Array iteration is a common access pattern

Pinning or Copying

- Primitive arrays can be pinned in memory
  - While pinned, native code can access directly
  - Imposes limitations on the GC
  - Requires that arrays be laid out sequentially
- Arrays can also be copied to native memory
  - Updated contents can then be copied back
  - Most useful for smaller arrays
  - VM can optimize copy operation

Implementing Pinning

- Pinning can be implemented at zero overhead
  - If the garbage collector is non-moving
  - And the VM lays out arrays sequentially
- Neither condition is guaranteed
  - Some VMs use arraylets to break up arrays
  - Others lay arrays out in reverse order
- Can implement pinning semantics by copying
  - Introduces unexpected overhead

Native Safepoints

- Some operations require that threads be paused
- Native threads are a little more flexible
  - They work largely independent of the VM
  - Only interact at well defined points
- Define safepoints as those interactions
  - Accessing managed objects
  - Calling Java methods
  - Returning control to the VM

Exception Handling

- Exception handling in native code is tricky
  - Some native languages support exceptions
  - Others don't, or use non-standard approaches
- JNI doesn't tie in with native mechanisms

- Exceptions thrown in the VM set a flag
  - Native code must check after each call
  - Calling another method before clearing exception is bad

- `JNIEnv` lets native code raise Java exceptions

Invocation API

- Flip side of the native interface
  - Lets the JVM run embedded in a native application
- Use cases similar for Java to native interface

- Environment is similar
  - `JNI_CreateJavaVM` method returns a `JNIEnv`
  - Interactions follow the same rules

- Native code can eventually destroy the VM
  - Individual threads can detach
  - Main thread can unload the VM context