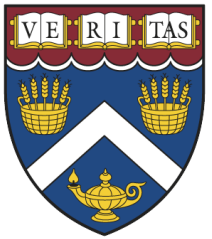
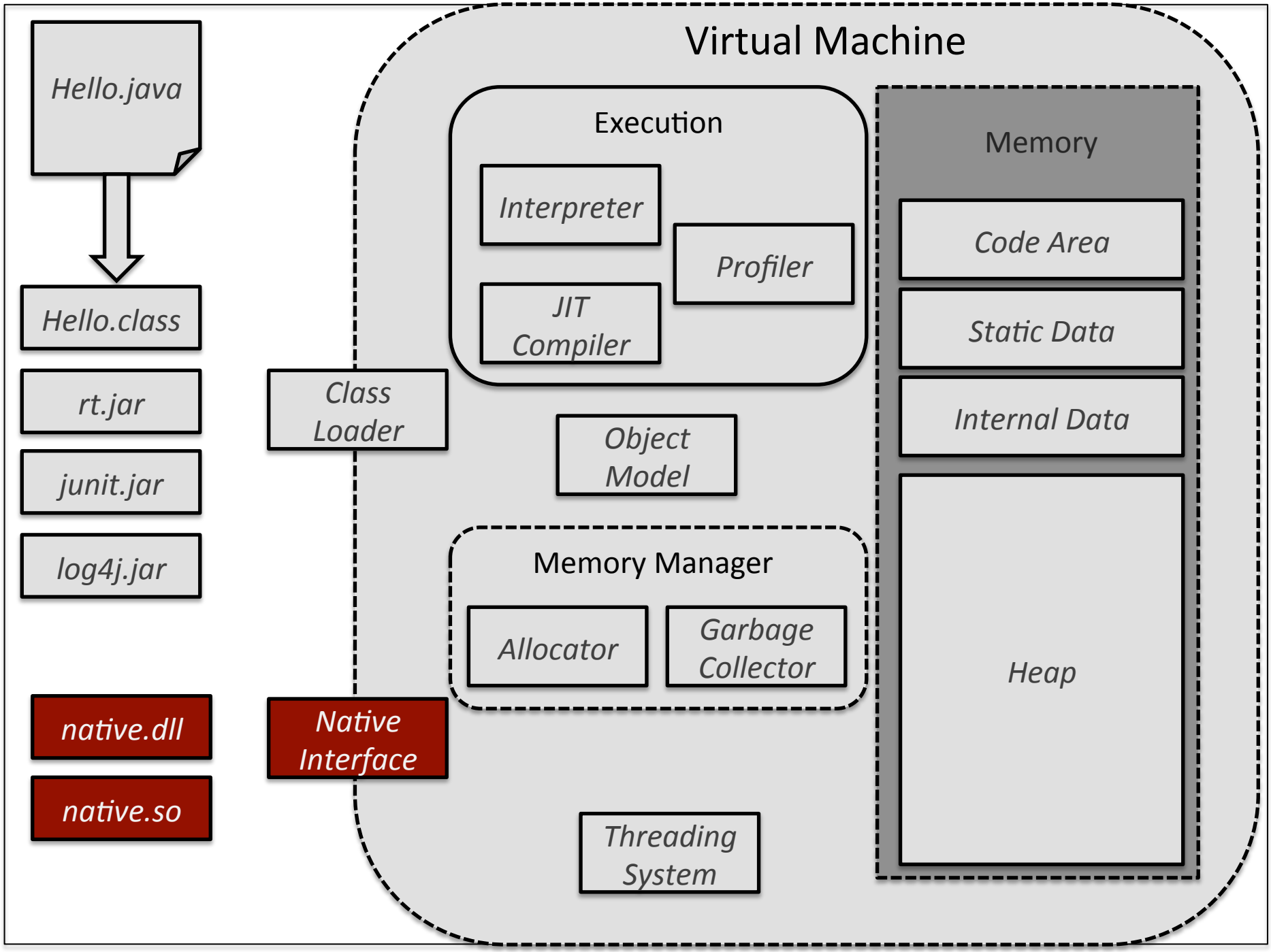
# Native Code




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

# Native Code

- Managed language not always enough
  - Want to integrate with existing code base
  - Want better performance for a critical section
  - Want to take advantage of platform features
- Most managed runtimes have a native interface
  - Also known as a Foreign Function Interface
  - Gives a way to integrate with native binaries

# Native Code Characteristics

- Not managed by the VM
  - Limited support for memory management
  - Not aware of VM data structures and conventions
  - Uses direct pointers to memory
- Not compiled by the JIT
  - Doesn't have barriers, safepoints, stack maps built in
- May not be modifiable
  - Can contain platform-specific code

# The Java Native Interface

- Part of the language and JVM specifications
- Defines interactions between native and VM
- Requires a structure from the native code
  - Native wrappers around existing libraries
- Use the javah tool to generate header files

```
public class JniExample {

  public static void main(final String[] args) {
    System.loadLibrary("jniExample");
    JniExample obj = new JniExample();
    obj.helloWorld();
    int sum = obj.add(17, 25);
  }

  private native void helloWorld();
  private native int add(int i, int j);

}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JniExample */

#ifndef _Included_JniExample
#define _Included_JniExample
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JniExample
 * Method:    helloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_JniExample_helloWorld
  (JNIEnv *, jobject);

/*
 * Class:     JniExample
 * Method:    add
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_JniExample_add
  (JNIEnv *, jobject, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

# JniExample.h

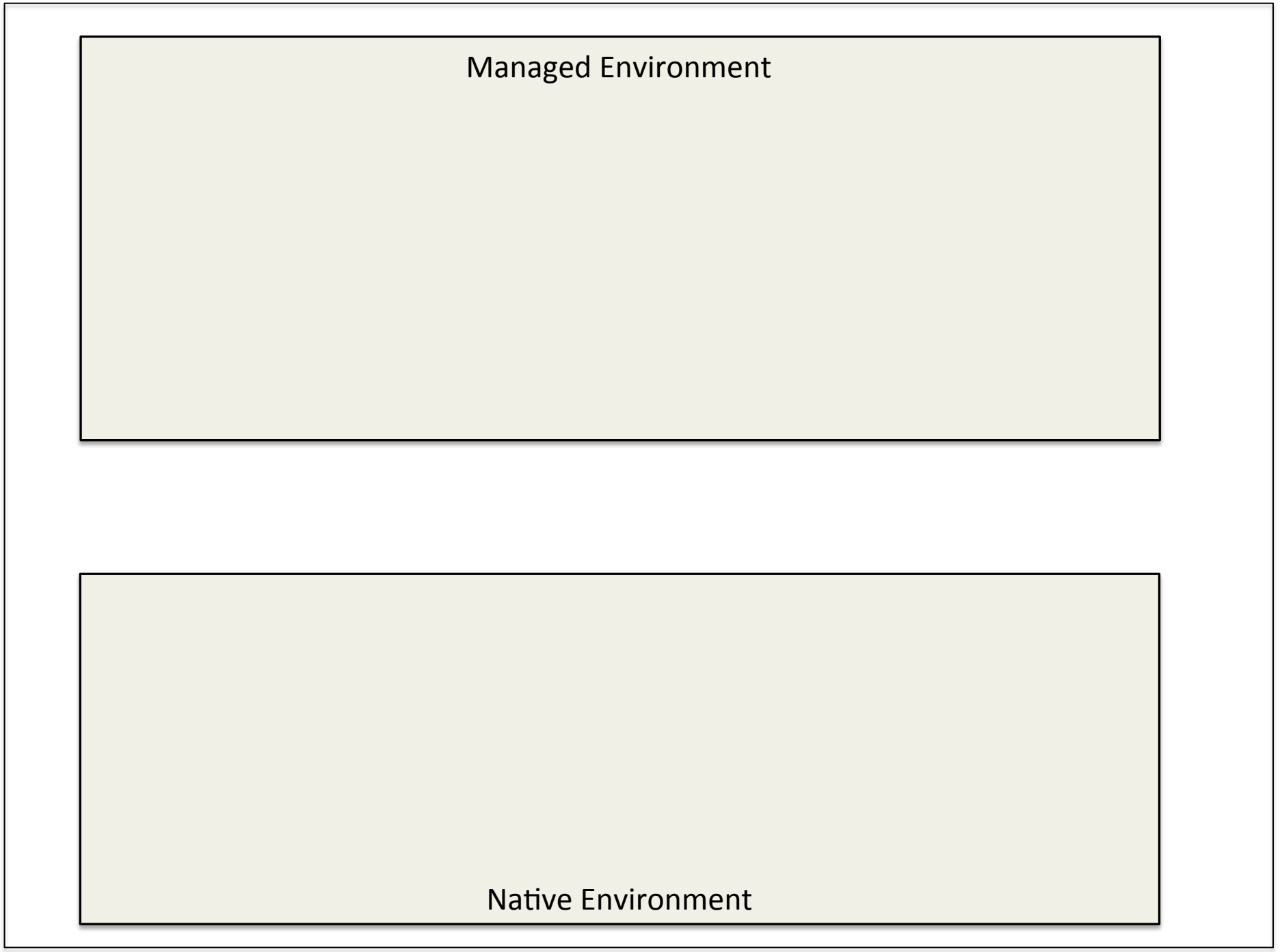
- C/C++ header file
- Gives definitions for our native functions
- Must rebuild when we change native signatures
- Follows standard naming convention
  - `Java_<class>_<method>`
  - Overloaded methods have signature appended

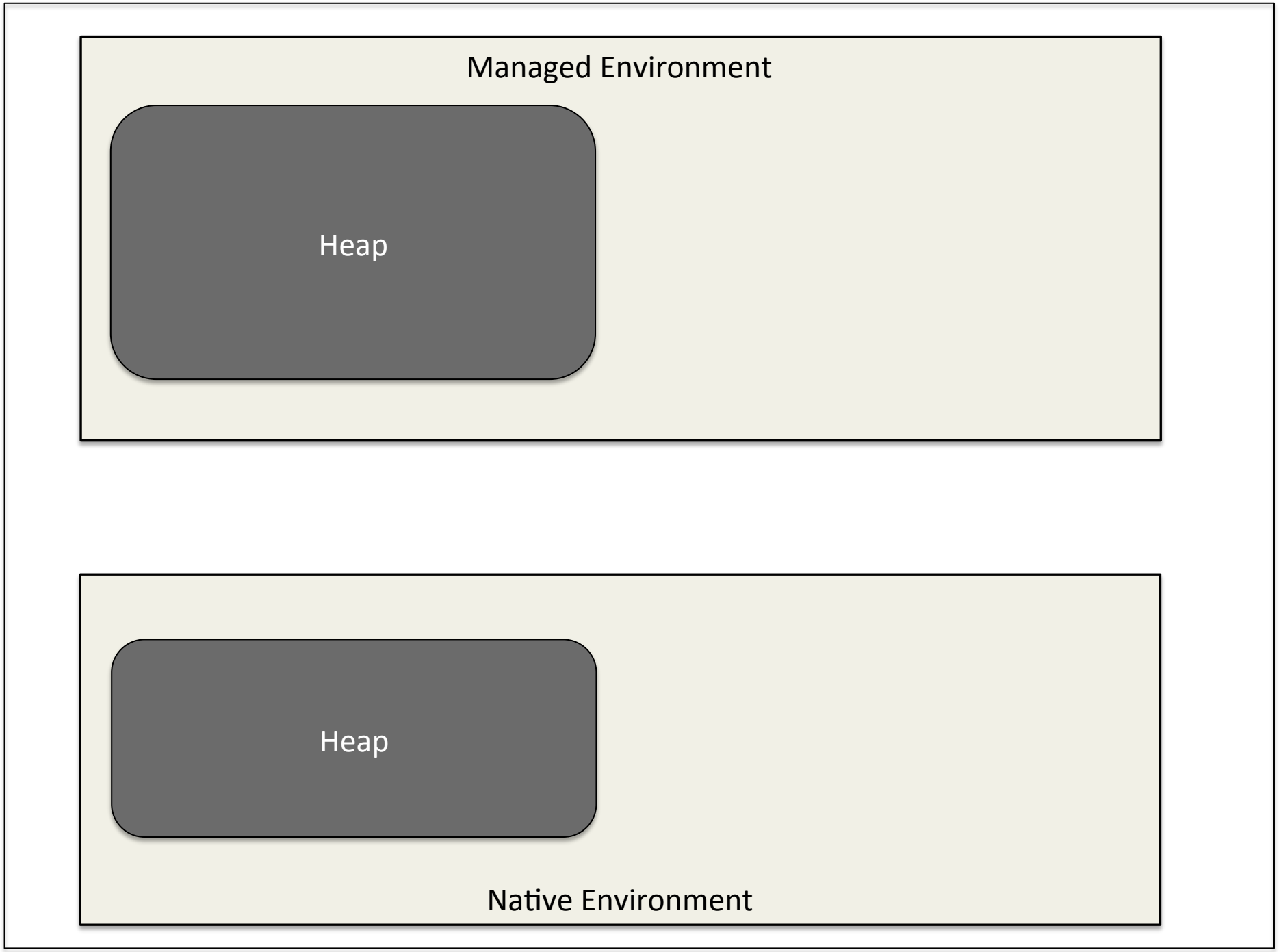
```
#include <jni.h>
#include <stdio.h>
#include "JniExample.h"


JNIEXPORT void JNICALL
  Java_JniExample_helloWorld(JNIEnv * env, jobject thisObj) {
   printf("Hello World\n");
   return;
}

JNIEXPORT jint JNICALL
  Java_JniExample_add(JNIEnv * env, jobject thisObj,
                      jint i, jint j) {
    return i + j;
}
```

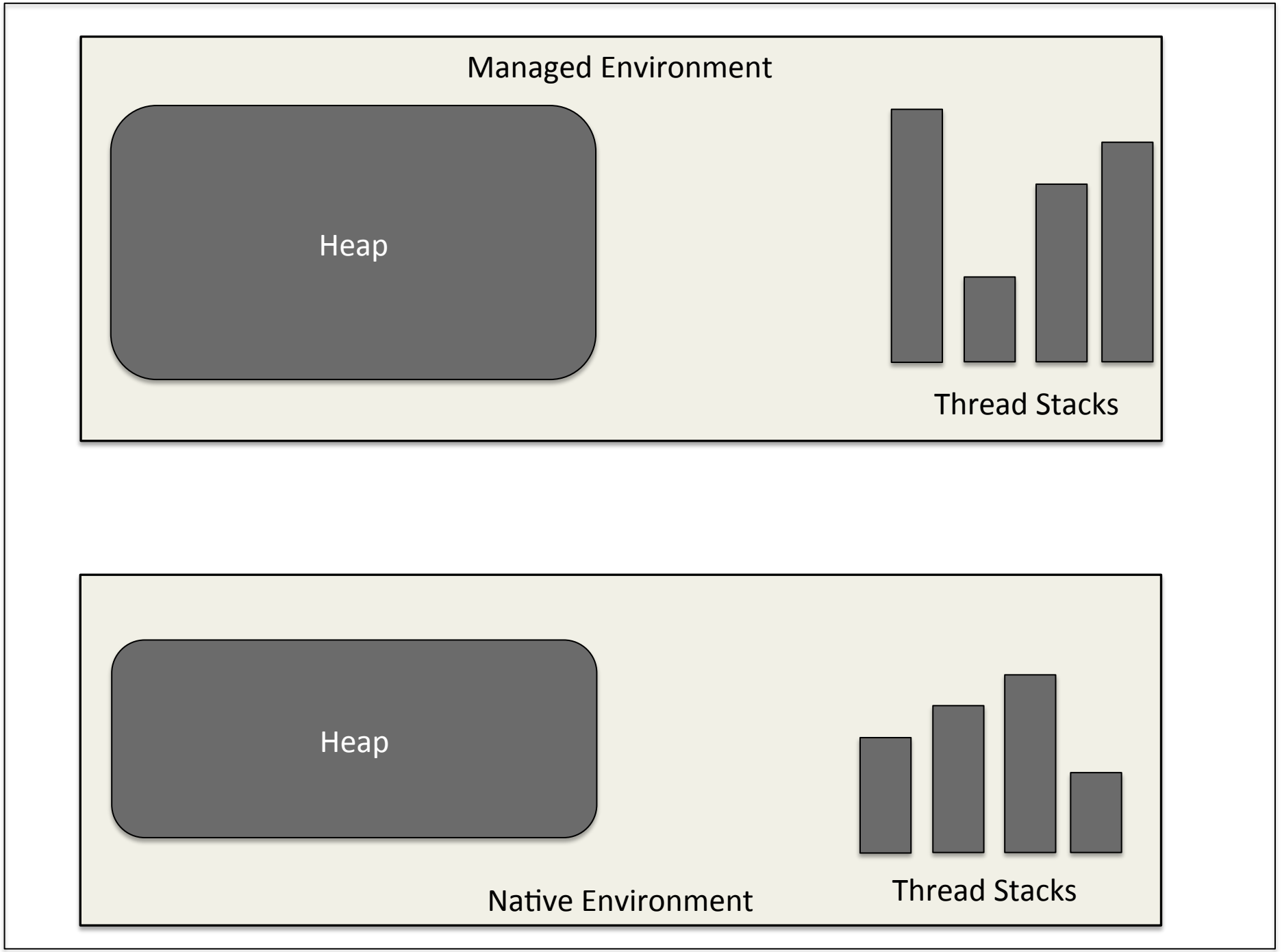
```
$ gcc -c -I/System/Library/Frameworks/JavaVM.framework/
Versions/A/Headers/ JniExample.c

$ libtool -dynamic -lSystem JniExample.o -o
libJniExample.dylib

$ java JniExample

Hello World
The answer is 42
```

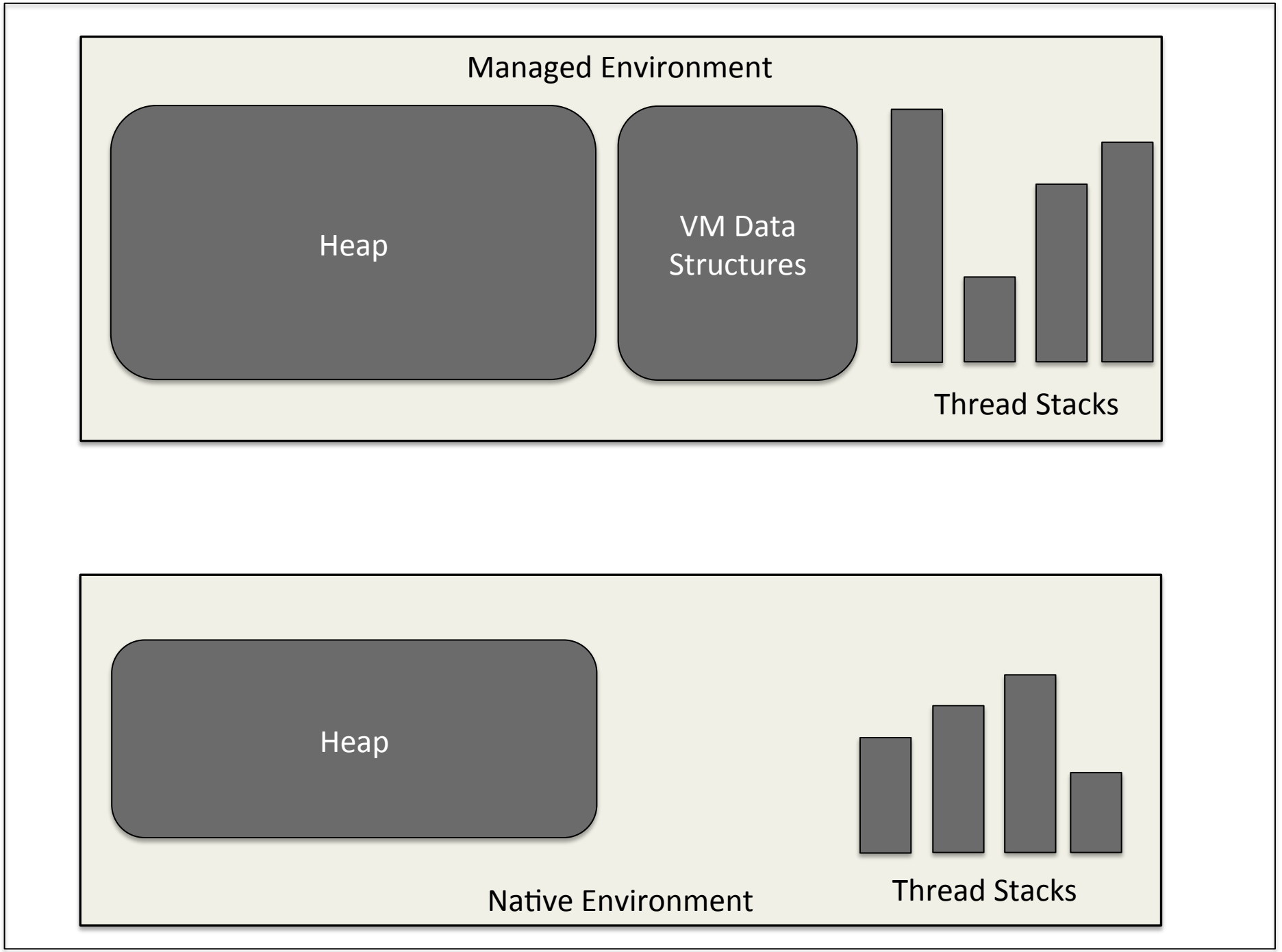
# Shared Libraries

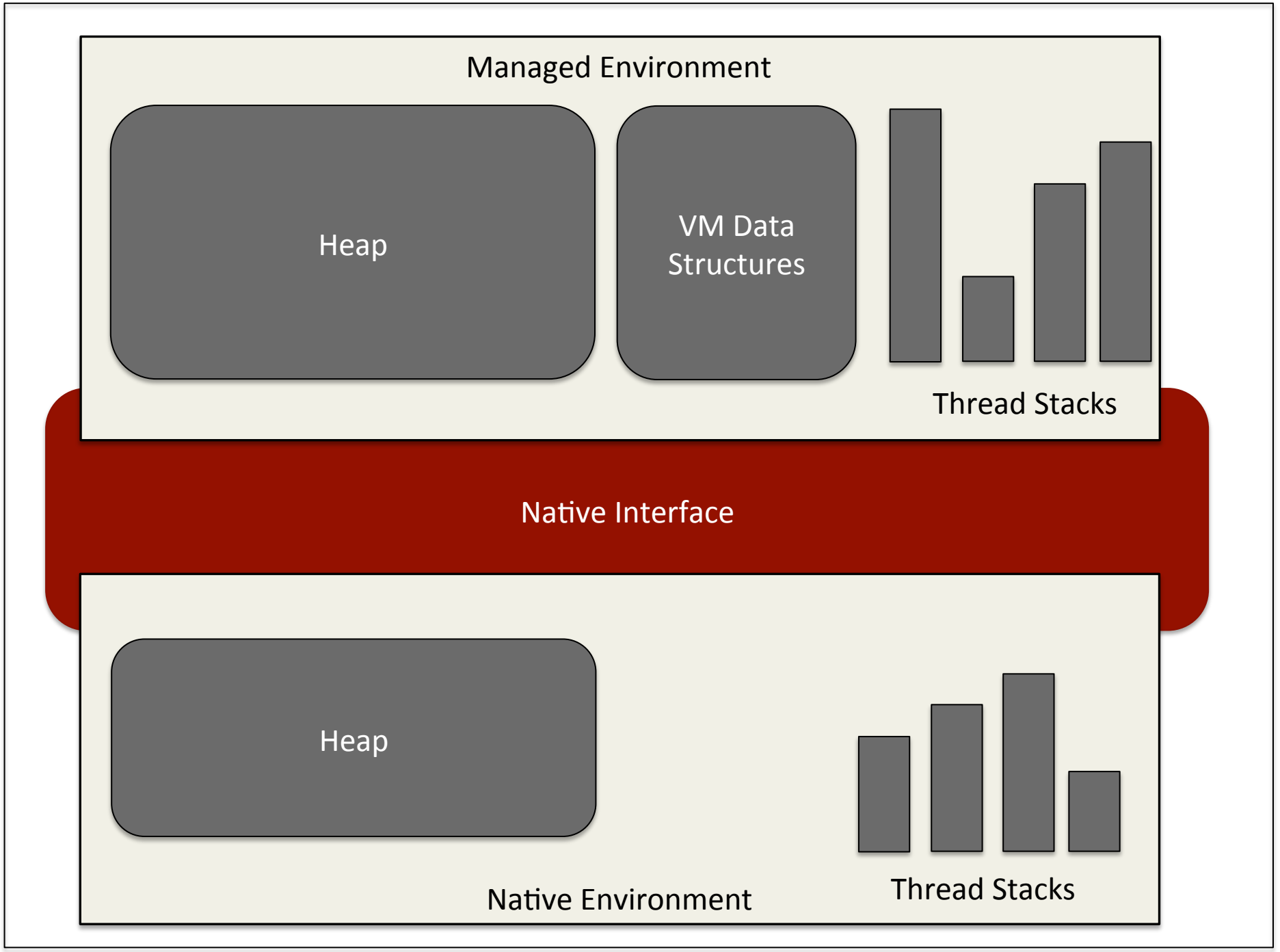
- Format of native library is platform specific
  - `.so`, `.dylib`, `.dll` etc
- Native code built for a specific environment
  - Can optimize for local hardware resources
  - Must be built separately for each platform
- JDK provides standard header files


Managed Environment
Native Environment


Managed Environment
Heap
Heap
Native Environment


Managed Environment
Heap
Thread Stacks
Heap
Native Environment
Thread Stacks


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


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

# Bi-Directional Interface

- Managed to native code
  - `native` keyword indicates external function
  - Loading mechanism to link and import libraries
- Native to managed
  - More involved interface
  - Type mapping
  - Reference to `this` object
  - Reference to `JNIEnv` interface

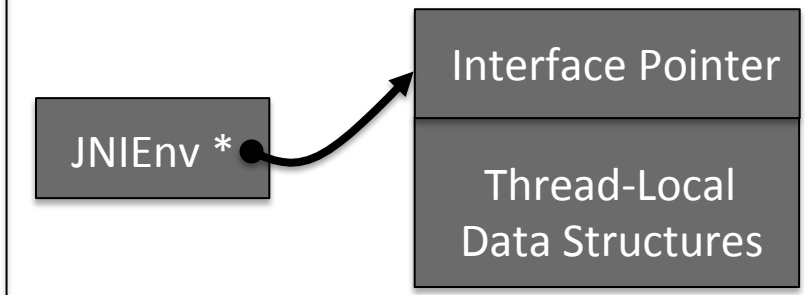
# Type Mapping

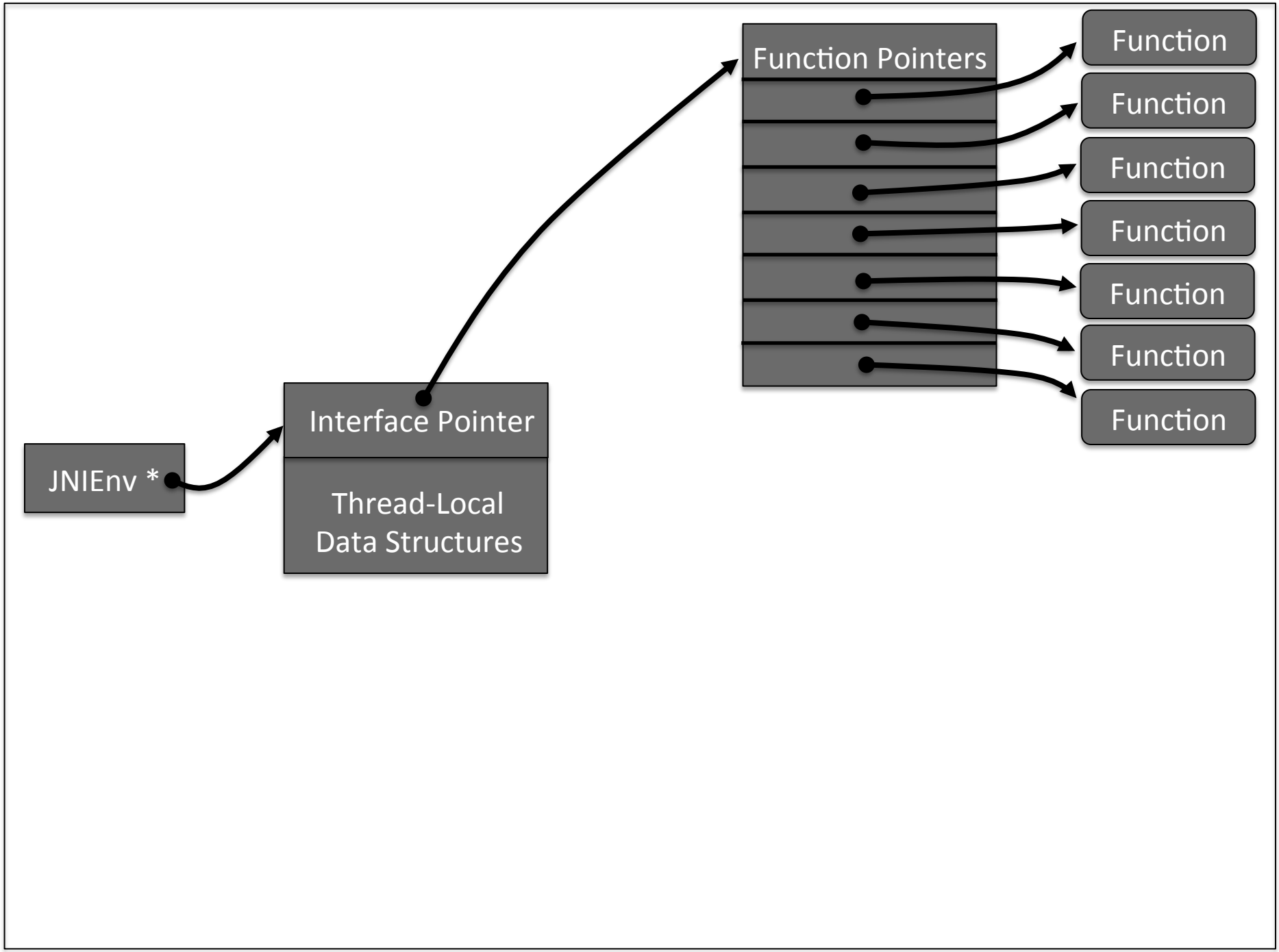
- Primitive types have direct mapping
  - `jboolean`, `jbyte`, `jchar`, etc
  - Sizes are fixed regardless of architecture
- References are of type `jobject`
  - C++ builds class hierarchy
  - C `typedefs` all classes to be `jobject`
- JNI methods expose object fields and methods

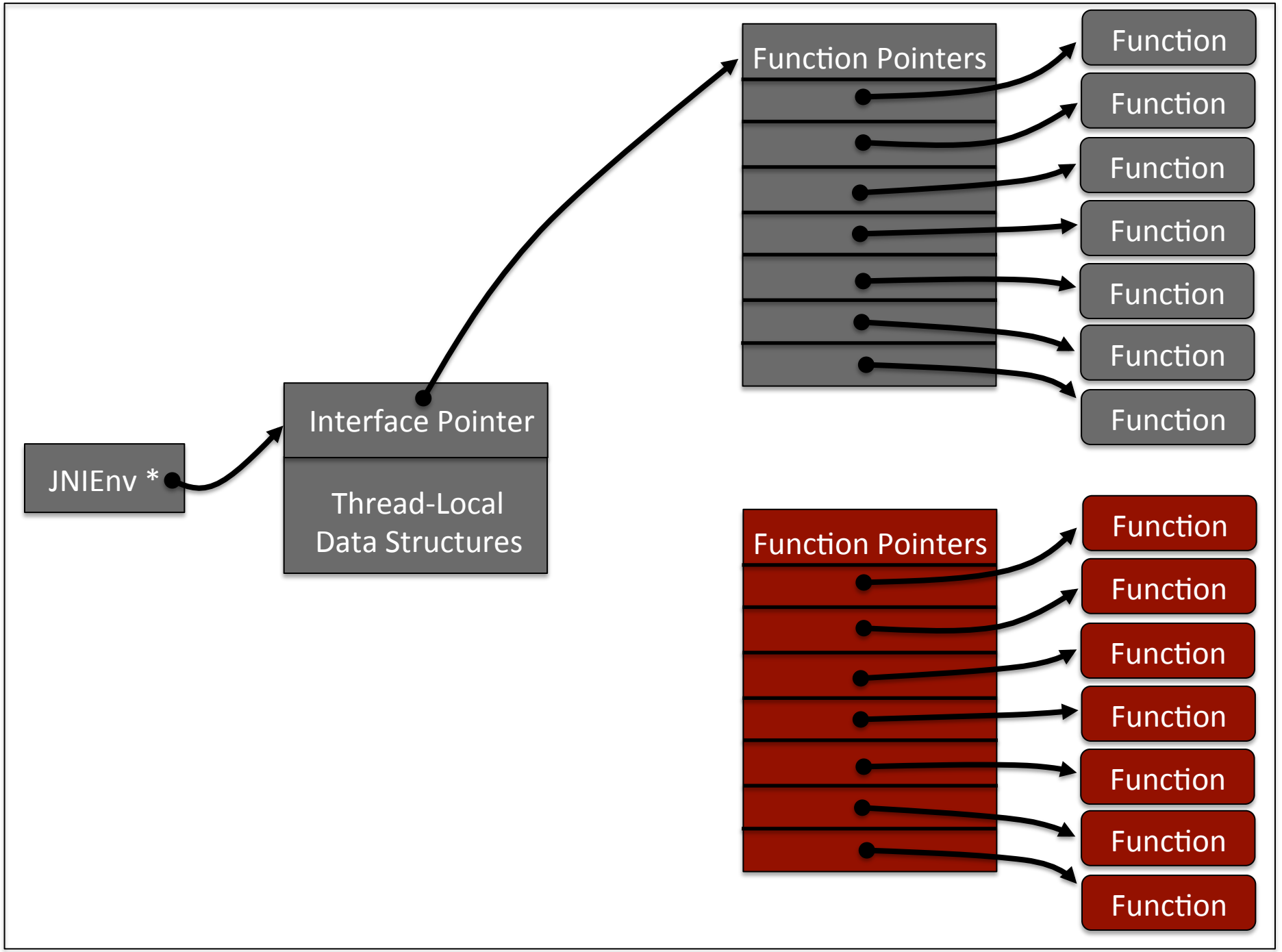
# JNIEnv Interface

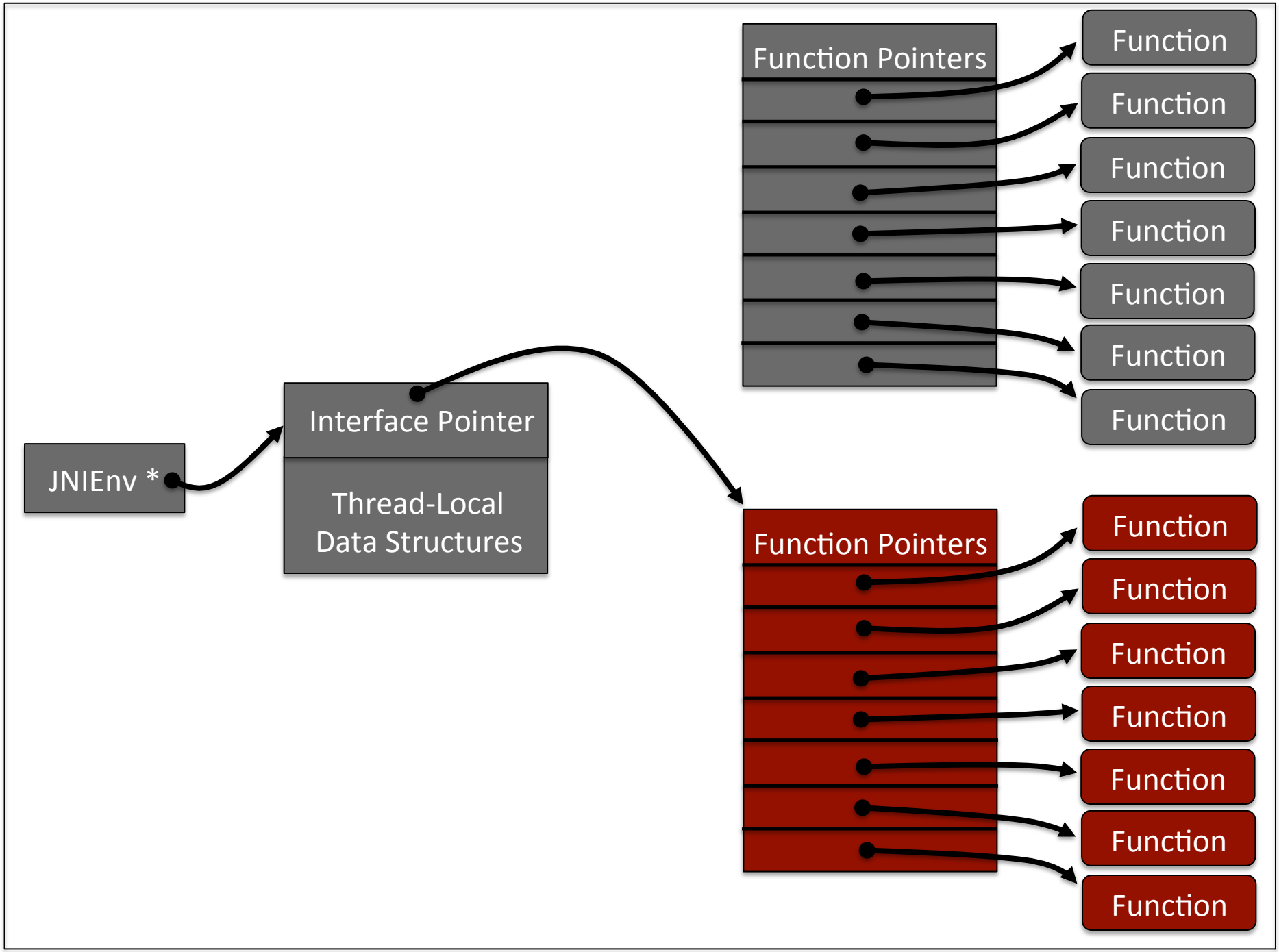
- Provides access to JVM internals
  - Class structures
  - Objects on the heap
- Lets native code invoke managed methods
- Double pointer allows implementation changes
  - -Xcheck:jni


JNIEnv *


Interface Pointer
JNIEnv *
Thread-Local
Data Structures

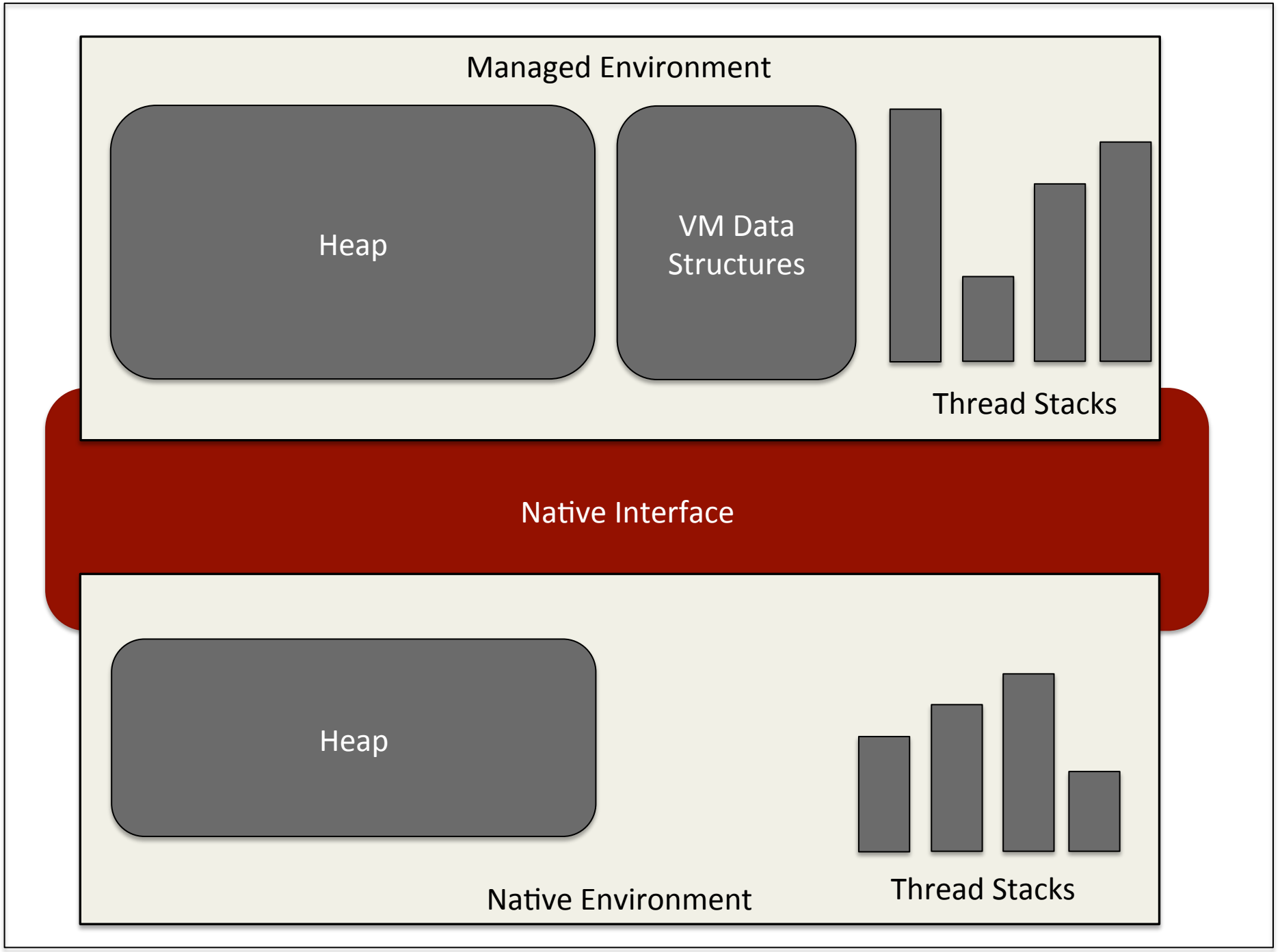

Function Pointers
Function
Function
Function
Function
Function
Function
Function
Interface Pointer
Thread-Local
Data Structures
JNIEnv *

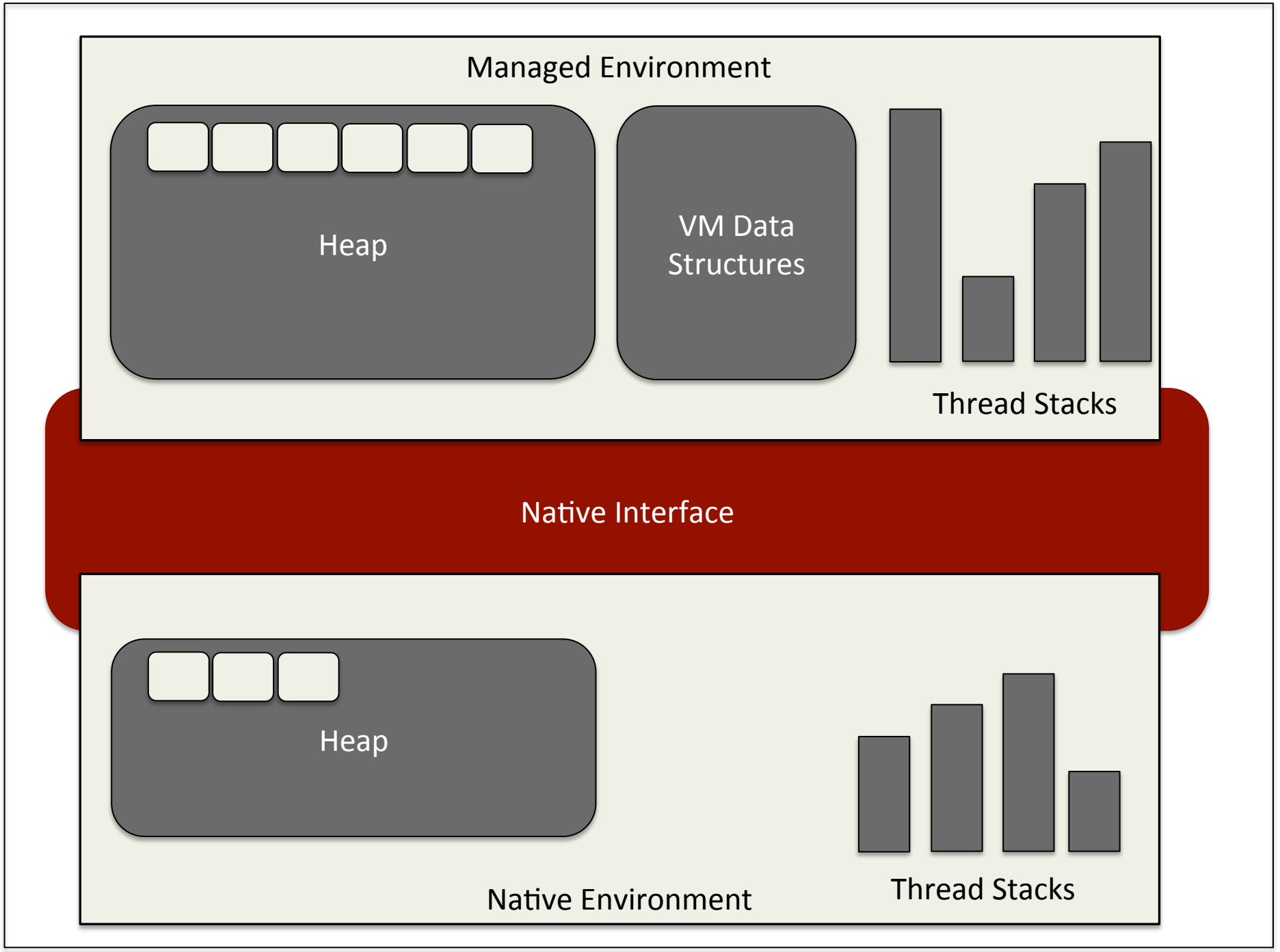

Function Pointers
Function
Function
Function
Function
Function
Function
Function
Interface Pointer
JNIEnv *
Thread-Local
Data Structures
Function Pointers
Function
Function
Function
Function
Function
Function
Function

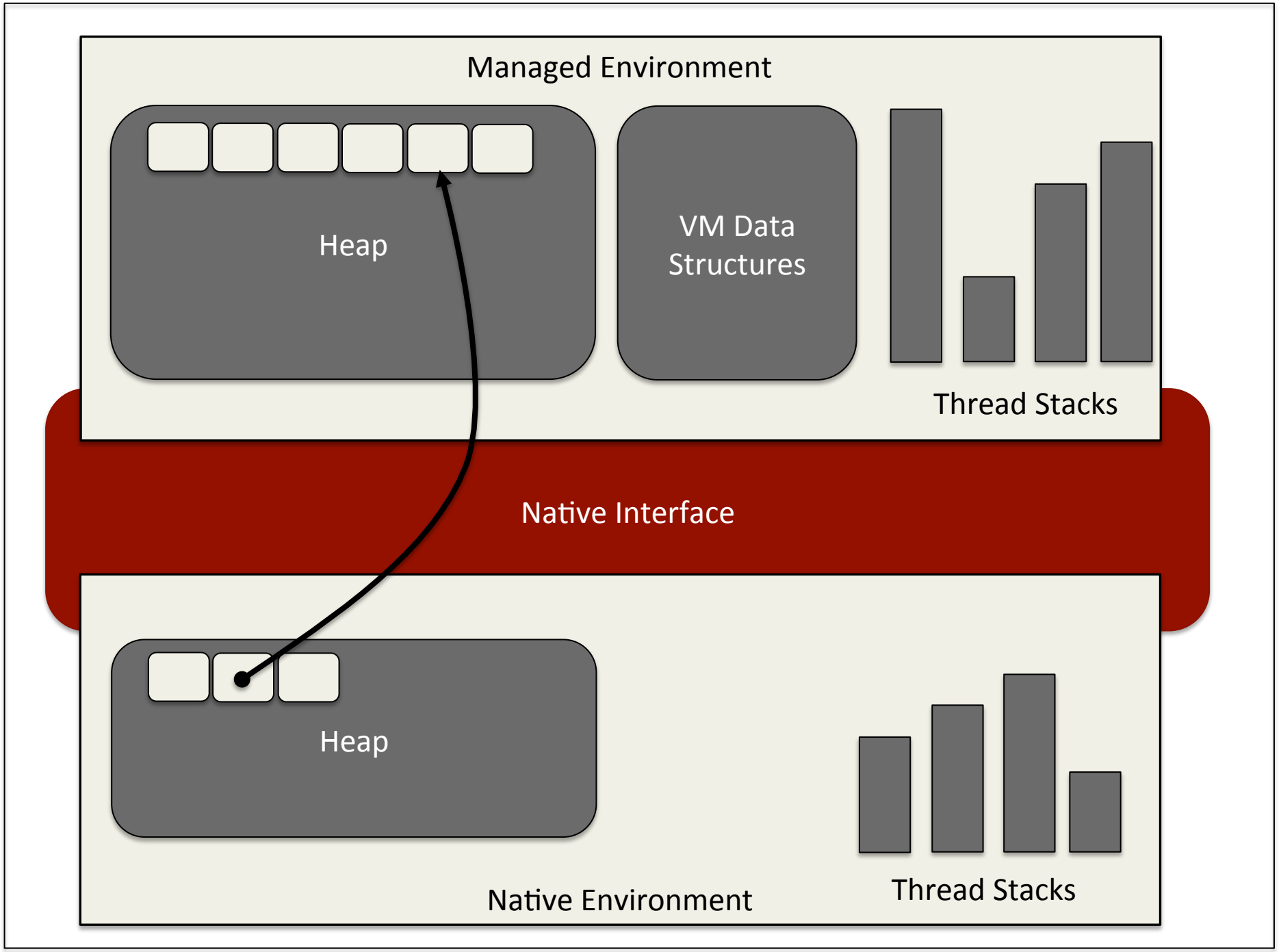

Function Pointers
Function
Function
Function
Function
Function
Function
Function
Interface Pointer
JNIEnv *
Thread-Local
Data Structures
Function Pointers
Function
Function
Function
Function
Function
Function
Function

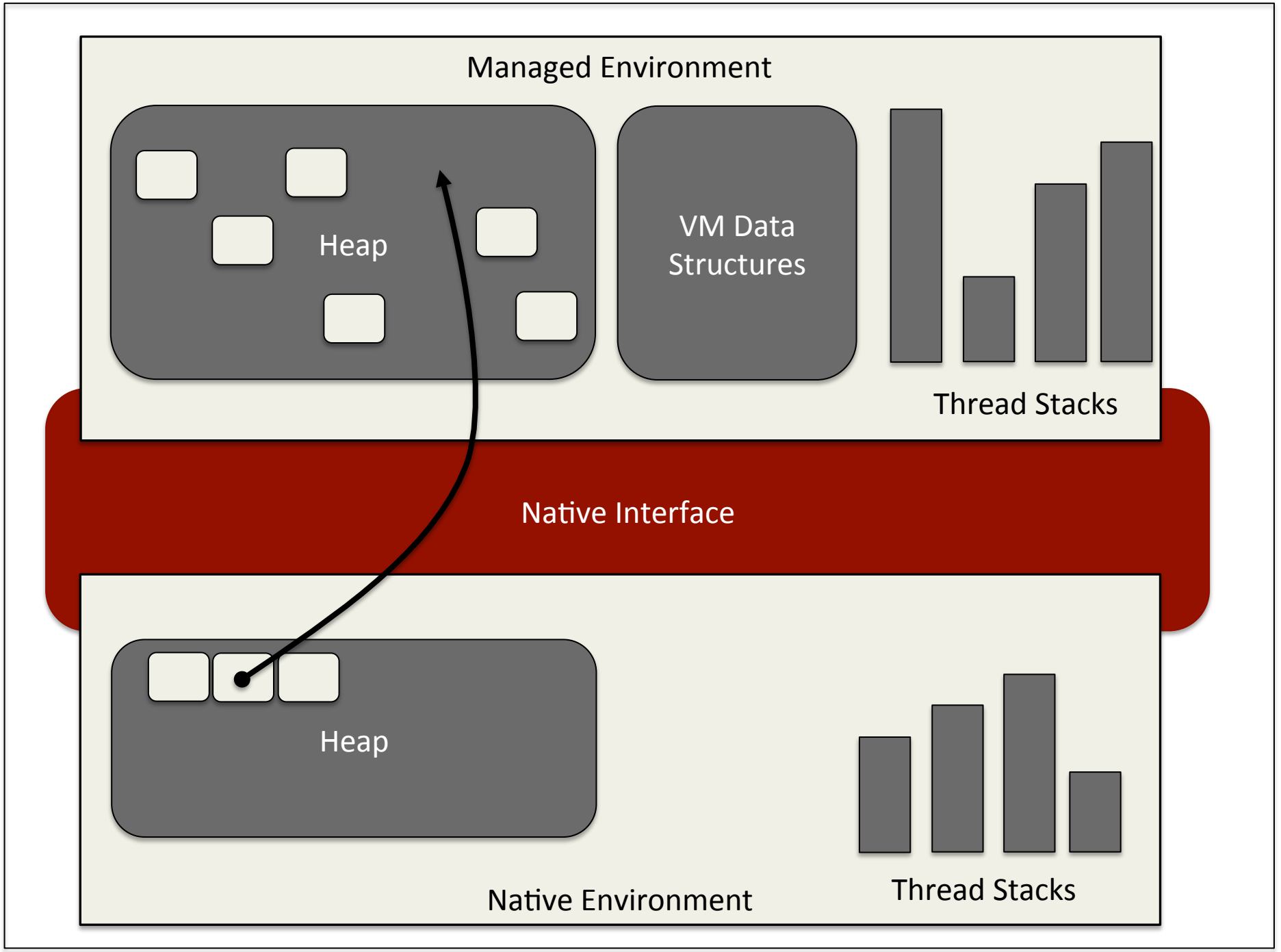
# Object References

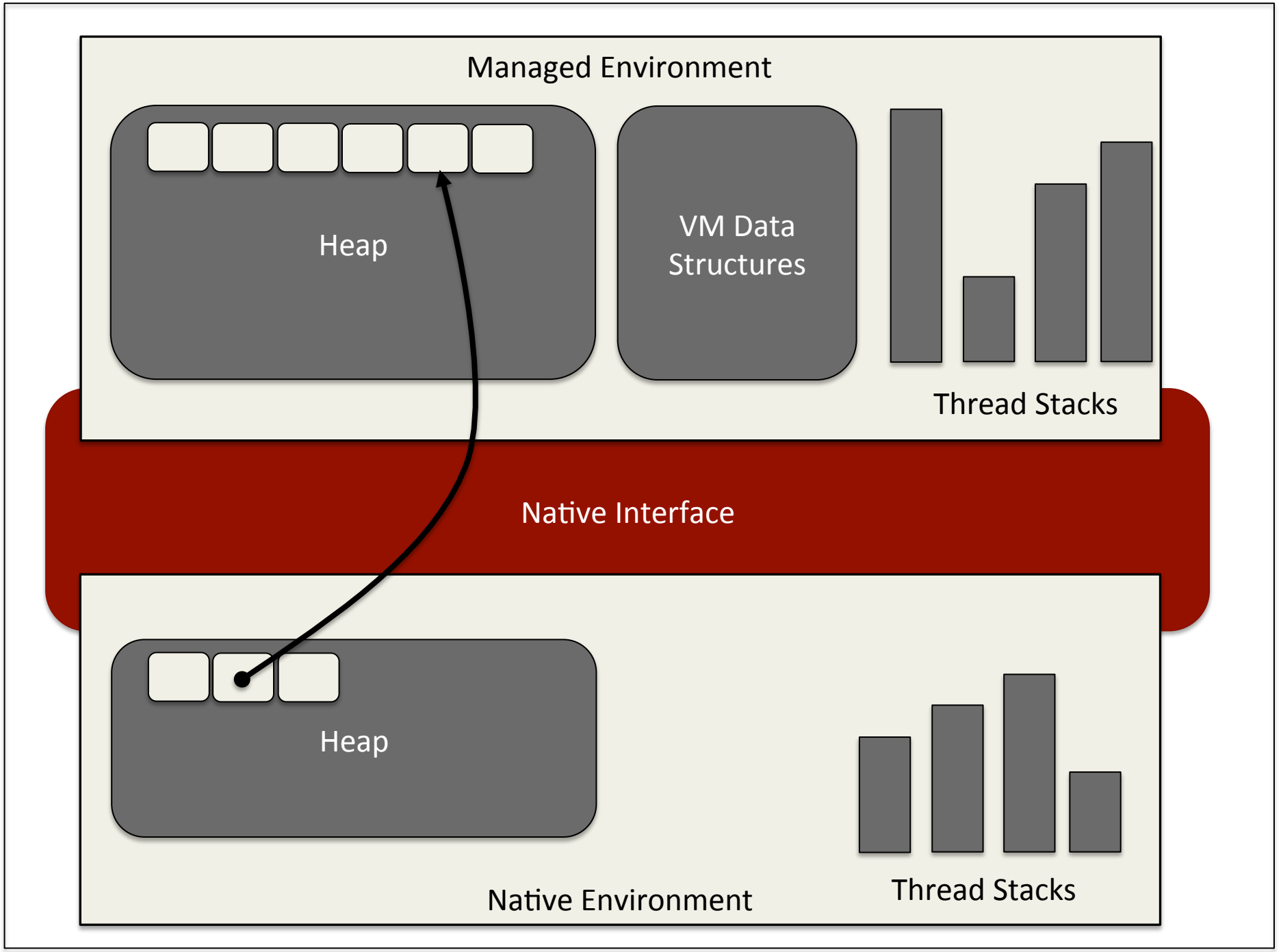
- We now know that accurate GC is important
  - Required if objects are to be moved
  - Moving is necessary for high-performance GC
- JIT compilers go to great lengths to be accurate
  - Stack maps
  - Write barriers
- We can't assume that native code does the same

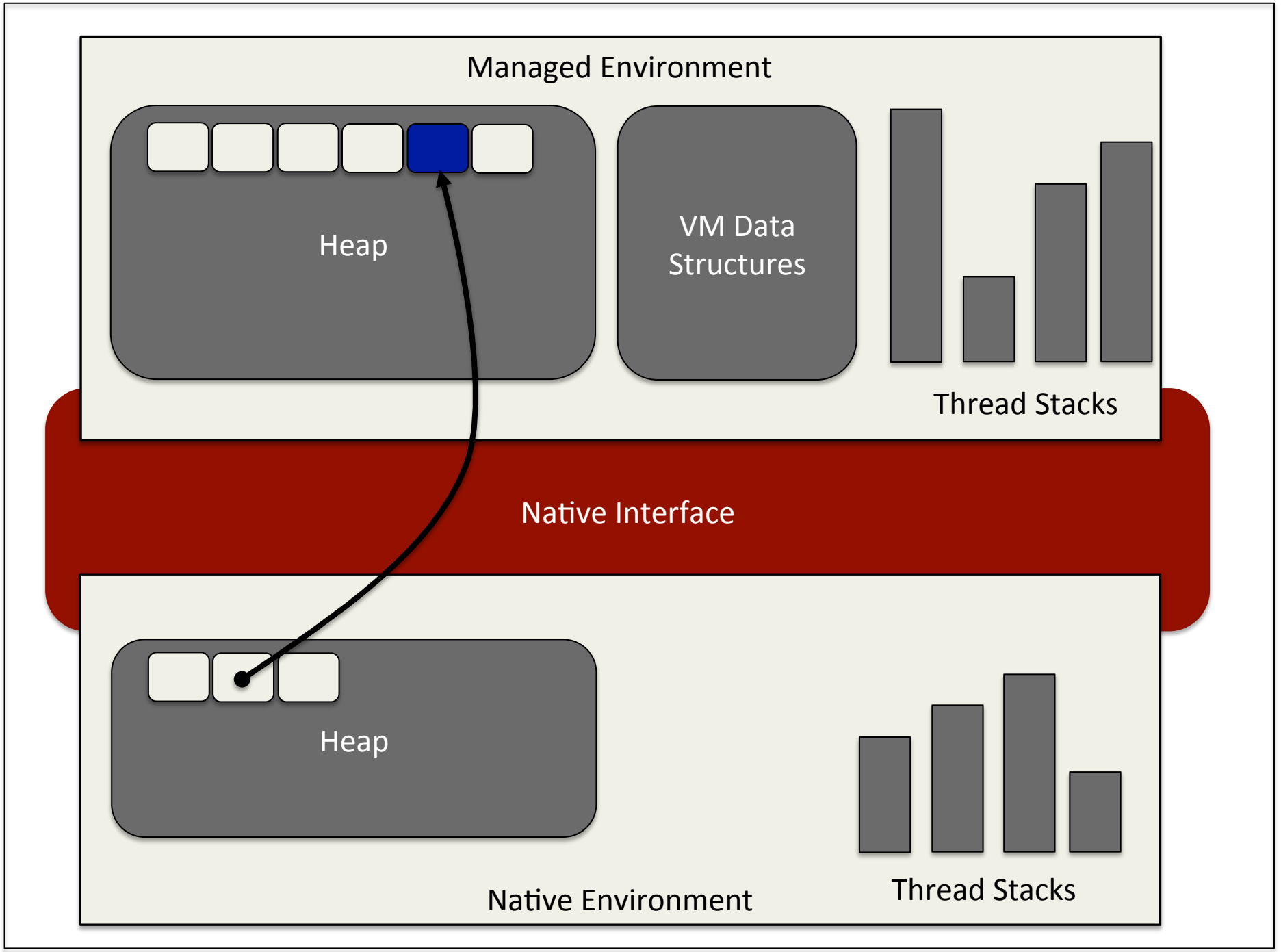

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

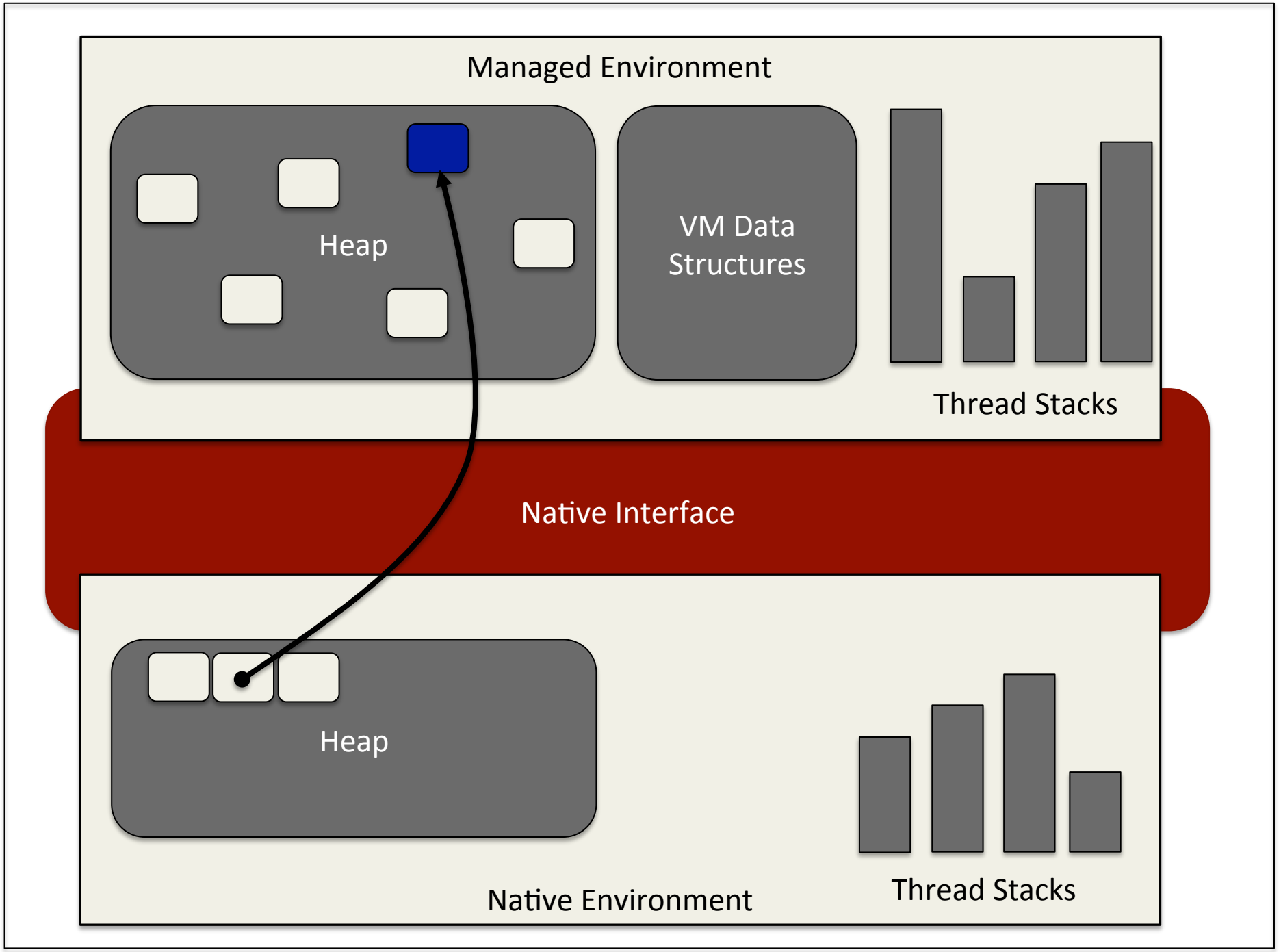

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks


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
Thread Stacks
Native Environment


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

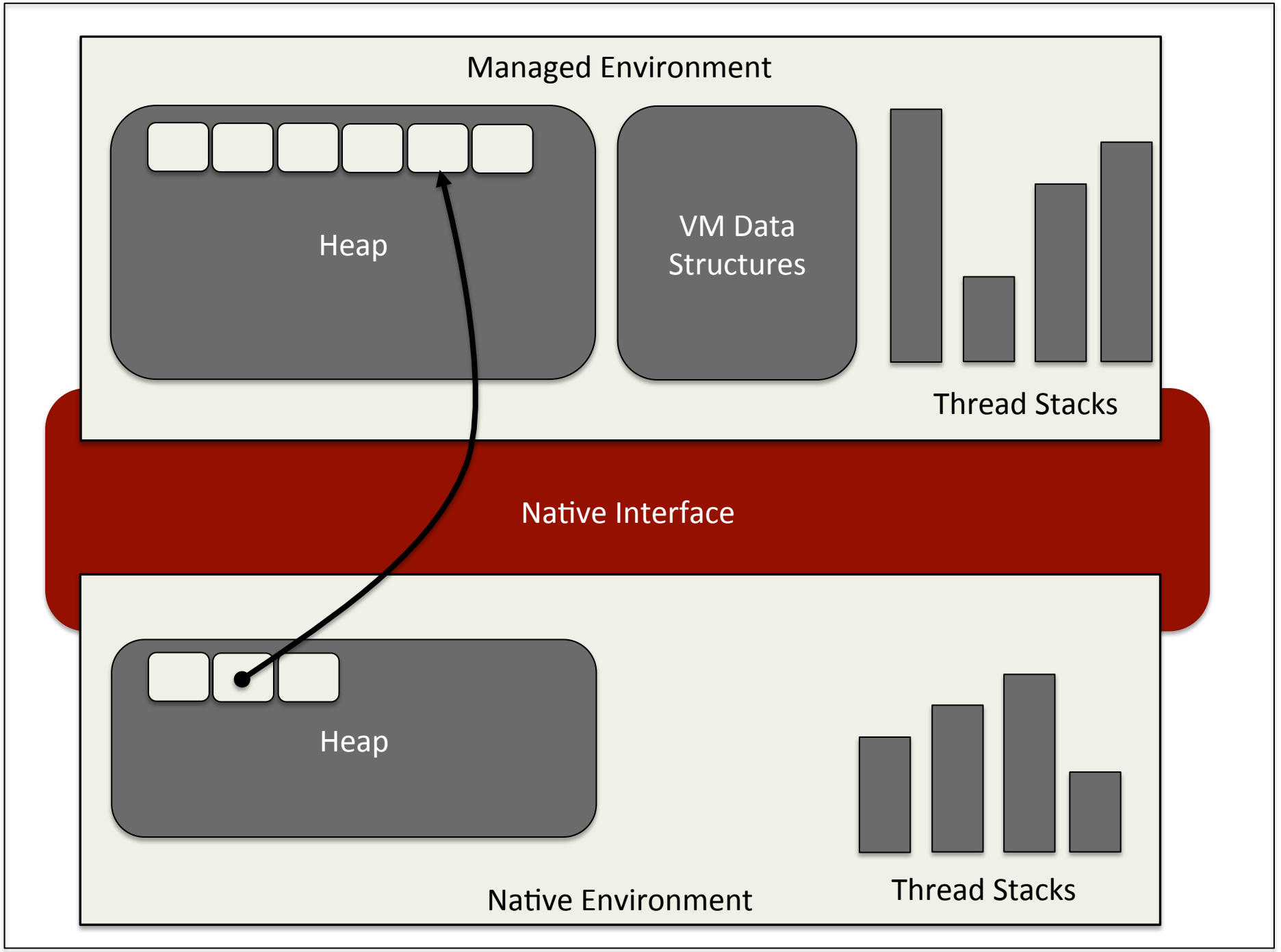

Managed Environment
Heap
VM Data Structures
Thread Stacks
Native Interface
Heap
Native Environment
Thread Stacks

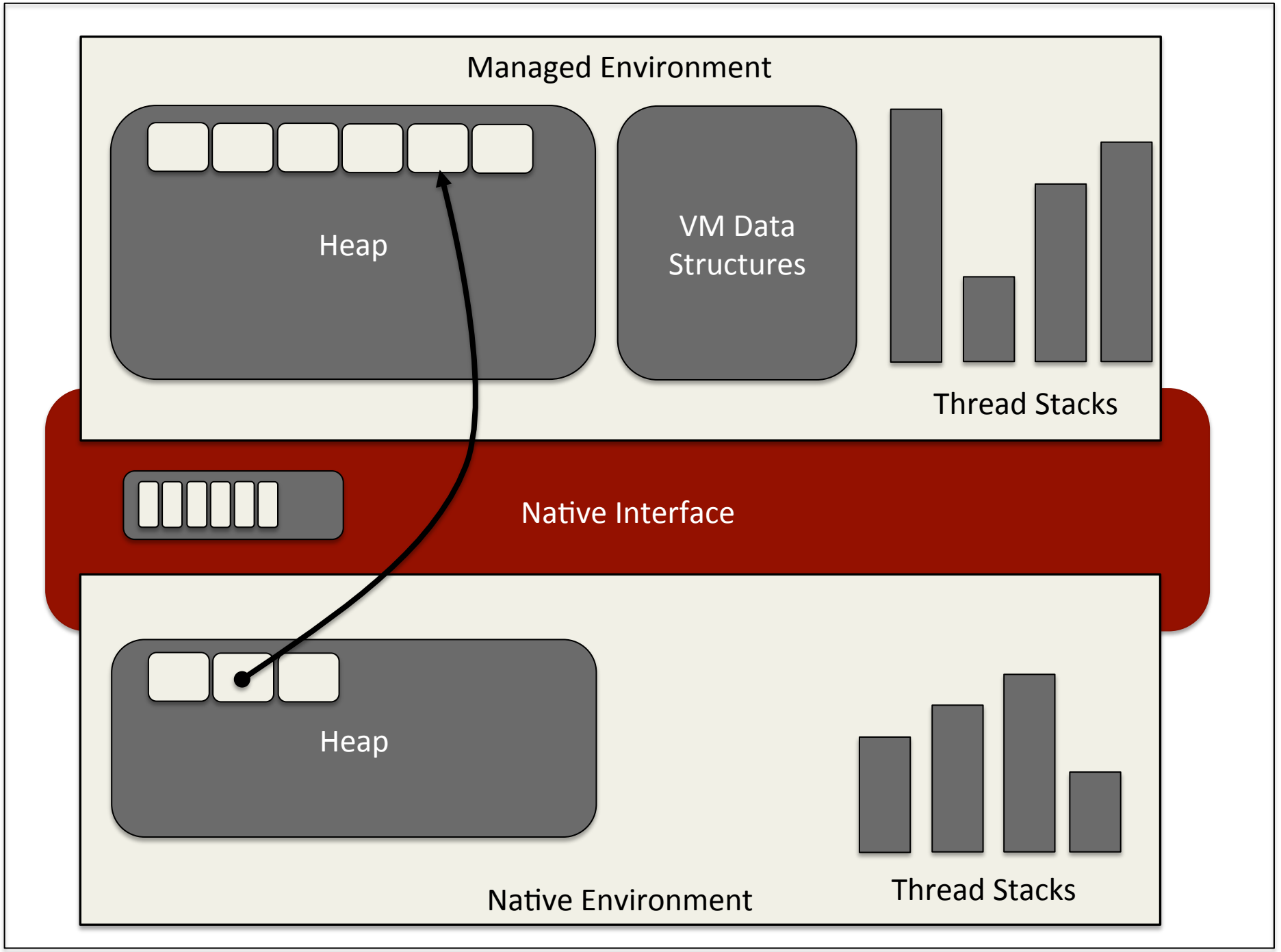
# Pinned Objects

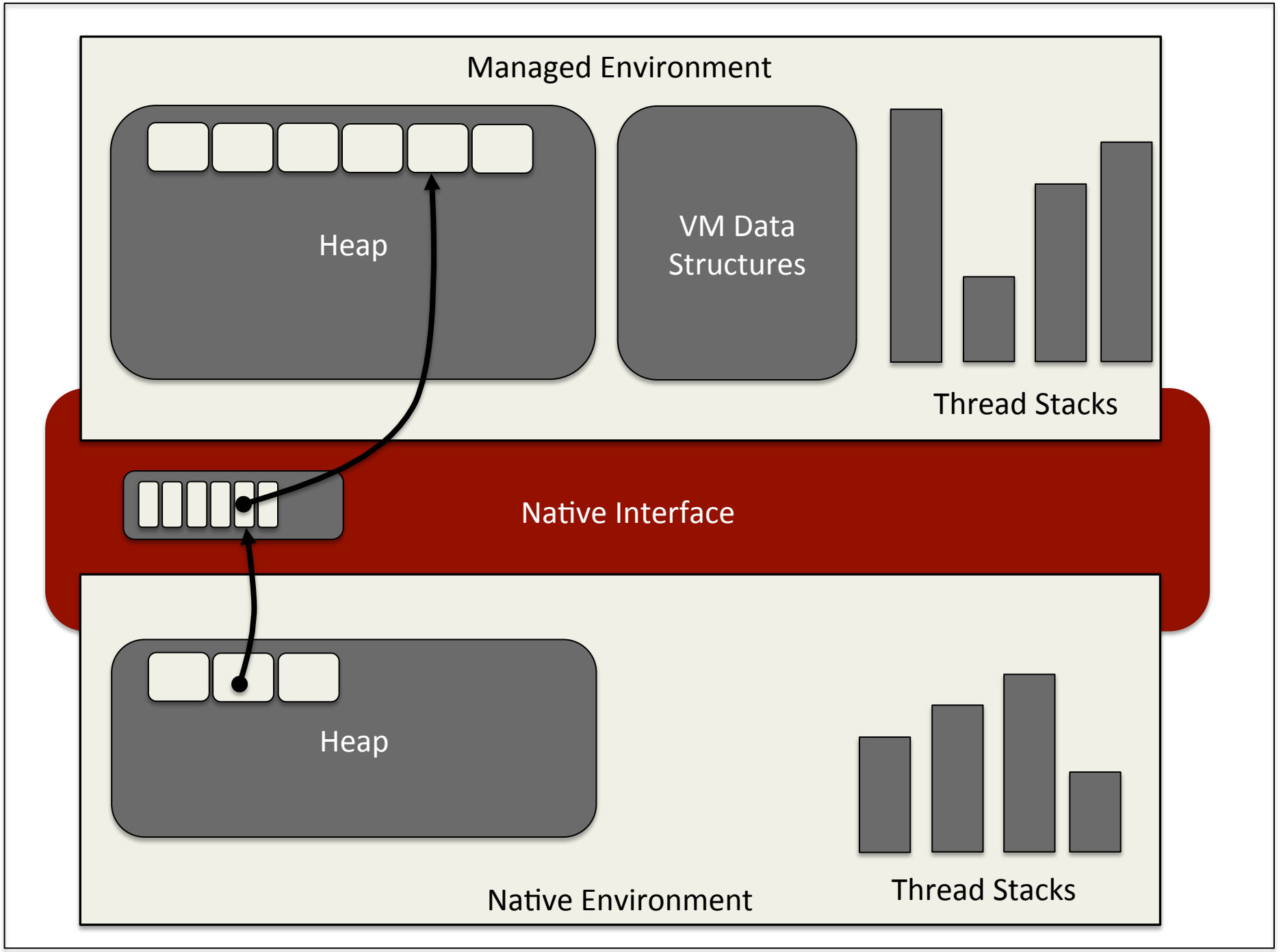
- Objects referred to by native code can be pinned
  - Guarantee that the GC will not move them
  - Allows native code to access safely
- Pinning limits our memory management options
  - Reference counting and Mark/Sweep are OK
  - Bump pointer allocation doesn't work
  - Rules out copying collectors
    - Or at least makes them harder
- We may not want to dismiss pinning completely

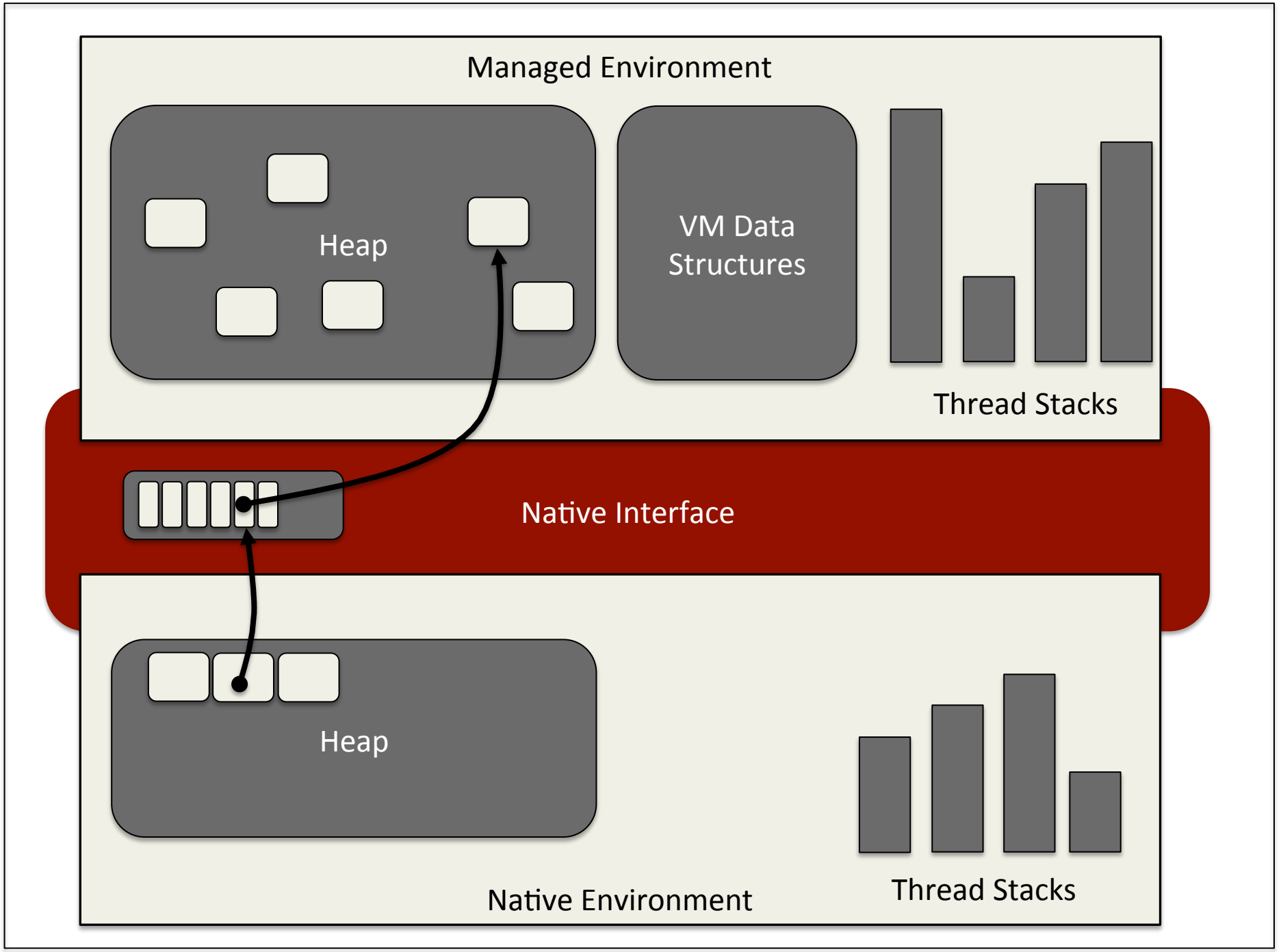
# Object Layout

- It's not ideal for native code to access the heap
  - The VM controls the layout of objects
- Object layout is a VM implementation detail
  - The VM is free to change the layout
  - May even change object layout on the fly
- We need to indirect access to pinned objects
  - Supported by the `JNIEnv` object


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
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