As well as objects, static data belonging to classes must be tracked. Static data is held in a location called the *Java Table Of Contents* (JTOC). The JTOC is organized in two directions: positive offsets in the JTOC hold the contents of static fields that contain references and reference literals. A literal value in Java is something like a string literal, something that the bytecode can directly reference but that doesn't have a field location. Negative addresses in the JTOC are responsible for holding primitive values. Splitting the values in this way allows the garbage collector to easily consider which references from static fields should prevent an object from being considered garbage.

Runtime Memory Layout

The boot image writer is responsible for laying out the memory of the virtual machine when it first boots. All of the objects needed in the boot image are there because they are referenced by code that will start the Java application. Figure 10-4 gives an overview of the regions of memory that are in use when Jikes RVM executes.

A number of key items can be seen in Figure 10-4:

Boot image runner
: The boot image runner and its stack comprise the loader responsible for loading the boot image.

Native libraries
: Memory is required for native libraries that are used by the class library. This is described further in the later section "Native Interface."

MMTk spaces
: These are the different garbage-collected heaps in use by MMTk to support the running application.

Root map
: Information for the garbage collector on fields that may be reachable from the boot image. More information is given later in "Garbage Collection."

Code image
: The executable code for static and virtual methods that are reachable directly from the JTOC or from TIBs, respectively. Code is written to a separate region of the boot image to provide support for memory protection.

Data image
: The data image is created by first writing the boot record and the JTOC and then performing a traversal of the objects reachable from the JTOC. The objects are traversed in the bootstrap JVM using Java's reflection API. The order in which objects are traversed can impact locality, and thereby performance, so the traversal mechanism is configurable (Rogers et al. 2008).