CONFIGURABLE JVM THREADING

A DISSERTATION SUBMITTED TO THE UNIVERSITY OF MANCHESTER

FOR THE DEGREE OF MASTER OF SCIENCE

IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2007

By

Rahul Mehta

School of Computer Science

Contents

Abstract **6**

Declaration **7**

Copyright **8**

Acknowledgements **9**

1. Introduction **10**

1.1 Motivation and Objective ……. 10

1.2 Organization of Thesis. …… 12

2. Infrastructure **13**

2.1 JikesRVM ……... 13

2.1.1 JikesRVM in Java ……... 13

2.2 How Jikes RVM Starts ……... 14

2.2.1 Bootstrapping – Boot image Creation and loading ……... 14

2.3 Magic Classes …… 17

2.4 JikesRVM's Major Sub-systems ……... 17

2.4.1 Object Model ……. 18

2.4.1.1 Object Header …… 20

2.4.1.2 JTOC ……. 21

2.4.2 JikesRVM's Core Runtime Subsystems ……... 22

2.4.2.1 Exception Management ……. 22

2.4.2.2 Thread Management ……. 23

2.4.2.3 I/O Management and Reflection ……... 24

2.4.3 Memory management ……... 24

2.4.3.1 How GC works in JikesRVM ……... 25

2.4.4 Compiler subsystem ……. 25

2.5 Summary ……... 26

3. Current Thread Model 27

3.1 Overview of Current M:N Green Threading Model ……. 27

3.2 Thread Scheduling in JikesRVM's M:N Model …….. 30

3.2.1 Thread Switching …….. 30

3.2.2 JikesRVM Load Balancing Mechanism ……... 32

3.2.3 Scheduling among mutator and collector threads …… 33

3.2.3.1 JikesRVM's parallel garbage collection …..…….. 34

3.3 Thread states in JikesRVM …….. 35

3.4 Thread Queues ……. 37

3.4.1 Processor-local Queues …… 37

3.4.2 Global Queues …….....……. 38

3.5 Synchronization ……..……... 40

3.5.1 Mutual exclusion (locking/unlocking) ……. 40

3.5.1.1 Processor-lock …….. 40

3.5.1.2 Thin and thick lock …….. 42

3.5.2 Cooperation (wait/notify) ……..…… 43

3.6 Drawbacks of M:N green thread model ……..…….. 44

3.7 Summary ……..……...…….. 45

4. Re-factoring and Design Pattern 47

4.1 Introduction ……. 47

4.2 Our approach for re-factorization …… 48

4.2.1 Factory Method – Design Pattern …… 48

4.2.2 Why we used Factory Design Pattern ……. 48

4.2.3 Factory Design Patter in Our Framework ……... 49

4.3 Our Design Attempt …… 52

4.4 Re-factoring at code-level (Code-Snippet) ……. 53

4.5 Flexible Threading model – User's choice ……. 55

4.5.1 Properties defined in VM_Properties ……..…. 55

4.5.2 Prefixes defined in VM_CommandLineArgs ……. 56

4.5.3 Configure build.xml …… 57
4.6 Summary ……..……... 57

5 Native Thread Model 58
5.1 Introduction ……..……... 58
5.2 Native Thread Model ……..……... 58
5.3 Native Pthread Scheduling ……..……... 60
5.4 Binding Java thread to Kernel Thread - CPU Affinity ……..…….. 61
5.5 Implementation of yield method …… 61
5.6 Thread Synchronization – Monitor …… 63
5.6.1 Mutex …… 63
5.6.2 Condition Variable …… 64
5.6.2.1 Waiting on a condition variable …… 64
5.6.2.2 Signaling condition variable ……... 65
5.6.3 Mutual Exclusion Implementation …… 66
5.6.4 Cooperation –Implementation …… 68
5.7 Thread Interruption ……..… 70
5.8 Adding System calls in JikesRVM …… 72
5.9 Comparison between Native thread and Green thread model …… 73
5.10 Summary …… 74

6. Conclusions and Future work 76
6.1 Conclusions …… 76
6.2 Future Work …… 77

7. Appendices 78

8. References 80

List of Figures

Figure 2.1 Boot Image Creation and Loading ……. 16
Figure 2.2 JikesRVM – Subsystems and Core Services ……..……. 18
Figure 2.3 Layout of an array object and a scalar object in JikesRVM ……... 20
Figure 2.4 JikesRVM Table of Contents *(JTOC)* ……..…… 22
Figure 3.1 JikesRVM's M:N green threading Model ……..……. 29
Figure 3.2 JikesRVM's M:N green thread Scheduling ……..…….. 30
Figure 3.3 Transition b/w mutator and collector threads and phases of parallel GC ……..... 34
Figure 3.4 Thread States in JikesRVM …….…….. 37
Figure 4.1 Division of the JikesRVM's scheduler API …….. 47
Figure 4.2 UML Diagram of Factorization of Scheduler API …..…….. 49
Figure 4.3 Structure of Factory Pattern in SchedulerAPI ……...…… 50
Figure 4.4 Program Flow ……...…….. 51
Figure 5.1 Native Thread Model in JikesRVM …….…….. 59
Figure 5.2 Thread yield in native model …….. 62
Figure 5.3 Mutual Exclusion ……..……. 67
Figure 5.4 Implementing Wait/notify semantics using Pthread functions ….. 69
Figure 5.5 Thread Cancellation ……... 71

Abstract

Multi-threading is one of the most important features of modern programming languages: running the multiple threads at a time on Symmetric Multiprocessing (SMP) environment gives the high performance in terms of execution speed and parallelism. In addition, multithreaded programs can improve the throughput on SMP environment by utilizing the available processors more effectively. This dissertation presents a means of implementing Java threads using underlying native mechanism.

This dissertation first presents the key details of JikesRVM. Then, it describes the existing multi-threaded green model in JikesRVM followed by a number of issues in this threading system. This dissertation explains how we can exploit the multi-processor environment by direct binding of Java threads to operating system's threads.

This dissertation illustrates the clean refactorization of existing thread model into two separate thread models. With this refactorization, users can choose the more appropriate threading models according to the nature of their applications. This dissertation concludes by explaining design and implementation of major components of native thread model in JikesRVM using POSIX pthread library.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

It gives me immense pleasure to express my profound sense of gratitude to my supervisor **Prof. Ian Watson** who provided the opportunity as well as all the facilities to carry out this project. His admirable forbearance, valuable guidance and motivation helped me complete this thesis work.

I am deeply indebted to my co-supervisor **Dr. Ian Rogers** whose help, valuable suggestions and stimulating support made it feasible to carry the present work to conclusion. His thorough understanding of this work, insightful discussions and explanations were really impressive and encouraging. Furthermore, his informative feedback led to overall improvement of this work.

I also express my thanks to my colleagues in APT group, Mr. Mikel Baer and Mr. Mohammad Ansari for their interesting ideas and discussions on JikesRVM's threading model. Furthermore, I am grateful to Mr. Andrew Dinn for his valuable hints and prompt help on JikesRVM's threading model, through long emails. I am also thankful to Dr. Chris Kirkham and Dr. Mikel Lujan for their comments and valuable suggestions during my presentation.

I am in dearth of proper words to express my feelings towards my parents who apart from providing me the best available education, have always encouraged me in all my endeavors. I also appreciate the affection and moral support given by my elder brothers Sanjiv and Saurabh who have always pumped a new zeal and spirit in me at all the challenging times. I also express my thanks to my friend Mr. Siddarth who, with all his courtesy and alacrity helped me in this endeavor. Finally, I would like to express my gratitude to all those whose help and support led to the completion of this dissertation.

(RAHUL MEHTA)

CHAPTER 1

Introduction

In this thesis, we present the existing green threading m:n model of the JikesRVM, implementation of a clean refactorization of existing thread model into green thread and native thread models, an attempt to give direction for native thread model using POSIX thread library. We used the last version of JikesRVM (JikesRVM 2.9.0) in order to do our research and development in threading model. This work was carried out in the Advanced Processor Technology (APT) group, Computer Science Department at the University of Manchester.

1.1 Motivation and Objective

The main aim to design native threading model in Jikes **R**esearch **V**irtual **M**achine is to provide true concurrency and parallelism in multiprocessor environment and is to gain performance from multiprocessing hardware (parallelism). A further motive is to give threading control to operating system and to improve the modularity.

Currently, JikesRVM supports an **m:n** green thread model where scheduling is done in user space; m Java threads map on to n virtual processors (which are further mapped on operating system's threads). We experienced a number of performance issues in the existing model. For instance, the current model is using a 'jacket' routine around input/output blocking operations; this jacket routine intercepts the blocking I/O operations and replaces with non-blocking (asynchronous I/O) operations. However, JikesRVM is not always able to intercept blocking I/O operations in native code and cannot set 'yield points'* in the native code. This could be the cause of deadlock of

**yield point is described in section 3.1, 3.3.3 and 3.6.*

VM_Processor and they would not be able to schedule the other threads. Direct mapping of Java threads to operating system threads (pthreads) and giving control to operating system to schedule the threads by using a regular native model can overcome such issues. Building on the operating system's native pre-emptive scheduling provides the tight coupling with kernel so that applications can use the system resources efficiently and optimally which would increase application throughput and responsiveness. Furthermore, a number of empirical studies indicate that pthread ends up becoming faster [1].

We also aim to provide the choice of both green thread and native thread to the users so that they can exploit the flexible threading system of JikesRVM by having this choice according to nature of their application.

For example, they can use native model for normal cases where intensive computation is required and can use m:n green thread model (existing JikesRVM threading model) when they are going to have very large number of threads such as for server applications where numerous client requests need to be processed.

The objectives of this work are to examine the behaviour of the existing threading model and to enhance its capabilities by a clean refactorization and introducing a new native thread model.

Organization of Thesis

This thesis is organized into six chapters. In each chapter some specific details of the JikesRVM, m:n threading model and native model have been discussed. It will also describe our analysis for future pthread model by native support. Following is a succinct description of each chapter:

Chapter 2 discusses the key details of the JikesRVM, its architecture, bootstrapping mechanism and major subsystems such as runtime service, memory management, object model, compiler.

Chapter 3 discusses the current threading model, thread queues, synchronization, locking/unlocking, thread cooperation, thread states in JikesRVM, load balancing, garbage collection and issues in existing threading model.

Chapter 4 discusses the factory design pattern, refactorization of existing threading model, clean approach for keeping both models and separate command line arguments.

Chapter 5 discusses an implementation of native threads using the POSIX thread library and its benefits over the m:n, some major components of pthread such as mutex, conditional variables and cancellation point and indicates how the native model will look like in the real system

Chapter 6 Conclusions and future work

CHAPTER 2

Infrastructure

This dissertation is based on the threading model of **J**ikes**RVM** and describes the current threading model (m:n green thread model) and future one-to-one native model using Linux POSIX library. This chapter explains the fundamentals of JikesRVM and its main subsystems briefly.

2.1 JikesRVM

JikesRVM is a **R**esearch **V**irtual **M**achine initially developed by IBM. It is now an open-source project. The salient features of JikesRVM are that it is implemented in the Java programming language and is self-hosted, which means that its Java code runs on itself without requiring another virtual machine.

Fundamentally, JikesRVM comprises two bytecode-to-native compilers: baseline and optimizing; both having their own advantages. It also implements modern garbage collectors and includes an adaptive compilation infrastructure. JikesRVM does not use interpreter, in other words JikesRVM is completely based on the Just-In-Time compilation with the choice between two different compilers. This section describes important details of Jikesrvm that are useful to understand this dissertation and JikesRVM.

2.1 JikesRVM in Java

Almost all of the components of JikesRVM are written in Java, this means that the different components of JikesRVM have tight coupling and that these components support each other more strongly as compared to other available virtual machines: for

example, the compilers compile the code of the garbage collectors and the garbage collectors reclaim memory of unused objects allocated by the compilers. However, not all the components of Jikes RVM are written in Java, some of these are written in the C programming language. The Boot Image Runner, a tiny C program is responsible for starting Jikes RVM. Magic mechanism provides low-levels system code that is necessary to implement Jikes virtual machine. We will adhere with this and will explain briefly in the next section.

Another interesting aspect of JikesRVM is its self-hosting which has both, the upside and downside. One of the benefits is that all of the components of virtual machine get advantages from each other's improvements; for instance, the performance of the allocation sequence of the garbage collector benefits from the compiler's ability to inline and optimize it. On the other hand, the major challenge is that different components of JikesRVM are under more stress because they have to service not only the application but also the virtual machine itself.

2.2 How Jikes RVM Starts

2.2.1 Bootstrapping – Boot image Creation and loading

JikesRVM starts from the *boot image*, a set of files containing the compiled code and Java objects that are needed to start execution of JikesRVM. Before the JikesRVM loads into memory, a set of essential core services - a class loader, a JIT compiler, garbage collector and an object allocator - are required for operation. Generally, these initial core services for a JVM are written in native code, but JikesRVM is written in Java and it has no underlying run-time routines. Thus all the essential core services are assembled into an executable boot image. This *boot image,* also called a snapshot of JikesRVM [2], is written into a file and this file must be loaded into memory and then executed to run the JikesRVM. For creating the Boot Image, the host JVM would need to execute a Java program, called the *boot image writer*.

The boot image writer executes the initialization code for various Jikes RVM classes; also known as primordial classes, to pre-allocate Java objects. The boot-image writer constructs a mock-up of running JikesRVM and then put into a boot image. This builds a boot image for the execution of JikesRVM. The JVM, which runs the boot-image writer program, is called the *bootstrap* JVM.

When the source JVM (e.g. Sun Hotspot) runs the boot-image writer program, it instantiates the Java objects of the Jikes virtual machine in objects of source JVM. Then it uses Java's built-in reflection facility to translate these mock-up objects from the object model of bootstrap JVM to JikesVM's object model and storing them in a boot-image array. This is the transformation process of creating the boot-image.

A short C program called the boot-image runner starts JikesRVM. The Boot-runner loads the boot image into memory and jumps to the address of the instructions responsible for starting the JVM, which were compiled from Java at build time and then branches to boot() method. The *VM.boot* method is the first Java method executed in the boot image. Once JikesRVM started running then it would not need the bootstrap JVM anymore. The whole process of boot image creation and loading is indicated in the figure 2.1.

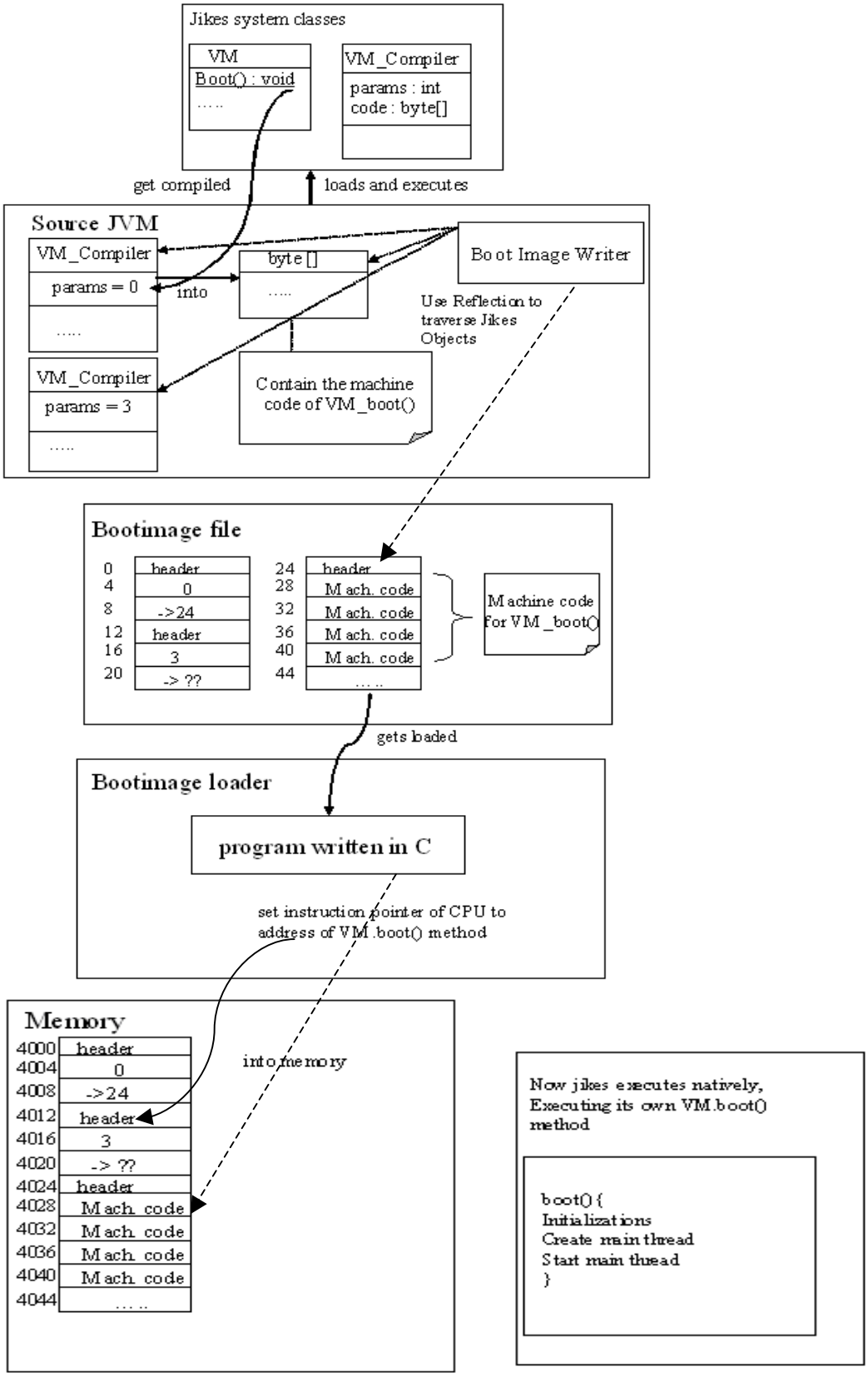


Figure 2.1 Boot Image Creation and Loading [3]

2.3 Magic Classes

As JikesRVM is written in Java, low level functionality, which is not possible to write in pure Java, is implemented through the special MAGIC mechanism. *Magic mechanism* allows JikesRVM to implement certain functionalities such as invoking underlying operating system's (OS) services, accessing machine registers, read/write value from/to memory, make system calls etc. In addition, input/output requires access to OS services and pointer manipulation that are unknown to Java. Thus magic helps in achieving these low-level services.

Magic mechanism looks like normal Java object with methods. They are identified by the compiler and translated into low-level operations. Magic class contains empty methods. When the compiler identifies these methods; it inserts the corresponding machine code in line.

2.4 JikesRVM's Major Sub-systems

Figure 2.2 indicates the main sub-systems and the core services of the JikesRVM and also the process through which they communicate with the underlying operating system.

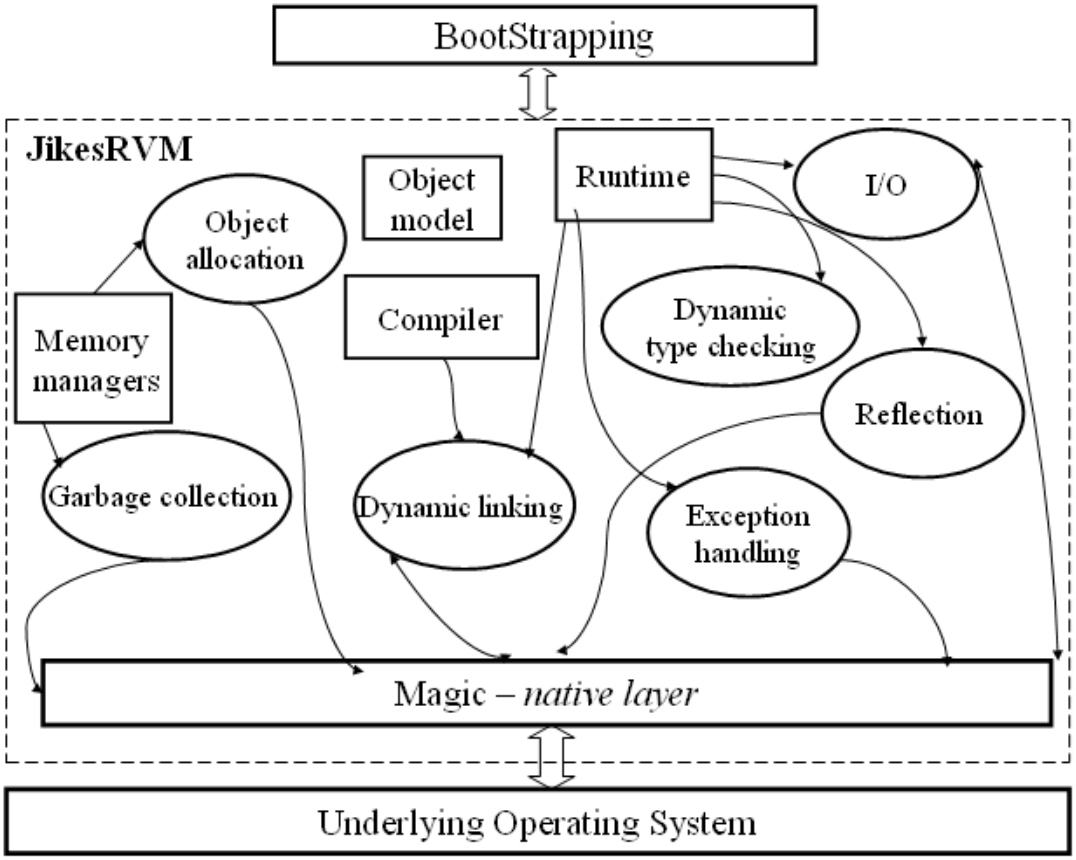


Figure 2.2 JikesRVM – Subsystems and Core Services

JikesRVM can be divided into three major components: core runtime services, compilers and memory managers/garbage collection. Succinct details about each component and their functions have been provided in the following sections followed by the Object Model as it underlies the other systems.

2.4.1 Object Model

JikesRVM's object model specifies how to represent objects in memory. The VM_ObjectModel class defines JikesRVM's object in source code. Generally, Java virtual machines perform operations on certain types of data and these data types are defined by the Java library. In the Java programming language, a data type can be divided

into two categories: primitive and reference data types. Primitive type variables and reference type variables hold primitive values (int, double, etc.) and reference values respectively. Reference values refer to objects but they are not objects themselves. Objects are either *arrays* having elements or *scalars* (class instance) having fields. In addition, one other reference value is the *null* value, which specifies that the reference variable does not refer to any object.

JikesRVM's object model is based and developed on the following requirements:

1. Allowing fast access to instance field and array elements
2. Virtual dispatch method should be fast
3. Performing Null pointer checks by the hardware
4. Less frequent operations should not be slow
5. Supporting fast access to static objects and methods
6. Reducing overhead related to object storage (e.g. object header size) in order to minimize heap space overhead [2].

If the reference to an object is stored in a register then the fields of the object can be accessed at a fixed displacement in a single RISC instruction. In contrast, for the array access, the reference points to the first element of the array and the remaining elements are positioned in the ascending order. The number of elements is kept before its first element as indicated in the figure 2.3. Thus the array elements can be accessed through base and scaled index addressing. In JikesRVM, arrays grow up from the object reference; while scalar objects grow down from the object reference (see Figure 2.3).

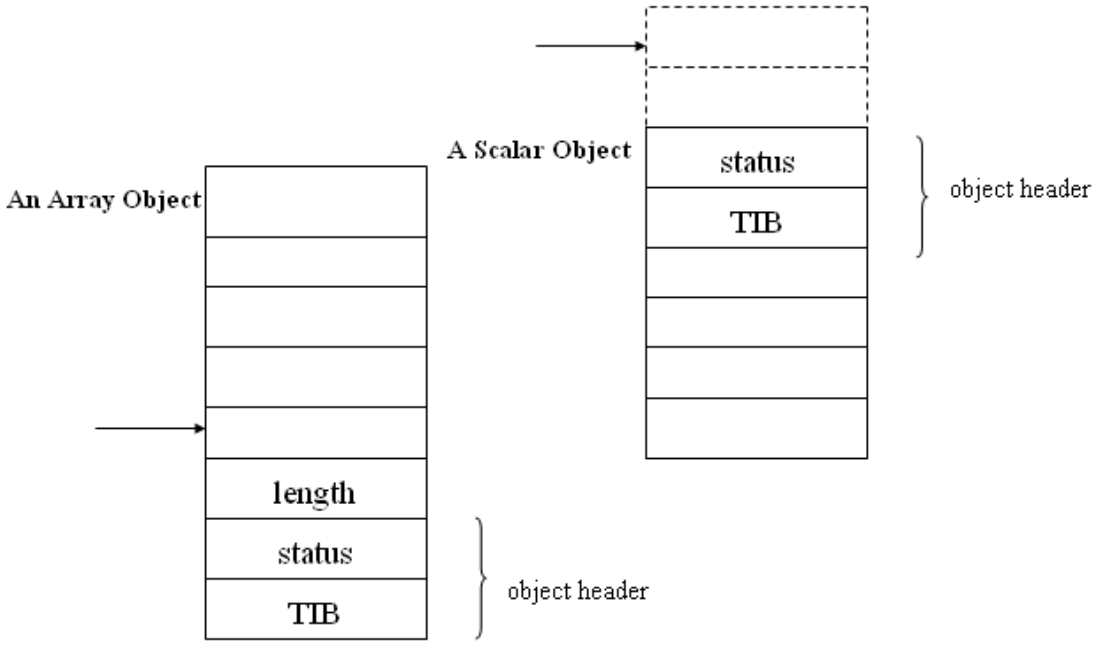


Figure 2.3 Layout of an array object and a scalar object in JikesRVM [2]

2.4.1.1 Object Header

In standard configurations, the JikesRVM has a two-word object header attached with each object, which supports virtual method dispatch, dynamic type checking, memory management, synchronization, and hashing. It is positioned twelve bytes (3-word) below the value of a reference to the object because this leaves space for the length field, if object is an array.

One word of the object header is *status* and the other is the *TIB*. The *status word* is further divided into three bit fields. The first bit field is used for locking (for associating a lock state); it contains a pointer to a lock object or direct presentation of the lock, we will see this in further chapters. The second bit field holds the default identity hash value of objects. The third bit field is used by the memory management system for associating garbage collection information. This can include a combination of reference count, forwarding pointer, and other GC information.

The other word of an object header is a reference to the *Type Information Block* (TIB) for the object's class. A TIB is an array of Java object references and serves as Jikes's virtual method table. Its first element specifies the object's class. The remaining components are compiled method bodies (executable code) for the virtual methods of the class [2].
The object header implementation is defined by three Java classes in the JikesRVM: VM_JavaHeader, which supports locking, TIB access, and hash code; garbage collection information is supported by VM_AllocatorHeader; and there is one more Java program that is called VM_MiscHeader, which supports adding additional fields to all objects.

2.4.1.2 JTOC – methods and fields

JikesRVM Table of Contents (JTOC) is declared as an array of *ints* in JikesRVM but contains values of all types because JikesRVM uses a descriptor array (co-indexed with the JTOC) to identify the type of each entry. The reference to this array (*JTOC array*) is maintained in a machine register is called JTOC register. JTOC stores all the static fields and the references to all the static method bodies. All the JikesRVM's global data structures are also accessible through pointers hold in JTOC. Moreover, numeric constants and literals also reside in JTOC. The *JTOC* also contains pointers to TIBs in order to enable fast-dynamic type checking. In the *JTOC,* reference and non-reference values are indexed positively and negatively respectively with respect to the middle of the table so that garbage collector can differentiate them. In addition, the JTOC register always points to the middle element. Figure 2.4 illustrates the JTOC in memory.

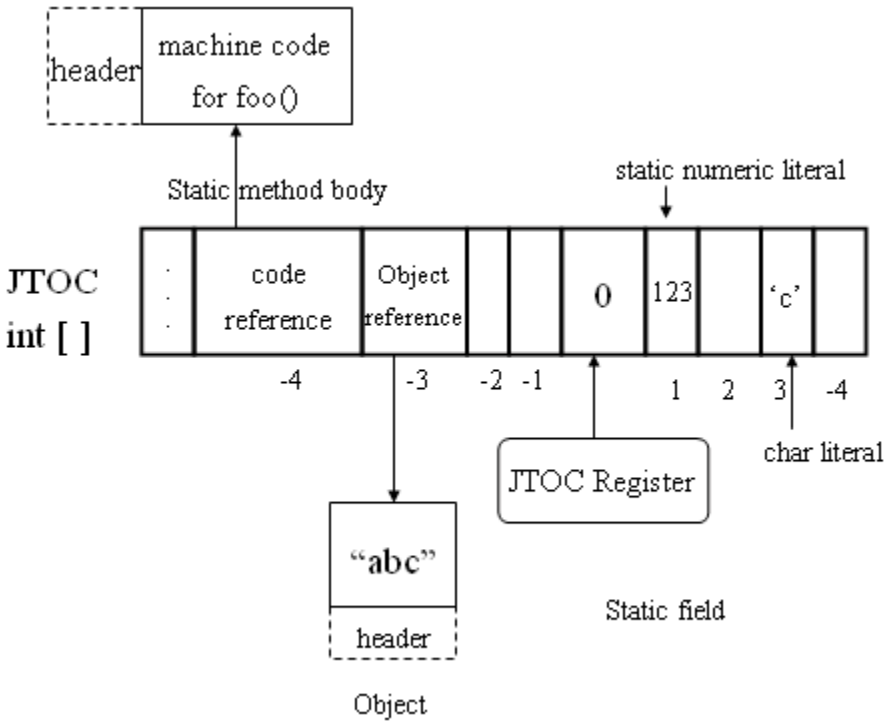


Figure 2.4 JikesRVM Table of Contents *(JTOC)* [4]

2.4.2 JikesRVM's Core Runtime Subsystems

One of the unique components of the JikesRVM is its runtime service; it includes exception handling, thread scheduling, dynamic class loading, dynamic type checking, reflection, Input/Output, interface invocation etc. These all are implemented in Java with the use of MAGIC classes. However, these are conventionally implemented using native code (typically in C/C++, assembler code) in other JVMs. These are explained briefly in the section below:

2.4.2.1 Exception Management

In JikesRVM, exceptions can be explicitly generated either by software (example athrow) or by hardware (both synchronous and asynchronous exceptions). The cause of hardware exception could be if a null-pointer is dereferenced, number divided by zero, and stack-overflow. JikesRVM handles the software exceptions internally and the hardware

exceptions are handled through the native support. A small C interrupt handler catches hardware exceptions particularly. This interrupt handler calls a Java method and this method builds the appropriate exception and passes it to the delieverException method. Then VM_Runtime.delieverException has to perform two tasks; first, it must save in the exception object information that would allow a stack trace to be printed, if needed. It does this by the walking up the stack and recording the compiled method identifiers and next instructor pointer for each stack frame. The second task involves, transferring control to appropriate catch block. This also involves walking the stack [5]. When a catch block is found, update in VM_Register is made so catch block can find compiler-specific exception object and then modify the VM_Register to resume execution at first instruction in catch block. If no catch block is encountered, JikesRVM kills the thread.
In order to walk the thread stack, the exception handler uses the memory access facilities. It also exploits magic services for restoring the register state and transferring control whenever an appropriate catch block is found.

2.4.2.2 Thread Management

Currently, JikesRVM is using m:n green threading model, which means that all threads are scheduled by the virtual-machine itself in the user space. JikesRVM multiplexes m Java threads to the n operating system threads which ensure that virtual machine is running at least n threads simultaneously. JikesRVM does not map Java threads to operating system threads directly; it multiplexes Java threads on *virtual processors* that further implemented as OS threads. This model is also called user-level model, because threads are scheduled in user-space; no calls into operating system are required to handle any of the thread detail.

The fundamental goals of mapping threads on virtual processor are to support rapid thread switching and fast transition between Java threads (mutators) and garbage collection threads. In this model, underlying operating system does not know about the threads scheduling at all; it is up to the JikesRVM to handle this. We will explain about

the current m:n green threading model and will discuss the future native thread model in next chapters.

2.4.2.3 I/O Management and Reflection

I/O: The input/output operations need operating system's support. JikesRVM uses system calls mechanism to communicate with the underlying operating system. In order to read a block from a file or write something in it, an operating system routine is called with a Java array for copying its result.

Reflection: With the use of reflection mechanism JikesRVM allows run-time access to fields and invocation of methods. For passing the parameters to and from the caller, JikesRVM creates additional artificial frames on thread stack. When the method returns, the stack frame disposed of and the result returns to the reflective call.

2.4.3 Memory Management

Memory management is itself a huge topic but we will describe an overview of it along with how garbage collector (GC) works in JikesRVM. JikesRVM supports four major types of manager systems, these are copying, noncopying, generational copying and generational noncopying. Each memory manager consists of a *concurrent object allocator* and a stop-the-world, parallel, type accurate *garbage collector*. In addition, JikesRVM can be configured to use different allocation managers provided by the MMTk. These allocation managers automatically divide the available memory as they see fit.

More specifically, in MMTk, there is a main class that is called *Plan*, which is used to interface to memory manager and contains different implementation for each type of memory manager. Most of the plans are inherited from class *StopTheWorldGC* [6] that ensures that all the active Java threads are suspended before reclamation is started, in the next section we will see how it happens in JikesRVM.

2.4.3.1 How GC works in JikesRVM

A collector thread is associated with each virtual processor in JikesRVM. JikesRVM operates in one of two modes either normal Java threads are running, and the GC threads are idle or GC threads are running and the Java threads are idle. In JikesRVM, mutator threads request for the collection thread when the allocator cannot satisfy the space request by them.

When the mutator threads run, collector threads wait in the waiting queue. Once the mutators trigger the collector threads, they start reclaiming memory on their virtual processors. Furthermore, GC begins when the executing thread on each processor reaches the safe point. In the next chapter, we will describe how mutator and collector threads schedule on virtual processors interchangeably and about the parallel GC and its phases.

2.4.4 Compiler Subsystem

The core function of JikesRVM compiler subsystem is to generate the machine code from the Java bytecode. The compiler subsystem comprises two types of compilers: the baseline compiler and the optimising compiler. They can be differentiated from each other in terms of compilation time and the quality of code that they generate.

The mechanism employed by the baseline compiler is very simple and straightforward and it does the byte-by-byte translation of Java byte codes into machine code. It directly translates bytecode into machine code, thus the baseline compiler generates machine code very quickly. However the downside of base compiler is the poor performance of the machine code generated by it, because the base compiler does not implement any optimisation. The other compiler is the optimizing compiler and it produces high performance machine code because of its complex optimization. However, optimising compiler takes longer compiling time than the baseline compiler, which is a cause of higher cost compilation.

Another important task of the JikesRVM's compilers is to maintain tables that support both exception handling and that allow memory managers to find out the object references on the thread stack. Reference maps help in locating the object reference for each safe point in the method body. When a call to garbage collection is made, each of the methods represented on the thread stack will be at garbage collection safe point. Thus for any given *safe point* within the method body [2], the compiler that created the method body must be able to illustrate where the live references exist.

2.5 Summary

In summary, we have discussed the JikesRVM and it main subsystems and the core services briefly, which are important to understand the JikesRVM. Now in next few chapters, we will discuss the current threading model and performance issues in it and how we can resolve them by introducing the native thread model.

CHAPTER 3

Current Thread Model

3.1 Overview of Current M:N Green Threading Model

As in the preamble of JikesRVM in the previous chapter, we discussed about the current thread implementation of JikesRVM in brief, which is m:n green thread model. We call *m:n* model because several Java threads *(M)* schedule and implement on limited underlying operating system's thread *(N)*. Thus it ensures that JikesRVM is running an utmost of n threads simultaneously. We also call it a *green model* because the thread scheduling in JikesRVM is done in the user-space. Therefore the responsibility of thread scheduling is on the virtual machine.

JikesRVM introduced the concept of a virtual processor in order to achieve quicker thread switching among the threads and fast transition between mutators and the collector threads. The Virtual processors *(VM_Processor.java)* are the Java objects implemented as operating system's pthread. As we mentioned previously, all Java threads running on the top of JikesRVM map to the virtual processors, which are further, multiplexed on the underlying OS threads.

In the current model, the underlying operating system does not know anything about the threads scheduling at all; it is up to the JikesVM to manage them; JikesVM holds all the details of the threading API in scheduler package. The benefit of the current model is that it is portable because this model does not depend on the specific operating system to provide any thread specific details.

JikesRVM thread scheduling is based on *quasi-preemptive* which means neither fully preemptive nor 'run until blocked' since it is driven by the JikesRVM compiler. However, in JikesRVM threads can be preempted but only at predefined yield-point. The

compiler sets up a special code for yield point within each compiled method body that causes currently running thread to request its virtual processor if it can continue running or not. If the *VP* grants the execution, the thread continues until a new yield point is reached, otherwise it suspends itself in favor of other threads so that the virtual processor can execute another virtual thread. In addition, at the yield point, JikesRVM compiler provides information about object references on the thread's stack, which could be used by GC to reclaim memory. So here the noticeable point is the process through which these three components work together (compiler, garbage collection and threading model).

All Java threads such as application threads, system threads (garbage collector threads, idle threads) etc. derive from *virtual machine thread,* VM_Thread. JikesRVM does not directly map them to operating system threads. Instead, JikesRVM creates a *virtual processor object (VM_Processor)* for each pthread in use and Java threads map on them. Each virtual processor is bound to a pthread (normally one pthread for each physical CPU). If the users want to exploit the multi-processor architecture with JikesRVM, they need to specify the number of virtual processors as a command line argument; this number must not be more than the number of physical CPUs. For example –X:processors=5, in this case JikesRVM will create five virtual processors and all the *vm* threads will be scheduled on them, otherwise by default JikesRVM creates only one virtual processor. A thread stops execution when it calls the *yield* method voluntarily, or interrupted and blocked by a lock.

A couple of queues are associated with each virtual processor, also called the processor-local queues, and there are other queues called the global queues (they are common to all virtual processors) so all virtual processors need to synchronize access to them. These queues hold threads such as runnable threads, idle threads, blocked threads, waiting threads and transferring threads etc.
Threads in different states reside in separate queues such as an idleQueue only holds an idle thread that will execute whenever virtual processors have nothing to do, a readyQueue holds only ready to execute threads and so on.

JikesRVM's scheduler follows FIFO and RR policy that means it always selects the first thread from the readyQueue and assigns to the virtual processor for execution and if it suspends then adds in to the rear of the queue. In addition, JikesRVM has a timer interrupt event (timer-tick) that sets timeSliceExpired. If the time slice expired of the current thread, it suspends execution in favor of other threads and enqueue in the run queue. If the runQueue associated with its VP is empty then suspended thread schedules again on the same VP otherwise this thread can move to a random virtual processor if its current processor has other runnable work.

There is no time slicing mechanism for JikesRVM's scheduler as Java Runtime Environment does not provide this. Currently, JikesRVM has no priority mechanism, which means all the threads run at the same priority. The following figures 3.1 and 3.2 indicate the high-level and detailed thread scheduling respectively, in JikesRVM's m:n threading model.

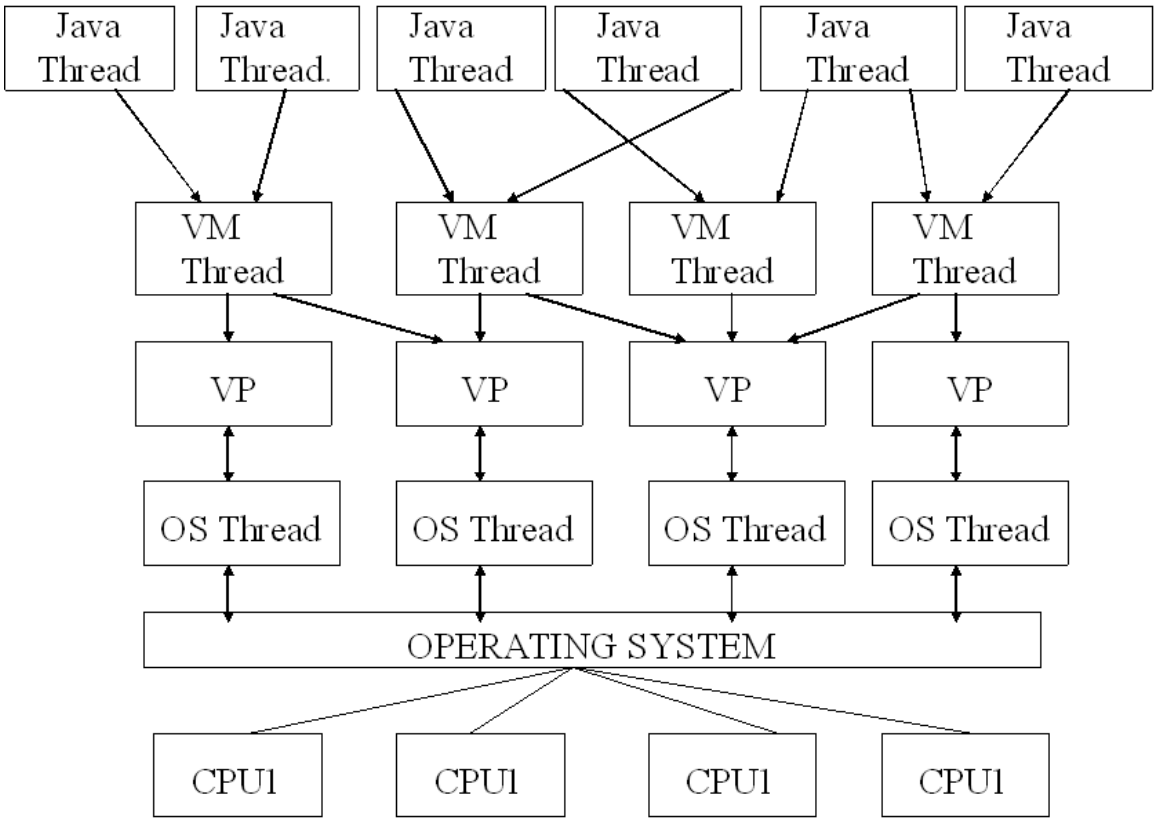


Figure 3.1 JikesRVM's M:N threading Model

3.2 Thread Scheduling in JikesRVM's M:N model

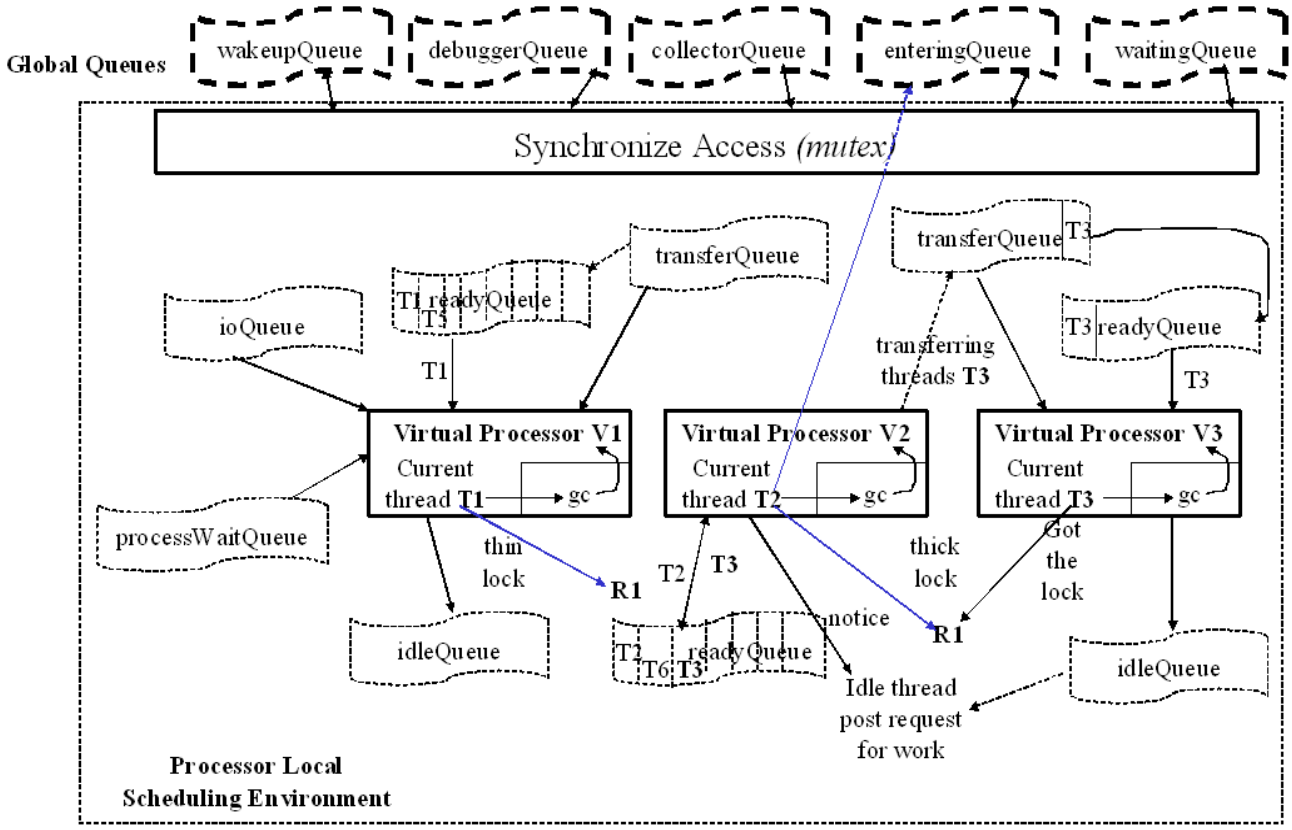


Figure 3.2 JikesRVM's M:N green thread Scheduling

3.2.1 Thread Switching

Switching from one thread to another in the current threading model is a complex operation and is normally managed by the four main methods: *yield()* and *morph()* (of VM_Thread) and *dispatch()* (of VM_Processor) and *threadSwitch()* of (VM_Magic).

Let's take an example of two threads A and B. *Yield()* suspends execution of the current thread (A), in favor of some other thread (B), in other words it gives the chance to thread

B to execute on the *Virtual Processor*. It places thread A on any indicated queue either processor-local (like ioQueue) or global queue (such as proxyWaitingQueue etc.). For global queues, it locks the queue before putting the thread (*A*) on it. It also ensures whether this thread is not available on any other queue. This method is also responsible to set the value *true* to thread's *beingDispatched* field in order to prevent it (*thread A*) to not schedule on some other virtual processor as thread A's stack in use by some other dispatcher. Then it transfers the control to *morph* method of VM_Thread.

morph() performs some housekeeping operations and transfers the control to the *dispatch* method of VM_Processor which is responsible to select the next thread (*B*) to execute. *dispatch()* of VM_Processor does selection of the next thread to be executed on the processor by invoking the *getRunnableThread()* of the VM_Processor.

getRunnableThread() will check the transferQueue to see whether the next thread (B) is a collector thread (if yes it would return the thread) or mutator thread. If it is mutator then it would check whether this thread's stack (B's stack) is used by other dispatcher, if yes it would enqueue that thread back to the transferQueue. If the thread's stack is not used by any other dispatcher then this method transfers the thread (B) from the transferQueue to readyQueue for the execution.

dispatch() makes the current running thread (A) to previous thread and take thread (B) from the *readyQueue* and makes it *active thread* and then invokes the *VM_Magic.threadSwitch* to save the hardware context of thread A and restore the hardware context of B.

VM_Magic.threadSwitch method switches the threads; saves and restores some registers such as non-volatile *fpr (frame pointer), gpr (g*eneral purpose) registers etc. In addition, it takes two parameters currentThread and restoreRegs; *currentThread* parameter is for currently running thread and *restoreRegs* parameter is for registers from which it restores the hardware state of another thread B. Finally, it appears as if B's previous call to dispatch has just returned and it continues executing [7].

A diagram illustrates thread switching at code level is attached in appendices section (appendix A) at the end of this dissertation.

3.2.2 JikesRVM Load Balancing Mechanism - migration of threads among virtual processors

In JikesRVM load balancing mechanism, a thread moves from one virtual processor to another (if transferring virtual processor has much runnable work and other virtual processor has nothing to do). This transfer of threads or sharing work among virtual processors happens because either a timer tick or a collector thread interrupts the thread. These movements of threads among virtual processors will not applicable if the thread is last executable thread on the current *virtual processor.*

We have mentioned that an idleQueue is associated with each virtual processor, which contains only one, idle thread. Thus, when a *VM processor* has no other executable thread then its idle thread wakes up and runs in order to post request for work. Then it enters in the busy-wait cycle for a very short time (approx. 0.001 seconds). If no work arrives in this period, the virtual processor surrenders rest of its time slice back to underlying operating system [7]. However, if any virtual processor notices that this one needs work, it will add an extra runnable thread (if it has extra) by transferring to this processor's transferQueue. And when work arrives, the idle thread goes back to the idleQueue of the processor and the transferred thread starts execution.

Load balancing is managed by three main methods of VM_processor class of scheduler package: transferThread(), scheduleThread() and chooseNextProcessor(). s*cheduleThread()* puts the thread on the most lightly loaded virtual processor. This method is responsible to check whether the thread is the last runnable thread on the processor, if it is, then *scheduleThread()* does not allow it to move on other virtual processor. It also checks that other virtual processor(s) is/are idle; if they are idle then it transfers the threads by invoking the *transferThread() to* add the threads to virtual processor's transferQueue in order to transfer the threads. Furthermore, this method

distributes the threads to available virtual processor in a round-robin fashion by invoking the *chooseNextProcessor()*.

3.2.3 Scheduling among mutator threads and collector threads:

As we discussed in the previous chapter that one collector thread associated with each virtual processor so it allows JikesRVM to run in one of two modes: either the Java threads are running or the collector threads are running. When the mutator threads are running, the collector threads will stay idle and when the garbage collector threads start executing, mutator threads will go to sleep and will only resume automatically when GC threads return.

When normal Java threads makes a request for memory space that the allocator cannot satisfy or the application makes an explicit call (System.gc()) then garbage collection is triggered. Once a collection is requested, the collector threads are notified and scheduled for execution like normal threads on their corresponding virtual processors. When the garbage collector threads start executing, they disable the thread switching on their corresponding virtual processors and they intimate other GC threads that they have control of their virtual processors. In addition, these collector threads perform some initialization and synchronize themselves with each other at the first rendezvous instance. It is called the *parallel garbage collection.* JikesRVM executes its garbage collector threads in parallel.
Another salient feature of JikesRVM's scheduling is, when GC threads are running, all mutators must be at yield points. Since all yield points are safe points in JikesRVM, the collector threads can start with collection to reclaim the memory space. JikesRVM uses stop-the-world garbage collection algorithm as it stops the thread switching and execution of mutators. Furthermore, when all the mutator threads reach the safe points then collector threads start reclaiming the memory.

When the collection is finished, the GC threads re-enable the thread-switching on their virtual processors and then wait until another collection request being made. In contrast,

mutator threads do not need notification; they start up automatically as the GC threads release their virtual processors.

3.2.3.1 JikesRVM's parallel garbage collection

In order to perform the parallel garbage collection, all collector threads must synchronize themselves at the end of each of three phases as indicated in the figure 3.3. For this purpose, JikesRVM provides the rendezvous mechanism where no GC thread proceeds before the rendezvous point until all GC threads reach it. In other words, all the GC threads meet at the rendezvous point and start claiming the memory together.

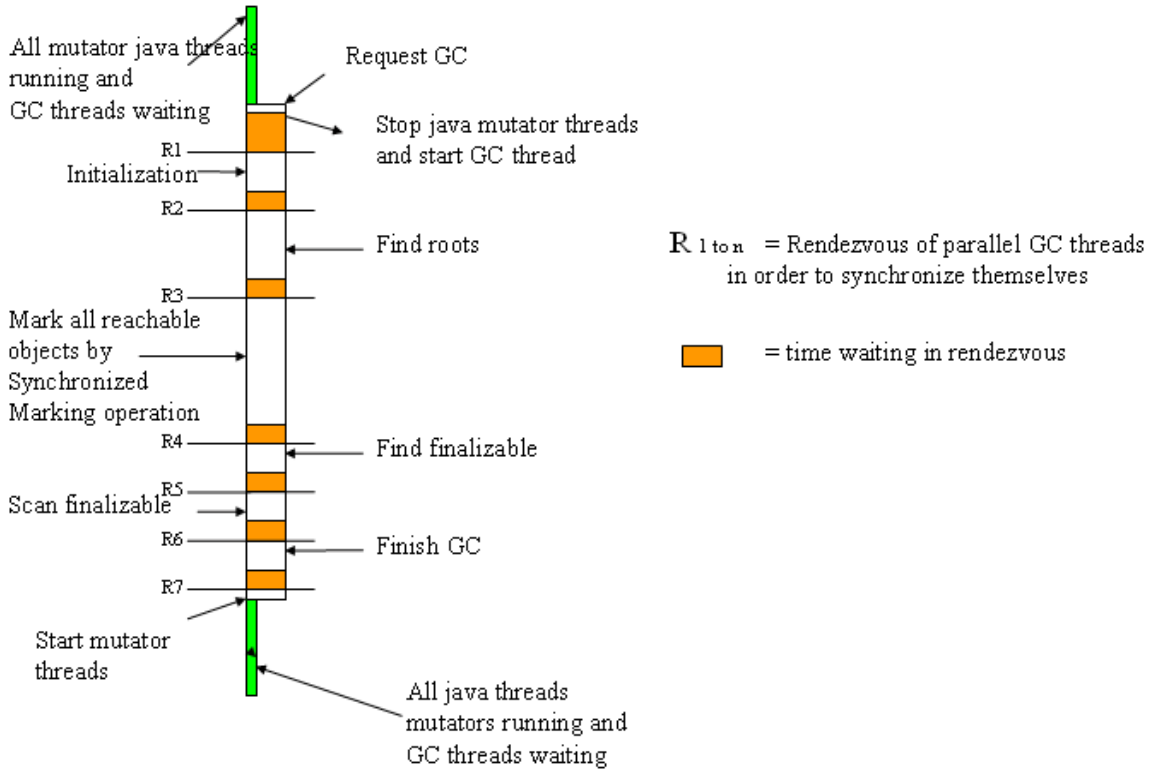


Fig. 3.3 Transition b/w mutator and collector threads and phases of parallel GC [8]

In initialization phase, all collector threads participate. All collector threads (copying collector threads) make new copies of their own objects and their virtual processor's object because the old copy will be discarded after collection and changes are made into

new copy (this is the downside of copying managers because they waste about half of this memory).

In root identification and scan phase, GC threads contend for each mutator's stack and for the JTOC and scan them in parallel for roots and then mark them. They place marked roots (also known as object reference) into the work queue and then they mark the objects that are accessible by the object references available in work queue. Each GC thread marks one object because of synchronized marking operation. Then for scanning the object reference, it removes the objects from the global work queue**.** In finish phase, collector threads get the empty space for the next mutator cycle. All three phases and switching among mutator and collection threads are indicated in the above figure [8].

3.2 Thread States in JikesRVM

At a conceptual level, every thread in the JikesRVM can be in one of the eight states of its lifecycle. Figure 3.4 shows the thread states. These are following:

New: When a thread creates (that is, when thread's constructor is called) then it will be in new state. It remains in initial condition until its *start* method is called. It can start execution immediately if no other thread(s) is running on the available cpu(s) or can wait for its turn.

Runnable/Active: A thread starts running once its *start* method called. When the scheduler assigns thread to available CPU and when they start using CPU's cycle time then they come in the runnable state.

Ready: threads that are in readyQueue are waiting for a chance to run. When the active threads finish their task or block into waiting state then threads in ready state become active and schedule on the processor from the ready state.

Blocked: A thread is in a blocked state if it is waiting for a lock on monitor. This will remain in the block state until the owner of the lock releases it.

I/O Waiting: A thread is in I/O waiting state if it cannot be run because it is waiting for some specific event to occur. For instance, an active thread can be suspended due to blocking input/output operations and then it deadlocks the virtual processor until it resumes.

Suspended: A thread is in a suspended state if it is waiting for On-stack replacement (OSR)*. Threads are rescheduled by recompilation when OSR is done.

Waiting/ Sleeping: A Thread is waiting for some event or for some specific time is in this state. A thread can be in a sleep state for some milliseconds. Waiting thread could be notified by the other threads through sending the signals. Furthermore, waiting threads can also wait for some specified time (*timedwait*). In these cases, threads do not hog the CPU.

Termination: A thread is in exiting state if its *run()* method returns or its *exit()* method has been called. Threads can be terminated by their own will or killed forcibly. In other words, if they finish their work then they naturally terminates, or they can exit the JikesRVM by calling *killInternal()* method.

** OSR is used to improve performance with adaptive recompilation, transferring execution from slow code to faster code* [9].

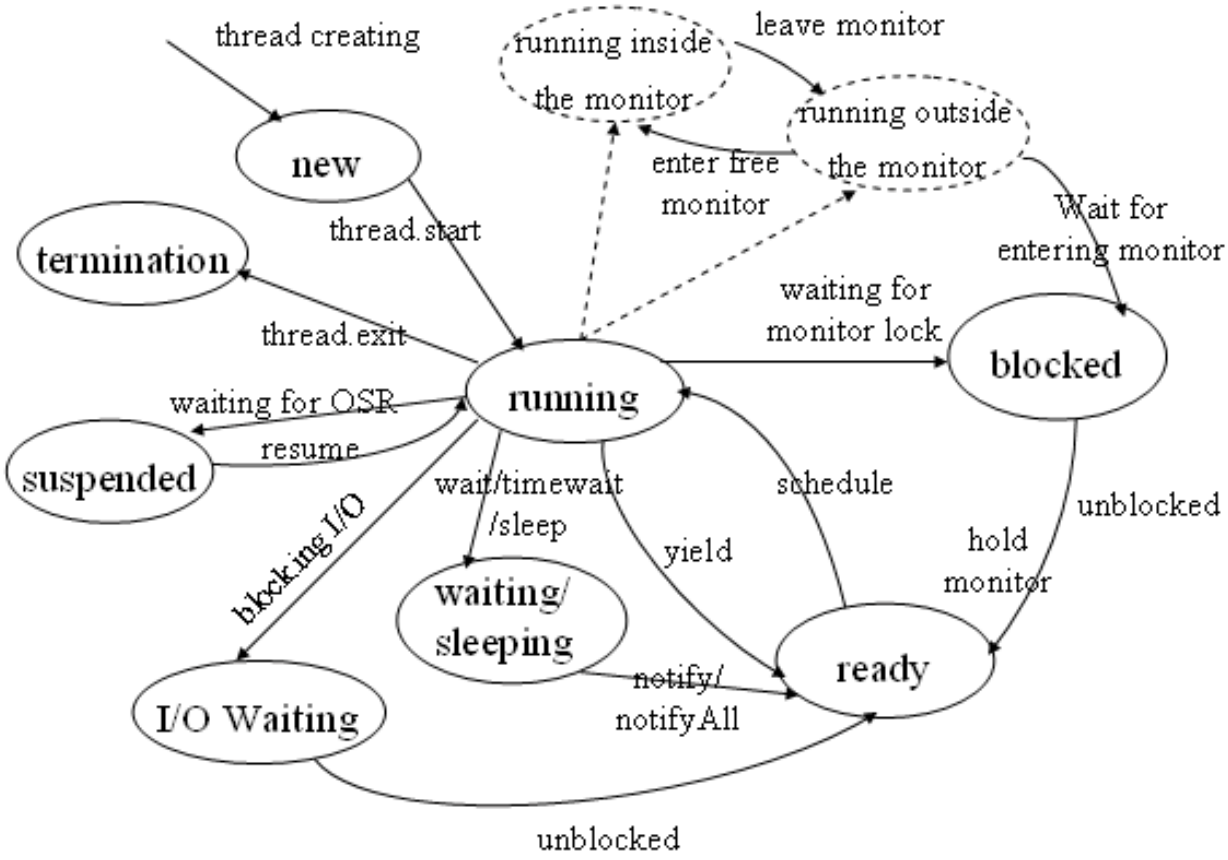


Figure 3.4 Thread States in JikesRVM

3.4 Thread Queues

Threads that compete for CPU resources in order to execute are either stay in the thread queues or proxied. In JikesRVM, there are two kinds of queues: processor-local queues and the global queues.

3.4.1 Processor-local queues are associated with each virtual processor since a *VP* can only execute one thread at a time so there is no need to synchronize access. In contrast, global queues need synchronized access because they are common for all threads which are running on different virtual processors. A number of processor-local queues are following:

An ***idleQueue*** holds an idle thread that will execute (post request for work) whenever virtual processors have nothing to do.

A ***readyQueue*** holds only ready to execute threads, which are waiting for a time slice in which they have to run.

The ***transferQueue*** helps in load balancing by transferring threads among virtual processors when the other processor has nothing to do. Any virtual processors can put the threads on this queue then the thread moves from this queue to readyQueue for execution.

The ***ioQueue*** holds a number of threads that are waiting for availability of input/output data. To prevent from blocking the thread I/O operations in the green thread model maintain a several descriptors that need to be checked to poll for data availability. When data becomes available on file descriptors, determined using the non-blocking *select* system call, then the corresponding thread resumes its execution.

The ***processWaitQueue*** holds threads that are waiting for a sub-process to execute. This is used to implement the exitValue() and waitFor() of java.lang.process.

3.4.2 Global queues need to synchronize access. The global queues are following:

The ***processorQueue*** handles the set of virtual processors. Its instance *deadVPQueue* holds a special virtual processor and pthread so that when a virtual machine thread (*VM_Thread*) calls the native code for the first time, the virtual processor and pthread are created (virtual processor implemented as pthread) and run together until the virtual thread terminates and after serving the caller thread the virtual processor enqueues back into deadVPQueue for recycling. Thus if other threads call to native code then a request to reuse the virtual processors from deadVPQueue is made in order to execute them.

The ***wakeupQueue*** (type of VM_ProxyWakeupQueue) is a queue of proxies for threads that are awaiting timeout on this object. This queue is the mechanism to implement to

Java sleep semantics. In order to implement timed-waits, a thread needs to be in both waiting queue and the wakeup queue. But the current threading model allows a thread to be in one queue at a time so in order to perform this both queues are of proxies for thread rather than thread. Unlike thread, proxy can be on both a waiting queue and a wakeup queue. The VM_Proxy class of the scheduler package is responsible for representing the same thread on more than one proxy queue.

The ***waitingQueue*** (type of VM_ProxyWaitingQueue) is a queue of proxies for threads that are awaiting notification on this object. When a notification is received from an other thread, the thread is taken from this queue (waiting queue) and transferred to the readyQueue. This queue is the mechanisms to implement to Java's wait/notify semantics.

The ***enteringQueue*** contains a number of threads that are contending for a lock, so it is also called the lock queues and is guarded by mutex.

The ***CollectorQueue*** all collector threads reside in this queue. c*ollect()* of VM_CollectorThread is called by the mutator thread when the object allocator is not able to manage the requested memory. The caller thread also pass the handshake object when called the *collect()* for collection.

The ***DebuggerQueue*** contains one debugger thread, which can be scheduled by an external signal.

3.5 Synchronization

Synchronization is required for concurrent execution on Symmetric Multiprocessing (SMP) environments. Mapping the Java threads to virtual processors allows the tight integration of synchronization support with thread switching in the m:n green thread model.

The JikesRVM's thread synchronization is based on monitor mechanism as it is developed in Java. The JikesRVM implements both kinds of thread synchronization: cooperation and mutual exclusion. Cooperation is supported via wait and notify methods of *object*, it enables threads to work together to achieve a common goal. For mutual exclusion, JikesRVM implements the locking-unlocking in order to enable multiple threads to independently work on shared data without intervention with each other.

3.5.1 Mutual Exclusion (Locking/Unlocking)

In JikesRVM, both thread scheduling and load balancing require atomicity and singular access to the global data structure. In addition, the user threads also need to synchronize access to their global data. In order to get the synchronized access; JikesRVM's scheduler uses three types of locks:

1. **processor lock**
2. **thin lock**
3. **thick lock**

3.5.1.1 Processor locks play a significant role in thread scheduling and load balancing. They underlie other locking mechanisms. Processor locks are implemented as Java objects in JikesRVM, with a single field *latestContender* that identifies the virtual processor that owns the lock. They are intended to be held only for a short period as they "busy-wait". Furthermore, they cannot be acquired recursively.

The *lock()* method of (VM_ProcessorLock class) in the scheduler package performs operations to acquire the processor lock for the thread that is running on the virtual processor. Getting and releasing a lock involves atomically reading this lock field *latestContender* and setting the value to this. If this field is *null* it means this lock is not owned, otherwise this field points to the virtual processor's id which owns this lock; in other words, this field identifies the owner of the lock. A processor lock is released by storing *null* value into the owner field.

If the virtual processor fails to acquire a lock due to contention, then it tries again by spinning on this processor lock's *latestContender* field. Processor locking also implements the MCS (Mellor-Crummey and Scott) locking mechanism [10]. When MCS Locking is set, the processors spin on processor local data with the last virtual processor on a circular queue (of virtual processors); spinning until it gets the lock. It also updates this queue by adding itself into this circular waiting queue, for this a processor must succeed in setting the latestContender field to IN_FLUX. The major advantage of MCS locking is it's a queue based spin locking mechanism [10].

If an attempt to lock or unlock a processor-lock has failed, assuming contention with another processor, a *backoff* mechanism is used which delays for a different time period on each processor to try to solve contention and to some extent in order to increase the likelihood that a subsequent retry will succeed in locking or unlocking.

In addition, a thread will not yield control of a VM_Processor while it owns a processor lock because it cannot release the lock until it resumes execution. The identity of the virtual processor which owns the lock is maintained in a dedicated Processor (PR) Register.

3.5.1.2 Thin and Thick Locks

The JikesRVM's other locking scheme is based on the thin locks also called light-weight locks: if there is no contention among threads then thin locks are used to lock the resource (object) and the bits in the object field are used for this purpose (as discussed in the second chapter section 2.4.1, bits in the status word fields in the object header are used for locking). In contrast, in case of contention (if two or more threads are competing for same object) these bits in the object header represent a heavy lock.

In contrast to the processor lock, thin locks can be recursively held by the same thread. One bit from the bit field in the status word tells whether a thick lock is associated with the object or not. If the thick lock is not associated then the remaining bits are divided in to two fields: thin lock owner *ownerId* subfield that represents the thread holding a thin lock on the object and the recursion count field *recursionCount,* it records the number of times the owner thread has acquired the lock. On the other hand, if the thin lock is associated, the rest of the bits in the locking field become the index of this lock in the global array of thick locks. This global array is partitioned into virtual processor regions to allow unsynchronized allocation of thick locks. If any thread does not lock the object then all the bits in the locking fields are set to zero [5].

In addition, if the lock is not acquired (that means all locking bits are zero) then in order to acquire the lock, the thread sets the owner bit field to its identifier. The identifier of the thread, which is currently running on a virtual processor, is kept in a dedicated *thread identifier* (TI) register.

When an attempt to lock an object fails there are three situations either *try again (busy-wait cycle) or yield and then try again* or *inflate the lock* [11]. Currently, the situation is handled through yielding ‘forty’ times*, and then inflating. Inflation means transformation the thin lock into the thick lock [12].

** The current value was for the portBOB benchmark on a 12-way SMP (AIX) in the Fall of '99. (VM_ThinLock.java)*

The thick lock is defined as VM_Lock in the scheduler package of JikesRVM. It has six fields: the *mutex field* is a processor lock that synchronizes access to thick lock or in other words it handles contention for the data structures of this lock. The *lockedObject* is a reference to the object being locked. The *ownerId* contains the id of the thread that owns the lock. The r*ecursionCount* is responsible for recording the number of times the owning thread has held the lock. The *entering* field is a queue of threads contending for this lock guarded by the lock. *waiting* field is the queue of threads awaiting the notifications on lockedObject guarded by the mutex [5].

Currently, each processor maintains a pool of free locks. When a processor inflates a lock, it is taken from this pool and when a processor deflates a lock it gets added to the processors pool [13]. In contrast, deflate*()* gets invoked when the lock is unlocked and there is nothing on either of its queues.

Above-mentioned locks implement the mutual exclusion, and it refers to the mutually exclusive execution by multiple threads.

3.5.2 Cooperation (Wait/notify)

The VM_Lock class of the scheduler package provides the JikesRVM's support for monitors and also support for wait/notify synchronization (methods of java.lang.Object). Cooperation is useful when one thread needs some data in a particular state and another thread is responsible for getting the data into that state. The JikesRVM implements this form of behaviour using the wait/notify/notifyAll semantics. In this type of monitor, a thread that currently holds the lock/monitor can suspend itself by executing the *wait()* of VM_Lock. Once a thread executes *wait(),* it releases the monitor and enqueue into a *waitingQueue* in favor of some other threads so that they can acquire the lock.

This thread will wait into a *waitingQueue* until some time later another thread executes the *notify()*. Furthermore, the enteringQueue can schedule other threads (as enteringQueue contains a number of threads that are contending for lock. Once another thread executes notify, it continues to hold the monitor until it frees the monitor of its own wish (either completing its own task or executing the *wait*). When the notifying thread frees the monitor, the waiting thread will wake up from the waitingQueue and will re-acquire the lock.

The waiting thread suspended itself because the data locked by it, is not in a state that would allow the thread to continue execution. In the same way the notifying thread executes the notify method after it had put the data protected by lock into a state required by the waiting thread.

3.6 Drawbacks of M:N Green Thread Scheduling

We list the issues with m:n green threading below:

1. One of the problems in the current threading model is with blocked native methods. Most of the native input/output operations are blocking I/O and the Java threads that call these operations will block until the input/output operation finishes. For instance, if a virtual processor (VM_Processor) schedules five Java threads and one of the threads calls the blocking I/O operation, as a result the whole VM_Processor will be blocked until that I/O operation completes and will be unable to schedule the other four threads. However, JikesRVM avoids this problem by capturing/hijacking blocking input/output and replacing them with non-blocking operations. The calling thread is then suspended and placed into IOQueue. The virtual processor (VM_Processor) checks the awaiting i/o operations at short polling intervals and after they complete the operations, the virtual processor brings the calling thread back into the runningQueue from the Input Output Queue (VM_ThreadIOQueue).

But the JikesRVM can not insert the 'yield points'* into native methods and can not always be able to intercept blocking I/O operations in native code. In other words, it is fairly complex to support the blocking native code together with the m:n threading in JikesRVM. ()

2. Another drawback is that too much control logic is embedded in the low-level, in order to call and return (transition) between C and Java code, which makes the code harder to maintain.
3. Java threads may be scheduled for execution by different operating system's thread (pthreads) at different stages during in its execution. Thus, it can increase the performance cost caused by cache invalidation due to thread switching.
4. In current thread model scheduling is *non-preemptive* scheduling; and no priorities are assigned to threads so threads are unable to take optimum benefits of processors, as they are not scheduled by operating system. For true concurrency, underlying Operating System's incorporation is required
5. Transition between mutator and collector threads could be a performance issue if the number of mutator threads is large.
6. Threads in this model are lacking the cooperation of Operating System.

3.7 Summary

In this chapter, we have discussed the current threading model supported by JikesRVM and how the Java threads map on to virtual processors and how the virtual processor execute them efficiently with the support of other threading components such as thread queues and locks. We have also seen the fast transition among threads including mutator and garbage collector threads.

Please note: yield points are the hidden thread switch points in the compiled code of the method inserted by JikesRVM compiler [14].

We also indicated a set of thread queues in the entire threading system which plays a pivotal role in multi-threading environment. Synchronization is another important component of Jikes threading supported by Java monitors and some synchronized methods of java.lang.Object. JikesRVM also supports the implementation of an efficient way for load balancing and distributes work among virtual processors to achieve high performance in SMP environment. However, at the end of chapter we have noted drawbacks in the present model which raise some issues in terms of performance. In next few chapters, we will analyze a new threading model called native thread model (using POSIX thread library) in order to remove those performance issues and to make the JikesRVM threading more robust.

CHAPTER 4

Re-factoring and Design Pattern

4.1 Introduction:

In the previous chapters we talked about JikesRVM's components and current thread scheduling model. In this chapter, we will specify the work that we contributed and the approach that we adopted for re-factoring the existing scheduler API in order to introduce new native thread model without breaching the consistency of the present system. In addition, our aim is to keep the code organization simple and clean. For this purpose, we decided to re-factor the current system into two models and sub-divide the existing scheduler package, where all multi-threaded programming resides*, into two packages: one for each model (green thread and native thread). We kept the classes of scheduler package as a base wrapper classes and they contain common functionalities of both the thread models. In essence, these two models contain the code for the specific threading system. For example, on the one hand all user space thread scheduling will reside in the green thread model; on the other hand all kernel-level preemptive scheduling will be contained into native model. The following block diagram represents package structure for the scheduler API:

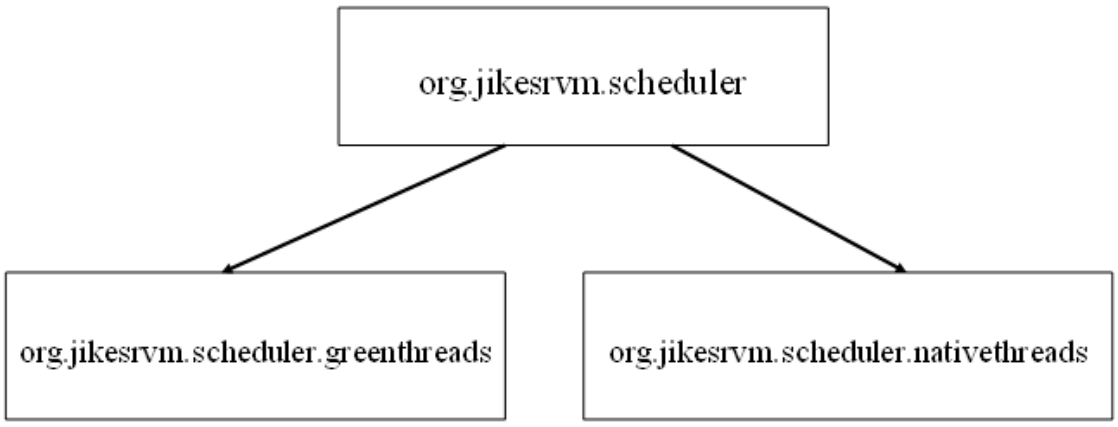


Figure 4.1 Division of the JikesRVM's scheduler API

**In practice, lazy programming practices had placed scheduling code outside of this package.*

We will discuss our approach and the design pattern we adopted next. We will also see some code examples to better explain this work.

4.2 Our approach to re-factorization

With the aim of simple and clean code organization, we decided to implement one popular design pattern called the 'factory design pattern'as it was matched our requirements. In addition, we adopted and implemented this in order to keep the consistency among classes in the scheduler API.

4.2.1 Factory Method Pattern

"Define an interface for creating an object, but let the subclasses decide which class to instantiate [15]. *The Factory method lets a class defer instantiation to subclasses*".

The main purpose of this method is to create the objects without specifying the *exact class* of objects that will be created; in other words, subclasses decide which class to instantiate. Above all, factory methods are static methods that return an instance of the sub-class at run-time.

4.2.2 Why we used the factory design pattern

We used this pattern because it proved an efficient design model for refactorization. Furthermore, a number of reasons of using factory method are to obtain the reference of sub-classes saves lots of work, an easy implementation and most importantly if the requirements change in the future we would not need to make changes in every class that uses our base class (e.g. VM_Thread). We will need to make only one change in one class in order to meet the new requirements. The main advantage of factory method is new threading model (e.g. native model for Windows) can be added without changing the framework. Moreover, this is useful when we don't know what concrete implementation whether green thread or native pthread of VM_Thread has to instantiate. We delegate this

responsibility to the factory method. The UML class diagram of our factorization is given below in fig. 4.2, considering only one class VM_Thread in scheduler API:

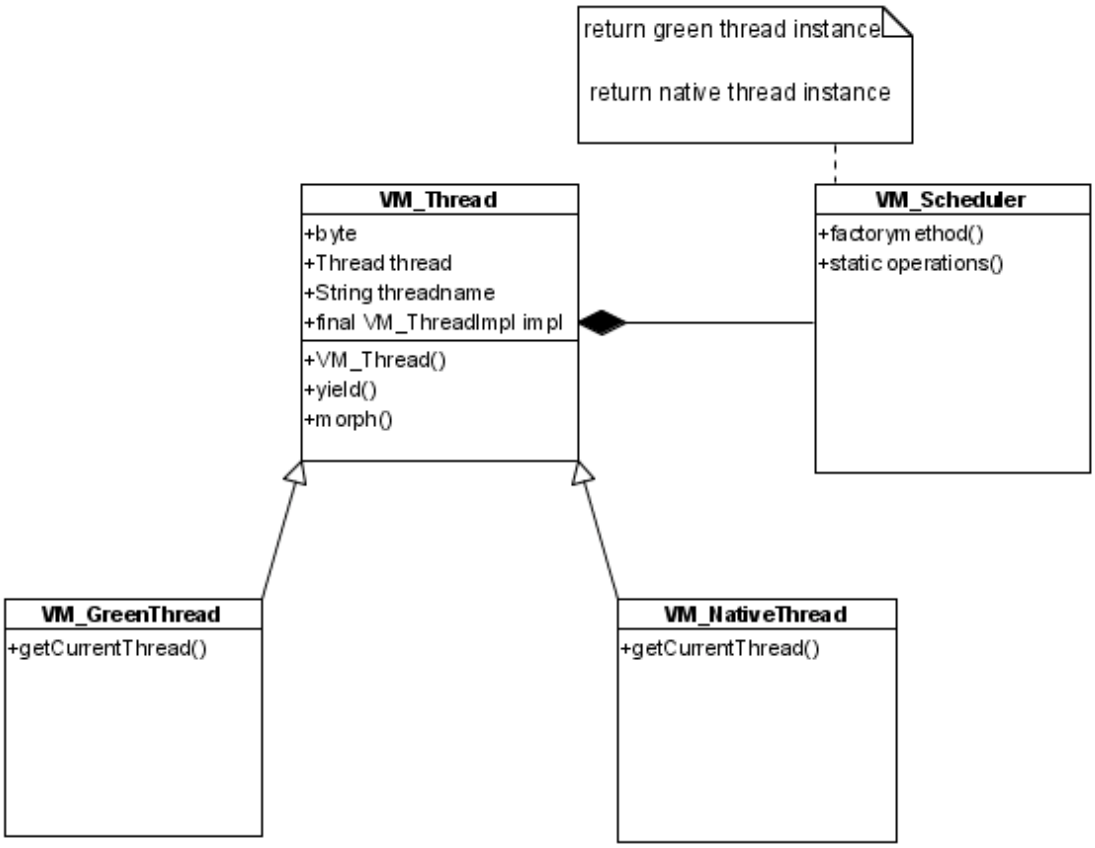


Figure 4.2 UML Diagram of Factorization of Scheduler API

4.2.3 Factory Design Pattern in Our Framework

The factory method pattern returns an instance one of possible class (green or native). Which class it will return depends on provided arguments on the command line by the users. Usually both of the classes (VM_GreenThread or VM_NativeThread) it returns extend the base class such as VM_Thread, but each of them is optimized for specific type

of threading system; in other words to realize their application specific implementations. For instance, to provide a green thread mechanism, we defined a class VM_GreenThread.

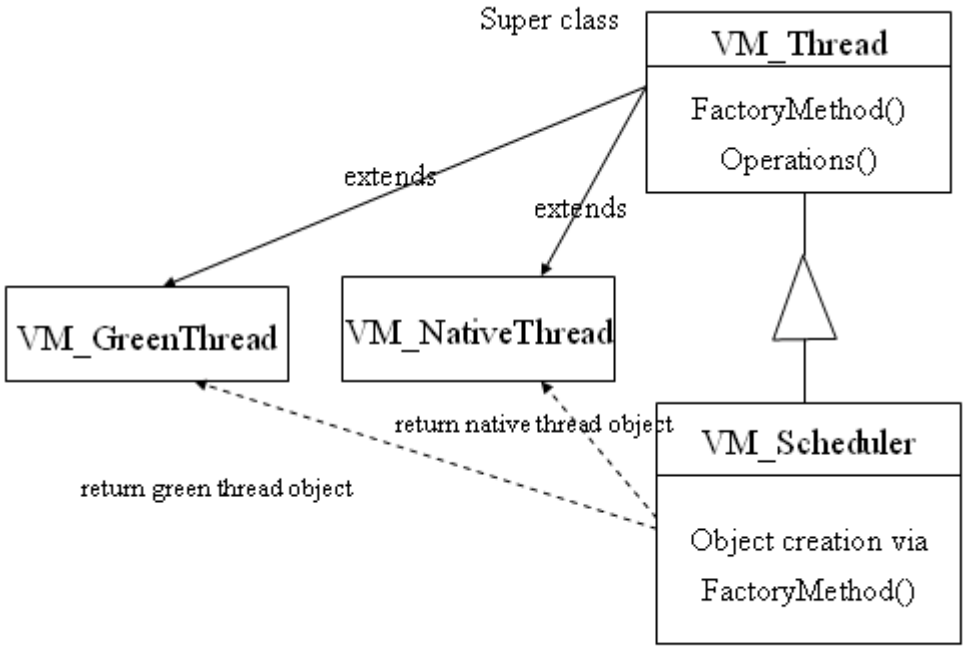


Figure 4.3 Structure of Factory Pattern in SchedulerAPI

In the structure presented above, This VM_Thread wrapper class is responsible for managing the threading systems and will create them as required, when the users pass the arguments; -vmt for green and -pt for native posix thread, for example. We defined one more class is called VM_Scheduler which creates the factory method (s) to return an instance of either VM_GreenThread or VM_NativeThread.

This factory method in the VM_Scheduler decides at runtime which subclass has to instantiate by using the arguments passed by the user at command line. Then, VM_Scheduler will create the instance of either green or native model and pass them to virtual machine thread system (VM_GreenThread) or OS's native thread system (VM_NativeThread) for invoking their functionality. The same design pattern applies to

other classes in the scheduler package. The program flow is indicated in the following figure.

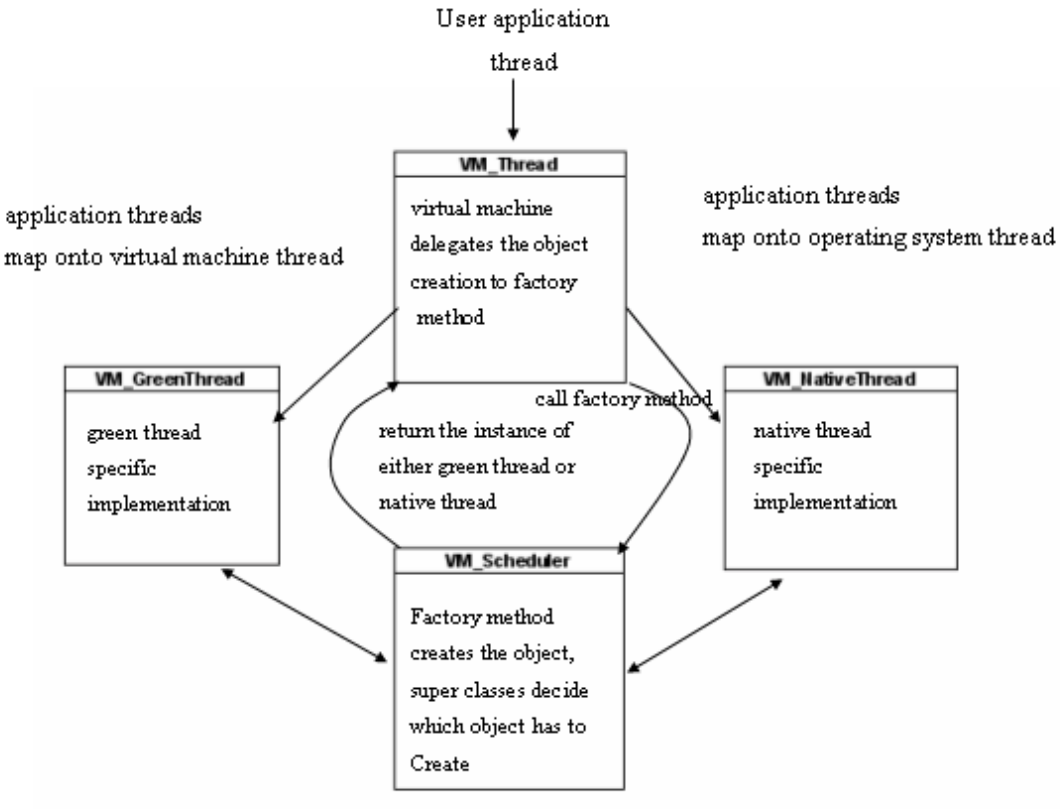


Figure 4.4 Program Flow

4.3 Our Design Attempt

Our design attempts to make classes in scheduler package thin wrapper around the implementers (green or pthread model) of interfaces located in *org.jikesrvm.scheduler*. For instance, making class VM_Thread a thin wrapper around implementors VM_GreenThread and VM_NativeThread. So, VM_Thread maintains a reference to an instance of the implementing class and redirects instance methods to it. Static methods, including the calls to create the implementing instance are redirected via static classes in VM_Scheduler. We used this design pattern with many of the classes originally located in the scheduler package.

We retained the public fields in the thin wrapper classes in order to keep the consistency with other subsystems of the JikesRVM and maintaining a single copy of data. For this, our implementor classes maintain a back-link to the VM_Scheduler class.

4.4 Re-factoring at Code Level:

```
// VM_Thread – super class of green thread and native thread
public abstract class VM_Thread {

protected VM_Thread(byte[] stack, Thread thread, String name,

boolean daemon, boolean system, int priority)

{

…..

}
public static void yieldpointFromPrologue() {

org.jikesrvm.scheduler.greenthreads.VM_GreenThread.yieldpoint(PRO
LOGUE);
 }

public static void yieldpointFromPrologue() {
org.jikesrvm.scheduler.greenthreads.VM_GreenThread.yieldpoint(PRO
LOGUE);
  } }
```

```
// VM_Scheduler, also consist of factory methods
public abstract class VM_Scheduler {

 // back-link
private static final VM_Scheduler singleton = new
VM_GreenScheduler();

public static class ThreadModel extends
       org.jikesrvm.scheduler.greenthreads.VM_GreenThread {

       public ThreadModel(byte[] stack, String s) {
   // passing to implementor class VM_GreenThread
       super(stack, s);
       }
              public ThreadModel(String s) {
      super(s);
   }

public static final class LockModel extends
org.jikesrvm.scheduler.greenthreads.VM_GreenLock {
 }
// factory method
```

```
private static VM_Scheduler getScheduler() {
  return singleton;
       }

// static method

public static VM_Thread getCurrentThread() {
    return
VM_Magic.objectAsThread(VM_Processor.getCurrentProcessor().active
Thread)
   }


// green thread implementor class VM_GreenThread

// extend from VM_Thread

public class VM_GreenThread extends VM_Thread {

public VM_GreenThread(byte[] stack, String name) {
    this(stack,
       null, // java.lang.Thread
       name,
       true, // daemon
       true, // system
      Thread.NORM_PRIORITY);
  }

// thread specific implementation
public static void yieldpoint(int whereFrom) {
….
}



// native thread Implementor class

public class VM_NativeThread extends VM_Thread {

public VM_NativeThread(byte[] stack, String name) {
    this(stack,
       null, // java.lang.Thread
       name,
       true, // daemon
       true, // system
       Thread.NORM_PRIORITY);
 }
protected void notifyInternal(Object o, VM_Lock l) {
…
 }
```

4.5 Flexible Threading model – User's choice

We have given the users a choice of both the threading models by providing command line arguments so according to their applications they can choose the desired model. If they are using an SMP environment then native model is a better choice because on the multi-processor system, this model can easily split threads among processors and can greatly improve the performance. Although, if users are running applications which create a large number of threads such as server applications then in these cases green thread model has proven better and faster.

In order to supply the command line arguments so that Jikes virtual machine behaves according to them, we made some changes in the Jikesrvm's configuration files such as VM_Properties, VM_CommandLineArgs and build.xml. We are discussing them briefly in the following text.

4.5.1 Properties defined in VM_Properties

In current situation, there is no command line argument for the selection of threading model since there is by default only one threading model. So in order to take input from the user, we defined two properties both for green thread model and for pthread model in the VM_Properties.java. Properties defined in this Java program control the behavior of JikesRVM and can be set from the command-line.

Below is the code snippet which specifies the properties:

```
public class VM_Properties extends VM_Options {
……
public static int verboseBoot = 0;
….
public static boolean greenThread = true;
public static boolean pthread = false;
}
```

So here the property for green thread is true. The reason is if users do not specify their choice then by default green model will execute and threads will be scheduler at user-space.

4.5.2 Prefixes defined in VM_CommandLineArgs

A list of possible prefixes for command line arguments is defined in the VM_CommandlineArgs. For making standard prefix, we have added two entries in the VM_CommandlineArgs and edited two more cases in *earlyProcessCommandLineArguments* method. This method is responsible for processing of several command-line arguments that need to be handled early in the booting and contains only those command line arguments that require fully booted Virtual Machine to handle. The code snippet is for doing this is as follows:

```
public class VM_CommandLineArgs {
  ......
  public static final int PROCESSORS_ARG      = 29;

  public static final int GREEN_MODEL         = 30;
  public static final int PTHREAD_MODEL       = 31;
  ........
    new Prefix("-X:processors=",       PROCESSORS_ARG),
    ....
    new Prefix("-vmt",    GREEN_MODEL ),
    new Prefix("-pt",    PTHREAD_MODEL ),
...
   switch (type) {

      case GREEN_MODEL:
        VM.greenThread=true;
        break;
      case PTHREAD_MODEL:
        VM.greenThread=false;
        VM.pthread=true;
        break;
.....
// other code…
case PROCESSORS_ARG: // "-X:processors=<n>" or "-X:processors=all"
....
}
```

It ensures the encapsulation among threading models; if the program runs with a particular threading model chosen by user on command line then JikesRVM's threading system will not switch from one thread model to another in the middle of execution. For example, if user chooses native model on the command-line then the thread scheduling will be done with native model only and program control cannot get into green thread model.

4.5.3 Configure build.xml

In order to add new configuration details in JikesRVM, we added some details in the build.xml that contains all the configuration details of Jikes. Following the tradition of build.xml in setting attributes, we added two arguments in this; you can find some changes into build.xml in appendices section (appendix b). Moreover, you can also find the new command line options for executing programs with the green and pthread model.

4.6 Summary

In this chapter, we have discussed our refactoring mechanism, which we contributed in JikesRVM and have explained our choice of using factory pattern. We also copied a small code snippet here which is important to understand the refactorization. This design pattern we applied for most of the classes which are originally located in the scheduler package. Some of them are thread model specific (e.g. green thread model) so they have brought in to green thread model without factorization. For instance, most of the queues existed in scheduler package that are not useful for native pthread model, we are keeping them in green thread model package (*org.jikesrvm.sceduler.greenthreads*). We have also given choice to users by providing them command-line arguments so that they can choose the better model according to their requirements. For this we have made changes in some configuration files of JikesRVM.

In essence, this factorization ensures a clean and consistent organization of the code and provides flexibility to future requirements.

CHAPTER 5

Native Thread Model

5.1 Introduction

In this chapter, we will discuss the native thread model and how this model will work in the real system. Furthermore, we will present an analysis of how to implement the major components of the native threading model such as monitorEnter/monitorExit, cooperation, scheduling, yield, and thread cancellation. by using POSIX pthread library.

The native thread model defines that each Java thread created by JikesRVM corresponds directly to a single thread in the OS kernel, also known as 1:1 threading model. In other words, one Java thread maps to one pthread and the OS scheduler further multiplexes this pthread to kernel thread (*NB*: in Linux there is a 1:1 correlation). Unlike the green thread model, in this model JikesRVM is not responsible for scheduling the threads, the kernel is the only one which selects and schedules the threads. In essence, all scheduling of threads is done by kernel.

.

5.2 Native Thread Model

This 1:1 native model allows many threads to run simultaneously on different processors in Symmetric multiprocessing (SMP)* environment. This also allows threads to continue to run, even on the uniprocessor, if one or more threads issues a blocking system call. This model is simple to understand and transparent to the programmers because there is tight coupling between the programmatic abstraction (user threads) and the kernel thread. [16]. In addition, in this model every thread can be thought as a process. The operating system scheduler makes no distinction in this case between a process and a thread.

** SMP is a multiprocessor architecture where two or more identical processors are connected to a shared memory* [17].

The native thread model is a preemptive threading model, which means thread switching can occur at any time. In addition, threads are scheduled on a priority based mechanisms. Therefore, in this model, if one thread uses its whole time slices, it gets preempted by OS scheduler and another Java thread gets to run instead. The scheduling of the Java threads mapped to native threads is controlled by the underlying operating system's scheduler. Figure 5.1 shows a high-level overview of the pthread native model in JikesRVM.

The downside of this threading model is the overhead for each kernel thread (memory, slots in the scheduling algorithm) that may be excessive for programs that create a large number of Java threads as each thread creation involves a separate kernel thread creation, also called Light-Weight Process (LWP) so it requires additional kernel resources.

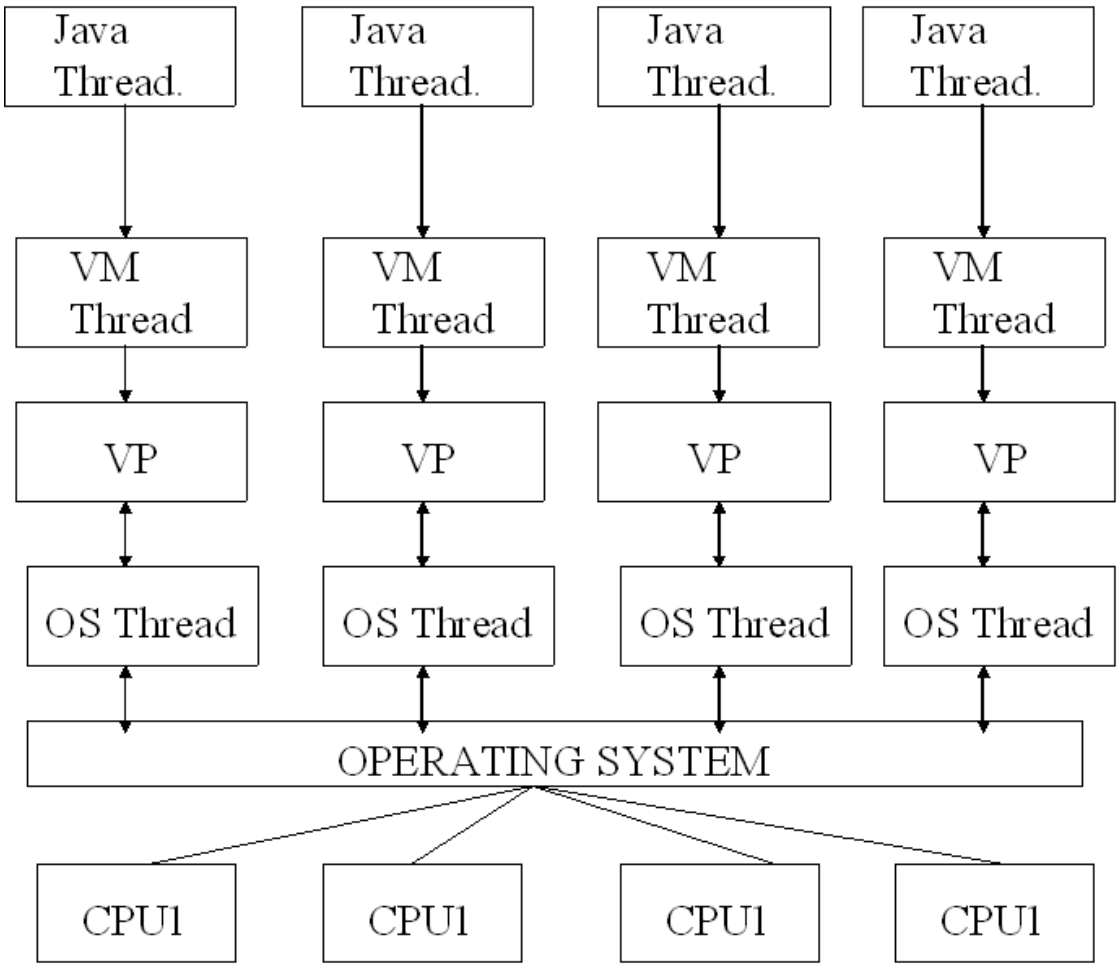


Figure 5.1 Native Thread Model in JikesRVM

5.3 Native Pthread scheduling

Linux pthread library defines two ways of scheduling: one is *process scope* scheduling; all of the scheduling is local (similar to JikesRVM's green thread scheduling) and another is *system contention scope* scheduling where scheduling is done by the kernel. We will use the latter one in the future native thread model of JikesRVM. This scheduling is also called global or bound thread scheduling. This scheduling mechanism also has a policy and priority associated with threads, which further refines the scheduling details at the kernel level [1]. In this scheduling mechanism, the operating system (LINUX) schedules threads just like it schedules processes. That means that threads are scheduled on a preemptive, priority-based mechanism which is a property of the OS.

In system contention scheduling, each Java thread is permanently bound to a LWP meaning the thread will only run on that particular LWP. With this scheduling, Java threads will get the maximum execution time as they will almost never be in a ready state, they will be either active (running), sleeping on a condition variable or blocked because of their tight binding with kernel threads. In addition, they will never be prevented from chewing CPU time by other Java threads. Thus, the JikesRVM users will use this native model when they know that their programs are computationally intensive. Therefore, multi-threaded Java applications with this scheduling will have less thread switching as compared to green thread scheduling.

Our effort will be creating one pthread for each Java thread and mapping that pthreads to individual physical CPUs, which can ensure fast execution for Java threads. This will be helpful for applications that have multiple threads that spend a significant amount of time executing code without blocking.

We will see the best throughput when the number of running threads is equal to the number of CPUs on the machine. If there is a lower number of running threads than available CPUs then there will be idle CPUs and if there are more than available CPUs then the LWPs will compete for the CPU time. In addition, there is never really an

advantage to having more LWPs than CPUs - even if user applications have hundreds of threads that the user wants to time-slice [18]. In order to run such kind of applications, we are able to create hundreds of LWPs by using modern POSIX thread library.

5.4 Binding Java Threads to Kernel Threads - CPU Affinity

The fundamental concept of binding each thread to a separate operating system's thread is cache memory latency because each processor has its external caches of significant size (e.g. 1-4 megabytes). So, replacing the contents of such a cache completely can take a very long time. If a light-weight process is running on CPU 2 and it is context switched off for a short time, then the vast majority of that cache will still be valid. So it would be much better for that LWP to go back onto CPU 2 [1]. Linux library provides the ability to bind one processes (LWP) to one physical CPU, is called CPU affinity. The point is to say that always run this particular process to this particular CPU. The scheduler then obeys the order, and the process runs only on the allowed processor. The operating system will optimize the CPU affinity by itself.

5.5 Implementation of yield method

In contrast to *yield* method of green model, we will implement this via pthread library method `pthread_yield`. This method explicitly forces the calling thread to give up the control of its processor, and then the thread will wait before it is scheduled again on the processor. As we have already discusses in chapter 2 and 3, yield points are also safe point where garbage collection triggers and reclaims unused memory. When a thread executes the yield, the conditions will be checked, if it's a garbage collection point (or GC safe point) then the JikesRVM' garbage collector will start reclaiming the memory. Or, if it is a generic yield then it will call the pthread yield method (`pthread_yield`). Calling thread's state will store into the registers and scheduler will select the next available thread and restore its states from the registers. If the virtual processor does not have runnable threads then the calling thread will immediately reschedule. Figure 5.2

shows this mechanism, thread A calls the yield method and then JikesRVM passes to `pthread_yield` method of pthread library using system call.

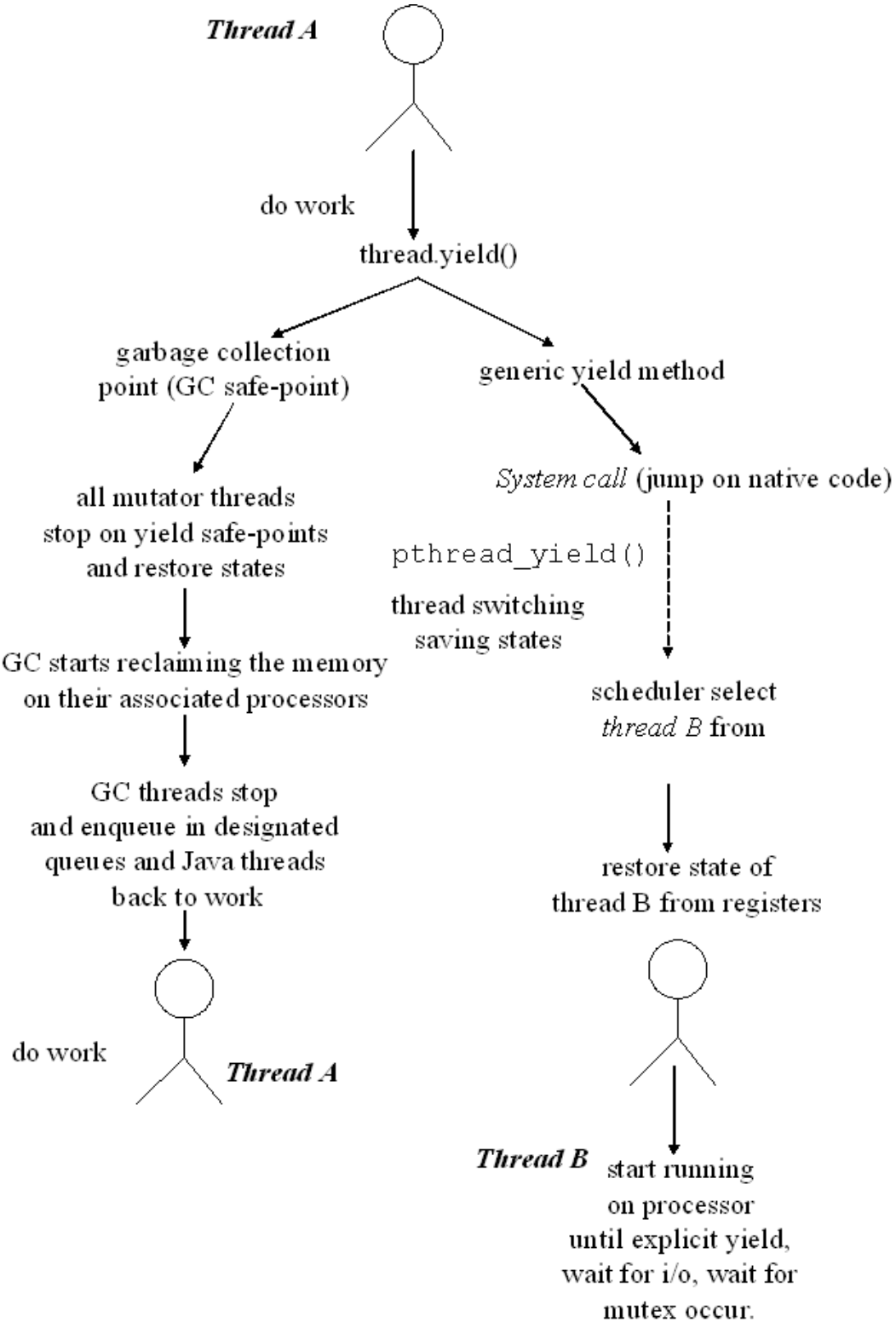


Figure 5.2 Thread yield in native model

5.6 Thread Synchronization - Monitor

In the native model, **mutual exclusion** and **cooperation** map well to mutex and condition variables. We briefly discuss the pthread mutex and condition variables below and then we discuss their use in implementations.

5.6.1 Mutex

Pthread mutex are used for protecting shared data when multiple writes occur. A mutex variable operates as lock in order to protect a shared data. The concept of a POSIX mutex in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. As a result, even if many threads try to acquire a lock, only one thread will own it. Other threads will block and will wait until the owning thread releases the lock. In essence, we will use mutexes for locking the global data to ensure safety when several threads update the same variable and to ensure that only the owner of the mutex is updating the protected global variables. In particular, thick lock will be implemented by using these mutex and condition variable in native thread model.

In addition, whenever a thread needs to acquire a lock on the specified mutex variable, it will call `pthread_mutex_lock()` routine of the pthread library. If the mutex is already held by another thread then this call will block the calling thread and it will wait on the pthread condition variable until mutex is released. In contrast, if the owner thread calls `pthread_mutex_unlock()` then it unlocks the mutex. When a thread finishes its operation on protected global data then it calls this routine in favor of other threads that are waiting to hold the mutex. This method will return error if the mutex was already locked and was held by another thread.

5.6.2 Condition Variable

Condition variables play a significant role in thread synchronization and provide capability of inter-communication among threads both, in one process and in different processes. With the use of conditional variables, we can allow threads to wait without wasting CPU time until some events occur. In addition, several threads can wait on a conditional variable, until some other threads *signals* (in other words, send the notifications) this conditional variable. After being notified, one of the threads waiting on the conditional variable wakes up and performs the operation. By using *broadcast* method of pthread library (work similar to notifyAll of Java primitives), it is possible to wake up all the threads waiting on the conditional variable.

5.6.2.1 Waiting on a Condition Variable

To get the protected global data in some desired state, a thread can wait for the signal calling either `pthread_cond_wait()` or `pthread_cond_timedwait()` methods of pthread library. Both methods take a condition variable and a mutex as arguments. This mutex should be locked before calling the wait function as the condition variables used in conjunction with mutex variables. When these methods are called by thread, calling thread unlocks the mutex, and suspends the execution (wait on condition variable) until other threads signal the condition variable. If the thread awakes by this notification then the mutex is automatically locked again by the wait function, and the wait function returns. In comparison with `pthread_cond_wait()`, `pthread_cond_timedwait()` allows us to specify a timeout for the waiting. In contrast, the `pthread_cond_wait()` would wait for an indefinite period if it was never signaled.

5.6.2.2 Signaling Conditional Variable

For signaling a condition variable, we can use both `pthread_cond_signal()` and `pthread_cond_broadcast()` functions to wake up only one thread and all threads waiting on this variable respectively. They will implement notify/notifyAll primitives of Java. But in our analysis, we will try not to use *broadcast* method, as it could be the cause of more contention among threads for a shared data because all threads wake up together and contend for one single data resource. Below is an example of signal and broadcast methods:

```
//initialize the condition variable
pthread_cond_t   b  =  PTHREAD_COND_INITIALIZER;

int a = pthread_cond_signal(&b);  // signal only one thread
Or
int  a  =  pthread_cond_broadcast(&b);  // signal all threads that
// are waiting on cond. var.
```

If more than one thread is blocked on a condition variable, the scheduling policy will determine the order in which threads are unblocked [19].

We will use the conditional variables in native thread model to give the JikesRVM same functionality of thread cooperation as in the green thread model.

5.6.3 Mutual Exclusion Implementation – monitorEnter/monitorExit

In order to implement the mutual exclusion capability of monitors in the native pthread model, JikesRVM associates a lock with each object. A lock ensures that only one thread can own the global resources at a time. If a thread holds a lock then no other thread can hold a lock on the same resources (or data) at the same time.

In addition, in Java it is allowed that a single thread can lock the same object multiple times by spinning on it. Thus for each object, the JikesRVM maintains a count of the number of times that the object has been locked. Initially, an unlocked object has count of zero. When a thread acquires the lock for the first time, the count will incremented to one. Each time the owner thread acquires a lock on the same object, as only the owner of lock is allowed to lock it again, the count will be incremented. Reversely, each time the thread releases the lock, the count is decremented by one and when the count reaches zero, the lock is released and is made available to other threads and now other thread can obtain the lock.

In the figure 5.3, a thread in JikesRVM's native model will request a lock when it arrives at the beginning of a monitor region. The monitor region is a piece of code that needs to be executed as one inseparable operation. In other words, it ensures that only one thread is able to execute that monitor region (code) from start to end without other threads concurrently executing the same code. An object reference is associated with each of the monitor regions in Java applications. Therefore, when a thread reaches the first instruction of the monitor region, the thread must obtain a lock on the referenced object. Otherwise, the thread is not allowed to execute the code until it obtains the lock. Once the thread obtained the lock and performs operation in the protected block, also called *critical section,* thread switching will be disabled. When the thread completes operation and leaves the block, it releases the lock on the associated object and enables the thread switching. The following block represents the critical section.

pthread_mutex_lock(pthread_mutex_t *mutex*)*

...

...

pthread_mutex_unlock(pthread_mutex_t *mutex*)*

Monitor Region/Critical Section

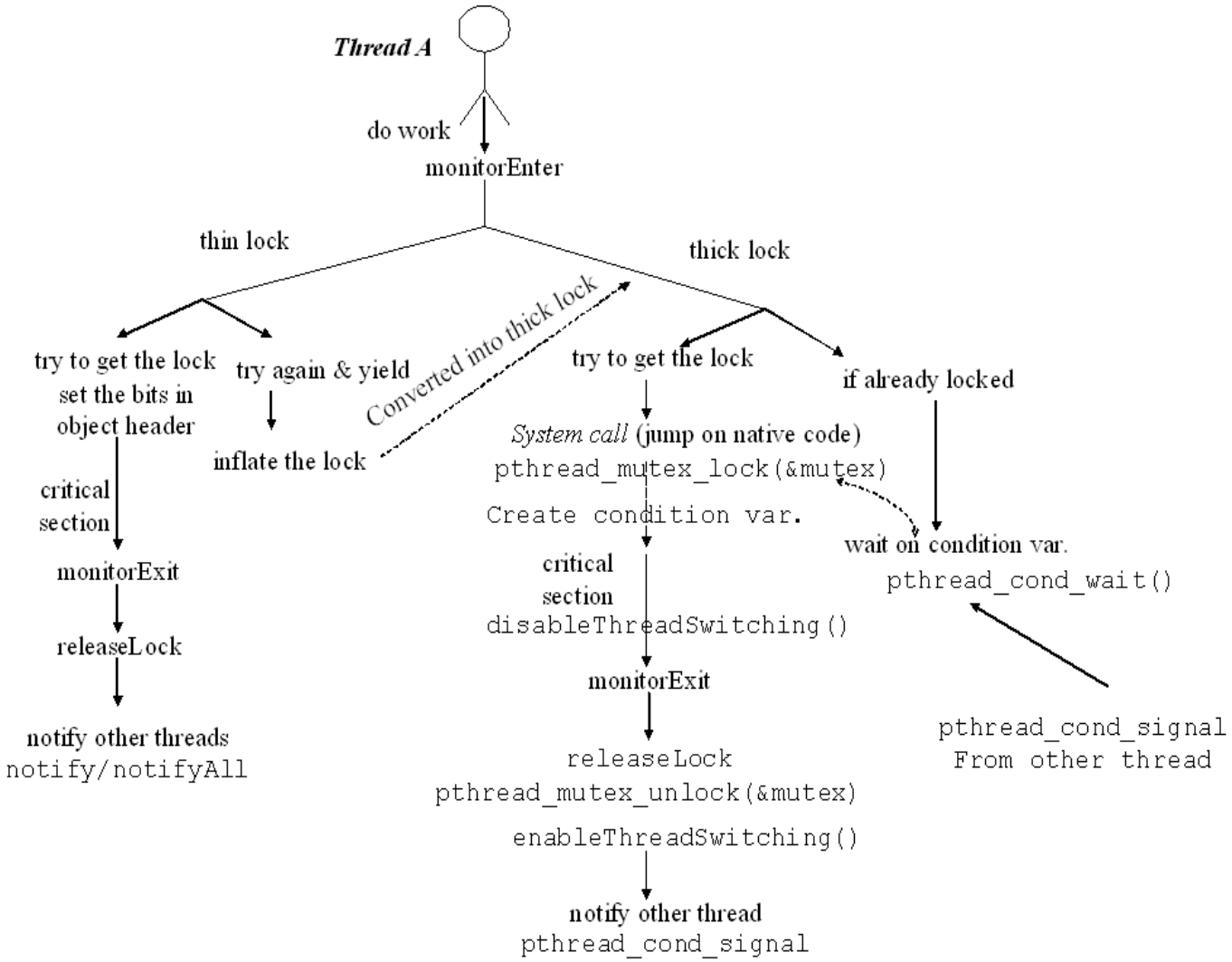


Figure 5.3 Mutual Exclusion

In the above figure 5.3, thread *A* is trying to acquire a lock and implementing the Java primitives' monitorEnter/monitorExit with the help of *pthread mutex* and *conditional* variables. Thread *A* needs to enter into the monitor region in order to perform some operation on the protected shared resource. In order to enter into monitor region, thread A tries to lock the mutex by thin lock first. If there is no contention then the thread grabs the lock and performs the operation (the thin locking is the same as in green thread model as discussed in third chapter: Section3.5). If the thread could not get the lock, it will try again and eventually it will inflate the lock (i.e. transform the thin lock into thick lock). In this situation, if the thread tries to lock the thick lock and it is already locked by some other threads, it will wait on the conditional variable and will enter into the blocked state. After getting signaled from the owing thread it will lock the mutex by calling `pthread_mutex_lock` function of pthread library. Once it will hold the mutex, it can enter in the critical section and can perform operation on protected global data. After finishing its task, thread A will exit from the monitor and release the lock. Furthermore, it will notify the other waiting threads for this lock by sending signals via `pthread_cond_signal` method of pthread library.

5.6.4 Cooperation Implementation

As indicated in below figure 5.4, there are two threads (*thread A and B*), working together in a cooperative manner. Thread *A* wants the protected global data (int a) in some state (e.g. value of *a* =9) in order to process its task. Thus, *A* locks the *global data a* and checks the value of it, if condition does not satisfy then it suspends execution and waits on the condition variable for the signal from other *thread B,* by calling `pthread_cond_wait()`routine, until the value of *a* comes in desired state. This method will be used while *mutex* is locked, and this will also allow the thread to free the mutex automatically while it waits for some event.

As in figure, thread *B* modifies global data (add 7 to it) and brings in to desired state for thread *A* (a=9), thread *B* will signal *A*, which is waiting on the condition variable, via `pthread_cond_signal()` routine and will unlock the mutex. After receiving the

notification, *thread A* will wake up and mutex will be automatically locked for use by the thread A. Now thread A can process its operation on the shared global data and will explicitly unlock this shared data after use in favor of other thread's execution. Later, it can destroy the condition variable by calling the pthread_cond_destroy() method of pthread library. `pthread_cond_signal()` routine signals only one thread. In addition, if there are several threads waiting on a condition variable then we can use `pthread_cond_broadcast()` method to notify them. `pthread_cond_broadcast()` will implement *notifyAll* method of green model in JikesRVM.

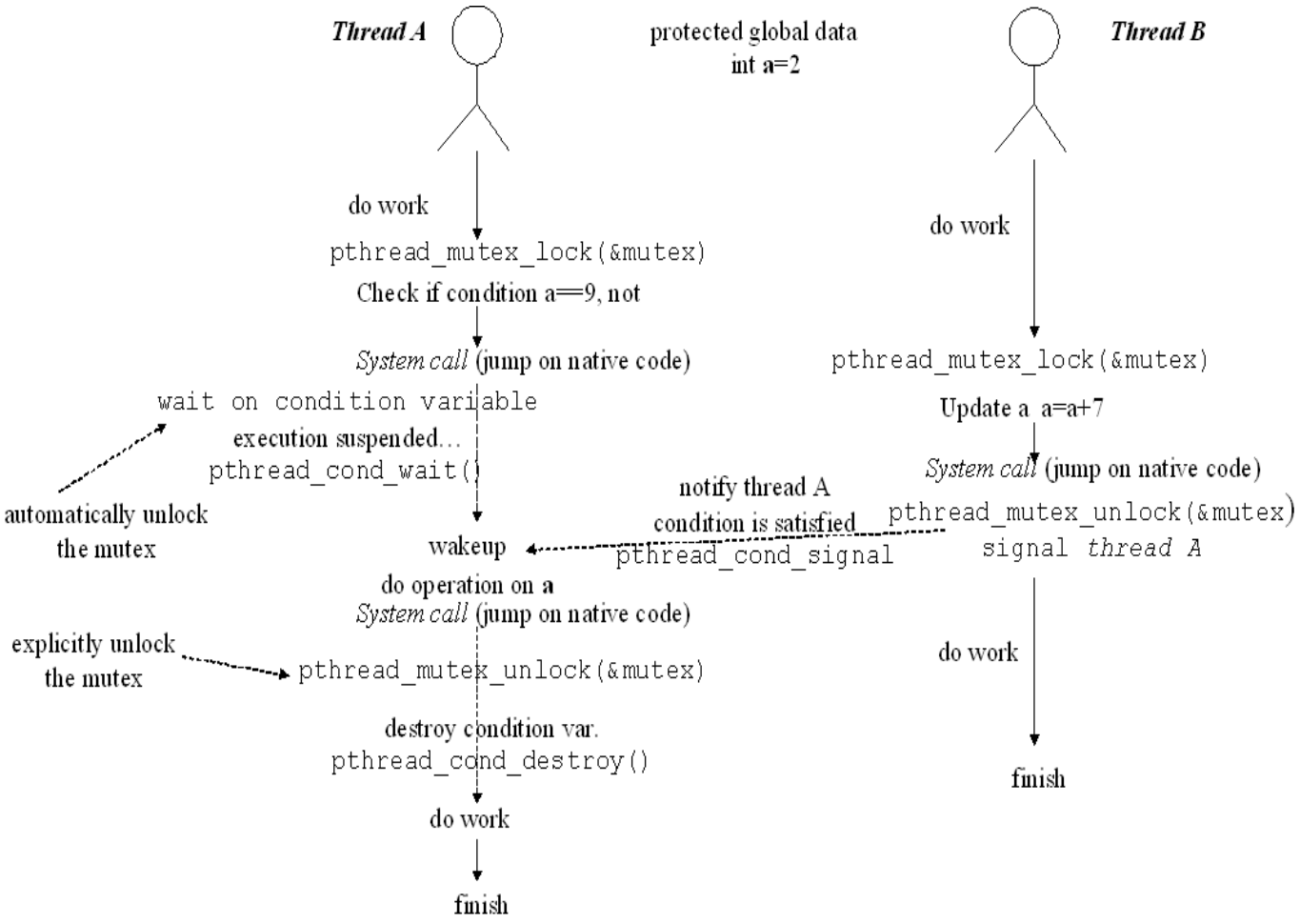


Figure 5.4 Implementation of Wait/notify semantics using Pthread functions

5.7 Thread Interruption - Cancellation and Termination

The thread termination is associated with several issues and stopping a thread safely, quickly, and reliably is not always easy. So it's better to use a cooperative mechanism (by which task and the code requesting cancellation follow an agreed way to stop the thread) that lets one thread ask another to stop what it is doing because there is no safe way to pre-emptively stop a thread [20].

Cancellation a thread means, a thread sends request to terminate the execution of another thread before it has finished. There are a number of reasons why we might want to cancel an activity such as we can click on the 'cancel' or 'stop' button in GUI application (e.g. stop button in Internet explorer). The thread processes the request based on its state. It may act immediately and terminate the thread, may act on the request when it reaches the *cancellation point* (discussed below) or may ignore it.

Cancellation point

In some situations, a thread can be in a state where it can not handle the cancellation requests immediately such as holding a lock; in such cases thread defers requests until the cancellation point. There could be many reasons for cancellation point such as when a thread is in suspended or waiting state. Moreover, some *system calls* that cause the thread to block such as read(), wait(), select() etc. are also cancellation point.

We have two approached to terminate a thread:

Asynchronous cancellation - Asynchronous cancellation terminates the target thread immediately

Deferred cancellation – it allows the target thread to periodically check if it should be cancelled.

We will use the latter approach in the native model, as the *Thread.stop* method is deprecated from Java because it is unsafe as it leaves the shared resources in inconsistent state.

In native model, we will provide the capability to terminate threads cleanly as pthread library has the capability for cancelling a thread safely. In order to terminate a thread, we will use the `pthread_cancel` method of pthread library. This method takes the thread id as parameter and then sends the cancellation request to that thread. The pthread_self() function returns the thread ID of the calling thread. Following are the statement, which we will use to terminate the thread:

```
p_thread thread_id;
thread_id= pthread_self();
pthread_cancel(thread_id); // thread_id is the id of running thread
```

When the user's Java thread invokes interrupt method, it would be implemented as pthread_cancel() in native model. First, it will creates a cancellation point in the calling thread with pthread_testcancel() function. When the thread ensures the cancellation point then it calls pthread_cancel() function to terminate the thread safely. Figure 5.5 shows the thread cancellation operation in native model.

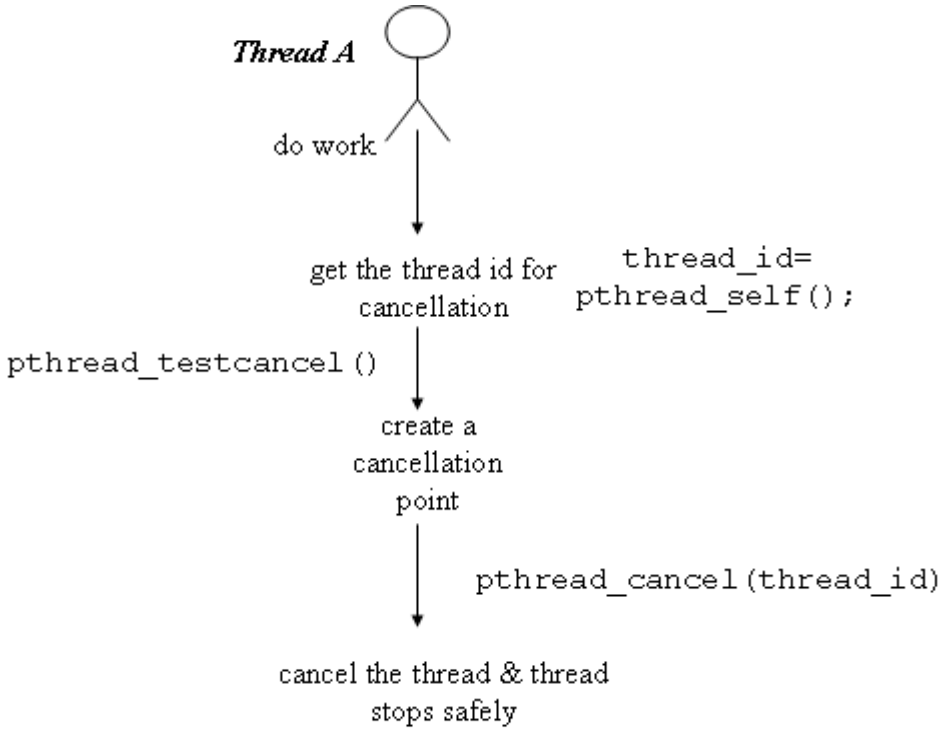


Figure 5.5 Thread Cancellations

5.8 Adding System Calls in JikesRVM

In order to use the Linux POSIX library methods, we will use the *system calls* mechanism in the native threading model. In particular, to add a system call in JikesRVM, we will have to make changes in three classes. We are mentioning these changes with an example of adding a system call for locking a mutex. These classes are following

1. Add an abstract method to *VM_SysCall* class; this class supports for low-level invocation of C library with static addresses. For example, all classes must have the following signature:

```
public abstract Address sysMutexLock(Address mutex);
```

2. Add the function in sys.C which actually does the work; this class provides support services from operating system required by Java classes.

```
Address sysMutexLock (pthread_mutex_t *mutex)
{
.....
pthread_mutex_lock(mutex);
.....
}
```

3. Add a field with the name of function in VM_BootRecord; in this class there must be matching field name (methodnameIP) for each method declared in the VM_SysCall. For example,

```
public Address sysMutexLockIP;
```

5.9 Comparison between Native Model and Green Model

1. The native Pthread model is transparent because of tight coupling between the user threads and the kernel threads. Native thread model has ability to take advantage of multiprocessor environments.

2. The native model can have advantage over green model if the scheduled threads are CPU intensive that means most of the time threads use CPU cycle time and rarely go into waiting state. For example, applications like complex numerical calculations, in such cases, threads will not go into wait state and run until die or finish on their designated kernel threads. Furthermore, this can ensure the maximum execution speed for that thread and also prevents the performance cost caused by cache invalidation due to thread switching. We will evaluate these performance issues in further implementation of native model.

3. In m:n green thread model, if one thread makes a blocking system call then other threads block, but this is not the case in native model as each thread is running on different processors. In addition, to achieve high-performance on a symmetric multiprocessor (SMP) we need one thread per processor as no CPU time is wasted in context switching.

4. Native thread model supports relatively simpler for libraries than the green thread model as it uses OS's thread scheduling and virtual machine does not have to bother with thread scheduling.

5. Native thread model is pre-emptive and JikesRVM's green model is not fully preemptive. Native threads can switch between threads pre-emptively, they can switch control at anytime whereas green threads switches only when control is given explicitly by Thread.yield and Object.wait()

6. Green thread model is platform independent and whereas native model is platform-specific as it uses capabilities of underlying operating system's scheduler.

7. In the native thread model, garbage collector can take more time to stop and restart the threads because each thread will be dealt as a process* in this model, whereas JikesRVM's green model provides a quick garbage collection (as discussed in section 3.2.3). We can evaluate this performance issue after the implementation of pthread model.

8. M:N green thread model does not use priority mechanism and threads are scheduled by counters and time outs. In contrast, native threads are scheduled by underlying OS's scheduler and scheduler uses the priority mechanism in thread scheduling.

9. In green thread model, threads are created in user-level space so they use less kernel resources compare to native thread model where each Java thread involves a separate LWP creation and it require additional kernel resources such as LWP has its own memory space, file-descriptor and runtime environment.

5.10 Summary

In this chapter, we described a means for implementing the native model in JikesRVM by using the POSIX pthread library. We described some major portion of native threading such as how we can bind Java threads to kernel threads, synchronization: mutual exclusion, cooperation and implementation of yield method.

** processes are heavier than threads.*

In addition, we discussed the clean approach for thread cancellation and termination and how we can stop and resume the threads in thread scheduling.

We also discussed the fundamental details of contention scope of threads (process and global) and also stated some specific details about the pthread attributes such as priority, scope and policy. At the end of the chapter, we gave some distinctions between native thread and green thread models. Furthermore, we also specified how we can add system calls in the JikesRVM as we are intending to use system calls for passing the flow from Java classes to pthread library.

CHAPTER 6

6.1 Conclusions

In this dissertation we have shown how the direct mapping of Java threads to Operating System's thread in one-to-one fashion and passing the control to operating system's scheduler in order to improve the performance of the JikesRVM's threading model. Furthermore, by introducing the native model, JikesRVM's threading model can exploit the SMP platform efficiently with the cooperation of underlying operating system.

This dissertation explained the main features and key information of Jikes Research Virtual Machine; particularly bootstrapping, object model, magic mechanism and other main subsystems of JikesRVM including runtime core services, memory management, garbage collection, compilers. Then, it provided substantial information about the existing threading model followed by a number of issues that we experienced. Specifically, the third chapter described how threads were scheduled by m:n green thread scheduling and also some details about thread synchronization, JikesRVM's thread queues and states, and thread switching. It also described load balancing among virtual processors.

Furthermore, this research work introduced a new native threading model by refactorization of existing thread model into two threading models (green and native).

The fifth chapter has shown the design and implementation indication for new native threading model by using POSIX pthread library. Later, we can evaluate the performance improvement of this model over JikesRVM's existing green thread model, particularly when the users run the CPU intensive applications.

In essence, our refactorization gives the flexibility to users for choosing either of thread models according to the nature of their application. Furthermore, user applications can obtain the underlying operating system's support for fast execution in SMP environment.

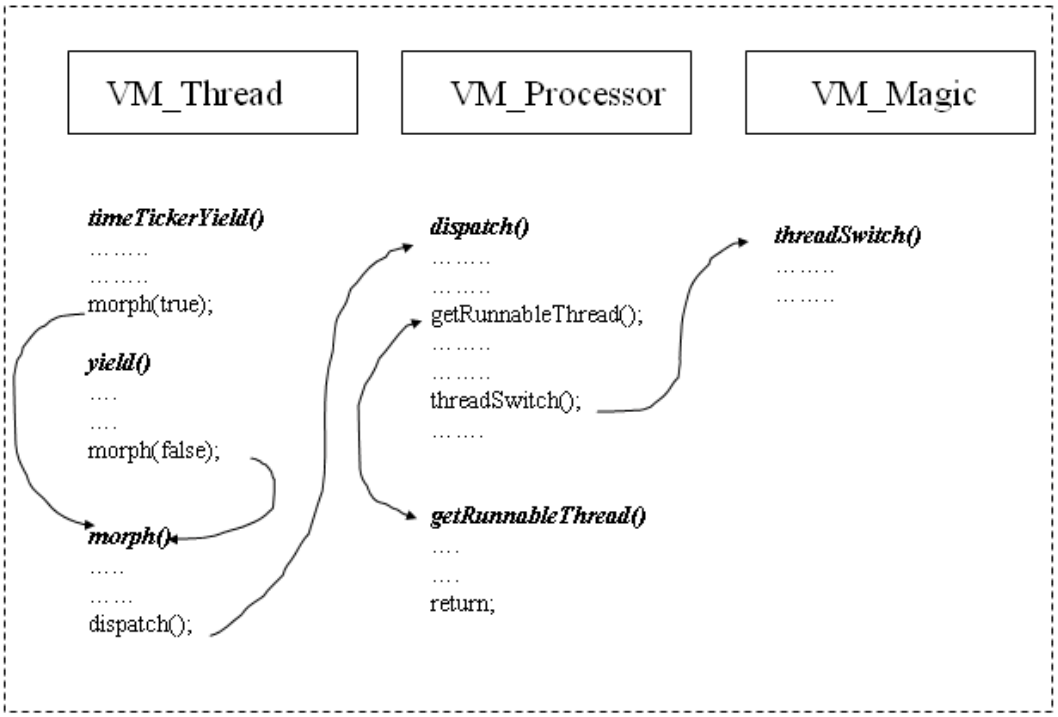
6.2 Future work

As we contributed the source code for this refactorization into the JikesRVM's new release, there is a substantial amount of work remaining in this direction for future. First and foremost work is to bring the native pthread design model into the implementation. After the implementation of this native thread model for Linux operating system, we also intend to implement the native model for windows operating system with support of window native thread library (win32 thread library) so that windows users can also benefit from JikesRVM's fast threading models.

Furthermore, efforts are required to introduce new locking algorithm called "Lock Reservation" [21] in the JikesRVM's threading models. In this strategy, we will evaluate the ways to reserve the lock for threads. The advantage of the lock reservation is to reduce the cost of subsequent lock operations by the thread because when lock reservation is made by a thread, the runtime system will allow the thread to acquire the lock with a few instructions involving no atomic operation [21].

7. APPENDICES

Appendix A



Thread Switching in JikesRVM at code level

Appendix B

build.xml

```
   <condition property="thread.filter" value="-DRVM_WITH_GREENTHREAD=1">
     <equals arg1="${thread.model}" arg2="greenthread"/>
   </condition>
   <condition property="thread.filter" value="-DRVM_WITH_PTHREAD=1">
     <equals arg1="${thread.model}" arg2="pthread"/>
   </condition>


<condition property="pp_RVM_WITH_GREENTHREAD" value="true" else ="false">
     <equals arg1="${thread.model}" arg2="greenthread"/>
   </condition>
   <condition property="pp_RVM_WITH_PTHREAD" value="true" else ="false">
     <equals arg1="${thread.model}" arg2="pthread"/>
   </condition>


<filter token="_RVM_WITH_GREENTHREAD_"
value="${pp_RVM_WITH_GREENTHREAD}"/>
       <filter token="_RVM_WITH_PTHREAD_"
value="${pp_RVM_WITH_PTHREAD}"/>
```

Running the RVM - command line

// for green thread model

[root@cspool125 jikesrvm]# rvm -vmt HelloWorld

// for pthread model

[root@cspool125 jikesrvm]# rvm -pt HelloWorld

8. BIBLIOGRAPHY

1. Bil Lewis & Daniel J. Berg (1998) *Multithreaded Programming with PTHREADS,* 2nd ed., Prentice Hall PTR

2. Bowen Alpern, John J. Barton, Ton Ngo, Janice C. Shepherd, Mark Mergen & Derek Lieber, et al. (1999) Implementing Jalapeno in Java. IBM Systems Journal 2000, VOL 39, p. 211-238

3. Mezini, M. Walkthrough of a Java VM: JikesRVM, Software Technology Group, Technische Universität Darmstadt

4. JikesRVM, 2007, *Source code VM_Statics.java*, Revision 13393, viewed 20 September 2007, <http://jikesrvm.svn.sourceforge.net/viewvc/jikesrvm/rvmroot/trunk/rvm/src/org/jikesrvm/runtime/VM_Statics.java?revision=13393&view=markup>

5. *The Jikes Research Virtual Machine (RVM)* 2000, Introductory, Independently developed as part of the Jalapeno research project at Thomas J. Watson Research Center, IBM Corporation, viewed 22 September 2007, <http://www.ibm.com/developerworks/java/library/j-jalapeno/#author>

6. *Memory Manager Toolkit (MMTk),* Jikes Research Virtual Machine, viewed 10 September 2007, <http://jikesrvm.org/MMTk >

7. *Thread Management,* Jikes Research Virtual Machine, viewed 01 September 2007, < http://jikesrvm.org/Thread+Management>

8. Hind M. & Attanasio D, Jalapeno's support for Memory Management, International Conference on Parallel Architectures and Compilation Techniques, 2001, p. 5-8

9. Stephen J. Fink & Feng Qian, Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement (2003). In Proceedings of the International Symposium on Code Generation and Optimization (CGO'03), San Francisco, California, IEEE Computer Society Washington, DC, USA, 2003, p. 241-252

10. M. L. Scott & W. N. Scherer (2001). Scalable QueueBased Spin Locks with Timeout. ACM SIG{\-}PLAN Notices, Volume 36, Issue 7, p. 44 - 52

11. David F. Bacon, et. al. Thin locks: featherweight synchronization for Java, In Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, Montreal, Quebec, Canada, ACM Press New York, NY, USA, 1998, p. 258-268

12. JikesRVM, 2007, *Source code VM_ThinLock.java*, Revision 13632, viewed 28 August 2007, <http://jikesrvm.svn.sourceforge.net/viewvc/jikesrvm/rvmroot/trunk/rvm/src/org/jikesrvm/scheduler/VM_ThinLock.java?revision=13401&view=markup>

13. JikesRVM, 2007, *Source code VM_Lock.java*, Revision 13699, viewed 08 September 2007, <http://jikesrvm.svn.sourceforge.net/viewvc/jikesrvm/rvmroot/trunk/rvm/src/org/jikesrvm/scheduler/VM_Lock.java?revision=13699&view=markup>

14. *Uninterruptible Code,* Jikes Research Virtual Machine, viewed 22 September 2007, < http://www.jikesrvm.org/Uninterruptible+Code >

15. E. Gamma, R. Helm & R. Johnson & J. Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Systems*, p. 107-116 Addison-Wesley.

16. *MxN Threads on HP-UX 11i: Consequences and Implications,* Hewlett-Packard site, viewed 19 September 2007, <http://h21007.www2.hp.com/portal/download/files/unprot/hpux/MXN.pdf>

17. Amit, G., Caspi, Y., Vitale, R., Pinhas, A.T., et al. Scalability of Multimedia Applications on Next-Generation Processors, Multimedia and Expo, 2006 IEEE International Conference on, 2006, p. 17-20

18. Scott Oaks & Henry Wong (1999), *Java Threads,* 2nd ed. O'Reilly & Associates Inc.

19. *Linux man page help,* Signaling Condition Variable, viewed 27 September 2007

20. Brian Goetz, et al (2006), Java Concurrency in Practice, Addison Wesley

21. Kiyokuni Kawachiya, Akira Koseki & Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations, *In Proceedings of the ACM OOPSLA Conference*, Seattle, Washington, USA ACM Press New York, NY, USA, 2002, p. 130-141