CPU Resource Management for the Java Platform

by

James C. Pang
B.Sc., University of Victoria, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

[redacted]

Dr. Eric G. Manning (Department of E.C.E.)

[redacted]

Dr. Mantis H. M. Cheng (Department of Computer Science)

[redacted]

Dr. Kin F. Li, External Examiner (Department of E.C.E.)

Supervisor: Dr. G. C. Shoja

Abstract

It has been well observed in the literature that multimedia applications are soft real-time. Real-time programming, whether hard or soft, is about resource management. An integrated multimedia system must possess capabilities to allocate and police the usage of system resources such as CPU bandwidth so that certain Quality of Service (QoS) parameters can be expressed and guaranteed. Studies of CPU scheduling and bandwidth management abound. However, current-generation, commercially available general purpose computing platforms such as Java still lack the necessary facilities to support QoS guarantees for soft real-time tasks. To remedy this lack, we developed a new thread model for Java that enables CPU resource management, and a system architecture that supports a service fraction-based thread scheduler in the Java multi-threading framework. We have implemented this model and system architecture in a new Java virtual machine, named Q-JVM, based on the reference implementation from Sun MicroSystems. Our analysis of this implementation is also presented. Our test results show that Q-JVM is able to provide CPU resource management and QoS guarantees for soft real-time tasks.

Examiners:

[redacted]

Dr. Gholamali C. Shoja, Supervisor (Department of Computer Science)

[redacted]

Dr. Eric G. Manning (Department of E.C.E.)

[redacted]

Dr. Mantis H. M. Cheng (Department of Computer Science)

[redacted]

Dr. Kin F. Li, External Examiner (Department of E.C.E.)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to take this opportunity to thank my supervisor Dr. Ali Shoja for his continuous support during my M.Sc. program here at the University of Victoria. I am very grateful for his guidance, encouragement, and patience throughout the development of this thesis.

I would also like to thank Dr. Eric Manning, Dr. Mantis Cheng, and other members of my thesis committee for taking time from their busy schedules to offer valuable suggestions on this thesis.

In addition, I would like to thank members of the PANDA group for their companionship during the course of my graduate study. Their insightful comments and suggestions, and the discussions and brain-storming sessions have been a great help in my pursuit of this interesting area.

Moreover, I gratefully acknowledge the National Science and Engineering Research Council of Canada and SONY Corporation for their financial support.

Finally, I would like to express my sincere gratitude for my wife and my parents. Without their selfless support, this would not have been possible.

To Watson

1 Introduction

The many-fold increase in raw processing power of microprocessors and in network bandwidth over the last decade has made possible a wide variety of new applications which are multimedia in nature. These applications handle data that represent digital continuous media (CM), such as digital audio and video data, using relatively inexpensive and commercially available computing hardware. Furthermore, processing of digital continuous media can now be integrated into general purpose computing platforms, instead of using special-purpose hardware.

1.1 Integrated Digital Continuous Media Processing

First, let's consider a home entertainment unit as an example. Traditionally, home entertainment units consist of separate audio/visual equipments that are interconnected via special purpose cables. Some of these, such as a compact disk player or a DVD player, are digital. However, each piece is a self-contained, dedicated unit with application-specific hardware and software/firmware. With current technology, a home computer can already handle digital audio and video, e.g. CD quality audio and MPEG video[10] from DVD, in real time without the assistance of any special purpose hardware. In the near future, one may expect integrated home information/entertainment appliances that are capable of handling digital audio and video processing from either broadcast or pre-recorded sources, as well as fulfilling normal home computing requirements, such as word processing, home finance, web browsing, and even Internet-based video conferencing. In such an

environment, the processing of continuous media is integrated with conventional computing on a single general purpose computing platform.

Integrated continuous media processing presents unique challenges to the underlying supporting environment: It imposes real-time requirements on the host operating system and its subsystems, as continuous media data must be presented continuously in time at a predetermined rate in order to correctly represent the information being carried. However, programs that process CM data are often classified as being *soft real-time*; they only require the operating system to statistically guarantee quality of service (QoS) parameters such as delay and throughput. They often have deadlines for various tasks; fortunately, missing a particular deadline is not fatal, as long as it is not missed by too much, and most other deadlines are not missed.

At the same time, these programs are often resource intensive: Decoding a broadcast quality MPEG-2 stream, and rendering it to a bitmapped display at 30 frames per second could easily overwhelm a moderate personal workstation without special system software or hardware support. Hence, the supporting platform must be able to partition, monitor, and police the usage of system resources, so that all current application programs can make adequate progress, even in the presence of heavy system load.

1.2 The Java Platform

Although multimedia computing is supported to varying degrees on a number of current generation operating environments, the Java platform [25] has the most desirable characteristics.

First of all, the Java language environment[2][12] is a small, easily-understood system. Its syntax borrows heavily from both C and C++, so that experienced C and C++ programmers can migrate easily to this new environment, and become productive quickly. Furthermore, Java is a modern language; it does not suffer from many of the pitfalls of C and C++. The most notable example is that Java does not support the manipulation of pointer types; as a result, it eliminates the source of the most common programming faults - memory access violations.

Java (in the text hereafter, the word *Java* refers to the Java language environment, or

the Java platform [12][25][29]) is also desirable because it is object oriented, has built-in support for multithreaded programming, and has automatic memory management. The Java programming language is designed to be object oriented from the ground up. It supports encapsulation, polymorphism, inheritance, and dynamic binding; however, it does not support multiple inheritance in order to avoid many known pitfalls. Instead, it supports sub-typing via Java Interface[2]. This simple and effective object oriented programming framework makes Java ideal for developing complex application software in network based environments.

Java supports multithreading at language level, with built-in synchronization primitives. Multithreading provides the means to build applications with many concurrent threads of activity, thus allowing a high degree of interactivity. This is essential for multimedia applications, and provides a natural way of constructing software that processes continuous media data.

In addition, Java provides automatic garbage collection. Objects are created using the "new" operator, and are freed automatically by the run-time system once they are no longer referenced. This not only frees programmers from keeping track of memory usage manually, thus improving their productivity, but also eliminates another major source of common programming errors - memory leaks.

Furthermore, Java provides a comprehensive set of APIs for multimedia processing. The Java 2D API, 3D API, and Advanced Imaging API provide sophisticated processing capabilities for both 2D images, and three dimensional geometry constructs. The Java Sound API, Speech API, and Media Framework API [21] provide a simple, unified architecture, messaging protocol, and programming interface for continuous media processing, including media players, media capture, and conferencing.

Most importantly, Java is portable, dynamic, and robust. The interpreted nature of Java enables Java applications to run on a variety of platforms without modification, as long as the target platform supports a Java Virtual Machine. This also enables the platform to be very dynamic: Linking of modules and classes is incremental and on demand; it is even possible to link previously unknown classes and code modules across a network. Security features are designed into the language and run-time system: un-trusted code is

subject to verification, and is executed in a sandbox [12], thus denying it direct access to critical system resources, and avoiding the risk of invasion.

Last but not least, Java is lightweight, and embeddable. It was originally designed for embedded applications [2]. With the release of version 1.0 of the *EmbeddedJava* specification, it is again being positioned to target embedded devices. Examples of such devices include pagers, telephone equipment, printers, and digital TV set-top boxes. The embedded Java virtual machine has a very small memory footprint: It requires only 256KB of ROM [25]. The class libraries are divided into the core API and the extension APIs. Only the core API is guaranteed to be available on all platforms. Devices can have only those Java classes that support their applications, along with the embedded Java virtual machine and core API, built as one executable image in ROM. This helps reduce the memory footprint further, and makes Java ideal for devices whose cost constraint limits the amount of memory available.

1.2.1 Supporting Soft Real-Time Tasks on Java

Unfortunately, Java presently does not have the facility to support soft real-time processing. Real-time programming is a matter of managing resources, most noticeably the CPU resource. A real-time programmer must start with resource control, before building an application around that layer. However, Java does not provide any mechanism which can be used to monitor, manage, or police the usage of the CPU resource. Furthermore, its thread scheduling facility uses a static priority-based model. A scheduler using such a model has been implemented in UNIX System V Release 4. It is claimed to support real-time tasks; however, it has been shown to be largely ineffective for multimedia applications [32]. Thus, in order to realize the full potential of Java as the platform of choice for multimedia computing, it must be extended with resource management facilities suitable for soft real-time processing.

Furthermore, Java takes a "hands off" approach to thread scheduling: the scheduling algorithm is not fully specified in the specifications[2][27], and indeed, was left to the virtual machine implementor and thus became platform dependent. This leads to inconsistencies across platforms, and further difficulty in providing predictable quality of service for soft real-time applications.

1.3 Objectives

Our objective is to enhance the Java platform so that it can provide resource management and quality of service guarantees for soft real-time tasks, in order to better support integrated continuous media processing. To achieve this objective, we have focused on the following two areas:

1. replacing the static priority-based thread scheduler with one that is designed to support soft real-time tasks; and
2. introducing a CPU resource management facility for the Java platform.

In particular, we studied the thread scheduling facility as implemented in Sun's reference implementation of Java Virtual Machine version 1.1, various threads mapping strategies, and resource management algorithms such as the Move-To-Rear List Scheduling (MTR-LS) algorithm [6] that are designed for integrated multimedia processing. We then proposed an extension to the Java thread model and the underlying threads package for facilitating QoS specification, policy management, and guarantees for the CPU resource. We have produced a Java Virtual Machine, named Q-JVM, on the Solaris operating system which employs an enhanced version of the MTR-LS algorithm in its thread scheduler. It is backward-compatible with the reference implementation of JVM version 1.1.5 from Sun. We have further developed a set of Java application programming interfaces to provide access to the advanced capabilities of Q-JVM.

1.4 The Rest of this Thesis

Chapter 2 surveys resource management schemes, including the Move-To-Rear List Scheduling algorithm, and real-time extensions to the Java platform. Chapter 3 investigates the multithreading support in Java, and various threads mapping strategies. Chapter 4 details our strategy for enhancing the Java platform. Chapter 5 provides some insight into our implementation of Q-JVM based on Sun's JVM version 1.1.5, and our analysis of this implementation. The Java application programming interface (API) for QoS and CPU resource management is presented in Chapter 6. Chapter 7 covers our experimentation on the new JVM, and gives some performance measurement results. Chapter 8 concludes this thesis.

2 Previous Work

It is well understood that continuous media processing is soft real-time [1][13][14][28]: it imposes real-time constraints and deadlines, as the CM data must be presented continuously in time at a predetermined rate in order to properly convey their meaning. However, missing a particular deadline is not fatal, as long as it is not missed by too much, and as long as most other deadlines are not missed.

Real-time programming, whether hard or soft real-time, is a matter of managing resources, most noticeably the CPU resource. The operating environment must be able to provision resources in a deterministic fashion, so that quality of service parameters such as delay and throughput can be guaranteed. For soft real-time applications, this requirement may be relaxed a little: the operating environment may only need to statistically guarantee certain QoS parameters such as delay and throughput.

There has been much research in resource management for soft real-time applications. Some authors have studied existing systems that claim to support real-time multimedia applications, and found that static priority-based scheduling is not sufficient for multimedia soft real-time applications [26][28][32]. Others like [14] have borrowed from link scheduling, or proposed new algorithms [6] for managing the CPU resource. At the same time, researchers have realized the potential of the Java platform for embedded real-time processing, and proposed extensions to the base platform [36].

2.1 Static Priority Based Scheduling

UNIX System V Release 4 (SVR4) incorporates a static priority based process scheduler. By incorporating this scheduler, SVR4 claims to provide system support for real-time and multimedia applications. An extensive quantitative analysis of this process scheduler was conducted by Nieh et. al. [32]. Their quantitative measurements of the performance of a mix of real-time, batch, and interactive applications on that system demonstrated that this process scheduler was largely ineffective. It could even produce system lockup. Their conclusion was that the existence of a static priority based real-time process scheduler in no way allows a user to deal with the problem of CPU resource contention presented by multimedia applications.

Hard real-time scheduling algorithms, such as Earliest Deadline first (EDF) and the Rate Monotonic Algorithm (RMA), are not suitable for integrated continuous media processing as well [26][28]. These algorithms are designed for applications that have strict deadlines, and whose deadlines must be met or the application will not function correctly. For this class of application, the operating system must be able to deterministically guarantee delay and throughput that may be experienced by various tasks. However, these algorithms satisfy the real-time requirement at the expense of efficiency. For example, RMA can only achieve a maximum of 60% CPU utilization if all real-time requirements are to be met. This is undesirable in an integrated environment where all applications, real-time or not, must make steady progress and efficiency in CPU utilization is a requirement. Algorithms like EDF are more efficient. However, they often require the computational needs of all tasks to be known before a schedule can be prepared. This is undesirable in dynamic systems that process integrated CM data, as such information is not usually available. Even for tasks that process video streams with fixed frame rates, the computational requirement is variable from frame to frame. If reservation must be made for the peak rate, then CPU utilization will suffer as in the earlier case.

Soft real-time applications do have deadlines; however, their deadlines are more flexible. They only require the operating environment to statistically guarantee the QoS parameters. It is evident that static priority based and hard real-time solutions are unsuitable for soft real-time tasks. Thus, one must investigate other solutions.

2.2 Resource Management Based on Fair Queueing

Fair queuing algorithms in general, were originally developed for scheduling packets for Integrated Service Packet Switching Network[40]; later, they were adopted for CPU scheduling. Much research was carried out in this field [7][8][11][16][17].

Weighted Fair Queuing (WFQ) [8] is the earliest known fair scheduling algorithm. It was designed to schedule network packets onto a link by emulating a hypothetical weighted round robin server in which the service received by each flow in a round is infinitesimal and proportional to the weight of the flow. It assigns each packet a start tag, which is the time it arrives at the link server. It then calculates a finish tag using a real-time simulation of the weighted round robin server, and schedules the packets in increasing order of their finish tags. Because of this real-time simulation, WFQ is very computationally intensive [11]. Moreover, it does not provide fairness when the available bandwidth fluctuates over time due to, for example, sporadic interrupt processing [16]. At the same time, it requires the length of the quanta, i.e. the size of the packets, to be known before hand. These characteristics makes WFQ unsuitable for CPU scheduling as interrupt processing is a necessity and computational requirements are seldom available before scheduling.

Fair Queuing based on Start time (FQS) is proposed as an improvement to WFQ: it calculates the tags exactly as does WFQ; however, it schedules packets in the increasing order of the start tags instead of the finish tags [17]. In other words, it does not require the length of quanta to be known before scheduling is to take place. This makes it suitable for CPU scheduling. However, it suffers from the other two drawbacks of WFQ, namely it is computationally expensive and it does not provide fairness when available bandwidth fluctuates.

Self Clocked Fair Queuing (SCFQ) [7][11] was designed to reduce the computational complexity of fair queuing algorithms such as WFQ. It also uses start and finish tags, and schedules packets in increasing order of finish tags. However, it approximates the real-time simulation of the weighted round robin server. The approximation is very efficient and yields results that are only a factor of two away from the optimal value [11]. Unfortunately, it achieves efficiency at the expense of a larger maximum scheduling delay

[15]. This may be unacceptable for many applications. Moreover, while scheduling by finish tag is acceptable for packet scheduling, it is unsuitable for CPU scheduling.

2.3 The SFQ algorithm

The Start-time Fair Queuing (SFQ) algorithm [14][16] is a new fair queueing scheduling scheme that is designed to address the two major problems of traditional fair queueing algorithms, namely their high complexity and inability to guarantee fairness when the available bandwidth of the server fluctuates. Like other fair queueing algorithms, SFQ was originally designed for link packet scheduling. Later, it was adopted for CPU scheduling in a hierarchical resource management framework [14] targeted to serve multimedia applications with soft real-time requirements.

Similar to other fair queueing algorithms, SFQ assigns a start tag and a finish tag to each packet. However, it schedules packets in the increasing order of their start tags. A packet's start tag is the larger of the time this packet is requested to be sent and the finish tag of the last packet in the same flow. The finish tag of a packet is calculated using a very straightforward formula which takes into account the start tag of this packet and the length of the packet. This calculation of the finish tag is very inexpensive. It brings the complexity of SFQ to $O(log(n))$ [16], where n is the number of flows. Analysis of SFQ in [16] has also shown that it provides fairness, throughput, and delay guarantees regardless of fluctuations in available bandwidth. In fact, the authors of [16] claim that no other known algorithm achieves better fairness than SFQ.

Later in [14], SFQ was adopted for scheduling threads in a hierarchical CPU scheduling framework. The adaptation is straightforward. Each thread is associated with a start tag; it is adjusted each time the thread requests CPU. Finish tags are calculated as in the case for link scheduling. However, the length of packets are substituted with the length of quanta, i.e. the time a thread runs before it blocks or is preempted.

The scheduling framework proposed in [14] manages the CPU as a resource, and allocates it in a predictable manner. As its link scheduling counterpart does for the network flows, a SFQ-based CPU scheduler provides soft real-time guarantees for threads in terms of fairness, throughput and response time. It provides these quality of service

guarantees even when the available server bandwidth fluctuates [14]. This is particularly important for CPU scheduling as the available bandwidth to application programs is inevitably variable due to interrupt processing and other high priority system tasks such as paging.

Unfortunately, although SFQ is recognized to be superior to classical fair queueing algorithms for CPU scheduling, its delay bound increases linearly with the number of threads in the system [38]. Thus, it is not ideal for a general-purpose environment.

2.4 The MTR-LS algorithm

The Move-To-Rear List Scheduling algorithm (MTR-LS) is a recent development in resource scheduling algorithms. It is aimed to provide predictable service in a general purpose system with multiple resources including CPU, disk, and the network [6]. Besides the usual quality of service parameters such as fairness, bandwidth partitioning and delay bound, it also supports a new criterion called *cumulative service guarantee.* MTR-LS is able to guarantee that the real service obtained by a process, given its specified service rate (or the reserved service fraction), does not fall behind the ideal service it would have accumulated on a dedicated processor at that service rate by more than a constant amount. MTR-LS is the first scheduling algorithm which articulates cumulative service as a quality of service parameter. Other resource management algorithms such as SFQ are not analyzed with respect to cumulative service guarantee; however, it is not clear whether they support this QoS criterion implicitly.

2.4.1 Key Concepts

Similar to the fair queueing algorithms, MTR-LS manages resources by assigning a weight, called *service fraction,* to each thread. Unlike weights, however, this parameter specifies the amount of the server required in absolute terms. For example, if the server is a 50 MIPS CPU, then assigning a thread a *service fraction* of 0.1 reserves 10% of the CPU for this thread. In other words, this thread would expect to have the use of to a 5 MIPS virtual CPU.

Weights, on the other hand, can determine server reservation only when considered in

relation with the aggregate weight allocated to all threads. For example, assigning a thread a weight of 5 does not mean anything, until we know that there are only three other threads in the system, each of which has a weight of 5. In this case, the server reservation of the first thread is 25% of the CPU. If another thread of weight 5 enters the system at this time, then the server reservation of the first thread is changed to 20%.

Another key concept in MTR-LS is the *virtual service time*. It is the time required to complete a task when the server has to serve more than one task at the same time. For example, let B denote the service rate of a CPU server. A thread T_i with a service fraction reservation α_i would expect a "virtual" server whose service rate is $B \cdot \alpha_i$. If T_i has ω units of work to accomplish, such an idealized server will take $\omega/(B \cdot \alpha_i)$ time units to complete this work. This time is the *virtual service time* for completing ω units of work on this server. Since the real server only runs one thread at a time at its full service rate, the real time it takes for the server to accomplish ω units of work will be less than the virtual service time. The difference is the time when this thread is waiting for the server to become available.

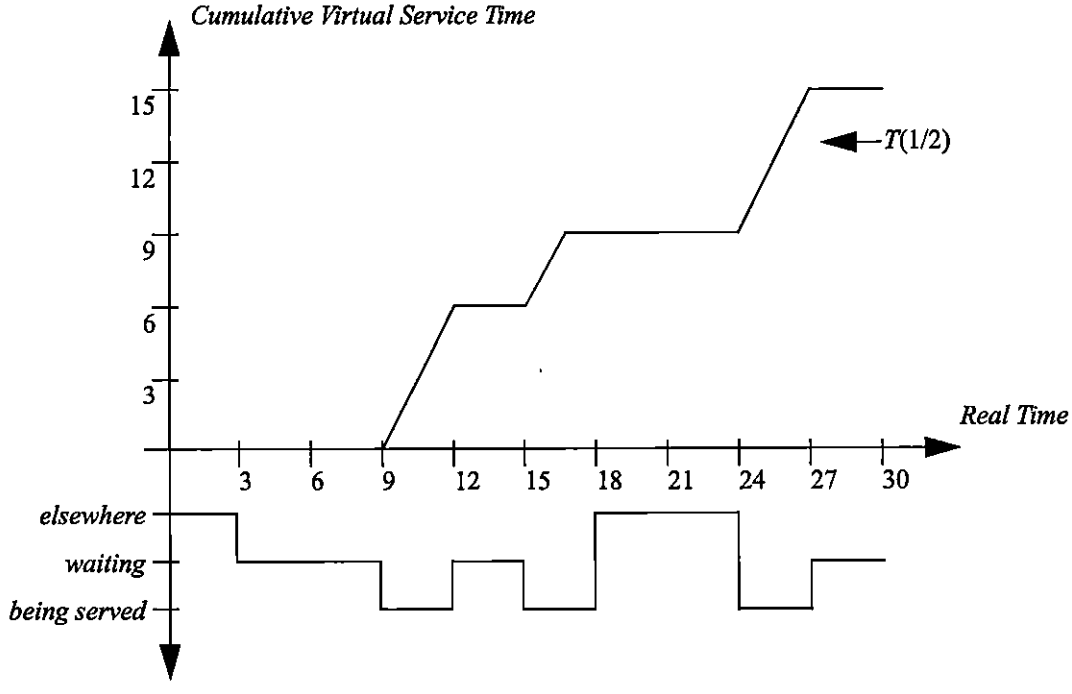


Figure 1: Waiting time, real service time, and virtual service time.

The concept of virtual service time can be illustrated using Figure 1. This graph shows a thread T with a service fraction of 1/2. The lower part of the graph shows the status of T at various times: it is either elsewhere (not requiring service), waiting for the server to become available, or being served on the server. From the graph, we can read the following quantities: from real time unit 0 to 15, T spends 9 time units waiting for the server to become available and only 3 time units to run on the server. However, since T has a service fraction of 1/2, the virtual service time it accumulated during this 3 real time units is 6. This is because it would have taken T 6 time units to complete the same amount of work if it were executed on a server that has only 1/2 of the service rate. Similarly, between real time 21 and 30, T spent 3 time units waiting and another 3 time units being serviced by the server. Nevertheless, it accumulated 6 units of virtual service time.

The most important concept in MTR-LS is the *cumulative service guarantee*. This guarantee means that the real system will keep pace with an ideal execution based on the server reservation. In other words, the real time for a thread to obtain service, including both the waiting time and the real service time, must not be greater than the virtual service time accumulated for this service by more than a constant amount. In the example shown in Figure 1, thread T accumulated 9 units of virtual service time during the first 18 real time units. However, during this period, it spent 9 units of real time waiting for the server and 4.5 units of real time being serviced. Thus, the total time to provide the service is 13.5 time units. The difference between the total real time required to accumulate the service and the accumulated virtual service time is (13.5-9)=4.5. If the scheduling algorithm for this server were to provide cumulative service guarantee, then this difference must be bounded by some constant.

The difference between the cumulative service guarantee and the delay bounds provided by traditional soft real-time scheduling algorithms like SFQ is that the former bounds the service time for the life time of the thread while the latter only bounds the delay each time the server is requested, i.e., the scheduling delay. The scheduling delay may accumulate, resulting the real server falling behind the idealized virtual server indefinitely.

2.4.2 MTR-LS Scheduling Policy

The central data structure of the Move-To-Rear List-Scheduling algorithm is a *service list* L. This list contains all threads that are active in the system. When a thread is created, it is time stamped and added to the list. It remains on the list as long as it is active regardless of whether it is runnable or blocked. It is removed from the list whenever it terminates. The service list is ordered by the time stamps of the active threads; threads with earlier time stamps appear near the beginning of the service list. The MTR-LS policy schedules runnable threads in the order they appear on this list.

Another important element of the MTR-LS algorithm is a system constant T, called *virtual time quantum*. It is the maximum time for the server to service all threads on L exactly once. Associated with each thread t is a quantum *left*. This is the maximum time this thread can run on the server before it is preempted. Initially, the value of *left* is set to $\alpha \cdot T$, where α is the service fraction of t.

A *decision epoch* occurs when the current thread blocks, its current quantum expires, or when another thread becomes runnable. At each decision epoch, the current thread's *left* value is recomputed. It is decremented by the actual amount of service t obtained during the period. If the result is zero, then t is assigned a new time stamp and its position on the service list is adjusted according to this new time stamp. In other words, it is moved to the rear of the service list. Its *left* is reset to $\alpha \cdot T$. The first runnable thread on L is then scheduled to run.

2.4.3 Properties of the MTR-LS Policy

The properties of MTR-LS with respect to complexity, fairness guarantee, delay bound, and cumulative service guarantee have been shown analytically in [6]. We will only list the properties here with some elaboration. Detailed proof can be found in the original paper.

The essence of the MTR-LS algorithm is the maintenance of the service list. Since L can be implemented as a heap, and the complexity of maintaining a heap is $O(ln(n))$ where n is the number of entries in the heap, the authors of [6] conclude that the complexity of MTR-LS is also $O(ln(n))$ when n is the number of active processes in the system.

The MTR-LS scheduling policy provides a fairness guarantee that is dependent upon the system constant T, the virtual time quantum. It does not, however, depend on whether the sum of service fractions of all active processes is less than or equal to 1. The MTR-LS algorithm guarantees that the cumulative virtual service time of two competing processes will not differ by more than $2T$ [6].

Using some worst case scenario analysis, the authors of [6] have also shown that MTR-LS provides a cumulative service guarantee and a delay bound guarantee. However, unlike the fairness guarantee, the total service fractions assigned to all processes in the system must not be greater than 1 in order for these guarantees to be honoured. Furthermore, if arbitrary preemption is not permitted on the server, then the server must reserve some capacity that is greater than the service fraction allocation to any single process. Therefore, admission control is necessary for providing cumulative service guarantee and a delay bound guarantee.

2.5 Other Resource Management Schemes

Lottery scheduling [42] is a randomized fair scheduling algorithm in OS context. Due to its randomized nature, it achieves fairness only over large time intervals. This limitation was later addressed by the Stride scheduling algorithm [41]. The Stride algorithm is a variant of WFQ; thus it suffers similar drawbacks as WFQ, namely requiring prior knowledge of the resource requirement, and high complexity. Later, another proportional share resource allocation algorithm, known as Earliest Eligible Virtual Deadline First, was proposed [38]. It supports a performance bound that is similar to the cumulative service guarantee of MTR-LS; however, it was only analyzed for servers that support arbitrarily small preemption intervals, and its complexity is higher than that of MTR-LS.

Hierarchical resources partitioning schemes have also been proposed. SFQ was presented in [14] in the context of a hierarchical scheduling framework that is able to accommodate heterogeneous schedulers using different algorithms. Another such scheme based on tickets and currencies was proposed in [42]. In this framework, a thread is allocated "tickets" in some currency, and the currency is in turn funded by tickets in some

other currency. This hierarchy of currencies can all be converted back to some base currency, and threads are allocated CPU resource in proportion to the value of their tickets expressed in the base currency, according to a lottery scheme. This scheme is similar to the framework purposed in [14]. However, it does not permit heterogeneous schedulers to be used throughout the hierarchy. In addition, this framework was not able to provide any performance guarantees, and is more computationally complex than MTR-LS.

Several other efforts have investigated scheduling techniques for multimedia computing systems [1][13][22][30][33]. In particular, [13] details a Split kernel-user Level Scheduling (SLS) framework and a Deadline Work-ahead Scheduling (DWS) algorithm designed specifically for processing of digital continuous media. The SMART algorithm proposed in [33] was the answer from Nieh et. al. to the ineffectiveness of the static priority-based scheduler found in UNIX SVR4 and Sun's Solaris operating system in handling real-time multimedia applications. Unfortunately, these algorithms are either computationally complex (in the order of $O(n)$), or complicated to implement. Furthermore, their theoretical properties are not fully known.

In contrast, MTR-LS is simple, and efficient (in the order of $O(ln(n))$); its theoretical properties are fully known; it does not require prior knowledge about resource requirement; it does not even require admission control when only a fairness guarantee is desired. Admission control is only required for providing delay bound and cumulative service guarantees. Moreover, the delay bound and cumulative service guarantees are bounded by some system constant; they are independent of the number of processes in the system. Most importantly, MTR-LS supports servers with arbitrary positive preemption intervals: it honours all guarantees when a portion of the server's capacity is reserved and not allocated to competing processes. This situation most realistically models a CPU server, where arbitrary preemption is often restricted for efficiency reasons. We therefore chose MTR-LS as the starting point for providing soft real-time scheduling for Java threads.

2.6 The Utility Model

The Utility model and its associated system architecture, the Padma Architecture, is a

resource management model and an end system architecture, respectively, for adaptive multimedia systems (AMS) with multiple concurrent sessions [24]. In such a system, the quality of individual sessions is dynamically adapted to the available resources and run-time user preferences.

The Utility Model captures the behavior of adaptive systems; it deals with admission control, quality adaptation and integrated resource allocation problems in a unified way. It is based on the concepts of quality profile, quality-resource mapping, session and system utility, and resource constraints. In this model, each session provides a QoS profile which is a set of operating qualities arranged from the lowest acceptable quality to the highest desired quality. Any operating quality may be mapped to the required resources using a quality-resource mapping, and to a session utility using a quality-utility mapping. The system will then find an operating quality for each session such that the overall system utility (such as system revenue) is maximized subject to the system resource constraints and the session resource constraints as required to meet QoS guarantees.

The Padma Architecture is a layered end-system architecture that features integrated and adaptive resource management based on the Utility Model. It supports hierarchical quality and resource management, and the use of Meta-Interfaces [19] to encapsulate and protect the mechanisms of quality adaptation. It is flexible and dynamic, and is an ideal total end-system solution for multimedia computing [24].

However, the Utility model and the Padma Architecture do not address the lower-level resource management problems. It assumes a host system that supports QoS directives, resource management and usage policing. Thus, in order to deploy end systems that fully support multimedia computing, it is imperative to support quality of service in host operating systems. As we have seen earlier, current OS products such as Unix, Windows, MacOS, and BeOS do not provide such support. The work documented in this thesis is an attempt to fulfill this requirement for the CPU resource.

2.7 Real-Time Extensions for Java

Other researchers have also realized the potential of the Java platform, and considered extensions to this environment for real-time computing [34]. The extension proposed by

Nilsen [36] uses a number of techniques such as analysis of worst-case execution time, measurement of representative function invocations, rate monotonic analysis, static cyclic scheduling, and real-time garbage collection with possible hardware assistance. It relies on an off-line analyzer and scheduler to provide information to a real-time executive for deterministic execution. It provides some flexibility by supporting resource negotiation between the real-time executive and the analyzer using a complicated protocol.

These extensions are intended to enhance the Java platform for embedded hard real-time environments [36], as Java was originally targeted [12]. Nilsen recognizes that providing predictable real-time service requires a specially designed Java virtual machine [35] that is not necessarily compatible with the standard environment. His PERC system focuses on the complex analysis of code execution and inter-task blocking. It is not fully compatible with Sun's reference implementation, and requires applications to be modified and specifically optimized for the new platform. It also places much emphasis on real-time garbage collection with possible hardware assistance, building on his considerable expertise in this field. The system is thus expensive to implement, and suited only for very specific applications.

Our purpose, on the other hand, is to provide resource management and QoS guarantees for soft real-time tasks in a general purpose computing environment. We recognize that automatic garbage collection is a major stumbling block for providing real-time service, but prefer to reduce this to a resource management problem. We would build a simple and effective resource management algorithm such as MTR-LS into the base platform to regulate the CPU resource consumption and enable QoS guarantees. Such an extension will not meet hard real-time requirements; however, we believe it is sufficient to provide soft real-time responses.

3 Multithreading Support in Java

As stated in earlier chapters, our objective is to enhance the Java platform so that it can provide resource management and quality of service guarantees for soft real-time tasks. Our strategy is to introduce a compatible Java platform with a QoS based threads model. In particular, we would like to replace the standard static priority-based thread scheduling facility in the JVM with one that supports QoS guarantees. However, before we can propose any extensions to the platform, we must have a firm understanding of how the standard one works. We must understand why multi-threading is important to Java, what are the distinct characteristics of Java threads, how are threads supported on host platforms in general, and how are Java threads supported by Sun's reference implementations of Java virtual machines. This chapter lays the groundwork for Chapter 4, where our new model is discussed in detail.

3.1 Multithreaded Programming

Multithreading is an important programming technique. Historically, threading was first exploited to take advantage of the natural parallelism exhibited in some computationally intensive algorithms. Such algorithms are traditionally implemented using separate processes that communicate using operating system specific mechanisms like signals and shared memory space. Threading makes these algorithms easier to implement, and reduces the complexity of the implementation.

The popularity of threading has increased in recent years with the widespread

adaptation of graphical user interfaces. Threading allows users to perceive better system performance by allowing programmers to delegate normal computation and the processing of user input to different threads, thus making the program more responsive. It does not, however, boost the real performance of these programs.

With the advent of affordable multi-processor workstations, multi-threading become a hot topic. It is the most natural way for applications to take advantage of the multiple processors on a single machine by having more than one units of execution at the same time. Inherently parallel programs will not only benefit from reduced complexity, but also realize increased performance.

Java is a modern general purpose computing platform that takes full advantage of multithreaded programming. Multi-threading is an integral part of the Java language. It is not only used to implement algorithms that lend themselves naturally to multithreading, but also used to support asynchronous behaviours in Java. For example, the Java platform does not have the notion of an alarm, or an asynchronous timer event. A Java programmer would need to set up a dedicated thread to sleep for a specified time interval, and then notify other threads that the timer has expired. In fact, many Java API libraries use threads without any awareness on the part of the application programmers.

3.2 Java threads

The Java platform supports multi-threading at the language level. It presents a unified, platform-neutral threads API to application programmers. It also presents a standard interface for inter-thread synchronization and communication.

Multi-threading at Java language level is supported by the `java.lang.Thread` class and the `java.lang.ThreadGroup` class. The `java.lang.Thread` class facilitates the creation, destruction, state manipulation and querying, synchronization and priority adjustment of threads, and a number of other miscellaneous operations. The `java.lang.ThreadGroup` allows a programmer to manipulate a number of threads using a single command. It also provides the basis that the Java's security mechanism uses to screen the control of threads, and interaction among threads.

3.2.1 Thread Scheduling

As specified in the language specification [2], the scheduling of Java threads is based on their static priority. At the Java language level, there are ten different priorities, denoted by integers 1 to 10. Larger integer value means higher scheduling priority. When a thread is first created, it is assigned the "Normal Priority:" 5. A programmer can change the priority of threads to any valid value at any time after the thread is created. The necessary control and management mechanism for manipulating priority is provided in the `java.lang.Thread` class.

A Java Virtual Machine (JVM) is expected to implement a *preemptive priority-based* scheduler. Under such a scheduler, the currently running thread is always the one having the highest priority among all runnable threads. For example, suppose that there are only two runnable threads in the system, one at priority 8, and the other at priority 5. Then, the priority 5 thread will not run until the priority 8 thread either blocks or terminates. If, however, a priority 10 thread becomes runnable, it will preempt the priority 8 thread, and become the current running thread.

Unfortunately, in what seems to be a contradiction, the Java language specification also explicitly warns against relying on priority to determine order of execution. It notes that a virtual machine may elect to run a low priority thread, even when there are higher priority runnable threads in the system, in order to avoid starvation [2].

The language specification also leaves the scheduling of equal priority threads unspecified. Indeed, as Java threads are mapped onto whatever means that are supported by the underlying environment, they exhibit the properties of their host threading system. For example, on the Windows NT platform under JVM version 1.1.5, equal priority Java threads are time sliced; moreover, lower priority threads run at a reduced rate, rather than lie dormant until all higher priority threads are done. This behaviour is consistent with that of native threads supported by Windows NT. However, under Solaris, the same version of JVM will adhere strictly to the priority-based model. Furthermore, equal priority threads are scheduled in random order, and run to completion, i.e., they are not time-sliced[1]. This behaviour is dictated by the supporting threading package, the Green Threads library.

It is not difficult to see that this platform is non-deterministic, and leads to

inconsistencies among different implementations of the JVM. Furthermore, the Java language specification does not have any other provisions for managing the CPU as a resource, other than the *preemptive priority-based* scheme discussed above. However, such a scheme has already been shown conclusively to be ineffective for supporting multimedia computing in a general purpose environment [32].

3.2.2 Synchronization

Synchronization is also supported in Java at the language level. The Java language provides the *synchronized* keyword, and supplies methods *wait*(), *notify*(), and *notifyAll*() in the *Object* class.

The *synchronized* keyword allows a programmer to synchronize methods or blocks of code. When a thread enters a synchronized method, or a synchronized block of code, it can be sure that it has exclusive access to the object which implements this method, or the object on which this block of code is synchronized, respectively.

The methods *wait*(), *notify*(), and *notifyAll*() are implemented in the class *Object*. This class is the superclass of all other classes in the Java language. Thus, this set of methods allow threads within Java to communicate and cooperate: they provide the mechanism for Java threads to inform one another about their situations. For example, if a thread T_i is interested in knowing when a condition becomes true, then it can *wait* on some object *Obj*. T_i will be suspended until it is notified. When the anticipated condition becomes true, another thread T_j can call the *notify*() method of *Obj*. T_i can then proceed. Alternatively, T_j can call the *notifyAll*() method of *Obj*; then, all threads waiting on *Obj* will be notified and resume their execution.

The synchronization mechanism in Java is based on monitors [18]. Each Java object has an associated monitor. When a thread enters a synchronized method or block, it must have acquired the monitor associated with the object implementing the method, or the object on which the block is synchronized. The monitor mechanism guarantees that there

---

1. This is the default behaviour when the JVM is configured to use the Green Thread package. However, there is a command-line switch to the Java interpreter that will cause the JVM to time-slice equal priority threads. The time slice quantum can also be specified. Equal priority threads are time-sliced by default when Solaris native threads are used for mapping Java threads.

can be only one thread inside a monitor at any given time. When a thread T_i is inside a monitor M_i, all other threads wishing to enter this monitor must wait on its wait queue. When T_i exits M_i, the first thread on its wait queue is allowed to enter.

The wait and notify methods are supported using a similar mechanism. When a thread waits on some object *Obj*, it is put in the condition queue of the monitor associated with *Obj*, and suspended. When the condition becomes true, another thread can call the *notify*() method of *Obj* to wake up a thread in the condition queue of its monitor. Alternatively, the signalling thread can call *notifyAll*() to wake up all threads in that queue.

3.3 Threads Mapping Strategies

As have been discussed earlier, Java threads are supported by the multithreading mechanisms of its underlying environment, and must be mapped to lower level threads at the virtual machine level. Over the years, many threads mapping models have been proposed and implemented. Most of these models can be grouped into one of three main types: Many-to-One, One-to-One, and Many-to-Many [20]. The mapping of Java threads is no exception.

3.3.1 Many-to-One Model

This model maps many user threads to a single kernel thread, as illustrated in Figure 2. It allows the application to create any number of threads that can execute concurrently. Many-to-One implementations are also referred to as user-level threads. In such an implementation, all threads activities are restricted to user space. Additionally, only one scheduling entity is known to the operating system, and only one thread can access the kernel at any given time. Such implementations do not exploit hardware parallelism, even if more than one processor are available. Therefore, this multithreading model only provide limited concurrency.

The advantage of this model is that it does not require the kernel to support multithreading; it can be implemented on operating environments as primitive as MS-DOS and MS-Windows 3.x. It is also very efficient, as all scheduling decisions and context switches can be handled in user space, without kernel intervention. This is a

considerable advantage in uniprocessor environments where the additional heavy context switches to and from the kernel does not leverage any benefit in terms of increased parallelism. It is thus ideal for embedded control devices such as TV set-top boxes that do not have multiple processors, and CPU bandwidth is at a premium.

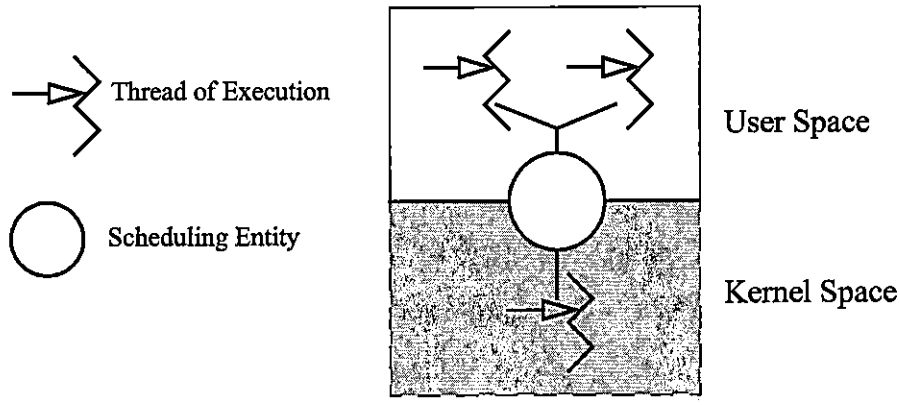


Figure 2: Many-to-One Multithreading Model

3.3.2 One-to-One Model

The One-to-One model maps one user thread to one kernel thread (Figure 3). In this model, each user-level thread created by the application is known to the kernel, and all threads can access the kernel at the same time. The kernel schedules all threads directly. Thus, this model is able to exploit any hardware parallelism that may be present.

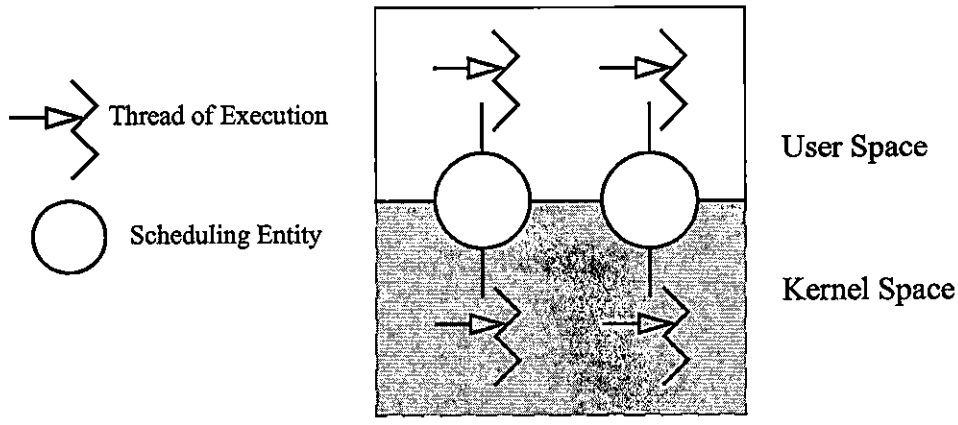


Figure 3: One-to-One Multithreading Model

The main problem with this model is its inefficiency in using kernel resources. It

places restrictions on the programmer to be careful and frugal with threads, as each thread consumes some kernel resource, and adds more cost to the process. As a consequence, many implementations of the One-to-One model, such as Windows NT and the OS/2 threads packages, limit the number of threads each process can create, as well as the total number of threads supported on the system.

3.3.3 Many-to-Many Model

The many-to-many model maps many user-level threads to many kernel-level threads (Figure 4). It avoids the limitations of the One-to-One model, while extending the concurrency of the Many-to-One model. This model, also referred to as the two-level model, minimizes programming effort while reducing the cost of each thread.

In the Many-to-Many model, a user-level threads library provides scheduling of user-level threads above kernel threads. It allows programmers to create fast and cheap threads as with the Many-to-One model. At run time, it multiplexes and schedules runnable threads onto *execution resource*, a pool of kernel threads. It automatically grows or shrinks the pool of kernel threads to meet the demands of the application and the available concurrency from the operating system [39]. The kernel is only required to manage a much smaller set of threads that are currently active.

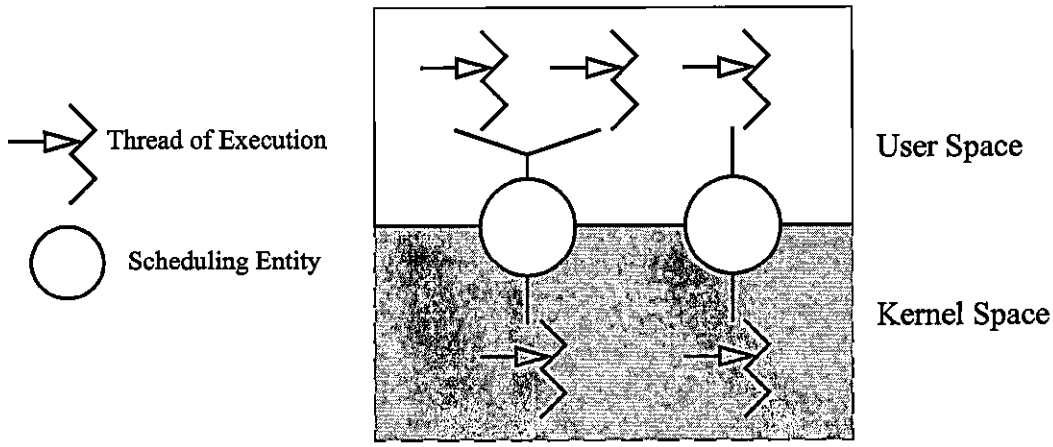


Figure 4: Many-to-Many Multithreading Model

Optionally, a programmer can request some user-level threads be bound to kernel threads, and creating an exclusive connection between the two. This is useful when the application needs to maintain strict control over its own concurrency. However, unbound

user-level threads are more flexible, as they allow the threads library to make most efficient use of kernel resources, and take advantage of available concurrency.

Although there are a number of operating systems that support hosts with multiple processors with shared memory, and have multithreaded kernels, Sun's Solaris operating system is the only commercial system that currently support the Many-to-Many model [20]. The Solaris kernel supports scheduling entities named Light Weight Processes (LWP). A user-level native thread library is employed to facilitate multithreaded programming; it also manages threads and pools of LWPs at run time [39].

Aside from its unavailability on platforms other than Solaris, this model is also hampered by its complexity. Efficient and effective management of the *execution resource* pools is a problem that has not drawn too much attention. The implementation may also be tricky. Furthermore, the advantage of such a scheme on single-processor hosts is uncertain: using more than one kernel scheduling entity will not increase parallelism, but will incur undue overhead; using only one LWP will reduce the Many-to-Many model to the much simpler Many-to-One model.

3.4 Sun's Java Virtual Machine

As the inventor of Java, Sun MicroSystems provides reference implementations of the Java Virtual Machine on both the Windows platform and the Solaris platform. The threads mapping strategies are different in different versions of the virtual machine, and on different platforms. For the rest of this thesis, we will only refer to the current version, version 1.1, of the JVM.

Sun's implementations of JVM are multithreaded; each Java thread has a dedicated JVM thread that interprets its stream of byte-codes. There are as many JVM threads as there are Java threads. Thus, the mapping from Java threads to JVM threads is one-to-one.

The JVM threads are, in turn, supported by the multi-threading mechanism of the underlying operating system. Depending on the capabilities of the supporting system, a VM implementor can map JVM threads to lower level scheduling entities using any of the three forms detailed in the previous section.

On the Microsoft Win32 platform, including Windows 9x and NT, the Sun JVM uses

Windows' native threads package. Each JVM thread is implemented as a native thread. Thus, the mapping from Java threads to kernel threads is One-to-One. Furthermore, the scheduling of these threads is left to the Windows thread scheduler.

On the Solaris platform, however, the situation is a little more complicated. The JVM can be configured to use either Green Thread or Solaris native threads. The threads mapping model and scheduling are different under the two threads packages.

Green Thread is a user-level threads package that uses the preemptive Many-to-One threads mapping model: It bootstraps a single threaded UNIX process into a multi-threaded environment, and multiplexes a number of user threads onto this single scheduling entity. The Green Threads package implements non-blocking IO, context switching, and timer service using a unprivileged user-level library. It also handles scheduling of all its threads in the user space. Configured with this package, the Sun JVM will use one Green thread for each JVM thread; thus the mapping from Java threads to kernel scheduling entities is Many-to-One.

When configured with Solaris native threads, Sun's JVM uses one native thread for each JVM thread. The Solaris native threads package is a system supplied library that uses a two-level, Many-to-Many mapping model: i.e., the collection of user threads is multiplexed onto a pools of kernel threads, or Light-Weight Processes (LWPs). Thus, the mapping from Java threads to kernel scheduling entities is Many-to-Many. The Solaris native threads package handles the scheduling of user threads onto a number of kernel threads. User threads run in the context of some LWP. When it blocks on some kernel service, the native threads package will allocate another LWP on which to schedule user threads [39]. This option is very attractive on modern hardware with multiple CPUs, as it can take advantage of hardware parallelism.

As the mapping of Java threads to lower level threads depends on the particular implementation, it is not difficult to see that there may be inconsistencies among different Java virtual machines. Indeed, even the Sun implementations on Win32 and Solaris are inconsistent. Some multithreaded Java programs work well under Solaris, but enter into race conditions under Win32; others may deadlock with the Solaris version, yet run correctly with the Win32 JVM.

The principal reason for this phenomenon is the different scheduling algorithms used in various mappings. In the Win32 version, thread scheduling is handled by the Windows scheduler. In the Solaris Green Thread version, thread scheduling is done at the user level by the Green Thread library. With Solaris native threads, there are two levels of scheduling: one is at the user-level, where the native threads library schedules user threads onto a pool of Solaris Light Weight Processes; the other is at the kernel level, where the Solaris kernel schedules LWPs onto available processors. Each of these schedulers uses a different algorithm. Some of these algorithms, like the one in the Solaris native threads library, are not even published. Their intricacies are major hindrances to consistency. This situation results in a highly unpredictable system; it makes the system unsuitable for real-time programming, both hard and soft.

We are now ready to introduce our new multithreading model for the Java platform. The new model presented in the next chapter is able to ensure cross platform consistency; at the same time, it provides management facilities for the CPU resource and soft real-time scheduling for Java Threads.

4 A New Model

Although the Java platform has many characteristics that make it desirable for integrated multimedia processing, some of its intricacies also make it unsuitable for such duties. The problem lies chiefly within its threads model: it has no provision for supporting the management of the CPU resource; and it does not force different implementations of the virtual machine to be compatible. To address these problems, we propose an enhanced threads model that conforms to the JVM specification in [39], provides primitives for managing the CPU as a resource for soft real-time tasks, and ensures compatibilities among different implementations. In particular, we propose to incorporate a standard thread scheduling facility based on the Move-To-Rear List Scheduling algorithm across all JVM platforms.

4.1 A New Threads Model

In order to support soft real-time tasks, the platform must allow the specification and provision of certain quality of service guarantees for the CPU resource; it must promote a unified approach to thread scheduling to ensure compatibility across implementations; furthermore, it should require minimum effort in the part of application programmers, and be able to run all existing applications without modification. Our proposal is to adopt the Green Thread package as the standard threads package, and use an enhanced version of the Move-To-Rear List Scheduling algorithm for thread scheduling at the VM level. The basic MTR-LS algorithm would need to be modified to be suitable as a root scheduler for

scheduling system threads which must respect strict priorities. We will also tailor the Green Thread library to take advantage of the capabilities of host platforms, yet maintain its thread model and scheduling properties constant across implementations. In addition, we will develop an extended Java threads API that is 100% compatible with the standard one. In addition to the standard threads control mechanisms, it will facilitate access to the advanced CPU resource management capabilities.

4.2 Why Green Threads?

The more fundamental questions are actually the following: Why use a user-level threads package at all? Why not use whatever threads package that is available on the host platform? This is precisely the current state of affairs with the Java platform. The advantage is that one can exploit the supporting platform to the maximum extent possible. The down side is, as discussed earlier, the inconsistency of the threads model across different implementations of the JVM.

To ensure cross-platform consistency, one must adopt a common threads model and thread scheduling algorithm on all platforms. The Green Thread library is an integral part of the Java 1.1 source tree. It provides multithreading support for all host platforms that do not have native multithreading capabilities. It was the standard threads package for all Java implementations before version 1.1. It was specifically designed for the Java language. Hence, it makes sense to adopt Green Thread as the standard for our new model.

The reference implementation of the Java Virtual Machine from Sun has a clearly defined Host Programming Interface (HPI) as shown in Appendix A. This interface specifies the features which a supporting platform must have in order to host a Java virtual machine. It separates the platform dependent code from the platform independent code (Figure 5) within an implementation of the JVM. Porting Sun's JVM to other platforms is thus an exercise of implementing the HPI on the host platform.

The Green Threads library implements the parts of the Java HPI which support multi-threading and monitors on host platforms that do not have these features. However, most platforms, Win32 and Solaris included, do not support monitors directly. Hence, this part of the Green Thread library is always present for all platforms.

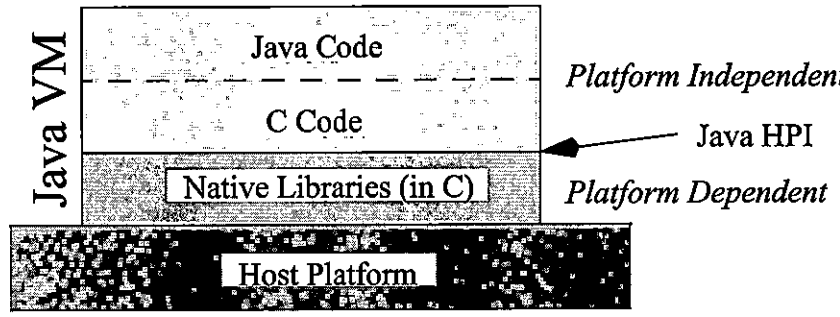


Figure 5: A Java Platform Block Diagram

Our proposal is to include the entire Green Thread library API in the Java HPI, and extend it with features that are required for CPU resource management. Porting the JVM would become the exercise of adapting the Green Thread library to the new platform, plus implementing the other features in the HPI. One can still exploit special capabilities available in particular host platforms; the problem has now become how to best implement the Green Thread library using features found on the host environment.

Another distinct advantage of this approach, in addition to providing a consistent threads model, is that a known thread scheduling algorithm can be specified for all versions of the JVM. Using an extra layer affords us the luxury to specify a thread scheduling algorithm independent of multithreading capabilities of the hosts. As a result, scheduling characteristics of JVMs across all platforms can be known and consistent. Thus, multithreaded Java programs can truly be "*written once, run everywhere.*"

4.3 Threads Scheduling

We further propose that the Move-to-Rear List-Scheduling algorithm be adopted as the basis of a standard thread scheduling algorithm. The basic MTR-LS algorithm needs to be modified to be suitable as a root scheduler that schedules system threads which must express strict priorities. This new thread scheduler will replace the static priority-based scheduler in the Green Thread library, and will become responsible for scheduling all threads in the Java Virtual Machine. This approach allows us to regulate CPU resource consumption both for the Java threads, and for threads spawned by native code libraries. Using a MTR-List Scheduler has a number of additional benefits.

As discussed in Section 2.4, the Move-To-Rear List Scheduling algorithm is a resource scheduling algorithm aimed at providing guaranteed service [6]. Besides the usual quality of service parameters such as fairness, bandwidth partitioning and delay bound, it supports a new QoS criterion called a *cumulative service guarantee*. This is important for soft real-time tasks, especially for multimedia applications: it guarantees that the delay a particular process experiences will not accumulate over its life time, and that the service rate it perceives will not be less than the admitted service rate by more than a constant amount. This essentially guarantees that timing requirements can be met without explicitly specifying deadlines to the scheduler, provided that an adequate service fraction is reserved. Other resource management algorithms such as SFQ only guarantee bounded delay at each phase. Although they may support cumulative service guarantee implicitly, MTR-LS is an algorithm whose support for this QoS criterion has been proven theoretically.

In addition, MTR-LS is simple, efficient, and flexible. The theoretical analysis of the scheduling algorithm is complex; however, the algorithm itself is quite straightforward. A straightforward implementation of this algorithm could have complexity in the order of $O(ln(n))$, where n is the number of active scheduling entities in the system. This is at the same level as SFQ, and is one of the best among resource management algorithms [6].

MTR-LS does not require prior analysis of resource requirements. The level of guarantees it provides can be adjusted; it does not even require admission control when only a fairness guarantee is desired. Admission control is however required to provide delay bound and cumulative service guarantees. Moreover, the delay bound and cumulative service guarantees are bounded by some system constant, and not by the number of scheduling entities in the system. This system constant, as well as the preemption interval, can be adjusted either at run time or at system start-up time to reflect changes in timing requirements. With such an algorithm in place, the new Java platform will have great flexibility for resource management.

Our proposal is to incorporate resource requirement specification and resource consumption enforcement mechanisms in the Java virtual machine. However, we will not force applications to use any specific resource management policies. We will not even

require admission control in the base platform. The policies are left to be implemented by application-level resource managers.

Last but not least, MTR-LS is based on service fractions, not on priority; it avoids an inherent problem of priority-based schedulers on multiprocessors. One fundamental characteristic implied by a priority-based scheduler is that low priority threads will not run if there are higher priority threads that are runnable. On a host with multiple CPUs, however, the scheduling algorithm would attempt to utilize all available CPUs. Consider the following common scenario: the system has one high priority thread and a number of lower priority threads runnable at the same time; and there are two CPUs available. In such a situation, should the scheduler schedule one high priority thread and a low priority thread on the two CPUs? Or should it only schedule the high priority thread, and let the other CPU idle? Doing the former will break the abstraction of a priority-based scheduler; however, doing the latter will result in under-utilization of the CPU.

A service fraction-based scheduling algorithm will not have to face this dilemma, as it does not imply any ordering in its scheduling policy. Having two CPUs only means that the service rate of the server is doubled, and the server can handle two threads simultaneously. The MTR-LS algorithm already supports multiple CPUs. However, it would treat them as separate CPU servers. The intrinsic details of applying MTR-LS to multiprocessors with shared memory may need to be worked out, and the theoretical properties re-analyzed; nevertheless, the overall abstraction can be preserved.

Unfortunately, this advantage of a service fraction based scheduling algorithm would become a disadvantage if the scheduling algorithm is to be used in a root scheduler for system threads which must use strict priorities to specify their urgency. For example, in a priority based system, the system thread that services timers and clock interrupts would have a priority higher than that of any other threads in the system. The scheduler would then preempt the current thread and schedule the timer and clock service thread as soon as it becomes runnable. With a service fraction based scheduler, it is no longer possible to express this urgency among threads. Hence, the MTR-LS algorithm must be modified if it were to be used in a root scheduler for scheduling system threads. Details of this modification is discussed in the next chapter.

4.4 Threads Mapping

As the MTR-LS algorithm has not yet been extended to multiprocessors with shared memory, although such an extension would be possible and natural, we would propose a Many-to-One mapping from Java threads to scheduling entities in the host environment: each Java thread will be mapped onto one VM thread (Green thread); all the VM threads will be multiplexed onto the same process. Such an arrangement is more efficient on commonly available hardware, and portable across different platforms.

The majority of computers available today have only one CPU. This is especially true for home computers and embedded devices like WebTV, which are most likely to use Java for multimedia computing. On such platforms, there is not much hardware parallelism to be exploited. Hence, using the Many-to-One model is not much of a hindrance. On the contrary, it is an advantage, as Many-to-One models are more efficient on uniprocessors than other models. Many frequently used threads operations, such as context switches, become more lightweight as the host operating system need not be involved.

Moreover, there are still a number of platforms, including many derivatives of the UNIX operating system, that do not support multithreaded programming. The Many-to-One model does not require the host environment to support multithreading. Thus, it can be ported to different platforms with relative ease.

When the MTR-LS algorithm is extended for multiprocessor scheduling, we may then support the Many-to-Many model on platforms that exhibit true hardware parallelism. We could use one native scheduling entity for each available processor, and schedule user threads onto the available execution resource. As has been discussed earlier, this change would not effect the overall abstraction, thanks to the service fraction-based scheduling architecture. Cross-platform consistency of the threads model will be preserved.

4.5 Managing the CPU Resource

Our new multithreading model supports management of the CPU resource using a service fraction-based scheme. A service fraction would be assigned to each thread, which specifies its resource allocation. The resource management algorithm based on the MTR-LS algorithm will be implemented in the root scheduler; it guarantees that the allocation of

resources is fair with respect to the service fractions and some system constant. Furthermore, our scheduler also honours a delay bound guarantee, and the cumulative service guarantee, when an application-level resource manager elects to use admission control to enforce certain conditions as set out in Section 2.4.3 and [6].

As stated earlier, the scheduling facility, as is implemented in the new Java virtual machine, will not impose any resource management policies. Controls for adjusting service factions and system constants will be exposed to the upper layer via a Java interface class. An application-level resource manager written in Java will then be able to fine tune the system performance using these controls. The resource manager is free to oversubscribe the CPU, if it so wishes. The timing guarantees and the throughput guarantees will be voided; however, the fairness guarantee will still be honoured.

The new scheduling algorithm will be installed below the VM level; thus, it is able to regulate the CPU resource usage for all VM threads, including threads from the native code libraries. It can, therefore, manage the CPU resource effectively.

This JVM architecture is best implemented on bare hardware, as this is the only option where the JVM's thread scheduler will have absolute control over the CPU resource. Unfortunately, circumstances often limit a JVM implementor to adopt a host platform and build the virtual machine using facilities available from the host operating system. As such, the JVM implementor must develop effective means for CPU resource accounting for the particular host platform using available facilities. Otherwise, the ability of the scheduler to manage the CPU resource will be compromised.

In the next chapter, we will discuss our implementation of this JVM architecture on the Solaris platform. We will discuss, in detail, our enhancement to the MTR-LS algorithm in order for it to allow system threads to express their urgency, and our means for the JVM to effectively manage the CPU resource while running on top of a general purpose operating system.

5 Implementation

The new multithreading model presented in the last chapter has been implemented in a new Java Virtual Machine named Q-JVM. This new Java virtual machine is based on Sun's reference implementation of JVM version 1.1.5. As Sun's Java Virtual Machine was developed in C, the majority of our work was also done using this programming language. There was no significant change to the platform-independent part of the virtual machine. The platform-dependent part of the JVM, thread scheduling in particular, was modified to use a service fraction-based scheduling algorithm in the root scheduler and provide resource management primitives to the upper layers. Q-JVM was developed on the Solaris operating system; however, it does not use any Solaris-specific features.

5.1 Solaris

The source code for JVM version 1.1.5 was obtained from Sun directly; it supports both Windows 95/NT and Solaris. As discussed in Chapter 3, Java Threads are mapped onto Windows native threads using a One-to-One mapping on the Windows platform. On the Solaris platform, however, one has the option to map Java threads onto either the Solaris native threads using a Many-to-Many model or the Green threads using a Many-to-One model. As our model calls for the adoption of Green threads as the standard, we chose to implement our ideas on the Solaris platform as it already has Green Thread ported to it. As our implementation makes little assumption on the support provided by the underlying operating system, it is relatively easy to be ported to other platforms.

5.2 Green Threads

Before presenting our modifications, let's first examine the Green Thread package as it is found in Sun's JVM. Green Thread is a traditional priority-based threads package. It makes extensive use of monitors [18]; it relies on three system threads and monitors for its operation; and it uses a stateless scheduler function which runs on the preempted thread's stack. It is also tightly integrated with the Java Virtual Machine.

Scheduling in Green Thread is based on priority. Priorities are represented by integer values. A larger integer represents a higher priority. Although the basic architecture does not limit the range of priorities, user threads use only ten different priorities: integers from 10 to 1. This Java threads use only these priority values.

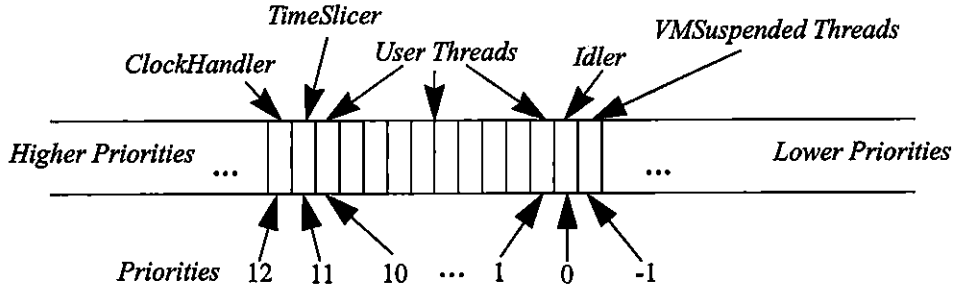


Figure 6: Priorities of Green Threads

Green threads can have any valid integer values as their priorities. Larger integer values represent higher priorities. However, user threads only use priorities 10 to 1. System threads such as the *ClockHandler*, the *TimeSlicer*, and the *Idler* use other values. When user threads are suspended by the virtual machine for garbage collection, their priorities are temporarily changed to -1.

The three system threads in Green Thread are the *ClockHandler* thread, the *TimeSlicer* thread, and the *Idler* thread. The *ClockHandler* thread is responsible for maintaining timeouts for all other threads. It is the highest priority thread in the system: It runs at priority 12. The *ClockHandler* thread lies dormant most of the time. It wakes up whenever a timeout expires or when one is registered or removed. When it wakes up, it notifies all threads whose timeouts have expired. It then scans all the currently active timeouts to locate the next timeout. After registering with the operating system to deliver a signal to it at the next timeout, it suspends itself again.

The *TimeSlicer* thread runs at priority 11. Like the *ClockHandler* thread, it too lies dormant most of the time. When it wakes up, it registers a timeout that expires at the end

of the next preemption interval, and then goes to sleep again. Since it has a higher priority than any other threads in the system, except the *ClockHandler* thread, it will be scheduled as soon as it is runnable, i.e., when its timeout expires. By waking up and going to sleep again, it preempts the current user thread, and allows the scheduler to schedule a new thread according to its scheduling policy.

The use of this time slicing mechanism is optional in Sun's implementation of Green Thread for the Java virtual machine. By default, this thread is not loaded. In this case, threads may be blocked only when they perform some system operation such as I/O or attempting to enter a monitor. An operator can enable time slicing and set quantum size via command line switches to the Java virtual machine.

The *Idler* thread runs only when the system is idle. It has the lowest priority in the system, priority 0. Since the default scheduler is static priority-based, the *Idler* thread will be scheduled only when there are no other runnable threads. Whenever it is scheduled, the *Idler* thread will reclaim memory occupied by the stacks of terminated threads, and then yield CPU to the underlying operating system.

In addition to the three system threads, Green Thread also maintains two other user threads in its system space: the *GC* thread, and the *Finalizer* thread. These two threads are not really system threads as they are not vital to the threads operation; they serve purposes that are specific to the virtual machine only. The *GC* thread runs the garbage collection routines in the JVM, while the *Finalizer* thread runs the `finalize()` routines of discarded Java objects. Both of these two threads run at the lowest priority of user threads, priority 1. However, when the *GC* thread runs, it runs to completion and thus is non-preemptable. Furthermore, when a low memory situation is detected, the Java VM may suspend all user threads and run the garbage collection routines on its own behalf. It is interesting to note that the asynchronous garbage collection facilitated by the *GC* thread may be disabled via a command-line switch to the JVM; however, synchronous GC initiated by the Java VM cannot be avoided once a low memory situation develops.

Green Thread keeps track of threads in priority queues that are implemented as linked lists. There is an *active queue* for all active threads: threads that have been created but not yet terminated. A *runnable queue* links together all threads that are currently runnable.

When a thread becomes runnable, either from a blocked state or from being preempted, it is added to this queue according to its priority. When the scheduler is invoked, it always schedules the first thread in the *runnable queue*. Monitors [18] also use queues to manage threads. Every monitor has a *wait queue* for threads that are waiting to enter it, and a *condition variable wait queue* for threads that are waiting for some conditions to become true.

Monitors are used extensively in Green Thread; they are the foundations for most threads operations. When a thread is suspended, it is usually[1] put on either the wait queue or the condition variable wait queue of some monitor; when it is resumed, it is deleted from the wait queues, and inserted into the runnable queue.

Let's examine the operation of the *ClockHandler* thread as an example. As we have mentioned earlier, this thread is responsible for servicing timeouts for other threads. The *ClockHandler* thread waits on the condition variable wait queue of the *system clock monitor* after it has scheduled a timeout with the operating system.

When the timeout expires, the operating system will deliver a signal to the Green Threads library. The timeout alarm signal handler which was installed when the threads library bootstraps will first preempt the currently running thread and insert it back into the runnable queue. Then, it notifies the *ClockHandler* thread by removing it from the condition variable wait queue of the *system clock monitor* and inserting it into the runnable queue. As the *ClockHandler* thread is the highest priority thread in the system, it will end up at the front of this queue.

When the signal handler exits, the scheduler function is invoked. The scheduler function is very simple: it always schedules the first thread in the runnable queue. As a result, the *ClockHandler* will be scheduled to run immediately.

It is also evident from this example that the scheduling policy in Green Thread is implemented in the queues. Higher priority threads are inserted into the front of the queues before all lower priority threads; equal priority threads are inserted in the order of their arrival. The scheduler function is only responsible for scheduling the first thread it finds

---

1. The exception is that when the JVM decides to collect garbage when a low memory situation is detected, it suspends all threads by changing their priorities to -1. This priority is lower then the priority of all threads, including the *Idler* thread which runs at priority 0.

on the runnable queue. Since the *Idler* thread is always runnable and never blocks, the scheduler function is always be able to find some threads to run.

In addition to the queues and monitors which are global to all threads, each Green thread has a block of private information that helps to facilitate its management. This information includes its priority, its state, the lists of monitors it is waiting on, and information about its stack memory and machine context. These private data structure are initialized and maintained by the Green Thread library for scheduling and monitor operations.

5.3 Extension to Green Thread

Our purpose is to support soft real-time scheduling of Green threads and, in turn, Java threads. Our model calls for the adaptation of the MTR-LS algorithm in the thread scheduler. Our principle is to adhere to the Green Thread abstractions as closely as possible. The main reason for insisting on this principle is to avoid major rework of the upper-level Java runtime environment and provide maximum compatibility for existing applications written in Java.

In the final implementation, the system threads and the monitor infrastructure were largely unchanged. However, the thread private data structure, the queues and the scheduler function were extended to accommodate the new scheduling policy. The preemption and context switching mechanisms were also modified to keep track of CPU usage by individual threads.

5.3.1 Thread Private Data Structures

The first change is to extend the thread data structure to include fields for a time stamp, a service fraction specification, and the *left* value, for each thread. As we recall from Section 2.4, the *left* value of a thread records the time remaining for its current quantum. The thread is moved to the rear of the list when this value becomes zero.

The service fraction is specified by the user. When a thread is created, it is assigned a default service fraction. An authorized user can change this assignment at any time. This value is used to calculate the initial value of *left* before each quantum begins.

The time stamp is used to determine the position of a thread in the service list L: threads having earlier time stamps appear at near the head of L; threads having later time stamps appear at the rear. A thread is moved to the rear of the list when it is assigned a new time stamp that is later than the time stamps of all other active threads in the system.

However, we implemented the time stamp not as a reading of the clock; instead, it is implemented as a 64 bit long integer, where larger integer values represent earlier time. A thread's time stamp is recalculated every time it finishes its quantum. When it is assigned a new time stamp, it is given a 64 bit integer that is smaller than all assigned time stamp values in the system.

This decision stems from our realization that there is a parallel between time stamps and priorities: threads with earlier time stamps are nearer the front of the service list than threads with later time stamps, and need to be serviced before other threads. In effect, these threads have a higher effective priority than other threads. In the original Green Thread library, larger integer values represent higher priority. Hence, we decided to use larger integer values to represent earlier time stamps.

Taking advantage of this parallel between time stamps and priorities greatly simplified the implementation. As the time stamps can be used to express priorities among threads, other parts of the Green Thread library such as the monitor mechanism which assume a priority-based threads model can be used with minimum modifications.

5.3.2 The Service List and the Runnable Queue

The key to implementing the MTR-LS algorithm is the service list. It is supposed to be an ordered list of all active threads in the system, ordered by their time stamps. However, the order of threads is significant only when they are scheduled. The scheduler must always schedule the runnable thread with the earliest time stamp.

Our implementation keeps the *active queue* data structure in Green Threads. In order to avoid unnecessary overhead, threads in this queue are not sorted in any particular order. The runnable threads, however, are kept in the runnable queue and are sorted by their time stamps in this queue; threads with earlier time stamps are nearer the front of the queue. Thus, when the scheduler is invoked, it simply schedules the thread that is at the front of

the queue. This is guaranteed to be the thread with the earliest time stamp.

The *runnable queue* is implemented as a priority queue that uses the time stamp as priority. The original *runnable queue* in Green Thread uses a linked list implementation. Such a data structure is quite inefficient for priority queues: its time complexity is in the order of $O(n)$, where n is the number of threads in the queue. To remedy this inefficiency, we developed a new priority queue based on the heap data structure. The time complexity of this queue is in the order of $O(ln(n))$. This new queue abstract data type (ADT) enabled us to realize the full potential of the scheduling algorithm in terms of efficiency. Other parts of the Green Thread library such as the monitor code which uses priority queues to manage threads are also modified accordingly to take advantage of this ADT.

5.3.3 Resource Usage Accounting

The new scheduling algorithm manages the CPU as a resource. Hence, CPU bandwidth consumption of all threads must be accounted for. This is accomplished using a pair of inline functions added to the preemption and context switching mechanism.

The first inline function is invoked just before the control is transferred to the newly scheduled thread. This function saves the system hardware clock reading in a variable private to the scheduler. The hardware clock is continuously updated. It is accurate to the nearest microsecond.

The second inline function is invoked right after the context is switched away from a thread. It takes another reading of the system clock, and subtracts the last reading of the clock when the current thread is scheduled. The difference is the time that the last scheduled thread has run on the CPU. This difference is subtracted from the *left* value of this thread.

The context switching code is executed every time the control of the CPU changes hands. It is called even when signal handlers are invoked. Thus, this pair of inline functions installed in the context switching code is able to track CPU usage of all Green threads that are managed by the library. This includes any Java threads which are mapped onto Green threads, and all threads spawned by native code libraries. However, this mechanism does not track CPU bandwidth consumed by the context switching code itself,

the signal handlers and the scheduler. These routines are very simple and short, and their invocations are statistically predictable. Hence, their resource consumption is expected to be a constant but negligible amount. This discrepancy is dealt with by reserving a small portion of the CPU that is not allocated to any threads.

Another problem for resource consumption accounting is caused by the fact that, when the JVM runs on top of a multi-programming environment, the operating system may preempt the Java VM runtime environment and give the CPU to another program. For example, assume at time t a thread T is scheduled, and at time $t + \Delta t_1$ the VM runtime is preempted by the operating system. Green Thread's context switching mechanism is not invoked as this preemption is transparent to the user code. At time $t + \Delta t_1 + \Delta t_2$, the VM runtime regains control of the CPU and T continues its execution. At time $t + \Delta t_1 + \Delta t_2 + \Delta t_3$, thread T is preempted by Green Thread. The resource accounting mechanism would record that T has executed for $\Delta t_1 + \Delta t_2 + \Delta t_3$ time units. However, because of the preemption by the operating system, T has only had $\Delta t_1 + \Delta t_3$ time units of CPU time. The other part, Δt_2, was taken away by the operating system to execute system functions and other user programs. Our experiments on Solaris have shown that the effect of such operating system preemption is significant even when the system is sparsely loaded with only the standard mix of daemons.

The solution to this problem is found partially outside the scope of our modification to the Green Threads library and Java VM runtime environment. Most modern operating systems like Solaris and Windows NT have a so called "real-time" scheduling facility. It is designed for programs that require maximum control over their own scheduling. Using such a facility, a user program may arrange to have higher priority than any other tasks in the system; it will preempt even system activities such as paging when it becomes runnable. In effect, the user program becomes the system's root scheduler. It will run as long as it desires; other tasks in the system may run only when this program gives up control of the CPU. This is a dangerous facility to use, as not leaving enough CPU time for critical system tasks will result in their starvation and produce system deadlocks.

Our solution is to take advantage of this facility to run our new virtual machine as a

real-time process, but at the same time create a dedicated VM thread, named the *OS thread*, that does nothing but yields CPU to the operating system. This thread is just another user thread, and is allocated a service fraction. The runtime environment is thus able to regulate its CPU resource consumption, and in turn, control the CPU bandwidth consumed by the operating system and other concurrent user programs. The default service fraction allocated to the *OS thread* is 10% of the total CPU bandwidth. It is adjustable to account for different system load via a command line switch to the Java virtual machine. It can also be adjusted via the resource management API by a user-level resource manager.

This problem of operating system stealing CPU resource from our resource manager would certainly not exist if the underlying platform did not support multiprogramming. For example, if we were to adopt our threads model to JavaOS which runs directly on hardware, the Green Thread scheduler would in fact be the root scheduler. Thus, losing CPU bandwidth to other competing tasks is no longer a concern. For other "pseudo" operating systems such as Windows 9.x, the situation may need to be further analyzed.

5.3.4 Decision Epochs and the Scheduler Function

A *decision epoch* occurs when a thread is blocked or preempted, and the scheduler is invoked to schedule a new thread for execution. In our new implementation of Green Thread, a *decision epoch* occurs when the current thread is preempted by a signal handler, when it performs some system activity such as I/O, monitor operations or thread creation, or when it yields. Our scheduler is a function that runs on the stack of the blocked (or preempted) thread. It does not keep states between invocations other than the runnable queue. In the new implementation, we kept this basic architecture.

As we have discussed earlier in this chapter, Green Thread relies on asynchronous signals from the operating system for notification of events such as timer expiration or changes in device status. Handling a signal will cause the currently running thread to be preempted and a new thread to be scheduled.

Signal handling is asynchronous and is split into two parts: a very simple signal handling function and a more elaborate signal handling thread. The primary role of the

signal handling function is to notify the signal handling thread when a signal is delivered. For example, the *ClockHandler* thread relies on a signal handling function to notify it when a timeout expires. It then wakes up and delivers the timeout.

When a signal is delivered, the current thread is suspended and the corresponding signal handler function is invoked. It first invokes the context switching code to disable any further signals, calculates the CPU resource consumption of the current thread since the context was switched to it, and saves its context. The signal handling function then notifies the signal handling thread by removing it from the condition wait queue of the respective monitor and making it runnable[1]. Finally, it invokes the scheduler function to schedule a new thread, still using the stack of the preempted thread.

A thread may also be blocked when it performs system operations. For example, a thread will be blocked if it performs a read from a file and there is no data available, or if it performs a monitor entry operation but the monitor is already occupied by another thread. When a thread blocks, the context switching code is invoked to disable interrupts, calculate resource consumption and save context, as in the case of signal handling. The scheduler function is also invoked in the same manner to schedule another thread.

However, even though creating a new thread will lead to a decision epoch, it alone will not cause the current thread to be preempted. A newly created thread will have the latest time stamp of all user threads active in the system. The current thread still has the earliest time stamp of all runnable threads after the creation, unless some other condition has changed in the meantime.

The scheduler function is invoked at every *decision epoch*. When invoked, it first fetches the thread with the earliest time stamp from the runnable queue and checks its *left* value. If this value is less than or equal to zero, it moves this thread to the rear of the list by assigning it a new time stamp that is later than any time stamps currently assigned to any user thread and reinserting it into the runnable queue. The *left* of this thread is re-initiated to be $\alpha \cdot T + left$, where α is the service fraction of this thread, T is the virtual time

1. Each signal handling function / signal handling thread pair has a dedicated monitor. No other threads would be waiting on or waiting to enter this monitor. It would be a serious error condition if another thread has already entered the monitor when the signal handler thread is notified. Thus, when a signal handling thread is removed from its monitor's condition wait queue, it can enter the runnable state directly.

quantum, and *left* is this thread's last *left* value. The scheduler function repeats this operation until it finds a thread with a positive *left* value. It then invokes the context switching code to record the start time of this quantum and transfer the control of CPU to the scheduled thread.

Finally, a thread may yield voluntarily. In this case, the context switch is done in the same fashion as described before. However, the yielding thread's *left* value is set to zero. As a result, the next time this thread comes to the attention of the scheduler function it will be moved to the rear of the list, instead of being rescheduled immediately.

5.3.5 Time Slicing

Time slicing is important to the new scheduling algorithm: The MTR-LS algorithm specifies that when a thread is scheduled it shall not run for longer than the smaller of its *left* value and a preemption interval. Our implementation employs a time slicing mechanism based on that of the original Green Thread library.

As part of the initialization routine, the new Green Thread library loads the *TimeSlicer* thread with a preemption interval that is adjustable via a command line switch to Q-JVM. Every time the scheduler runs, it sets a timeout for *TimeSlicer* that will expire after Δt time units, where Δt is the smaller of a preemption interval and the *left* value of the thread being scheduled. When this timeout expires, *TimeSlicer* becomes runnable. It will preempt the current thread, and cause a rescheduling. If the current thread is blocked for other reasons such as a monitor operation before this timer expires, the scheduler will cancel the timeout and set a new one when the next thread is scheduled.

This approach does have one practical limitation: the resolution of the operating system's alarm clock is only 10 milliseconds. This means that when a timeout is scheduled to expire after 1000 microseconds, an alarm signal may not actually be delivered after 10 milliseconds. If this happens, then the current thread may be allowed to consume more CPU bandwidth than it is allotted in its current quantum. This may have a negative impact on timing guarantees when such guarantees are desired. Unfortunately, there is no remedy for this problem except to switch to a supporting platform that has a higher resolution alarm clock, when one is available.

5.4 Extension to MTR-LS

The thread scheduler of the new Java VM (Q-JVM) uses a resource oriented scheduling policy based on the MTR-LS algorithm. This scheduler manages not only the user threads but also system threads such as the *ClockHandler* thread, the *TimeSlicer* thread and the *idler* thread. As we have discussed in the previous chapter, the MTR-LS algorithm is based on service fractions and CPU resource consumption. It does not have the inherent notion of urgency as expressed with priorities.

However, system threads require strict priorities. For example, the *ClockHandler* thread must run as soon as it becomes runnable; i.e., it must have absolutely the highest priority of all threads in the system. Similarly, the *TimeSlicer* thread must have a priority lower than the *ClockHandler* thread but higher than that of any other threads; and the *idler* thread must have a priority that is lower than all other threads in the system.

In order to employ the resource management algorithm successfully in a thread scheduler, one must first resolve this apparent conflict between the resource-oriented scheduling policy and the strictly priority-based policy: how to express absolute priorities in a service fraction-based scheduling framework.

One obvious solution is not to use the service fraction-based resource management algorithm at the root level. Instead, one could employ a two-level hierarchical scheduling architecture as shown in Figure 7. A simple strict priority-based scheduler may be used at the bottom level to schedule system threads and a higher-level scheduling class. The higher-level scheduling class can use the resource management algorithm to schedule user

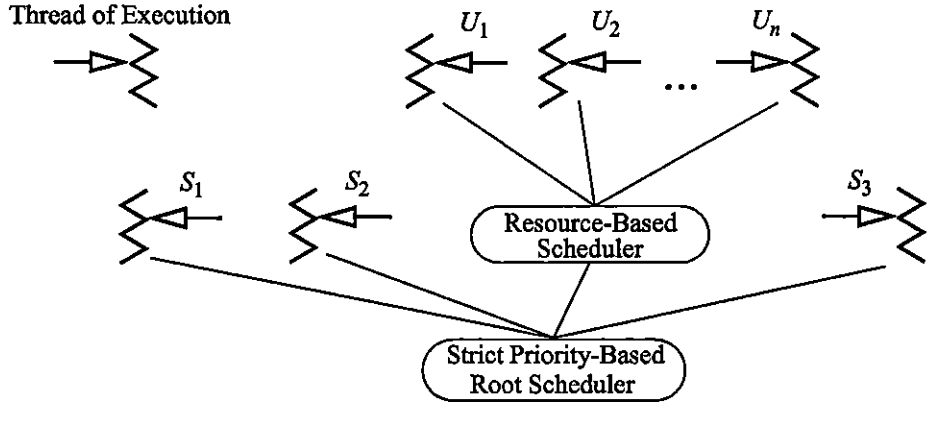


Figure 7: A Two-Level Hierarchical Scheduling Architecture

threads based on their resource consumption.

The drawback of this approach is its inefficiency and complexity. Using more layers of indirection always incurs higher overhead. This is alright if the majority of affected entities benefit from the additional layer. In this particular situation, however, overhead incurred for the optimization for a very small number of system threads penalizes the performance of a usually much larger population of user threads. The additional complexity involved in implementing this hierarchical framework may not be justified, for the same reason.

Our solution is to extend the MTR-LS algorithm to handle the special situation where system threads must be able to express their absolute priorities. We took advantage of our earlier realization (in Section 5.3.1) that MTR-LS schedules threads according to their positions on the service list L, and that the **time stamps** which determine the positions of threads on L can serve effectively as priorities.

Under MTR-LS, a thread will not be scheduled if there are other threads ahead of it on the service list. Hence, if high priority system threads are placed at the beginning of the list ahead of all user threads, then as soon as they become runnable they will be scheduled. Similarly, if the low priority system threads are placed at the rear of the list after all user threads, then they will not be scheduled until there are no other threads that are runnable.

A map of positions of the system threads and user threads on the service list L in our final implementation is shown in Figure 8. The *ClockHandler* thread and the *TimeSlicer* thread are assigned the two earliest time stamps. This puts them at the beginning of the service list. At the same time, the *idler* thread is assigned an artificially late time stamp which puts it near the end of the list[1].

All of the system threads are tagged so that when context is switched away from them, their *left* values are not updated. As a result, they will never be moved to the rear of the service list; thus, their positions are stationary. User threads, in the meantime, are

---

1. The last few positions on the service list L are reserved for implementing *vmSuspend*. This function is used by the Java VM to suspend all user threads when a low memory situation develops, before it starts garbage collection. The original Green Thread implements this function by setting the priority of all user threads to be lower than that of the *idler* thread. We adopt a similar approach by moving all user threads to a position behind the *idler* thread on L. Of course, their original positions on L are also remembered.

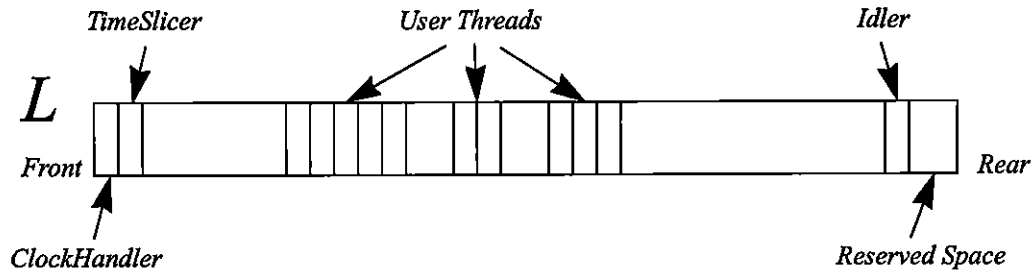


Figure 8: Mapping of System Threads on the Service List.

managed as usual: their *left* values are updated according to their resource consumption; their positions on the service list are adjusted when their *left* values become zero. When a user thread is moved to the rear of the list, it is assigned a time stamp which is later than that of all other user threads and its *left* value re-initialized.

This scheme is conceptually equivalent to the two-level hierarchical scheduling architecture proposed earlier in this section; it just collapses the two levels into one. The basic scheduling discipline in our implementation can be viewed as strictly priority-based, with the time stamps serving as global priorities. The scheduler also implements the second level of the hierarchy; it adjusts the global priorities of user threads according to the MTR-LS algorithm.

This approach takes full advantage of the natural ordering of threads on the service list and the parallel between time stamps and priorities. It provides a simple and straight-forward solution to the problem of using the MTR-LS algorithm in a root scheduler which must allow system threads to express their absolute priorities.

It must be noted that under this scheme, the system threads are not subject to resource consumption accounting. The accounting code ignores them when the context is switched away from the system threads. And the scheduler never changes their position on the list. The system simply does not track their resource consumption.

However, the system threads do consume CPU bandwidth. If left unchecked, this may lead to over subscription of the CPU resource when admission control is desired. Fortunately, we observe that their resource consumption can be bounded. Both high priority system threads, the *ClockHandler* thread and the *TimeSlicer* thread, are sporadic; their invocations are statistically predictable. As they perform a very simple and dedicated

function each time they are scheduled, their resource consumption is expected to be a small and statistically constant amount. To account for this amount, the allocated CPU bandwidth counter is set to 1% when the Green Thread library initializes, before any other threads are created. This in effect reserves 1% of total available bandwidth to account for the consumption by the two high priority system threads. This estimate is arbitrary; more careful analysis is necessary in order to find an optimal value.

The resource consumption by the other system thread, *Idler*, is negligible. This thread is only scheduled when there are no other runnable threads, i.e. when there is no resource contention. It is preempted as soon as another thread requests access to the CPU. When there is no contention, management becomes unnecessary. Hence, the resource manager does not need to track its resource consumption in any case.

Finally we note that Java VM threads such as the *Finalizer* thread and the *GC* thread are treated as user threads. They are subjected to resource consumption accounting and enforcement as are any regular user threads. They are assigned service fractions for 1% and 5% of the total CPU bandwidth, respectively, when they are started at system initialization. Again, these values are arbitrary. They may be different for different applications, and more analysis of their behavioural patterns is required to discover optimal values for specific applications.

The default service fraction assignments for the system threads, the *Finalizer* thread and the *GC* thread, may be adjusted by user-level resource managers via the high-level application programming interface discussed in the next chapter.

5.5 Time Stamp Inversion

Another interesting problem is *time stamp inversion* which could occur when monitors are used for synchronization. It is similar to *priority inversion* in priority-based systems. With the original Green Thread, priority inversion occurs when a lower priority thread which is executing inside a monitor blocks a higher priority thread which wishes to enter the same monitor. On the new platform with the extended MTR-LS algorithm, a similar situation may develop as the time stamps serve as global priorities.

Consider the following example of a priority inversion: let T_1 and T_2 be two threads

where T_1 has a lower priority than T_2. Assume that at time t, T_2 is blocked and T_1 is executing inside a monitor M. Suppose also that at time $t + \Delta t_1$, T_2 becomes runnable. As it has a higher priority than T_1, it will preempt T_1 and be scheduled. Let's further suppose that at time $t + \Delta t_1 + \Delta t_2$, T_2 requests to enter monitor M which is currently held by T_1. At this point, the system has two options: the first one is to kick T_1 out of the monitor, and let T_2 continue. This preserves the semantics of the priority-driven system; however, it breaks the monitor's guarantee, namely no other threads will be allowed to enter an occupied monitor until its owner gives it up. Hence this is not a viable alternative. The second option is to honour the monitor's semantics and let T_1 block T_2. As T_1 has a lower priority than T_2 but is allowed to prevent T_2 from running, a priority inversion occurs.

Priority inversion may lead to starvation. T_2's attempt to acquire monitor M will not be successful as the monitor is already occupied. It cannot make any progress until T_1 exits M. However, T_1 may be blocked by another thread T_3 which has a priority that is higher than that of T_1 but lower than the priority of T_2. T_2 is then deprived of access to the CPU by a lower priority thread T_3. This may lead to its starvation if T_3 is continuously runnable.

A similar situation could develop in the new version of Green Thread which uses the extended MTR-LS algorithm for threads scheduling. Suppose T_1 is the runnable thread that has the earliest time stamp, and is running inside a monitor M. Suppose also that T_2 has a later time stamp than T_1, but is waiting to enter M. When T_1 finishes its quantum, i.e. when its *left* value becomes zero, the scheduler will move it to the rear of the list by assigning it a new time stamp. Since T_1 no longer has the earliest time stamp, it will be preempted. However, it will not give up its resources including the monitors it is currently holding. As a result, T_2 cannot proceed even though it may have the earliest time stamp in the system, as it is blocked at the entrance of M. A *time stamp inversion* occurs. This situation is similar to priority inversion, and may also lead to peculiar circumstances.

5.5.1 Solution to Time Stamp Inversion

The usual solution to priority inversion is priority inheritance. When priority inversion

occurs, the priority of the blocking thread (the lower priority one) can be boosted to the same level as that of blocked thread. This allows the lower priority thread to finish its work and release the monitor as quickly as possible, thus avoiding any other peculiar circumstances. This is also the solution adopted by the original Green Thread library.

We used a similar strategy, time stamp inheritance, to solve the time stamp inversion problem on the new Q-JVM. The solution is twofold. First, when a thread attempts to enter an occupied monitor, it will be blocked and its time stamp will be compared with the current effective time stamp of the monitor's owner. If it has an earlier time stamp, then the monitor owner will inherit the earlier time stamp as its new effective time stamp. Of course, the monitor owner's original time stamp is always remembered and restored when it exits the monitor. Second, when a thread that is inside a monitor finishes its quantum (i.e., when its *left value* becomes 0), it will be assigned a new (latest) time stamp and its *left* value re-initialized. However, if the monitor's wait queue is not empty, i.e. there are other threads waiting to enter this monitor, then the new time stamp would be stored as the *original* time stamp of the monitor owner. The effective time stamp of the monitor owner is not affected. As a result, it will not be blocked because of its new (later) time stamp. Nevertheless, it will fall back to its rightful place on the service list when it exits the monitor.

This scheme is quite simple; but it effectively resolves a potentially disastrous situation. It allows a thread holding a contended resource to finish its critical section as quickly as possible by permitting it to borrow time from its next quantum. This allows the scheduler to be fair even when synchronization is required. Unfortunately, this may also have an adverse effect on some quality of service guarantees, such as the delay bound guarantee. However, it may be argued that maintaining operational correctness and avoiding system failure are more important than honouring QoS guarantees.

5.6 Remarks

In this chapter, we presented Q-JVM which is the implementation of our new threads model for Java based on Sun's Java Virtual Machine version 1.1.5. We discussed many of the design decisions and the reasons why they were made. We also investigated some

problems of adopting a resource-oriented scheduling algorithm, namely the MTR-LS algorithm, in a root-level scheduler and our solutions to these problems. As well, we discussed some practical limitations of our implementation with respect to the original MTR-LS algorithm as it is specified in [6].

In the next chapter, we will present our high-level Java application programming interface that will enable Java programs to access the advanced resource management capabilities built into this new version of the Java virtual machine. In particular, we will present the enhanced threads and thread group API, as well as how a user-level resource manager may be constructed using these facilities.

6 CPU Resource Management API

Our new Java platform developed in this project relies on a high level application programming interface to provide access to the advanced resource management capabilities of Q-JVM. This API is encapsulated into two Java classes: the `panda.threads.QThread` class and the `panda.threads.QThreadGroup` class. These two classes supersede the `java.lang.Thread` class and the `java.lang.ThreadGroup` class, respectively. The original classes are also adapted to provide backwards compatibility for existing Java programs. In addition, a number of system constants can be adjusted via command line options to the JVM to fine tune the system performance.

6.1 QThread

The new `panda.threads.QThread` class provides resource and, in turn, quality of service management to Java threads. As presented in Appendix B, this class supports all normal threads operations such as thread creation, termination and communication. At the same time, it supports the specification of resource requirements in terms of service fraction reservations.

6.1.1 Service Fractions

The minimum service fraction a thread can reserve is one thousandth of the aggregate CPU bandwidth, and there is a total of 1000 units of service fraction to be allotted.

When a thread is created, a service fraction reservation may be specified. However, if

one is not specified, a default value of 15 units (1.5% of the total CPU bandwidth) will be assigned to it. Methods are provided for a thread, or another authorized agent, to query or change its service fraction reservation at any time. When this reservation is changed, the thread's time stamp and its *left* value are recalculated as if the thread were just created. As a result, it is moved to the rear of the list regardless of how much time it has left in its current quantum as specified by its *left*.

One alternative to the above policy is to simply record the new service fraction reservation. When this thread's current quantum expires, the scheduling mechanism will reassign a new time stamp to it, and recalculate the *left* value according to the new reservation. The advantage of this approach is its simplicity. The reservation adjustment code would not need to consider the current state of this thread, whether it is in a runnable state or in one of the suspended states, and adjust its time stamp accordingly. The scheduling code will do that later. The disadvantage of this approach is its latency. The adjustment to the service fraction reservation does not take effect until the current quantum of this thread expires. This latency may be undesirable.

Access control to thread manipulation methods can be implemented using the standard Java convention: an optional Security Manager may be installed to regulate access to particular threads. The default Security Manager allows access within the same thread group. Interested readers may refer to the Java language specification [2] for full detail.

6.1.2 Supporting Resource Management

In addition to methods that control individual threads, the QThread class also provides class methods intended for a user-level resource manager. As our new thread model specified in Section 4.5, the scheduling mechanism built into Q-JVM does not enforce any resource management policies. For example, an application is free to create three threads, each having a service fraction reservation of 500 units. This over-subscribes the CPU by 50%. As such, the timing guarantees will no longer be valid; however, the fairness guarantee will still be honoured: each thread will receive about the same amount of service, i.e. 1/3 of the total available CPU bandwidth.

The class methods `getAllocatedServiceFractions()` and `getAvailableService`

`Fraction()` return the total number of service fractions currently allocated, and the number of service fractions still available, respectively. A user-level resource manager can use these methods to keep track of resource allocation and to build an admission control mechanism. Note that when the system is first started, the available service fractions will be less than 1000 units, as some CPU bandwidth is reserved for high priority system threads like interrupt handlers and threads spawned by the JVM, such as the garbage collector thread. Note also that when admission control is not enforced, the number of allocated service fractions may become greater than 1000 units and available service fractions may become negative.

In addition, class methods are provided for adjusting the service fraction reservations of the two "system" threads spawned by the Java Virtual Machine. The default service fraction reservations for the *GC* thread and *Finalizer* thread are 50 units and 10 units, respectively. An authorized thread may adjust these assignments according to their work load. This mechanism allows a user-level resource manager to regulate resource consumption by these "system" tasks. This was not possible on standard Java platforms.

Moreover, user threads can query whether the *OS thread* is loaded and its resource reservation. This thread is responsible to yield to the operating system when Q-JVM is running in the *real-time* scheduling class (see Section 5.3.3). Its service fraction dictates how much CPU bandwidth is allocated to the operating system to perform system functions. The default allocation is 100 units. A resource manager written in Java is able to change this assignment according to system load.

Finally, we would note that access to these class methods that manipulate the "system" tasks, including the *OS thread* can be secured using the standard Java security mechanism. A "security manager" may be installed to regulate access to these methods [2].

6.1.3 Providing Compatibility

On our new Java platform, the class `panda.threads.QThread` replaces `java.lang.Thread` as the main threads abstraction class. However, a new implementation of the `java.lang.Thread` is also provided to ensure compatibility with existing Java code. Providing such a compatibility layer also fulfills our contractual obligation to be

compatible with the standard Java platform from Sun as spelled out in the licensing agreement for the source code of Sun's Java Virtual Machine.

The `java.lang.Thread` class is implemented as a subclass of `QThread`. The thread control methods such as `suspend()` and `resume()` are mapped to their counterparts in the `QThread` class directly. However, the thread creation and priority manipulation methods have to be modified. Under the original `java.lang.Thread` class, a thread inherits the priority of its parent thread when it is created. It does not have provision for the specification of a priority. Thus, all thread creation methods in `java.lang.Thread` are mapped to the respective methods in `QThread` with default service fraction assignment.

The original Java thread abstraction uses priorities to manage threads. The pair of methods `[s|g]etPriority()` are provided in the `Thread` class to facilitate priority management. The new thread abstraction manages threads using service fractions. As a result, the `setPriority()` method is mapped to the `setServiceFraction()` method in `QThread` and the specified priority is translated into a service fraction using the following formula:

```
service fraction = (10 + Java Priority)
```

As the legal Java priorities range from 10 to 1 with 10 being the highest priority, the translated service fractions range from 20 units to 11 units. This mapping assigns 2% of total CPU bandwidth to a Java thread having the highest priority, and 1.1% to a thread with the lowest priority.

Similarly, the `getPriority()` method is mapped to the `getServiceFraction` method in `QThread` with the same translation formula. If a thread has a service fraction reservation greater than 20 units, it will be reported as having a priority of 10. By the same token, threads having service fractions of less than 11 units will be seen as priority 1.

As `java.lang.Thread` is a subclass of `QThread`, all methods in `QThread` still work for threads created using the legacy `java.lang.Thread` API. In other words, service fraction assignments of legacy threads may be manipulated using respective methods in the `QThread` API. As a result, a user-level resource manager written to the new threads API is able to load legacy applications without modification, and regulate their resource consumption as though they were written to the new API.

6.2 QThreadGroup

Another new class, `panda.threads.QThreadGroup`, is developed to support CPU bandwidth partitioning for thread groups. As presented in Appendix C, the `QThreadGroup` class is structured as a subclass of the standard Java thread group class `java.lang.ThreadGroup`. It augments the original thread group class with resource management capabilities. It implements the basic policy that the aggregate bandwidth allotment of all child threads of a QThreadGroup must not exceed the allotment to the QThreadGroup itself. As we do not wish to impose any resource management policies on user applications, the use of this facility is optional. In other words, QThreads are not forced to be members of some QThreadGroup. Nevertheless, they are still members of some instances of `java.lang.ThreadGroup`, as per Java specification.

Thread groups are used to manage a group of threads that have some common characteristics. For example, a video server application could put threads serving each of its clients into a separate group and manage them as a whole: all threads in a group could, for example, be terminated using a single command when the particular client closes the connection. Thread groups are also the basic unit for implementing security for thread operations like priority adjustment in Java. Before each secured thread operation is carried out, the security manager associated with the thread group which the target thread belongs to is consulted. Therefore, it is a natural extension to use thread groups to manage resource partitioning for QThreads.

The `QThreadGroup` class is a subclass of the `java.lang.ThreadGroup` class. It inherits all methods of its super class; thus it is able to support management functions found on the original Java platform. However, the priority manipulation methods of the standard Java `ThreadGroup` class are overwritten. In their place, `QThreadGroup` provides methods for CPU bandwidth partitioning.

Each QThreadGroup is allocated a service fraction. It is the ceiling of the sum of the service fractions of all its member threads. Methods are provided for querying the total service fraction allotted to a QThreadGroup, aggregate service fractions assigned to its member threads, and the service fractions not yet assigned to any thread. When a QThreadGroup is first created, a service fraction must be specified for it. This assignment

may be changed later subject to resource availability in its parent group and the current allocation to its children.

When a QThread is created in a QThreadGroup, its service fraction allotment is subtracted from the available portion of the group. If the requested allotment is greater than what is available, the thread creation will fail. When the service fraction of a QThread is adjusted, a similar process takes place: the old allotment is added back to the available amount, and the new allotment is subtracted. If the adjustment will cause the aggregate allocation to be greater than the total available bandwidth, an exception will be thrown to abort the operation.

Like its precursor, QThreadGroups may be nested. A QThreadGroup object may have other QThreadGroups, as well as ordinary QThreads, as its children. The same resource management policy as described above applies to child QThreadGroups as it does to child QThreads. Using nested QThreadGroups, a user-level resource manager could implement hierarchical resource partitioning. Although the hierarchy is only logical, and is not directly supported by the underlying Java Virtual Machine (Q-JVM), the resource partitioning is guaranteed by the scheduling algorithm.

When a QThread or a QThreadGroup is created, it is created in the same QThread group as its creator, if the creator is a member of such a group. QThreads and QThread-Groups may migrate to other QThreadGroups, subject to resource availability in the target group. Furthermore, migration operations must be approved by the security manager. However, the default security manager does not have any restriction on migration.

QThreadGroups may be created by any thread, contingent upon approval by the security manager. They may be destroyed only when they are empty, and after a successful security check. Again, the default security manager does not impose any restrictions.

The `QThreadGroup` class is developed as a subclass of its predecessor so that its use would not be mandatory. It implements resource management policies which could assist in the construction of a user-level resource manager. However, these policies are not forced upon applications that wish to develop their own.

6.3 The System Constants

While the thread scheduling facility built into the new VM uses service fractions to specify resource allocation, its timing characteristics are determined by a set of system constants: the virtual time quantum, and the preemption interval [6]. These parameters and a number of others can be specified via command-line arguments to Q-JVM.

The virtual time quantum T is the real time required to service each active thread exactly once when the system is fully loaded, i.e., when all service fractions are allocated. The scheduling algorithm guarantees that the service received by two competing threads, in terms of virtual service time, will not differ by more than $2T$. It also guarantees that the total real time taken to complete some service is bounded by the virtual service time of performing this service plus a constant function of T. Furthermore, it guarantees that the difference between the service a thread receives, in terms of virtual service time, and the service it should receive on an idealized server is bounded by a constant function of T (the cumulative service guarantee) [6].

The preemption interval is the maximum time a thread can run before a decision epoch is encountered. This value affects the scheduling delay, and the granularity of the fairness guarantee and the cumulative service guarantee. Using large preemption intervals is more efficient, as fewer context switches are involved. However, smaller preemption intervals yield finer-grained guarantees.

An operator can change these values via command-line arguments to Q-JVM. If they are not specified, default values are used. The default value for the virtual time quantum is 100 milliseconds; and the default value for preemption interval is 20 milliseconds. The author feels that these are reasonable starting points, as the system clock resolution is only 10 milliseconds. However, the optimal values for these parameters depend on the timing requirements of particular applications; some experimentation may be necessary. These values cannot be changed once the system is started.

In addition to the virtual time quantum and preemption interval, an operator is also able to specify whether the *OS thread* should be loaded, and if so what is the service fraction allocated to the operating system. As we recall from Section 5.3.3, the *OS thread* does nothing but yield CPU to the operating system. This is necessary when Q-JVM is

loaded into the *real-time* scheduling class of the underlying OS with a higher priority than even system processes such as the pager, and any other resident user processes like the daemons in a UNIX system. By default, this thread is not started. The default service fraction assignment to this thread is 100 units (10% of the total CPU bandwidth), if one is not specified when it is started.

6.4 User-Level resource Management

As we have discussed earlier, our new Java platform provides mechanisms for the monitoring, policy specification and management, and usage policing for the CPU resource. It does not, however, force any specific resource management policy onto application programs. Resource management policy specification and management are the responsibility of particular applications which demand such control, or of middleware packages that provide value-added services.

The default setup keeps track of how much resource is used. However, it allows applications to over-subscribe the CPU, and thus to forego the timing and cumulative service guarantees. Nevertheless, the fairness guarantee is always preserved because of the inherent characteristics of the scheduling algorithm employed in Q-JVM.

Application software or middleware packages such as the Java media framework may implement resource management policies specific to their application at the user level, using the `QThread` and `QThreadGroup` API. Such a quality of service oriented application or middleware will be able to allocate CPU bandwidth to individual threads, or groups of threads. These allocations, once in place, will be enforced by our Q-JVM. This was impossible on previous Java platforms where only priority-based scheduling was supported.

Furthermore, a user-level resource manager may fine-tune other parameters such as the virtual time quantum, the preemption intervals, and the service fraction allotted to the operating system for executing other tasks. Using these parameters, the resource manager is able to exercise great control over system performance and the quality of service characteristics for threads under its management. As user-level resource managers are application specific, they are the best candidates to make decisions on these issues.

6.5 Multimedia Applications

The new platform is backward compatible with the standard Java environment: all existing applications written in Java will run without any modification or re-compilation. Unfortunately, legacy applications would have been programmed to use the old threads API; thus they are unable to take full advantage of the new platform.

Legacy applications are supported on the new platform through the modified `java.lang.Thread` API. Thread priorities are mapped into service fractions. The mapped service fractions range from 20 units for the highest priority threads to 11 units for the lowest priority ones. Threads with normal priority will be assigned a service fraction of 15 units.

As resource allocation is not tailored to application requirements, one should not expect a dramatic improvement in performance when moving an unmodified legacy application onto the new platform. The priorities are mapped into relatively small service fractions, and the difference among mapped service fractions are rather small. As a result, demanding soft real-time threads are unable to get more CPU bandwidth, and the best-effort threads are unable to yield. The only advantage of using the new platform in this situation is the fairness guarantee provided by the scheduling facility.

In order to take full advantage of the new platform, legacy multimedia applications such as the media players shipped with the JMF package need to be restructured to use the resource management facility. A user-level resource manager will be needed. It may be as simple as an admission control procedure, or as complex as an implementation of the Utility Model [24]. Threads that have soft real-time requirements need to be identified; proper service fractions must be assigned to all threads. With an intelligent resource manager that implements the Utility Model, this task can be left until run time. The manager can adjust resource allocation until the system settles into a stable state. However, if the resource manager is rather simple, the programmer might need to manually assign service fractions to different threads, and adjust them by trial and error.

When a new multimedia application or a middleware is developed, it is suggested that an intelligent dynamic resource manager should be employed. The manager will oversee the resource allocation, and manage it proactively. At the same time, the applications

should be constructed in a way such that it can negotiate with the resource manager, and reactively adapt to the availability of system resources.

Existing middleware such as the Java Media Framework will need minor adjustments: Components in which Java threads are used need to be modified to use the `QThread` API. The specification of resource allocation parameters for these components should be left to the application that uses these components. However, some extension to their application programming interface may be necessary in order to extend the QoS controls to upper level programs. Optionally, a resource manager may be added to the package that will implement some specific policy, or a general purpose mechanism like the Utility Model.

Unfortunately, as our aim is to provide an infrastructure for supporting soft real-time and multimedia applications, we must refrain from further investigating application level issues such as ones outlined in this section. We have developed test scenarios that demonstrate the capabilities of our new Java platform for supporting soft real-time tasks. They are detailed in the next chapter. However, we must leave the development of multimedia applications and middleware as future work.

7 Experimental Results and Analysis

To experimentally verify the viability of our new Java platform, we developed a test suite to measure its performance. Tests were designed to instrument the effect of using Solaris' so-called real-time scheduling facility and the scheduling overhead of Q-JVM. We showed that the new virtual machine is able to provide predictable resource allocation and resource partitioning. A similar test suite was run on a un-modified Java VM, and the results were compared to those gathered on the enhanced platform. Finally, to verify compatibility, we tested our new Java platform using standard applications such as the *CaffeineMark* benchmark suite, the *HotJava* web browser and applets based on the Java Media Framework without any modification.

7.1 Test Setup

The performance evaluation was done on a Sun SPARCserver 20 with a 60MHz Ultra-SPARC CPU and 64 MB of main memory running Solaris 2.4. All experiments were conducted in multi-user mode with the standard complement of daemons like *lpd*, *sendmail*, *NFS*, and a very lightly loaded *HTTP* server. Moreover, the experiments were conducted when there were no interactive user activities, unless otherwise stated.

We developed a new test suite, called *RaceTest*, to measure throughput. It is a simple Java application that simulates a multithreaded, CPU-intensive program. Most of our experiments were carried out using this test suite.

7.1.1 The RaceTest Test Suite

The *RaceTest* test suite uses a strategy similar to the well known *Dhrystone* benchmark. It is multithreaded. Each of its threads executes an arithmetic calculation in a loop. The number of loops completed in a given time is used as the performance metric.

The basic application has two major components: the *runners* and the *observer*. The *runners* are threads that execute the calculations and loops. The *observer* is a dedicated thread that keeps track of the results.

The runner threads are very simple. The thread body is given in Figure 9 below. The number of loops to run can be controlled via the variables `loops` and `period`. These variables are normally initialized to some large value. The *observer* is responsible to kill the runner threads once a specified time interval has elapsed. However, in the event that the *observer* fails to do so, these variables will prevent the *runners* from running indefinitely. The complete code for the `Runner` class is given in Appendix D.

```
public void run() {
      for (int i=loops; i>0; i--) {
            for (int j=period; j>0; j--) {
                  long tmp = (loops * period) / 1000 + tick;
                  tick++;
            }
      }
}
```

Figure 9: The `run()` method of the Runners.

The *observer* thread is more complicated. First, it allocates a large memory block to hold statistics which will be collected during a session. Then, it starts a number of *runners* and enters a loop monitoring their progress. The body of this monitoring loop is given in Figure 10 on the next page.

During each loop, the *observer* first sleeps for 500 milliseconds. When it wakes up, it samples each *runner* to read the value of variable `tick`. It stores these numbers in the pre-allocated memory block together with the current system clock reading. The *observer* exits the loop when a predetermined number of *observation loops* has been completed, or when one of the *runners* dies. It then prints the collected data from the memory block to the console.

```
// memory bank for storing statistics.
long[][] stats = new long [NUMRUNNERS+1][loops_desired];
int loops_completed = 0;
boolean not_done = true;
do {
      try {Thread.sleep (500);}
      catch (InterruptedException e) {};
      for (int i = 0; i < NUMRUNNERS; i++)
            stats[i][loops_completed] = runners[i].getTick();
      // save a time stamp as well
      stats[NUMRUNNERS][loops_completed]
            = java.lang.System.currentTimeMillis();

      not_done = (++loops_completed < loops_desired);
      for (int i = 0; i < NUMRUNNERS && not_done; i++)
            not_done = not_done && runners[i].isAlive();
} while (not_done);
```

Figure 10: Main body of the *Observer* threads.

The *observer* thread is wrapped in the *RaceTest* application. An operator can specify parameters for *observer* like the number of *observation loops* to complete, the number of *runners* to start, and the service fraction allocated to each *runner*, via command line options to this application. After processing the command line, the application starts up the *observer* thread, which in turn starts the *runners*. The complete source code of the *RaceTest* application and the *observer* thread is given in Appendix E.

In order to collect comparable statistics on the standard Java platform, another version of the *RaceTest* application was developed. It uses the standard Java threads API to start the *observer* and the *runners*. It allows an operator to specify priorities for the *runners* instead of service fractions. Furthermore, the *observer* thread runs at the maximum user priority: 10. Otherwise, it is identical to the version developed for Q-JVM.

A standard Java virtual machine was built from the un-modified source as received from Sun to run this version of *RaceTest*. By default, time slicing is not enabled on the standard JVM. Threads are not preempted until there is some system activity such as the expiration of a timer. However, our Q-JVM has strict, quantum-based preemption. In order to simulate this as closely as possible, the standard JVM is always started with time slicing enabled and a quantum size of 20 milliseconds, unless otherwise specified.

Besides this basic application, a shell script was developed to generate a greedy

system load for simulation of CPU resource contention in a general purpose computing environment. This script uses `tar` and `gzip` to archive a large directory tree repeatedly. When executed alone, it results in a steady system load average of around 1 (full load) on our SPARC server test hardware. This script is executed when we need to simulate a busy system for the experiments. The source code of this script is given in Appendix F.

Finally, a number of shell scripts were developed to automate data collection. Each script encapsulates a few related test scenarios. They load the *RaceTest* with appropriate parameters, start the load generator when necessary, and store the results in predefined directories. An example of such a script is given in Appendix G. Results of some of the test scenarios are presented in the following sections.

7.2 Baseline Performance

Before measuring the performance of the enhanced platform, we ran the test suite on the standard platform to establish a baseline for performance. Tests were conducted on a standard version of the Java virtual machine compiled from un-modified source as received from Sun. To create an environment that is as close as possible to that of the Q-JVM, the standard JVM was started with time slicing enabled and a quantum of 20 milliseconds. The performance numbers are listed in Table 1 below:

	1 Thread	4 Threads	10 Threads
Aggregate Average Throughput (loops per second)	602,797 (L.L.) 268,221 (H.L.)	602,084 (L.L.) 303,672(H.L.)	601,040 (L.L.) 333,136(H.L.)
Average Standard Deviation (% of throughput)	2.52% (L.L.) 10.53% (H.L.)	22.05% (L.L.) 27.70% (H.L.)	26.04% (L.L.) 33.97% (H.L.)
Total Work Completed (loops completed in 500 seconds)	307,078,345 (L.L.) 143,729,374 (H.L.)	306,891,079 (L.L.) 160,742,950 (H.L.)	306,381,115 (L.L.) 175,038,651 (H.L.)

Table 1: Baseline Performance

Legend: L.L. - Lightly Loaded System; H.L. - Heavily Loaded System.

The *RaceTest* application was run with one, four, and ten *runner* threads. All threads were started at the normal priority, priority 5. The first row in Table 1 reports the

aggregate average throughput of all threads measured in loops completed per second. In the case of one thread only, its throughput is reported directly. When there is more than one *runners*, the average throughput of each thread over the entire sampling period is calculated, and the total is reported.

The average throughput of a thread is calculated as follows. The *observer* reports the progress of all *runners*, in terms of loops completed so far, every 500 milliseconds together with a system clock reading. It takes a total of 1000 samples. By subtracting each sample (starting from the second sample) from its predecessor, the number of loops completed in each sampling interval can be calculated. The duration of each interval is calculated in the same fashion. This yields 999 pairs of values that are used to calculate throughput during each sampling interval. The throughput figures are normalized to loops per second. The reported figure is the average of all 999 samples.

The second row in Table 1 measures the variance of the throughput data. For each *runner*, the standard deviation of its 999 samples of throughput is calculated. The figure is then reported as a percentage of the average throughput for this thread. In the case where there is more than one *runner*, the average of the standard deviation figures from all *runners* is reported. This figure gives an indication of how much jitter a thread experiences. Small values represent better quality of service.

The last row in Table 1 reports the total number of loops completed by all *runners*. This figure measures work completed in the entire sampling interval of 500 seconds by the whole system.

Two figures are reported in each cell in Table 1. The first number, tagged by (L.L.), is the result of running the test suite on a *lightly loaded* system with only the standard complement of daemon processes but no interactive user activity. The other number, tagged by (H.L.), is the result of running the test suite on a *heavily loaded* system. The extra system load is generated by the *GenLoad* script.

From the data given in Table 1, we observe the following. When the system is lightly loaded, the aggregate average throughput decreases with the increase of the number of *runners*. This may be attributed to the scheduling overhead of the JVM. However, under heavy load when there is competition for the CPU resource from other activities in the

system, the aggregate average throughput increases with the number of *runners*. This phenomenon may be attributed to the fact that the JVM is competing more aggressively with other applications when it has more threads activity.

Finally we note that the jitter increased significantly, from 2.52% to 10.53%, when another application started to compete for CPU with the JVM. This is expected as the time slicing mechanism is effected by the multi-programming operating system. The lower figure, 2.52%, may be viewed as the *white noise* level, as there is no resource contention in this scenario. This jitter is considered as dictated by the inherent property of the scheduler used in the standard version of the JVM. It may also be viewed as the property of the scheduler that the jitter increases with the number of threads. This effect is more pronounced on a lightly loaded system; it is nevertheless present in both cases.

7.3 Effect of the RT Scheduling Class

We have discussed in Section 5.3.3 that a multi-programming operating system may steal CPU cycles from threads whose resource consumption is monitored by our enhanced Java virtual machine (Q-JVM). To remedy this problem, we suggested that the *real-time* scheduling facility available on many modern operating systems such as Solaris should be utilized. This facility permits Q-JVM to run without interruption until it is ready to release the CPU. This will result in more accurate resource accounting and better Quality of Service. We developed a series of tests to experimentally validate this claim.

First, we used a standard UNIX utility to verify the resource allocation. We used the *top* command to view the list of processes on the system, and their CPU consumption. This command is able to give the CPU usage of each process as a percentage. It is not exact. However, we believe it serves as an effective indication.

The first step is to load the *GenLoad* script by itself. We verify that it consumes nearly 100% of the CPU. Then, we loaded a copy of Q-JVM without using the *real-time* facility. We note that the virtual machine consumes less than 50% of the CPU and commands from the *GenLoad* script consumes a little more than 50% of the CPU resource. Finally, we loaded Q-JVM using the Solaris real-time (RT) scheduling class with various service fraction assignments to the *OS thread*. We were able to verify that the CPU consumption

of the virtual machine versus other parts of the system, as reported by *top,* corresponds to the service fraction assignment to the *OS thread.* For example, when the *OS thread* is assigned a service fraction of 30%, output from *top* will show that Q-JVM is consuming about 70% of the CPU; the rest of the system, including commands from the *GenLoad* script, will be consuming about 30%.

Next, we conducted more quantitative tests using the *RaceTest* suite. First, we ran the test suite on a lightly loaded system in the *time sharing* (TS) class. Then, we loaded the test application into the RT class with a service fraction of 15% allocated to the *OS thread.* Next, we started the *GenLoad* script and ran the test suite again in the TS class. Finally, with the *GenLoad* script still running, we loaded the *RaceTest* application into the RT scheduling class with 15% of the CPU assigned to the *OS thread.* In all cases, only one *runner* was started in *RaceTest,* with a service fraction assignment of 800, or 80% of the CPU. The results are given in Table 2.

	Lightly Loaded System, TS class	**Busy System TS Class**	**Lightly Loaded System, RT class**	**Busy System, RT class**
Average Throughput (loops per second)	601,582	315,971	521,247	507,638
Standard Deviation (% of throughput)	2.76%	16.44%	1.84%	2.65%
Total Work Completed (loops)	308,475,947	168,466,249	269,188,585	261,487,200

Table 2: Effect of the RT Scheduling Class

Comparing average throughput and total work completed as reported in the first two columns of Table 2, we can see the effect of the *GenLoad* script: it effectively cuts down the CPU bandwidth available to the JVM to about half of what was available before it was started. Comparing also this data to column 1 in Table 1, we observe that Q-JVM is a little more aggressive in acquiring CPU cycles than the standard version: on Q-JVM, the *runner* is able to complete 54% as much work on a busy system as on a lightly loaded system; on the standard JVM, it can complete only 46% as much work.

Another observation that we can make from the same comparison is that there is more sample variance in the throughput data collected on the new Q-JVM than the same

collected on the standard one: The standard deviation of throughput on busy and lightly-loaded system is 16.44% and 2.76% respectively on Q-JVM; the corresponding figures from the standard JVM are 10.53% and 2.52%. This indicates that Q-JVM is more susceptible to the operating system stealing CPU cycles from its resource manager, when it is not using the *real-time* facility of the underlying OS.

The situation is much different when Q-JVM is started under the *real-time* scheduling class of Solaris. The data in columns 3 and 4 of Table 2 shows that the standard deviation of throughput is much less in this case than in any previous cases. Even on a busy system, the application threads experience less jitter on Q-JVM than they did on the standard JVM with no extra load.

This data also shows that Q-JVM running in the *real-time* scheduling class of Solaris only uses around 85% of the total CPU bandwidth regardless of the system load. When the system was lightly loaded, the average throughput and the total work completed were 86.6% and 87.3%, respectively, of what were achieved by the same setup running in the "time-sharing" class. Under heavy load, the results are 84.4% and 84.7% for average throughput and total work completed, respectively.

This phenomenon can be explained as the following. The *OS thread* in Q-JVM is responsible for yielding CPU to the rest of the system. It uses the `pause()` system call for this purpose. As a result, when it is scheduled, it will cause the entire VM process to be suspended until some signal is delivered. In other words, the *OS thread* is always busy and always consumes all CPU resources allocated to it. If there is no other processes contending for the CPU, the system will simply idle when Q-JVM yields.

7.4 Scheduling Overhead

A major concern in using a complex resource management scheme such as the one built into the new Q-JVM in place of a simple scheduler is that the scheduling overhead may be high. To evaluate this overhead, we compared the aggregate average throughput achieved by the *RaceTest* application on the new virtual machine versus on the standard JVM. The results are presented in Table 3 on the next page. In order to ensure that the collected data are directly comparable, both Q-JVM and the standard JVM are started under the *time-*

sharing scheduling class. All tests are done on a lightly loaded system with no interactive user activity.

The most distinguished feature of Table 3 is that the differences between all data items are very small: all figures are within 0.5% of each other. This indicates that the overall scheduling overhead of Q-JVM is very small comparing to the standard Java virtual machine. However, as the differences in all cases are smaller than the sample variance, one must be reminded that they may not be statistically significant.

Nevertheless, we may observe the following two trends when comparing data from configuration 1 and 2 on Q-JVM: 1) the scheduling overhead increases with the number of active threads; however, the increase is not great. 2) the scheduling overhead is smaller when the total allocated service fraction is closer to 100%.

	1 Thread	4 Threads	10 Threads
Standard JVM	602,797	602,084	601,040
Q-JVM Configuration 1	601,399	599,860	599,714
Q-JVM Configuration 2	601,582	600,939	599,882

Table 3: Effect of Scheduling Overhead on Thread Throughput

All reported figures are *Aggregate Average Throughput* (in loops per second) as defined in Section 7.2. On the standard Java VM, all threads are started with the normal priority, priority 5. On the enhanced Q-JVM, all threads are assigned the normal service fraction, 15 or 1.5% of total CPU bandwidth, in configuration 1. In configuration 2, however, all threads are assigned equal service fractions totaling 800. For example, when there are 10 threads, each is assigned a service fraction of 80.

The first trend is evident when we examine each row in Table 3 individually. We see a decrease in aggregate average throughput as we move from left to right. This is understandable, as more active threads means more context switches, thus the higher overhead. We also notice that the overhead increases faster on Q-JVM than on the standard virtual machine when the number of threads increases from 1 to 4. But, the increase in overhead is smaller when the number of threads increases from 4 to 10 on Q-JVM than on the standard one. This may indicate that Q-JVM is more scalable; it handles large number of threads better than the standard Java virtual machine.

The second trend is most pronounced when there were 4 threads. In configuration 1, the total allocated service fraction was 60. It increased to 800 in the second configuration. Recall that a thread is moved to the rear of the service list L when its *left* value becomes zero. This value is then re-initialized to be the product of the thread's service fraction and the virtual time quantum, i.e., the system constant T. With smaller service fraction assignments, the threads' quantum expires more often; more calculation and manipulation is thus necessary. When the service fraction assignments increase, less turnover is involved. This somewhat reduces the scheduling overhead. Consequently, throughput of the *runners* improves.

The same effect is less pronounced but still visible in the other two cases. When there was only one *runner*, the scheduler does not have as many active threads to juggle; i.e, the scheduling overhead was not very large to begin with. Thus the improvement is small when the service fraction assignment to the *runner* was enlarged. As the number of threads increased, the savings in overhead became more significant. However, when the number of threads was increased to 10, the benefit of less overhead per threads started to be overpowered by the increase in overhead resulting from having more threads to schedule. This is the reason why the improvement in aggregate throughput was insignificant when the aggregate service fraction assignment was increased, in the case where there are 10 threads.

7.5 Predictable Resource Allocation

We have argued that one major benefit of the new platform will be predictable resource allocation. In particular, it will give equal access to the CPU for threads with equal service fraction assignments. This is not possible on the standard platform: there is no provision to specify CPU resource allocation. One may assign equal priorities to competing threads and hope that they can gain equal access. We predicted that one will still see a degree of unfairness in such an arrangement.

To validate this claim, we compared average throughput of 10 *runners* in the following three configurations on a lightly loaded system. First, we started the *RaceTest* application with 10 *runners* on Q-JVM. Each *runner* thread is assigned a service fraction of 90 units,

or 9% of the CPU. Then, we ran the test on the standard JVM with time slicing enabled and a quantum size of 20 milliseconds. All threads are assigned the same priority, 5. Next, we ran the same test again on the standard platform with the default setting, i.e., with the time slicing feature disabled. In order to get comparable results, all tests are conducted in the *time-sharing* class of Solaris.

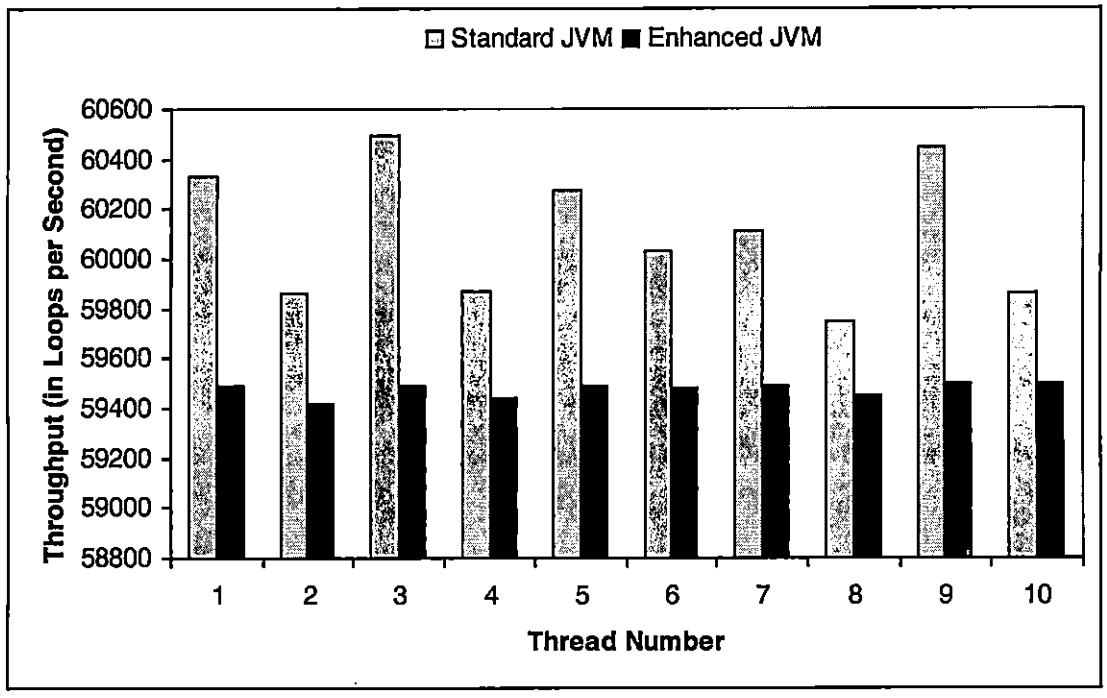


Figure 11: Comparison of throughput of threads on the standard JVM *with time slicing* and on Q-JVM

The comparison of results from the first two configurations is shown in Figure 11 above. Data from each of the 10 *runners* in both configurations are graphed individually. From the graph it is clear that the throughputs that the threads are able to achieve are more uniform on Q-JVM than on the standard JVM. Upon close examination, we find that the differences of throughputs among threads on Q-JVM are all within 0.1% of their average. However, the differences among threads on the standard JVM are around 1% of their average. This is an one decimal order of magnitude improvement in predictability and fairness. Note that the differences of throughput among different threads and between different virtual machines are somewhat exaggerated in the above figure because of the particular scale used. Note also that the standard JVM was run with time slicing enabled. This feature is not enabled in the default configuration.

When we compared data collected on Q-JVM in the last configuration to that from running the standard JVM in its default configuration, we see a much larger difference. This comparison is shown in Figure 12 below. Reading from the graph, we notice that the best performing runner received over 50% more bandwidth than the worst performing one on the standard JVM. In other words, when time slicing is not enabled, resource allocation becomes highly unpredictable on the standard platform. Note that the scale used in this diagram is much different than that of Figure 11, and the throughput axis does not start from 0.

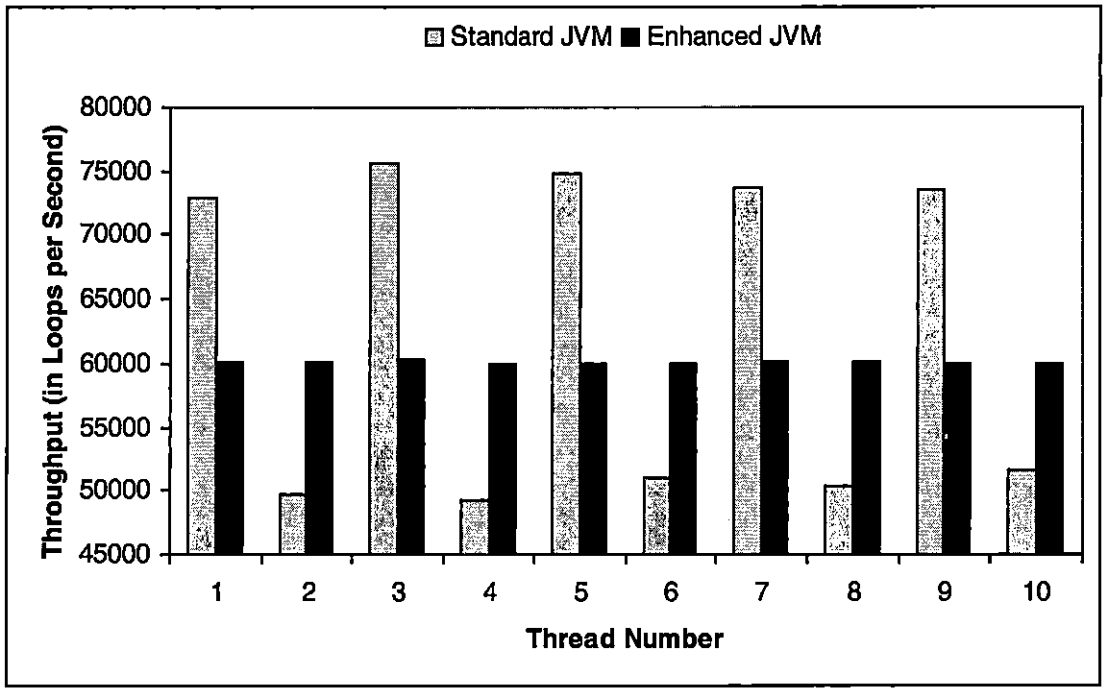


Figure 12: Comparison of throughput of threads on the standard JVM *without time slicing* and on Q-JVM

After close examination of the raw data, we discover that under this configuration rescheduling usually occurs twice in each 500 millisecond sampling interval. The first rescheduling is caused by the *observer* thread. Since it runs at the maximum user priority, it is scheduled as soon as it becomes runnable, i.e., when its timer expires. This preempts the current *runner* thread. The cause of the second preemption is unknown; however, it usually occurs around 10 milliseconds before *observer*'s timer expires. This preemption is sometimes skipped.

As a result, two *runner* threads are serviced on the CPU during each sampling interval,

with the first thread runs for about 490 ms and the second one runs for 10 ms. If the second preemption is never skipped, we would expect a result where the odd numbered runners receive 50 times more CPU bandwidth than the even numbered ones. As this preemption is skipped "pseudo randomly", we see the current result where the odd numbered threads receive around 50% more CPU bandwidth than the even numbered ones.

From this analysis, it is evident that the *observer* thread has a huge impact on the performance of the scheduler in the standard JVM. It negatively affects the fairness and predictability of resource allocation on this platform. This is a serious defect as it is common to have high priority threads that wake up only periodically in soft real-time applications. However, having such threads may cause starvation on the standard Java VM in default configuration.

7.6 Resource Partitioning

Another major benefit of the new platform is its support for resource partitioning. A user-level resource manager is able to assign portions of CPU bandwidth to specific threads, and have the underlying virtual machine enforce the allocation. In the previous section, we have already shown that the new Q-JVM will ensure equal access to the CPU when all threads are assigned the same service fractions. In this section, we will show that the bandwidth a thread receives can be adjusted using service fraction assignments.

This experiment was again conducted using the *RaceTest* suite on Q-JVM running in the Solaris *time-sharing* scheduling class. The application was setup to start two runners each time. The service fraction assigned to each runner was varied for each run. The experiments successfully demonstrated that the throughput achieved by each runner corresponded to its service fraction assignments. We present data from two test runs below to illustrate the result.

In the first case, two *runners* were started with service fraction assignments of 600 units and 300 units respectively. This in effect reserved 60% of the CPU bandwidth for the first *runner* and 30% for the second. Some of the remaining 10% of the bandwidth was taken by the virtual machine for system activities such as the *finalizer* thread. The rest was divided arbitrarily by the thread scheduler among competing threads.

The average throughput (in loops per second) and total work completed (in number of loops finished) of both *runners* are listed in Table 4 below. As shown in the table, CPU bandwidth received by "Runner 1" and "Runner 2" were 63% and 37%, respectively, of the total bandwidth, regardless of whether bandwidth is measured by average throughput or total work completed. This result indicates that both *runners* received more than enough bandwidth to satisfy their respective reservations. The work completed by both *runners* over time is graphed in Figure 13 on the next page.

	Service Fraction Assignment	Average Throughput (loops / second)	Percentage of Total	Total Work Completed (loops)	Percentage of Total
Runner 1	600	376,231	63%	195,258,977	63%
Runner 2	300	221,727	37%	115,065,578	37%
Total	900	597,958	100%	310,324,555	100%

Table 4: Resource Partitioning Case One

Compare this data to that of Table 1 on page 66. Suppose that the throughput in the case where there is a single thread running on the standard JVM on a lightly loaded system is the base-line performance for 100% CPU bandwidth. The bandwidth received by "Runner 1" and "Runner 2" are still 62% and 36% of that base-line, respectively.

The analysis for the second case is essentially the same. In this case, the runners were assigned service fractions of 700 units and 200 units. In turn, they received 74% and 26% of the total CPU bandwidth, respectively. The results are listed in Table 5 below.

	Service Fraction Assignment	Average Throughput (loops / second)	Percentage of Total	Total Work Completed (loops)	Percentage of Total
Runner 1	700	442,728	74%	229,558,894	74%
Runner 2	200	158,069	26%	82,036,339	26%
Total	900	600,798	100%	311,595,233	100%

Table 5: Resource Partitioning Case Two

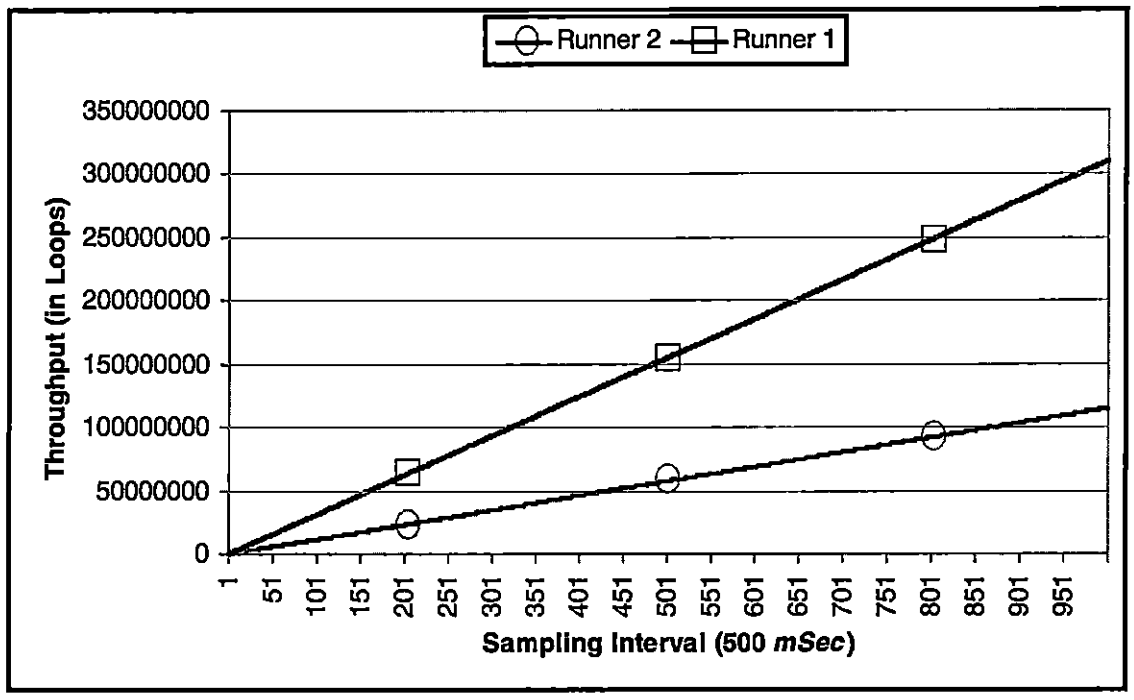


Figure 13: Throughput of Two Threads Over Time

7.7 Other Tests

In addition to the quantitative tests described in the previous sections, we also developed a number of other tests to analyze the new platform in more complex situations. Some of these tests exercise the graphics sub-system, while others uses standard applications to measure the comparative performance and compatibility of the new platform with respect to the standard one.

7.7.1 *Spheres* and *Clock*

Spheres and *Clock* are two applications that exercise the graphics sub-system of a Java virtual Machine. They are intended to demonstrate the resource allocation mechanism of the new platform when more complex components are involved. However, they are structured so that they will run on both the new and the standard platforms.

The *Spheres* application emulates a gas chamber with four gas molecules. Each gas molecules has an initial momentum; and the system is totally elastic. This application

shows how these molecules bounce around in the gas chamber. A screen shot of this application is shown in Figure 14 below.

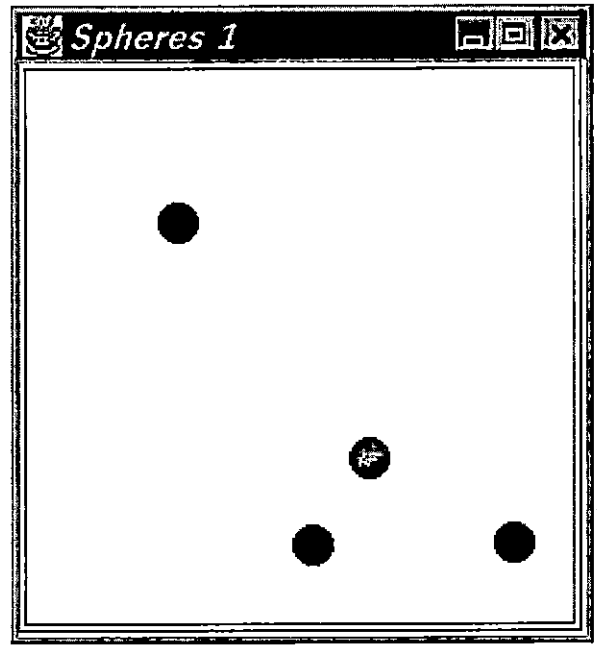


Figure 14: A Screen Shot of the *Spheres* Application

Despite its name, the *Clock* application does not emulate a conventional clock. It has a set of four long hands fixed on the 12 o'clock, 3 o'clock, 6 o'clock, and the 9 o'clock positions. It has another set of four short hands fixed between the long hands. A ball is moved around the clock face in a circle centered at the cross point of the hands. A screen shot of the application window is shown in Figure 15 on the next page.

Both of these two applications are single threaded, and run in their own windows. The applications' threads are CPU intensive: they sit in tight loops to calculate the next positions of the balls (or ball), repaint the screen using double buffering, and hand over the screen update requests to the GUI sub-system as quickly as possible.

A special loader program was developed. If the standard JVM is used, the loader will use the standard Java threads API to start the applications. However, if Q-JVM is detected, the *QThread* API will be used to take advantage of the resource management features. An operator is able to specify how many copies of *Spheres* or *Clock* applications to start, and their respective priorities or service fractions, via command line options to the loader. For example, one could request the loader to start 2 copies of *Clock* with service fractions 350

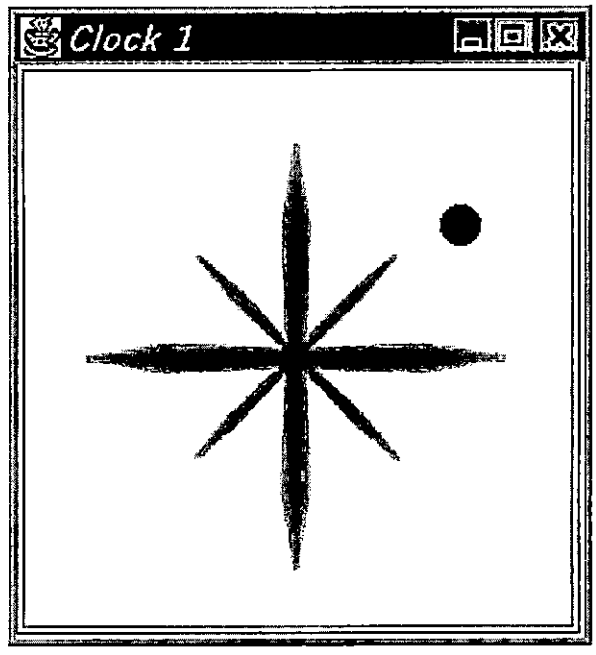


Figure 15: A Screen Shot of the *Clock* Application

units and 250 units respectively, and another copy of *Spheres* with a service fraction of 200 units. All three will then be loaded into the same Java virtual machine, and start running simultaneously.

We tested the applications on both platforms. On the standard Java virtual machine, all instances of the test applications had to be given equal priorities. Otherwise, the lower priority applications could not make any progress until all of the higher priority ones were killed. For example, we loaded one instance of the *Spheres* application and another instance of the *Clock* application at priorities 8 and 5 respectively. We observed that the balls in the *Spheres* application's window bounced around but the ball on the clock face did not move at all. However, once the *Spheres* application was terminated, the *Clock* application started to make steady progress, as indicated by the ball moving around its clock face. When we loaded two instances of the *Spheres* application with the same priority, both windows were animated, showing the progress of the application threads.

On the enhanced platform, however, we were able to observe directly that applications with larger service fraction assignments ran proportionally faster than the ones with smaller service fraction assignments. For example, when we started two instances of the *Clock* application with service fractions of 600 units and 300 units respectively, we were

able to observe that the ball in the first *Clock* application's window advances approximately twice as fast as the ball in the second *Clock* application's window. This test illustrates graphically the advantage of our enhanced Q-JVM over the standard Java virtual machine for managing the CPU resource.

7.7.2 CaffeineMark

CaffeineMark [37] is a popular benchmark suite for the Java platform developed by *Pendragon Software*. It analyses Java system performance in many different areas, including integer and float point calculation, loops, logic operations, string manipulation, method invocation, image manipulation and graphics operations using the Abstract Windowing Toolkit, and common GUI operations. This multithreaded benchmark suite gives an indication of the overall performance of a Java platform.

Running this test suite on our enhanced Java platform fulfills two purposes: first, this yields a score that is directly comparable to scores from a wide range of Java platforms; second, being able to run this benchmark suite successfully is evidence of the compatibility of the new platform with respect to the standard one.

Test scores from the new Q-JVM and a standard JVM compiled from the unmodified source are listed in Table 6 below. A detailed breakdown of each score is not given, as we are only interested in the overall performance.

	Q-JVM (in TS Class)	**Q-JVM (in RT Class)**	**Standard JVM**
Scores	137	148	144

Table 6: *CaffeineMark* Scores on Various Test Systems

The test suite was unmodified, and was run on our SPARC-server test system with no interactive user activities. The display was an X server running on a Windows NT workstation. The first column of Table 6 reports the score of Q-JVM running in the *time-sharing* scheduling class of Solaris. The second column reports the score when it was running in the *real-time* scheduling class with no service fraction assigned to system activities originated outside of the JVM. The standard JVM was started in the *time-*

sharing class as it normally was.

The result given in Table 6 confirms our earlier analysis, that the overhead of our enhanced Q-JVM is small comparing to the standard Java virtual machine under similar conditions. It also indicates that our new platform is compatible with the standard one from Sun.

7.7.3 The HotJava Browser

HotJava is an full-featured web browser developed by Sun for the Java platform. It is a complex multithreaded application written in Java. It supports the latest HTML and e-mail standards and a host of media types and web-browsing features. The ability to run this application on Q-JVM is a strong indication that it is compatible with the standard Java platform.

We used the *HotJava* browser version 1.1.4 extensively on our new Java platform. The browser was not modified in any way; it was supported by the compatibility threads API supplied by the new platform. During testing, we found that the browser functioned as expected. Furthermore, it was able to complete the *CaffeineMark* benchmark suite as well. The scores were 121 and 135 when running in the *time-sharing* class and *real-time* class, respectively.

7.7.4 Java Media Framework

We also tested our new platform using the Java Media Framework (JMF) version 1.0 from Sun. The JMF is a set of Java extension APIs that are designed for multimedia computing. It supports animated presentations using 2D and 3D graphics, as well as integrated digital continuous media processing.

This test was not exhaustive; it was targeted at verifying compatibility with the standard platform. We were able to run all the sample applications shipped with the JMF package; furthermore, we were able to use the included media player applet to play back pre-recorded movie and sound tracks in various formats with satisfactory performance.

For these tests, the JMF package was not modified in any way. In other words, it was running on top of the compatibility threads API, and was unable to take advantage of the

enhanced resource management facility of the underlying platform.

Porting such a comprehensive package to our new platform is not a trivial task. It involves finding a suitable resource management policy, and developing a resource manager and a security manager for the framework. Furthermore, all areas where threads are used must be identified and modified to use the enhanced *QThread* API.

Although this work is very interesting, it unfortunately falls outside of the scope of our thesis. Our aim has been to provide an infrastructure for supporting soft real-time and multimedia applications. The issues of developing multimedia applications and middleware that are able to take advantage of our enhanced platform are thus left to be explored as future work.

8 Conclusion

Recent advances in computing hardware have made possible new classes of multimedia applications that can process digital continuous media (CM). Integrated continuous media processing presents unique challenges to the underlying supporting environment: it imposes real-time requirements on the host operating system and its subsystems, as continuous media data must be presented continuously in time in a predetermined rate in order to convey their meaning. However, applications that process CM data are often classified as being soft real-time, as missing a particular deadline is not fatal as long as it is not missed by too much, and as long as most other deadlines are not missed.

Real-time programming, whether hard or soft, is about resource management. An integrated multimedia system must possess capabilities to allocate, partition, and police the usage of system resources such as CPU bandwidth, so that certain Quality of Service (QoS) parameters can be expressed and guaranteed. Studies in CPU scheduling and bandwidth management abound. However, the current generation of commercially available general purpose computing platforms such as UNIX, Windows, and the Java platform still lacks the necessary facility to support QoS management for soft real-time tasks. Even recent systems that claim to have been developed for multimedia processing, such as BeOS [3][4][5], do not have any provisions for CPU resource management.

We further contended that the Java platform has many characteristics that make it very desirable for integrated multimedia processing: it is a simple and small language; it is dynamic and object-oriented; it supports a rich set of APIs for multimedia processing. Most importantly, it was designed for embedded applications, and is an ideal platform for

media-capable integrated devices of the future.

The work presented in this thesis is a significant effort in bringing CPU resource management to a general purpose computing platform. In particular, we developed a new thread model and implemented a resource management algorithm in a new Java virtual machine, named Q-JVM, based on Sun's JVM version 1.1.5. Furthermore, we developed a resource management API for the CPU resource that supersedes the original `java.lang.Thread` and `java.lang.ThreadGroup` API. Our preliminary test results show that our Q-JVM is compatible with the standard JVM and is able to support resource allocation and management among threads based on service fractions.

8.1 Summary

Our work began with a survey of resource management algorithms for the CPU resource. We discovered that both hard real-time scheduling algorithms, such as the Rate Monotonic Scheduling algorithm, and static priority-based scheduling algorithms, like the one found in UNIX SVR4, are not sufficient for managing the CPU resource for soft real-time tasks.

Fortunately, resource management has been a hot topic of recent research. A large body of work exists in this area. Some of these works are based on previous research for managing other types of resources like network link scheduling, while others are based on some economics principles. Yet, few are ground-breaking fresh ideas. Among these, we found the Move-to-Rear List Scheduling algorithm developed at the Bell Labs[6]. It is a resource management algorithm designed for integrated multimedia systems. It can manage many types of resources, and provides quality of service guarantees for soft real-time tasks.

We then investigated multithreading support on the Java platform. We studied multi-threaded programming using the Java language, and strategies for mapping Java threads onto native threads supported by host environments. Finally, we examined Sun's reference implementation of the Java Virtual Machine Version 1.1.5, in order to determine the best strategy to transplant resource management capabilities into this platform.

Armed with this knowledge, we set out to define a new thread model for the Java platform. We proposed to adopt the Green Thread package as the standard threads package

for Java virtual machines, and use a modified version of the MTR-LS algorithm for thread scheduling at the VM level. We will tailor the Green Thread library to take advantage of the capabilities of host platforms, yet maintain its thread model and scheduling properties constant across implementations. In addition, we proposed an extended Java threads API, that facilitates access to the advanced CPU resource management capabilities of the new platform. The standard Java threads API would be preserved and adapted to the new one.

As a proof of concept, we implemented our model in Q-JVM. This new Java VM is based on the source code of the reference implementation of Java Virtual Machine (version 1.1.5) licensed from Sun Microsystems. Our modification to the platform is concentrated in the virtual machine, in particular, we implemented an extended version of the MTR-LS algorithm in the Q-JVM thread scheduler. Other parts of the Java platform were not affected. A number of design decisions and issues which surfaced during the implementation are also addressed in this thesis.

To enable access to the advanced resource management capabilities of Q-JVM, we developed a set of new threads API for Java. Application programmers can build a user-level resource manager tailored to their application using the new `panda.threads.QThread` and `panda.threads.QThreadGroup` application program interface.

The `QThread` API enables resource and, in turn, quality of service management for Java threads. It supports all normal threads operations such as thread creation, termination and communication. At the same time, it supports specification of resource requirements in terms of service fraction reservations. The standard Java threads API is adapted and mapped to this API.

The `QThreadGroup` API facilitates CPU bandwidth partitioning for groups of threads. It augments the original thread group class with resource management capabilities. It implements the basic policy that the aggregate bandwidth allotment of all child threads of a `QThreadGroup` must not exceed the allotment to the `QThreadGroup` itself. As we do not wish to impose any resource management policies on application programmers, the use of this facility is optional.

To experimentally verify the viability of our new platform, we designed a test suite to measure its performance. Tests were developed to instrument the effect of using Solaris'

real-time scheduling facility and the scheduling overhead of Q-JVM. We showed that the new platform is able to provide predictable resource allocation and resource partitioning. A similar test suite was run on an un-modified Java VM, and the results were compared to those gathered on the enhanced platform.

Finally, to verify compatibility, we tested Q-JVM using a standard benchmark, *CaffeineMark*, the *HotJava* browser and applications based on the Java Media Framework without any modifications. The test results show that the new platform performs as expected.

8.2 Future Work

There are still much work to be done to transform the Java platform completely into a quality of service oriented platform suitable for continuous media processing. The most immediate work involves examining the class libraries that come with the virtual machine, such as the Abstract Windowing Toolkit API and networking APIs, and port them to the new `QThread` API. Extension APIs such as the Java Media Framework must be examined as well, in order for them to take full advantage of our new platform.

It would also be interesting to investigate the recently released *Java 2* platform in detail, and assess its impact on the work presented in this thesis. Our preliminary study showed that while it made improvements in many areas, the *Java 2* platform still lacks the resource management features necessary for supporting soft real-time applications. It seems that the majority of the enhancements is outside of the scope of the virtual machine and the core language features. Instead, they have been concentrated on extension APIs. One notable exception, however, is an improved garbage collector. It is claimed to be more suitable for real-time applications. The full impact of this improvement still needs to be studied.

Another direction to take could be in application and middleware development. A multimedia middleware based on the Java Media Framework could be realized using our enhanced platform. In addition to the media manipulation features supported by the JMF, it would feature a general purpose and extensible resource manager and security manager. Such managers could employ advanced resource management models like the Utility

Model [24]. However, development of such managers have to be the subject of future research.

Finally, one could also study the possibility of extending the MTR-LS algorithm to multiprocessors. Our understanding is that the MTR-LS algorithm is naturally extensible to such platforms. In fact, work may have already been underway at the Bell labs for such an extension. Once a multiprocessor version of the MTR-LS algorithm is in place, our threads model for Java could be extended again to take advantage of modern hardware where true hardware parallelism exists. For example, we would be able to use a *n*-to-*m* threads mapping model to multiplex a number of Java threads over a pool of operating system threads. The size of the native thread pool can vary according to available processors. Nevertheless, the scheduling properties of MTR-LS would have been preserved.

References

[1] D. P. Anderson. Meta-Scheduling for Distributed Continuous Media. *ACM Transactions on Computer Systems*, 11(3):226-252, August 1993.

[2] K. Arnold and J. Gosling. The Java Programming Language, Second Edition, *ISBN: 0-201-31006-6*. Addision-Wesley, Reading, Massachusetts. 1998.

[3] Be, Inc. The Media OS - A Technical White Paper. Available at *http://www.be.com/products/beos/mediaos.html*, December 1998.

[4] Be, Inc. The Be Operating System Product Datasheet, BeOS Release 4. Available at *http://www.be.com/products/beos/*beos_datasheet.html, December 1998.

[5] Be, Inc. The Be Operating System Product Specifications, BeOS Release 4. Available at *http://www.be.com/products/beos/*beos_specifications.html, December 1998.

[6] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Move-To-Rear List Scheduling: a new scheduling algorithm for providing QoS guarantees. In *Proceedings of The Fifth ACM International Multimedia Conference*, pp. 63-73, November 9-13, 1997.

[7] J. Davin and A. Heybey. A Simulation Study of Fair Queueing and Policy Enforcement. *Computer Communication Review*, 20(5):23-29, October 1990.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM*, pp. 1-12, September 1989.

[9] B. Ford and S, Susarla. CPU Inheritance Scheduling. In *Proceedings of Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 91-105, 1996.

[10] D. Le Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46-58, April 1991.

[11] S.J. Golestani. A Self-Clocked Fair Queueing Scheme for Broadband Applications. In *Proceedings of the 13th Annual Joint Conference of the IEEE Computer and Communications Societies on Networking for Global Communication*. 2:636-646, IEEE Computer Society Press, June 1994.

[12] J. Gosling, and H. McGilton. The Java Language Environment, A White Paper. Available at *http://www.javasoft.com/docs/white/langenv/*. JavaSoft, May 1996.

[13] R. Govindan and D.P. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 68-80, October, 1991.

[14] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pp. 107-121, October 1996.

[15] P. Goyal, S.S. Lam, and H.M. Vin. Determining End-to-End Delay Bounds In Heterogeneous Networks. In *Proceedings of the Workshop on Network and Operating System Support for Digital Audio and Video*, p. 273, April 1995.

[16] P. Goyal, H.M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. In *Proceedings of ACM SIGCOMM'96*, pp. 157-168, August 1996.

[17] A. Greenberg and N. Madras. How Fair is Fair Queueing. *The Journal of ACM*, 39(3):568-598, July 1992.

[18] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, October 1974.

[19] M. Horie, J.C. Pang, E. Manning and G.C. Shoja. Using Meta-Interfaces to Support Secure, Dynamic System Reconfiguration", In the Proceeding of the 4th International Conference on Configurable Distributed Systems, May 4-6, 1998, Annapolis, Maryland, USA.

[20] JavaSoft. Multithreaded Implementation and Comparisons, A White Paper. Available at *http://solaris.javasoft.com/developer/news/whitepapers/mtwp.html*. Part No.: 96168-001. JavaSoft. April 1996.

[21] JavaSoft, Silicon Graphics, and Intel Corporation. Java Media Framework Media Player API. Available at *http://java.sun.com/products/java-media/jmf/forDevelopers/playerguide/index.html*. JavaSoft. May 1998.

[22] K. Jeffay, D.L. Stone, and F.D. Smith. Kernel Support for Live Digital Audio and Video.*Computer Communications*, 15:388-395, July/August 1992.

[23] K. Jeffay and D.L. Stone. Accounting for Internet Handling Costs in Dynamic Priority Task Systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium Raleigh-Durham*, NC, pp. 212-221, December 1993.

[24] M. S. Khan. Quality Adaptation in a Multi-session Multimedia System: Model, Algorithms and Architecture. *Ph.D. Dissertation*. Department of ECE, University of Victoria. May, 1998.

[25] D. Kramer, et. al. The Java Platform - a White Paper. Available at *ftp://ftp.javasoft.com/docs/papers/JavaPlatform.ps*, JavaSoft, May 1996.

[26] K. Lee. Performance Bounds in Communication Networks With Variable-Rate Links. In *Proceedings of ACM SIGCOMM'95*, pp. 126-136, October 1995.

[27] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. *Addison-Wesley*, Reading, Massachusetts. September 1996.

[28] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprocessing in a Hard-Real Time Environment. *The Journal of ACM*, 20:46-61, January 1973.

[29] P. Madany, et. al. JavaOS: A Stand-alone Java Environment. Available at *ftp://ftp.javasoft.com/docs/papers/JavaOS.cover.ps*, JavaSoft, May 1996.

[30] C.W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *Proceedings of the IEEE ICMCS'94*, pp. 90-99, May 1994.

[31] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, Raleigh-Durham, NC, pp. 120-130, December 1993.

[32] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 41-53, November 1993.

[33] J. Nieh, M.S. Lam. The Design, Implementation and Evaluation of SMART: A scheduler for Multimedia Applications. In *Proceedings of the 16th Symposium on Operating Systems Principles*, St. Malo, France, pp. 184-197, October, 1997.

[34] K. Nilsen. Java for Real-Time. In the *Journal of Real-Time Systems*, 11(2):197-205, Number 2, 1996.

[35] K. Nilsen. Issues in the Design and Implementation of Real-Time Java. *Java Developer's Journal*, 1(1):44, June 1996.

[36] K. Nilsen. Adding Real Time Capabilities to Java. In the *Communications of the ACM*, 41(6):49-56, 1998.

[37] Pendragon Software. The CaffeineMark Benchmark Suite. http://www.web-fayre.com/

[38] I. Stoica, H. Abdel-Wahab, and K. Jeffay. A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. In *Proceedings of the IEEE Real Time Systems Symposium*, December 1996.

[39] SunSoft. Solaris Multithreaded Programming Guide. Sun MicroSystems Inc., Mountain View, California. 1994.

[40] A. Varma and D. Stiliadis. Design and Analysis of Frame-Based Fair Queueing: A New Traffic Scheduling Algorithm for Packet Switched Networks. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ACM SIGMETRICS Performance Evaluation Review, 24(1):104-115, May 1996.

[41] C. Waldspurger and W. Weihl. Stride Scheduling: Deterministic Proportional-share Resource Management. *Technical Report TM-528*, MIT, Laboratory for Computer Science, June 1995.

[42] C.A. Waldspurger and W.E. Weihl. Lottery Scheduling: Flexible Proportional-share Resource Management. In *Proceedings of Symposium on Operating System Design and Implementation*, pp. 1-11, November 1994.

Appendices

Appendix A Excerpt of Java Virtual Machine HPI

The following is an excerpt from the Java virtual machine Host Programming Interface (HPI). It shows the part that is related to threads and monitors management. It includes our modifications for the adoptation of a service fraction based resource management algorithm in the root thread scheduler. The original file is also included with the binary release of Java virtual machine version 1.1.5 in the `include` subdirectory as `sys_api.h`.

```
/*
 * @(#)sys_api.h1.63 98/06/08
 *
 * System or Host dependent API.  This defines the "porting layer"
 * for POSIX.1 compliant operating systems.
 */
#ifndef _SYS_API_H_
#define _SYS_API_H_

...<code deleted>...

/*
 * System API for threads
 */
typedef struct  sys_thread sys_thread_t;
typedef struct  sys_mon sys_mon_t;
typedef void *  stackp_t;
int     sysThreadBootstrap(sys_thread_t **, void *);
void    sysThreadInitializeSystemThreads(void);
int     sysThreadCreate(long, uint_t flags, void *(*)(void *),
                        ys_thread_t **, void *);
void    sysThreadExit(void);
sys_thread_t * sysThreadSelf(void);
void  sysThreadYield(void);
```

```
int   sysThreadVMSuspend(sys_thread_t *, sys_thread_t *);
void  sysThreadVMSuspendMe(void);
int   sysThreadVMUnsuspend(sys_thread_t *);
int   sysThreadSuspend(sys_thread_t *);
int   sysThreadResume(sys_thread_t *);
int   sysThreadSetPriority(sys_thread_t *, int);
int   sysThreadGetPriority(sys_thread_t *, int *);
/* The following code is added by JCP, June 1998 */
#if defined(USE_MTR_LS)
int sysThreadSetServiceFraction(sys_thread_t *, int);
int sysThreadGetServiceFraction(sys_thread_t *, int *);
int sysThreadGetAvailableServiceFraction();
int sysThreadGetAllocatedServiceFraction();
int sysThreadCreateWeighted(long, uint_t flags, void *(*)(void *),
                            sys_thread_t **, int, void *);
#endif  /* USE_MTR_LS */
/* end of addition */
void *   sysThreadStackPointer(sys_thread_t *);
stackp_t sysThreadStackBase(sys_thread_t *);
void   sysThreadSetStackBase(sys_thread_t *, stackp_t);
int    sysThreadSingle(void);
void   sysThreadMulti(void);
int    sysThreadEnumerateOver(int (*)(sys_thread_t *, void *),
                              void *);
void   sysThreadInit(sys_thread_t *, stackp_t);
void * sysThreadGetBackPtr(sys_thread_t *);
int    sysThreadCheckStack(void);
int     sysInterruptsPending(void);
void    sysThreadPostException(sys_thread_t *, void *);
void    sysThreadDumpInfo(sys_thread_t *);
void    sysThreadInterrupt(sys_thread_t *);
int     sysThreadIsInterrupted(sys_thread_t *, long);
int     sysThreadAlloc(sys_thread_t **, stackp_t, void *);
int     sysThreadFree(sys_thread_t *);
/*
 * System API for monitors
 */
int     sysMonitorSizeof(void);
int     sysMonitorInit(sys_mon_t *);
int     sysMonitorDestroy(sys_mon_t *);
int     sysMonitorEnter(sys_mon_t *);
bool_t  sysMonitorEntered(sys_mon_t *);
int     sysMonitorExit(sys_mon_t *);
int     sysMonitorNotify(sys_mon_t *);
int     sysMonitorNotifyAll(sys_mon_t *);
int     sysMonitorWait(sys_mon_t *, int, bool_t);
void    sysMonitorDumpInfo(sys_mon_t *);
bool_t  sysMonitorInUse(sys_mon_t *);
#define SYS_OK          0
#define SYS_ERR        -1
#define SYS_INTRPT     -2    /* Operation was interrupted */
#define SYS_TIMEOUT    -3    /* A timer ran out */
#define SYS_NOMEM      -5    /* Ran out of memory */
```

```
#define SYS_NORESOURCE -6     /* Ran out of some system resource */
/*
 * Symbolic constant to be used when waiting indefinitly on a
 * condition variable
 */

...<code deleted>...

#define SYS_TIMEOUT_INFINITY -1

#endif /* !_SYS_API_H_ */
```

Appendix B The `QThread` API

Class panda.threads.QThread

```
java.lang.Object
   |
   +----panda.threads.QThread
```

```
public class QThread
extends Object
```

A *QThread* is a thread that supports Quality of Service (QoS) specification and provision. A modified version of Sun's Java Virtual Machine (1.1.5) allows a service fraction to be specified for each thread, and schedul all runnnable threads using a modified Move-To-Rear List Scheduling (MTR-LS) algorithm.

QThreads do not have priorities; instead, each QThread has an associated Service Fraction. Service Fraction is specified with an integer in the range of 1 to 1000. Each unit of service fraction denote 0.1% of the total available CPU bandwidth. For example, if a QThread is assigned a service fraction of 50, then it will get no less than 5% of the total CPU bandwidth. The default service fraction for all threads is 15 (1.5% of CPU). The service fraction of any thread can be changed to any positive value upto the maximum allowed. However, this class does not enforce any admission control policies.

Normally, 1% of the CPU bandwidth is reserved for all system activities, such as signal processing. The GC thread and the Finalizer thread are allocated service fractions of 50 units and 10 units, respectively.

See Also:

Runnable, Thread

Variable Index

- **MAX_SERVICEFRACTION**
  The maximum service fraction that a thread can have.
- **MIN_SERVICEFRACTION**
  The minimum service fraction that a thread can have.
- **NORM_SERVICEFRACTION**
  The default service fraction that is assigned to a thread.
- **TOTAL_SERVICEFRACTION**
  The total service fraction available.

Constructor Index

- **QThread()**
  Allocates a new QThread object.
- **QThread(int)**
  Allocates a new QThread object.
- **QThread(Runnable)**
  Allocates a new QThread object.
- **QThread(Runnable, int)**
  Allocates a new QThread object.
- **QThread(Runnable, String)**
  Allocates a new QThread object.
- **QThread(Runnable, String, int)**
  Allocates a new QThread object.
- **QThread(String)**
  Allocates a new QThread object.
- **QThread(String, int)**
  Allocates a new QThread object.
- **QThread(ThreadGroup, Runnable)**
  Allocates a new QThread object.
- **QThread(ThreadGroup, Runnable, int)**
  Allocates a new QThread object.
- **QThread(ThreadGroup, Runnable, String)**
  Allocates a new QThread object.
- **QThread(ThreadGroup, Runnable, String, int)**
  Allocates a new QThread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
- **QThread(ThreadGroup, String)**
  Allocates a new QThread object.
- **QThread(ThreadGroup, String, int)**
  Allocates a new QThread object.

Method Index

- **activeCount()**
  Returns the current number of active threads in this thread group.
- **checkAccess()**
  Determines if the currently running thread has permission to modify this thread.
- **countStackFrames()**
  Counts the number of stack frames in this thread.
- **currentThread()**
  Returns a reference to the currently executing thread object.
- **destroy()**
  Destroys this thread, without any cleanup.
- **dumpStack()**
  Prints a stack trace of the current thread.
- **enumerate(Thread[])**
  Copies into the specified array every active thread in this thread group and its subgroups.

- **getAllocatedServiceFraction()**
  Returns total CPU bandwidth that has been allocated to all threads.
- **getAvailableServiceFraction()**
  Returns currently available CPU bandwidth.
- **getFinalizerServiceFraction()**
  Returns CPU bandwidth that has been allocated to the Finalizer thread.
- **getGCServiceFraction()**
  Returns CPU bandwidth that has been allocated to the GC thread.
- **getName()**
  Returns this thread's name.
- **getReservedServiceFraction()**
  Get the reserved service fraction.
- **getServiceFraction()**
  Returns this thread's service fraction.
- **getSystemTServiceFraction()**
  Returns CPU bandwidth that has been allocated to the system thread, if it is loaded.
- **getThreadGroup()**
  Returns this thread's thread group.
- **interrupt()**
  Interrupts this thread.
- **interrupted()**
  Tests if the current thread has been interrupted.
- **isAlive()**
  Tests if this thread is alive.
- **isDaemon()**
  Tests if this thread is a daemon thread.
- **isInterrupted()**
  Tests if the current thread has been interrupted.
- **join()**
  Waits for this thread to die.
- **join(long)**
  Waits at most millis milliseconds for this thread to die.
- **join(long, int)**
  Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.
- **resume()**
  Resumes a suspended thread.
- **run()**
  If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
- **setDaemon(boolean)**
  Marks this thread as either a daemon thread or a user thread.
- **setFinalizerServiceFraction(int)**
  Changes the service fraction assignment for the Finalizer thread.
- **setGCServiceFraction(int)**
  Changes the service fraction assignment for the GC thread.
- **setName(String)**
  Changes the name of this thread to be equal to the argument name.
- **setReservedServiceFraction(int)**
  Changes the reserved service fraction.
- **setServiceFraction(int)**
  Changes the service fraction reservation of this thread.
- **setSystemTServiceFraction(int)**
  Changes the service fraction assignment for the system thread.

- **sleep(long)**
  Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **sleep(long, int)**
  Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
- **start()**
  Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
- **stop()**
  Forces the thread to stop executing.
- **stop(Throwable)**
  Forces the thread to stop executing.
- **suspend()**
  Suspends this thread.
- **systemThreadLoaded()**
  Returns whether the system thread is loaded.
- **toString()**
  Returns a string representation of this thread, including the thread's name, service fraction, and thread group.
- **yield()**
  Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Variables

- **MIN_SERVICEFRACTION**
  `public static final int MIN_SERVICEFRACTION`

  The minimum service fraction that a thread can have.

- **NORM_SERVICEFRACTION**
  `public static final int NORM_SERVICEFRACTION`

  The default service fraction that is assigned to a thread.

- **MAX_SERVICEFRACTION**
  `public static final int MAX_SERVICEFRACTION`

  The maximum service fraction that a thread can have.

- **TOTAL_SERVICEFRACTION**
  `public static final int TOTAL_SERVICEFRACTION`

  The total service fraction available.

Constructors

- **QThread**
  `public QThread()`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, null, gname, NORM_SERVICEFRACTION)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

  Threads created this way must have overridden their `run()` method to actually do anything.

  See Also:
  Thread, QThread

- **QThread**
  `public QThread(int service_fraction)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, null, gname, service_fraction)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

  Threads created this way must have overridden their `run()` method to actually do anything.

  Parameters:
  service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.
  Throws: InsufficientCpuResourceException
  If the specified service fraction is greated than the maximum allowed.
  Throws: IllegalArgumentException
  If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.
  See Also:
  Thread, QThread

- **QThread**
  `public QThread(Runnable target)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, target, gname, NORM_SERVICEFRACTION)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

  Parameters:
  target - the object whose run method is called.
  See Also:
  Thread, QThread

- **QThread**
  `public QThread(Runnable target,int service_fraction)`

Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, target, gname, service_fraction)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

Parameters:
target - the object whose run method is called.
service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.
Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.
Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.
See Also:
Thread, QThread

- **QThread**
  `public QThread(ThreadGroup group,Runnable target)`

Allocates a new `QThread` object. This constructor has the same effect as `QThread(group, target, gname,NORM_SERVICEFRACTION)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

Parameters:
group - the thread group.
target - the object whose run method is called.
Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.
See Also:
Thread, QThread

- **QThread**
  `public QThread(ThreadGroup group,Runnable target, int service_fraction)`

Allocates a new `QThread` object. This constructor has the same effect as `QThread(group, target, gname, service_fraction)`, where ***gname*** is a newly generated name. Automatically generated names are of the form "`QThread-`"+*n*, where ***n*** is an integer.

Parameters:
group - the thread group.
target - the object whose run method is called.
service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.
Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.
Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.
Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.
See Also:
Thread, QThread

- **QThread**
  `public QThread(String name)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, null, name, NORM_SERVICEFRACTION)`.

  Parameters:
  name - the name of the new thread.
  See Also:
  Thread, QThread

- **QThread**
  `public QThread(String name,int service_fraction)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, null, name, service_fraction)`.

  Parameters:
  name - the name of the new thread.
  service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.
  Throws: InsufficientCpuResourceException
  If the specified service fraction is greated than the maximum allowed.
  Throws: IllegalArgumentException
  If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.
  See Also:
  Thread, QThread

- **QThread**
  `public QThread(ThreadGroup group, String name)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(group, null, name, NORM_SERVICEFRACTION)`.

  Parameters:
  group - the thread group.
  name - the name of the new thread.
  Throws: SecurityException
  if the current thread cannot create a thread in the specified thread group.
  See Also:
  Thread, QThread

- **QThread**
  `public QThread(ThreadGroup group, String name, int service_fraction)`

  Allocates a new `QThread` object. This constructor has the same effect as `QThread(group, null, name, service_fraction)`.

  Parameters:

group - the thread group.
name - the name of the new thread.
service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.

Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.

Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.

Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.

See Also:
Thread, QThread

- **QThread**

```
public QThread(Runnable target, String name)
```

Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, target, name, NORM_SERVICEFRACTION)`.

Parameters:
target - the object whose run method is called.
name - the name of the new thread.

See Also:
Thread, QThread

- **QThread**

```
public QThread(Runnable target, String name, int service_fraction)
```

Allocates a new `QThread` object. This constructor has the same effect as `QThread(null, target, name, service_fraction)`.

Parameters:
target - the object whose run method is called.
name - the name of the new thread.
service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.

Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.

Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.

See Also:
Thread, QThread

- **QThread**

```
public QThread(ThreadGroup group, Runnable target, String name)
```

Allocates a new `QThread` object. This constructor has the same effect as `QThread(group, target, name, NORM_SERVICEFRACTION)`.

Parameters:

group - the thread group.
target - the object whose run method is called.
name - the name of the new thread.

Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.

See Also:
Thread, QThread

- **QThread**

```
public QThread(ThreadGroup group,
               Runnable target,
               String name,
               int service_fraction)
```

Allocates a new `QThread` object so that it has `target` as its run object, has the specified name as its name, and belongs to the thread group referred to by `group`.

If `group` is not `null`, the `checkAccess` method of that thread group is called with no arguments; this may result in throwing a `SecurityException`; if `group` is `null`, the new process belongs to the same group as the thread that is creating the new thread.

If the `target` argument is not `null`, the `run` method of the `target` is called when this thread is started. If the `target` argument is `null`, this thread's `run` method is called when this thread is started.

The specified service fraction of the newly created thread is checked aganist the maximum allowed for any single thread. If the check fails, an `InsufficientCpuResourceException` is thrown. Otherwise, the thread's service fraction is set to the specified value, and this value is deducted from the available amount. `setServiceFraction()` may be used to change the service fraction reservation later.

The newly created thread is initially marked as being a daemon thread if and only if the thread creating it is currently marked as a daemon thread. The method `setDaemon` may be used to change whether or not a thread is a daemon.

Parameters:
group - the thread group.
target - the object whose run method is called.
name - the name of the new thread.
service_fraction - the desired service fraction. In 1/1000 of CPU bandwidth.

Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.

Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.

Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.

See Also:
run, run, setDaemon, checkAccess, setServiceFraction

Methods

- **setServiceFraction**

```
public final void setServiceFraction(int newServiceFraction)
```

Changes the service fraction reservation of this thread.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Than, the requested new service fraction is checked against the maximum allowed value for any single thread. If this thread's adjusted service service would be greater than the maximum, an `InsufficientCpuResourceException` is thrown.

Otherwise, the service fraction for this thread is set to the specified new value.

Parameters:
newServiceFraction - New service fraction for this thread. Specified in 1/1000s of total CPU bandwidth.

Throws: IllegalArgumentException
If the specified service fraction is not in the range `MIN_SERVICEFRACTION` to `MAX_SERVICEFRACTION`.

Throws: InsufficientCpuResourceException
If the specified service fraction is greated than the maximum allowed.

See Also:
checkAccess, getServiceFraction, MIN_SERVICEFRACTION, MAX_SERVICEFRACTION

- **getServiceFraction**

```
public final int getServiceFraction()
```

Returns this thread's service fraction.

Returns:
this thread's service fraction.

See Also:
setServiceFraction

- **setReservedServiceFraction**

```
public final void setReservedServiceFraction(int newReserve)
```

Changes the reserved service fraction.

The reserved service fraction is a portion of the available CPU bandwidth that is set aside, and not allocated to any thread. The default is 10. The purpose of this reservation is to account for system activities such as signal processing.

The specified new reserve value must not exceed the current available service fraction.

Parameters:
newReserve - The portion of total CPU bandwidth to be reserved. Specified in 1/1000s of total CPU bandwidth.
Throws: SecurityException
If the current thread is not authorized to change this reservation.
Throws: IllegalArgumentException
If the specified service fraction reserve is greater than the current available CPU bandwidth.
See Also:
getReservedServiceFraction

- **getReservedServiceFraction**

```
public static final int getReservedServiceFraction()
```

Get the reserved service fraction.

The reserved service fraction is a portion of the available CPU bandwidth that is set aside, and not allocated to any thread. The default is 10. The purpose of this reservation is to account for system activities such as signal processing.

See Also:
setReservedServiceFraction

- **getAvailableServiceFraction**

```
public static final int getAvailableServiceFraction()
```

Returns currently available CPU bandwidth. Specified in 1/1000s of total CPU bandwidth.

- **getAllocatedServiceFraction**

```
public static final int getAllocatedServiceFraction()
```

Returns total CPU bandwidth that has been allocated to all threads. Specified in 1/1000s of total CPU bandwidth.

- **getGCServiceFraction**

```
public static final native int getGCServiceFraction()
```

Returns CPU bandwidth that has been allocated to the GC thread. Specified in 1/1000s of total CPU bandwidth.

See Also:
setGCServiceFraction

- **setGCServiceFraction**

```
public static final void setGCServiceFraction(int gcServiceFraction)
```

Changes the service fraction assignment for the GC thread.

The specified new reserve value must not exceed the maximum allowed for a signal thread.

Parameters:
gcServiceFraction - The portion of total CPU bandwidth to be reserved for the GC thread.
Throws: SecurityException
If the current thread is not authorized to change this reservation.
Throws: IllegalArgumentException
If the specified service fraction reserve is greater than the maximum for a single thread.
See Also:
getGCServiceFraction

- **getFinalizerServiceFraction**

```
public static final native int getFinalizerServiceFraction()
```

Returns CPU bandwidth that has been allocated to the Finalizer thread. Specified in 1/1000s of total CPU bandwidth.

See Also:
setFinalizerServiceFraction

- **setFinalizerServiceFraction**

```
public static final void setFinalizerServiceFraction(int finalizerSF)
```

Changes the service fraction assignment for the Finalizer thread.

The specified new reserve value must not exceed the maximum allowed for a signal thread.

Parameters:
finalizerSF - The portion of total CPU bandwidth to be reserved for the Finalizer thread.
Throws: SecurityException
If the current thread is not authorized to change this reservation.
Throws: IllegalArgumentException
If the specified service fraction reserve is greater than the maximum for a single thread.
See Also:
getFinalizerServiceFraction

- **systemThreadLoaded**

```
public static final native boolean systemThreadLoaded()
```

Returns whether the system thread is loaded. The system thread is only loaded when the JVM is started in the "real-time" scheduling class of Solaris. It is responsible for yielding CPU to the rest of the system. Its service fraction allotment is in effect the CPU bandwidth assigned to the operating system for all activities other than our virtual machine.

See Also:
getSystemTServiceFraction, setSystemTServiceFraction

- **getSystemTServiceFraction**

```
public static final native int getSystemTServiceFraction()
```

Returns CPU bandwidth that has been allocated to the system thread, if it is loaded. Otherwise returns 0.

See Also:
systemThreadLoaded, setSystemTServiceFraction

- **setSystemTServiceFraction**

```
public static final void setSystemTServiceFraction(int systemTSF)
```

Changes the service fraction assignment for the system thread.

The specified new reserve value must not exceed the maximum allowed for a signal thread.

Parameters:
systemTSF - The portion of total CPU bandwidth to be reserved for the system thread.
Throws: SecurityException
If the current thread is not authorized to change this reservation.
Throws: IllegalArgumentException
If the specified service fraction reserve is greater than the maximum for a single thread.
See Also:
systemThreadLoaded, getSystemTServiceFraction

- **start**

```
public native synchronized void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

Throws: IllegalThreadStateException
if the thread was already started.
See Also:
run, stop

- **toString**

```
public String toString()
```

Returns a string representation of this thread, including the thread's name, service fraction, and thread group.

Returns:
a string representation of this thread.
Overrides:
toString in class Object

- **currentThread**

```
public static native QThread currentThread()
```

Returns a reference to the currently executing thread object.

Returns:
the currently executing thread.

- **yield**

```
public static native void yield()
```

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

- **sleep**

```
public static native void sleep(long millis)
      throws InterruptedException
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.

Parameters:
millis - the length of time to sleep in milliseconds.
Throws: InterruptedException
if another thread has interrupted this thread.
See Also:
java.lang.Object.notify

- **sleep**

```
public static void sleep(long millis,
                         int nanos) throws InterruptedException
```

Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds. The thread does not lose ownership of any monitors.

Parameters:
millis - the length of time to sleep in milliseconds.
nanos - 0-999999 additional nanoseconds to sleep.
Throws: IllegalArgumentException
if the value of millis is negative or the value of nanos is not in the range 0-999999.
Throws: InterruptedException
if another thread has interrupted this thread.
See Also:
java.lang.Object.notify

- **run**

```
public void run()
```

If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.

Subclasses of `Thread` should override this method.

See Also:
start, stop, QThread, run

- **stop**

```
public final void stop()
```

Forces the thread to stop executing.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

The thread represented by this thread is forced to stop whatever it is doing abnormally and to throw a newly created `ThreadDeath` object as an exception.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

An application should not normally try to catch `ThreadDeath` unless it must do some extraordinary cleanup operation (note that the throwing of `ThreadDeath` causes `finally` clauses of `try` statements to be executed before the thread officially dies). If a `catch` clause catches a `ThreadDeath` object, it is important to rethrow the object so that the thread actually dies.

The top-level error handler that reacts to otherwise uncaught exceptions does not print out a message or otherwise notify the application if the uncaught exception is an instance of `ThreadDeath`.

Throws: SecurityException
if the current thread cannot modify this thread.
See Also:
checkAccess, run, start, ThreadDeath, uncaughtException

- **stop**

```
public final synchronized void stop(Throwable o)
```

Forces the thread to stop executing.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the argument `obj` is null, a `NullPointerException` is thrown (in the current thread).

The thread represented by this thread is forced to complete whatever it is doing abnormally and to throw the `Throwable` object `obj` as an exception. This is an unusual action to take; normally, the `stop` method that takes no arguments should be used.

It is permitted to stop a thread that has not yet been started. If the thread is eventually started, it immediately terminates.

Parameters:
obj - the Throwable object to be thrown.
Throws: SecurityException
if the current thread cannot modify this thread.
See Also:
checkAccess, run, start, stop

- **interrupt**

```
public void interrupt()
```

Interrupts this thread.

- **interrupted**

```
public static boolean interrupted()
```

Tests if the current thread has been interrupted. Note that `interrupted` is a static method, while `isInterrupted` is called on the current `Thread` instance.

Returns:
`true` if the current thread has been interrupted; `false` otherwise.
See Also:
isInterrupted

- **isInterrupted**

```
public boolean isInterrupted()
```

Tests if the current thread has been interrupted. Note that `isInterrupted` is called on the current `Thread` instance; by contrast, `interrupted` is a static method.

Returns:
`true` if this thread has been interrupted; `false` otherwise.
See Also:
interrupted

- **destroy**

```
public void destroy()
```

Destroys this thread, without any cleanup. Any monitors it has locked remain locked. (This method is not implemented.

- **isAlive**

```
public final native boolean isAlive()
```

Tests if this thread is alive. A thread is alive if it has been started and has not yet died.

Returns:
`true` if this thread is alive; `false` otherwise.

- **suspend**

```
public final void suspend()
```

Suspends this thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException` (in the current thread).

If the thread is alive, it is suspended and makes no further progress unless and until it is resumed.

Throws: SecurityException
if the current thread cannot modify this thread.
See Also:
checkAccess, isAlive

- **resume**

```
public final void resume()
```

Resumes a suspended thread.

First, the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`(in the current thread).

If the thread is alive but suspended, it is resumed and is permitted to make progress in its execution.

Throws: SecurityException
if the current thread cannot modify this thread.
See Also:
checkAccess, isAlive

- **setName**

```
public final void setName(String name)
```

Changes the name of this thread to be equal to the argument name.

First the `checkAccess` method of this thread is called with no arguments. This may result in throwing a `SecurityException`.

Parameters:
name - the new name for this thread.
Throws: SecurityException
if the current thread cannot modify this thread.
See Also:
checkAccess, getName

- **getName**

```
public final String getName()
```

Returns this thread's name.

Returns:
this thread's name.
See Also:
setName

- **getThreadGroup**

```
public final ThreadGroup getThreadGroup()
```

Returns this thread's thread group.

Returns:
this thread's thread group.

- **activeCount**

```
public static int activeCount()
```

Returns the current number of active threads in this thread group.

Returns:
the current number of threads in this thread's thread group.

- **enumerate**

```
public static int enumerate(Thread tarray[])
```

Copies into the specified array every active thread in this thread group and its subgroups. This method simply calls the `enumerate` method of this thread's thread group with the array argument.

Returns:
the number of threads put into the array.
See Also:
java.lang.ThreadGroup.enumerate

- **countStackFrames**

```
public native int countStackFrames()
```

Counts the number of stack frames in this thread. The thread must be suspended.

Returns:
the number of stack frames in this thread.
Throws: IllegalThreadStateException
if this thread is not suspended.

- **join**

```
public final synchronized void join(long millis)
        throws InterruptedException
```

Waits at most `millis` milliseconds for this thread to die. A timeout of `0` means to wait forever.

Parameters:
millis - the time to wait in milliseconds.
Throws: InterruptedException
if another thread has interrupted the current thread.

- **join**

```
public final synchronized void join(long millis,
                                    int nanos) throws InterruptedException
```

Waits at most `millis` milliseconds plus `nanos` nanoseconds for this thread to die.

Parameters:
millis - the time to wait in milliseconds.
nanos - 0-999999 additional nanoseconds to wait.
Throws: IllegalArgumentException
if the value of millis is negative the value of nanos is not in the range 0-999999.
Throws: InterruptedException
if another thread has interrupted the current thread.

- **join**

```
public final void join() throws InterruptedException
```

Waits for this thread to die.

Throws: InterruptedException
if another thread has interrupted the current thread.

- **dumpStack**

```
public static void dumpStack()
```

Prints a stack trace of the current thread. This method is used only for debugging.

See Also:
java.lang.Throwable.printStackTrace

- **setDaemon**

```
public final void setDaemon(boolean on)
```

Marks this thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.

This method must be called before the thread is started.

Parameters:
on - if `true`, marks this thread as a daemon thread.
Throws: IllegalThreadStateException
if this thread is active.
See Also:
isDaemon

- **isDaemon**

```
public final boolean isDaemon()
```

Tests if this thread is a daemon thread.

Returns:
`true` if this thread is a daemon thread; `false` otherwise.
See Also:
setDaemon

- **checkAccess**

```
public void checkAccess()
```

Determines if the currently running thread has permission to modify this thread.

If there is a security manager, its checkAccess method is called with this thread as its argument. This may result in throwing a SecurityException.

Throws: SecurityException
if the current thread is not allowed to access this thread.
See Also:
java.lang.ThreadGroup.checkAccess

Appendix C The `QThreadGroup` API

Class panda.threads.QThreadGroup

```
java.lang.Object
   |
   +----java.lang.ThreadGroup
           |
           +----panda.threads.QThreadGroup
```

```
public class QThreadGroup
extends ThreadGroup
```

A *QThreadGroup* is a thread group that supports CPU bandwidth partitioning. It is a sub-class of `java.lang.ThreadGroup`. As such, it supports all operations supported by the Java ThreadGroup, except the priority manipulation methods. This class implements the basic policy that the aggregate bandwidth allotment of all child threads of a `QThreadGroup` must not exceed the allotment to the `QThreadGroup` itself. Use of this class is optional. In other words, QThreads are not forced to be members of some `QThreadGroup`. However, they are always members of some instances of `java.lang.ThreadGroup`.

A *QThreadGroup* may be a member of either a `java.lang.ThreadGroup` or another `QThreadGroup`. If it is a member of an ordinary ThreadGroup, then it is the root of a resource allocation tree. Otherwise, its resource allocation is deducted from its parent.

This class uses the same security mechanism as implemented by `java.lang.ThreadGroup`.

See Also:
java.lang.ThreadGroup, QThread

Constructor Index

- **QThreadGroup(QThreadGroup, String, int)**
  Creates a QThreadGroup that is the child of another QThreadGroup.
- **QThreadGroup(ThreadGroup, int)**
  Creates a QThreadGroup that is the child of an ordinary java ThreadGroup.

Method Index

- **add**(QThread)
  Adds the specified QThread to this group.

- **destroyG()**
  Destroys this QThreadGroup and all of its subgroups.
- **getAllocatedServiceFraction()**
  Returns the amount of service fractions that has been allocated.
- **getCurrentAvailableServiceFraction()**
  Returns the amount of service fractions that is available now.
- **getTotalServiceFraction()**
  Returns the total amount of service fractions allocated to this group.
- **remove(QThread)**
  Removes the specified Thread from this group.
- **setTotalServiceFraction(int)**
  Adjust the total service fraction for this group.
- **toString()**
  Returns a string representation of this Thread group.

Constructors

- **QThreadGroup**

```
public QThreadGroup(ThreadGroup parent, int availableResource)
```

Creates a QThreadGroup that is the child of an ordinary java ThreadGroup. This creates a resource allocation tree.

Parameters:
parent - the parent thread group.
availableResource - resource allocated to this QThread group. Expressed in terms of service fractions.
Throws: NullPointerException
if the thread group argument is null.
Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.
See Also:
SecurityException, checkAccess, QThread

- **QThreadGroup**

```
public QThreadGroup(QThreadGroup parent, String name,
                    int availableResource)
```

Creates a QThreadGroup that is the child of another QThreadGroup.

Parameters:
parent - the parent QThread group.
name - the name of the new QThread group.
availableResource - resource allocated to this QThread group. Expressed in terms of service fractions.
Throws: NullPointerException
if the thread group argument is null.
Throws: SecurityException
if the current thread cannot create a thread in the specified thread group.

Throws: InsufficientCpuResourceException
If the specified service fraction is greated than what is available in the parent group.
See Also:
SecurityException, checkAccess, QThread

Methods

- **getTotalServiceFraction**

```
public final int getTotalServiceFraction()
```

Returns the total amount of service fractions allocated to this group. The combined service fractions of the QThreads and QThreadGroups that are part of this group cannot be higher than this amount.
Returns:
the total amount of service fractions allocated to this group.

- **getCurrentAvailableServiceFraction**

```
public final int getCurrentAvailableServiceFraction()
```

Returns the amount of service fractions that is available now. The combined service fractions of the QThreads and QThreadGroups that are part of this group cannot be higher than this amount.
Returns:
the current available service fractions for this group.

- **getAllocatedServiceFraction**

```
public final int getAllocatedServiceFraction()
```

Returns the amount of service fractions that has been allocated. The combined service fractions of the QThreads and QThreadGroups that are part of this group cannot be higher than this amount.
Returns:
the current allocated service fractions for this group.

- **setTotalServiceFraction**

```
public final void setTotalServiceFraction(int new_sf)
```

Adjust the total service fraction for this group.

First, the `checkAccess` method of this thread group is called with no arguments; this may result in a security exception.

If the new service fraction is smaller than what is currently allocated, an IllegalArgumentException is thrown.

Parameters:
new_sf - the new service fraction of this group.
Throws: SecurityException
if the current thread cannot modify this QThread group.

Throws: IllegalArgumentException
If the specified service fraction is smaller than what has been allocated to the child QThreads and QThreadGroups.
See Also:
SecurityException, checkAccess, QThread

- **destroyG**

```
public final void destroyG()
```

Destroys this QThreadGroup and all of its subgroups. This QThread group must be empty, indicating that all threads that had been in this thread group have since stopped.

Throws: IllegalThreadStateException
if the thread group is not empty or if the thread group has already been destroyed.
Throws: SecurityException
if the current thread cannot modify this thread group.

- **add**

```
public void add(QThread t)
```

Adds the specified QThread to this group.

Parameters:
t - the Thread to be added
Throws: IllegalThreadStateException
If this group has been destroyed.
Throws: InsufficientCpuResourceException
If the specified service fraction of t is greater than what is available in this group.

- **remove**

```
public void remove(QThread t)
```

Removes the specified Thread from this group. It is a no-op if the specified thread is not a member of this group.

Parameters:
t - the Thread to be removed

- **toString**

```
public String toString()
```

Returns a string representation of this Thread group.

Returns:
a string representation of this thread group.
Overrides:
toString in class ThreadGroup

Appendix D The Runner Class

```
public class runner implements Runnable {

    private static int id_pool = 1;
    private int  ID     = 0;
    private int  loops  = 10000;
    private int  period = 10000;
    private long tick   = 0;
    private boolean report = true;

    public runner() {
        this.ID = id_pool++;
    }

    public runner(boolean report) {
        this();
        this.report = report;
    }

    public runner(int loops, int period, boolean report) {
        this();
        this.loops  = loops;
        this.period = period;
        this.report = report;
    }

    public void run() {
        for (int i=loops; i>0; i--) {
            for (int j=period; j>0 ; j--) {
                long tmp = (loops * period) / 1000 + tick;
                 tick++;
            }
            if (report) {
                report();
            }
        }
    }

    public void report () {
        System.out.println("Runner #" + ID + ", tick = " + tick);
    }

    public long getTick () {
        return tick;
    }
}
```

Appendix E The RaceTest Application

```
import java.util.Date;
import panda.threads.*;
/**
  * System performance tester.
  * <P>
  * Tests the system performance using the runner class.
  * This program starts a specified number of runners as QThreads
  * with specified service fraction, and collect statistics from
  * the runners.
  *
  * Syntax: java RaceTest numRunner weight+ [loops [period [report]]]
  *
  * numRunner -- the number of runners to start.
  * weight+   -- the service fractions assigned to the runners.
  *              There should be one "weight" for each runner.
  * loops     -- number of loops to run for each runner. Optional.
  *              The default is 1000.
  * period    -- number of inner loops for each loop. Optional.
  *              default is 100000.
  * report    -- whether the runners themself should report progress.
  *              The defualt is no. If anything is supplied on this
  *              position, the variable will be set to true.
  *
  * This class assumes the enhanced JVM.
  */
public class RaceTest extends Thread {

  private static int NUMRUNNERS;
  private static int[] service_fractions;
  private static int loops  = 1000;
  private static int period = 100000;
  private static boolean report = false;

  private static void usage () {
    System.out.println
      ("Usage: RaceTest numRunner weight+ [loops [period [report]]]");
    System.out.println ("");
  }

  public static void main(String[] args) {

    int argc = 0;
    // get the number of runners and their service fraction
    if (args.length > 1) {
      NUMRUNNERS = Integer.parseInt(args[argc++]);
      if (args.length > NUMRUNNERS) { //should be at least NUMRUNNERS+1
        service_fractions = new int[NUMRUNNERS];
        for (int i=0; i<NUMRUNNERS; i++) {
          service_fractions[i] = Integer.parseInt(args[argc++]);
        }
```

```
    } else {
      System.out.println
      ("Must specify service fractions for all runners.");
      usage();
      System.exit(-1);
    }
  } else {
    System.out.println ("Insufficient number of arguments.");
    usage();
    System.exit(-1);
  }

  // get the optional period and loops specification
  if (args.length > (++argc)) { //there are more arguments
    loops = Integer.parseInt(args[argc]);
    if (args.length > (++argc)) {
      period = Integer.parseInt(args[argc]);
      if (args.length > (++argc)) {
        report = true;
      }
    }
  }
  RaceTest observer = new RaceTest();
  //observer.setPriority(Thread.MAX_PRIORITY);
  observer.start();
}

// The observer thread. Monitors the runner thread.
public void run () {

  QThread[] threads = new QThread[NUMRUNNERS];
  runner[]  runners = new runner[NUMRUNNERS];

  // setup the runners
  for (int i = 0; i < NUMRUNNERS; i++) {
    runners[i] = new runner((loops*(10/NUMRUNNERS)),period,report);
    // we manipulate the number of loops for the runner to make sure
    // that it will last longer than what we needed to collect date.
    threads[i] = new QThread(runners[i], service_fractions[i] );
  }
  System.out.println("Currently available service fraction = " +
                     QThread.getAvailableServiceFraction());
  System.out.println("Currently allocated service fraction = " +
                     QThread.getAllocatedServiceFraction());
  System.out.println("");
 System.out.println("Running tests .... (test takes around " + loops/2
                     + " seconds.)");
  System.out.println("");

  long test_start = java.lang.System.currentTimeMillis();
  for (int i = 0; i < NUMRUNNERS; i++)
    threads[i].start();
```

```
    // memory bank for storing statistics.
    long[][] stats = new long [NUMRUNNERS+1][loops];
    // the last colum in the above memory bank is for a time stamp
    // collect statistics
    int loops_completed = 0;
    boolean not_done = true;
    do {
      try { Thread.sleep (500); }
      catch (InterruptedException e) {};

      for (int i = 0; i < NUMRUNNERS; i++)
        stats[i][loops_completed] = runners[i].getTick();

      // save a time stamp as well
      stats[NUMRUNNERS][loops_completed]
        = java.lang.System.currentTimeMillis();

      not_done = (++loops_completed < loops);
      // stop the collection as soon as one of the runner finishes
      // or when "loops" number of loops has been completed.
      for (int i = 0; i < NUMRUNNERS && not_done; i++)
        not_done = not_done && threads[i].isAlive();
    } while (not_done);

    long test_end = java.lang.System.currentTimeMillis();
    // output the stats
    System.out.println("Total time to completion: " +
                       (test_end - test_start) / 1000 +
                       " seconds.");
    System.out.println("");

    for (int i = 1; i <= NUMRUNNERS; i++)
      System.out.print("\tRunner " + i);
    System.out.println("\tTime Stamp");

    for (int j = 0; j < loops_completed; j++) {
      for (int i = 0; i <= NUMRUNNERS; i++) {
        System.out.print("    \t" + stats[i][j]);
      }
      System.out.println("");
    }
    java.lang.Runtime.getRuntime().exit(0);
  }
}
```

Appendix F The GenLoad Shell Script

```
#! /bin/csh -f
#
# generate a greedy system load
#
# Copyright (C) 1998,  J. C. Pang
#

# only run this on spring
if (${HOST} != "spring") then
    echo You can only run this script on host Spring
    exit 1;
endif

# default lock file
set LOCK_FILE='/tmp/GenLoad.lock'
# did the user specify a different lock file?
if ( ${#argv} > 0 ) then
    set LOCK_FILE=${argv[1]}
endif

# make sure we don't run more than one copies.
if ( -e ${LOCK_FILE} ) then
    echo Another copy of $0 is already running.
    exit 1;
endif
# create the lock file
touch ${LOCK_FILE}

# run a large program until our lock file becomes unwritable
while ( -w ${LOCK_FILE} )
    echo $0 running >> ${LOCK_FILE}
    echo `uptime`   >> ${LOCK_FILE}
    echo ""         >> ${LOCK_FILE}
    ( cd /linux/cpang/JDK ;     \
      tar cf - jdk-MTRLS/CHANGES jdk-MTRLS/COPYRIGHT jdk-MTRLS/LICENSE \
               jdk-MTRLS/README  jdk-MTRLS/bin       jdk-MTRLS/include \
               jdk-MTRLS/lib ) \
    | gzip -9 >> /dev/null
end
# We are done when someone changes the write permission on our lock file.

# Last log entry
/bin/chmod u+w ${LOCK_FILE}
echo GenLoad exited `date`
```

Appendix G The GenStats Script

```
#! /bin/csh -f
#
# Generate statistics
#
# Copyright (C) 1998,  J. C. Pang

# only run this script on spring
if (${HOST} != "spring") then
    echo You can only run this script on host Spring
    exit 1;
endif

# our environment
set TEST_HOME="~/JVM"
set SRC_DIR='tests'
set BIN_DIR='bin'

# the various programs
set JAVAC='/home/cpang/JDK/jdk/bin/javac'
set LOAD_RT='/public/bin/super priocntl -e -c RT -p 15'
set MTRLS_JAVA='/home/cpang/JDK/jdk/bin/java'
set STD_JAVA='/home/cpang/JDK/jdk-std/bin/java -ts20'
set SYS_THREAD='-sw15'

# the test program for the standard Java VM
set CMD_STD='RaceTestStd'
set S_STD='1 5'
set Q_STD='4 5 5 5 5'
set T_STD='10 5 5 5 5 5 5 5 5 5 5'

# the test program for enhanced JVM
set CMD='RaceTest'
# various arguments for the test program
set S='1 15'
set SM='1 800'
set Q='4 15 15 15 15'
set QM='4 200 200 200 200'
set QP='4 120 60 30 15'
set QPM='4 500 250 100 50'
set T='10 15 15 15 15 15 15 15 15 15 15'
set TM='10 80 80 80 80 80 80 80 80 80 80'

# these arrays controls the execution
set ARGS=("${S}" "${SM}" "${Q}" "${QM}" "${QP}" "${QPM}" "${T}" "${TM}")
set LOGFILE_EXT=( S SM  Q QM QP QPM T TM )
set     INDICES=( 1  2  3  4  5  6  7  8 )

# default test setup
set RECOMPILE='true'
set NOLOAD='false'
```

```
set LOG_DIR='stats'
set LOGFILE_PREFIX='RaceTest.stats'

# process the command line arguments
while (${#argv} != 0)
  switch (${argv[1]})
  case -norecompile:
    set RECOMPILE='false'
  breaksw
  case -noload:
    set NOLOAD='true'
  breaksw
  case -statsdir:
    shift argv
    set LOG_DIR=${argv[1]}
  breaksw
  case -help:
    echo "Usage: $0 [-help] | [ [-norecompile] [-noload] \
         [-statsdir <stats dir>] ]"
    echo ""
    echo "The defaults are: "
    echo "    recompile java source, "
    echo "    generate load, and "
    echo "    store statistics in ${TEST_HOME}/${LOG_DIR}"
    echo ""
    echo "Note that <stats dir> must be relative to ${TEST_HOME}."
    echo "IMPORTANT: all previous information in the stats directory "
    echo "will be lost."
    exit 0
  breaksw
  default:
    echo Invalid command line switch: ${argv[1-]}
    exit 1
  breaksw
  endsw
    # advance to the next argument
    shift argv
end

# now we can begin
echo Started on `date`
set OLD_DIR=`pwd`
chdir ${TEST_HOME}/${SRC_DIR}

# make sure all our classes are up to date
if ( ${RECOMPILE} == 'true' ) then
    ${JAVAC} *.java
endif

# clean the stats directory
/usr/bin/rm -rf ${TEST_HOME}/${LOG_DIR}
mkdir -p ${TEST_HOME}/${LOG_DIR}
```

```
# start the load generator if necessary
if ( ${NOLOAD} != 'true' ) then
    set GENLOAD_LOCK="${TEST_HOME}/${LOG_DIR}/GenLoad.lock"
    ${TEST_HOME}/${BIN_DIR}/GenLoad.sh ${GENLOAD_LOCK} &
endif

# Now we can start to generate the statistics

# First, standard Java VM
( echo ${STD_JAVA} ${CMD_STD} ${S_STD} ; echo "" ; \
       ${STD_JAVA} ${CMD_STD} ${S_STD} ) \
 > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.Std.S
( echo ${STD_JAVA} ${CMD_STD} ${Q_STD} ; echo "" ; \
       ${STD_JAVA} ${CMD_STD} ${Q_STD} ) \
 > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.Std.Q
( echo ${STD_JAVA} ${CMD_STD} ${T_STD} ; echo "" ; \
       ${STD_JAVA} ${CMD_STD} ${T_STD} ) \
 > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.Std.T
# now run in TS class without the system thread
foreach I ( ${INDICES} )
    ( echo ${MTRLS_JAVA} ${CMD} ${ARGS[${I}]} ; \
      echo "" ; \
      ${MTRLS_JAVA} ${CMD} ${ARGS[${I}]} ) \
    > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.${LOGFILE_EXT[${I}]}.T
end
# run in RT class with the system thread loaded
foreach I ( ${INDICES} )
    ( echo ${LOAD_RT} ${MTRLS_JAVA} ${SYS_THREAD} ${CMD} ${ARGS[${I}]}; \
      echo "" ; \
      ${LOAD_RT} ${MTRLS_JAVA} ${SYS_THREAD} ${CMD} ${ARGS[${I}]} ) \
    > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.${LOGFILE_EXT[${I}]}.S
end
# now run in RT class without loading the system thread
foreach I ( ${INDICES} )
    ( echo ${LOAD_RT} ${MTRLS_JAVA} ${CMD} ${ARGS[${I}]} ; \
      echo "" ; \
      ${LOAD_RT} ${MTRLS_JAVA} ${SYS_THREAD} ${CMD} ${ARGS[${I}]} ) \
    > ${TEST_HOME}/${LOG_DIR}/${LOGFILE_PREFIX}.${LOGFILE_EXT[${I}]}.R
end

# All Done.

# stop the load generator if we ever started it
if ( ${NOLOAD} != 'true' ) then
    /bin/chmod a-w ${GENLOAD_LOCK}
endif

# Finish it up
chdir ${OLD_DIR}
echo Finished at `date`
echo ""
```

Vita

Surname: Pang Given Names: James Chun

Place of Birth: Shanghai, Zhong Guo.

Education Institutions Attended:

University of Victoria	1992 —1999
Camosun College	1991 —1992
University of Chemical Technology at East of Zhong Guo	1986 —1988

Degrees:

B.Sc. (Co-op)	University of Victoria	1996

Honours and Awards:

NSERC Post Graduate Scholarship	1996 —1998
The University of Victoria President's Research Scholarship	1996 —1998
BC Advanced System Institute Graduate Scholarship	1996
The University of Victoria President's Scholarship	1995
The University of Victoria Faculty Scholarship	1995
The Woods Trust Scholarship	1995
NSERC Industrial Undergraduate Student Research Award	1995
Academic Excellence Award of the UCT-EZG	1988
First Class Academic Excellence Award (UCT-EZG)	1987

Publications:

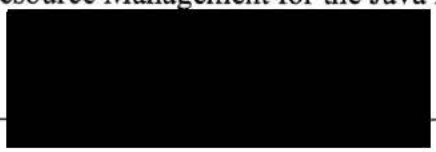
M. Horie, J.C. Pang, E. Manning and G.C. Shoja. Designing Meta-Interfaces for Object-Oriented Operating Systems. In *Proceedings of IEEE PACRIM'97*. August 1997, Victoria, BC, Canada.

M. Horie, J.C. Pang, E. Manning and G.C. Shoja. Using Meta-Interfaces to Support Secure Dynamic System Reconfiguration. In *Proceedings of the 4th International Conference on Configurable Distributed Systems*. May 4-6, 1998, Annapolis, Maryland, USA.

Partial Copyright Licence

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library from any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: CPU Resource Management for the Java Platform.

Author 

James Chun Pang

Date Mar. 26, 1999

March 1999