Real-Time Java™ Programming

With Java RTS

Eric J. Bruno ▪ Greg Bollella

Library of Congress Cataloging-in-Publication Data
Bruno, Eric J., 1969-
Real-time Java programming with Java RTS / Eric J. Bruno, Greg Bollella.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-13-714298-9 (pbk. : alk. paper) 1. Java (Computer program language) 2. Real-time programming. 3. Application program interfaces (Computer software) I. Bollella, Gregory. II. Title.

QA76.73.J38B823 2009
005.2'739—dc22

2009015739

ISBN-13: 978-0-13-714298-9
ISBN-10: 0-13-714298-6
Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, May 2009

Preface

"To achieve great things, two things are needed: a plan, and not quite enough time."

—Leonard Bernstein

As this book is being written, the age of real-time programming, especially with Java, has only begun. In the near future, however, we predict that real-time Java—namely, the Java Real-Time System (Java RTS) from Sun Microsystems—will be used predominantly in real-time application areas, such as software in the financial world, critical control systems, manufacturing lines, military and other government systems, and so on. This prediction arrives in the shadow of a time when Java was once considered "too slow" to be used in the development of mission-critical and other enterprise systems.

However, just as Java had proven quickly to the world that it could perform well enough for even the most demanding enterprise systems, Java RTS is actively proving itself in the real-time space. Gone is the *necessity* for complicated, specialized, real-time languages and operating environments; Java RTS brings all of the productivity and familiarity of the Java language to systems with real-time requirements. Literally decades of research, knowledge, and advancement found in the real-time discipline is now at the fingertips of every Java developer, requiring little more understanding than that of a new library of classes.

It was in the late '90s that Greg Bollella had the idea for a real-time version of Java while in Chicago on a business trip. Soon after, JSR-001 was started and the specification was in its early stages. The specification was led by Greg, and involved many of the best minds involved in real-time scheduling theory, microprocessor design, embedded systems design, and language design. In the end, a specification was finalized that defines how Java is to behave in the real-time space, and Java RTS was built to conform to it and make it a reality.

Defining "Real-Time"

Although the first chapter of this book discusses and defines real-time systems thoroughly, it's best to set the stage early and agree upon a precise definition. With *real-time programming*, the overall goal is to ensure that a system performs its tasks, in response to real-world events, before a defined deadline. Regardless of whether that deadline is measured in microseconds or days, as long as the task is required to complete before that deadline, the system is considered real-time. That simple definition is the foundation for an entire discipline in computer science, with years of research and development within both academia and industry.

To put it differently, the time delay from when a real-world event occurs (such as an object passing over a sensor, or the arrival of a stock market data-feed tick) to the time some code finishes processing that event should be bounded. The ability to meet this deadline must be predictable and guaranteed, all the time, in order to provide the determinism needed for a real-time system.

Meeting the requirements for real-time systems can be so demanding that dedicated programming languages, operating systems, frameworks, and scheduling algorithms have been created. Distinct areas of study, sets of tools, and even entire companies have been formed to solve real-time problems. It is precisely for these reasons that the Sun Java Real-Time System has been created.

The Real-Time Specification for Java

The Java Real-Time System from Sun is a 100%-compatible implementation of the *Real-Time Specification for Java* (RTSJ) . The RTSJ is also known as JSR-001; the very first Java specification request (JSR) for which the entire Java Community Process (JCP) was created. Real-time Java is the first of many firsts, and has paved a way for not only what it was intended (real-time programming in Java), but also for the foundation of Java's growth in terms of language features through the creation of the JCP.

The RTSJ is a standard that defines how Java applications are to behave in a real-world environment. It was created by experts in many disciplines (i.e., embedded systems design, language design, operating system design, processor design, real-time scheduling theory, and so on), from many companies, from all around the world—a truly global effort. The number one goal, other than defining how real-time behavior could be achieved with Java, was to not *change* the Java language at all. Meeting this goal was important to ensure that Java, as the

thousands of developers know it today, can be used in a real-time context, with no compromise.

In order for an implementation of Java to call itself "real-time," it *must* conform to the RTSJ. Anything else is non-standard, as defined by the Java community as a whole. This ensures that developing a real-time application in Java will work with any RTSJ-compliant JVM, and will behave as defined in a real-time environment, without the need for specialized extensions or hardware. Anything else would violate the very principles Java was founded upon. Java RTS is compliant with the RTSJ, and is therefore standards-based.

This Book's Focus and Audience

Although other books have been written about the RTSJ, this book focuses on Java RTS, its APIs, and what it takes to build real-time applications in Java. The intent is to provide you with practical knowledge and examples of how to build real-time applications with Java. Wherever possible, key teachings will be presented through the use of actual working code examples, as well as visual diagrams to make complicated concepts clear.

Although Java RTS is the focus, all of the knowledge gained from this book will be RTSJ-compliant by default. An understanding of the RTSJ is not a prerequisite for this book, but it's highly recommended that you read this specification since it's the basis for the material presented herein. You can find the latest specification and related material at http://www.rtsj.org.

This book's primary audience is comprised of architects and developers (of all levels) who need to build an application with time-critical code. There are different types of developers that this book targets:

- Java developers who are currently building applications with real-time requirements with or without Java RTS
- Java developers who are interested in learning the Java RTS APIs for future projects
- Non-Java real-time application developers who wish to use Java for real-time application development
- Architects and developers who wish to use Java RTS in order to deploy on a general-purpose operating system (as opposed to a specialized operating system, platform, or language)

To each of these developers, Java RTS and the RTSJ represent freedom from specialized hardware, operating systems, and languages. When this freedom is coupled with Java's productivity gains and large developer community, Java RTS represents a wise business choice as well, and a potentially huge savings in money.

Structure of the Book

This book has been broken down into three main sections:

Part I—Real-Time Computing Concepts: The first part lays the groundwork for using Java RTS. It clearly defines the concept of real-time in the computing world, and discusses many of the theories of real-time application design and development.
This includes Chapters 1 through 4.

Part II—Inside Java RTS: The second part dives deep into the Java RTS APIs, providing ample code samples to illustrate the inner workings and use of Java RTS. You'll gain a deeper understanding of the RTSJ principals, as applied to real problems.
This includes Chapters 5 through 10.

Part III—Using Java RTS: The third and final part discusses a comprehensive case study where Java RTS has been used to solve actual real-time system problems, as well as tools used to develop and debug Java RTS applications.
This includes Chapters 11 and 12.

The chapter breakdown of this book is as follows:

Chapter 1—Real-Time for the Rest of Us: This chapter provides a thorough definition of real-time systems, and then compares them to high-performance systems and those with high-throughput requirements. Other key terms, such as predictability, jitter, latency, and determinism are defined and explored. The second half of this chapter contains a high-level discussion of real-time scheduling. Analogies, descriptions, and visuals will be used to bring the concepts down to earth; at a level where the average programmer should be comfortable.

Chapter 2—Real-Time and Java SE: This chapter explores the use of standard Java in real-time environments. Issues that arise, such as the execution of the Java garbage collector, and the just-in-time compiler, will be discussed as sources of trouble. The chapter discusses, in detail, garbage collection in Java

SE 6 and the forthcoming Java SE 7, and concludes with an overview of real-time garbage collection algorithms.

Chapter 3—The Real-Time Specification for Java: The RTSJ defines how Java should behave in the real-time space. In fact, the RTSJ was the first Java Specification Request (JSR), and a big reason why the Java Community Process (JCP) was formed. The specification, led by Greg Bollella, included experts from around the globe, from both academia and industry, and is known today as a standard for real-time Java development. This chapter provides a brief overview of the RTSJ.

Chapter 4—The Sun Java Real-Time System: This chapter begins our exploration of real-time Java with a discussion of Java RTS, Sun's product that implements the RTSJ. Reading this chapter will help you get up and running with a working Java RTS system on either Solaris or Linux.

Chapter 5—Threads, Scheduling, and the New Memory Models: As the first chapter to dive deep into the Java RTS APIs, the focus of the chapter is on the new threading models available to you, along with the different memory models introduced with the RTSJ.

Chapter 6—Synchronization: This chapter explores how thread synchronization is changed in Java RTS. It also dives into the internals to examine some of the enhancements made in the Java VM to minimize latency when synchronizing multiple threads' access to shared resources.

Chapter 7—The Real-Time Clock API: Java RTS provides support for high-resolution timers, and deterministic timer objects. In this chapter, we'll examine the real-time Clock API, and how timer objects can be created for deterministic operation.

Chapter 8—Asynchronous Events: Java RTS gives you more control over how work is scheduled in a system. This chapter examines the classes that are available to you to control how events are handled in your real-time application.

Chapter 9—Asynchronous Transfer of Control and Thread Termination: In this chapter, we'll explore the fine-grained control Java RTS provides for schedulable objects in terms of transferring control from one method to another, and terminating tasks.

Chapter 10—Inside the Real-Time Garbage Collector: This chapter dives deep into the inner workings of the real-time garbage collector, and describes how it operates. Having a solid understanding of how the RTGC works, how it

affects your system, and how it can be tuned will help you build more efficient Java RTS applications.

Chapter 11—An Equities Trading System: This chapter explores the use of Java RTS in a financial application that closely mimics those used by traders, investment banks, and exchanges in the world of finance. In the world of investing, banking, and finance, latencies due to garbage collection introduce the risk of missing important market events. The resulting delays translate directly to lost money in these markets.

Chapter 12—Java RTS Tools: Here, we discuss the tools available to develop and debug Java RTS applications.

Staying Up-to-Date

No book is ever complete, and important technologies such as Java RTS aren't static—they constantly evolve. Check the web site, http://www.ericbruno.com/realtime, for updates and extra content, as well as the complete code for the samples and many of the case studies. We'll also keep you up to date on changes to the RTSJ, as well as to Sun's implementation, Java RTS.

1

Real-Time for the Rest of Us

"Let him who would enjoy a good future waste none of his present."
—Roger Babson

THERE are many misunderstandings about what real-time is, even amongst seasoned enterprise Java developers. Some confuse it with high-performance, or fast, computing; others think of dynamic applications such as instant messaging. Neither one is necessarily an example of a real-time system. Therefore real-time does not always equal "real fast," although good performance is often desirable and achievable. In fact, real-time is often orthogonal with high-throughput systems; there's a trade-off in throughput in many cases. The best way to avoid all of this confusion is to think of it this way: application performance and throughput requirements can be solved with faster, or additional, hardware; real-time requirements, in general, cannot.

This chapter will define real-time computing, and will explain why throwing hardware at a real-time requirement will almost never do any good. We'll discuss the qualities of a real-time system, define key terms used in the discipline, and examine tools, languages, and environments available to real-time developers outside of the Java world. By the end of this chapter, you'll have a good real-time foundation to build upon.

Qualities of Real-Time Systems

The goal of a real-time system is to respond to real-world events before a measurable deadline, or within a bounded time frame. However, a real-time system is also about precision. The measured speed of a system's response to an event is

important, but what's also important is the system's ability to respond at precisely the right moment in time. Access to a high-resolution timer to perform actions on precise time periods is often a requirement. These two qualities together best define a real-time application's acceptable behavior: the ability to respond to an event before a deadline, and accurately perform periodic processing, regardless of overall system load. Before we go any further, it's important to examine the term deadline a little more closely, as well as some other terms often used in the context of real-time systems.

The term *deadline* can have one of two meanings. First, it can be a deadline *relative* to an event, such as a notification or message in some form. In this case, the system must respond to that event within a certain amount of time of receiving that event, or from when that event originally occurred. One example of a relative deadline is an elevator as it passes over a sensor indicating that it's almost at the floor it's meant to stop at. The real-time software within the elevator must respond to that event within milliseconds of passing the sensor, or it won't be able to stop at the intended floor. The occupants of an elevator that skips stops are certain to consider this an error.

Relative Deadline (D_i): the amount of time *after* a request is made that the system needs to respond.

Absolute Deadline (d_i): the precise point in time that a task must be completed, regardless of task start time, or request arrival.

Often, with an *absolute* deadline, a real-time system checks for a particular system state on a regular interval. Some examples of this are an aircraft flight control system, or a nuclear power plant's core temperature monitoring system. In both of these cases, critical data is continuously polled, such as altitude, or core temperature. Failing to monitor these values at precise points in time can cause these systems to go into a bad state with potentially catastrophic results.

Regardless of the type of deadline, relative or absolute, time is still a main component in proper system behavior. It's not enough that an elevator's software knows and responds to a floor sensor; it must do so within a deadline in order to behave correctly. Also, a flight control system must be able to move an aircraft's control surfaces at the precise time, in reaction to the most recent and accurate set of data, in order to fly the aircraft correctly (without crashing!).

For example, let's say we have a system requirement to send a response to a request within one millisecond. If the system responds within 500 microseconds every time, you may think the requirement has been met. However, if the request is delayed, outside the system under measurement, the response will not have

been sent at the right moment in time (even if it's sent within one millisecond). Remember, we're talking about "real" time here; the one-millisecond requirement applies to when the originating system sent the original request.

Figure 1-1 illustrates the problem. Here you see that the system in question has responded to the request within one millisecond, but it was at the wrong time because the request was delayed in delivery. A real-time system must adhere to the end-to-end deadline.

In a real-time system, the time delay from when a real-world event occurs (such as an object passing over a sensor, or the arrival of a stock market data-feed tick) to the time some code finishes processing that event should be reasonably bounded. The ability to meet this deadline must be predictable and guaranteed, all the time, in order to provide the determinism needed for a real-time system.

What Is "Bounded"?

When we use the term *bounded* in relation to a *bounded amount of time*, what we really imply is a reasonable amount of time for the system to respond. In other words, saying that the elevator responds to sensor events within a ten-year bounded timeframe is unreasonable. It must do so according to a time requirement that allows it to function properly. Therefore, when we use the term bounded, it's relative to the proper operation of the time-critical event we're describing.

When discussing real-time systems, the basic element of execution is often referred to as a job, or task. (For a more accurate definition of jobs and tasks in real-time systems, see the note on Jobs and Tasks in Real-Time Systems). There can be one or more tasks in a given system, and therefore tasks can either be

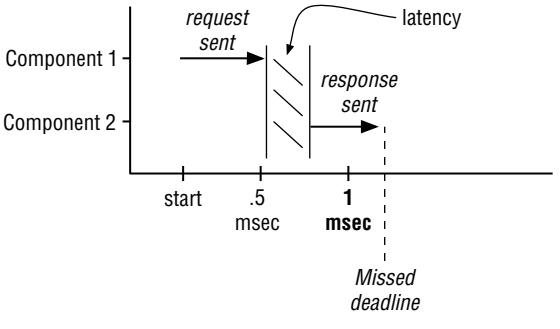

Figure 1-1 The response time was good, but the deadline was missed. This is not a real-time system.

running or waiting. On a uniprocessor machine, only one task can be running at a single point in time, as opposed to multiprocessor machines that can execute more than one task at a time.

Note: Jobs and Tasks in Real-Time Systems At this point in the discussion, it's fair to accurately define the terms *job* and *task* as used in discussions of real-time scheduling theory. Formally speaking, a job is any unit of work that can be scheduled and processed, while a task is a group of related jobs that work together to achieve some function. In this classic definition, a task contains related jobs, where those jobs have real-time constraints.

However, to keep the discussions light and simple in this book we will not distinguish between tasks and jobs; a unit of schedulable work will simply be referred to as a task. Therefore, in this book, a task represents a thread of execution and is synonymous with an OS thread.

Regardless, discussions often revolve around the arrival of a system event, or the start of task execution, which can sometimes be one and the same. To clarify, we say that a task can be in one of the three main states:

Eligible-for-Execution: the task is eligible (ready) to execute.

Executing: the task is currently executing (running) on a processor.

Blocked: the task is neither executing, nor eligible to begin executing. It's blocked for some reason, and this reason is usually stated as part of the state; i.e., blocked-for-IO, blocked-for-release-event, and so on.

With these task states defined, we can begin to discuss how tasks are scheduled in a real-time system. First, the following definitions must be stated:

Release Time (r_i): sometimes called *arrival time*, or *request time*, this is the time that a task becomes ready to execute.

Start Time (s_i): the time that a task begins executing. As stated above, these concepts may be combined for simplification in many discussions. For example, a task may be started because of a request, or it may be started as part of a predefined schedule. This book shall attempt to separate these concepts when necessary to avoid confusion.

Finish Time (f_i): the time when a task is complete.

Task Completion Time ($C_i = f_i - r_i$): the amount of time a particular task takes to complete its processing by subtracting the task's arrival time from its finish time. This is also referred to as the cost of task execution.

Lateness (L_i): the difference between the task finish time and its deadline; note that this value is negative if a task completes before its deadline, zero if it completes at its deadline, and positive if it completes after its deadline.

These terms and their associated abbreviations will be used throughout the book. To further clarify them, and to gain a better understanding of real-time systems, let's explore the factors that affect a system's ability to meet its deadlines.

Predictability and Determinism

Other important qualities of a real-time system are that of *predictability* and *determinism*. A real-time system must behave in a way that can be predicted mathematically. This refers to the system's deadline in terms of relative and absolute time. For instance, it must be mathematically predictable to determine if the amount of work to be done can be completed before a given deadline. Factors that go into this calculation are system workload, the number of CPUs (or CPU cores) available for processing, running threads in the real-time system, process and thread priorities, and the operating system scheduling algorithm.

Determinism represents the ability to ensure the execution of an application without concern that outside factors will upset the execution in unpredictable ways. In other words, the application will behave as intended in terms of functionality, performance, and response time, all of the time without question. In many respects, determinism and predictability are related, in that one results in the other. However, the important distinction is that a deterministic system puts the control of execution behavior in the hands of the application developer. Predictability is then the result of proper programming practice on a system that enables such behavior. This book will explore the statement "proper programming practice" in relation to real-time applications written in Java because using a real-time language or operating system is never enough—discipline is also required.

Another aspect of deterministic application behavior is that it's fixed, more or less. This means that unforeseen events, such as garbage collection in Java, must never upset a real-time application's ability to meet its deadlines, and hence become less predictable. A real-time system such as an anti-lock brake system, or

an airplane's flight-control system, must always be 100% deterministic and predictable or human lives may be at stake.

Many practical discussions of real-time systems and their requirements involve the terms latency and jitter. Let's examine these now, and form precise definitions that we'll use in our discussion going forward.

Identifying Latency

Much of the discussion so far has been about responding to an event before a deadline. This is certainly a requirement of a real-time system. *Latency* is a measure of the time between a particular event and a system's response to that event, and it's quite often a focus for real-time developers. Because of this, latency is often a key measurement in any real-time system. In particular, the usual focus is to minimize system latency. However, in a real-time system the true goal is simply to *normalize* latency, not minimize it. In other words, the goal is to make latency a known, reasonably small, and consistent quantity that can then be predicted. Whether the latency in question is measured in seconds, or microseconds, the fact that it can be predicted is what truly matters to real-time developers. Nonetheless, more often than not, real-time systems also include the requirement that latency be minimized and bounded, often in the sub-millisecond range.

To meet a system's real-time requirements, all sources of latency must be identified and measured. To do this, you need the support of your host system's operating system, its environment, network relationship, and programming language.

Identifying Jitter

The definition of *jitter* includes the detection of irregular variations, or unsteadiness, in some measured quantity. For example, in an electronic device, jitter often refers to a fluctuation in an electrical signal. In a real-time system, jitter is the fluctuation in latency for similar event processing. Simply measuring the latency of message processing for one event, or averaging it over many events, is not enough. For instance, if the average latency from request to response for a certain web server is 250 milliseconds, we have no insight into jitter. If we look at all of the numbers that go into the average (all of the individual request/response round-trip times) we can begin to examine it. Instead, as a real-time developer, you must look at the distribution and standard deviation of the responses over time.

Other Causes of Jitter

Jitter can also be caused by the scheduling algorithm in your real-time system. For instance, in a system with many real-time periodic tasks, a particular task can finish anywhere within its time period. If it sometimes finishes early in its period (well before the deadline), but other times it finishes just before the period end, it has jitter. In some systems, there are control algorithms that cannot tolerate this.

For example, say we have a task with a 100-millisecond period, where the execution cost is 10 milliseconds, and the worst-case latency is always less than 1 millisecond. In a complex system, this task can finish anywhere from the period boundary + 10 milliseconds processing time + 1 millisecond latency, to the period boundary + 100 milliseconds (start of next period). One solution to this problem, the one proposed by the Real-Time Specification for Java (RTSJ) in particular, is to set the deadline to 12 milliseconds.

The chart in Figure 1-2 shows a sampling of latency data for a web server's request/response round-trip time. You can see that although the average of 250 milliseconds seems pleasing, a look at the individual numbers shows that some of the responses were delivered with up to one-second latency. These "large" latency

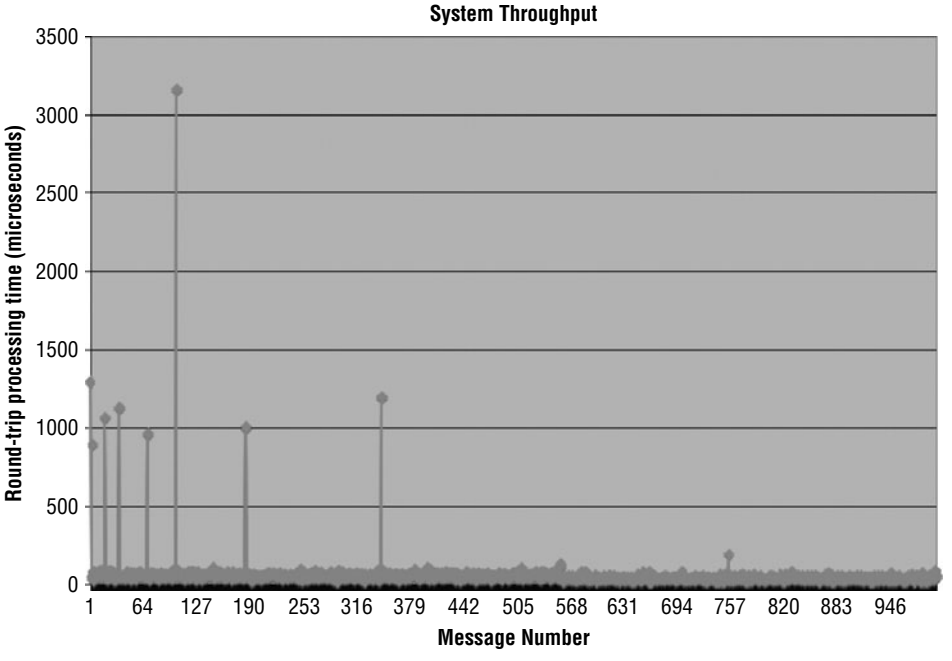

Figure 1-2 The average response time measurement of a transaction can cover up latency outliers.

responses stand out above most of the others, and are hence labeled *outliers*, since they fall outside of the normal, or even acceptable, response time range.

However, if the system being measured is simply a web application without real-time requirements, this chart should not be alarming; the outliers simply aren't important, as the average is acceptable. However, if the system were truly a real-time system, these outliers could represent disaster. In a real-time system, *every* response must be sent within a bounded amount of latency.

Hard and Soft Real-Time

In the real-time problem domain, discussions often involve the terms *hard real-time* and *soft real-time*. Contrary to what many people assume, these terms have nothing to do with the size of the deadline, or the consequence of missing that deadline. It's a common misconception that a hard real-time system has a smaller, or tighter, deadline in terms of overall time than a soft real-time system. Instead, a hard real-time system is one that cannot miss a single deadline or the system will go into an abnormal state. In other words, the correctness of the system depends not only on the responses it generates, but the time frame in which each and every response is delivered. A soft real-time system is one that may have a similar deadline in terms of time, but it instead has the tolerance to miss a deadline occasionally without generating an error condition.

For example, let's compare a hypothetical video player software application to an automated foreign exchange trading application. Both systems have real-time qualities:

- The video player must retrieve and display video frames continuously, with each frame being updated by a deadline of, say, one millisecond.
- A foreign exchange trade must be settled (moneys transferred between accounts) within exactly two days of the trade execution.

The video player has a far more constraining deadline at one millisecond compared to the two-day deadline of the trading system. However, according to our definition, the trading system qualifies as a hard real-time system, and the video player as a soft real-time system, since a missed settlement trade puts the entire trade, and trading system, into a bad state—the trade needs to be rolled back, money is lost, and a trading relationship strained. For the video player, an occasional missed deadline results in some dropped frames and a slight loss of video quality, but the overall system is still valid. However, this is still real-time since the system must not miss too many deadlines (and drop too many frames) or it, too, will be considered an error.

Additionally, the severity of the consequence of missing a deadline has nothing to do with the definition of hard versus soft. Looking closer at the video player software in the previous example, the requirement to match audio to the corresponding video stream is also a real-time requirement. In this case, many people don't consider the video player to be as critical as an anti-lock brake system, or a missile-tracking system. However, the requirement to align audio with its corresponding video is a hard real-time constraint because not doing so is considered an error condition. This shows that whereas the consequence of a misaligned video/audio stream is minimal, it's still a hard real-time constraint since the result is an error condition.

Therefore, to summarize hard and soft real-time constraints, a hard real-time system goes into a bad state when a *single* deadline is missed, whereas a soft real-time system has a more flexible deadline, and can tolerate occasional misses. In reality, it's best to avoid these terms and their distinction and instead focus on whether a system has a real-time requirement at all. If there truly is a deadline the system must respond within, then the system qualifies as real-time, and every effort should be made to ensure the deadline is met each time.

Isochronal Real-Time

In some cases, the requirement to respond to an event before a deadline is not enough; it must not be sent too early either. In many control systems, responses must be sent within a window of time after the request, and before the absolute deadline (see Figure 1-3). Such a system has an *isochronal real-time* requirement.

Although clearly distinct from a hard real-time task that needs to complete any time before its deadline, in most cases isochronal real-time tasks are simply classified as hard real-time with an additional timing constraint. This certainly makes it easier to describe the tasks in a system design. However, this added constraint does make a difference to the real-time task scheduler, which is something we'll explore later in this chapter.

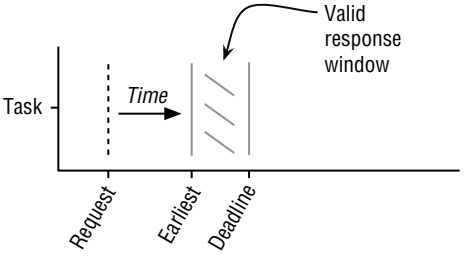

Figure 1-3 Isochronal real-time: the deadline must be met, but a response must not be sent too early, either.

Real-Time Versus Real Fast

Application or system performance is a relative measurement. When a system is said to be fast or slow, it's usually in comparison to something else. Perhaps it's an older system, a user expectation, or a comparison to an analogous real-world system. In general, performance is more of a relative measurement than a precise mathematical statement. As discussed earlier in this chapter, real-time does not necessarily equal real fast.

Instead, whereas the objective of fast computing is to minimize the average response time of a given set of tasks, the objective of real-time computing is to meet the individual time-critical requirement of each task. Consider this anecdote: there once was a man who drowned in a river with an *average* depth of 6 inches [Buttazzo05]. Of course, the key to that sentence is the use of the average depth, which implies the river is deeper at some points. A real-time system is characterized by its deadline, which is the maximum time within which it must complete its execution, not the average.

However, the *goals* of most real-time systems are to meet critical deadlines *and* to perform optimally and efficiently. For example, a system with sub-millisecond deadlines will most likely require high-performance computer hardware and software. For this reason, real-time systems programming is often associated with high-performance computing (HPC). However, it's important to remember that high-performance does not imply real-time, and vice versa.

Real-Time Versus Throughput

Another area of system performance that is often confused with real-time is that of system throughput. Throughput is often used to describe the number of requests, events, or other operations, that a software system can process in any given time frame. You often hear of software positively characterized with terms like "messages-per-second," or "requests-per-second." A system with high throughput can give its operators a false sense of security when used in a real-time context.

This is because a system with high throughput is not necessarily a real-time system, although it is often misunderstood to be. For instance, a system that supports thousands of requests per second may have some responses with up to a second of latency. Even though a majority of the requests may be handled with low latency, the existence of some messages with large latency represents outliers (those outside the normal response time). In other words, with this example, most requestors received their responses well within the one-second window. However, there

were some that waited the full second for their response. Because the degree of, and the amount of, these outliers are unpredictable, high-throughput systems are not necessarily real-time systems.

Typically, real-time systems exhibit lower average throughput than non-real-time systems. This engineering trade-off is well known and accepted in the real-time community. This is due to many factors that include trade-offs as to how tasks are scheduled and how resources are allocated. We'll explore these factors in relation to Java RTS throughout this book.

Task Completion Value

In a modern computing system, the basic element of execution is called a thread, or a task. A process is defined as an application launched by the user (either explicitly through a command, or implicitly by logging in) that contains one or more threads of execution. Regardless of how each task begins executing, the basic unit of execution is a thread. To simplify things going forward, the thread will be the focus of the discussion.

The value, or usefulness, that a task has to any running system is usually dependent upon when it gets its work done, not just that the work is done properly. Even non-real-time systems have this quality. For example, the chart in Figure 1-4 shows that the value of tasks in a non-real-time system usually increases as more of them are completed over time. This is evidence of the throughput quality of non-real-time systems, as explained in the previous section.

For a soft real-time system, the value of task completion rapidly decreases once the task's deadline passes. Although the correct answer may have been generated, it gets more and more useless as time passes (see Figure 1-5).

Contrast this to a hard real-time system, where the task has zero value after the deadline (see Figure 1-6).

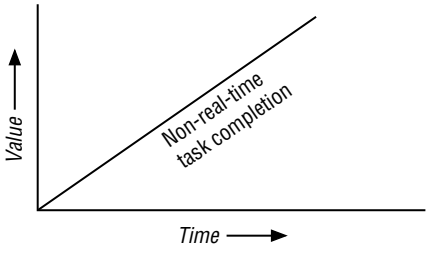

Figure 1-4 In a non-real-time system, the perceived value of task completion is directly proportional to the total number completed over time.

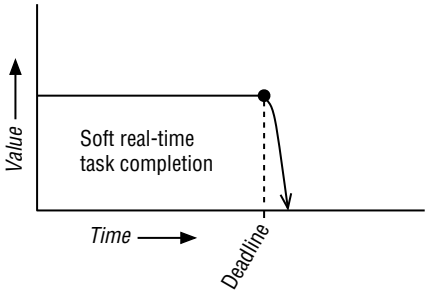

Figure 1-5 In a soft real-time system, the value of task completion, after the deadline, decays over time.

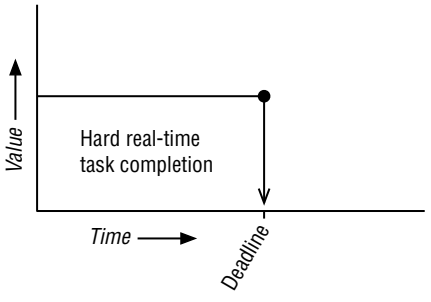

Figure 1-6 The value of task completion in a hard real-time system is zero the moment the deadline passes.

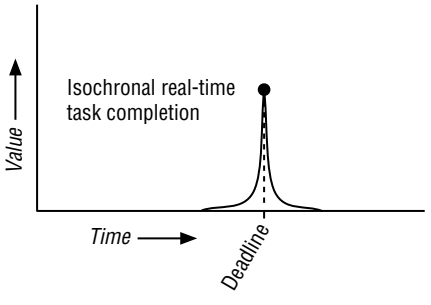

Figure 1-7 The value of task completion in a firm, isochronal, real-time system is zero if it completes early, or late.

The discussion so far assumes that task completion anytime before the deadline is acceptable. In some cases, as with firm, or isochronal, real-time systems, the task must complete before the deadline, but no earlier than a predefined value. In this case, the value of task completion before *and* after the deadline is, or quickly goes to, zero (see Figure 1-7).

Of course, these graphs are only general visual representations of task completion value in non-real-time and real-time systems; actual value is derived on a case-by-case basis. Later in this chapter, we'll examine this in more detail as task cost functions are used to calculate efficient real-time scheduling algorithms.

Real-Time Computing

Now that you have an understanding of what real-time is and what it means, it's time to expand on it. *Real-time computing* is the study and practice of building applications with real-world time-critical constraints. Real-time systems must respond to external, often physical, real-world events at a certain time, or by a deadline. A real-time system often includes both the hardware and the software in its entirety. Traditionally, real-time systems were purpose-built systems implemented for specific use; it's only recently that the real-time community has focused on general-purpose computing systems (both hardware and/or software) to solve real-time problems.

Today, the need for specialized, dedicated, hardware for real-time systems has mostly disappeared. For instance, modern chipsets include programmable interrupt controllers with latency resolution small enough for demanding real-time applications. As a result, support for real-time requirements has moved to software; i.e., specialized schedulers and resource controllers. Algorithms that were once etched into special circuitry are now implemented in software on general-purpose computers.

This is not to say that hardware support isn't needed in a real-time system. For example, many real-time systems will likely require access to a programmable interrupt controller for low-latency interrupts and scheduling, a high-resolution clock for precise timing, direct physical memory access, or a high-speed memory cache. Most modern computer hardware, including servers, workstations, and even desktops and laptops, support these requirements. The bottom line is whether the operating system software running on this hardware supports access to these hardware facilities.

The operating system may, in fact, support real-time tasks directly through its scheduling implementation, or may at least allow alternative scheduling algorithms be put in place. However, many general-purpose operating systems schedule tasks to achieve different goals than a real-time system. Other factors, such as overall system throughput, foreground application performance, and GUI refresh rates, may be favored over an individual task's latency requirements. In fact, in a general-purpose system, there may be no way to accurately specify or measure an application's latency requirements and actual results.

However, it *is* still possible to achieve real-time behavior, and meet real-time tasks' deadlines, on general-purpose operating systems. In fact, this is one of the

charter goals that Java RTS, and the RTSJ, set out to solve: real-time behavior in Java on general-purpose hardware and real-time operating systems. In reality, only a subset of general-purpose systems can be supported.

The remainder of this chapter provides an overview of the theory and mechanics involved in scheduling tasks in a real-time system. To be clear, real-time scheduling theory requires a great deal of math to describe and understand thoroughly. There is good reason for this: when a system has requirements to meet every deadline for actions that may have dire consequences if missed, you need to make assurances with the utmost precision. Characterizing and guaranteeing system behavior with mathematics is the only way to do it. However, we'll attempt to discuss the subject without overburdening you with deep mathematical concepts. Instead, analogies, descriptions, and visuals will be used to bring the concepts down to earth, at a level where the average programmer should be comfortable. For those who are interested in the deeper math and science of the subject, references to further reading material are provided.

The Highway Analogy

One simple way to describe the dynamics of scheduling tasks in a real-time system is to use a highway analogy. When driving a car, we've all experienced the impact of high volume; namely, the unpredictable amount of time spent waiting in traffic instead of making progress towards a destination. This situation is strikingly similar to scheduling tasks in a real-time system, or any system, for that matter. In the case of automobile traffic, the items being scheduled are cars, and the resource that they're all sharing is road space. Comparatively, a computer system schedules tasks, and the resource they share is CPU time. (Of course, they also share memory, IO, disk access, and so on, but let's keep it simple for now.)

In the highway analogy, the lanes represent overall computer resources, or time available to process tasks. More capable computers can be loosely described as having more lanes available, while less capable systems have fewer. A car is equivalent to a task that has been released (eligible for execution). Looking at Figure 1-8, you can see tasks "traveling" down individual lanes, making forward

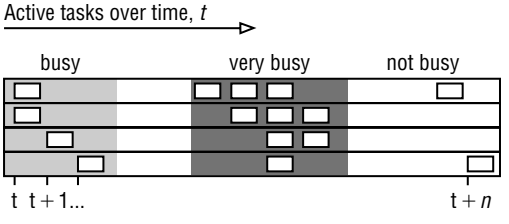

Figure 1-8 As with cars on a highway, when there are more tasks executing, the system slows down, and execution times become unpredictable.

progress over time. At moments when more tasks share the highway, the entire system is considered to be busy, and usually all tasks will execute slower. This is similar to the effects that high volumes of cars have on individual car speeds; they each slow down as they share the highway. Since, in this scenario, all tasks share the resources (the highway) equally, they are all impacted in a similar, but unpredictable, way. It's impossible to deterministically know when an individual task will be able to complete.

In the real world, engineers designing road systems have come up with a solution to this problem: a dedicated lane.

The Highway Analogy—Adding a Priority Lane

Figure 1-9 proposes a specialized solution to this problem: a dedicated high-priority lane (sometimes called a carpool, or HOV lane, on a real highway). We refer to it as specialized because it doesn't help all tasks in the system (or all cars on the highway), only those that meet the requirements to enter the high-priority lane. Those tasks (or cars) receive precedence over all others, and move at a more predictable pace. Similarly, in a real-time system, dedicating system resources to high-priority tasks ensures that those tasks gain predictability, are less prone to traffic delay, and therefore complete more or less on time. Only the normal (lower-priority) tasks feel the effects of high system volume.

This analogy goes a long way towards describing, and modeling, the dynamics of a real-time system. For instance:

- Tasks in the high-priority lane gain execution precedence over other tasks.
- Tasks in the high-priority lane receive a dedicated amount of system resources to ensure they complete on time.
- When the system is busy, only normal tasks feel the impact; tasks in the high-priority lane are almost completely unaffected.

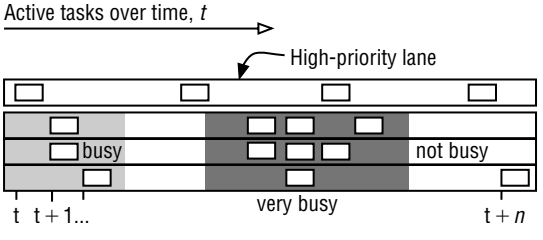

Figure 1-9 Introducing a high-priority lane to a highway ensures that the cars in that lane are less susceptible to traffic, and therefore travel more predictably towards their destinations.

- Overall, the system loses throughput, as fewer lanes are available to execute tasks.
- Tasks only enter the high-priority lane at certain checkpoints.
- Some system overhead is required at the checkpoints. Just as cars need to cautiously (and slowly) enter and exit a carpool lane, tasks are slightly impacted.
- Tasks may be denied access to the high-priority lane if their entry would adversely affect the other tasks already running.

Additionally, metering lights are used at the on-ramps to many highways. These lights control the flow of additional cars (analogy: new tasks) onto the highway to ensure the cars already on the highway are impacted as little as possible. These lights are analogous to the admission control algorithm of the scheduler in real-time system.

Most importantly, this analogy shows that there's no magic involved in supporting a real-time system; it, too, has its limits. For instance, there's a limit to the number of high-priority tasks that can execute and meet their deadlines without causing all tasks to miss their deadlines. Also, because of the need to dedicate resources to real-time tasks, the added checkpoints for acceptance of real-time tasks, the need to more tightly control access to shared resources, and the need to perform additional task monitoring; the system as a whole will assuredly lose some performance and/or throughput. However, in a real-time system, predictability trumps throughput, which can be recovered by other, less-complicated, means.

As we explore the details of common scheduling algorithms used in actual real-time systems, you will also see that simply "adding more lanes" doesn't always resolve the problem effectively. There are practical limits to any solution. In fact, in some cases that we'll explore, adding processors to a computer can cause previously feasible schedules to become infeasible. Task scheduling involves many system dynamics, where the varying combination of tasks and available resources at different points in time represents a difficult problem to solve deterministically. However, it *can* be done. Let's begin to explore some of the common algorithms used, and the constraints they deal with.

Real-Time Scheduling

A task by itself represents a useless body of instructions. For a task to be useful and meaningful, it must be processed, and it therefore must have scheduled execution time on a processor. The act of scheduling processor time to a task, or

thread, and assigning it a free processor is often called *dispatching*. Real-time schedulers can schedule individual tasks for execution either offline (prior to the system entering its running state) or online (while the system is in an active, running state). Regardless of when it occurs, all scheduling work is done according to a predefined algorithm, or set of rules.

Scheduling Constraints

Many factors, or constraints, are taken into account when scheduling real-time tasks. Because each application—and hence each task—is different, these constraints may vary from system to system. However, they all generally fall into a subset of constraints that need to be considered. The first constraint is the amount of available resources; whether they're tasks in a computer or construction workers on a job, having what you need to complete a task is important. The second constraint is task precedence; to avoid chaos, and to ensure proper system behavior, certain tasks may need to be executed before others. The third constraint is timing; each task has its own deadline, some tasks execute longer than others, some may execute on a steady period, and others may vary between release times.

Each constraint contains many of its own factors that need to be considered further when scheduling tasks. Let's examine these constraints in more detail now, and how they may affect task scheduling.

Resources

Because we speak in terms of processors, tasks, and execution times, most people think of only the CPU as the main resource to be scheduled. This isn't always the case. For example, recalling the highway analogy above, cars represented tasks, and the road represented the processor. More realistically, in a real-time network packet switching system, tasks represent data packets, and the processor represents an available communication line. Similarly, in a real-time file system, a task is a file, and the processor is an individual disk platter/head combination. However, regardless of the actual physical work being done (sending a packet, writing a file, or executing a thread), we will refer to all of them as simply a task. Further, regardless of the actual component doing the work (a network card/communication link, a disk controller, or a CPU), we will refer to all of them as simply a processor.

It's easy to see how the availability of critical system resources, such as the processor, is important to a scheduling algorithm. After all, to execute a thread, there must be a CPU available to execute it. However, it goes beyond just the processor; other resources, such as shared objects that require synchronization across tasks,

are to be considered. This may be something as abstract as a shared object in memory, or something more concrete such as a shared region of memory, or the system bus. Regardless, there is often a need to lock shared resources to avoid errors due to concurrent access. This effectively limits a resource to being updated atomically, by one task at a time. Since resource locking synchronizes access to a shared resource, one task may become blocked when attempting to access a resourced that is currently locked by another task.

In a real-time system, it's important to ensure that high-priority, hard real-time tasks continue to make progress towards completion. Resource locking is commonly a problem since priority inversion can cause tasks to execute out of order. In many general-purpose operating systems, resource locking can lead to priority inversion, resulting in unbounded latency for critical tasks, and missed deadlines in a hard real-time system. For instance, in Figure 1-10, we see three tasks, T1, T2, and T3, each with decreasing priority, respectively.

In this scenario, task T3 is released shortly after the system starts. Although it's the lowest-priority task in the system, it begins execution immediately because no other tasks have been released. Early in its execution, it acquires a lock on resource R1. At about time t + 1.5, task T1 is released and preempts the lower-priority task, T3. At time t + 3, task T2 is released but cannot execute because task T1 is still executing. At some point after t + 3, task T1 attempts to acquire a lock on R1, but blocks because it's already locked. Because of this, T2 gains execution precedence over T1 even though it has a lower priority. When T2 completes, T3 continues again until it releases R1, at which point T1 is finally able to resume.

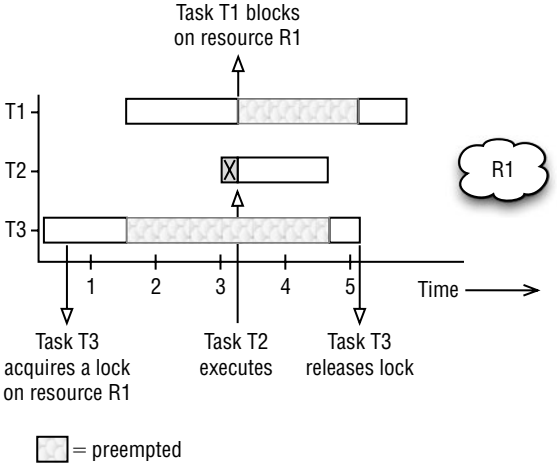

Figure 1-10 Resource locking can lead to priority inversion. In a real-time system, this scenario must be controlled, or avoided.

In a general-purpose computer system, this may be a common and acceptable situation. In a real-time system, however, this violates the principal of task priority execution, and must be avoided. The time frame from shortly after t + 3, to shortly after t + 5, represents unbounded latency that adds an unknown amount of delay to the critical task, T1. As a result, real-time systems must implement some form of priority inversion control, such as priority inheritance, to maintain forward progress of critical tasks. This will be discussed later in the chapter; for now, let's continue our discussion on scheduling constraints.

Precedence

In many systems, real-time systems included, tasks cannot be scheduled in arbitrary order. There is often a precedence of events that govern which threads must be scheduled first. Typically, the threads themselves imply the ordering through resource sharing, or some form of communication, such as a locking mechanism. One example is a system where a task *T1* must wait for another task *T2* to release a lock on an object before it can begin execution. This act of *task synchronization* must be performed at the OS kernel level so as to notify the scheduler of the task precedence that exists.

In the example above the scheduler will block task *T1*, and will allow task *T2* to execute. When task *T2* completes its processing and releases its lock on the synchronized object that the two tasks share, the scheduler can dispatch task *T1*. The task precedence dictated by the synchronization in this example must be obeyed even if there are other processors in the system that are ready to execute waiting tasks. Therefore, in this example, regardless of the number of available processors, task *T1* will *always* wait for task *T2* to complete and hence release task *T1* to begin.

Notation: T2 < T1, or T2 $\rightarrow$ T1 for immediate task precedence

Task precedence can get complex, as a system with multiple tasks (each with its own dependencies) must be scheduled according to the precedence rules. Real-time system designers must understand task precedence fully before a system begins execution to ensure that real-time deadlines can be met. To do this, precedence relationships can be predetermined and represented by a graph, such as the one shown in Figure 1-11.

In this diagram, each node on the graph represents a task. The nodes at the top must execute and complete first before threads below them can execute. These rules are repeated at each level of the graph. For instance, the scheduler for the tasks represented in Figure 1-11 will begin by dispatching task *T2* to execute first. Once *T2* is complete, task *T1* will execute, while tasks *T3* and *T4* are blocked.

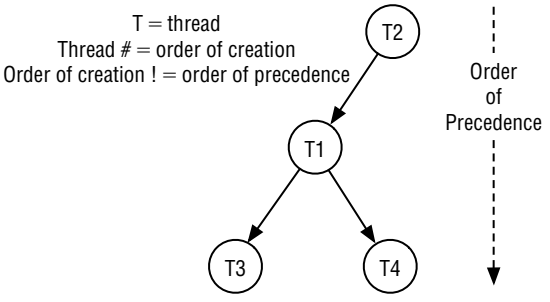

Figure 1-11 A directed acyclic graph helps to summarize task dependencies in a system.

Once *T1* completes, tasks *T3* and *T4* will be eligible to be dispatched. If there is more than one processor ready, both tasks *T3* and *T4* can be scheduled to run simultaneously—this graph does not indicate precedence between them.

Timing

Real-time tasks are labeled as such because they have time-related constraints, usually to do with a deadline for processing. As a result, schedulers are concerned with many time-related parameters to ensure that a task can complete on or before its deadline. For instance, schedulers need to consider the following, per task:

Deadline: of course, understanding the task's deadline (relative or absolute) and its value is critical to determining if it can be scheduled feasibly.

Period: many real-time tasks need to execute at regular time intervals. Others do not, and instead respond to events as they happen. The scheduler must distinguish each task's type (periodic, aperiodic, and sporadic), along with the timing characteristics that might apply. We'll discuss periodic tasks in the next section.

Release Time: important information when scheduling periodic tasks, to determine if a set of tasks can be scheduled feasibly.

The Variation in Releases Times: tasks vary slightly from period to period as to when they will actually be eligible for execution.

Start Time: some tasks will not be able to execute as soon as they are released (due to locks, higher-priority tasks running, and so on).

Execution Time: task cost functions need to know the best- and worst-case execution times for a task.

Other parameters are calculated as a result of scheduling a given set of tasks, such as:

Lateness: the difference between a task's actual completion time, and its deadline.

Tardiness: the amount of time a task executes after its deadline. This is not always equal to the lateness value, as a task may have been preempted by another task, which caused it to miss its deadline.

Laxity: the amount of time after its release time that a task can be delayed without missing its deadline.

Slack Time: same as laxity (above). However, this term is sometimes used to describe the amount of time a task can be preempted during its execution and still meet its deadline.

Scheduling Algorithms

The three constraint sets above are directly used in scheduling algorithms to determine a feasible schedule for a set of real-time tasks. Feeding into the algorithm as parameters are the set of tasks, T, to be scheduled; the set of processors, P, available to execute them; and the set of resources, R, available for all tasks in the system.

Scheduling algorithms differ in their criterion in determining when, and for how long, a task can execute on a processor. Real-time scheduling algorithms exist to ensure that regardless of system load—or the amount of threads eligible to be dispatched—the time-critical tasks of the system get ample processing time, at the right time, to meet their deadlines. Let's examine some of the high-level characteristics of real-time schedulers.

Preemptive Versus Non-Preemptive Scheduling

To understand scheduling algorithms, we need to agree upon some basic concepts. For instance, most operating systems allow a task to be assigned a priority. Higher-priority tasks, when ready to execute, will be given precedence over lower-priority tasks. An algorithm is said to be preemptive if a running task can be interrupted by another, higher priority task. However, there are scheduling algorithms in the real-time world that are non-preemptive [Liu00]. In this case, once a thread executes, it continues to do so until it completes its task.

Some algorithms allow a mixture of preemptive and non-preemptive tasks with classes of thread priorities, and specific support for real-time tasks. On a

multi-processor system, this hybrid approach works well, since relatively long-running non-preemptive tasks do not prevent the system from performing other tasks, as long as there are more processors available than real-time tasks. The advantage to the real-time programmer is increased control over system behavior, which results in a more deterministic system overall.

Context Switching

In addition to the added determinism it provides, another reason to choose a non-preemptive scheduler is to avoid unforeseen *context switches*, which occur when one thread preempts another thread that's already running. The operating system must expend processor time executing code that saves the state of the already running thread, and then resets internal data structures and processor registers to begin execution of the preempting thread so that it can begin execution. This entire preemption process is summarized as *dispatching*, as mentioned earlier in this chapter. The time it takes a particular system to perform this preemption task is called *dispatch latency*, and can interfere with a system's ability to respond to an event by its deadline.

As a system becomes increasingly busy, with more threads competing for processing time, a considerable percentage of time can be spent dispatching threads, resulting in an additive effect in terms of dispatch latency. A system that spends more time dispatching threads than actually performing real work is said to be thrashing. Real-time systems must guarantee that even under high system load, thrashing due to context switches will not occur, or that it at least won't interfere with the high-priority threads performing real-time processing in the system. OS kernels with hybrid preemption/non-preemption support are useful in these cases, allowing real-time processing to be guaranteed on a general-purpose system.

Dynamic and Static Scheduling

One way to control context switching while still allowing preemption is through *static scheduling*. For instance, in a *dynamically scheduled* system, tasks are dispatched on the fly as the system is running, based upon parameters that may change over time (such as task priority). All decisions as to which task to schedule at each point in time are made while the system is running, and are based on the system state at certain checkpoints. This is a common scheduling algorithm used in many popular operating systems.

With static scheduling, however, the execution eligibility of a task never changes once it's assigned. The important scheduling parameters for each task, such as the priority, are determined when those tasks first enter the system based upon the

state of the system and its current set of tasks. For instance, a static scheduling algorithm may assign a priority to a new task based on its period (defined below) relative to the periods of the other tasks in the system. In this way, a static scheduling algorithm can be used either offline or online (while the system is running).

Task Migration

Another important parameter to real-time task scheduling is the ability of a task to be executed by different processors in a multiprocessor system. For instance, in some systems, once a task begins execution on a particular processor, it cannot migrate to another processor even if it's preempted, and another processor is idle. In this type of system, task migration is not allowed.

However, many systems do allow tasks to migrate between available processors as tasks are preempted, and different processors become available. This type of system supports *dynamic task migration*, as there are no restrictions placed on tasks in relation to processors. Of course, there are systems that fall in between, which support *restricted task migration*. One example is a system with many processors, where certain tasks are restricted to a subset of total processors within the system. The remaining processors may be dedicated to a task, or small set of tasks, for example. Another example might be that on multiprocessor/multi-board systems, the OS might limit task migration to CPUs on the same board to take advantage of locality of reference.

An example of a common real-time operating system that supports restricted task migration is Solaris. Solaris allows you to define individual processor sets, assign physical processors (or processor cores) to the defined sets, and then dedicate a processor set to a particular application. That processor set will then execute only the threads (tasks) within that application, and no others. The tasks will be able to migrate across the physical processors within the set, but not the others.

Periodic, Aperiodic, and Sporadic Tasks

Earlier in this chapter, we discussed relative and absolute deadlines, and compared the differences using some examples. A task that is released at a regular time interval is often called a *periodic* task. An aperiodic task has no known period; external events are generated asynchronously at unknown, often random, time intervals. To be precise, with an aperiodic task, the time interval between releases varies, but is always greater than or equal to zero:

For aperiodic task, t_i: $\forall i \in N, (r_{i+1} - r_i) \geq 0$

In comparison, for a periodic task, the length of the time interval between any two adjacent releases is always constant:

For periodic task, t_i: $\forall i \in N, (r_{i+1} - r_i) = C$

Whether a thread is periodic or aperiodic can make a difference to the scheduler, as one is to do work on a known time interval, and the other is released when an event notification arrives—such as a network message—at unknown points in time. It's obvious that if all threads in a system were periodic, it would be easier to work out a schedule than if all threads were aperiodic.

A system with multiple aperiodic threads must be carefully planned and measured as unforeseen events, and combinations of events, can occur at any time. This situation may seem orthogonal to the real-time requirement of predictability, but with careful consideration of thread priorities, it's mathematically possible to schedule such a system. Later in this chapter, we'll examine how the most common scheduling algorithms handle aperiodic tasks through what's called a *sporadic server*.

A common example of a sporadic task in a hard real-time system is the autopilot control in an airplane's flight control system. The human pilot may switch the autopilot on, and subsequently off, at specific points in time during a flight. It's impossible to determine precisely when these points in time may occur, but when they do, the system must respond within a deadline. It would certainly be considered an error condition if, when the pilot switched off the autopilot, that the system took an unduly large amount of time to give the human pilot control of the aircraft.

A sporadic task is one where the length of the time interval between two adjacent releases is always greater than or equal to a constant (which itself is non-zero):

For sporadic task, t_i: $\forall i \in N, (r_{i+1} - r_i) \geq K$

Execution Cost Functions

When scheduling real-time tasks, it's necessary to have a mathematical basis upon which to make scheduling decisions. To do this, we need to take into account the following calculations of task execution cost:

Total completion time, $t_c = max(f_i) - min(a_i)$

This cost function calculates the overall completion time for a set of tasks by subtracting the time the last task finished from the time the first task started; note that these can be different tasks.

Late task, late(t_i) $= \begin{cases} \textit{if } d_i > 0 \textit{ and } f_i > d_i \textit{ then 1 else 0} \\ \textit{if } D_i > 0 \textit{ and } C_i > D_i \textit{ then 1 else 0} \end{cases}$

This function returns the value 1, indicating true, if the task's finish time exceeds its absolute deadline, or if the task's completion time exceeds its relative deadline, depending upon the type of deadline the task has.

Number of late tasks, $N_{late} = \sum_{i=1}^{n} late(t_i)$

This function calculates the total number of tasks that miss their deadline in a given real-time system with n tasks. Note: To simplify the equations and the discussion going forward, let's define a task's deadline, d_i, where d_i equals its absolute deadline d_i when a task has an absolute deadline, or $d_i = a_i + D_i$ when a task has a relative deadline.

Maximum lateness, $L_{max} = max(f_i - d_i)$: this function calculates the task with the maximum lateness (missed its deadline by the greatest amount of time) using d_i as we've just defined it. Note that this value can be negative (when all tasks finish before their deadlines), or positive.

Average response time, $R_{avg} = \frac{1}{n}\left(\sum_{i=1}^{n} C_i\right)$

This function calculates the average response time for a given real-time system by dividing the sum of all task completion times by the total number of tasks in the system.

Classification of Real-Time Schedulers

Let's look now at the common types of schedulers used in real-time systems. We've already explored some features of scheduling algorithms in general, such as preemption, static and dynamic scheduling, and support for aperiodic and periodic tasks. Real-time systems go further in breaking down task scheduling

For instance, real-time schedulers can be broken down into two overall categories:

Guarantee-Based: these algorithms are often static, often non-preemptive, and rigid to ensure that all tasks can complete their work by their given deadline. In dynamic systems, conservative measurements are typically used to ensure that the arrival of new tasks will not cause any existing tasks to miss

their deadlines as a result. Often, these algorithms will err on the side of not allowing a new task to start if there is a chance it can disrupt the system. This is done to avoid a domino effect, where the introduction of a new task causes all existing tasks to miss their deadlines. This pessimistic approach can sometimes mean that tasks may be blocked from starting that would *not* have caused a problem in reality.

Best-Effort Based: these algorithms are dynamic, more optimistic, and less conservative, when it comes to new task arrival. These schedulers are typically used in systems with soft real-time constraints, such that a missed deadline due to new task arrival is generally tolerable. They are classified a "best-effort" because these schedulers almost always allow new tasks into the system, and will do their best to ensure that all tasks complete their processing on or close to their deadlines. This results in a very responsive, efficient, real-time system that is best suited for soft real-time tasks where hard guarantees are not needed.

Within both classifications, the different algorithms must be one of the following to be considered as real-time:

Feasible: sometimes called an heuristic algorithm, this scheduler searches for a feasible schedule whereby all tasks complete their work at or before their respective deadlines. A feasible schedule still guarantees the real-time behavior of the system. A schedule is deemed *infeasible* if one or more tasks within a real-time system miss a deadline.

Optimal: an optimal scheduling algorithm is one that will always find a feasible schedule (all tasks complete on or before their deadlines) if one exists. An optimal algorithm may not always produce the *best* schedule, but it will always produce a schedule that meets every task's deadline if one is possible.

To achieve a feasible schedule for tasks in a hard real-time system, there are three common approaches often used. These are:

Clock-Driven: sometimes called time-driven, this approach schedules task execution based on known time qualities of each task in the system before the system starts. Typically, all decisions are made offline, and scheduling activities are performed during well-known points in time while the system is running. The result is a guaranteed real-time schedule with predictable execution, where the scheduler operates with very little overhead. The time-based quality(s) used to make scheduling decisions vary among the different clock-driven algorithms often used.

Weighted Round-Robin: similar to round-robin schedulers used in time-sharing systems, this approach applies a different weight value to each task in the system based upon some criterion. Tasks with higher weight are given more execution time, or higher priority, resulting in their ability to complete their work sooner. This approach is only suitable in certain types of systems (such as those with very little interdependency amongst the individual tasks in the system).

Priority-Driven: sometimes called event-driven, scheduling decisions are made when important system events occur (such as the release of a task, the availability of a resource, an IO event, and so on) with the intent to keep resources as busy as possible. With this approach, tasks are given a priority, placed in priority-ordered queues when released, and are then dispatched at the earliest point in time that a processor is free. No task is unduly delayed when an idle processor exists (a trait that can be a liability in some cases). Algorithms within this scheduling class are further sub-classed as fixed-priority (where task priorities and execution orders remain constant during system execution), and dynamic-priority (where tasks' priorities can change, and all scheduling is done online while the system is running).

There are many different real-time scheduling algorithms in existence, each of which works best with certain types of systems. For instance, some schedulers analyze tasks offline, before the system begins to execute. This type of system requires all tasks be known before execution, and that no new tasks be introduced while the system is in a time-critical mode.

Other schedulers work while a system is online, and must contain additional logic to ensure that as new tasks enter the system, a new feasible schedule can be generated. In either offline or online scenarios, some schedulers work only with fixed-priority tasks, while others allow tasks' priorities to change while the system is running.

The dynamic-priority, online schedulers tend to be the most complex and least predictable, while static-priority schedulers tend to be best for hard real-time systems. Let's look at some common algorithms that fall into these categories, and examine their characteristics:

First-Come-First-Served (FIFO): sometimes referred to as a first-in-first-out (FIFO) schedule, this is a dynamic-priority algorithm that schedules tasks based on their release times (execution eligibility).

Earliest-Deadline-First Scheduler (EDF): in this dynamic preemptive scheduler, at any instant the executing task is the task with the closest deadline.

Shortest-Execution-Time-First Scheduler (SETF): in this non-preemptive scheduler, the task with the smallest execution time is scheduled first.

Least-Slack-Time Scheduler (LST): tasks are given higher priority based on their slack time. In this dynamic-priority algorithm, task slack time is calculated as the task's deadline minus the current point in time minus the time required to complete processing. The task with the smallest slack time is given highest priority. Slack times—and hence priorities—are recalculated at certain points in time, and tasks are subsequently rescheduled.

Latest-Release Time-First Scheduler (LRT): sometimes called reverse-EDF, tasks are scheduled backward, in the sense that release times are treated as deadlines, and deadlines are treated as release times. The scheduler works backwards in time and assigns tasks to processors based on their deadlines, working back to the start of execution. Think of it as reading from right to left. This scheduler performs all scheduling offline, before the system begins to execute.

Rate-Monotonic Scheduler (RM): the inverse of a task's period is called its *rate*. Used in systems with periodic tasks and static priorities, this fixed-priority preemptive algorithm assigns higher priorities to tasks with shorter periods (higher rates).

Deadline-Monotonic Scheduler (DM): in this static, fixed-priority preemptive scheduler, the task with the shortest relative deadline is scheduled first. In a real-time system, when each task's relative deadline equals its period, the RM and DM schedulers generate the same schedule.

In some systems, such as those that run both real-time and general-purpose applications together, a scheduling algorithm is used in conjunction with a partitioning scheme to achieve optimal system behavior. With this approach, tasks are partitioned, or grouped, based upon certain criteria. The partitioning scheme can be either fixed, or adaptive:

Fixed-Partition Scheduling: this is a dynamic, preemptive, scheduler that assigns time budgets, in terms of total processing time allowed, to different groups of tasks called partitions. Each partition, as a whole, is bounded not to exceed its total budget of processor time (i.e., 10% of the CPU).

Adaptive-Partition Scheduling: this is a dynamic, preemptive, scheduler where a percentage of the processor time (and sometimes system resources) are reserved for a particular group of tasks (partition). When the system reaches 100% utilization, hard limits are imposed on the non-real-time partition in order to meet the needs of the real-time tasks. When the system is less than 100% utilized, however, active partitions will be allowed to borrow from the budget reserved for other, non-active, partitions.

The intent of the Java Real-Time System is to hide this complexity from you. However, to truly appreciate its implementation, and to understand why your Java applications behave as they do within it, you should at least have a cursory understanding of the theory behind its implementation.

Both the RTSJ and Sun's Java RTS are meant to provide real-time behavior to Java applications even on general-purpose hardware with a real-time OS. However, this is a deviation from what has been classically accepted as a real-time system. In the past, you had to write code in a language designed for real-time applications and use dedicated, special-purpose hardware to run them on. Again, to gain a true appreciation of the real-time space, and the problems that Java RTS had to overcome, let's take a brief look at some common real-time languages and operating systems.

Multi-Core Processors and Scheduling

For the most part, modern operating systems treat individual processor cores as individual processors themselves. This is an accurate representation, as each core *is* a complete processor in its own right, capable of acting like an independent processor—albeit sometimes with a shared on-chip cache. Therefore, it's not required for the OS to treat individual cores any differently than individual processors.

However, some OS kernels do take special care in scheduling threads on separate cores as opposed to separate physical processors. Solaris 10, for instance, will schedule eligible threads to distribute them across physical processors first, then across the cores of individual processors [McDougall07]. For instance, on a system with two dual-core processors, Solaris will dispatch the first thread on core 1 of processor 1, and the second thread on core 1 of processor 2 (see Figure 1-12).

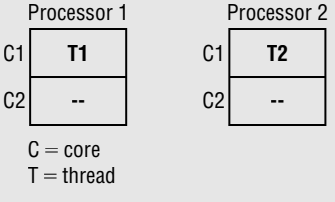

Figure 1-12 The Solaris kernel attempts to balance threads across CPUs to help with heat dissipation.

continued

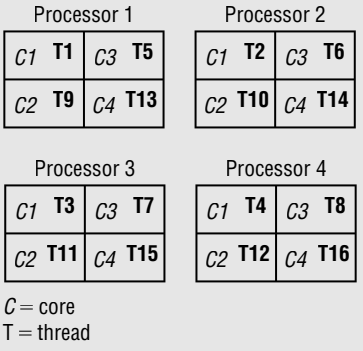

Figure 1-13 Solaris kernel threads spread equally across all available cores. From this point onward, additional threads will be scheduled on the next available (free) core.

Threads 3 and 4 will be dispatched to core 2 of processor 1, and core 2 of processor 2, respectively. In this case, even though thread 2 could have been scheduled on core 1 of processor 1, this would upset the balance. The intent is to distribute running threads across physical processors *and* their cores evenly. For instance, on a system with four quad-core processors, with sixteen dispatched and running threads, the thread-to-processor/core distribution should look as shown in Figure 1-13.

This distribution of threads over available cores has little bearing on our real-time discussion, although it's important to know that it does occur.

Real-Time Operating Systems

To meet the needs of the most demanding real-time systems, specialized operating systems are available. Besides having the required features to support predictable task execution for time-critical applications, the real-time OS will also have its own real-time constraints. For example, core scheduling and other OS-specific functionality must behave in predictable, measurable, ways, with high efficiency and low latency. In other words, the OS and its scheduling activities must never contribute to unbounded task latency, or be unpredictable in any sense.

A real-time OS has thread schedulers built-in that support real-time systems. Most often, these operating systems allow programmers to control the computer hardware at the lowest level, including processor interrupts, physical memory access, and low-level input/output (I/O) processing. This is done so that the real-time application developer can remove as much of the non-deterministic behavior and end-to-end latency found in general-purpose operating systems.

Many real-time operating systems target embedded systems with dedicated functionality and limited hardware resources. This means that these systems are not meant to be general-purpose systems, and using a real-time OS proves to be a valuable tool in these cases. However, as mentioned before in this chapter, with the continuing advances in hardware capabilities at lower cost, the need for specialized hardware *and* operating systems has diminished significantly. A dedicated real-time OS comes at a high cost relative to the general-purpose operating systems available today. Many argue that the state of the industry has reached a point that this extra cost is no longer justified. However, recall that while adding additional, more powerful, hardware can help improve raw performance and throughput, it almost never makes an unpredictable system behave predictably.

Regardless, there are instances, such as with embedded systems, where a real-time OS continues to be the only option. While this book does not go into detail regarding real-time operating systems, it's important to know that they do exist and serve a specialized purpose. Real-time applications typically require the same set of services from the OS—such as disk IO, networking support, a file system, user IO, and so on—as general purpose applications do. However, the real-time application requires its OS to make guarantees that are both measurable and predictable.

RT-POSIX Operating System Extensions

With the intent to unify the real-time application and OS space, the Real-Time Portable Operating System based-on Unix (RT-POSIX 1003.1b) standard was created. POSIX is a standard that has been very successful and widely adopted in a wide-range of OS implementations, for both mission-critical and general-purpose computing. RT-POSIX defines an extension to it that addresses the needs of hard and soft real-time systems.

The standard defines a minimum set of features that a compliant OS must implement. Additional features can be implemented, provided that they do not conflict with or hereby negate the required features. For example, RT-POSIX-compliant operating systems must have the following traits:

Preemption: true task preemption must be supported using task priority rules that are strictly obeyed.

Priority Inversion Control: although it cannot guard well against deadlocks, priority inheritance ensures that lower-priority threads will never be able to block higher-priority threads due to classic priority inversion.

Periodic, Aperiodic, and Sporadic Threads: the POSIX standard requires only processes be implemented. For real-time applications, threads of different

priority may need to run to perform a task. For this reason, the RT-POSIX standard requires the OS be able to schedule tasks down to the thread level, and that applications be able to create and destroy those threads. Underlying OS threads must be able to support the various types of task release events common to real-time applications. For instance, a thread should be able to specify its period and then rely on the OS to wake it up at precise time boundaries that match that period. Simply performing a call to "sleep" from within the task's code (as is done in a non-real-time OS) is not sufficient.

High-Resolution Timers: such timers should be made available to real-time applications, as well as the real-time OS itself so that it can dispatch threads with as little latency and jitter as possible. For example, an OS scheduler with a 10-millisecond tick size will, at best, experience up to 10 milliseconds latency during thread dispatch processing. In general, the larger the tick count, the higher the max latency and jitter values will grow. Both are qualities to be avoided in real-time systems. The RT-POSIX standard states that up to 32 timers per process must be supported, and that timer overruns (when a timer goes beyond its chosen duration) be recorded.

Schedulers: the kernel needs to support both deadline-monotonic and rate-monotonic deadline scheduling in order to support the common real-time scheduling algorithms such as weighted round-robin, fixed-priority, and earliest-deadline-first scheduling. The type of scheduler used can be defined down to the thread level, where two threads of the same process may be scheduled differently.

Scheduled Interrupt Handling: the ability to create a preemptable task that processes low lever device interrupts. This is sometimes called a software interrupt. In contrast, a general-purpose OS typically handles these events completely itself, making the data or status available through some other means.

Synchronous and Asynchronous IO: synchronous IO operations provide real-time tasks more control over IO-related tasks, such as file and network operations. Asynchronous IO allows the system to progress task execution while IO is occurring, or waiting to occur. For instance, a task that reads a network packet can process the packet's payload even while it waits for the next packet to arrive.

Inter-Task Communication: to facilitate predictable, low-latency, communication between tasks, queues should be provided at the OS level. This also ensures fast, consistent, and measurable performance, with support for message prioritization. These queues are sometimes made available to tasks both local to the system, and remote. Regardless, message delivery must be prioritized, and at least eight prioritized signals are supported per process.

Priority Inheritance: to guard against deadline misses due to priority inversion (where a lower-priority task's priority is raised above that of a higher-priority task) priority inheritance needs to be implemented at the kernel level, or at least emulated.

Resource Quotas: the ability to monitor and control the usage of system resources such as memory and processor time to ensure the system behaves in a predictable way even when under heavy load.

Memory Sharing: shared memory (between processes) and memory-mapped files must be supported.

Memory Locking: applications must be able to control the memory residency of their code sections through functions that will either lock all of its code, or only the portions specified.

Real-Time File System: file systems that ensure files are made up of contiguous disk blocks, pre-allocate files of fixed size to be used on-demand while the system is running, and offer sequential access, provide the most predictable behavior and timing characteristics.

Synchronization: OS primitives that support efficient resource sharing between tasks with priority inheritance, and ceiling priority (both of which are protocols to control or avoid thread priority inversion).

Further Reading

Much of the information for this chapter was gathered from papers and texts on the subject of real-time systems and scheduling theory. For readers who are interested, below is a list of reading material that will help build a complete foundation for real-time theory and practice:

[Buttazzo05] Buttazzo, Georgia C., *Hard Real-Time Computing Systems*. Springer, 2005.

[Klein93] Klein, Mark, et al., *A Practitioner's Guide to Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[Layland73] Liu, C.L. and Layland, James W., *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*. Journal of the ACM, 1973 (Available at http://portal.acm.org/citation.cfm?id=321743).

[Liu00] Liu, Jane W. S., *Real-Time Systems*. Prentice Hall, 2000.

[McDougall07] Mauro, Jim, McDougall, Richard, *Solaris Internals*. Prentice Hall, 2007.

Index

B

C

D

E

I

J

K

L

M

N

O

P

Q

R

S

T