AN OPERATING SYSTEM ARCHITECTURE FOR REAL-TIME TASKING ABSTRACTIONS

By

Siu Ling Ann Lo

B.Sc. (Computer Science) The University of British Columbia, 1981
M.Math. (Computer Science) The University of Waterloo, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1995
© Siu Ling Ann Lo, 1995
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 26, 1995
Abstract

A flexible real-time tasking abstraction is necessary to support diversified and evolving application-level timing constraints. Previous research has focused on rigid solutions and failed to provide any insight to a more flexible design alternative. A common approach has been to implement a real-time scheduling algorithm in the operating system kernel as an integral part of the task model. When the real-time requirements of applications evolve and a new scheduling algorithm is needed, the kernel is modified. This monolithic approach is out of step with the press for software reusability when software development and maintenance costs continue to escalate.

In order to explore for a flexible design, we have examined the concepts behind existing real-time task models and classified them into competitive and cooperative task models. In contrast to the common belief that they are fundamentally different, our conclusion is that although the design of these task models are incompatible with each other, their underlying concepts are not. This paves the way for our architectural design which supports evolving real-time requirements at the application level. We propose a task abstraction where tasks cooperate and participate in making real-time scheduling decisions. This design allows for the implementation of competitive task models and their scheduling algorithms on top of a common kernel. It also provides more flexibility in task structuring since applications can be structured as cooperating tasks.

We have implemented our design in the Mach 3.0 kernel and the r-kernel for experimentation and for proving the practicality of our abstraction in terms of performance and consistency with the overall design of the systems. The lessons we have learnt from implementation have strengthened our belief in the importance of providing a flexible scheduling interface which can be used by the application programmer. The flexibility of our solution is illustrated by considering the demands placed on the kernel by a modern multimedia application.
## Contents

Abstract .................................................. ii

Contents .................................................. iii

List of Tables ............................................. vi

List of Figures ............................................ vii

Acknowledgement ......................................... viii

Dedication ................................................ ix

1 Introduction ............................................. 1
   1.1 Motivation ............................................. 2
   1.2 Thesis Statement ..................................... 4
   1.3 Contributions ........................................ 4
   1.4 Synopsis of the Dissertation .................... 7

2 Background .............................................. 9
   2.1 Task Models .......................................... 9
       2.1.1 Processes and Threads ......................... 10
       2.1.2 Tasks .......................................... 11
       2.1.3 Our Terminology ................................ 12
   2.2 Real-Time Scheduling .............................. 13
       2.2.1 Terminology ................................... 13
       2.2.2 Meeting Deadlines ............................ 15
       2.2.3 Responsiveness ................................ 18

3 Real-Time Task Models ............................... 20
   3.1 Competitive Task Models ......................... 20
       3.1.1 Motivations and Objectives .................. 20
       3.1.2 Supporting Timing Correctness ............ 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>New Approaches to Task Synchronization</td>
<td>24</td>
</tr>
<tr>
<td>3.1.4</td>
<td>An Overview of Existing Models</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Flexibility Issues</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>A Comparison with the Cooperating Task Models</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Cooperating Tasks</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Task Structuring</td>
<td>31</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Operating System Design</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Timing Support and Scheduling</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>A Structured Approach</td>
<td>34</td>
</tr>
<tr>
<td>3.3.1</td>
<td>A Separation of Concepts</td>
<td>35</td>
</tr>
<tr>
<td>3.3.2</td>
<td>An Architectural Design</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>Design of an Operating System Architecture</td>
<td>40</td>
</tr>
<tr>
<td>4.1</td>
<td>Core Scheduling Rules</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Real-Time Task Model</td>
<td>43</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Definitions of Tasks</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Time Capabilities</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Alarm Capabilities</td>
<td>44</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Synchronization and Communication Primitives</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Kernel Primitives and User Libraries</td>
<td>45</td>
</tr>
<tr>
<td>4.4</td>
<td>Schedulability Analysis</td>
<td>47</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Review of the Techniques Using the Priority Rule</td>
<td>48</td>
</tr>
<tr>
<td>4.4.2</td>
<td>An Extended Analysis Using the Priority Rule</td>
<td>62</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Review of the Techniques Using the EDF Rule</td>
<td>70</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Review of the Techniques Using the Hierarchical Rule</td>
<td>72</td>
</tr>
<tr>
<td>4.4.5</td>
<td>An Extended Analysis Using the Hierarchical Rule</td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>Kernel Implementation</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>The CPU Scheduler</td>
<td>78</td>
</tr>
<tr>
<td>5.1.1</td>
<td>The FCFS Rule at the Same Priority Level</td>
<td>78</td>
</tr>
<tr>
<td>5.1.2</td>
<td>The Idle Thread</td>
<td>80</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Stack Handoff</td>
<td>80</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Priority Changes</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>The Alarm Capability</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Kernel Debugging</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>Performance Measurements</td>
<td>87</td>
</tr>
<tr>
<td>6</td>
<td>A Flexible Scheduling Abstraction</td>
<td>92</td>
</tr>
<tr>
<td>6.1</td>
<td>Hard Real-Time Systems</td>
<td>92</td>
</tr>
<tr>
<td>6.1.1</td>
<td>The Spring Kernel</td>
<td>93</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The Real-Time Mach Kernel</td>
<td>98</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Other Scheduling Policies</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Performance Measurements on one CPU (microseconds) . . . . . . . . . . . . . . . . . . . . . 90
5.2 Performance of set_thread_sched_attr() in a multiprocessor 88k (microseconds) . . . . 91
# List of Figures

<table>
<thead>
<tr>
<th>Section</th>
<th>Figure Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Control Flow of Cooperating Tasks</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>The Server/Client Model</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>The Mutual Exclusion Model</td>
<td>32</td>
</tr>
<tr>
<td>3.4</td>
<td>A Software Architecture</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>The Hierarchical Scheduler</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Kernel Primitives and User Library Calls</td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Slowing Effect due to a Higher Priority Task</td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>Code Sequence containing Inter-Level Procedural Calls</td>
<td>54</td>
</tr>
<tr>
<td>4.5</td>
<td>Completion Time affected by an Earlier Instance</td>
<td>58</td>
</tr>
<tr>
<td>4.6</td>
<td>Transformation into the Canonical Form</td>
<td>65</td>
</tr>
<tr>
<td>4.7</td>
<td>A Multiply Preemptive Task that Waits</td>
<td>66</td>
</tr>
<tr>
<td>5.1</td>
<td>Insertions and Deletions to the Ready Queue</td>
<td>79</td>
</tr>
<tr>
<td>5.2</td>
<td>The Idle Thread</td>
<td>81</td>
</tr>
<tr>
<td>5.3</td>
<td>Stack Handoff</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Playing a Sound Sequence</td>
<td>88</td>
</tr>
<tr>
<td>6.1</td>
<td>Using Threads to implement the Spring Resource Scheduling Heuristics</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>A Spring Task</td>
<td>97</td>
</tr>
<tr>
<td>6.3</td>
<td>A Cooperating Task Structure</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Priority Inheritance performed by a Resource Administrator</td>
<td>100</td>
</tr>
<tr>
<td>6.5</td>
<td>Scheduling Policies and Schedulability Analysis</td>
<td>102</td>
</tr>
<tr>
<td>6.6</td>
<td>Implementation of Competitive Task Models</td>
<td>103</td>
</tr>
<tr>
<td>6.7</td>
<td>Cooperating Tasks at the Application Level</td>
<td>104</td>
</tr>
<tr>
<td>6.8</td>
<td>An Example of Multi-Stage Process</td>
<td>107</td>
</tr>
</tbody>
</table>
Acknowledgement

I am very fortunate to have Norm Hutchinson and Sam Chanson as my research supervisors. Their guidance and support are remarkable. Norm could read my mind when I struggled to put ideas into words. His enthusiasm and wisdom magically transformed my lone research into an enjoyable experience with fruitful results. I cannot thank him enough. Sam communicated with me regularly and provided feedback even when he was very busy during his extended leave in Hong Kong. His help is very valuable.

I am grateful to have Alan Wagner as a member of my supervisory committee. His thoroughness and suggestions are much appreciated. Special thanks go to Scott Flinn. His sound application was very useful to our experimentation.

The research of this dissertation was supported by an NSERC Postgraduate Scholarship, a Bell Northern Research Postgraduate Scholarship, and research assistantship provided by Norm and Sam. The financial support greatly helped me to concentrate on my research.

Many thanks to my fellow students who have provided me friendship and interesting intellectual discussions. A word of appreciation goes to the staff of the computing facilities who are always ready to help.

Finally, I would like to thank my family for their emotional support. The unconditional love of my late mother and my father provide me the strength. My ten brothers and sisters, as well as their families, always have faith in my ability. I am happy and proud of having a great family.
To the memory of my Mother

Mrs. Yuk Lin Lo
Chapter 1

Introduction

A fundamental role of the operating system kernel is to provide an abstraction of tasks which serves as a base for constructing software systems. The scheduling of tasks as well as the operations available for task synchronization and communications are among the key considerations in the design of the task abstraction. While much experience with task abstractions has been gained for parallel and distributed systems in the past decade, real-time systems are less understood. The design of a real-time task abstraction is particularly challenging because of the complexity of real-time software. Real-time programs must respond to external, real-world, events which are typically not synchronized with the execution of the program. When resources are limited the various tasks that make up a real-time program must share these resources in a manner that preserves required performance guarantees. By comparison, parallel and distributed systems handle asynchronous events but do not necessarily resolve real-time constraints.

The operating system kernel, being the base layer of software in a computer system, is also crucial to overall system performance and extensibility. Kernel overheads in context switching and task communication have always been subjects of optimization. Extensibility is difficult to quantify, but the research on operating systems has long ascertained the value of designing kernel primitives which can be used to implement complex operating system services. An example is the well-known evolution from
monolithic kernel architectures to microkernel architectures. During this evolution, many operating system services have been moved out of the kernel and are now implemented as cooperating tasks. The resulting improvements in flexibility, maintainability, and understandability are profound.

Existing research on real-time kernels has achieved some success — many issues related to task scheduling and synchronization have been investigated, and some novel real-time task abstractions have been designed and implemented. However, little consideration has been given to the flexibility and modularity required in the support of application-level scheduling needs, although it is well known that application requirements are diverse. Each task abstraction is designed for a specific scheduling problem. The kernel designed to support a task abstraction must be completely rewritten to support another.

This dissertation provides an architectural solution to the support of real-time task abstractions and application-level scheduling requirements. We have analyzed the concepts behind existing task abstractions and concluded that it is feasible to provide a common abstraction to support different scheduling objectives. We have implemented our design in the Mach 3.0 kernel and the r-kernel for experimentation and for proving the practicality of our abstraction in terms of performance and consistency with the overall design of the systems. Further experience with our design has been gained by developing a continuous media application. Our experience has ascertained the need to provide flexibility for the evolving and dynamic requirements of real-time software, and has strengthened our belief in our architectural design.

1.1 Motivation

The complexity of real-time software has motivated the design of real-time task abstractions and real-time kernels. Real-time software shares some common requirements with non-real-time software, most notably the need to handle asynchronous events and the desire for minimal kernel overhead. A
fundamental concern specific to real-time software, however, is to satisfy real-time requirements when resources are limited. Because of this, the task scheduling algorithm is of much more importance to real-time software than the local optimization of kernel operations.

Extensive research in scheduling theory has addressed the finite resource problem and motivated the notion of incorporating real-time constraints into program structure. In this approach, a program is structured as a collection of tasks with predefined real-time requirements. One of the main objectives of this architecture is to relieve a programmer from the complicated task of resolving the scheduling problems of the entire system. The programmer only needs to specify the real-time requirements of each component of his program. The operating system predicts whether these real-time requirements can be met. If the result is positive, the operating system also guarantees to provide this program the resources it needs. This new approach to program structuring was originally designed for hard real-time systems where the failure of a hard real-time task can potentially cause a catastrophe. Currently, the abstraction of real-time tasks is gaining popularity in a wider range of applications which are often described as soft real-time because the penalty of a timing error is not as severe as in hard real-time systems. Examples of such real-time applications include multimedia applications, telecommunications controllers, and satellite ground systems. The abstraction of real-time tasks has inspired the design of new real-time kernels as well as the restructuring of existing operating systems. The latter is important because of the need to run real-time applications such as multimedia in the same environment as complex non-real-time servers and applications.

When evaluated from a software engineering viewpoint, existing real-time task models are considerably lacking in their support for modularity and expandability. Each model is designed for a specific scheduling algorithm, and its task structure and primitives are not applicable to other real-time problems. When these real-time task models are implemented in the operating system, a monolithic approach is used. An operating system kernel designed for one task model is not well equipped to support another
task model. Such a design methodology is not beneficial to the majority of real-time applications which have diversified real-time requirements. The incompatibility of different task models also makes it very difficult to maintain complex real-time systems whose real-time parameters continue to change because of new functionality.

Real-time kernel design have taken two approaches to providing more flexibility in the choice of scheduling algorithms. One solution is to implement multiple schedulers in the kernel, allowing applications to choose the scheduler that best suits their needs [Tokuda et al. 90, Tokuda et al. 87]. The other solution pushes the CPU scheduling algorithms into user space. The kernel makes upcalls to the application each time a scheduling decision is required [Anderson et al. 91]. Neither of these solutions provide architectural support for the application programmer to implement application-level scheduling policies. The second approach at least makes it possible but the user is forced to implement the entire scheduling algorithm.

1.2 Thesis Statement

An architectural solution that provides a common task abstraction as well as the support of application-level scheduling policies has significant advantages over the monolithic approach including flexibility, maintainability, and extensibility.

1.3 Contributions

The main contributions of this dissertation are ordered by their significance as follows:

1. The design and implementation of a common real-time task abstraction for flexible scheduling objectives.
1.3. CONTRIBUTIONS

This is the most significant contribution of the dissertation since the common real-time task abstraction is the main infrastructure which supports flexibility at both the kernel and the application level. While the existing real-time task models have been designed for specific scheduling problems, our task abstraction provides a basis for the implementation of complex scheduling algorithms. Our abstraction supports both tasks with real-time requirements and tasks without. It provides uniformity in the implementation of real-time and non-real-time tasks not merely in the system interface but also at the conceptual level. This allows for substantial flexibility in the development of complex, long-lived systems whose real-time characteristics evolve during their lifetimes. We have implemented our abstraction by using the Mach 3.0 kernel and the r-kernel as development platforms.

2. A design technique by which a complex scheduler is structured as a set of real-time tasks.

The design technique is significant in that it completes the architectural solution to scheduling abstractions. Our scheduling interface provides a means to structure a complex scheduling mechanism as a collection of cooperating tasks, thus offering a modular alternative to the implementation of the scheduling mechanism in a monolithic kernel. We provide the multitasking structures which are useful for the implementation.

3. Recognition and clarification of the common misconceptions in the design of real-time tasks and real-time task synchronization.

The conceptual analysis which provided the foundation of our design is also a significant contribution to the understanding of real-time task abstractions. The traditional tight coupling between task models and their underlying scheduling mechanisms has led to the approach of designing a real-time task model for a specific scheduling problem. In this approach, all real-time task primitives such as synchronization are designed to support the performance objective of the underlying scheduling algorithm. Although such an approach is well intended, it has reduced the value of
task models to nothing more than scheduling abstractions. We have analyzed existing real-time
task models and examined the useful concepts that have been undermined and misunderstood.
By separating the concepts in the design of task models, we have decoupled complex scheduling
abstractions from a more general and flexible task abstraction.

4. Recognition of subtle and complex problems in the implementation of the kernel schedulers of
two existing operating systems.

This is a relevant contribution to the development of modular, reliable real-time kernels. The
Mach 3.0 kernel and the r-kernel represent two different categories of kernel. While the Mach
3.0 kernel provides rich functionality to support the implementation of a time-sharing system, the
r-kernel has been designed to provide fast performance in a multiprocessor environment and is
much simpler. We have found some inconsistency and errors in the schedulers of both kernels,
especially the Mach 3.0 kernel, which contribute to unpredictable timing behavior. The way
scheduling code was organized in the Mach 3.0 kernel further added extra complexity to any
kernel modification. Our experience is useful to kernel developers and provides insight into the
practical problems beyond the theoretical aspects of scheduling.

5. Evaluation of our task abstraction using a multimedia application.

The significance of this work is in its contribution to the development of modern multimedia
applications. Human perception is very sensitive to slight variations in rhythmic patterns and other
aspects of timing. We have developed an application with a sound server for experimenting with
our task abstraction and for testing the performance of our kernel implementation. This application
gives a good example of the flexibility required in the underlying scheduling mechanism. Our
structuring of this application is also useful to the development of other multimedia applications.
6. A technique used to analyze the deterministic performance of the core scheduler of our task abstraction.

This technique is a relevant contribution to the schedulability analysis of realistic systems. We have examined the existing methods of schedulability analysis which are applicable to our scheduler and which do not impose strict assumptions on task arrival times. We have found some inadequacies in some of these methods and have extended them to consider a wider and more realistic application domain.

1.4 Synopsis of the Dissertation

Chapter 2 provides the background essential to the understanding of kernel design and real-time scheduling. Because of the many issues covered by this dissertation, a more in-depth description and analysis of individual issues is presented along with our solutions in the subsequent chapters.

Chapter 3 explores the concepts behind existing real-time task models and examines the strengths and weaknesses of these models. A discussion is given on the misconceptions which have contributed to the incompatibility of these task models deep at the conceptual level. The recognition of the value of discarding scheduling as the main motivation for real-time task structuring has steered us towards an architectural solution. A software architecture is outlined which supports the seemingly incompatible motives of real-time tasking models in a common framework with vastly improved expandability and modularity.

Chapter 4 describes our operating system architecture. The description proceeds from the bottom up — the core scheduler which provides task scheduling, the task model, and a facility called alarm capabilities. A significant contribution of our work is that each component of our architecture is simple yet adequate to contribute to the functionality and flexibility of the overall design. Since our core scheduler encompasses a number of efficient rules which can be used by the application as the sole
1.4. SYNOPSIS OF THE DISSERTATION

scheduling algorithms, this chapter includes a discussion of the existing methods for schedulability analysis using these rules. We have extended some of these methods to consider the server/client environment commonly used in many systems.

Chapter 5 discusses our implementation experience and performance measurements. The problems we have encountered with the Mach 3.0 kernel and the r-kernel are illustrative of the complexity of the kernel schedulers. They constitute strong evidence that our architectural approach of supporting application-level scheduling is more advantageous than the monolithic kernel approach. Some design tradeoffs in the implementation of the alarm capability mechanism are presented. Chapter 5 ends with a section on performance measurements which provide a basis to judge the practicality of our design.

Chapter 6 presents a collection of applications of our scheduling abstraction. This collection ranges from the complex scheduling mechanisms of existing monolithic kernels to novel application-level scheduling policies. They provide interesting examples of the utility of our architectural design, and, in particular, the scheduling abstraction. Our experimentation with a sound server has not only contributed to the evaluation of our design, but has also unexpectedly facilitated our kernel debugging as described in Chapter 5.

The final chapter draws together conclusions from preceding chapters and suggests directions for further research.
Chapter 2

Background

The concept of tasks has evolved in different directions in mainstream operating systems and real-time systems. During this evolution, the real-time community has developed its own terminology which is quite different from that of the rest of the operating systems community. Section 2.1 serves to provide an introduction to the concepts of tasks and to clarify the terminology used in this dissertation. Since real-time scheduling is an integral component of existing real-time task models, a broad discussion of the real-time requirements of applications is useful for the evaluation of these task models. Section 2.2 discusses real-time requirements and real-time scheduling algorithms. A number of questions are raised about the possibility of generalizing the real-time requirements of applications.

2.1 Task Models

A task model is by definition the model of computation supported by an operating system. In addition to the scheduling algorithm, the task model may include interfaces for task creation and destruction, task synchronization and communication, as well as error detection, reporting, and handling. The abstraction of tasks has evolved for many years, much inspired by the needs of distributed and parallel computing systems.
2.1. TASK MODELS

2.1.1 Processes and Threads

In mainstream operating systems, Unix has served as a standard to be compared with. It supports a process model where a process provides an abstraction of a single executing CPU with exclusive access to certain system resources such as memory and files. The Unix kernel manages all system resources and uses a time-sharing algorithm to schedule processes. Because of the introduction of distributed and parallel computers, the idea of structuring a program as one or more processes which cooperate with each other efficiently has become very attractive. The early design of Unix also allowed for process communication, but process communication was not considered a frequent operation, and context switches among processes were expensive.

The idea of inexpensive cooperating processes has inspired the notion of light-weight processes in operating systems such as Thoth [Cheriton 82]. A light-weight process also provides an abstraction of a single executing CPU, but it does not own any other system resources. Such a simplification in the notion of processes has contributed to a significant improvement in both performance and maintainability. Context switching times, as well as the process synchronization and communication overhead, have been made much less expensive. System services such as the file system are no longer implemented in the kernel, but rather, as processes on top of the kernel. This architectural shift has been proven to provide a substantial improvement in the maintainability and expandability of the operating system. The smaller and simpler kernel is called a microkernel, as opposed to the complex, monolithic Unix kernel.

In recent years, the term thread has been commonly used to refer to a similar concept as light-weight process. The Mach kernel, a microkernel developed by CMU, has been widely published and referenced. It provides an abstraction which consists of tasks and threads [Black et al. 92, Baron et al. 90]. In Mach, a task is an execution environment and is the basic unit of resource allocation. It consists of an address space and communication access to system and server facilities. A thread is the basic unit of execution.
2.1. TASK MODELS

It executes within the context of exactly one task. The notion of a Unix process is thus represented by a task with a single thread of control.

2.1.2 Tasks

The evolution of task models in real-time systems has been very different from the mainstream operating systems. The design of real-time kernels had been dominated by the requirements of embedded computer systems which had diversified architectures and often very limited memory. It was common to develop a small kernel for each embedded application so that optimizations in CPU and memory demands could be customized for the application. In order to optimize the use of memory, some small kernels had limited functionality and only supported a foreground task and a background task. This approach of specialized kernels is very expensive because of the need to consider on-going enhancement and maintenance costs. As embedded systems evolve and are upgraded with new functionality, the difficulty of maintaining a kernel for each application increases. It is now recognized that the savings in hardware costs are insignificant compared to the increased cost of software maintenance. Hence, the trend is to use the same kernel for a wider range of different products. The microkernel approach has been found very useful, and the kernel is often called a multitasking kernel. Some examples of these kernels are Harmony [Gentleman 87], pSOS+ [Thompson 90] and XMS [telesis 84]. The XMS kernel has been used not only as a kernel for embedded applications, it is also the kernel of the XMS distributed operating system which runs on workstations.

The multitasking kernels, however, only provide a simple CPU scheduler for task scheduling and are not concerned with the real-time requirements of the application. In real-time systems, an overriding concern is to achieve predictable performance. Deadlines should be satisfied, and any failure to satisfy a deadline should be considered an error. The prominent requirement of predictability has motivated a number of novel ideas in task scheduling, and has inspired an extensive redesign of real-time kernels. Fundamental concepts such as real-time tasks, real-time thread synchronization, and the run-time
handling of real-time errors have been studied.

The approach of supporting real-time tasks is not without its drawbacks. The notion of real-time tasks is tightly coupled with the underlying scheduling algorithm. However, scheduling algorithms differ so much from each other that a kernel designed to implement one algorithm must be completely rewritten to support another, and the incompatibility of real-time task models pushes for a fundamental restructuring in the application. Some scheduling algorithms consider all resources in the system such as CPUs, memory, and files. An inflexible, monolithic approach has again been used to implement these algorithms in the kernel. An additional consideration is that the need for real-time scheduling support is no longer restricted to embedded systems; general-purpose operating systems are now being asked to satisfy the evolving real-time requirements of applications such as multimedia and communication protocols. The approach of making frequent changes to a complex kernel in order to support individual applications does not comply with the requirement of having stability in the base layer software for all users.

2.1.3 Our Terminology

We use the real-time convention where a task represents a basic abstraction of execution. Our notion of tasks is thus similar to processes or threads. Since thread is the terminology used in both the Mach kernel and the r-kernel, we use the term thread when we discuss our implementation (see Chapter 4, 5 and 6). In our design, tasks and threads refer to the same concept. We also define some terms:

cooperative task model

An architectural model for software systems that involves multiple tasks communicating and synchronizing in order to achieve a common goal.

competitive task model

A task model where tasks do not cooperate through communication or synchronization. The only
interaction between tasks is competition for shared resources.

2.2 Real-Time Scheduling

Most research on scheduling tends to take a microscopic view of real-time requirements. This is largely due to the NP-completeness [Garey & Johnson 79] of the general resource scheduling problem. Also, the complexity of some scheduling problems can vary greatly when their assumptions are changed in small ways [Graham et al. 79]. Since the effectiveness of a heuristic algorithm and the assumptions it makes are closely coupled, it is essential to discuss the assumptions and objectives.

Real-time requirements in scheduling models appear to be polarized into the objective of meeting deadlines and the objective of responsiveness. These two requirements are often considered so dissimilar that only one of them is adopted. Before the discussions of these two different requirements, we introduce the common terminology in real-time scheduling.

2.2.1 Terminology

scheduling algorithm

An algorithm that determines how resources are to be shared among multiple tasks that require them. The primary resource is the CPU, but hardware resources such as I/O devices and software resources such as critical sections also require scheduling.

schedulability analysis

_Schedulability analysis_ refers to a method of verifying that a feasible schedule exists for a collection of tasks. Generally, schedulability analysis assumes the use of a specific scheduling algorithm.

response time

The _response time_ for a task refers to the elapsed time between the request to execute this task
2.2. REAL-TIME SCHEDULING

and the completion of its execution.

critical instant
A request to execute a task is subject to a delay which depends on the instantaneous load of the system. A critical instant for a task is a time instant at which a request for this task will have the longest response time.

critical time zone
A critical time zone for a task is the time interval between a critical instant and the completion of the execution of the task.

feasible
A schedule is feasible if all tasks in the schedule complete before their deadlines. A set of tasks is feasible on \( m \) processors if there exists a feasible schedule for this set of tasks on \( m \) processors.

off-line, on-line
An off-line scheduling algorithm assumes a complete knowledge of the arrival times of all tasks in advance. An on-line scheduling algorithm does not make this assumption; hence, it updates the schedule whenever a new task request arrives. Off-line and on-line schedulability analysis are defined in a similar way. An off-line scheduling algorithm is sometimes referred as pre-run-time since it is applied before the application runs.

optimal
A scheduling algorithm is optimal if it can produce a feasible schedule whenever such a schedule exists.

phasing
The response time for a task \( \tau \) is dependent on the arrival times of some other tasks. The phasings of these tasks relative to \( \tau \) refer to their relative arrival times.
2.2. REAL-TIME SCHEDULING

priority inversion, priority inheritance

In a preemptive priority system, a task can be preempted by any task of a higher priority. The priority inversion problem refers to the situation where a task is instead preempted by another task of a lower priority. This occurs when tasks of different priorities share the same resource and will be discussed further in Section 3.1.3. Priority inheritance algorithms aim at solving the priority inversion problem by having the tasks holding shared resources to inherit the priorities of waiting tasks.

2.2.2 Meeting Deadlines

In real-time systems, some activities must be completed before their deadlines in order to avoid associated penalties. These activities are often divided into hard real-time and soft real-time based on the severity of the associated penalty. The penalty for hard real-time activities not meeting their deadlines is potentially an intolerable degradation of system performance or even a fatal catastrophe. The penalty for soft real-time activities is much milder and is waived under certain circumstances. The scheduling algorithms for hard real-time activities have often been modified or extended to consider soft real-time activities.

Because of the severe penalty of missing hard deadlines, it is necessary to perform off-line schedulability analysis which uses the worst-case arrival times and resource requirements of activities to prove that all hard deadlines can be satisfied. Conservatively, resources are always pre-allocated for hard real-time activities to prevent resource contention.

The scheduling of soft real-time activities is more flexible than hard real-time activities and raises the following questions. Should the objective be to minimize the ratio of deadlines which are missed to deadlines which are met? Are the soft real-time activities weighted so that some deadlines are more important than others? Despite these questions, since the resource reservation approach for hard real-
time activities provides a solution to avoid failures caused by resource contention, a similar approach has been suggested for soft real-time activities in the form of a dynamic guarantee. In this approach, schedulability analysis is performed on-line to predict whether the available resources will be sufficient for a real-time activity to meet its deadline. If the result is positive, then the soft real-time activity would be guaranteed its worst-case resource requirement. Once a soft real-time activity is scheduled, it should not fail due to resource contention.

A common design approach has been to define a real-time task as one associated with a deadline and to schedule real-time tasks so that they can meet their deadlines. The main scheduling algorithms implemented in real-time kernels include:

1. **Earliest-Deadline-First**

   The earliest-deadline-first rule (EDF) schedules tasks according to non-decreasing deadlines [Blazewicz 76]. If a feasible schedule exists, then the EDF rule will find one. The real-time kernels that implement the earliest-deadline-first rule include CHAOS [Schwan et al. 90], ARTS [Tokuda & Mercer 89], RT Mach [Tokuda et al. 90], and some others [Halang & Stoyenko 91].

2. **Rate-Monotonic**

   The rate-monotonic assignment rule [Liu & Layland 73] assumes the use of a fixed priority preemptive scheduler. It is an optimal static priority assignment rule for periodic tasks with constant periods and execution times. Using this rule, periodic tasks with higher request rates are assigned higher priorities regardless of their execution times.

   The rate-monotonic rule is enhanced to deal with the priority inversion problem in [Rajkumar 91, Sha et al. 87]. The priority inversion problem occurs when tasks of different priorities share the same critical sections in a first-come-first-served order. This results in the possibility that higher priority tasks may be blocked by lower priority tasks for an indefinite period. Sha et al.
2.2. REAL-TIME SCHEDULING

propose a number of priority inheritance schemes where a task may be blocked by any number of higher priority tasks but will be blocked by at most one lower priority task, i.e., the one currently executing the critical section. When a task executing a critical section blocks other higher priority tasks, the execution of the critical section inherits the highest priority level among all the blocked tasks.

The ARTS and the RT Mach kernel support the fixed priority preemptive scheduler as an alternative to the earliest-deadline-first scheduler. The rate-monotonic rule and a priority inheritance protocol are used in the implementation of the object model of ARTS and the real-time thread model of RT Mach.

3. Resource Scheduling Heuristics

The earliest-deadline-first rule and the rate-monotonic rule do not deal with the problem of multiple resources. Some on-line resource scheduling heuristics define a task as a request for multiple resources. In this approach, a heuristic algorithm is used to search for an optimal schedule every time a new task request arrives. An example can be found in [Zhao et al. 87] which is implemented in the Spring kernel [Stankovic & Ramaritham 91].

The above scheduling algorithms can be used for both hard and soft real-time tasks. In the case of hard real-time tasks, off-line schedulability analysis using worst-case arrival times is necessary. For soft real-time tasks, on-line schedulability analysis provides a form of dynamic guarantee as discussed at the beginning of this section. Although these scheduling algorithms share the same objective of meeting deadlines, they have little else in common, thus contributing to the fundamental incompatibility among different task models. A further investigation into these real-time task models is provided in Section 3.1.

Some off-line schedulers only handle hard real-time tasks with predictable arrival times [Shepard & Gagné 91, Xu & Parnas 90]. They produce schedules which specify when each task should be executed
2.2. REAL-TIME SCHEDULING

and suspended. In this case, there is no need for a special on-line scheduling algorithm other than a run-time executive which uses a clock to execute tasks according to the schedule.

2.2.3 Responsiveness

The definition of a general performance metric for responsiveness is an open problem. The quantification of responsiveness has been found very difficult in the performance modelling and evaluation of complex systems. The concept of responsiveness, however, is essential in real-time systems. A common practice has been to use adhoc approaches to define and to achieve responsiveness for individual systems.

In performance modelling, responsiveness generally refers to response times of real-time tasks. There are cases where improving responsiveness is more natural than artificially imposing a deadline on a real-time activity. Even if soft deadlines exist, since they are not critical, the mean response time of soft real-time tasks is sometimes used as a performance metric. In an environment where both hard and soft real-time tasks exist, a common approach is to maintain the schedulability of hard real-time tasks while maintaining an expected response time of soft real-time tasks. For example, in [Lehoczky et al. 87] the rate-monotonic algorithm is enhanced to jointly schedule hard periodic tasks and soft aperiodic tasks.

In a broader sense, responsiveness is not restricted to just response times. For example, in order to gracefully degrade during transient overload, some algorithms attempt to schedule important tasks ahead of less important tasks. The performance objective does not need to include a measure of response times. The objective can be to satisfy all important deadlines and as many less important deadlines as possible. However, there is also the constraint that an important task should be scheduled as early as possible on condition that the number of missed deadlines would not increase. Capturing this general notion of responsiveness is difficult, nevertheless we believe that it is fundamental in most realistic problems.

In some situations, requiring both responsiveness and meeting deadlines may place conflicting
demands on resource contention. For example, a simple on-line scheduler has lower overhead than a complex on-line scheduler and is therefore more responsive. However, a complex on-line scheduler which finds a better approximate schedule will allow more deadlines to be satisfied. Another example is in adaptive real-time systems [Muntz et al. 91] where deadlines may be dynamically altered according to external events. If an external event happens at a time when the available resources are tight for satisfying the existing deadlines, should resources be preempted to handle this external event?

The open-endedness of possible solutions to responsiveness has been an important motive of our architectural support for application-specific scheduling mechanisms. The need to capture both the notion of deadlines and the notion of responsiveness in reactive and adaptive systems has motivated our framework for dynamic scheduling as described in Section 6.2.1.
Chapter 3

Real-Time Task Models

Existing real-time task models were designed to satisfy a variety of different real-time requirements and it is therefore not surprising that they are incompatible with each other. Their incompatibility makes it very difficult to maintain complex real-time systems whose real-time characteristics evolved during their lifetime. In this chapter, the design objectives, strengths, and weaknesses of various real-time task models are examined and compared. In particular, the concepts contributing to the various designs are investigated. Task models are classified into competitive and cooperative. In contrast to the common belief that they are fundamentally different, our conclusion is that although the design of these task models are incompatible with each other, their underlying concepts are not. Our work is the first to suggest the possibility of combining the concepts of various task models in a common design. This paves the way for an architectural design which supports evolving real-time requirements at the application level.

3.1 Competitive Task Models

3.1.1 Motivations and Objectives

The growth of the discipline of real-time computing has been driven by what are now called hard real-time systems [Stankovic 88]. A hard real-time system is defined as one in which the failure of an
activity to finish before a deadline can cause an intolerable degradation of system performance or even a fatal catastrophe. More recently, there is increasing consideration of soft real-time activities which are also given deadlines, but the penalty for failing to meet a deadline is much milder and is waived under certain circumstances.

An oft-cited requirement of hard real-time systems is that they satisfy the deadlines of all of the hard real-time activities and that they use a best-effort strategy to satisfy soft deadlines. Moreover, a common objective is to relieve programmers from the complicated task of resolving the scheduling problems of the entire system. The design of real-time task models is aimed at this objective and can be considered at two levels. First, at the programming level, the goal is to provide programming constructs that support timing correctness. Second, at the operating system level, the goal is to provide the appropriate scheduling support for the real-time task models so that the programmer only needs to be concerned with the application-level timing requirements.

These goals have so far motivated the approach of what this thesis calls competitive tasks. In this approach, a task is defined as a schedulable entity with resource requirements and a deadline. An on-line task scheduler uses the task definitions to predict whether a task can be scheduled to meet its deadline. The job of real-time programming is made easier since timing requirements can be specified in the program structure. Presently, a number of operating system kernels and languages have been developed to support various competitive task models.

The scope of problems that a competitive task model can deal with, however, is largely dependent on its task scheduler. Since most realistic scheduling problems are NP-complete, it is common to design a scheduling solution based on significant restrictions in the task model. This strategy has been successful in the realm of hard real-time tasks which benefit greatly from a guarantee of timing correctness. However, when soft real-time tasks are considered, the timing requirements are more demanding. Some on-going research has been to broaden the scope of timing objectives of real-time
tasks (see Section 3.1.4). Because of their assumption that the task scheduler should be responsible for satisfying these timing objectives, the limitations of their task schedulers again restrict their real-time task models.

Competitive task models share the assumption that timing correctness can be supported by complex task scheduling. This does not imply that a similar task interface can be used by different competitive task models, since they are likely based on different scheduling policies and timing requirements. A closer examination of three well-known competitive task models in Section 3.1.4 will reveal that they do not have much in common in terms of their task primitives and task structuring principles.

3.1.2 Supporting Timing Correctness

It is common in competitive task models that on-line schedulability analysis is used to provide some form of guarantee of timing correctness. Assuming that the worst case resource requirements of a task are known when it arrives, these requirements are used to predict whether this task can complete before its deadline. The guarantee of timing correctness is based on the strategy that only the tasks that pass the schedulability analysis are accepted.

The availability of an on-line schedulability check does not imply that off-line schedulability analysis is no longer needed. A real-time application typically consists of multiple tasks which arrive at different times and have different resource requirements. Before the application is executed, its overall resource requirements should be estimated and used in a schedulability analysis. This off-line schedulability check makes sure that all hard real-time tasks will meet their deadlines. An on-line schedulability check, on the other hand, provides a means for a soft real-time task to complete before its deadline once this task is accepted. Moreover, the arrival times of soft real-time tasks do not need to be predictable.

The ability to schedule soft real-time tasks is commonly considered an advantage over the traditional approach of using a time executive. Using a time executive requires that a complete knowledge of the
3.1. COMPETITIVE TASK MODELS

exact arrival times and worst case resource requirements of all task requests (both hard and soft real-time requests) be known in advance. Using this knowledge, an off-line scheduler produces a schedule which describes the times at which individual task requests should be executed. Hence, only a time executive is needed during run time.

In an uniprocessor environment, the most commonly used on-line scheduling algorithms are the earliest-deadline-first algorithm and the preemptive fixed priority scheduling algorithm. The earliest-deadline-first rule schedules tasks according to their non-decreasing deadlines. It will produce a schedule in which all tasks meet their deadlines, if such a schedule exists [Blazewicz 76]. Besides this interesting property, the earliest-deadline-first rule is better than other algorithms in that the deadline of a task is already specified in the timing objectives. No mapping from a timing objective into a different scheduling attribute such as a fixed priority is needed.

When compared with the earliest-deadline-first algorithm, the preemptive fixed priority algorithm may seem to be at a disadvantage since the latter is not optimal in scheduling and since a priority assignment rule is needed. However, the preemptive fixed priority algorithm is generally preferred over the earliest-deadline-first rule in the handling of system overload [Ramos-Thuel & Strosnider 91]. Also, [Liu & Layland 73] describes a rate-monotonic assignment rule which assigns priorities to periodic tasks. That same paper describes a simple way to test schedulability and proves that the rate-monotonic rule is optimal among all fixed priority assignment algorithms of periodic tasks that have constant execution time and whose deadlines are the same as their periods (and while this description may appear very restrictive – many hard real-time applications can be made to satisfy it without much trouble).

A well-known on-line scheduling algorithm designed for the multiprocessor environment is the resource scheduling heuristic [Zhao et al. 87] which is adopted in the Spring kernel [Stankovic & Ramamritham 91, Stankovic & Ramamritham 87]. This algorithm schedules task requests for simultaneous use of multiple resources which include CPUs. The preemptive fixed priority algorithm
has also been a subject of research interest in the multiprocessor case (e.g., [Dhall & Liu 78, Rajkumar et al. 88]).

3.1.3 New Approaches to Task Synchronization

In competitive task models, the timing of task synchronization events must be predictable by the task scheduler so that schedulability analysis can succeed. This is achieved by restricting the situations where task synchronization can happen, and by bounding the time spent on synchronization.

For example, in some resource scheduling algorithms (e.g., [Zhao et al. 87]), a task is guaranteed to have exclusive access to multiple resources during a specific time interval. Any access to a resource is thus synchronous. This type of task ordering is sometimes called *time-based task synchronization*.

In recent years, the sharing of critical sections among asynchronous tasks of different priorities has been a subject of active research. In the *priority inversion* problem [Sha et al. 87], a high priority task waiting to enter a critical section can be delayed by tasks of lower priorities which do not need this critical section at all. It happens when the critical section is being used by a low priority task, and the low priority task is preempted by newly arrived medium priority tasks. *Priority inheritance protocols* [Rajkumar 91] are designed to bound blocking time in accordance with task priority. In the basic priority inheritance protocol, a task which is already in a critical section inherits the highest priority of the tasks waiting to enter the same critical section. Moreover, the blocked tasks are queued in descending order of their priorities. This policy guarantees that a task can only be blocked by higher priority tasks and at most one lower priority for each critical section. When a priority inheritance policy is used to implement a task synchronization primitive, the synchronization mechanism is often called a *real-time synchronization* mechanism because of the predictable, bounded blocking time.
3.1. COMPETITIVE TASK MODELS

3.1.4 An Overview of Existing Models

The concept of timing predictability and the idea of using an on-line schedulability check to provide dynamic timing guarantees have motivated the design of various real-time task models. Despite their differences in detailed design and scope, these task models often have some common assumptions and share some common infrastructure. In order to avoid repetitious and distracting details, we have chosen three models as representatives for our discussion.

Spring

The Spring kernel [Stankovic & Ramamritham 91] is a well-known example of a real-time kernel which provides an on-line resource scheduling algorithm to the application programmer. In Spring, a task is characterized by its resource requirements, which include: a worst case execution time, a deadline, a preemptive or non-preemptive property, the maximum number and type of resources needed (which can be multiple CPU’s or memory segments), a precedence graph and a communication graph1. A task can be of one of three types: critical, essential, and unessential. Critical tasks are always preallocated resources so that they will run to completion. Essential tasks may arrive at unpredictable times and are handled by the on-line resource scheduling mechanism. In this mechanism, when a task arrives, the scheduling algorithm searches for a new schedule where the deadlines of all tasks can be satisfied. If a feasible schedule can be discovered, the newly arrived task is guaranteed to complete before its deadline. The guarantee consists of assigning to the task all the resources it requires when the task is activated. The resources are released only when the task completes. If the task does not finish before its deadline, a system error will be generated. Unessential tasks may also have resource requirements, but they are considered background tasks and are not offered performance guarantees.

The present Spring task model considers more scheduling problems than the earlier design described

1The communication can be asynchronous or synchronous.
3.1. COMPETITIVE TASK MODELS

in [Stankovic & Ramamritham 87] which was primarily concerned with essential tasks. Nevertheless, the Spring designers argue that no real-time synchronization mechanism is required in Spring, since any resource conflict can be avoided by scheduling tasks to run at different times if they contend for a given resource. An exception to this happens when it is necessary to synchronize with physical devices. To resolve this problem, the solutions offered by Spring include bounded waiting and polling. Blocked message passing is prohibited because of the possibility of unbounded blocking time. The Spring kernel has been implemented in a multiprocessor node consisting of four Motorola 68020-based MVME136A boards [Stankovic & Ramamritham 91]. A Spring scheduling co-processor has also been designed and implemented [Niehaus et al. 93].

Real-Time Mach

Real-Time Mach [Tokuda et al. 90, Nakajima et al. 93] is an extension of Mach [Black et al. 92] for real-time systems. One of the design goals of Real-Time Mach (or RT Mach) is to extend the basic Mach system functions to support real-time applications without changing the original interface of Mach. Other than the ability to support existing Mach applications, Real-Time Mach can be distinguished from other real-time kernels in its support of periodic threads and real-time thread synchronization. In Real-Time Mach, real-time threads have priority over non-real-time threads. A real-time thread may be defined as either periodic or aperiodic. A periodic thread is defined by its worst case execution time, period, start time, phase offset, and the task's semantic value. A aperiodic thread is defined by similar parameters except that period and phase offset are replaced by worst case interarrival time. A real-time thread can also be defined as either hard real-time or soft real-time. In the case of soft real-time threads, average values are used instead of worst case values in the definitions of deadlines and interarrival times.

Besides real-time threads, Real-Time Mach implements a real-time mutex variable mechanism and
3.1. COMPETITIVE TASK MODELS

a real-time IPC mechanism. Real-Time Mach uses a number of algorithms based on the rate-monotonic rule [Liu & Layland 73, Lehoczky et al. 87] to compute the priorities of real-time threads. In order to solve the priority inversion problem (Section 3.1.3), priority inheritance protocols are used in the implementation of real-time mutex variables and real-time IPC. In the basic priority inheritance scheme, the acceptor of a message inherits the priority of the invoker. The language support for this is described in [Ishikawa et al. 92].

The Model of Imprecise Computation

The model of imprecise computation [Lin et al. 87] is based on the idea that the intermediate or imprecise results of a computation can be useful even if the computation does not have enough time to complete its execution before its deadline. For example, in image processing, the contrast in a picture can be sharpened by enhancement routines in an iterative fashion. The time saved in producing imprecise results instead of perfect results can be better used to satisfy all deadlines. In a dynamic environment where the workload can change, this strategy provides a more flexible alternative to the assumption of using worst case execution times in schedulability analysis.

In [Lin et al. 87], a server/client model is described where the server process returns imprecise results to the client by a deadline. [Chung et al. 90] defines a task as consisting of a mandatory part and an optional part. The mandatory part must be completed before a deadline, while the optional part shares the same deadline but may remain unfinished. By separating the optional part from the mandatory part, different scheduling algorithms can be applied to the two parts to optimize the performance. For example, the mandatory parts may be assigned priorities according to the rate-monotonic rule. The optional parts may be scheduled according to the earliest-deadline-first rule.

The model of imprecise computation has not particularly addressed the problem of sharing resources among tasks. Presumably, priority inheritance protocols can be employed when the tasks are assigned
priorities according to the rate-monotonic rule.

3.1.5 Flexibility Issues

The competitive task models can be considered as a direct realization of scheduling policies, since each competitive task model is designed to support a scheduling policy, and each competitive task is under the absolute control of the task scheduler. This causes limitations in both task structuring and operating system design.

Task Structuring and Scheduling

In the competitive task models, a real-time task is commonly defined as a \textit{schedulable} entity with a deadline and worst case resource requirements. The methodology of structuring an application into multiple schedulable tasks is by nature tightly coupled with the underlying scheduling algorithm. Since the assumptions made by various scheduling algorithms are very different, the task structure designed for one task model is not easily convertible to the other task structures. For example, in Spring, a task is characterized by its worst case requirements for multiple resources. The exclusive access of a task to a resource is achieved by the implementation of a resource heuristic algorithm. In Real-Time Mach, on the other hand, a real-time thread is merely associated with its worst case CPU requirement. In order for real-time threads to share other resources such as critical sections, an additional real-time mutex variable mechanism is needed. The issue of incompatible task structures can be a serious drawback for complex, long-lived software systems where timing parameters change as functionality and hardware evolve. A scheduling algorithm which is sufficient at one time may need to be replaced later.

Besides the incompatibility of task structures and primitives, there is a lack of generality in the competitive task models. Some competitive task models allow real-time and non-real-time tasks to coexist in the same application. However, their task primitives are always specially designed for real-
3.1. COMPETITIVE TASK MODELS

time tasks and are not applicable to non-real-time tasks. This asymmetric approach greatly restricts the scope of the models. If real-time and non-real-time tasks are used differently depending on whether there is a resource contention problem, then there is no consideration of the threshold situations which are in between. An important role of real-time programs is to interact with real world events. Whether there is a resource contention problem depends on the selection of hardware and the functionality of the real-time program. When a real-time program evolves, old resource contention problems can disappear, and new resource contention problems can emerge.

Operating System Design

Competitive task models are commonly implemented in a monolithic operating system kernel. Even in the micro-kernel approach where the kernel is designed to provide the minimal set of services, the kernel is where the task abstraction and CPU scheduling algorithms are implemented. However, when the task scheduling algorithms are complex and involve the sharing of multiple resources among tasks in complex ways, the kernel no longer remains minimal. A new problem is that resource scheduling algorithms can differ so much from each other that a kernel designed to implement one algorithm must be completely rewritten to support another. Another problem is that this approach does not make it easier for an application programmer to implement an application-specific scheduling algorithm.

In order to offer more flexibility, some systems implement multiple schedulers in the kernel, and allow applications to choose the scheduler that best suits their needs [Tokuda et al. 90]. While this approach allows solutions among a pre-determined set of schedulers, it does not support the implementation of an application-specific scheduling algorithm. Another strategy is to move the scheduling algorithm from the kernel to user space [Anderson et al. 90]. In this approach, the kernel makes upcalls to the application each time a scheduling decision is required. There is, however, no architectural support for the implementation of the scheduling algorithm and task model. Each
3.2. A COMPARISON WITH THE COOPERATING TASK MODELS

implementation must be built "from scratch". Moreover, debugging is difficult because the scheduling algorithm is split between the kernel and user space.

3.2 A Comparison with the Cooperating Task Models

The cooperating task model has existed for a much longer time than the competitive task models. This section describes the concept of cooperating tasks, discusses their purposes and advantages, and explains why the cooperative task model lacks timing support relative to the competitive task models. Although the cooperative task model does not offer any timing support, it provides natural modularity for task structuring.

3.2.1 Cooperating Tasks

The idea of structuring programs as many concurrent and cooperating tasks has been a breakthrough from sequential programming in terms of software maintainability. In sequential programming, the code handling multiple events is often interleaved to meet ordering and/or timing requirements. This makes the control flow difficult to understand, and the timing dependencies among code sections difficult to determine. In the cooperative task paradigm, a task is defined as a unit of sequential and synchronous execution [Cheriton 82]. A program that must respond to asynchronous events is structured as one or more tasks. Each task responds only to the events synchronized with its execution. This synchronous nature of the individual tasks makes the control flow of the program more understandable. The concurrency of real world events is also directly reflected in the program structure. Examples of structuring a system as multiple cooperating tasks can be found in many operating systems. They are structured as a number of server tasks such as the file server and various device servers. Mach [Black et al. 92], Amoeba [Mullender et al. 90], V [Cheriton 88], XMS [telesis 84], and Thoth [Cheriton et al. 79].

\[In that paper, this model is introduced as the multi-process model, but since this paper is concerned with multiple task models, we will continue to use the term cooperative task model in order to reduce confusion.\]
3.2. A COMPARISON WITH THE COOPERATING TASK MODELS

are examples of such systems.

3.2.2 Task Structuring

A basic principle in designing a system in which multiple tasks cooperatively work toward a common goal is that the concurrent tasks should work autonomously until they need to share data or other resources (Figure 3.1). While actively sharing resources, these tasks must conform to some common rules of ordering. The mechanisms that support task synchronization and communication include semaphores, monitors, conditional critical regions, rendezvous and various types of message passing [Cheriton 82, Gentleman 81, Reed & Kanodia 79, Hoare 74, Dijkstra 68]. There has been much debate in comparing these mechanisms, the general conclusion is that the functionalities of many of them are equivalent. The choice of which mechanism to use is often a matter of programming style. Nevertheless, two resource models are prevalent: the server/client model and the mutual exclusion model. In the server/client model, a resource is managed by a server task. All client tasks access this resource by exchanging messages with the server task (Figure 3.2). The mutual exclusion mechanism may be implemented by using semaphores or critical regions. In the mutual exclusion model, a task may proceed to execute shared code in its own context. (Figure 3.3).
3.2. A COMPARISON WITH THE COOPERATING TASK MODELS

Figure 3.2: The Server/Client Model

Figure 3.3: The Mutual Exclusion Model
3.2.3 Operating System Design

The cooperating task model has had a great impact on operating system design. Historically, operating systems were first organized as single tasks that concerned themselves with resource management between competing application tasks. In recent years, software engineering concerns such as flexibility, understandability, and maintainability have fueled a movement toward modular, micro-kernel architectures. In this approach, the micro-kernel provides a task abstraction which includes a CPU scheduler and some task synchronization and communication mechanisms. The operating system is structured as a number of cooperating server tasks. Since many task synchronization and communication mechanisms are functionally equivalent, there are no restrictions on which particular mechanism a micro-kernel should provide.

3.2.4 Timing Support and Scheduling

When compared with the competitive task model, the main disadvantage of the cooperating task model is that it does not offer any direct support for meeting deadlines. This does not imply, however, that the cooperating task model has nothing to offer to real-time problems. On the contrary, there are many examples of real-time kernels that implement the cooperating task model, including Thoth [Cheriton et al. 79], Harmony [Gentleman 87], RT-Unix [Furht et al. 91], and commercial kernels such as pSOS+ [Thompson 90]. Experience has shown that the abstraction of cooperating tasks is very useful in implementing systems that must interact with the real world and respond to real world events. For many problems, it is sufficient to use a preemptive fixed priority scheduling algorithm to schedule tasks, and to use a first-come-first-serve (FCFS) rule to order tasks which are blocked by a synchronization primitive.

These solutions are not without drawbacks. First, it is necessary to assume that off-line schedulability analysis has been performed in order to ensure that all timing requirements can be fulfilled. Second,
when client tasks of different priorities share the same critical sections, the priority inversion problem 
(see Section 3.1.1) appears. Third, it is common to use a server task to manage a resource such as a device and to schedule client accesses to this resource. Different resources are managed and scheduled by autonomous servers. However, in the integrated resource scheduling approach, CPU time is a component of the resource request, and global resource availability must be considered by the scheduling algorithm. This centralized mechanism is opposite to the concept of autonomous servers.

The above issues are addressed in the design of competitive task models. Not only is on-line schedulability analysis supported in competitive task models, the failure to meet a deadline can also be detected. In Real-Time Mach (see Section 3.1.4), real-time tasking primitives are designed to resolve the priority inversion problem. In Spring (see Section 3.1.4), the kernel provides a resource heuristic algorithm, and the server/client model is simply not supported.

Hence, in terms of timing support and scheduling, competitive task models provide many advantages which are not offered by the cooperative task model. However, as discussed in Section 3.1.5, competitive task models also suffer from a lack of flexibility in task structuring and operating system design. Although lacking in some functionality, the cooperative task model offers natural modularity in both task structuring and operating system design. An interesting challenge is to combine the advantages of the two models in one single solution. We believe that this is possible and proceed to describe the architecture of one such solution.

### 3.3 A Structured Approach

Our approach is to first separate different design objectives and then to seek a structured way to satisfy each objective. In the previous sections, we have discussed the differences among the various task models. These differences alone are not sufficient to lead to the design of a modular real-time tasking solution. In Section 3.3.1, we discuss how subtle concepts should be separated from each other so
3.3. A STRUCTURED APPROACH

that they do not mingle together in the design. Section 3.3.2 describes the design resulting from such reasoning.

3.3.1 A Separation of Concepts

It is beneficial to separate some concepts related to real-time tasking:

The Cooperative Task Model from Scheduling

Since it is common to use the preemptive fixed priority scheduler to implement the cooperative task model, the priority scheduler is sometimes mistakenly considered as part of the cooperating task model. As discussed in Section 3.2.1, the design rationale of the cooperative task model is to provide natural modularity to the handling of asynchronous activities. It is concerned with the communication and synchronization among asynchronous tasks. However, it is not concerned with why and how tasks are asynchronous, or how tasks are scheduled overall. This property makes the cooperative task model more general than other models.

Task Synchronization from Scheduling

In a real-time system, there are always some external asynchronous events which are not under program control. In such situations, task synchronization is a result of the natural parallelism which is beyond the scope of scheduling. On the other hand, when asynchronous task requests to access a common resource are scheduled by a resource scheduler, it would be misleading to consider this scheduling problem as a task synchronization problem, since the tasks do not really synchronize with each other.

It is awkward to disguise a scheduling problem as a task synchronization problem. In the handling of external asynchronous events, the event types and the ordering of events are very important. On the other hand, when asynchronous resource requests are processed according to a scheduling algorithm,
3.3. A STRUCTURED APPROACH

the emphasis is on resource allocation instead.

CPU Scheduling from Resource Scheduling

The idea of using a server task to manage and schedule a resource has been proven very useful in many existing systems that employ the cooperating task model (see Section 3.2.2) [Black et al. 92, Mullender et al. 90, Cheriton 88, telesis 94]. The common design of using autonomous server tasks to schedule client requests, however, is not appropriate for the implementation of resource scheduling algorithms which must consider all resources including CPU time simultaneously (see Section 3.2.4). This results in the monolithic approach where a resource scheduling algorithm is implemented in the kernel. Because of the flexibility and modularity offered by the multitasking approach, however, we have attempted to seek for better solutions in other multitasking designs. A major problem is that the abstraction used for the scheduling of non-CPU resources is not applicable for CPU scheduling.

In order to illustrate this problem, let us consider the multitasking designs where a resource manager task is used to coordinate various server tasks via the use of a task synchronization and communication mechanism. For example, [Gentleman 81] describes a multitasking structure where an administrator task receives client requests and unblocks worker tasks to perform the actual service. The unblocking of the worker tasks by the administrator task is effectively a scheduling operation on the resources managed by the worker tasks. When the client requests consist of only non-CPU resources such as I/O devices, the administrator task can be used to implement the entire resource scheduling policy. However, if the CPU is one of the multiple resources to be considered, then the problem becomes much more complicated because a CPU scheduler is already used to implement the cooperating task model. A solution is to divide CPU time into discrete units and to manage these time units like other resources such as memory (e.g., [Bomberger et al. 92]). It is, however, not a general solution because time is infinite and continuous in many scheduling models, and because the performance overhead is excessive.
3.3. A STRUCTURED APPROACH

The realization that CPU scheduling is fundamentally different from the scheduling of other resources is useful in the design of a solution. It indicates that a novel way for a resource manager to schedule the CPU is needed. Our solution is to run an administrator task at a high priority and allow it to change the scheduling attributes of other tasks dynamically. The administrator task has complete knowledge of all the resources in the system and implements the resource scheduling algorithm. It orders client requests according to this algorithm and schedules each client request by raising either the priority of the client task selected to run or those of the server tasks selected to serve this client. These priorities are lowered after this client request is serviced. In this design, the support of dynamic changes of priorities is effectively a means of allowing cooperating tasks to participate in CPU scheduling.

3.3.2 An Architectural Design

A key conclusion from the proceeding sections is that cooperating tasks can be exploited to provide a more structured solution to real-time scheduling. Instead of implementing the entire resource scheduling mechanism in a monolithic operating system kernel, the mechanism can be implemented as a team of cooperating tasks. This approach allows for the implementation of various competitive task models on top of the same kernel. It also becomes convenient to develop and maintain application-specific scheduling algorithms, since they can be implemented at the task level instead of in the kernel.

Another advantage of the cooperating task model is that it provides a general mechanism for program structuring which is independent of real-time scheduling. Hence, an even more flexible solution is to use cooperating tasks to implement a real-time application. When the scheduling needs of the application evolve, only the scheduling servers need to be changed.

A software architecture that embodies these ideas is shown in Figure 3.4. A task is defined as a unit of sequential and synchronous execution in the same way as the cooperative task model. Task synchronization and communication primitives are provided so that asynchronous tasks may cooperate.
### 3.3. A Structured Approach

<table>
<thead>
<tr>
<th>Timing and Functional Specifications</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Cooperative Task Structure</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource Scheduling Algorithms</th>
<th>Operating System Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handling Mechanisms of Timing Errors</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task Synchronization and Communication Primitives</th>
<th>Kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Scheduling and an Interface for Upper Levels</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4: A Software Architecture
These primitives use a simple FCFS strategy and are not concerned with overall scheduling. The FCFS rule helps to preserve the order of external events as they are detected.

A significant characteristic of the CPU scheduler is that it provides an interface that allows upper levels, such as a resource scheduler, to control the CPU scheduler directly when this is necessary. If the resource scheduling algorithm needs a complete knowledge of all resources, the knowledge is maintained by a team of cooperating tasks instead of the CPU scheduler. Since the timing error handlers consume some CPU time, they are also scheduled according to the global resource scheduling scheme. The scope of timing errors does not need to be confined to resource contention problems. It can cover different types of errors such as when a device does not respond on time. Moreover, the timing error handlers can be used to simulate bounded waiting and other similar real-time operations.

Our architectural design supports both the competitive task models and the cooperative task model. It provides a common abstraction for the implementation of competitive task models. This common abstraction includes a scheduling interface which can be used by cooperating tasks. It naturally supports existing applications which use the cooperative task model. Since the effectiveness of this architectural support is best demonstrated by examples, a more detailed discussion is deferred until after the description of our CPU scheduler in Chapter 4. While timing error handling mechanisms are described in Section 4.2.3, the ways in which complex scheduling algorithms are implemented as cooperating tasks are described in Chapter 6.
Chapter 4

Design of an Operating System Architecture

Our goal is to design and implement an abstraction in the operating system for applications with evolving real-time requirements. This abstraction should allow for flexibility in the implementation of application-level schedulers and, at the same time, better modularity of the kernel and application. From the previous discussions in Chapter 3, we observe that the disassociation of the cooperating task model from a specific view of timing requirements allows the latter to be independently managed. This characteristic can be exploited to provide a more structured solution to the implementation of scheduling mechanisms and competitive task models. A brief outline of such an architectural solution is given in Section 3.3.2. In this chapter, we describe the various components of the architecture.

The design rationale of our architectural support is as follows:

1. The prime reason why a scheduling mechanism is implemented in the kernel is for efficiency. Complex scheduling mechanisms can easily be implemented as a set of cooperating tasks on top of the kernel scheduler. Often the cost of these scheduling mechanisms due to kernel overhead is a small fraction of the total cost of the scheduling. For such cases, there is no good reason why the flexibility of an application-level implementation should be traded for the efficiency of a
4.1. CORE SCHEDULING RULES

kernel implementation.

2. In real world modelling, we should consider both the external timing constraints imposed by the real world events and the internal timing constraints set by the real-time program itself. The best way to handle an external event is application-dependent, but the kernel should at least provide a means for the real-time program to detect external events and decide whether the internal timing constraints need to be modified. There should also be ways for the application to dynamically change scheduling decisions.

Our kernel provides a combination of scheduling rules which are designed to support stringent timing requirements in real-time applications. Section 4.4 discusses how the response times and the schedulability of tasks can be determined. We have improved some existing techniques to consider realistic systems which use the server/client model. The existing schedulability methods and our extended analysis are presented. Since our kernel scheduler also has the unconventional role of serving as a base layer of complex application-level scheduling mechanisms, the application domain is vast. The flexibility gained by using our architectural approach is discussed in Chapter 6.

4.1 Core Scheduling Rules

We provide a hierarchical CPU scheduling scheme to support our task model. In this hierarchical scheme, each task \( t_i \) is assigned a priority attribute \( p_i \). It may also be assigned an earliest starting time \( s_i \) and deadline \( d_i \). If \( s_i \) is specified, then task \( t_i \) will be suspended until \( s_i \). Tasks are scheduled strictly according to priority. A running task may be preempted by a higher priority task. At the same priority level, tasks with deadlines are scheduled by the earliest-deadline-first rule, and tasks without deadlines are scheduled by the first-come-first-served rule. When tasks with deadlines and tasks without deadlines share the same priority level, the former are scheduled before the latter. A diagram of the hierarchical scheduler is shown in Figure 4.1.
4.1. CORE SCHEDULING RULES

Tasks newly activated

Earliest starting time > now

Earliest starting time <= now

Level

N

N-1

. .

2

1

Tasks at the same priority level are ordered by EDF and FCFS

Time queue

Increasing priority

Figure 4.1: The Hierarchical Scheduler
As far as the computation overhead of the hierarchical scheduler is concerned, neither the fixed priority rule nor the earliest-deadline-first rule adds significant overhead to the task context switching time. We support task starting times as scheduling attributes in anticipation of the need to use absolute time as a point of reference for dynamic changes in scheduling attributes. When scheduling attributes are changed dynamically by an application, there is always the question of whether these changes may be delayed by temporary resource congestion. Using absolute time as a point of reference eliminates this undesirable delay.

Our scheduler does not perform any schedulability analysis before accepting a task for execution. A main reason is that internal timing constraints may be dynamically altered. Another reason is that performing dynamic schedulability checks at the low level is generally overly restrictive. First, if off-line schedulability analysis has already been performed but some tasks do not behave as expected, then there will still be a timing error. Second, if dynamic schedulability analysis does not have sufficient information, such as expected arrival times of important tasks and resource requirements of incoming tasks, then the results of these analysis will not be accurate. Since this information is application-dependent, customized schedulability analysis is more attractive in terms of efficiency and flexibility. Some applications may actually prefer to delay the schedulability check so as not to disturb the running task. Therefore, dynamic schedulability checking is not done in our core scheduler. It can, of course, be done by a higher level scheduler.

4.2 Real-Time Task Model

4.2.1 Definitions of Tasks

Our overriding desire for flexibility has steered us towards a more general and “natural” definition of tasks. In our task model, a task is a unit of sequential and synchronous execution. It is not necessarily the smallest unit subject to analysis for worst case execution time. Instead, a task may consist of a
number of consecutive segments which may be associated with different timing constraints and assigned
different scheduling attributes. Alternatively, a task may be viewed as being associated with one set of
scheduling attributes at any point in time, but the scheduling attributes can be changed as the execution
proceeds from one task segment to the next. The change in scheduling attributes is controlled by the
application.

Our definition of tasks is not concerned with address domain and other resources such as memory
and stacks. Using the terminology of some operating systems such as Mach [Black et al. 92], tasks
in our model are referred as threads. Without loss of generality, tasks and threads are interchangeable
terminology in this dissertation.

4.2.2 Time Capabilities

In our task model, each task is assigned a time capability containing the timing attributes of the task.
The task scheduler owns a real-time clock which is used to timestamp device interrupts and to define
earliest starting times ($s_i$'s) and deadlines ($d_i$'s). Device interrupts may be considered as the detection
and reporting of some simple real world events. Their timestamps may be used to compute the $s_i$'s and
$d_i$'s of the activities caused by these events. It is also possible that an application reads the real-time
clock during initialization and uses this time value as a reference to calculate the $s_i$'s and $d_i$'s.

The time capability of a task may be changed. For example, a periodic task with a mandatory part
and an optional part [Chung et al. 90] initially executes at the priority level of the mandatory part. Upon
completion of the mandatory part, this task may specify a deadline for its optional part and lower its
priority.

4.2.3 Alarm Capabilities

The hierarchical scheduler supports an alarm capability mechanism for the detection and handling of
timing errors. An alarm capability is associated with a thread $t_{alarm}$.u-italic A real-time application acquires an
alarm capability in the anticipation that a future program event $X$ may not happen on time. The earliest starting time of $t_{alarm}$ is assigned the latest time that this program event should happen. The application also supplies an error handling function which will be called by $t_{alarm}$, and specifies the priority and optional deadline of $t_{alarm}$. If event $X$ occurs before the alarm is sounded, the application cancels the alarm. If the alarm is cancelled on time, the alarm capability mechanism resets the scheduling attributes of $t_{alarm}$ so that it will not run. Otherwise, $t_{alarm}$ will run the error handling function at the specified time.

The alarm capability mechanism provides a flexible means to handle timing errors since the application controls both the error handling function and the scheduling attributes that apply to the error handler. This allows for the implementation of application-specific graceful degradation strategies during system overload. Moreover, a wide scope of timing errors can be accommodated, whether they are caused by resource congestion, or simply because some external events do not happen when expected.

4.2.4 Synchronization and Communication Primitives

The task synchronization and communication primitives of the cooperating task model are applicable in our abstraction. The FCFS rule employed by these primitives is used to preserve the order of external events as they are detected. Should there be any resource contention issue, it is considered at a higher level than the synchronization and communication primitives. By using this design strategy, application-specific scheduling can be performed at the task level. Examples are illustrated in Chapter 6.

4.3 Kernel Primitives and User Libraries

We have implemented our design by using the Mach 3.0 kernel [Black et al. 92] and the r-kernel [Ritchie & Neufeld 93] as development platforms. The most important primitive in our design is the one that changes the scheduling attributes of a thread (see Section 4.2.1). It has been implemented as
4.3. KERNEL PRIMITIVES AND USER LIBRARIES

a kernel primitive in Mach 3.0 and as a user call in the r-kernel. The other functions we provide are all implemented as user calls. A list of these functions is as shown in Figure 4.2 using the Mach 3.0 syntax.

/* scheduling attributes */
typedef struct sched_attr_t {
    struct time_value_t startingtime;
    int priority;
    struct time_value_t deadline;
} sched_attr;

/* default value when a thread does not have a deadline */
extern struct time_value_t nodeadline;

/* infinite starting time */
extern struct time_value_t infinite;

/* the list of functions */
typedef void (*func_t) ( int arg );
kern_return_t set_thread_sched_attr( thread_t thread_id, sched_attr attr);
thread_t rthread_create( thread_t *thread_id, sched_attr attr );
int alloc_alarm_pool( int num_of_alarms );
int get_alarm_from_pool( alarmID *id );
int return_alarm_to_pool( alarmID *id );
int set_alarm( func_t alarm_func, int arg, sched_attr attr, alarmID *id );
int reset_alarm( alarmID alarm_id );

Figure 4.2: Kernel Primitives and User Library Calls

The function set_thread_sched_attr() unifies all aspects of thread scheduling operations which include the suspend and resume operations. A thread is suspended when its starting time is set to infinite (see Figure 4.2). It is resumed when its starting time is set to the present time or a past time. The function set_alarm() sets the scheduling attributes of the alarm thread identified by alarm_id. The function alarm_func forms the body of the alarm thread and is passed arg as an argument. An alarm is cancelled
by calling \texttt{reset\_alarm()} which suspends the alarm thread if the latter has not started to run. If the alarm thread has started to run, this call does nothing.

4.4 \textit{Schedulability Analysis}

The complexity of scheduling problems can vary greatly when their assumptions are changed in small ways [Graham et al. 79]. In our design, the problem of schedulability is highly dependent on the applications themselves since different applications have different assumptions of task arrival times and task execution times. Moreover, an application-level scheduler may dynamically change the scheduling attributes of other tasks. Since our core scheduling rules do not assume any task characterization in terms of arrival times and execution times, the problem domain of schedulability is too general to consider.

We choose to discuss schedulability analysis primarily in the context of soft real-time systems where the assumptions of task characterization are not as rigid as hard real-time systems. Although schedulability analysis has been an active research topic, the focus has been on hard real-time systems. While hard real-time systems are designed to support a particular notion of timing correctness at all costs, the design of soft real-time systems often involve tradeoffs among cost, a stated timing objective, functionality, and maintainability. A direct application of the analysis of hard real-time systems to soft real-time system is awkward and prone to errors. A software designer needs to understand schedulability analysis and the impact of various assumptions in order to make engineering decisions.

This section explains several existing schedulability tests which are based on a variety of assumptions but nevertheless may be appropriate for soft real-time systems. Our discussion of these methods is also intended to provide insight into how new schedulability tests can be designed for other assumptions and how our core scheduling rules can be better used. Since there is no existing work which considers the server/client model commonly used in many soft real-time systems, we provide an analysis for these systems. Our analysis extends and improves the methods described in [Harter 87] and [Harbour et
al. 91]. The method in [Harter 87] restricts the priority of a server task to one level higher than its client tasks. The method in [Harbour et al. 91] allows the server task to have any priority, but it only considers periodic client tasks with constant period. Furthermore, a task is not permitted to wait for I/O before it finishes execution in a period. Neither of these methods are applicable to the common server/client system where client tasks may invoke server tasks of any higher priority, and where both client tasks and server tasks may wait for I/O at any time. Our schedulability techniques described in Section 4.4.2 and Section 4.4.5 are applicable to such an environment. We have discovered some pitfalls in the method described in [Harter 87] and have corrected them in our schedulability analysis.

The discussion in this section focus on the uniprocessor environment unless otherwise stated. Our hierarchical scheduler is analyzed in three ways: (1) when only the preemptive fixed priority rule is used, (2) when only the preemptive EDF rule is used, (3) when both rules are used.

This section uses the following notation:
Let \( T_1, T_2, T_3, \ldots, T_m \) be a set of \( m \) tasks to be scheduled.
Let \( e_i \) be the worst case execution time of task \( T_i \).
Let \( s_i, p_i \) and \( d_i \) be the earliest starting time, priority and deadline of \( T_i \) respectively.
If the tasks are periodic, then the period of \( T_i \) is denoted by \( P_i \).

4.4.1 Review of the Techniques Using the Priority Rule

The Rate-Monotonic Rule

A Schedulability Test

When priorities are assigned to periodic tasks using the rate-monotonic rule (see Section 2.2.2), we can use a very simple schedulability test which depends only on the utilization factor. The latter is defined as the fraction of processor time spent in the execution of the task set. For \( m \) tasks, the utilization factor is:
4.4. SCHEDULABILITY ANALYSIS

This set of $m$ tasks is schedulable if $U$ is less than the least upper bound $U_{\text{min}}$. [Liu & Layland 73]
prove the following:

**Theorem 4.1** For a set of $m$ tasks with fixed priority order, the least upper bound to processor utilization is $U_{\text{min}} = m(2^{1/m} - 1)$.

An Example of Some Common Techniques

The derivation of $U_{\text{min}}$ for $m$ tasks is not intuitive. Liu & Layland's proof for two tasks, however, provides insight into how timing can be affected by a higher priority task. It thus serves as a good example technique for schedulability analysis:

Let $\tau_1$ have a smaller period than $\tau_2$. It is assigned a higher priority because of the rate-monotonic rule. The critical time zone for $\tau_2$ occurs when $\tau_1$ and $\tau_2$ become ready at the same time (see Section 2.2.1). During the critical time zone for $\tau_2$, there are at most $\lceil P_2/P_1 \rceil$ requests of $\tau_1$. The question is how large the execution time of task $\tau_2$ can be increased to fully utilize the available CPU time within the critical time zone. There are two cases:

1. The execution time $e_1$ is short enough that all requests of $\tau_1$ are completed before the next request of $\tau_2$. When the total length of all periods of $\tau_1$ within $\tau_2$ is subtracted from the period of $\tau_2$, the time left gives a constraint for $e_1$:

   $$e_1 \leq P_2 - P_1 \left\lfloor \frac{P_2}{P_1} \right\rfloor$$
The largest possible value of \( e_2 \) is:

\[
    e_2 = P_2 - e_1 \left[ \frac{P_2}{P_1} \right]
\]

The processor utilization factor decreases monotonically in \( e_1 \):

\[
    U = \frac{e_1}{P_1} + \frac{e_2}{P_2} = 1 + e_1 \left[ \frac{1}{P_1} - \frac{1}{P_2} \left[ \frac{P_2}{P_1} \right] \right]
\]

2. The execution of the \([P_2/P_1]^{th}\) request for \( \tau_1 \) overlaps the second request for \( \tau_2 \). In this case we have

\[
    e_1 \geq P_2 - P_1 \left[ \frac{P_2}{P_1} \right]
\]

The largest possible \( e_2 \) uses up all time not occupied by \( \tau_1 \) before the \([P_2/P_1]^{th}\) request:

\[
    e_2 = (P_1 - e_1) \left[ \frac{P_2}{P_1} \right]
\]

The processor utilization factor is monotonically increasing in \( e_1 \):

\[
    U = \frac{P_1}{P_2} \left[ \frac{P_2}{P_1} \right] + e_1 \left[ \frac{1}{P_1} - \frac{1}{P_2} \left[ \frac{P_2}{P_1} \right] \right]
\]

The minimum of \( U \) occurs at the boundary between the above two cases for:

\[
    e_1 = P_2 - P_1 \left[ \frac{P_2}{P_1} \right]
\]

By inserting this value for \( U \) in the first case,

\[
    U = 1 - \frac{P_1}{P_2} \left[ \left[ \frac{P_2}{P_1} \right] - \frac{P_2}{P_1} \right] \left[ \frac{P_2}{P_1} - \frac{P_2}{P_1} \right]
\]

For brevity, let \( I = [P_2/P_1] \) and \( f \) be the fractional part of \((P_2/P_1)\). The above equation can be rewritten as:

\[
    U = 1 - \frac{f(1-f)}{I + f}
\]
Since $U$ is monotonically increasing with $I$, minimum $U$ occurs at the smallest possible $I$ which is 1. Minimizing $U$ over $f$, we obtain

$$f_{\text{min}} = 2^{1/2} - 1$$

The minimum value of $U$ is

$$U_{\text{min}} = 2(2^{1/2} - 1) \approx 0.83$$

For large task sets, the least upper bound to processor utilization approaches $\ln 2$. Since this bound represents the worst case estimate, a better utilization can be obtained in actual systems, but in order to verify the better utilization, a different schedulability test is needed.

**Response Times for Cyclic Tasks**

**Assumptions**

A level-driven analysis is described in [Harter 87] for realistic real-time systems. It considers cyclic tasks which repeatedly execute and then wait voluntarily for an external event. In the simplest case, a cycle consists of an execution phase and a waiting phase. Multiple execution phases and waiting phases in the same cycle are also permitted. There are a lower bound and an upper bound on the duration of the cycle. Each task is associated with one priority level, but procedural calls to the next higher level are permitted. At each priority level, there can be more than one task. Since the priority of the caller is raised during the interlevel procedure call, the call is made atomic with respect to the calling level.

**Dilation in Execution Time due to Higher Priority Tasks**

The analysis provides an upper bound and a lower bound to the response time at each priority level. Since our emphasis is on schedulability, we only consider the worst case response time in this section. A main strategy of the level-driven analysis is to derive the execution time for a particular code sequence in the absence of higher priority tasks and then increase it by the amount that higher priority tasks would
4.4. SCHEDULABILITY ANALYSIS

use up during that time. The slowing effect due to all higher priority tasks is considered as a whole. An iterative method is provided which refines the approximation of this slowing effect until the worst case bound is derived. An example of the slowing effect due to a higher priority task is shown in Figure 4.3. The CPU utilization of the higher priority task is $2/4 = 0.5$. Since the remaining CPU bandwidth is

(i) In the absence of a higher priority task, an execution code sequence $X$ at a level takes 5 time units to finish.

The higher priority task starts at the same time as the execution code sequence $X$.

Figure 4.3: Slowing Effect due to a Higher Priority Task

(ii) A higher priority task takes 2 time units to execute, and the shortest cycle is 4 time units. $X$ now takes 11 time units to finish.

The higher priority task starts at the same time as the execution code sequence $X$.

Since $1 - 0.5 = 0.5$, the average time for the execution code sequence $X$ to finish is $5/0.5 = 10$ time units. Since 10 time units are long enough for the higher priority task to start the third cycle, the execution code sequence $X$ should be slowed down by $3 \times 2 = 6$ time units. It thus takes $5 + 6 = 11$ time units for $X$ to finish.
4.4. SCHEDULABILITY ANALYSIS

An Iterative Algorithm

Let $n$ be the time it takes for an execution code sequence at level $m$ to finish in the absence of any higher priority tasks. Let $n'$ be the time it takes in the presence of all higher priority tasks. Let $C_{\text{min}_i}$ be the shortest time between two occurrences of $\tau_i$. Assuming that there are no inter-level procedure calls, the iterative algorithm to compute $n'$ is as follows:

1. Compute

$$\sum_{i=1}^{m-1} \frac{e_i}{C_{\text{min}_i}}$$

If it is greater than 1, then the CPU utilization of the higher priority tasks is too high, and the algorithm should terminate here. Otherwise, compute an approximation of $n'$:

$$n'_{\text{approx}} = n \left[ 1/ \left( 1 - \sum_{i=1}^{m-1} \frac{e_i}{C_{\text{min}_i}} \right) \right]$$

2. Assign

$$n'_{\text{approx}} = n + \sum_{i=1}^{m-1} \left[ \frac{n'_{\text{approx}}}{C_{\text{min}_i}} \right] e_i$$

If

$$n'_{\text{approx}} < n + \sum_{i=1}^{m-1} \left[ \frac{n'_{\text{approx}}}{C_{\text{min}_i}} \right] e_i$$

then go back to (2) and do it again. Otherwise, set $n'$ to $n'_{\text{approx}}$. 

4.4. SCHEDULABILITY ANALYSIS

Procedure Calls to the Next Higher Level

If the execution code sequence at level $m$ consists of calls to procedures at level $m - 1$, then the code sequence is divided into subsequences and analyzed, as illustrated by the example in Figure 4.4.

Denote the dilated execution time starting from P to Q by $e'_{PQ}$.

A:
... statements ... $e'_{AB}$
B:
call proc1; $e'_{BC}$ - dilated by levels higher than priority $m - 1$
C:
... statements ... $e'_{CD}$
D:
call proc2; $e'_{DE}$ - dilated by levels higher than priority $m - 1$
E:
... statements ... $e'_{EF}$
F:

$e'_{AF} = e'_{AB} + e'_{BC} + e'_{CD} + e'_{DE} + e'_{EF}$

Figure 4.4: Code Sequence containing Inter-Level Procedural Calls

Derivation of Cycle Times

The cycle times can be considered in one of two ways. If a cycle is triggered by an external event which is independent of any prior code execution, then the cycle time is part of the environment and must be given. Otherwise, the cycle is dependent on the execution time and the waiting time. For example, a task can initiate an I/O operation and then wait on the device signal. The shortest cycle time $C_{min_i}$ can be computed using the shortest execution time $e_{min_i}$ and the shortest waiting time for I/O $IO_{min_i}$:

$$C_{min_i} = e_{min_i} + IO_{min_i}$$
The longest cycle time is also important since it represents the worst case frequency at which a task may execute. It is assumed that no overtaking can happen at the same priority level; that is, no task can have two chances to run before a given task has a chance. The worst case happens when the execution is delayed by all tasks at the same priority level, as well as a procedure call by the next lower level:

\[ C_{\text{max}} = e_i + IO_{\text{max}} + \sum_{j \neq i, p_j = p_i} e_j + \max \text{ time for a procedure call at level } m \]

In the presence of higher priority tasks, the execution time in the above formula is subject to dilation, and the dilated result can be computed using the iterative method described earlier.

Application to Real Systems

Unlike most existing schedulability tests which focus on periodic tasks with constant periods, Harter’s method considers cyclic tasks and addresses a fundamental concern of existing software systems – the derivation of the longest possible cycle time when I/O is initiated by software. However, some further improvements in the modelling and in the approximation method are necessary. The assumption that inter-level procedure calls can only be made to the next higher priority level is too restrictive. Also, when an execution code sequence is divided into subsequences for analysis, the iterative algorithm is applied to each subsequence (see Figure 4.4). The iterative algorithm, however, assumes the worst case phasings of higher priority tasks relative to the subsequence being analyzed. An overly pessimistic approximation is thus derived when the estimates of all subsequences are added together. These problems are addressed by the techniques described in [Harbour et al. 91], which, however, only considers periodic tasks and does not permit multiple waiting phases in a period. When multiple waiting phases are permitted, the dilation is rather complex, and Harter’s analysis does not cover all worst cases either. Our analysis for a server/client model makes more flexible assumptions than Harter’s analysis and provides more accurate bounds than Harter’s. This is accomplished by modifying some of Harbour et al.’s techniques.
Periodic Tasks with Varying Execution Priority

Assumptions

A comprehensive schedulability test for periodic tasks with varying execution priority is described in [Harbour et al. 91]. Periodic tasks are decomposed into serially executed subtasks, where each subtask is characterized by an execution time and a fixed priority, and is permitted to have a deadline.\(^1\) The schedulability test determines whether every subtask can meet its deadline.

There is no restriction imposed on the ways priority can be varied. A task initiated by an interrupt is considered to execute at the interrupt level and then at a lower priority. When priority inheritance protocols are used, the priority of a task accessing a critical section is temporary raised. A strict assumption of the analysis, however, is that tasks do not suspend themselves at any instant between their activation and their completion.

In this analysis, each periodic task \(\tau_i\) is composed of subtasks \(\tau_{i1}, ..., \tau_{im(i)}\). Each subtask \(\tau_{ij}\) is characterized by a worst case execution time \(e_{ij}\), a deadline \(d_{ij}\) which is relative to the arrival time of \(\tau_i\), and a fixed priority \(p_{ij}\). The minimum priority of all subtasks of \(\tau_i\) is denoted by \(p_{\text{min}}\). The period of \(\tau_i\) is denoted by \(P_i\). The worst case execution time of \(\tau_i\) is denoted by \(e_i\) which is also \(\sum_{j=1}^{m(i)} e_{ij}\). The deadline of \(\tau_i\) can exceed the period of \(\tau_i\).

The Canonical Form

The analysis is simplified by transforming each task to its canonical form which consists of consecutive subtasks which do not decrease in priority. The basic rule used in the transformation is to decrease the priority \(p_{ik}\) to \(p_{il}\) if \(l \geq k\) and \(p_{ik}\) is a higher priority than \(p_{il}\). The completion time of a task is the same as that of its canonical form. This can be observed by considering two consecutive subtasks \(\tau_{ij}\) and \(\tau_{ij+1}\) of strictly decreasing priority. Let \(f_{ij-1}\) and \(b_{ij+1}\) be respectively the completion time of

\(^1\)Note that a task is scheduled according to its priority, not its deadline.
4.4. SCHEDULABILITY ANALYSIS

\( \tau_{i,j-1} \) and the start time of \( \tau_{i,j+1} \). The interval \([f_{i,j-1}, b_{i,j+1}]\) consists of the execution of \( \tau_{i,j} \) and other tasks of higher or equal priority than \( \tau_{i,j+1} \). At time \( b_{i,j+1} \), all work of equal or higher priority than \( \tau_{i,j+1} \) should have finished, and the subtask can start to execute. If the priority of \( \tau_{i,j} \) is decreased to that of \( \tau_{i,j+1} \), then exactly the same work will be done during this interval, although possibly in another order. By repeatedly applying this reasoning, the completion time of task \( \tau_i \) is proved to be the same as its canonical form.

*Busy Period*

A stepwise procedure determines whether task \( \tau_i \) can meet its final deadline. The task is first transformed into its canonical form. The procedure determines the latest completion time of the first subtask. It then iteratively determines the latest completion time of the \((j+1)\)th subtask as a function of the \(j\)th subtask until the latest completion time of the final subtask has been determined. It is, however, insufficient to consider the completion time of the first instance of task \( \tau_i \). An example is illustrated in Figure 4.5. The criterion to ensure that a sufficient number of instances have been considered is based on the concept of *busy period*. The busy period idea was first exploited in [Lehoczky 90] for the computation of worst case bounds in fixed priority scheduling, assuming that periodic tasks can have deadlines exceeding their periods. In [Harbour et al. 91], the busy period idea is modified so that varying priorities are also considered. A \( \tau_i \)-busy period is a time interval within which the CPU is 100% utilized by tasks running at priority level \( p_{min,i} \) or higher. The busy period begins and ends with *idle instances* where no task runs at \( p_{min,i} \) or higher. The busy period has a number of interesting properties. Any instance of \( \tau_i \) started during the busy period is completed during the busy period. Moreover, all work of priority level \( p_{min,i} \) or higher which is ready at some time during the busy period is also finished by the end of the busy period. In [Lehoczky 90] pp.203-204, it has been proved that the longest response time of \( \tau_{im(i)} \) occurs during the \( \tau_i \)-busy period initiated by \( \tau_i \). It is thus necessary and sufficient to compute the responses times of all instances of \( \tau_{im(i)} \) during the busy period.
4.4. SCHEDULABILITY ANALYSIS

Task A - one subtask

Task B - two subtasks

Task A and B, \( P_{22} > P_1 > P_{21} \)

Figure 4.5: Completion Time affected by an Earlier Instance

\( t_1 \) completion time of the 1st instance of task B
\( t_2 \) completion time of the 2nd instance of task B

\( t_2 > t_1 \) since the high priority subtask of task B causes a delay to task A which in turn slows down the 2nd instance of task B
4.4. SCHEDULABILITY ANALYSIS

Slowing Effects

Assume that \( \tau_i \) is in canonical form. The slowing effect of other tasks on \( \tau_{i1} \) can be grouped into three categories: singly preemptive, multiply preemptive, and blocking. A singly preemptive task relative to \( \tau_{i1} \) starts with a higher priority than \( p_{i1} \) but later has a lower priority than \( p_{i1} \). It is singly preemptive since it can only preempt \( \tau_{i1} \) once. A multiply preemptive task relative to \( \tau_{i1} \) has all its priorities higher than \( p_{i1} \) and can preempt \( \tau_{i1} \) multiple times. A blocking task relative to \( \tau_{i1} \) contains two consecutive subtasks. The first subtask has a lower priority than \( p_{i1} \), and the second has a higher priority than \( p_{i1} \). Task \( \tau_i \) can only be subject to the blocking effect of one task which executes at a higher priority than \( p_{i1} \) when \( \tau_i \) arrives. In order to demonstrate the various slowing factors, tasks are categorized according to the way priorities are varied. An \textit{H-segment} is defined as a sequence of consecutive subtasks which have priorities greater than or equal \( p_i \). An \textit{L-segment} is defined as a sequence of consecutive subtasks which have priorities strictly less than \( p_i \). Multiply preemptive tasks are type \((H)\). Singly preemptive tasks are either \((HL)^+\) or \((HL)^+H\). Tasks which are solely blocking are \((LH)^+L^0\). The + sign denotes one or more patterns. The 0 sign denotes zero or one pattern.

The canonical form simplifies the consideration of the singly preemptive effect and the blocking effect. Since the canonical form starts with a lowest priority, the blocking effect only needs to be considered once for the entire task. A singly preemptive task relative to \( \tau_{i1} \) is also singly preemptive relative to the entire task. Multiply preemptive tasks relative to \( \tau_{i1} \), on the other hand, can become singly preemptive relative to \( \tau_{i2} \). Since only one blocking effect needs to be considered, the longest one should be chosen. A special consideration is when a task can be classified as singly preemptive or blocking. If the task is of type \((HL)^+\), it can be either singly preemptive or blocking but not both. In order for the task to be blocking, an inner \textit{H-segment} should start to execute when task \( \tau_i \) arrives. Once this segment completes, the next \textit{L-segment} has a lower priority than \( p_{i1} \) and cannot run during the \( \tau_i \)-busy period. If the task is of type \((HL)^+H\) and an inner \textit{H-segment} is chosen for blocking,
then the task cannot be considered for its preemptive effect either. However, if the last \textit{H-segment} is chosen for blocking, then both the singly preemptive effect and the blocking effect can be considered in the worst case. It is thus possible to determine:

\( B_i \) \hspace{1em} the longest blocking time

\( SP_{i1} \) \hspace{1em} the set of singly preemptive tasks relative to \( \tau_{i1} \)

\( MP_{i1} \) \hspace{1em} the set of multiply preemptive tasks relative to \( \tau_{i1} \)

For task \( \tau_s \) in \( SP_{i1} \), let \( e_s^h \) be the execution associated with the leading \textit{H-segment} of \( \tau_s \).

\textit{A Stepwise Procedure}

Once the blocking and preemptive effects have been analyzed, the stepwise procedure for determining latest completion times proceeds as follows:

1. Determine the \( \tau_i \)-busy period and the number of instances of \( \tau_i \) in the busy period.

The length of the \( \tau_i \)-busy period is the sum of the longest blocking time, the slowing effect due to the multiply preemptive tasks, the execution time of the leading \textit{H} segments of the singly preemptive tasks, and the execution time of all instances of \( \tau_i \) in the \( \tau_i \)-busy period. \( L_i \) is the minimum over all \( t > 0 \) such that

\[
B_i + \sum_{\tau_m \in MP_{i1}} \left\lfloor \frac{t}{P_m} \right\rfloor e_m + \sum_{\tau_s \in SP_{i1}} e_s^h + \left\lfloor \frac{t}{P_{i}} \right\rfloor e_i = t
\]

The number of instances of \( \tau_i \) in the busy period is:

\[
N_i = \left\lfloor \frac{L_i}{P_{i}} \right\rfloor
\]

2. Determine the latest completion time of the first subtask and then iteratively determine the latest completion of the \((j + 1)^{th}\) subtask as a function of the \(j^{th}\) subtask.
4.4. SCHEDULABILITY ANALYSIS

Let $E_{ij}^k$ be the latest completion time of the $k^{th}$ instance of subtask $\tau_{ij}$. $E_{i1}^1$ is the sum of the blocking time, the slowing effect due to multiply preemptive tasks and singly preemptive tasks, and the execution time of $\tau_{i1}$. $E_{i1}^k$ is the sum of $E_{i1}^1$ and the total execution time of $(k-1)$ instances of $\tau_{i1}$. $E_{i1}^k$ is the minimum over all $t > 0$ such that

$$B_i + \sum_{\tau_m \in MP_{i1}} \left[ \frac{t}{P_m} \right] e_m + \sum_{\tau_s \in SP_{i1}} e_s^h + (k-1)e_i + e_{i1} = t$$

The latest completion time of $\tau_{i2}$ is computed based on that of $\tau_{i1}$. The singly preemptive tasks relative to $\tau_{i1}$ do not need to be considered again since $\tau_{i2}$ has a higher priority than $\tau_{i1}$. Some multiply preemptive tasks relative to $\tau_{i1}$ become singly preemptive relative to $\tau_{i2}$.

$$SP_{i2} = \{ \tau_s \mid (\tau_s \in (MP_{i1} - MP_{i2})) \land (\exists l (p_{s1}, ..., p_{sl} \geq p_{i2}) \land (p_{sl+1} < p_{i2})) \}$$

$E_{i2}^k$ is the minimum over all $t > 0$ such that

$$E_{i1}^k + \sum_{\tau_m \in MP_{i2}} \left[ \frac{t}{P_m} \right] - \left[ \frac{E_{i1}^k}{P_m} \right] e_m + \sum_{\tau_s \in SP_{i2}} \min \left( 1, \left[ \frac{t}{P_s} \right] - \left[ \frac{E_{i1}^k}{P_s} \right] \right) e_s^h + e_{i2} = t$$

In the above equation, since $E_{i1}^k$ is used as the starting point for calculation, only the preemptive effect after the completion of $\tau_{i1}$ is considered in the two summations. The min in the second summation ensures that singly preemptive tasks can have at most one preemptive effect. By applying the same algorithm to other subtasks, we can compute the latest completion time of the last subtask of $\tau_i$. 
4.4.2 An Extended Analysis Using the Priority Rule

Motivation

We have improved the analysis described in [Harter 87] and [Harbour et al. 91] to consider a common server/client model used in existing systems. In these systems, the preemptive fixed priority scheduling rule is used to schedule server and client tasks (see Section 3.2.2). Server tasks are often autonomous in their management of devices and other shared resources, and run at higher priorities than client tasks. All client tasks access resources by invoking server tasks or exchanging messages with server tasks. The execution path associated with a client request can be considered as starting from a client priority, increasing to a server priority when a server is invoked, and then decreasing back to the client priority when the server replies to the client. Both server and client tasks can suspend and wait for events such as the completion of I/O.

There has been very little work which analyzes the response times in such an environment. An understanding of how response times are affected, however, is becoming important because of the increasing need to run real-time applications such as multimedia in the same environment as complex non-real-time servers and applications. Soft real-time applications have flexible timing requirements and can tolerate a certain level of system latency. However, it is still necessary to determine the impact when system servers of high priorities are shared among real-time and non-real-time applications. A special system server is the kernel which runs at a higher priority than all tasks and which can be invoked by both real-time and non-real-time tasks.

Existing work considers a more restrictive environment. The analysis described in [Harter 87] considers cyclic tasks which can make inter-level procedure calls. Since server tasks run at higher priorities than clients, the invocation of a server can be treated as an inter-level procedure call in Harter’s analysis. However, Harter’s assumption that inter-level procedure call can only be made to the
next higher priority level is too strict in most realistic situations. The analysis described in [Harbour et al. 91] allows for much flexibility in how priorities are varied. The assumption that tasks do not suspend after their activation and before their completion makes it inappropriate to apply the analysis when tasks need to perform I/O or to wait for some events before continuing execution.

**Assumptions**

Our analysis considers cyclic tasks since they form the majority of tasks in soft real-time systems. A cycle consists of at least an execution phase and a waiting phase. In more complex situations, a cycle can have an interleaving of multiple execution phases and waiting phases. We assume that a task is associated with a physical resource such as a stack. Thus, the deadline of a cyclic task should not exceed its period. When it is necessary to streamline execution and waiting by parallel activities, they are performed by two different tasks. Interrupts caused by spontaneous events in the physical environment are considered as cyclic tasks running at high priority levels. Server tasks which are activated by client requests, on the other hand, are not analyzed as individual cyclic tasks. Their execution paths are considered as part of the execution paths of their cyclic clients. It is assumed that a task can only obtain service from another task of a higher priority.

The assumptions of our analysis are as follows. Assume all tasks are independent of each other. Let $P_i$ be the shortest time between two occurrences of $\tau_i$. Let task $\tau_i$ consist of $m(i)$ consecutive segments: $s_{i1}, s_{i2}, \ldots, s_{im(i)}$. Each segment is either an execution segment or an I/O segment. An execution segment $s_{ij}$ is associated with a constant priority $p_{ij}$ and a worst case execution time $e_{ij}$. An I/O segment $s_{ik}$ represents a waiting phase and is defined by the longest waiting time $IO_{ik}$. Any consecutive I/O segments are grouped into a single I/O segment. The shortest waiting time is assumed to be 0. Any consecutive execution segments which have the same priority are grouped into a single segment. Some restrictions on how priority is varied apply. First, the priority of the first execution
segment is equal to that of the last execution segment. Second, this priority is the lowest possible among all execution segments of the same task.

In realistic soft real-time systems, there can be more than one possible execution path for a task. The definition that task $\tau_i$ consists of $m(i)$ consecutive segments may seem overly restrictive. The complexity of non-real-time software running at low priorities also makes it impractical to assume accurate estimates of their worst case execution time. Our approach to analyzing such a complex system is to rely on as few assumptions as possible. When the longest completion time of $\tau_i$ needs to be determined, it is reasonable to analyze each of its execution paths individually and to rely on the worst case estimates of $\tau_i$'s I/O waiting time and execution time. There may exist some non-real-time tasks which run at low priorities, and which can block $\tau_i$ by invoking servers at high priority. Such a service time should be short enough in order not to disrupt soft real-time tasks. Hence, it is reasonable to assume that the worst case execution time of the servers can be accurately estimated. We do not need the execution time of non-real-time tasks running at a low priority.

We modify a number of results reported in [Harbour et al. 91] because of our assumptions. They are respectively the canonical form, various types of slowing effects, and the busy period.

The Canonical Form

In our new definition of the canonical form, consecutive execution segments do not decrease in priority. However, when two execution segments $s_{ij-1}$ and $s_{ij+1}$ are separated by an I/O segment $s_{ij}$, $s_{ij+1}$ can have a lower priority than the execution segment $s_{ij-1}$. The proof that consecutive execution segments complete at the same time as their canonical form is the same as Harbour et al.'s. An example of the canonical form is shown in Figure 4.6.
4.4. SCHEDULABILITY ANALYSIS

priority pattern of task X in a cycle

<table>
<thead>
<tr>
<th>priority</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

canonical form of task X

<table>
<thead>
<tr>
<th>priority</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.6: Transformation into the Canonical Form

Slowing Effects

In order to categorize the slowing effects, we define an O-segment as a waiting phase in addition to Harbour et al.'s definitions of H-segments and L-segments. Two types of new slowing effects due to O-segments can be distinguished:

1. If $\tau_j$ starts with an L-segment and contains inner consecutive segments $OH$ relative to $\tau_i$, the L-segment can possibly execute when $\tau_i$ waits, and the H-segment can preempt $\tau_i$ when $\tau_i$ resumes. Task $\tau_j$ produces a singly preemptive effect on $\tau_i$ because the last segment of $\tau_j$ must also be an L-segment relative to $\tau_i$. If $\tau_j$ contains inner consecutive segments $(OH)^+$ relative to $\tau_i$, the worst case is that all the H-segments in these consecutive segments contribute to a slowing effect on $\tau_i$.

If $\tau_j$ does not contain any consecutive segment $OH$, but rather, it starts with an L-segment and contains inner consecutive segments $HO$, then $\tau_j$ can only be of type $((LHO)^+L)$. This type is similar to $((LIH)^+L)$ and contributes to a blocking effect. One way to consider the O-segment is that if it is immediately before an H-segment of $\tau_j$, then $\tau_j$ is preemptive. Otherwise, if it is immediately before an L-segment of $\tau_j$, then it does not directly contribute to any slowing effect.
2. Tasks of type $H(OH)^+$ can produce a worse multiply preemptive effect than tasks of type $(H)$. Figure 4.7 gives an example where task $A$ is of type $HOH$ relative to task $B$. The worst case is that two instances of task $A$ execute contiguously, a situation not considered by Harter's analysis. In Harter's analysis, when a high priority task has complex interleavings of execution and waiting phases in a cycle, the total execution time is used for the computation of dilation. However, the computation of dilation is based on regular phasings as shown in Figure 4.3 but not Figure 4.7.

If task $A$ does not wait before its computation in a cycle completes, task $B$ only needs to be slowed down by time $t$.

Figure 4.7: A Multiply Preemptive Task that Waits

If $\tau_i$ contains multiple execution and waiting phases in a cycle, each execution phase is subject to the blocking and multiply preemptive effects defined in [Harbour et al. 91], as well as a new singly preemptive effect and a new multiply preemptive effect. We do not consider the singly preemptive tasks in Harbour et al.'s analysis because they are of type $(HL)^+$ and $((HL)^+H)$ which do not conform to
our assumptions of priority. We derive the longest completion time of \( \tau_i \) by first computing the worst case execution time for each execution phase of \( \tau_i \) and then adding these execution times to the worst case waiting time of \( \tau_i \).

**Busy Period**

We do not use \( \tau_i \)-busy period in Harbour et al.'s analysis for the following reasons:

1. We assume that the deadline of a cyclic task cannot exceed its cycle.

2. The concept of \( \tau_i \)-busy period is necessary in Harbour et al.'s analysis because when priority can be varied in any way, there are situations where an instance of \( \tau_i \) can change the phasings of some blocking or preemptive tasks and subsequently affect the completion time of a later instance of \( \tau_i \) (see Figure 4.5). In our analysis, since task \( \tau_i \) should always start from and end at its lowest priority, this phasing problem cannot happen. The tasks that can be preempted or blocked by the last segment of \( \tau_i \) can also be preempted or blocked by the first segment of \( \tau_i \).

**A Schedulability Check**

Our method provides worst case bounds to the completion time of cyclic tasks, as well as a worst case bound to the completion of any execution phase. The schedulability test for a task \( \tau \) is as follows. First of all, \( \tau \) is transformed into the canonical form. Let it be denoted by \( C_1O_1C_2O_2\cdots C_n \) where \( C_i \) is an execution or computation phase and \( O_i \) is a waiting phase. Let \( E_i \) be the longest time it takes for the execution phase \( C_i \) to complete in the presence of other tasks. Let \( IO_i \) be the longest waiting time associated with the waiting phase \( O_i \). The worst case completion time of \( \tau_i \) is:

\[
\sum_{i=1}^{n-1} (E_i + IO_i) + E_n
\]
Let the execution phase $C_i$ be composed of subtasks $\tau_{i1}, \ldots, \tau_{i(l(i))}$. Let $E_{ij}$ denote the longest completion time of subtask $\tau_{ij}$. A stepwise procedure is applied to each execution phase $C_i$:

1. Determine the following:

   $B_i$ the longest blocking time
   $SP_{ij}$ the set of singly preemptive tasks relative to $\tau_{ij}$
   $MP_{ij}$ the set of type (H) tasks relative to $\tau_{ij}$
   $MP'_{ij}$ the set of type (H(OH)$^+$) tasks relative to $\tau_{ij}$

   Tasks of type ((LHO$^0$)$^+L$) are considered as blocking tasks. Since only one of them can be chosen for blocking, the one with the longest inner H-segment is chosen. Tasks that contain consecutive segments $L(OH)^+L$ are singly preemptive, and all H-segments in $(OH)^+$ can contribute to the preemptive effect. If a task contains more than one occurrences of consecutive segments $L(OH)^+L$, the total length of H-segments in each occurrence is computed, and the longest is used as the worst case preemptive effect. Since subtask $\tau_{i1}$ executes at the lowest priority of $C_i$, any task which is singly preemptive relative to $\tau_{ij}$ where $1 < j < l(i)$ must also be singly preemptive relative to $\tau_{i1}$. Hence, $SP_{i1}$ contains all the singly preemptive tasks relative to any subtask $\tau_{ij}$, $1 < j < l(i)$. For task $\tau_s$ in $SP_{ij}$, let $e^h_s$ denote its worst case preemptive effect.

   Both $MP_{ij}$ and $MP'_{ij}$ contain multiply preemptive tasks. $MP'_{ij}$ contributes to the slowing effect of $C_i$ differently from $MP_{ij}$ because of the situation shown in Figure 4.7. In the worst case, at least two instances of each task in $MP'_{ij}$ can preempt $C_i$.

2. Determine $E_{i1}$.

   The longest completion time of $\tau_{i1}$ is $E_{i1}$, the minimum over all $t > 0$ such that

   $$B_1 + \sum_{\tau_k \in MP_{i1}} \left[ \frac{t}{P_k} \right] e_k + \sum_{\tau_k \in MP'_{i1}} \left( e_k + \left[ \frac{t - \sum_{\tau_j \in MP'_{i1}} e_j}{P_k} \right] e_k \right) + \sum_{\tau_s \in SP_{i1}} e^h_s + e_{i1} = t$$
4.4. SCHEDULABILITY ANALYSIS

The above formula for $E_{11}$ is similar to Harbour et al.'s except for the term which considers $MP'_{i1}$. The latest completion time of $\tau_{12}$ is $E_{12}$, the minimum over all $t > 0$ such that

$$E_{11} + \sum_{\tau_k \in MP'_{12}} \left( \left\lceil \frac{t}{P_k} \right\rceil - \left\lceil \frac{E_{11}}{P_k} \right\rceil \right) e_k + \sum_{\tau_k \in MP'_{12}} \left( \frac{t - \sum_{\tau_j \in MP'_{12}} e_j}{P_k} - \left\lceil \frac{E_{11} - \sum_{\tau_j \in MP'_{12}} e_j}{P_k} \right\rceil \right) e_k + e_{12} = t$$

In [Harbour et al. 91], a multiply preemptive task relative to the first subtask of $C_1$ can be singly preemptive relative to the second subtask of $C_1$. However, in our environment, the set $SP_{i1}$ contains all singly preemptive tasks relative to any subtask $\tau_j$ where $1 \leq j \leq l(1)$. Hence, our computation of the longest completion time of $\tau_{12}$ does not need to consider $SP_{12}$.

By applying the same algorithm to other subtasks, the latest completion time of the subtask $\tau_{i(1)}$ can be determined. This is also $E_1$, the longest completion of the execution phase $C_1$.

3. Determine the latest completion time of $C_i$, $\forall i \ 1 < i \leq n$.

The latest completion time of $C_i$ = the latest completion of $C_{i-1} + IO_{i-1} + E_i$.

If $C_{i-1}$ ends at the same priority as the starting priority of $C_i$, then $E_i$ can be calculated in the same way as $E_1$. Otherwise, if $C_{i-1}$ ends at a higher priority than the starting priority of $C_i$, the tasks with a priority in between can run during the waiting time $IO_{i-1}$ and produce a similar busy period to that shown in Figure 4.5. Let us consider the first two instances of task $B$ as one single task with two execution phases. One way to compute the slowing effect on the second execution phase of task $B$ is to use the busy period in a similar way to Harbour et al.'s method. Another simpler approximation is to assume the worst case where two instances of task $A$ preempt the second execution phase of task $B$.

Let $MP''_i = \{ \tau \in MP_{i1} \mid \text{the priority of } \tau \text{ is less than the priority of } C_{i-1} \text{ and greater than the } \}$. 
4.4. SCHEDULABILITY ANALYSIS

priority of \( C_i \). The approximation \( E_{i1} \) is the minimum over all \( t > 0 \) such that

\[
B_i + \sum_{\tau_k \in MP_i - MP''_i} \left\lfloor \frac{t}{P_k} \right\rfloor e_k + \sum_{\tau_k \in MP''_i \cup SP_i} \left( e_k + \left\lfloor \frac{t - \sum_{\tau_j \in MP''_i} e_j}{P_k} \right\rfloor e_k \right) + \sum_{\tau_k \in SP_i} e_{h_k} + e_{i1} = t
\]

The above formula is similar to that for \( E_{11} \) except for the consideration of \( MP''_i \). \( MP''_i \) can only affect the first subtask of \( C_i \), since it has the lowest priority among all subtasks of \( C_i \). The computation for \( E_{ij} \) where \( j > 1 \) is the same as that for \( E_{1j} \). The sum of \( E_{ij} \)'s gives \( E_i \); and hence, the longest completion time for \( C_i \) can be derived.

4.4.3 Review of the Techniques Using the EDF Rule

The complexity of schedulability tests using the preemptive EDF rule varies a lot depending on the assumptions of the task set to be analyzed. Since the preemptive EDF rule is optimal for a uniprocessor, the schedulability test for the preemptive EDF rule also serves as a feasibility test in a uniprocessor. We can decide whether a task set is feasible on a uniprocessor by constructing a schedule using the preemptive EDF rule and by checking whether the resulting schedule is valid. The problem of deciding whether an arbitrary task set is feasible on \( n \) processors, however, is NP-hard for every \( n \geq 1 \).

Periodic Tasks

Let a task set to be scheduled consist of periodic tasks \( \tau_1, \tau_2, \ldots, \tau_n \). The schedulability test is simple if their deadlines are the same as their periods [Liu & Layland 73, Coffman 76]:

**Theorem 4.2** If \( d_i = P_i \) for all \( 1 \leq i \leq n \), then \( \sum_{i=1}^{n} e_i / P_i \leq m \) is both a necessary and sufficient condition for the task set to be feasible on \( m \) processors.
The technique of constructing a schedule using the preemptive EDF rule can be applied for other cases. It is necessary to determine first the time bound within which the schedule should be constructed. If \( s_i = s_j \) for all \( 1 \leq i, j \leq n \), then the time bound would be the least common multiple of \( P = P_1, P_2, \ldots, P_n \). If \( s_i \neq s_j \) for some \( 1 \leq i, j \leq n \), a time bound also exists and is given by \( s + 2P \), where \( s = \max\{s_1, s_2, \ldots, s_n\} \). Let \( C(t) \) be the \( n \)-tuple \((e_{1,t}, \ldots, e_{n,t})\), where \( e_{i,t} \) is the amount of time for which task \( \tau_i \) has executed at time \( t \) since its last request. [Leung & Merrill 80] prove the following:

**Theorem 4.3** The task set \( \{\tau_1, \ldots, \tau_n\} \) is feasible on a uniprocessor if and only if (1) all deadlines in the interval \([0, s + 2P]\) are met, and (2) \( C(s + P) = C(s + 2P) \).

[Leung & Merrill 80] show that the problem of deciding whether an arbitrary task set is feasible on \( m \) processors is NP-hard for \( m \geq 1 \). [Lawler & Martel 81] consider the same problem on a multiprocessor and proves the following:

**Theorem 4.4** An arbitrary task set is schedulable with respect to the preemptive EDF rule on \( m \geq 1 \) processors if and only if there exists a valid schedule which is cyclic with a period \( P \) (i.e., each processor does exactly the same thing at time \( t \) as it does at time \( t + P \)).

**Schedulability Analysis of Hard Real-Time Programs**

When the preemptive EDF rule is used in real systems, schedulability analysis is not confined to the scheduling rule. Other types of overhead unrelated to the preemptive EDF rule must also be considered. [Stoyenko et al. 91] describe a schedulability analyzer designed for hard real-time programs. In their model, each task is associated with a frame or a minimal period. A task can be activated by a signal or at a time specified at compile time. Once activated, a task must complete before the end of the current frame. Tasks can run on different processors and communicate with each other asynchronously. At every clock tick, ready tasks are scheduled by the EDF rule. The response time of each task \( \tau_i \) is derived as follows:
4.4. SCHEDULABILITY ANALYSIS

\[ GT_i = I_i + N_i + QD_i + ID_i + ND_i + CD_i + SD_i \]

where \( I_i \) is the total CPU time for executing interruptible sections of \( \tau_i \), \( N_i \) is the total CPU time for executing non-interruptible sections of \( \tau_i \), \( QD_i \) is the worst case amount of time \( \tau_i \) spends in the queues of shared critical regions, \( ID_i \) is the worst case amount of time \( \tau_i \) spends waiting due to other processes executing their interruptible parts, \( ND_i \) is the worst case amount of time \( P_i \) spends waiting due to other processes executing their interruptible parts, \( CD_i \) is the worst case amount of time \( P_i \) spends waiting due to inter-processor communication delays, \( SD_i \) is the worst case amount of time \( P_i \) spends delayed in wait and device-accessing statements.

\( I_i \) and \( N_i \) are derived by adding up the interruptible and noninterruptible segments of task \( \tau_i \) respectively. In order to derive good bounds on various types of task delays, a technique called frame superimposition is used. Frame superimposition is a form of run-time simulation. A task \( \tau_i \)'s frame is assumed to start at time \( t_0 \). The frames of other tasks are then positioned in various ways relative to \( t_0 \). In this process, frames are shifted exhaustively for every time unit and every task. Some combinations of frame positions are impossible because of the EDF rule. For every possible combination of frame positions, the various types of contention are recorded. Maximum values of contention are derived at the end of the frame superimposition.

Stoyenko et al.'s algorithm searches for a lower upper bound which is not as pessimistic as those derived by simplier methods. This exhaustive approach is important to hard real-time systems where the worst case performance must be accurately predicted.

4.4.4 Review of the Techniques Using the Hierarchical Rule

A restricted combination of fixed priority and EDF scheduling is briefly discussed in [Liu & Layland 73]. An analysis for such a combination has recently been reported in [Jeffay & Stone 93] in the context of hard real-time systems. In this combination, the preemptive priority rule is used to schedule
interrupt handlers, and the preemptive EDF rule is used to schedule application tasks. The application tasks are effectively running at a lower priority level than any interrupt handler. The schedulability analyzer described in [Stoyenko et al. 91] does not particularly address this combination, although noninterruptible sections can be considered as running at a higher priority than interruptible sections.

In soft real-time systems, the strategy of confining all application tasks to the lowest priority and scheduling them only by the preemptive EDF rule is overly restrictive. We describe a procedure to determine the response times when some tasks in the server/client model are scheduled by the preemptive EDF rule while most tasks including the non-real-time tasks are scheduled by the preemptive priority rule.

**Interrupt Costs**

[Jeffay & Stone 93] describe a schedulability test for a system consisting of periodic real-time tasks and periodic interrupt handlers. In their model, time is discrete, and interrupts and task invocations occur at clock ticks. Interrupt handlers and real-time tasks are independent, and the former execute at priority strictly greater than the latter. A test is provided to determine whether each instance of each real-time task completes execution at or before its deadline in the presence of the interrupt handlers. It is assumed that their deadlines are the same as their periods.

Let \( \tau_i \) denote either a real-time task or an interrupt handler. Let \( R \) be the set of indices of real-time tasks and \( I \) be the set of indices of interrupt handlers.

Let \( U \) be the processor utilization. It is given by:

\[
U = \sum_{i \in R} \frac{\epsilon_i}{P_i} + \sum_{i \in I} \frac{\epsilon_i}{P_i}
\]

Let \( f \) be a function from non-negative integers to non-negative integers defined by the following recurrence relation:
4.4. SCHEDULABILITY ANALYSIS

\[ f(0) = 0, \]

\[ f(l) = \begin{cases} 
  f(l-1) & \text{if } f(l-1) = \sum_{i \in I} \left\lfloor \frac{l}{P_i} \right\rfloor e_i \\
  f(l-1) + 1 & \text{if } f(l-1) < \sum_{i \in I} \left\lfloor \frac{l}{P_i} \right\rfloor e_i
\end{cases} \]

Note that \( \forall l \geq 0, f(l) \leq \min(l, \sum_{i \in I} \lfloor l/P_i \rfloor e_i) \).

[Jeffay & Stone 93] prove that the function \( f(l) \) is the least upper bound on the amount of processor time spent executing interrupt handlers in the interval \([t, t + l]\) for all \( t \geq 0 \) and \( l \geq 0 \). Let

\[ B = \frac{\sum_{i \in R} e_i}{1 - U} \]

and let \( P = \{ kP_i \mid (kP_i < B) \land (k \geq 0) \land (1 \leq i \leq n) \} \) be the set of non-negative multiples (which are less than \( B \)) of the periods of the real-time tasks. [Jeffay & Stone 93] prove the following:

**Theorem 4.5** All real-time tasks will meet their deadlines if and only if for all \( L \in P \),

\[ L - f(L) \geq \sum_{i \in R} \left\lfloor \frac{L}{P_i} \right\rfloor e_i \]

Based on processor utilization, the above test is not concerned with the response times of the interrupt handlers and real-time tasks. Since the interrupt handlers are independent of the real-time tasks, their response times can be derived using the method described in [Harbour et al. 91].

4.4.5 An Extended Analysis Using the Hierarchical Rule

Section 4.4.1 analyzes the server/client model which uses the preemptive fixed priority rule. This section gives an analysis of the model in the uniprocessor when the preemptive EDF rule is used to schedule the tasks at the same priority level. Since the preemptive EDF rule is an optimal CPU scheduling algorithm, it is advantageous to apply this rule to the tasks which are CPU-bound. For example, a server task running at a high priority can manage a number of worker tasks which run at a lower priority and
which are also scheduled according to the deadlines of their work. In the following, we extend our schedulability analysis in Section 4.4.1 to consider the preemptive EDF rule. The assumptions for the tasks with deadlines are:

1. They do not vary their priorities.
2. They are periodic with constant periods.
3. They do not wait before their completion.
4. The first instances of the tasks at the same priority level start at the same time.
5. Their deadlines do not exceed their periods.

It is reasonable to make stricter assumptions for tasks with deadlines than tasks without deadlines since the preemptive EDF rule is used primarily for CPU optimization. If there is only one fixed priority level for the entire system, we can construct a schedule using the preemptive EDF rule and then check the feasibility of the schedule. The time bound within which the schedule should be constructed is the least common multiple $P$ of the periods of the tasks.

If the system consists of tasks of different priorities, a complex situation occurs where a task $\tau$ can be preempted by the tasks of a higher priority as well as the tasks at the same priority level as $\tau$ but with an earlier deadline. We construct a schedule for $\tau$ by assuming the worst dilation from higher priority tasks. Let $P$ be the least common multiple of the periods of the tasks at the same priority level as $\tau$. The arrival times (or the earliest starting times) of all task instances at this priority level repeat periodically with the period $P$. Because of this, it is sufficient to find out whether all instances of $\tau$ complete before their deadlines in the period $P$. Let $\tau_i$ be the $i^{th}$ instance of $\tau$ in the period $P$. This task instance can be blocked or preempted by other task instances of an earlier deadline and of the same priority. Let $D$ be a set containing these task instances as well as the task instances $\tau_1, \tau_2, \cdots, \tau_i$. Let the starting times
of the task instances in $D$ be $s_1, s_2, \ldots, s_k$ in strictly increasing order. Our procedure to find the latest completion time of $\tau_i$ is:

1. $j = 1, D_0 = \emptyset, \text{starttime} = s_1$

2. $D_j = D_{j-1} \cup \{ \tau': \tau' \in D, \tau' \text{ arrives at } s_j \}$

3. Find the total execution time of the task instances in $D_j$. Use schedulability check for the preemptive priority rule to compute the dilated execution time $E_j$ due to tasks of a higher or equal priority to $\tau$. The longest completion time $C_j$ of the task instances in $D_j$ is $\text{starttime} + E_j$.

4. If $j = i$, then $C_j$ is also the longest completion time of $\tau$. Otherwise, continue with the next step.

5. If the worst case completion time $C_j$ is greater than or equal to $s_{j+1}$, increment $j$ and go back to step (2).

6. If the worst case completion time $C_j$ is less than $s_{j+1}$, the tasks in $D_j$ cannot affect the tasks starting at $s_{j+1}$. Set $D_j$ to empty. Set $\text{starttime}$ to $s_{j+1}$. Increment $j$ and go back to step (2).
Chapter 5

Kernel Implementation

We have implemented our design on two different development platforms: the Mach 3.0 kernel [Black et al. 92] and the r-kernel [Ritchie & Neufeld 93]. The Mach 3.0 kernel provides rich functionality for the implementation of a time-sharing operating system. The r-kernel has been designed for an ATM interface in a research project. It supports fast user-level thread management on multiprocessors. Both the Mach 3.0 kernel and the r-kernel provide some form of priority scheduling, but their notions of priority scheduling are not up to the standards expected by real-time applications. Section 5.1 discusses these problems and our solutions. The implementation of the alarm capability mechanism requires a mutual exclusion mechanism. Section 5.2 discusses the various design tradeoffs we have considered and our two different implementations of the alarm capability mechanism on top of the two kernels. An important motivation of our design has been that the application programmer who needs to implement an application-specific scheduling policy should not need to change the kernel scheduler. Our experience with kernel debugging has strengthened our belief that a flexible scheduling interface is essential. The effort required to modify the kernel scheduler was excessive. It required too much knowledge of the internal particulars of the kernel; much more than an application programmer should need to know. Our observations relative to this problem is described in Section 5.3. Section 5.4 presents our performance measurements.
5.1 The CPU Scheduler

The preemptive fixed priority based scheduling algorithm is well-known in the real-time community for its simplicity and predictable performance. However, our experience has been that some pitfalls in the kernel implementation can easily demolish the real-time behavior of the resulting scheduler. This is especially the case when the priority scheduler must be added to a kernel which already implements a non-real-time scheduler. A major problem is that the entry points to the kernel scheduler may not have been designed to enforce a consistent implementation of the scheduling algorithm. Attention must be paid to every kernel interface in order to achieve the predictable performance expected by a real-time scheduler.

5.1.1 The FCFS Rule at the Same Priority Level

In both kernels, there is a queue of ready threads at each priority level. A ready thread is inserted at the tail of the ready queue. When a running thread is to be selected from a ready queue, the thread at the head of the queue is chosen. At first glance, this strategy supports the FCFS rule at the same priority level. However, because the thread selected to run must be removed from the ready queue, the FCFS rule can easily be violated. The violation occurs when the running thread is preempted and re-inserted into the ready queue. Although the thread should retain its position at the head of the ready queue, it is inserted at the tail instead. An example is illustrated in Figure 5.1.

In the multiprocessor environment, the strategy of removing a running thread from its ready queue simplifies the design where different processors can access the ready queues simultaneously. Although this is not needed in the uniprocessor environment, the same algorithm has been used in Mach 3.0 for uniprocessors. Our solution for the uniprocessor environment is to provide two interfaces to insert a thread into its ready queue. Since we support the EDF rule, the EDF rule is always applied first. Two queuing operations are provided so that a thread can be inserted before or after all other threads of the
(i) Thread A runs at the highest priority.
    Ready queue of the next highest priority:

    head [B C D] tail

(ii) Thread A blocks.
     Thread B runs.
     Thread E arrives.

    [C D E]

(iii) Thread A unblocks and preempts thread B.
     Thread B is re-inserted to the ready queue.

    [C D E B]

    The FCFS rule is violated - see (i).

Figure 5.1: Insertions and Deletions to the Ready Queue
same deadline. Our solution for the multiprocessor environment is slightly different, since multiple threads can be inserted into and removed from the ready queue simultaneously. In order to maintain the FCFS rule, a timestamp is assigned to a thread when it is first inserted into the ready queue. Later, when this thread is selected to run, preempted, and re-inserted into the ready queue, this timestamp is used to determine the FCFS order among the threads with the same scheduling attributes. This solution also applies to the uniprocessor.

5.1.2 The Idle Thread

In the multiprocessor environment, the Mach 3.0 scheduler always considers idle processors first when a CPU is needed to run a thread. In the kernel, there is an idle thread which runs at the lowest priority and which looks for threads to execute. This idle thread approach is used for both the multiprocessor and uniprocessor environments.

The idle thread gives rise to another circumstance in which the priority scheduling rule is violated. An example (assuming a uniprocessor environment) is shown in Figure 5.2. In this example, thread A and B run periodically. The idle thread has the opportunity to run when both A and B sleep. However, when thread A wakes up, it is executed at the priority level of the idle thread. Although thread A should have a higher priority than thread B, thread A is preempted when thread B wakes up. There are two ways to resolve this problem: we could change the scheduling attributes of the idle thread, or we could prevent the idle thread from running other threads. We have chosen the second approach because it is simpler.

5.1.3 Stack Handoff

Stack handoff is an important optimization technique for the handling of kernel stacks in Mach 3.0 [Draves et al. 91]. It optimizes both space and time for the Mach RPC mechanism. Since RPC is the most frequently used operation, stack handoff provides the most substantial performance gain among
Priority of thread A > Priority of thread B > Priority of the idle thread

Figure 5.2: The Idle Thread
5.1. THE CPU SCHEDULER

various techniques. The following two RPC cases have been considered. First, a client thread sends a message to a server thread and waits for a reply. Second, the server thread sends a reply message to the client thread and waits for the next request message. In both cases, the sending thread wakes up the receiving thread and blocks itself until a message arrives.

The management of kernel stacks in the multiprocessor environment is subject to various considerations [Draves et al. 91]. The details of the stack handoff mechanism are beyond the scope of this thesis. In brief, a thread enters the kernel by sending a message to another thread. Inside the kernel, the sending thread checks whether there is a thread waiting to receive the message. If there is, the sending thread transfers its kernel stack to the receiving thread directly. Still in the kernel, the receiving thread is made to run in the context of the sender's system call. There are savings in both space and time when the receiving thread can inherit the the context of the sending thread in this way. After finishing with the system call, the receiving thread exits the kernel.

It may not be immediately obvious how the scheduling rules are violated in the handoff mechanism, since the receiving thread is usually a server which has a higher priority than its clients or the sending threads. The problem occurs when the server thread needs to send a reply message to its client, and there is a stack handoff from the server to the client.

An example is illustrated in Figure 5.3. A client thread \( C \) sends a message to a server thread \( A \). Before thread \( A \) replies to thread \( C \), thread \( B \) arrives into the system. Since thread \( B \) has a priority between thread \( A \) and thread \( C \), thread \( B \) should run when thread \( A \) blocks. When thread \( A \) sends a reply message to thread \( C \), however, there is a handoff to thread \( C \) causing a violation of priority scheduling. Our solution to this problem is to force each potential stack handoff to be examined by the scheduler. A handoff is permitted only if it does not violate the scheduling rules. Although there are fewer situations where the handoff optimization can occur, the predictability of the scheduling mechanism is maintained.
5.1. THE CPU SCHEDULER

Priority of thread A > Priority of thread B > Priority of thread C

Figure 5.3: Stack Handoff
5.2. THE ALARM CAPABILITY

5.1.4 Priority Changes

In our implementation of \texttt{set.thread.sched.attr()}, any rescheduling required by changes to scheduling attributes is performed immediately. Mach 3.0 (versions before MK84) used to support a function \texttt{thread.priority()} which sets the priority of a thread. However, its algorithm for rescheduling has been optimized for the time-sharing policy. In its algorithm, when the caller lowers its own priority, the scheduler does not check immediately whether another thread of a higher priority should become active. The check is delayed until the expiration of the current scheduling quantum. A similar problem happens in the r-kernel. Although the r-kernel does not support priority changes, checks for rescheduling are performed only when a quantum expires.

Our implementation of \texttt{set.thread.sched.attr()} works as follows. Since there is a ready queue for each priority level, a ready thread is moved from one ready queue to another when its priority changes. When this thread is inserted into the ready queue of the new priority, it is inserted before all threads of the same deadline. In the uniprocessor environment, a thread can raise its priority before entering a critical section. Once finished with the critical section, it resumes its original priority and can continue to run without being preempted by newly queued ready threads of the same priority.

5.2 The Alarm Capability

In the alarm capability mechanism, the function \texttt{reset_alarm()} is called by the application to stop an alarm function from executing but only if the execution has not yet started. In Mach 3.0, a reasonable design is to assign a lock for each alarm capability. This lock must be acquired by the alarm thread before the alarm function is called. Similarly, the function \texttt{reset_alarm()} must acquire this lock before cancelling the call to the alarm function. It is not necessary to use a spin lock mechanism here, since the thread which cannot get the lock should give up trying. Our algorithm works as follows. The alarm thread tries to acquire the lock before it calls the alarm function. If it cannot get the lock, the alarm...
5.3. KERNEL DEBUGGING

The function reset_alarm() also tries to get the lock before it accesses the alarm capability. From the information contained in the alarm capability, resetAlarm() determines whether to suspend the alarm thread.

The implementation for Mach 3.0, however, is not applicable for the r-kernel. In the r-kernel, any thread that owns a lock is non-preemptable. This non-preemptability is not needed for the execution of the alarm function. Moreover, it overrides the scheduling rules. Therefore, in our implementation, we use a flag in the alarm capability to indicate whether the alarm function has started to execute. A lock is used to protect the integrity of this flag.

5.3 Kernel Debugging

It has been much more difficult to change the kernel scheduler than we anticipated. The difficulty was caused by two erroneous assumptions on our part:

1. We had expected that the scheduling algorithm was implemented in a module which provided narrow interfaces for other kernel components. Our experience with Mach 3.0 has been that these interfaces are not narrow and furthermore they are bypassed by other kernel components (like the stack handoff and idle thread optimizations).

2. Since thread scheduling is implemented at the user level in r-kernel, we had expected minimal debugging at the kernel level. Our experience turned out to be the opposite. A bug in the kernel upcall mechanism caused many strange scheduling errors and mysterious system crashes. Debugging was complicated by the problem that this bug only occurred in the multiprocessor environment and was not detected before our changes to the thread scheduler.

Generally, the difficulty of kernel debugging is proportional to the complexity of the kernel. This has been the case in our debugging experience of the Mach 3.0 kernel which is substantially more
complex than the r-kernel. In the Mach 3.0 kernel, there are some functions which clearly implement the priority rule and the operations on the ready queues. However, they constitute only a small part of the scheduler. A large part of the scheduler has been integrated into various mechanisms which block the running thread and select a ready thread to run for different reasons. There are numerous procedures which can affect kernel scheduling. Because of the volume and complexity of the code in Mach 3.0, a combination of testing and code studying has been proven useful to uncover the problems discussed in Section 3.1. Some problems were more difficult to detect than others. The stack handoff problem was the last one we had found.

A major difficulty in extensive testing of the scheduler is to determine whether threads are preempted properly. In the absence of a software analyzer, thread activities have been monitored by designing each thread to print its progress from time to time. In order to avoid impossibly excessive output, a busy wait loop is often inserted before an output. However, some erratic preemptions can occur briefly without being noticed when the threads involved are in busy wait loops. A simple sound application we built has been unexpectedly helpful for discovering these problems. This application generates sound sequences by playing notes at time intervals specified by the user. It provides a good test for a real-time operating system because human perception is sensitive to even slight variations in rhythmic patterns [Flinn 93]. In our experiments, a variation of 20 milliseconds in the timing of generated sound is very noticeable. Scheduler bugs can easily be detected by unexpected variations in the rhythmic patterns. A bug we discovered in this way is illustrated in Figure 5.4. In this example, the sound thread runs periodically, and thread \( L \) is running at the same priority as the sound thread. The sound thread checks the present time before playing any note. If it is too late, the sound thread proceeds on to the next note in the sound sequence. If it is too early to play the next note, then the sound thread sets its starting time to the time when this note should be played. It is thus possible that the sound thread may not play any note during a period. In Figure 5.4 (a), because of the FCFS rule, a long silent period occurs when
thread $L$ runs until it finishes. This silent period should be even longer if thread $L$ is ever preempted by a thread of a higher priority. However, when the FCFS rule is violated as shown in Figure 5.4 (b), there are two shorter silent periods instead. The result is a striking variation in the rhythm.

### 5.4 Performance Measurements

The measurements of scheduling overheads are important to the application programmer since the real-time application must allow for such overheads. For example, an on-line scheduler which takes 200 ms to find an optimal schedule may serve well for some applications but not those with urgent deadlines. However, there are presently no standard benchmark programs which measure the overall performance of real-time operating systems. Kernel schedulers are often designed for different timing objectives; hence, their comparisons are performed at the algorithmic level but not the implementation level.

Our kernel scheduler implements an efficient hierarchical scheduling algorithm. The overhead of manipulating the hierarchical queue is in the order of a few microseconds for 486. It is of the same magnitude as the manipulation of a priority queue. The more interesting overheads are thus those associated with system calls. The scheduling interface which allows for dynamic changes to scheduling attributes is the most important since it serves as the basis of complex scheduling policies and adds to the implementation overhead of these policies. The primitives for alarms contribute to the overhead of the handling of timing errors. The primitive to create real-time tasks should only be needed during initialization, and its performance is useful for estimating the initialization time. The incorporation of these overheads into a higher-level scheduling mechanism is straightforward since the primitives must be called by the higher-level software in order to incur any overhead.

We measured the performance of our primitives by running Mach 3.0 with our modifications on both Sun/3 and 486 processors, and running the modified r-kernel on Motorola 88k processors. Table 5.1 gives the performance measurements on a single processor. The performance of `set_thread_sched_attr()`
(a) Thread L runs at the same priority as the sound thread. A long silent period when thread L runs.

(b) A bug - the FCFS rule is violated after the preemption of thread L. A distorted sound sequence when thread L starts to run.
5.4. PERFORMANCE MEASUREMENTS

has been measured in two ways. In one case, a thread changes its priority repeatedly, but no context switches occurs because it always runs at the highest priority. In the second case, a thread $t_1$ repeatedly lowers its priority from $p_1$ to $p_2$, while a second thread repeatedly raises the priority of $t_1$ to $p_1$. Since the second thread runs at a priority between $p_1$ and $p_2$, there is a context switch every time \texttt{set_thread_sched_attr()} is called by either of these two threads.

The performance of our primitives is generally good except for the primitives \texttt{rthread_create()} and \texttt{alloc_alarm_pool()} in Mach 3.0. In both cases, most of the overhead is due to the operation of creating a Mach thread. This is specific to the Mach implementation and does not affect the performance of the same primitives in the r-kernel. The call \texttt{set_thread_sched_attr()} in the r-kernel outperforms that in the Mach 3.0 kernel \textsuperscript{1} because the former supports thread scheduling in the user space. Kernel calls in Mach 3.0 are relatively expensive. The minimum overhead of making a kernel call in Mach 3.0 is 51 microseconds on Sun 3/60. The calls \texttt{set_alarm()} and \texttt{reset_alarm()} in Mach 3.0 also involve the overhead of crossing the kernel boundary. The calls \texttt{get_alarm()} and \texttt{return_alarm()}, on the other hand, perform better in Mach 3.0 than the r-kernel because of the two different implementations of lock mechanisms described in Section 3.2.

Table 5.2 gives a more detailed description of the performance of \texttt{set_thread_sched_attr()} on the Motorola 88k. The performance is divided into the local and the remote cases. In the local case, when the running thread raises the priority of another thread, the running thread is preempted. The performance is the same as that for a single CPU. In the remote case, this running thread continues to run in the same CPU, but the remote thread whose priority is raised preempts the running thread in another CPU. The remote case is slower than the local case because a mechanism is required to inform the remote CPUs about the need to reschedule. The performance of \texttt{set_thread_sched_attr()} has been measured for

\textsuperscript{1}Note that the 88k is slower than the 486. The SPECint92 benchmark is derived from the results of a set of integer benchmarks, and can be used to estimate a machine's single-tasking throughput on integer code. A larger number means a faster machine. The SPECint92 benchmarks are 17.4 and 30.1 for 88100 (25 MHz, 16K data cache and 16K instruction cache) and Intel 486DX (50 Mhz, 256K off-chip and 8K on-chip cache) respectively.
different types of changes in the scheduling attributes. Besides raising and lowering priority, context switches can be triggered in other ways. A thread can set its starting time to a later time, and another thread changes this starting time back to the present time. Alternatively, a thread can suspend itself, and be resumed by another thread. As shown in Table 5.2, there is more overhead in the remote case for changing priority. This is because remote CPUs are informed about the need to reschedule whether priorities are raised or lowered. In the other two cases, when a running thread suspends itself or sets its starting time to a later time, the remote CPUs are not notified. This optimization can also be applied when the priority of the running thread is lowered. The overhead caused by the remote notification can be halved. In the local case, the differences in the performance of \texttt{set\_thread\_sched\_attr()} are merely due to the differences in queuing.

<table>
<thead>
<tr>
<th></th>
<th>Mach 3.0 3/60</th>
<th>Mach 3.0 486</th>
<th>r-kernel 88k</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{set_thread_sched_attr}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no CSW</td>
<td>114.5</td>
<td>16.1</td>
<td>16.1</td>
</tr>
<tr>
<td>CSW</td>
<td>381.8</td>
<td>47.7</td>
<td>67.7</td>
</tr>
<tr>
<td>\texttt{rthread_create}</td>
<td>4150.0</td>
<td>1133.3</td>
<td>77.0</td>
</tr>
<tr>
<td>\texttt{alloc_alarm_pool(1)}</td>
<td>3803.8</td>
<td>1034.1</td>
<td>87.0</td>
</tr>
<tr>
<td>\texttt{get_alarm &amp; return_alarm}</td>
<td>6.9</td>
<td>1.7</td>
<td>18.0</td>
</tr>
<tr>
<td>\texttt{set_alarm &amp; reset_alarm}</td>
<td>263.4</td>
<td>39.7</td>
<td>65.6</td>
</tr>
</tbody>
</table>

Table 5.1: Performance Measurements on one CPU (microseconds)
### 5.4. PERFORMANCE MEASUREMENTS

<table>
<thead>
<tr>
<th>Function Description</th>
<th>Local</th>
<th>Remote</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_thread_sched_attr * 2</code> &lt;br&gt;raise/lower priority</td>
<td>134.3</td>
<td>214.7</td>
</tr>
<tr>
<td><code>set_thread_sched_attr * 2</code> &lt;br&gt;set/reset starting time</td>
<td>127.3</td>
<td>165.2</td>
</tr>
<tr>
<td><code>set_thread_sched_attr * 2</code> &lt;br&gt;suspend/resume</td>
<td>104.3</td>
<td>137.6</td>
</tr>
</tbody>
</table>

Table 5.2: Performance of `set_thread_sched_attr()` in a multiprocessor 88k (microseconds)
Chapter 6

A Flexible Scheduling Abstraction

Our design is useful to both hard real-time and soft real-time systems. It provides substantial flexibility in the implementation of real-time scheduling policies and at the same time, encourages modularity in both kernel and application design. In Section 6.1, we demonstrate how our architecture supports competitive task models by describing how to implement the scheduling mechanisms of the Spring kernel and the Real-Time Mach kernel as cooperating tasks on top of our kernel. Then, we describe ways to use our core scheduler to implement other common hard real-time scheduling policies. In Section 6.2, we explain how our abstraction facilitates soft real-time applications whose timing requirements are more complex and diversified than hard real-time applications. In terms of general timing support, our core scheduler provides a framework to support the responsiveness required by systems whose timing constraints change dynamically. As far as the scheduling policies specific to the application are concerned, our scheduling interface is particularly advantageous. We illustrate this by considering the requirements of a modern multimedia application.

6.1 Hard Real-Time Systems

Real-time kernels designed to support competitive task models have taken the monolithic approach of integrating the scheduling policy into the task model. The lack of flexibility inherent in this monolithic
6.1. HARD REAL-TIME SYSTEMS

design was discussed in Section 3.1.5, and a structured approach was described in Section 3.3. The benefits of our architectural support can be considered at two levels. First, our design provides a layered scheduling mechanism where complex scheduling mechanisms can easily be implemented on top of our CPU scheduler. Competitive task models can thus be implemented by cooperating tasks. Second, our task abstraction can be used directly when competitive task models are too rigid to accommodate the evolving and dynamic timing requirements of applications. In the following, we illustrate the flexibility of our solution by considering the scheduling policy of the Spring kernel, that of the Real-Time Mach kernel, as well as other common hard real-time scheduling policies.

6.1.1 The Spring Kernel

As discussed in Section 3.1.4, the Spring kernel provides three different scheduling strategies for three types of tasks: critical tasks, essential tasks, and unessential tasks. These scheduling strategies can easily be implemented by using our core scheduling rules. In Spring, tasks are defined by their requirements for multiple resources. We can further exploit the modularity of the cooperating task structure by structuring each Spring task as one or more cooperating tasks.

The Spring Scheduling Policy

The Spring kernel uses a resource scheduling heuristic [Zhao et al. 87] for critical tasks and essential tasks. It is assumed that the arrival times and worst case resource requirements of critical tasks are known in advance. A feasible schedule is derived from this information and resources are preallocated to all critical tasks. The arrival times of essential tasks are not, in general, predictable. When an essential task arrives, it is accepted into the schedule only if doing so does not affect the other tasks which have already been accepted. Otherwise, the newly arrived task is either rejected or placed in a waiting list in case some resources are released earlier than expected. Non-essential tasks are executed when neither
6.1. HARD REAL-TIME SYSTEMS

Critical nor essential tasks require the processor.

The Spring scheduling mechanism maintains information about the worst case resource requirements of all tasks in the system as well as the earliest available times of all resources. Since the execution of the resource scheduling heuristic itself also consumes CPU time and may invalidate the assumptions that the scheduler is making about available resources, Spring is particularly appropriate for multiprocessor systems where a processor can be statically allocated to running the heuristics. Some slight modification in the resource heuristic is necessary for uniprocessors. Alternatively, a co-processor can be used [Niehaus et al. 93].

Our core scheduler provides a simpler and yet more modular solution for the implementation of the Spring scheduling heuristics in both the multiprocessor and uniprocessor environment. In order to conform to the definition of tasks in the Spring system, critical tasks and essential tasks can be implemented as tasks in our design, except that they are preallocated resources and that they do not cooperate with each other. The resource scheduling heuristics are implemented as a task which is assigned a lower priority than all critical tasks and all essential tasks which have been accepted into the schedule. This priority assignment ensures that the execution of critical tasks and essential tasks cannot be affected by the execution of the resource scheduling heuristics. The resource scheduling task can be blocked until a task arrives. When awakened by a newly arrived task, the resource scheduling task can check easily from its schedule whether there will be enough time to run the scheduling heuristics before it proceeds.

The design of the resource scheduling task can easily be extended to consider early release of resources. When resources are released, the resource management modules can raise an event or send a message to unblock the resource scheduling task which then determines whether the resources have been released earlier than expected, and if so, checks whether it is now possible to accept a task which has been waiting for resources.
It would be a system error if resources are released later than expected. While the Spring kernel does not support customized error handling, our design allows for more flexibility by using alarm tasks to handle timing errors (see Section 4.2.3). Each alarm task calls an error handling procedure which can be supplied by the application programmer, and various error handling procedures can be designed for different devices and different types of deadlines.

The design of using cooperating tasks to implement resource scheduling and timing error handling is illustrated in Figure 6.1.

### Spring Tasks

Spring tasks have been designed to have a sequential structure in order to support deterministic real-time performance [Stankovic & Ramamritham 87]. In the handling of concurrent external activities, however, being restricted to sequential programming results in solutions that are very difficult to maintain (see Section 3.2.1). Using our task abstraction, modularity can be greatly improved by structuring Spring tasks and resource management modules as cooperating tasks. An example of a sequential Spring task $X$ is shown in Figure 6.2. It is changed from a sequential structure to a set of cooperating tasks which communicate and synchronize with each other in Figure 6.3. These cooperating tasks are assigned the same scheduling attributes as task $X$ and deliver the same service as $X$. When the scheduling needs of the application change, the impact on the structure of the cooperating tasks is minor. The software utilities which depend on the sequential structure of task $X$ are not applicable. Some new overhead is also introduced because of context switching. The improvement in modularity, however, will provide substantial savings in software maintenance and reuse.
6.1. HARD REAL-TIME SYSTEMS

Spring Task Request X
For an interval of t time units
Earliest starting time $T_1$
Deadline $T_2$
Resources required:
- CPUs, code segments,
- memory, peripherals

On-line schedulability analysis.
Allocate resources for exclusive access
from $T_3$ to $T_3 + t$.
Assign a high priority and a starting time $T_3$ to task X.
Assign a higher priority and a starting time $T_3 + t$ to an alarm task to ensure that the resources will be deallocated by $T_3 + t$.

$T_3$:

Use procedural interfaces to access resources.
Use bounded wait for external events.

$T_3 + t$:

If task X has finished and all resources are released, the alarm task suspends itself.
Otherwise, the alarm task can abort task X and release all its resources.

Figure 6.1: Using Threads to implement the Spring Resource Scheduling Heuristics
6.1. HARD REAL-TIME SYSTEMS

Competitive (Sequential) Version of Spring Task X

Interleaved code:

Get data from peripheral C repeatedly
until timeout or until the right data is obtained.

Get data from peripheral D repeatedly
until timeout or until the right data is obtained.

If the right data are obtained from C and D,
write to peripheral E.

If the right data are not obtained on timeout,
write to peripheral E.

Figure 6.2: A Spring Task

Replacing the sequential version of task X by cooperating tasks P, Q, R, and S

```
task S

Set the starting time of task S to timeout.
Accept messages from tasks Q, R, and S.
Write to peripheral E if messages are received from tasks Q and R,
or when a message is received from task S.
```

Get and process data from peripheral C.

```
Get and process data from peripheral D.
```

Figure 6.3: A Cooperating Task Structure
6.1.2 The Real-Time Mach Kernel

The Real-Time Mach kernel provides a number of scheduling policies based on the rate-monotonic assignment rule and priority inheritance protocols [Tokuda et al. 90] (see Section 3.1.4 and Section 3.1.3). These priority inheritance protocols are incorporated into the real-time thread synchronization mechanism so that blocking threads are not subject to the priority inversion problem. Using our abstraction, the scheduling policies provided by the Real-Time Mach kernel are implemented by a set of cooperating threads. The notions of thread synchronization and scheduling are separated so that the two can be independently managed (see Section 3.3.1).

Scheduling based on the Rate-Monotonic Assignment Rule

The Real-Time Mach kernel uses the rate-monotonic rule to assign fixed priorities to periodic threads. Aperiodic threads are also assigned priorities by a number of policies based on the rate-monotonic rule. Since we provide the function `set_thread_sched_attr()` which allows the priority of a thread to be changed dynamically, these priority assignment policies can easily be implemented by a scheduling thread. This thread runs at a higher priority than other threads and maintains the timing requirements of all threads. Alternatively, the priority assignment policies can be implemented in user library calls which run at a higher priority than other threads.

Periodic Threads

The Real-Time Mach kernel provides an abstraction for periodic threads with constant periods. In our design, this abstraction is implemented by a loop where the starting time of the thread is reset at the end of the loop. The function `set_thread_sched_attr()` is used to change the `starting_time` attribute of the thread.
Priority Inheritance Protocols

The priority inversion problem occurs when threads of different priorities are ordered by the FCFS rule in their access to a shared resource, and a thread holding a resource is preemptable (see Section 3.1.3). [Rajkumar 91] describes a number of priority inheritance protocols designed to bound blocking times in accordance with thread priority. In the basic priority inheritance protocol, a thread which holds a resource inherits the highest priority of the threads which are waiting for this resource. Moreover, the waiting threads are queued in descending order of their priorities. This strategy guarantees that a thread can only be blocked by higher priority threads and at most one lower priority thread for each resource.

Besides the basic scheme described above, there are other priority inheritance protocols which consider more complicated situations involving the sharing of multiple resources. These protocols need more information about the sharing of resources. For example, in the priority ceiling protocol, the priority ceiling of a resource is defined as the highest priority of all of the threads which may access this resource.

In our abstraction, these priority inheritance protocols can be implemented by using the function `set_thread_sched_attr()`. They can either be implemented in a user library or in special threads assigned responsibility for resource management. In the former approach, two library calls are needed; one called before getting a resource, and the other called after releasing the resource. The data structures describing which thread holds this resource and which threads are waiting for it are maintained by these two library calls. In order to ensure the integrity of these data structures, the library calls must not be executed by more than one thread at the same time. This can be achieved in a uniprocessor by having the caller's priority raised during the call and restored afterwards. A purpose of the call made before getting the resource is to check whether there is a thread holding the resource. If there is such a thread and it has a lower priority than the caller, then its priority is raised. When the caller releases the resource, it restores its priority which might have been raised by other threads.
Priority inheritance protocols can also be implemented in threads called resource administrators. Process structuring using the administrator concept and message passing is discussed in [Gentleman 81] without considering priorities. An administrator obtains service requests from the client threads, and unblocks worker threads to perform the actual service. A resource administrator is an administrator thread, except that it also queues the client requests and changes the priorities of worker threads according to a priority inheritance algorithm (Figure 6.4). This ensures that none of its potential clients could be blocked by a lower priority thread. In an alternative design, the resource administrator may queue client requests and let a worker thread perform the priority inheritance mechanism. The alternative design makes the administrator more responsive to new client requests.

Figure 6.4: Priority Inheritance performed by a Resource Administrator

When priority inheritance is separated from the synchronization primitives, there is more freedom in the design of interfaces for various priority inheritance protocols, and the application may modify and optimize the inheritance scheme where required.
6.1.3 Other Scheduling Policies

Since our core scheduler supports the preemptive fixed priority rule and the preemptive EDF rule, it provides an infrastructure for the implementation of various scheduling policies based on these two on-line scheduling rules. Some examples of such scheduling policies are the rate-monotonic rule and a variety of priority inheritance protocols (see Section 6.1.2). Non-preemptive rules can be implemented by having every task raise its priority during its initialization.

Our core scheduler also supports the policy of making a task ready at an absolute time. This provides the mechanism for the implementation of static schedules. In some scheduling schemes, an off-line or pre-run-time scheduling algorithm is used to produce a static schedule for tasks of critical importance [Shepard & Gagné 91, Xu & Parnas 90]. The static schedule specifies the starting time of each task and the resources the task is allocated. In our approach, this can be implemented by using our primitive to set the starting times of the critical tasks. Some scheduling policies use a static schedule for critical tasks and a dynamic schedule for other tasks. This can be accomplished by assigning a high priority to critical tasks as discussed for the Spring resource scheduling heuristics (see Section 6.1.1). Figure 6.5 summarizes the different ways our core scheduler can be used in conjunction with schedulability analysis.

Periodic tasks can be implemented by resetting the starting time in a loop. Besides applications with periodic activities, schedulers which run periodically can also be implemented as periodic tasks.

6.1.4 Architectural Support

Our scheduling interface allows for the use of cooperating tasks to structure mechanisms for both resource scheduling and timing errors. A resource scheduling algorithm can be implemented in a resource administrator task which schedules a number of worker tasks, each of which manages a resource. The resource administrator task may change the scheduling attributes of any task dynamically...
6.1. HARD REAL-TIME SYSTEMS

Timing Objectives
Exact Task Arrival Times
Resource Requirements of Individual Tasks
Task Dependency Information

PRE-RUN-TIME SCHEDULER

static schedule

task $t_1$ starts at $T_1$
task $t_2$ starts at $T_2$

A single scheduling task, or
Multiple tasks participate in
the procedures of
determining scheduling attributes

call at a high priority

Dynamically change the starting time,
priority, and deadline attributes

(b) On-line Schedulability Check

Figure 6.5: Scheduling Policies and Schedulability Analysis
6.1. HARD REAL-TIME SYSTEMS

using our scheduling interface. The handling of timing errors can be implemented by alarm tasks.

Competitive task models can be implemented on top of our task model since the underlying scheduling mechanism of competitive tasks can be performed by a set of cooperating tasks. The failure of a competitive task to meet a deadline is a timing error that can be handled by an alarm task. This results in an architecture as shown in Figure 6.6.

![Figure 6.6: Implementation of Competitive Task Models](image)

The architecture shown in Figure 6.6 allows for the use of a common kernel for various competitive task models. This solves the problem of proliferating kernels and allows for the continual use of software utilities designed for individual competitive task models. When the competitive task models are too rigid for a given application, the application can be structured as cooperating tasks instead. A competitive task can be implemented as one or more cooperating tasks which communicate with the resource administrator task and the worker tasks. A common set of task synchronization and communication primitives is used instead of the primitives specific to individual competitive task models, such as
6.2 SOFT REAL-TIME SYSTEMS

various real-time synchronization primitives. When competitive tasks are implemented as cooperating tasks, the architecture is as shown in Figure 6.7.

<table>
<thead>
<tr>
<th>Application</th>
<th>Cooperating Task Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface to Scheduler A</td>
<td>Interface to Scheduler B</td>
</tr>
<tr>
<td>Scheduler A</td>
<td>Scheduler B</td>
</tr>
</tbody>
</table>

Kernel - Task Abstraction, CPU Scheduling

Figure 6.7: Cooperating Tasks at the Application Level

6.2 Soft Real-Time Systems

The real-time kernels designed for soft real-time systems often support the cooperating task model (see Section 3.2.4). However, they only provide a task synchronization and communication mechanism for the implementation of resource schedulers. This is not adequate when the resource schedulers must schedule CPU as well as other resources (see Section 3.2.4 and Section 3.3.1). Since our design also provides a scheduling interface, it supports a more flexible cooperating task structure. Resource scheduling can be implemented as cooperating tasks like other components of resource management. The server/client paradigm continues to be useful.

Besides a flexible cooperating task structure, our design is concerned with both the general and application-specific timing requirements of soft real-time systems. Responsiveness is a general but necessary objective of soft real-time systems. In Section 6.2.1, we define responsiveness in the context
6.2. **SOFT REAL-TIME SYSTEMS**

of satisfying timing constraints and describe our scheduling support for this objective. Our scheduling interface is particularly useful to soft real-time systems which are much more "application-centric" than hard real-time systems. In Section 6.2.2, we describe a multimedia application which benefits from our scheduling interface.

### 6.2.1 Responsiveness and Dynamic Timing Constraints

During on-line scheduling, there is always a tradeoff between providing responsiveness to task requests and searching for an optimal schedule. A simple on-line scheduler has lower overhead than a complex on-line scheduler and is therefore more responsive. However, a complex on-line scheduler which finds a better schedule will allow more deadlines to be satisfied. In adaptive real-time systems [Muntz & Lichota 91] where deadlines may be dynamically altered according to external events, the tradeoff between responsiveness and optimal scheduling is especially complex. In order to implement application-level policies, it is necessary to support an abstraction of this event analysis process. We define such an abstraction and describe how our hierarchical scheduler provides a framework for the implementation of application-level policies which may prefer responsiveness over optimality.

The analysis of an external event can be very complicated. For example, a network interrupt may trigger one of a variety of different handling mechanisms. It can signal a serious network problem which should be resolved by resetting the network interface. It may be caused by the receipt of a packet which carries data urgently needed by a real-time database. On the other hand, it may be simply a transmit interrupt caused by the completion of the transmission of a packet. The analysis of external events can also be computationally expensive as, for example, in the case of robotics applications which detect and compute the trajectories of moving objects in real time. When the event analysis process is complicated and computationally expensive, scheduling the CPU is difficult as one should consider its influence on subsequent resource scheduling decisions.
We propose viewing the event analysis process as a multi-stage decision making process where each stage may consist of more than one independent computation unit. The decisions made by one stage may alter the timing constraints in the next, and so on. The relation among stages is different from precedence ordering. Different stages may operate at the same time dealing with different data. However, when a stage detects the need to change the timing constraints in the next stage, some processing in the next stage may be halted or altered accordingly. For example, consider device handling in the first stage. Each device handler may be subject to a timing constraint to copy volatile data. Some events, such as a critical device failure, may be discovered in this stage which indicate that some of the data read could be faulty and should be abandoned if not yet processed. The second stage consists of the selection and analysis of various device data. Similarly, various computation units may be involved in this stage. The stages are thus similar to priority levels, except that a lower stage corresponds to a higher priority level. Generally, a complex event analysis process is decomposed in a way that computation units at one level may determine the timing attributes and the significance of computations at lower levels. A number of possible operations on the computations at lower levels are:

- Examine scheduling attributes or other state variables specific to the application.

- Change scheduling attributes or other state variables.

An example is illustrated in Figure 6.8.

The concept of urgency is embedded in the multi-stage analysis process, but urgency is associated with the significance of scheduling decisions rather than a specific operation. Each computation module may have an associated timing constraint. The computation modules in the same stage may be scheduled according to their timing constraints whereas computation modules from different stages are scheduled according to urgency. As far as real-time performance is concerned, the multi-stage model is subject to similar analysis as other models which use priorities as importance values. The multi-stage model
6.2. SOFT REAL-TIME SYSTEMS

Figure 6.8: An Example of Multi-Stage Process
is different from other models in that it provides a framework for decision making which results in dynamically re-scheduling resources.

When the CPU is the only resource needed by the computation units, we can use the preemptive EDF rule to schedule the units at each stage, and the preemptive fixed priority rule to schedule the units at different stages. The simplicity of the preemptive fixed priority rule supports responsiveness to external events. It also supports the notion of overriding previous scheduling decisions.

### 6.2.2 A Multimedia Application

An important challenge in multimedia systems is to coordinate multiple sources of computer graphics, digital video, sampled and synthesized sound. Because of the sensitivity of human perception to slight variations in rhythmic patterns and other aspects of timing, and because of the need to synchronize various media, precise timing control is an essential requirement placed on the operating system. The timing problem is further complicated by the possibility of temporary resource congestion and the need to consider human perception.

In order to experiment with real-time requirements, we have developed a sound server which plays a sound sequence according to user data. Sound provides a good testbed for experiments because it is computationally intensive and because human hearing is especially sensitive to changes in rhythmic patterns when compared with the other senses. Moreover, the software control over the output of sound is periodic in nature, which is also typical to the handling of digital video and graphics. While designed as a tool to understand real-time performance, our sound server has exceeded our expectation by helping us to detect bugs in the performance of the kernel scheduler. Slight variations in rhythmic patterns in the order of milliseconds have been detected and have helped our debugging greatly (see Section 5.3). In the following, we examine the scheduling requirements of our sound server and describe how our scheduling interface meets its requirements. These requirements are also reflected in other media and
Periodic Activities

The software control over the output of sound, digital video and graphics is periodic in nature. In order to produce a sound stream, notes are played periodically. Similarly, images are displayed at a refresh rate. While there exists hardware designed to provide some periodic control of each media, precise timing control in the global synchronization of all media remains an issue [Flinn 93]. Because of the complexity of the synchronization problem, a software solution which allows for direct manipulation of media data is needed.

Our sound server serves as a good illustration of the kind of flexibility required in the implementation of periodic activities. It plays a sound sequence which is periodic, but which does not necessarily have a constant period, since a sequence can be the result of merging multiple independent sound streams together. The typical notion of periodic tasks characterized by a constant period, worst-case execution time, starting time, and abort time is not sufficiently general to implement this sound server. Although each individual sound sequence may have a constant period, the merged sequence does not. The function of aborting a task at a specific time does not make it easier to implement the policy of omitting some voice during congestion either, since the application must dynamically determine which sound to omit.

Hardware designed for sound may address the performance issue but not the flexibility problem. The early design of Real-Time Mach [Tokuda et al. 90] also adapted a rigid definition of periodic threads. Their recent design has been improved [Tokuda 93]. Dynamic changes to the period of a thread are allowed, and the abort time is replaced by a mechanism where a message is automatically sent to a death handler at a specified time.

Our approach provides a flexible and simple solution. A periodic activity can easily be implemented by a loop where the starting time of the task is reset at the end of the loop. Since the task may reset
the starting time to any value, it may also execute at irregular periods as required by the sound server. Alarm capabilities provide congestion detection and handling. Since there is no restriction on what program event to detect for timing failure, the alarm capabilities provide a flexible means for performing experiments on resource congestion.

**Scheduling Policies**

A major shortcoming of existing real-time software solutions is their lack of consideration for specific timing needs of human beings [Flinn 93]. For example, some complex real-time scheduling algorithms handle resource congestion by graceful degradation according to the cost and value of resources [Northcutt & Kuerner 91]. However, they are not natural for handling the problem of human perception. Some simpler solutions may be more appropriate when human perception is considered. If a machine becomes swamped during replay of digital video, it may be acceptable to simply drop frames until the resource contention is alleviated. If a synthesizer has reached its capacity to synthesize simultaneous voices, the omission of additional voices may go unnoticed. In situations where resources are predicted to be inadequate, the rate of sound generation and video replay may be reduced. Some experiments to determine the sensitivity of human perception to subtle variations in the timing of generated sound have been performed, and it was suggested that more needs to be learned about how errors in the synchronization of different media affect human perception [Flinn 93].

Our architectural support for application-level scheduling policies is appropriate for this experimental approach to scheduling. Schedulability analysis as well as resource scheduling can be designed and modified for individual application requirements, while a common task model and a core scheduler provide the framework for a modular implementation of the application.
6.2. SOFT REAL-TIME SYSTEMS

Tasking Structure of the Sound Server

We have implemented a sound server on the Mach 3.0 kernel with our kernel scheduler. The hardware configuration consists of a 486 workstation with a Sound Blaster Pro card. Our implementation includes a sound thread which controls the timing of the sound sequence and a thread which handles user input. The sound thread runs at a higher priority than all other applications so that the timing of the sound sequence is precise. After playing a note, the sound thread changes its starting time to the time when the next note should be played. This is done by calling `set_thread.sched_attr()`.

The thread which handles user input runs at a lower priority than the sound thread. Priority has been exploited as a means to maintain the integrity of the data shared between the two threads. When the thread which handles user input needs to change this data, it raises its priority to the same priority as the sound server. This ensures that the change will be completed before the sound thread runs again. In a multiprocessor environment where threads of different priorities can run simultaneously, such a solution is not adequate to maintain data integrity, and an extra locking mechanism would be required. In an uniprocessor environment, however, changing priority provides an inexpensive way to protect data.

The structuring of the sound thread and the user input thread reflects the concurrency of the two types of activities. The timing can be predicted by using the algorithms discussed in Section 4.4.1. When there is a congestion problem, the sound thread has total control over what notes to drop.
Chapter 7

Conclusions

Our overall conclusion is that an architectural approach to the support of real-time task abstractions and scheduling is feasible, practical and flexible. We have designed and implemented an architectural solution which provides a common task model and a scheduling interface for the implementation of application-level scheduling policies. This design allows for the implementation of competitive task models on top of a common kernel. It also provides more flexibility in task structuring since applications can be structured as cooperating tasks.

Our work is the first to combine the seemingly incompatible concepts of various task models in a common design. In Chapter 3, we have analyzed the competitive task models and the cooperative task model in terms of their strengths and weaknesses. The competitive task models provide many advantages in timing support which are not offered by the cooperative task model. However, the timing support and task primitives are tightly coupled with the task scheduler. There is a portability problem when the application evolves and requires another scheduling mechanism. The maintenance of the operating system is also difficult since competitive task models are commonly implemented in a monolithic kernel. The cooperative task model offers natural modularity in both task structuring and operating system design, but it is lacking in timing support. The common belief that the cooperative
task model was incompatible with the competitive task model did not help in providing insight to a more flexible design alternative. Despite of this common belief, we have analyzed and clarified some important concepts related to real-time tasking. We have found it possible to combine the advantages of two models in one design. This work has motivated our design, which is described in Chapter 4.

A novel idea in our design is to support a clean disassociation of the cooperating task model from a specific view of timing requirements so that the latter can be independently managed. This is accomplished by providing special scheduling support in the kernel. The task synchronization and communication mechanism is dedicated to the handling of asynchronous events without concern for resource contention problems. The kernel scheduler supports three scheduling attributes: the earliest starting time, the priority, and an optional deadline. It provides a set of core scheduling rules which are useful to applications with stringent timing requirements. Furthermore, a scheduling interface is provided so that any task scheduling attributes can be changed dynamically. Using this interface, complex scheduling mechanisms and competitive task models can easily be implemented as a set of cooperating tasks. An application can be structured as cooperating tasks or competitive tasks.

Schedulability analysis of the kernel scheduler is discussed in Chapter 4.4. Existing work is reviewed and our enhancement of two existing techniques is presented. The level-driven analysis described in [Harter 87] considers cyclic tasks with fixed priority, but a client task can only invoke a server task of the next higher priority. The analysis of periodic tasks with varying priority [Harbour et al. 91] does not impose any restriction on how priority can be varied. However, the period of each task must be constant, and the task cannot wait at any instant between their activation and their completion. Neither of these two schedulability analysis methods is sufficient for the server/client model where cyclic client tasks may invoke servers of any higher priority. We have combined and enhanced these two techniques to consider the server/client model which is commonly used in realistic systems.

Examples that demonstrate the flexibility of our solution for real-time software are provided in
Chapter 6. Our architectural design is the first to provide a modular structure to complex, monolithic real-time kernels. We have demonstrated the structuring of the complex scheduling mechanisms found in the Spring kernel and the RT Mach kernel as cooperating tasks on top of our kernel. Some systems need to be responsive to external events which can affect the internal timing constraints of the systems. Our scheduling interface provides direct support to the implementation of scheduling policies in these systems. The support of unconventional scheduling requirements in our design has been illustrated by the demands placed on the kernel by a modern multimedia application.

The practicality of our architectural design is supported by our implementation using the Mach 3.0 kernel and the r-kernel as development platforms. These two kernels are very different in complexity. The Mach 3.0 kernel provides rich functionality for the implementation of a time-sharing operating system. The r-kernel has been designed to provide fast thread management and is much simpler. We have implemented the same interface in both kernels and have described them in Chapter 4. Non-real-time software which uses other interfaces can also run, although they do not benefit from our real-time support. This has been particularly useful in the Mach environment, since we can continue to use the Unix server for our software development and at the same time test the real-time behavior of our sound server described in Chapter 6. Such flexibility is a proof of the utility of our design — real-time software can run on a common development platform originally designed for non-real-time software, and application-level scheduling policies are allowed.

An important motivation of our design has been that constantly changing the kernel for application-level scheduling requirements results in a less stable and maintainable kernel. The lessons we have learned from our implementation in the Mach 3.0 kernel and the r-kernel have strengthened our belief. Successful modification of the thread scheduler requires a lot of knowledge about kernel implementation as evidenced by the pitfalls we have encountered and described in Chapter 5. Such knowledge should not be required of an application programmer who needs to implement an application-level scheduler. By
providing a flexible scheduling interface, our design makes it possible for the application programmer to build the application-level scheduler on top of a stable thread scheduler and a reliable kernel.

7.1 Future Research

Realistic real-time systems are complex and dynamic. Much instrumental work is needed to further explore the methodology of structuring real-time programs as cooperating tasks when the scheduling requirements are dynamic and evolving. Besides scheduling issues, the use of a cooperating task structure to support graceful degradation and guarantee of quality of service requires more study. We expect that further insights and understanding will result from a wide application of our abstraction to real systems.

Our abstraction has been incorporated into the RT Threads package of the Distributed Continuous Media File Server project at UBC. There will be many tasks in both the client and server programs to handle different devices and to implement a variety of real-time functions. Some of these functions are independent of each other, but nevertheless their sharing of the same CPUs requires an overall consideration of scheduling problems and other issues such as graceful degradation. Experimentation is required to determine the best scheduling solution which provides a balance of the quality of the various functions. The cooperating task structures, the scheduling attributes of the tasks, and the times at which the scheduling attributes of the tasks are changed all require much thought. It is expected that the need to cope with dynamic changes in the workload will generate a wealth of problems which will provide insights into interesting principles of task structuring.

Our schedulability analysis of the server/client model could be extended to a multiprocessor environment or a distributed environment where the clients share the servers running on different nodes. The analysis of distributed soft real-time servers in real systems is an open problem.
Bibliography


