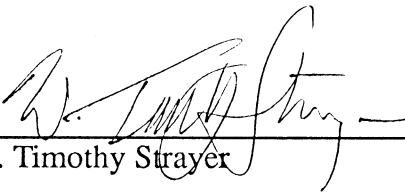



APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

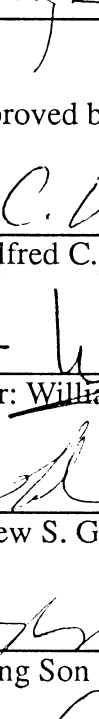


W. Timothy Strayer

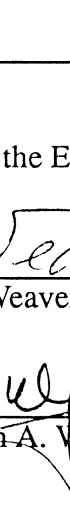
This dissertation has been read and approved by the Examining Committee:



Dissertation Advisor: Alfred C. Weaver



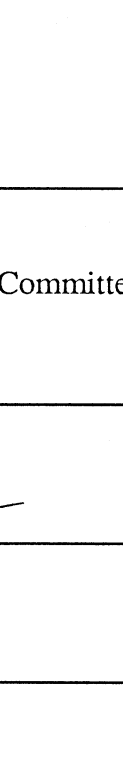
Dissertation Co-Director: William A. Wulf



Committee Chair: Andrew S. Grimshaw




Committee Member: Sang Son



Curriculum Representative: James H. Aylor

Accepted for the School of Engineering and Applied Science:



Dean Edgar A. Starke, Jr.
School of Engineering and Applied Science

May 1992

Function-Driven Scheduling: A General Framework
for Expression and Analysis of Scheduling

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

W. Timothy Strayer

May 1992

Abstract

Scheduling theory maintains that there are fundamental similarities in problems of sequence that transcend the characteristics of the particular tasks to be ordered or the resources to be used. Traditionally, scheduling policies are implemented using algorithms; we study scheduling algorithms to discover the various properties of the schedules they produce. To facilitate analysis the policies are typically limited to homogeneous task sets (e.g., all periodic tasks) and consider only one or very few task attributes. In some cases the results are so attractive that the task sets of systems are made to fit the algorithm rather than using a policy more appropriate to the system. We therefore make the following observation: if scheduling policies are driven by how well they can be expressed and analyzed, then we need a more general framework for expressing scheduling policies.

We introduce the *Importance Abstraction* as a general scheduling framework. The scheduling algorithm is invariant: choose the most important task at every point in time. Each task is described by a function, called an *importance function*, that profiles the task's importance to the system over time. The importance abstraction can express not only the traditional scheduling policies but a wide range of new policies based on how important individual tasks are to the system. Since the scheduling policy is described using functions rather than a single algorithm we can exploit the maturity of mathematical proof techniques when analyzing the schedule produced by the policy. Since this abstraction is applicable to any system of tasks and processors, we examine the communication subsystem as an example, and find that importance functions facilitate the expression of message discrimination policies as well as help unify scheduling across the operating system/communication subsystem domain boundary.

Acknowledgements

I thank and acknowledge the people who have helped and encouraged me as I pursued this degree—there were of course more than I can name here. Over the last six years I have had the pleasure of working with and learning from one of the most talented collections of people in this department: the Computer Networks Laboratory, in particular, Robert Simoncic, John Fenton, James McNabb, Alex Waterman, Fraser Street, Jeff Michel, and past lab members Jeffery Peden, Alex Colvin, Duke Harvey, Kelly Philips, Dave Minnich and Randy Simonson. I especially thank Bert Dempsey for the many exchanges of ideas, visions, critique, and abusive phrases. For all of their patience when “RTFM” was more appropriate, I thank Mark Smith, Ray Lubinsky, and Gina Bull. For all the times their quick intervention saved my neck, I thank Ginny Hilton, Carolyn Duprey, Kim Gregg, Barbara Graves, Pam Evans, and my friend in the front office, Tammy Ramsey.

I am deeply grateful for the good and lasting friendships I’ve made here, especially Bryan Catron and Paula Gabbert Catron, Kevin and Julie Treu, Pat and Cindy Heck, Ray Wagner, Phil Dickens, Molly and Bert Dempsey (see above), Mark Langfitt and Ellie Radford, Rob McGee and Catherine Thomas McGee, Rachel Lorey, and Karen LeMaire. And everyone knows that 228 was the coolest office. Thanks for all the softball games, the cookouts, the parties, and the tightness of a circle of friends.

I am grateful to Lee Cohen, my abiding best friend from college days, for providing an ear to bend, a place to hide, an excellent Bloody Mary to drink, and support as I plowed through this degree.

I owe a substantial portion of this degree to Carmen Pancarella who, through all of our ups and downs, has remained my best friend and most trusted confidant. She pulled me when I got stuck, pushed me when I got stubborn, and never let me forget that life is larger than graduate school. She kept my stomach full, my head from getting too large, my appointments from being missed, and my days from being boring.

I would be nowhere without the love and support of my family. They have heard the phrase “I’m about a year away from finishing” often over the last three years, yet they always believed, sometimes more than I, that I’d finish.

I very much appreciate the effort and guidance of my committee, Andrew Grimshaw, Sang Son, Jim Aylor, Bill Wulf, and Alf Weaver. When I first had some of the ideas presented in this dissertation, all I saw were dead-ends and limitations. I am forever grateful to Bill for seeing past these obstacles and sometimes having to convince me, rather than the other way around, that these ideas were good. My thank-you’s will never repay all of the time Bill has spent with me.

My greatest appreciation of all goes to Alf Weaver. From the beginning of my tenure at Virginia until the very last moments of our advisor/advisee relationship, Alf has given far beyond what was required. He has always treated me as a respected colleague, selflessly promoting me and all of his students above himself. I have learned a great deal from Alf, not just concerning research, but more importantly, how to make the most with what you’ve got, and that most of what you’ve got comes from the people around you. And all I know about wines, I learned from him. A person like Alf comes into one’s life perhaps once, if one is really lucky. Alf is more than a teacher and advisor; Alf is my friend.

This work is supported in part by the U.S. Naval Ocean Systems Center and the Office of Naval Research under contract number N00014-91-J-1514. In particular I thank Will Gex and Les Anderson of NOSC for their interest in this work.

Dedication

This dissertation is dedicated to my grandmother. Although she has no idea about the science behind precedence relations and resource allocation, she knows more about the art of real-time scheduling than any academician I know: the proof is quite literally in the bread pudding. And the dumplings and pickled beets and wilted lettuce and vinegar pie...

“A schedule defends from chaos and whim.”
—Annie Dillard, **The Writing Life**

Table of Contents

Abstract	iii
Acknowledgements	iv
Dedication	vi
Table of Contents	vii
List of Figures	x
Chapter 1 Introduction	1
1.1. Function-Driven Scheduling	2
1.2. Issues	3
1.3. The Goal of this Research	5
1.4. Motivation	5
1.5. Contribution	8
1.6. Thesis Overview	10
Chapter 2 Scheduling Theory	11
2.1. The Scheduling Problem	13
2.2. System Model	14
2.2.1. Tasks	15
2.2.1.1. Task Attributes	16
2.2.1.2. Task Constraints	17
2.2.1.3. Operations	18
2.2.1.4. Task Sets	19
2.2.1.5. Task Arrivals	19
2.2.2. Scheduler	21
2.2.2.1. When to Schedule	21
2.2.2.2. How Often to Schedule	22
2.2.2.3. Complexity of Scheduling	24
2.2.3. Resources	26
Chapter 3 Survey of Scheduling Techniques	28
3.1. Rate Monotonic Theory	28
3.1.1. Period Transformation	30

3.1.2. Priority Inheritance and Priority Ceiling Protocols	31
3.1.3. Deferrable Server	32
3.2. Survey of Function-Based Scheduling Techniques	33
3.2.1. Policy Functions	33
3.2.2. Time-Driven Scheduling	36
Chapter 4 Importance Abstraction	40
4.1. System Model	40
4.2. Importance Functions	43
4.2.1. Sets of Importance Functions	43
4.2.2. The Defining Property of an Importance Function Set	44
4.3. The Scheduler	45
Chapter 5 Expressiveness of the Importance Abstraction	47
5.1. Emulation of Traditional Scheduling Policies	48
5.2. Families of Importance Functions	53
5.3. Novel Policies using Importance Functions	54
Chapter 6 Analyzability of the Importance Abstraction	59
6.1. Policies with Statics Rankings	60
6.1.1. Determining Completion Time	61
6.1.2. Meeting Deadlines	63
6.1.3. Meeting Deadlines with Arbitrary Arrivals	66
6.1.4. Meeting Critical Deadlines, with Arbitrary Arrivals	68
6.1.5. Heterogeneous Task Set	72
6.1.6. Heterogeneous Task Sets with Critical Deadlines and Arbitrary Arrivals	74
6.2. Projections	76
Chapter 7 Implementing the Importance Abstraction Efficiently	79
7.1. Discrete Evaluation Points	79
7.2. Approximations	81
Chapter 8 The Communication Subsystem	83
8.1. Discrimination Techniques	84
8.1.1. Priorities	84
8.1.2. Levels of Service	87
8.2. The ISO Reference Model	89
8.2.1. Media Access Control Layer	91
8.2.2. Network Layer	93
8.2.3. Transport Layer	96
8.2.4. Summary of Issues	97
8.3. Using Importance Functions	98

8.3.1. Messages and Importance Functions	99
8.3.2. Packets and Importance Functions	101
8.3.3. Stations and Importance Functions	103
8.3.4. Decision Points	104
8.3.5. Other Uses for Importance Functions	104
8.3.6. Importance Functions and Levels of Service	106
8.3.7. Analysis	107
Chapter 9 Extended Example	112
9.1. Task Definitions	113
9.2. Task Level Scheduling	117
9.3. Message Level Scheduling	123
9.4. Packet Level Scheduling	124
Chapter 10 Conclusions	127
10.1. Summary of Work	128
10.2. Contributions	129
10.3. Future Research	131
Bibliography	133
Appendix A	137

List of Figures

Chapter 2

- Figure 2.1 — General System Model 14
Figure 2.2 — General Model for Scheduling 16

Chapter 3

- Figure 3.1 — Value Function Shapes 38

Chapter 4

- Figure 4.1 — The System Model 41
Figure 4.2 — Network Interface Unit and Network “Systems” 41
Figure 4.3 — Communication “Subsystem” within a Distributed “System” 42

Chapter 5

- Figure 5.1 — Importance Function Values for FCFS Policy 49
Figure 5.2 — Importance Function for an Nearest Deadline Task 51
Figure 5.3 — Importance Function Values for Round Robin Policy 53
Figure 5.4 — Valve Configuration for a Process Control Example 55

Chapter 8

- Figure 8.1 — Taxonomy of Discrimination Techniques 85
Figure 8.2 — The OSI Reference Model 90
Figure 8.3 — Medium Access Control 91
Figure 8.4 — An Internetwork: Stations, Routers, and Network Segments .. 95

Chapter 9

- Figure 9.1 The `file_server` Task Definition 114
Figure 9.2 The `video_transfer` Task Definition 115
Figure 9.3 The `send_alarm` Task Definition 116

1 Introduction

Scheduling theory maintains that there are fundamental similarities in task sequencing problems which transcend the characteristics of the particular tasks to be ordered or the resources to be used. Tasks arrive, require some amount of resources, and depart. The act of choosing which of the tasks receive attention, and the amount of resources granted, is called *scheduling*. Scheduling theory gives us a framework within which questions may be asked about schedules, such as, “Can this task set be scheduled such that some criteria is met?” and “Which tasks will be served under certain conditions, such as overload?” The thesis of this work is to examine a new method to derive answers to such questions.

A scheduler chooses which of the tasks currently contending for service will be granted service, yet the scheduler must be told how to make this decision, and how often. The scheduler has two primary sources of information: the attributes of and constraints on the tasks, and the characteristics of the system within which the tasks and the resources reside. The set of rules which instruct the scheduler *when* to make a decision and *how* to rank the tasks for the decision is called a *scheduling algorithm*.

There are two notable observations common to the use of scheduling algorithms: (1) scheduling algorithms tend to rely on one or a few specific task attributes or system characteristics for establishing a ranking criteria, and (2) all tasks are ranked according to this criteria. For example, the rate monotonic scheduling algorithm [LIU73] uses the *period* of each task to determine the task’s ranking, whether or not this is appropriate for each

individual task or for the system as its behavior evolves over time. The result is that schedules may not follow naturally from the task attributes and system characteristics.

1.1. Function-Driven Scheduling

Consider a model for scheduling tasks where each task is preemptable and has an associated function that profiles the task's importance over time. Assume that all of these functions are normalized so that comparing the values of any two tasks' functions at a particular point in time would indicate which of the two tasks is more important to the system. At any point in time the tasks in the system can be ranked according to the values of their functions, and thus according to their importance to the system at that moment in time.

If tasks were assigned functions that profiled how important they were to meeting the system goal, and the system ensured that at every point in time the most important task in the system was receiving service, then the resulting schedule of the tasks would be the *optimal schedule* with respect to meeting the system goal. Stated alternatively, the system would always be doing the best it possibly could under the circumstances. Note that this does not imply that the schedule will provide all of the work required by all of the tasks, or that the schedule will necessarily meet the goal of the system; however, it does state that if the goal can be met, this schedule will meet it. These functions associated with each task we term *importance functions*, and the scheduling model within which these functions serve we term the *importance abstraction*.

There is a spectrum of forms of expressing scheduling policies. Traditionally the scheduling policies are expressed using algorithms. Function-driven scheduling using the importance abstraction represents another point in this spectrum, where expressiveness derives from using functions to describe particular scheduling policies. Importance

functions are particularly expressive since each task is described by a function tailored to that task. These functions include as parameters those attributes and characteristics upon which the task's importance to the system is based. Consequently, schedules are produced by considering what conditions make an individual task important rather than by trying to find an algorithm that fits all criteria for a given task set. Since the principle component of the importance abstraction is a set of functions rather than an algorithm, the analysis of the scheduling of tasks benefits from the maturity of the analysis of functions.

1.2. Issues

There are three issues to be addressed concerning the usefulness of the importance abstraction: (1) how expressive the abstraction is, (2) how easily questions can be posed and answered once the scheduling policy is expressed within the abstraction, and (3) can policies expressed within the abstraction be implemented in an efficient manner.

The first issue considers the expressiveness of the importance abstraction. The importance abstraction can emulate the "traditional" scheduling algorithms by creating functions such that the schedule for a given task set produced by the importance abstraction is identical to that produced by the algorithm (see Section 5.1.). Consider the nearest deadline first algorithm, for example. For any two tasks i and j active at time t , with deadlines d_i and d_j , the importance functions $I_i(t)$ and $I_j(t)$ are constructed such that, for all t ,

$$d_i < d_j \Rightarrow I_i(t) > I_j(t) \quad (\text{Eq 1.1})$$

Since algorithms choose tasks according to specific conditions, we may, in general, construct a set of importance functions for a given task set such that a task within that set will become most important precisely when the algorithm would choose that task for service.

The concept of profiling a task's importance over time is intuitive, and using functions to express this importance seems more "natural" than using equivalent algorithms. By using functions we can easily encompass a wide range of scheduling parameters, and we can easily tailor importance profiles to individual tasks. It may be possible to express the same scheduling policy in an algorithmic domain, but designing an algorithm to achieve this generality would prove more cumbersome than a function-based approach.

The second issue considers the analyzability of the importance abstraction. By expressing the scheduling problem in a functional domain rather than an algorithmic one we gain the use of tools supplied by mathematics. We may invoke proof techniques that are more extensive than those used in proving properties of algorithms. If we can phrase our scheduling questions in terms of functional mathematics, then we may apply the machinery of the mathematics to help answer these questions. Very complex and subtle scheduling problems can be expressed using importance functions and, since we are dealing with functions, functional mathematics can be applied to help provide analysis.

The third issue considers the implementability of such a framework within a real system. Clearly it is impossible to evaluate each task's function at every moment in time to ensure that the most important task is always identified. However, it is sufficient to ensure that the most important task is known at every point in time; the importance functions require evaluation only when a new most important task must be chosen. If it is possible to identify when the evaluations must take place, it may be possible to implement this scheme in a cost-efficient manner. It may be the case that some restrictions must be placed on the functions so that the scheduling policy may be implemented efficiently. If this is so, it is essential to discover how such restrictions affect the expressiveness of the importance abstraction.

1.3. The Goal of this Research

We start with a system model wherein there is an identifiable and enumerable set of preemptable tasks. Initially these tasks are the “possible” tasks, but as the system runs, these possible tasks will become forms for their instantiated counterparts, the “active” tasks. Given a set of (possible) tasks, there exists an infinite number of sets of importance functions; that is, there is a *universe* of importance function sets that can profile the importance of the task set. When the system goal is stated, this universe of sets is partitioned into those that guarantee that the schedule they impose will meet the goal, and those sets that cannot make that guarantee. The set of importance function sets which guarantee to meet the system goal forms an equivalence class for that system’s characteristics. We seek the set of properties which uniquely defines this equivalence class so that, given a set of importance functions, we can apply the properties to determine if the set belongs in the equivalence class. The goal of this research is to set forth a method for determining this set of properties, if not in the general case, then for several example scheduling policies. Along the way we will

- describe and develop the importance abstraction as a model for expressing scheduling policies, both traditional and novel
- develop the analysis for a traditional scheduling policy using the importance abstraction
- describe a framework within which function-based scheduling is effective and efficient
- apply this abstraction to an application, namely a communication subsystem within a distributed system

1.4. Motivation

There are many scheduling algorithms whose properties are studied for complexity, schedulability, and solvability; three of the most common are *priority-driven*, *rate monotonic*, and *deadline-driven*. The highest-priority-next scheduling policy ensures a queueing discipline where the highest priority tasks enjoy first come, first served service

although no guarantees can be made regarding the remainder of the priority values. A priority value is associated with a task when the task becomes active, and remains the task's measure of importance throughout the lifetime of the task. Priority scheduling is particularly prolific in communication subsystems where priority is often used to access the common medium. In general, the schedules produced have only the property that the highest priority task will eventually receive some service. Choosing a single priority value to reflect the task's importance over its lifetime implies either that all of the task's attributes and system characteristics that could contribute to the task's importance are reduced to a single value, or that a task maintains the same ranking relative to all other tasks over that task's lifetime. Consequently, static priority-driven scheduling policies cannot easily adapt to changes in the system.

Rate monotonic theory provides a means of statically assigning priorities to tasks while retaining certain properties of the schedule produced. If all tasks are periodic, ranking tasks by the inverse of their periods will produce a schedule where no tasks will ever miss a periodic deadline if the task set meets certain loading conditions. This has been a landmark result, especially in real-time scheduling. Unfortunately, this theory applies only to periodic tasks; aperiodic (sporadic) tasks must be "fit" into a periodic server so that all tasks appear to the scheduler as periodic. The ranking of tasks by period works well when loading conditions are met, but a task set is rarely static, and as tasks enter and depart the system, transient overloads may occur. In rate monotonic theory, these overloads cause tasks with the longest periods to be shed first, a consequence that has no relationship to how critical the task may be to the system. Period transformation techniques have emerged to superimpose a criticality ranking on the task set by artificially splitting the critical long-period tasks into what appears to be many smaller periodic tasks. Such measures as a

sporadic task server and the period transformation techniques “bend” rate monotonic theory to include these cases.

Deadline-driven scheduling, including most prominently the nearest deadline first policy, uses the deadline of a task as the sole task attribute in scheduling a task set. Nearest deadline first guarantees that all tasks will meet their deadlines if the task set is schedulable. Unlike rate monotonic theory, there is no requirement that the tasks be periodic. However, the task set must be homogeneous in that all tasks must have a deadline, all tasks are required to meet their deadline with equal necessity, and no other conditions or attributes need be used to rank these tasks. Again, the criticality of a task is tied to an attribute that does not have anything to do with how critical the task is to the system.

Common problems with these and other well-known and traditional scheduling algorithms are that they typically do not naturally include the attributes and system characteristics which cause a task to be “important.” Scheduling decisions usually focus on a small number of attributes, while in general the importance of a task is conditioned on what is happening within the system at that particular time as well as the constraints of the task. When a scheduling policy like nearest deadline first is modified to include even a criticality ranking, the guarantees derived from analysis are no longer necessarily valid.

The analysis of scheduling algorithms is also a difficult problem. In highly constrained systems, such as real-time systems, where lives and money depend on the correct operation of the system, system designers typically attempt to analyze the system statically, prior to turning on the system. Often this is the only time when the system can be analyzed. Yet such systems are so complex that a complete static analysis, or proof of correct operation, is impossible. In scheduling theory, analyzing task sets quickly becomes complex. Many of the interesting scheduling problems are NP-complete, and those that are

not suffer from oversimplification of the problem. As a consequence, scheduling algorithms are typically chosen based on which can best be analyzed.

In accordance with our research goals, we seek a framework wherein the task's importance to the system is easily expressible, the questions regarding the schedules produced can be analyzed in a straightforward manner, and the framework is implementable in an efficient manner. A solution of this problem is the dissertation's contribution to the field.

1.5. Contribution

This dissertation contributes to the field of scheduling theory by

- the development of a function-based scheduling framework, namely the *importance abstraction*
- the exploration of several issues, in particular, the expressiveness of the importance abstraction, the analyzability of scheduling policies within the importance abstraction, and the efficient implementation of the importance abstraction
- the application of the importance abstraction to a real system, namely a communication subsystem

This dissertation introduces the *importance abstraction* as a framework for implementing scheduling policies. The importance abstraction is novel in that it uses a function-driven approach for describing the scheduling policy. There have been previous function-driven approaches, in particular those of [BERN71], [RUSC77], and [JENS85]; the importance abstraction is unique in its use of functions to describe how important a task is to the attainment of the system goal, and its use of these descriptions to perform analysis of the schedules produced.

The scheduler within the importance abstraction is simple and universal. The scheduler ensures that at every point in time the most important task, according to the importance functions, is being serviced. Since the scheduler is universal, various scheduling policies can be implemented within the abstraction simply by using different

sets of importance functions. Thus the importance abstraction is a framework within which a wide range of scheduling policies can be implemented and analyzed.

This dissertation explores three issues concerning the importance abstraction—how expressive the abstraction is, how conducive to analysis it is, and how the abstraction can be implemented efficiently. First, the use of a function to profile a task’s importance emphasizes the conditions under which the task becomes important. Since a function is associated with each task, these conditions can be individualized. By properly choosing the set of importance functions for a system, traditional scheduling algorithms can be emulated. Whereas algorithms generally consider a task set with homogeneous constraints to determine which task to schedule, the importance abstraction allows scheduling policies such that the conditions which make a task important are tailored to the task.

Second, since the importance abstraction shifts the description of the scheduling policy to a set of functions rather than a single algorithm, we can use the rich techniques of mathematics to examine scheduling policies rather than using algorithmic techniques.

Third, this dissertation also examines issues concerning the efficient implementation of the importance abstraction. In its general form, the importance abstraction assumes that each importance function is evaluated at every point in time. Practical systems cannot meet this assumption. We explore several methods for relaxing this assumption so that the abstraction can be implemented.

We show the application of the importance abstraction to a communication subsystem. Scheduling theory revolves mainly around the task scheduling within the operating system. The importance abstraction is a general framework, therefore the results apply to any system, including a communications subsystem.

1.6. Thesis Overview

This dissertation is organized as follows.

Chapter 2 offers a review of fundamental scheduling theory.

Chapter 3 provides the background survey of scheduling theory results, specifically those approaches which use functions to help perform scheduling.

Chapter 4 introduces the importance abstraction and the system model assumed.

Chapter 5 examines the expressiveness of the importance abstraction, providing examples of how traditional scheduling algorithms can be emulated within the abstraction, and showing how novel scheduling policies can be constructed.

Chapter 6 demonstrates the abstraction's usefulness for performing analysis of scheduling policies implemented within the importance abstraction.

Chapter 7 explores the issues involved in efficiently implementing the importance abstraction.

Chapter 8 describes an application of the importance abstraction to a system, namely a communication subsystem.

Chapter 9 is an extended example. This example includes tasks that employ a communication subsystem.

Chapter 10 offers conclusions drawn from this research, and offers avenues for future work.

2 Scheduling Theory

Scheduling is the act of sequencing tasks. It involves arranging, coordinating, and planning the use of resources to achieve some goal. More simply, it relates to the ordering of getting things done.

Scheduling theory maintains that there are fundamental similarities in problems of sequence which transcend the characteristics of the particular tasks to be ordered or the resources to be used. It is an abstraction of real-world tasks and resources into an optimization problem of the following form: given a collection of tasks to be performed within some system, where the tasks are subject to various constraints on when and how they may be performed, find a viable sequence of these tasks such that this sequence meets some objective better than any other sequence could.

Ordering everyday events to make more efficient use of the resources available or the time available to do things is a common occurrence. Yet, it was not until the Industrial Revolution of the 1900's that the first rudimentary scheduling techniques were employed to aid production. During the First World War, Henry Gantt developed the most recognized graphic representation of scheduling, the Gantt Chart, for organizing cargo on Allied ships. Then in the Second World War the British government called upon its national scientific trust to study the problem of allocating the country's dwindling resources. In the post war era, the United States' industries began in earnest to use these more formal techniques to make production more efficient. The science that grew out of the formalization of the techniques for allocating scarce resources is called Scheduling Theory, a branch of *Operations Research*.

Operations Research (OR) applications attempt to get the best practical result for a problem posed under certain constraints. It is a scientific method for providing a quantitative basis for decision making. The techniques of OR give a logical and systematic way of formulating a problem so that the tools of mathematics can be applied to find a solution. Linear programming is one such formulation.

In the 1950's computers were used to implement the logic of formal allocation and scheduling techniques. Although the computer was a powerful tool for finding solutions to linear programming problems, the allocation of the processing service within the computer was still done by humans. Today, a large segment of research within Operations Research studies the problem of task scheduling within computers for better utilization.

Computers were also used to monitor and control processes external to the computers themselves. Sensors placed within the external environment, such as within a chemical reaction chamber, relay the changes within that environment to the computer. The computer analyzes these changes and correspondingly modifies the process. Since the activity of the computer happens concurrently with the dynamics of the environment, the computer is said to be acting in *real-time*. Real-time systems are characterized by time-constrained processing. Much of the recent work in scheduling has focused on the utilization of processors such that the most important activities are accomplished within their time constraints.

This chapter is a brief introduction to the theory of scheduling to lay the foundation for exploring previous work in scheduling. It introduces scheduling and scheduling problems. It offers a model of an abstract system in which scheduling occurs to match resources with requests for service. The problem is also viewed from both the perspective of algorithms and queueing theory. The algorithmic interest is in the inherent complexity of scheduling problems and how they relate to other complex problems. Queueing theory

looks to derive closed forms models of systems where things are caused to wait. Finally it is suggested that, although related to both of these areas, scheduling theory is motivated by a different objective, and scheduling theory offers insights in addition to those possible by either algorithms or queueing theory.

2.1. The Scheduling Problem

The general problem of scheduling is to determine for a given set of tasks whether a sequence exists for performing the tasks such that the constraints on the tasks are met, and to produce the optimal sequence if one exists. There are really three problems: a decision problem, a construction problem, and an optimization problem.

It is first a decision problem in the algorithmic sense, requiring some *machine* to take as inputs the set of tasks and their constraints, and answer yes or no to the question, “Does there exist a sequence such that all tasks are satisfied and their constraints met?” The second problem is closely related: if a sequence exists, produce it. If the sequence does exist, then the answer to the first problem is yes since the second problem produced the proof by construction. Although there exist simple algorithms which can find solutions to scheduling problems with certain restricted constraints, in general the interesting problems have no such efficient algorithms.

When a scheduling algorithm can meet the constraints of every task, then the schedule produced is optimal with respect to the constraints of the tasks. However, the scheduler may use some objective when choosing tasks so that the tasks that are chosen will help meet the objective. An *optimal schedule* is one which meets the objective (optimizes an objective function) better than any other schedule, even if not all of the constraints of the tasks are met. An *optimal scheduling algorithm* is a machine which will always produce such an optimal schedule with respect to the objective function. If we can find such a

machine, then we can always find the third part of the scheduling problem: the optimal solution.

If finding the optimal schedule requires more time and effort than is justified by the particular application at hand, and it often does, then approximate schedules can be found by constructing machines which sacrifice the exactness of the solution for more efficiency in finding some close solution. These approximation algorithms still seek to optimize the objective; the difference is that they use less comprehensive methods to do so. Consequently, the optimization part of the scheduling problem is still a valid pursuit even when the scheduler is almost certain of not finding *the* optimal solution.

2.2. System Model

Given a specific scheduling problem instance we can apply formal or even *ad hoc* techniques to find a solution that best meets the problem requirements. However, in order to generalize the scheduling problem and the techniques used to find solutions, we must define a model which embodies essential characteristics without being specific to any particular instance of the problem. In this section we offer a general system model, shown pictorially in Figure 2.1.

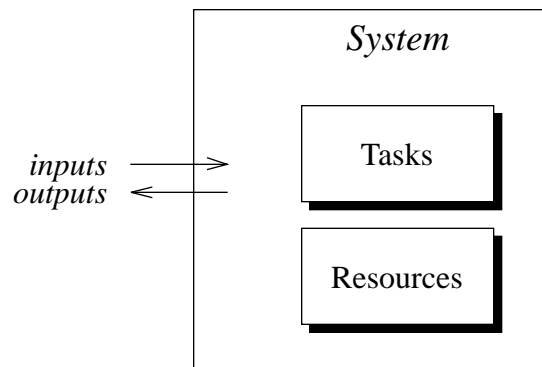


Figure 2.1 — General System Model

A *system* is a finite state automata which progresses from state to state by “processing” operations contained within a set of *tasks*. A system has an identifiable *goal* which drives each state transition. The set of tasks within the system represents the division of the system goal into many subgoals, where each task is responsible for accomplishing some part of the system goal. External events are communicated to the system via *inputs*, and external events are affected by the system via *outputs*. There is a set of internal resources for which tasks may compete and with which tasks may be processed. When the system is processing its tasks the system is said to be *running*.

In order to accomplish the system goal there must be some mapping, or schedule, of tasks onto resources over the time that the system is running. There are three required entities for scheduling: a set of tasks, a set of resources being requested to perform or aid in performing the tasks, and the scheduler. Figure 2.2 shows the relationship between the three entities. The tasks are kept in a task set *keep* of unspecified order until the scheduler is invoked, at which time some subset of the tasks is chosen according to task constraints and the state of the environment at that moment in time. The set of tasks currently being serviced at the set of resources is exchanged for the set of new tasks which will gain service. Some tasks may belong to both sets, some may have completed service and thus may leave the system, and some may be partially complete and must return to the keep for consideration in subsequent invocations of the scheduler.

2.2.1. Tasks

In practice we usually identify a task by what must be done. When performed, a task accomplishes some minor goal en route to some much larger goal. In scheduling, however, we use the term *task* as a shell to hold various attributes and conditions about what work must be done without necessarily identifying the work specifically. In this manner a task in scheduling is an abstraction for the specification of requested work.

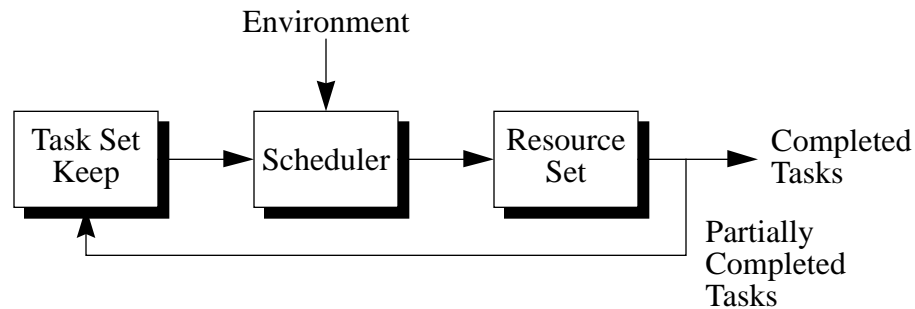


Figure 2.2 — General Model for Scheduling

2.2.1.1. Task Attributes

A task has two *inherent attributes*: the amount of time this work will take, and the earliest time that this work can commence. The amount of work required by the task is called the task *length*. It is not always possible to know exactly what length a task will have since the work that must be done by the task may vary with the conditions within the system. It is possible that the task may never finish. If a task is expected to finish, but there is variability in the finish time of the task, the length may be given as the worst-case for performing that task. Furthermore, there may be a different length (if it is possible to know it) for each different capacity within a set of processors. However, a task must always have a length, even if the length is unbounded, unknown, or indeterminate.

The earliest starting time is called the *ready time* of the task. This time is also known as a *release time* or an *arrival time*. It represents unambiguously the earliest time that the task may be performed. It is independent of, but may reflect, what other tasks have been performed and what resources or processors are available. Every task has this attribute, although it may not be known until the task actually is ready.

A task may have other attributes, such as a period, but these attributes do not necessarily apply to all tasks in the task set.

2.2.1.2. Task Constraints

Task *constraints* are conditions under which the task is to be performed. These conditions may include such constraints as *timeliness* constraints (when the work must be completed), *precedence* constraints (what work must precede this work), *processing* constraints (where this work is to be done), and *resource* constraints (what set of resources will be required for this work), or any combination of these.

Timeliness constraints specify the period of time during which the task is valid. When a task becomes ready it remains ready until a timeliness constraint specifies that it is no longer valid to perform this task. When this time of invalidity can be identified as a single instant in time, it is referred to as a *deadline*. Timeliness constraints also include such information as an optimal target completion time, which differs from the deadline in that a task may remain valid after this point in time. The most general specification of timeliness constraints is given by providing a function which specifies the value or criticality of the task as a function of time.

Precedence constraints establish a partial ordering on the set of tasks in a way that specifies that one or more tasks must be completed before a particular task can be started. These conditions can be represented by a directed acyclic graph where the nodes are the tasks and the arcs are the precedence relations.

The *processor* is the resource which can actually perform the task. Processor constraints dictate which processor or processors (commonly, but not necessarily, a central processing unit of a computer) must perform the task. If there is only one processor in the system then the constraint is trivial. The task may specify that one of several identical

processors may perform it, in which case the constraint is the specification of which group is appropriate. Dissimilarities in the processors may further constrain where a task can be performed. Such dissimilarities include but are not limited to processor capacity or speed, location, connectivity, or processing functionality.

Resource constraints dictate which of the various resources within the resource set are required by the tasks, and when they are required. These resources may be required by the task so that the task may be performed, and differ from the processors in that they are passive as far as performing the task is concerned. A task may have a list of resources which must be secured before or during the processing of the task. Resource constraints are trivial if no two tasks will ever require the same resource at the same time. Examples of resources within a computer include buffer space, disk access, and the communications subsystem.

Members of the task set are not required to have a common set of constraints. In general, a scheduler must weigh all of the constraints and all of the environment parameters when making scheduling decisions.

2.2.1.3. Operations

A task is comprised of the concatenation of one or more atomic units of action called *operations* [CONW67]. Since flow of control, which normally proceeds from one operation to the next concatenated operation, may be altered with conditional branching, there may be a large number of possible orderings of the operations. The actual ordering of the operations may, in general, not be known until runtime; hence the length of the task, which depends on the number and length of the operations within the task, may not be known until runtime as well. Thus the length of a task, although an inherent attribute of the task, may remain unknown until the task is actually completed.

An operation is indivisible but it may be arbitrarily small. Preemption, if allowed, cannot occur during the middle of an operation; rather it must wait until the operation is complete. Preemption can be instantaneous only in the theoretical sense that would imply that the length of an operation must approach arbitrarily close to zero.

2.2.1.4. Task Sets

A system has a finite set of tasks which belong to the set of *possible* tasks within the system. This set of possible tasks may be derived by inspection of the system. A member of this set of possible tasks may become instantiated, which is to say that an instance of the task is given an arrival time. When a possible task is instantiated, but before it arrives, that task is termed *inactive*, and belongs to the set of inactive tasks. Since a possible task may have an infinite number of instantiations (e.g., if the possible task is a *periodic* task) the inactive task set may have an infinite number of members. When the inactive task arrives, it becomes *active*, and joins the set of active tasks. At any point in time during the running of the system there is a finite number of tasks in the active task set.

2.2.1.5. Task Arrivals

There are three classifications of task arrivals: *deterministic arrivals*, *stochastic arrivals*, and *random arrivals*. The deterministic arrival class is a proper subset of stochastic arrival class, which is a proper subset of random arrival class.

Deterministic arrivals have several subclasses. The first is when all tasks simultaneously arrive prior to the start of the system. Scheduling problems of this type have a finite number of tasks, all with arrival time 0. Schedulers can statically consider and serve all tasks until no task remains to be served. Although this subclass is the most restrictive with regard to arrivals, and perhaps the most unrealistic, still there are relatively few problem specifications for which there are known efficient algorithms.

Another subclass of the deterministic arrival class represents tasks whose arrival times are known *a priori* but are not necessarily all simultaneously started at time 0. If some subset of the task set is related by the fact that each task arrival is separated by a constant interval, then that subset is called *periodic* and the member tasks are called *periodic tasks*. A periodic task subset has a period, a start time, and possibly an ending time. If there is no ending time, the periodic task subset, and consequently the entire task set, is an infinite set.

The stochastic arrival class also describes infinite task sets, but the arrival of an individual task cannot be known exactly. Rather, it is only known when the task actually arrives. However, the collection of arrivals of a subset of tasks may be related by a probability that there is some fixed amount of time between the arrivals. This is called the *interarrival time distribution*, and is given by

$$A(t) = \text{Prob} [\text{interarrival time} \leq t] \quad (\text{Eq 2.1})$$

where t is some fixed time interval.

Stochastic arrivals are most useful when considering the system as a flow system where tasks enter and leave and queues are formed at the server set. Queueing theory analyzes such stochastic flow systems and seeks mathematical expressions for several measures of interest. Such analysis includes answering questions like: (1) how long may a task expect to wait before being served, (2) how many tasks will be serviced before a newly arrived task is served, (3) what fraction of some time interval will the server set be busy or idle, and (4) how long will the intervals of continual busy work extend [KLEI75].

Random arrivals are not classified by identifying a period or an interarrival time, although it is possible that some tasks could have such a relationship. Whereas deterministic arrivals are required for static scheduling, in general the arrival of a task is not known until the task actually arrives. Scheduling such a task set requires on-the-fly

decisions since the composition of the active task set is dynamic. The scheduler must therefore make decisions in concert with the operation of the system, and the cost of such decisions is incorporated into the overhead of running the system. Whereas in static scheduling time and energy are not as critical when computing a solution to the problem at hand, the efficiency of the scheduling algorithm is the key to dynamic scheduling. It is in precisely this situation that optimal solutions may need to be abandoned for more efficient heuristics to reduce the impact of scheduling overhead.

2.2.2. Scheduler

The scheduler is responsible for surveying the set of active tasks and choosing certain of those tasks to receive service from the processor such that the constraints within the task are met and some objective function is optimized. The scheduler follows rules when making a scheduling decision; the collection of these rules is called a *scheduling algorithm*. How often the scheduler is invoked to allocate service is a function of how often the requests are generated, the nature of the constraints within the requests, the physical characteristics of the scheduler, and the nature of the resources. Theoretically, only a scheduler which can consider all requests at every moment in time and has no computational overhead when choosing the requests to receive service can provide an optimal schedule under all circumstances.

2.2.2.1. When to Schedule

A scheduler may be implemented to make all of the decisions prior to the actual running of the system. This is called *static* scheduling. It requires that the scheduler have complete and prior knowledge of the task set, and that the scheduler makes assumptions about the environment, either dismissing as negligible or compensating for any changes which may occur during the run of the system. The result is a list of (*time, task, duration*)

triples for each resource which completely determines when and for how long a task may be serviced. If all of the tasks can be known *a priori*, and the assumption on the environment can be made with confidence, then static scheduling provides a scheduling solution with very low run-time costs and a high degree of predictability. This is especially attractive for hard real-time systems where variability in scheduling overhead can cause tasks to miss timing constraints.

Dynamic schedulers examine the task set and the state of the system while the system is running. At each decision time the scheduler can more accurately account for the changes occurring within the environment, and can consequently make the decisions based on more available information. In particular, the scheduler does not need to know the entire task set prior to run-time; transient load conditions are more easily handled than in the static scheduler. However, the overhead incurred for the calculation of the schedule at run-time becomes an important component of the service time for the tasks. As the scheduling algorithm becomes more complicated, the time it takes to make the scheduling decision increases. For example, choosing a task from a task set on a first come, first served basis is a trivial scheduling algorithm, and thus requires very little overhead to implement. Choosing tasks from a task set such that all of the tasks will meet their individual deadlines, however, is computationally more difficult. It is a grave error to have a scheduling algorithm which requires an excessive amount of computation to schedule tasks with time constraints. Consequently, many scheduling algorithms use a set of simple heuristics to approximate optimal scheduling algorithms in an effort to lower run-time scheduling overhead.

2.2.2.2. How Often to Schedule

Another aspect of scheduling is how often to invoke the scheduler so that the current set of tasks being serviced may change. The period of time during which a decision is made

and a set of tasks is processed is called a *decision epoch*. The duration of decision epochs is independent of the scheduling algorithm; however, certain scheduling algorithms may imply certain decision frequencies. For example, scheduling by first come, first served would imply that decisions are most appropriately made only when the current task has been completely serviced or when a newly arriving task finds the server empty. A scheduler implementing a first come, first served algorithm may make decisions continuously, but this would be unnecessary.

Ruschitzka and Fabry [RUSC77] identify four *decision modes* which characterize when a new decision may be made. Each decision mode is progressively more general than the last, and each previous mode is a proper subset of the next. The four modes are nonpreemptive, quantum-oriented, preemptive, and processor sharing.

The *nonpreemptive* mode allows decisions only after the task currently being serviced is complete or when an arriving task finds the server empty. It is interesting to note that the scheduling algorithm may do its best to provide service to the highest priority task when the decision epoch arrives, but there is nothing that the algorithm can do to provide service to a newly arriving higher priority task if the server is currently busy until the current task has completed. (*Priority inversion* [GOOD88] is the term used to describe a system which allows high priority tasks to wait while lower priority tasks are being serviced.)

The *quantum-oriented* decision mode makes decisions every fixed quantum of time unless a task completes or a newly arriving task finds the server empty before the next decision epoch. If the quantum is infinitely long, then the quantum-oriented mode reverts to the nonpreemptive mode.

If decisions are also made when any new task arrives, regardless of whether the server is busy, then the decision mode is *preemptive*. If the relative priority of the tasks does

not change over time, then this decision mode avoids priority inversion. However, if tasks' priorities can change at different rates, then priority inversion is still possible since the change of priority does not itself invoke the scheduler.

Allowing the quantum size to become zero, and hence ensure that decisions are made continuously, defines the decision mode called *processor sharing*. Processor sharing is the most general decision mode, encompassing each of the other decision modes. The instant one task becomes higher priority than a task being serviced, the new higher priority task takes its place in the server. This mode is called processor sharing since the group of highest priority tasks, while they all remain highest priority, will effectively share the server (processor) even if there are more tasks to service than the server can handle; each task is receiving service concurrently with all other equally highest priority tasks.

2.2.2.3. Complexity of Scheduling

When a problem is stated and some solution to the problem is sought, the step-by-step procedure to solving the problem is called an algorithm. There are perhaps infinitely many algorithms for any one problem specification; what ranks the algorithms is their relative efficiency. Yet, not all problems have known algorithms which are efficient, and it is a very famous open problem if an efficient algorithm will *ever* be found for a certain class of problems. The theory of NP-completeness provides a straightforward technique for proving that finding an efficient algorithm for one problem is essentially equivalent to finding an efficient algorithm for a very large set of problems widely recognized to be difficult.

Efficiency is largely dependent on the situation at hand, but in the theory of problem complexity an efficient algorithm is one in which the running time or the space required can be bounded by a polynomial on the input size. Inefficient algorithms have solutions which

are bounded by an exponential on the input size, and are merely variations on exhaustive search. Polynomial time algorithms, on the other hand, are made possible only through some deeper insight into the structure of the problem [GARE79]. Undecidable problems are ones for which no algorithm will ever exist, even an exponential one. NP-complete problems are problems for which no polynomial time algorithm yet exists, and for which the discovery of one would imply that many other similarly difficult problems also have polynomial time algorithms.

Almost every interesting problem in scheduling theory is NP-complete. A few famous examples, like the rate monotonic algorithm, are exceptions. Usually a scheduling problem must be so restricted in order to find an efficient algorithm that its solution is virtually useless in a real-world sense. The following basic scheduling problem illustrates this point.

The Sequencing on One Processor problem [GARE79] is posed as a set T of tasks where each task in T has the following characteristics: a length, an arrival time, and a deadline. Recall that the length and the arrival time are inherent attributes of a task, and that the specification of a point in time like a deadline is a constraint. The question is whether there is a schedule that meets all of the deadlines. This is a decision problem; if it can be solved in polynomial time then the construction problem is guaranteed to be solvable in polynomial time as well. Unfortunately there is no such known algorithm, and this problem is NP-complete. In fact, the problem instance must be restricted in at least one of three ways to solve it in polynomial time: either all tasks have length of one unit, or the tasks may be preempted during service, or all tasks arrive at the same time, namely time 0. This problem is perhaps the simplest of the scheduling problems in the sense that all tasks are known *a priori* and there is no ordering placed on the tasks based on which are more important.

2.2.3. Resources

A resource is an identifiable entity needed for the accomplishment of some task. Certainly such an entity needed is the processor or set of processors which will perform the task. The resource set, however, includes anything required such that the task, or any task in the task set, may be performed.

The task of sweeping out the basement as part of the goal of spring cleaning requires three resources: the broom, the dustpan, and the person. The person is the “processor” since it is the resource that actually performs the task. The broom and the dustpan are required to do the work, but they do not perform the work.

Several more points may be made by continuing this analogy. There may be a set of two or more processors rather than just one. More than one person may sweep out the basement in parallel, or one may work until noon and another take over. In the division of household labor, each member of the family may have specific jobs, the sweeping out of the basement being assigned to one member in particular. In a more enlightened view, any one of the family may do this task, and the choice of who does it is simply a matter of who is available. The point is that the processor set may be specialized or general, and tasks may have the property that they may be done in parallel by multiple processors or serially by a set of processors.

Two or more tasks may require the same resources at the same time. One task is chosen over the others to be performed by supplying that task with all of the resources it needs until it is done. Then the other tasks may vie for the resources. If both the basement and the garage need sweeping, but there is only one person, one broom and one dustpan, then one area is chosen to be swept first while the other waits. Perhaps only one of the resources is required by two contending tasks. If the resource is sharable up to a certain

capacity, then there is no contention until the capacity is exceeded. If the capacity is one, then the resource cannot be shared; only one person can use the broom at a time.

One task may claim all of the resources it requires and not relinquish them until the task is completed. However, if resource constraints are examined to determine how the resources are needed, and when they are required, then the resource set can be “scheduled” onto the tasks requiring them. Certainly someone else can use the dustpan until the basement dust is in neat piles ready for collection.

3 Survey of Scheduling Techniques

Rate monotonic theory provides rich analytical results for scheduling algorithms designed to meet deadline for a periodic task set. Included in these results are simple tests for feasibility of a task set based on its aggregate utilization of the processor. Much work in real-time scheduling centers around this algorithm, both in exploiting the results and in seeking solutions to the various deficiencies to the basic algorithm. We present a brief overview, first because rate monotonic theory is pervasive within scheduling theory, and second because we revisit some of these results in our analysis section.

The importance abstraction has a function-based scheduling framework. We survey seminal work in using functions to aid in scheduling decisions. Typically the functions return a value which represents some aspect of the task's worth, such as *priority* and *value* (the importance functions return values which represent *importance* of a task). Functions provide flexibility in expressing this task's worth over the time that the task is active. The importance abstraction extends this work by also using the functional representation of the task's importance to perform analysis on the nature of the schedules produced.

3.1. Rate Monotonic Theory

In 1973, Liu and Layland introduced *rate monotonic* scheduling theory [LIU73] as a method for scheduling many periodic tasks on a single processor such that the scheduling algorithm used to do this was optimal. Dhall and Liu extended this work into the multiprocessor environment in [DHAL78]. The following discussion is drawn largely from Sha and Goodenough [SHA90], who present an excellent overview of the theory and recent

extensions which include aperiodic and sporadic tasks, as well as non-independent task relationships.

Rate monotonic scheduling theory in essence ensures that as long as the processor utilization of all tasks lies below a certain bound and appropriate scheduling algorithms are used, all tasks will meet their deadlines without the system designer knowing exactly when any given task will be running. Given a set of independent, preemptable periodic tasks, the rate monotonic scheduling algorithm gives each task a fixed priority by assigning higher priorities to tasks with shorter periods. All tasks are preemptable in that whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. A task set is said to be *schedulable* if all its deadlines are met (i.e., all periodic tasks finish execution before the end of their periods).

Any independent periodic task set may be subjected to a test to determine if that task set is schedulable regardless of when each individual task is started. Let C_i be the execution time for task τ_i , T_i be the period for task τ_i and n be the cardinality of the task set. For a statically assigned priority algorithm (the rate-monotonic algorithm, where priority is defined as the inverse of the task period), the following must be true for the task set T to be feasible:

$$\frac{C_1}{T_1} + \dots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1) \quad \text{Eq 3.1}$$

If the utilization (computation time over period) of all of the tasks is below the bound prescribed, then the tasks are guaranteed to be schedulable if they are scheduled according to the rate monotonic algorithm. This bound converges to $\ln 2$, or about 70% utilization of processor capacity, as the number of tasks goes to infinity. This algorithm is shown to be

optimal among all fixed priority scheduling algorithms with respect to meeting deadlines of periodic task sets.

Liu and Layland also show that a variation on this, the deadline driven scheduling algorithm, can provide 100% processor utilization on task sets where the priority can be assigned dynamically. This variation is also optimal with respect to meeting deadlines among all algorithms where priority assignment may be made during the run of the system. In the deadline driven scheduling algorithm, the priorities are assigned according to which task's deadline is nearest rather than by period length.

In [SHA90], Sha and Goodenough discuss the use of rate monotonic theory for real-time scheduling in the Ada tasking model. However, there are certain drawbacks to the unabridged rate monotonic scheduling policy, namely that (1) a task's period is not inherently related to how critical the task is to the system, (2) synchronization of a lower priority task can indefinitely delay a higher priority task when tasks share data or communications, and (3) there is no clear way to treat aperiodic tasks in this policy designed for periodic task sets. *Period transformation*, *priority inheritance* and *priority ceiling protocols*, and the *deferrable server protocol* address each of these issues respectively.

3.1.1. Period Transformation

One major problem with rate monotonic scheduling is that the priorities are assigned according to the period of the task rather than according to its criticality to the system. When all tasks can be scheduled without fear of some task exceeding its execution time, then no criticality measure need be placed on the tasks. However, execution times are necessarily stochastic, and scheduling is usually done with worst-case estimates which may be significantly longer than the average execution time. When tasks exceed their estimated

execution times, a transient overload occurs which may cause some tasks to miss their deadlines. Yet if tasks are prioritized according to their periods, some critical tasks may miss their deadlines if their periods are too long.

The *period transformation* technique [SHA86] is used to ensure that highly critical tasks are treated with higher priorities even if they have longer periods. The priority of a critical task can be raised by treating it like a task with a shorter period. The technique is to divide both the period and the worst-case execution time by some constant. Now the task looks like its period is shorter, but the total utilization is not affected. The task's execution is suspended after each execution time until its next "period" arrives.

This technique is designed, therefore, to decouple the criticality of a task from its period, while maintaining the benefits of the rate monotonic algorithm. If the tasks can be partitioned into critical and non-critical task sets, where the critical tasks are defined to be those which must receive service during a transient overload condition, then a period transformation can be applied to the critical tasks with the longest periods. Without period transformation, the longest period tasks would be subject to missed deadlines since they have low priority. The set of critical tasks, therefore, are period transformed until the longest period of the critical set is shorter than the shortest period of the non-critical set. Now all non-critical tasks will miss their deadlines before the first critical task will.

3.1.2. Priority Inheritance and Priority Ceiling Protocols

Priority inversion is defined as the phenomena of a task of higher priority being forced to wait on the completion of a task of lower priority. In certain cases the priority inversion can be unbounded. The *priority inheritance protocol* attempts to limit the amount of priority inversion by allowing a server task to inherit the priority of its highest priority *client* [SHA87]. A central theorem in priority inheritance specifies a sufficient worst-case

condition that characterizes the rate-monotonic schedulability of a given set of periodic tasks. The *priority ceiling protocol* minimizes the blocking of high priority tasks by guaranteeing that such a task will be blocked by at most one critical region of any lower priority task [GOOD88, LOCK88].

The *priority ceiling* of a critical region is defined to be the highest priority of all the tasks that may lock on that region. When a new task attempts to secure that region, it will be suspended unless its priority is higher than the priority ceilings of all regions currently locked by tasks other than this one. If the task is suspended, then the task that holds the lock on the region with the highest priority ceiling is said to be blocking this task, and hence inherits the priority of this task.

3.1.3. Deferrable Server

Current systems with hard real-time periodic tasks handle aperiodic tasks either by servicing them in background or by polling periodically for aperiodic tasks. If an aperiodic task is serviced in the background, it must wait until all periodic tasks have been serviced. If an aperiodic task arrives just after the polling time, the task must wait until the next polling time. In both of these cases the response time for aperiodic tasks suffers unnecessarily due to naive treatment of the task set.

The Deferrable Server algorithm [LEHO87, SPRU88] is designed to provide aperiodic tasks with a low response time without jeopardizing the periodic tasks. A new periodic task with highest priority is created to service the aperiodic tasks such that all tasks, including this aperiodic server, are guaranteed to meet their deadlines by the rate monotonic theory. Any aperiodic tasks are serviced at this highest priority as soon as they arrive as long as there is computation time left for this aperiodic server. When there are no aperiodic tasks, the computation time of the server is deferred until one arrives. The

computation time of this server is replenished each period. Thus, the response time for aperiodic tasks is minimized while the schedulability of the hard real-time periodic tasks is maintained.

3.2. Survey of Function-Based Scheduling Techniques

Bernstein and Sharp [BERN71] recognized that service given to a class of tasks could be controlled using a function such that various service profiles could be effected as the tasks grew older. Priority in this scheme is related to the difference between the function's projected service and the service actually attained. Ruschitzka and Fabry [RUSC77] used functions to describe the priority of a task directly. Within this model, various scheduling algorithms could be emulated by using an appropriate priority function. Jensen *et al.* [JENS85] used a function to profile a task's value to the system for completing at that time. The value functions did not directly drive the scheduling decisions in Jensen's model; rather they were used mostly as a metric for comparing the performance of other scheduling algorithms.

Below we survey these three techniques for using functions for making scheduling decisions.

3.2.1. Policy Functions

Bernstein and Sharp, in [BERN71], theorized that a scheduling algorithm that keeps track of the resource count of each task and orders the tasks according to how far a task is from the expected resource count at that task's age would provide the specified level of service for each task. They defined a *policy function* as a function which characterizes a class of tasks by specifying the amount of service those tasks should receive as a function of time. Each class of tasks within a system is characterized by a function that specifies the amount of service a task within a class should receive as a function of time. The shape of

the policy function controls the type of service received by that class of tasks. In this system the notion of *priority* corresponds to the difference between the service promised to the task by the policy function and the service actually received by the task. Consequently, the priority of a task changes at a constant rate while awaiting service and at another rate determined by the shape of the policy function while receiving service. The tasks which are most delinquent are therefore the highest priority tasks.

Since the shape of the policy function ultimately determines a task's priority, different scheduling policies may be implemented using the same basic scheduling algorithm by simply changing the policy functions of the tasks. Bernstein and Sharp consider piecewise functions as the policy functions for various classes of tasks. One such function proposed uses a curved portion in a region starting with the task's activation to give a task a limited amount of rapid service, followed by a linear portion for a more constant rate of service.

Ruschitzka and Fabry [RUSC77] extend the notion using functions for scheduling by introducing the *universal scheduling system* (USS) as a generalized scheduling framework to support arbitrary scheduling algorithms. There are three parts to the specification of a scheduling algorithm within the USS: the *decision mode*, the *priority function*, and an *arbitration rule*. The *decision mode* specifies how often scheduling decisions are made. The *priority function* is an arbitrary function of task and system parameters that determine the task's priority at the time of evaluation. The *arbitration rule* is used to break ties between tasks of the same priority. Ruschitzka and Fabry suggest that a scheduling algorithm can be emulated by appropriately specifying the decision mode, priority function, and arbitration rule such that the USS will make exactly the same scheduling decision at exactly the same time as would the algorithm.

Four decision modes are identified; each decision mode is progressively more general than the last, and each previous mode is a proper subset of the next. The four modes are *nonpreemptive*, *quantum-oriented*, *preemptive*, and *processor sharing*. The *nonpreemptive* mode allows decisions only after the task currently being serviced is complete or when an arriving task finds the server empty. The *quantum-oriented* decision mode makes decisions every fixed quantum of time unless a task completes or a newly arriving task finds the server empty before the time of the next decision. If the quantum is infinitely long, then the quantum-oriented mode reverts to the nonpreemptive mode. If decisions are also made when any new task arrives, regardless of whether the server is busy, then the decision mode is *preemptive*. Allowing the quantum size to become zero, and hence ensuring that decisions are made continuously, defines the decision mode called *processor sharing*. Processor sharing is the most general decision mode, encompassing each of the other decision modes—the instant one task becomes higher priority than a task being serviced, the new higher priority task takes its place in the server. This mode is called processor sharing since the group of highest priority tasks, while they all remain highest priority, will effectively share the server (processor) even if there are more tasks to service than the server can handle; each task is receiving service concurrently with all other equally highest priority tasks.

The priority function is an arbitrary function of task and system parameters. The priority of a given task is defined as the value of the priority function applied to the current values of the parameters. Ruschitzka and Fabry suggest that these parameters may include the memory requirements, the attained service time, the total service time, external priorities, timeliness, and system load. A priority function is defined for a scheduling algorithm such that, when the algorithm chooses a particular task for service, the priority function applied to that task will return the highest priority among all tasks.

The arbitration rule specifies how to resolve conflicts among jobs with equal highest priority. Ruschitzka and Fabry note that the advantage to specifying the arbitration rule, as well as the decision mode, is that this specification simplifies the priority function. Neither the decision mode nor arbitration rule is necessary since the priority function can be made to implement the various decision modes and arbitration rules.

Ruschitzka and Fabry continue by noting that a large class of scheduling algorithms can be defined by a priority function of only three arguments: the task's attained processing time, the current time, and the task's processing time requirement. Furthermore, an algorithm is called *time-invariant* if the difference between the priorities of any two tasks does not change as long as neither task receives service. Included in this class of algorithms is the policy-driven scheduling algorithms of [BERN71]. Ruschitzka and Fabry extend the work of Bernstein and Sharp by noting that, in general, time-invariant priorities are characterized by a policy function of an arbitrary number of arguments.

3.2.2. Time-Driven Scheduling

The primary notion in *time-driven scheduling* [JENS85, LOCK86, TOKU87, WEND88] is that the distinguishing characteristic of a real-time system is the concept that the value a task has to the system is dependent upon when that task completes. Each task has associated with it a *value function* $V_i(t)$ which returns the value to the system for completing task i at time t . The optimal schedule, therefore, arranges the tasks such that they complete at times which maximize the sum of their values to the system. Jensen *et al.* use this value sum as a metric for comparing the effectiveness of conventional scheduling algorithms.

It was observed in [JENS85] that task scheduling in real-time systems almost always uses some simple algorithm, like fixed priority, first in first out, or round robin. Often the

time-criticalness of the tasks is represented by a point in time called a *deadline*. Attempting to meet deadlines via fixed priority scheduling algorithms leads to rounds of testing and adjustment of the priorities, and results in a particularly fragile system. Assigning higher priorities to important tasks does not reflect the time-constrained characteristic of the tasks. Assigning higher priorities to tasks with nearer deadlines does not reflect the differences in importance among tasks.

The tasks with associated value functions do not employ the explicit use of a deadline. Rather, the existence and importance of deadlines depend on the nature of the value function. A *critical time* for a task is represented by a discontinuity in the task's value function. In this way the concept of hard and soft deadlines is replaced by a step function whose shape reflects the urgency of completing before a certain time.

Jensen *et al.* create an environment in which various scheduling algorithms can be evaluated through the use of a simulator. For tractability reasons the value functions are limited to having two parts, one prior to and one after the critical time, each consisting of the following five-constant form:

$$V_i(t) = K_1 + K_2t - K_3t^2 + K_4e^{-K_5t} \quad \text{Eq 3.2}$$

This form allows value functions which are constant, linear, quadratic, exponential, or a linear combination of any of these.

Jensen *et al.* report the simulation of several classical algorithms on a task set using various shapes for the value functions. These algorithms included shortest estimated completion time first, earliest deadline first, least slack time first, first in first out, random order, and a fixed priority where the priority was equal to the highest value that the value function could attain. Two experimental algorithms were also evaluated. The first used a *value density* (value at the projected completion time over the task length). The second

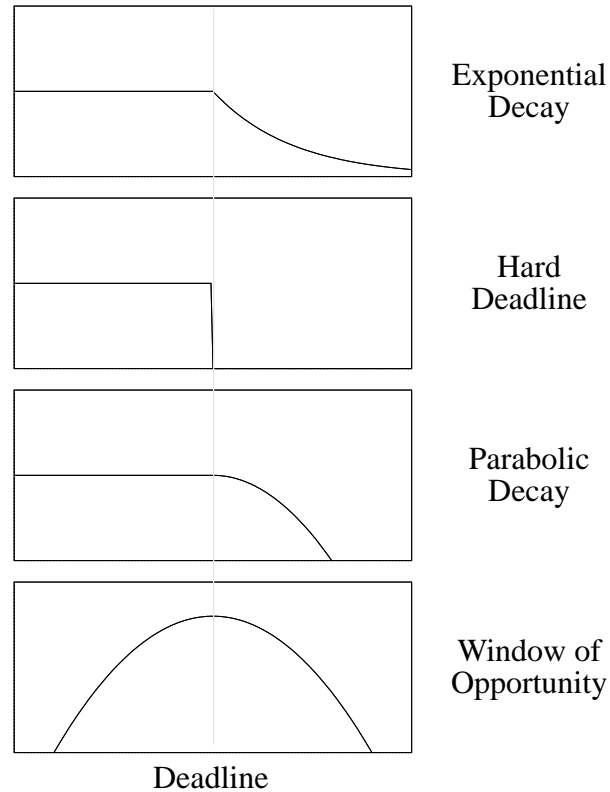


Figure 3.1 —Value Function Shapes

algorithm used a nearest deadline first algorithm, shedding the tasks with the lowest value densities during overload. Four shapes of value functions, shown in Figure 3.1, were used in separate executions to compute the total value generated by each of the scheduling algorithms. The results showed that the second experimental algorithm outperformed all others tested; this algorithm, called the Best-Effort Heuristic, is the focus of Locke's work in [LOCK86].

The implementation issues of time-driven scheduling, especially using the best-effort heuristic, are explored in [TOKU87] and [WEND88]. It was concluded that the high computational overhead of best-effort time-driven scheduling made implementation

impractical on a uniprocessor system. More reasonable performance could be gained by using a dedicated processor for only scheduling decisions.

4 Importance Abstraction

The *importance abstraction* is a framework within which we can describe scheduling policies by focusing on the importances of the tasks within a system. Every system has a goal and the tasks within the system are processed with the intent of meeting the system goal. A task within the system is viewed as “important” to the system *vis-a-vis* how that task can contribute to accomplishing the system goal. As the system progresses and its state changes, various tasks become more or less important to the system. The importance abstraction is a framework for expressing those conditions under which tasks within a system become important to the system.

The importance abstraction includes within its framework sets of importance functions that describe the tasks within a system, and a scheduler that uses the importance functions to determine which tasks should receive service. By using this abstraction to consider scheduling problems, we shift the emphasis from the analysis of the scheduling algorithm to the analysis of a set of functions.

4.1. System Model

We define a *system* as any entity with the following components: a set of inputs into and a set of outputs from this entity, a processor, and a set of tasks to be processed, as shown in Figure 4.1. The system “communicates” with the world outside of it through its inputs and outputs. The system reacts to inputs by changing the system state. The outputs from the system reflect these and other state changes, and are the means by which the system may affect the outside world. Since a system is designed to accomplish some goal, it is only

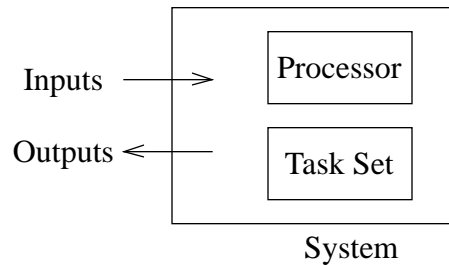


Figure 4.1 — The System Model

through these outputs that the degree to which the goal is accomplished can be gauged by an outside observer. The system makes choices about when and what tasks to process such that the system can move toward accomplishing its goal.

A network interface unit (NIU) is an example of a system that is itself a component within a larger system. The NIU attaches the host system to the local area network (LAN). An NIU is typically a front-end processor that attaches to both the host's backplane bus and the physical media of the LAN, as shown in Figure 4.2. Note that the NIU alone forms a system, taking inputs from both the backplane bus and the LAN, and placing outputs to both as well. The entire network, including the NIU of each attached host, is also considered a system. Within a distributed system, as shown in Figure 4.3, the hosts and the network are each subsystems.

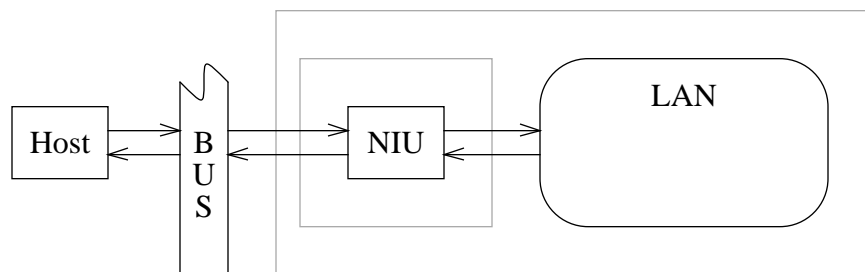


Figure 4.2 — Network Interface Unit and Network "Systems"

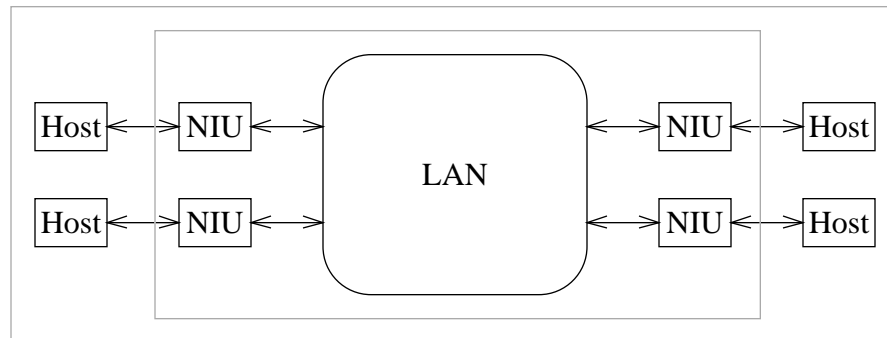


Figure 4.3 — Communication “Subsystem” within a Distributed “System”

Since the system is designed to accomplish some goal, each task within the system somehow contributes to accomplishing the system’s goal under system conditions and task attributes which are specific to that task. As these particular conditions arise within the system, the task becomes “important” to the system. At any particular point in time there exists a ranking of the tasks according to how important each task is to accomplishing the system goal. If, at that moment, a “most important” task exists, then the system could best move toward accomplishing the goal by performing that task at that moment. As conditions change, the importances of the tasks may change, and a new task may become “most important.”

Just as the state of the system changes with time as work is performed on the tasks and inputs are received, the composition of the task set also changes with time. At the system start time, when the system begins at some initial state, there exists an initial task set. As work is done on tasks within this set, the attributes of the tasks within the set change—in particular, the “work remaining” attribute of the task currently receiving service is decreasing. The membership of the task set also changes over time. Some tasks complete and are removed from the task set. Other tasks may simply outlive their usefulness and be removed from the task set. Still others arrive and join the task set.

Consequently, we can think of a snapshot of the task set as being a “state,” and the act of scheduling and servicing the tasks within the task set moves the task set from one state to another.

4.2. Importance Functions

If the importance of a task could be quantified at every point in time, it could be expressed as a function over time to profile a task’s importance to the system. Since the importance of a task depends upon the conditions of the system and the attributes of that task, these conditions and attributes must be the parameters to the function. If we can identify each possible task in the system, and under what conditions that task will become important to the system, we could associate with each task a function that reflects the task’s importance.

Consider a task set where each task in the set has associated with it a function, called an *importance function*, which includes all of the conditions and circumstances under which the task is important to the system. Let the function return a value that ranks that task among all other active tasks competing for a system resource according to how important it is that the task be given that resource at that moment in time. The importance abstraction uses sets of importance functions as a representation of the task set with respect to how the tasks within the task set should be ordered for service in order to accomplish some system goal.

4.2.1. Sets of Importance Functions

Assume that, for a given system, there is an importance function associated with each task in the task set. Let I_T be a set of importance functions for the task set T . The set I_T embodies those attributes and constraints of the tasks in T and system parameters considered important to accomplish a particular goal; therefore, we can consider the set of

importance functions as representatives of the tasks, and use these functions when asking questions regarding scheduling.

We can consider a *universe* of all sets of importance functions for the task set T , $U = \{I_T\}$, where each member of the universe imposes a schedule that will meet some particular system goal. Not every member of the universe of importance function sets will meet the same system goal; rather, it is the system goal that partitions the universe into two sets: those sets of importance functions which impose schedules which meet the system goal, and those importance function sets for which the system goal cannot be guaranteed. Thus, given a goal G , the universe U can be partitioned into $U_G = \{I_T \ni G \text{ is satisfied}\}$ and $U_{\bar{G}} = \{I_T \ni G \text{ is not satisfied}\}$.

4.2.2. The Defining Property of an Importance Function Set

Given a task set T within a particular system, and a goal G for that system, we seek the property P_G which defines the set U_G . We call this property a *defining property*. Since the goal G partitions the universe of importance function sets U into U_G and $U_{\bar{G}}$, the defining property P_G reflects those qualities of the sets of U_G that (1) make each set a member of U_G , and (2) distinguishes each set from sets in $U_{\bar{G}}$.

Since each importance function set in U_G imposes a schedule that meets the goal G , the schedules are termed *equivalent*. The importance function sets that impose these schedules are therefore members of an equivalence class. By discovering the defining property of an importance function set which causes that set to belong to U_G , we can determine if a given importance function set is a member of this equivalence class.

If the defining property holds for every member of the equivalence class and no others, that defining property represents the *necessary* and *sufficient* conditions on the set of importance functions for inclusion in the equivalence class. If a property holds for a

subset of the equivalence class and no others, then the property is a sufficient condition for inclusion in the equivalence class, but not a necessary condition.

4.3. The Scheduler

When a set of importance functions has been associated with a task set, the tasks within that task set are scheduled according to the values of the importance functions. By definition, the *optimal schedule* is achieved when the scheduler chooses the most important task (task with the highest valued importance function) at every point in time. Thus, at every point in time the scheduler must evaluate the function, $M:I_T \rightarrow T$, which takes the set of importance functions and returns a task. Without loss of generality assume that the tasks in T are indexed, in no particular order, so that a task is identified by its index. The function M evaluates each importance function in the set I_T and returns the task $i \in T$ whose importance function has the maximum value at that point in time. If, at some point in time, the scheduler finds that two or more tasks are most important simultaneously, the scheduler will arbitrarily choose one of those tasks as the task to receive service, and will continue to allow that task to receive service until some other task becomes most important.

We can express the actions of the scheduler with some mathematical constructs. The boolean relation $(M(I_T) = i)$ returns the value 1 if the most important task at the time of evaluation is the task i , and the value 0 otherwise.¹ By using this boolean relation as a function of time, we can ask how long a specific task has been most important over a certain period. Let the value $w_i|_{t_1}^{t_2}$ represent the amount of work applied to the task i from time t_1 to time t_2 (Appendix A describes in detail the properties of this construct). The equation

¹. The convention of using a boolean expression within a set of parentheses to denote a function that returns 1 if the boolean expression is true and 0 if it is false is used in Graham, Knuth, and Patashnik's book **Concrete Mathematics**, Addison-Wesley, Reading, MA (1988); they attribute the convention to Iverson in the programming language APL.

$$w_i \Big|_{t_1}^{t_2} = \int_{t_1}^{t_2} (M(I_T) = i) dt \quad \text{Eq 4.1}$$

shows the relationship between the importance functions and the amount of work done to a particular task. This equation states that the amount of work received by task i over the period from t_1 to t_2 is equal to the amount of time that the task i has been most important from time t_1 to t_2 . Note that if there are two or more tasks with equal importance at time t , the function M chooses one of these tasks arbitrarily.

5 Expressiveness of the Importance Abstraction

The importance abstraction is a novel framework for expressing scheduling policies. The actual scheduling algorithm is simple and universal: the scheduler chooses the most important task at every point in time. The most important task is found by evaluating the set of importance functions that profile the importance of each task. By using a function to profile the task importance, the scheduler considers the conditions under which an individual task becomes important without the scheduler or the scheduling algorithm maintaining the state of these conditions for each task. This shifts the description of the conditions for scheduling from the scheduler to the agents for the tasks. Consequently, complex scheduling policies (e.g., those with many conditions for determining which task is to be scheduled at any particular time) are easily expressed in the importance abstraction while the same policies may prove difficult and cumbersome to express as algorithms.

Traditional scheduling policies are typically based on one or only a few task attributes. Consequently, the algorithms that implement these policies use these attributes when determining the schedule. For example, the nearest deadline first scheduling policy considers only task deadlines; the algorithm dictates that the task with the nearest deadline is always scheduled for service. These scheduling policies can be also be implemented within the importance abstraction by devising importance functions based on the task attributed considered by the algorithms. The importance functions emulate the algorithm if a task becomes most important exactly when the task would be scheduled by the algorithm. In this chapter we give several examples of traditional scheduling policies and show

importance function sets that implement the policies by emulating the algorithms associated with the policies.

In addition to its ability to emulate the traditional scheduling algorithms, the importance abstraction provides the framework for implementing novel scheduling policies not intuitive when using algorithms. We offer an example of such a novel scheduling policy, and show how the policy can be expressed easily with importance functions.

5.1. Emulation of Traditional Scheduling Policies

An interesting and important aspect of the importance abstraction is the ability to *emulate* traditional scheduling policies within its framework. The importance abstraction is said to emulate an arbitrary scheduling policy in that it makes exactly the same scheduling decisions at exactly the same time.¹ In the importance abstraction the act of scheduling always remains the same: choose the most important task to perform at each decision point; the various scheduling policies are actually implemented by defining appropriate importance functions. The importance functions must be defined in such a way that a task becomes most important at precisely the same instant as the conventional scheduling policy would have chosen it.

First Come, First Served

In the First Come, First Served (FCFS) scheduling policy, the scheduler chooses the oldest of the active tasks to serve. That is, it finds the $\min(a_i)$, where a_i is the arrival time for the task i . To emulate this policy, we define importance functions for each task such that the task's importance is monotonically increasing with its age. There is an infinite class of importance functions for which this is true; we offer the most obvious:

¹The concept of creating a framework within which to emulate other scheduling policies was first presented by Ruschitzka and Fabry in [RUSC77] with the Universal Scheduling System.

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ t - a_i, & \text{if } (t \geq a_i) \end{cases} \quad \text{Eq 5.1}$$

Consider four tasks with arrival times as follows:

$$\text{task 1 : } a_1 = 0$$

$$\text{task 2 : } a_2 = 2$$

$$\text{task 3 : } a_3 = 3$$

$$\text{task 4 : } a_4 = 4$$

Eq 5.2

Let each task be associated with an importance function as defined above. Assume that each task requires 3 time units to finish. Figure 5.1 shows the graph of importance value versus time as each task gets older. Notice at time 5 there are 3 active tasks. The scheduler will always choose $M(I_7)$ which, for time 5, is $M(I_2(5)=3, I_3(5)=2, I_4(5)=1) = 2$, so task 2 is chosen.

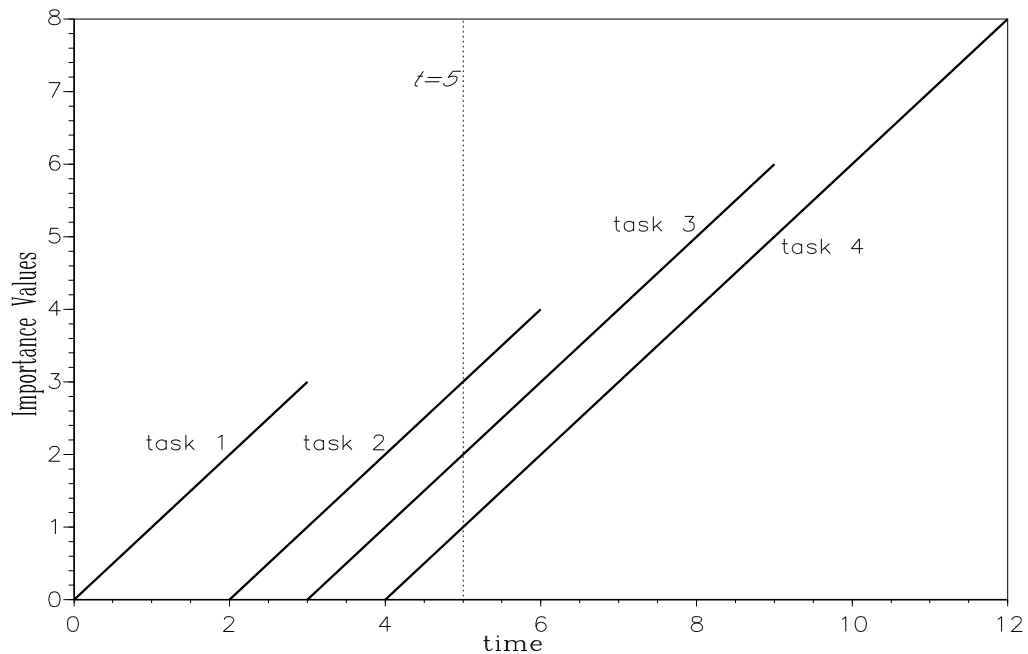


Figure 5.1 — Importance Function Values for FCFS Policy

Nearest Deadline First

Nearest Deadline First (NDF) is quite similar to FCFS in that we need a monotonically increasing function based on the nearness of a point in time; while FCFS uses arrival times, NDF uses deadlines as the basis for the importance functions. If the scheduler could chose the minimum of some set of values rather than the maximum, we could use the quantity $d_i - t$, where d_i is the task's deadline. Since the scheduler always chooses the most important, then we need a function which is monotonically increasing: the reciprocal of $d_i - t$ is one such function.

Consider the following importance function definition for each task:

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ (d_i - t)^{-1}, & \text{if } (d_i > t \geq a_i) \\ 0, & \text{if } (d_i \leq t) \end{cases} \quad \text{Eq 5.3}$$

Further, consider a task that arrives at time 3 and has a deadline of time 10. Figure 5.2 shows the graph of the importance values over time for this task. Notice that there are two discontinuities, one at the moment that the task becomes active and one at the moment it misses its deadline. Also notice that the task becomes infinitely important just as the deadline is reached.

Priority Driven

Tasks ranked by static priority are easily emulated by the importance abstraction by defining the importance functions as constant functions reflecting the relative ranking of the tasks. Any constant values will work as long as, for any two tasks i and j with priorities p_i and p_j , priorities equal to or greater than 0, the following always holds:

$$p_i > p_j \Rightarrow I_i(t) > I_j(t) \quad \text{Eq 5.4}$$

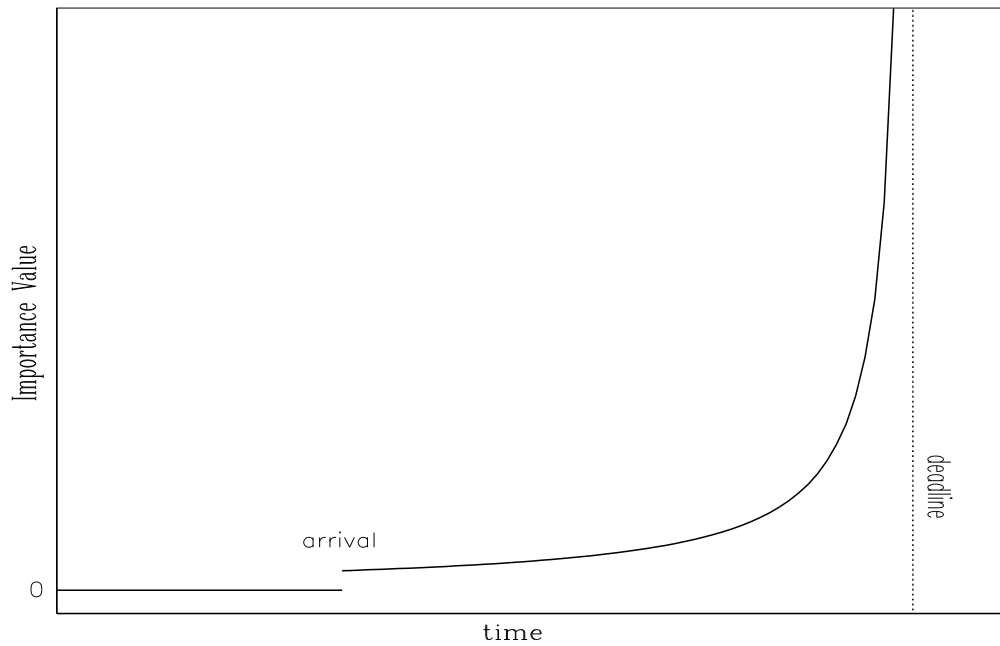


Figure 5.2 — Importance Function for an Nearest Deadline Task

An example of such a function is:

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ p_i, & \text{if } (t \geq a_i) \end{cases} \quad \text{Eq 5.5}$$

Rate Monotonic

Rate monotonic theory applies to those tasks which are periodic in nature; that is, an instantiation of the task is activated exactly once per time period. The priority of a task is statically assigned to be the inverse of that task's period, ρ_i . An importance function set that emulates this is:

$$\forall (i \in T), \quad I_i(t) = \begin{cases} \frac{1}{\rho_i}, & \text{if } (a_i \leq t \leq a_i + \rho_i) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 5.6}$$

Least Slack Time

The least slack time policy chooses the task that has the least difference between its projected finish time and its deadline. Previously we have considered a deadline as the time by which the task must start. Here the deadline is the time by which the task must finish. Slack time for task i is defined as $slack = d_i - l_i - t$, where l_i is the task length. The importance functions are easily given by replacing the d_i quantity in the deadline driven functions by the quantity $d_i - l_i$, thus:

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ (d_i - l_i - t)^{-1}, & \text{if } (d_i - l_i > t \geq a_i) \\ 0, & \text{if } (d_i - l_i \leq t) \end{cases} \quad \text{Eq 5.7}$$

Round Robin

In round robin scheduling each of the n tasks is given an equal share of the processor in turn until all n tasks have received a share. Although the order of service is arbitrary, once established, the order is maintained for subsequent cycles through the task set until one or more tasks complete or one or more tasks join. In general the share of the processor, or *time slice*, may either be fixed, and hence the period of the cycle varies with the size of the task set, or the period itself is fixed, and hence the time slices vary according to the set size.

Consider a set of importance functions which take the form

$$I_i(t) = \sin(bt + c_i) + d \quad \text{Eq 5.8}$$

where b determines the period, c_i is the offset for task i , and d , if greater than 1, shifts the function so that all values are positive. Let $d = 1$ and, for n tasks numbered 0 through $n-1$, let $c_i = (2\pi i)/n$. Figure 5.3 shows graphs of importance functions for four tasks. It is easily seen that each task is “most important” for precisely $1/n^{\text{th}}$ of the period, and that the order of service remains fixed.

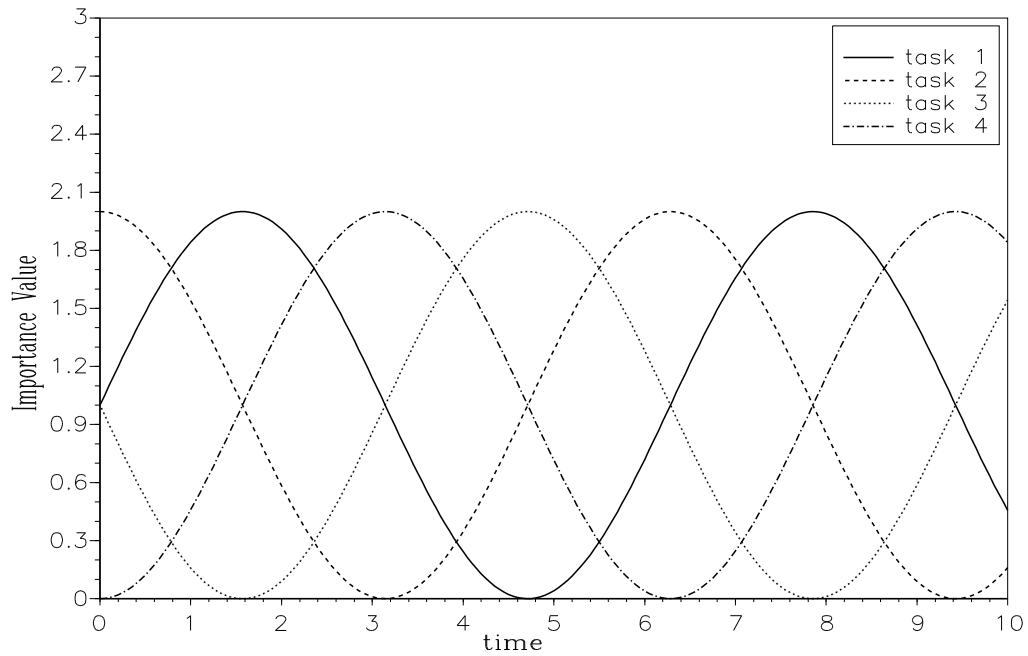


Figure 5.3 — Importance Function Values for Round Robin Policy

5.2. Families of Importance Functions

Often it is instructive to show a function in its general form, as with lines (given by $y = mx + b$) and circles (given by $x^2 + y^2 = b^2$). Since the importance abstraction is based on sets of functions, certain classes of functions, or *families*, can be expressed in general parametric forms where specific values are assigned according to the application. Jensen, in [JENS85], used a six parameter function to describe “value functions.” It is claimed that the “value” of most interesting tasks can be profiled by

$$V_i = K_1 + K_2 t - K_3 t^2 + K_4 e^{-K_5 t} \quad \text{Eq 5.9}$$

where appropriate assignments of the constants K_i could produce value functions which are constant, linear, quadratic, exponential, or a linear combination of any of these.

The importance abstraction allows task importance to be profiled using functions of many forms. We have seen already how sets of functions can be used to emulate traditional scheduling policies. Yet, for most of these examples, we have given specific forms of the functions where the parameters are attributes of the task, like the task's deadline. By examining the general forms of some of these functions, the families of functions available expand the expressiveness of the function form. For example, the nearest deadline first policy may be expressed as shown in Eq 5.3; more generally, however, the form of the function could be given as

$$I_i(t) = \frac{\alpha_i}{\beta_i(d_i - t)} \quad \text{Eq 5.10}$$

where α_i and β_i are constants specific to task i . A set of importance functions based on this family may not necessarily provide nearest deadline first service, but rather exhibit additional properties, such as giving preference to meeting the deadlines of the most critical tasks.

5.3. Novel Policies using Importance Functions

The so-called traditional scheduling policies, and the algorithms that are used to express them, often arise from the requirements of the scheduling mechanisms. Many aspects of tasks and task sets, which should logically be expressed as scheduling parameters, are ignored or simplified so that traditional scheduling mechanisms can be used. One of the most popular scheduling policies is priority ordering, where all aspects of the task are condensed into a single value. Another popular policy, rate monotonic, permits us to make static guarantees about the schedulability of a task set, but the task set must be expressed as a set of periodic tasks, even if this is inappropriate to do so. The importance abstraction, in addition to emulating traditional scheduling policies, allows us to focus on

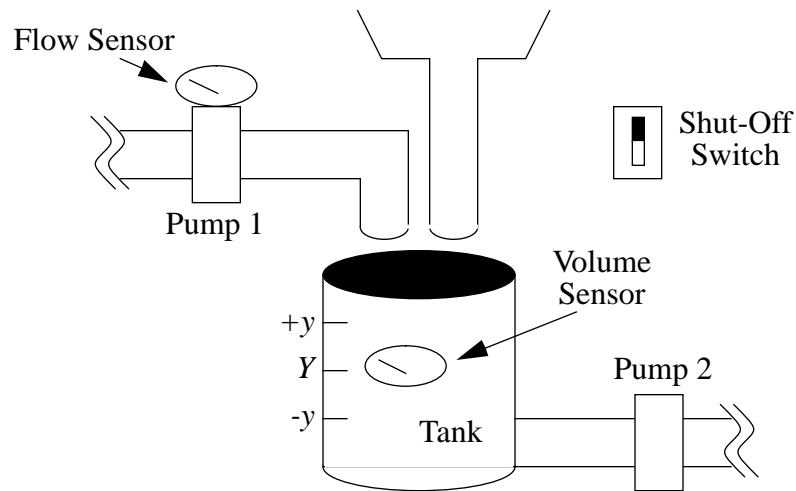


Figure 5.4 — Valve Configuration for a Process Control Example

attributes, conditions, or other events that are not traditionally parameters of scheduling algorithms.

An example of parameters that are difficult to consider in traditional scheduling models is continuously updated variables, as may be found in process control applications. Values from sensors, for example, may affect the choice of tasks to process. For example, a sensor may monitor the pressure on a pump such that when the pressure at the pump deviates significantly from a normal value, the task controlling the pump becomes important.

Let V be such a continuously updated variable such that the value of V at time t is given by $V(t)$. We can include this variable within a task's importance function by composition:

$$I_i(t) = f_i(V(t)) \quad \text{Eq 5.11}$$

Consider a process control application where a single processor maintains two pumps and a shut-down switch in a configuration shown in Figure 5.4. Pump 1 must

maintain a flow of $X \pm x$ units per second into the tank. Some other liquid is entering the tank at a constant rate. Pump 2 must ensure that the volume in the tank is maintained at $Y \pm y$ cubic units to ensure a proper mixture is produced. The Shut-Off Switch is used to turn off Pump 1 and close a valve on the other pipe in case there is either too much or too little volume in the tank, or if Pump 1 cannot maintain the proper flow. Assume that each pump and the switch is controlled by a separate task. The flow maintained by Pump 1 is given by $F(t)$, and the volume within the tank is given by $V(t)$.

We can define the importance functions for the three tasks that maintain these devices such that the task controlling a pump or the switch will become important whenever that device needs attention. Pump 1 needs attention when the flow deviates from the ideal value X by x ; Pump 2 needs attention when the volume in the tank deviates from Y by y . The Shut-Off Switch needs attention when one of three events occur: (1) the volume in the tank exceeds $Y + y$, in which case the tank is in danger of overflow, (2) the volume in the tank is less than $Y - y$, in which case the tank is in danger of being empty and Pump 2 may be damaged, or (3) the flow through Pump 1 cannot be maintained at $X \pm x$, in which case the mixture is spoiled and any more work is useless. The importance functions below are designed to ensure that these tasks are invoked at the proper times.

$$\begin{aligned} \text{Define: } \quad \Delta_1(t) &= |F(t) - X| \\ \Delta_2(t) &= |V(t) - Y| \end{aligned} \tag{Eq 5.12}$$

$$I_1(t) = \begin{cases} \Delta_1(t)y, & \text{if } \Delta_1(t) \leq x \\ xy, & \text{otherwise} \end{cases} \tag{Eq 5.13}$$

$$I_2(t) = \begin{cases} \Delta_2(t)x, & \text{if } \Delta_2(t) \leq y \\ xy, & \text{otherwise} \end{cases} \tag{Eq 5.14}$$

$$I_3(t) = \begin{cases} xy + 1, & \text{if } (\Delta_1(t) > x) \text{ or } (\Delta_2(t) > y) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 5.15}$$

The values $\Delta_1(t)$ and $\Delta_2(t)$ represent the deviations from the ideal values for the respective continuously updated variables.

Note that the design of the importance functions is based on the safety limits of the system so that the safety of the system is directly related to the behavior of the functions. Consequently, it is possible to know exactly which task will be most important under any set of circumstances. In this example, the pumps are serviced according to which metric is proportionally closer to its limit of deviation. These tasks attempt to correct the deviation. If the volume in a tank exceeds its limit, something is wrong that cannot be corrected by invoking the pump control tasks; the importance value becomes constant so that it will not compete with the task servicing the Shut-Off Switch. The switch is guaranteed to be serviced if the safety conditions are violated. Also, the functions are designed to allow a task to become important as a limit is approached rather than after a fault has occurred.

Consider conventional methods for servicing tasks in process control systems. Polling loops are often used to “visit” each task in a round robin manner. At each visit, the task may find that corrections are required. A limit may be placed on how long a task is serviced so as not to starve other tasks. Polling creates a lag time between the occurrence of the problem and the servicing of the control task. The scheduler has no notion of the state of the devices controlled by the tasks; rather, it is up to the tasks to investigate the status of only the device for which it is the controller, and take action accordingly. Consequently, it is much more difficult to assure safety to such a system since the safety depends upon the worst case poll time. The poll time itself depends on the worst case time spent servicing each of the other tasks. This service time should be long enough to allow the proper

corrections to be made (or satisfactorily started) but short enough to allow service to be given to other tasks whose devices also have problems.

Some other methods are not appropriate. Static priority-based scheduling policies cannot cope with the dynamics of the system. There are no deadlines in this system, so deadline-driven policies are inappropriate. Since these tasks are not naturally periodic, rate monotonic theory does not apply. Importance functions are a natural way to express the conditions under which the control tasks must take corrective action.

6 Analyzability of the Importance Abstraction

In Chapter 5 we offered sets of importance functions that could emulate several traditional scheduling algorithms. In particular we have shown that priority-driven, nearest deadline first, and rate monotonic algorithms are easily expressed in terms of functions. It is interesting to note that all three of these policies have the property that tasks are ranked according to some criteria (priority, period, or deadline) and, once ranked, remain in this ranking relative to all other active tasks until some task leaves or some task arrives. We term this property a *static ranking*. In this section we examine these three scheduling policies to determine the *defining property* and prove that the importance function sets given as candidate sets for emulating these policies do in fact meet the defining property. Since these policies all present a static ranking, the proof that the candidate importance function set meets the defining property is similar for each policy.

We continue by examining the nearest deadline first policy in particular. We prove results about nearest deadline first that are commonly accepted, namely that, for a fixed size task set and a priori task arrival knowledge, nearest deadline first will meet all deadlines if any policy can meet all deadlines. We then examine variations of this policy that are not so restrictive: arbitrary arrivals, arbitrary arrivals with a second level of ranking, heterogeneous task sets, and heterogeneous task sets with arbitrary arrivals and critical deadlines.

We conclude this chapter with a discussion of how the relationship between importance functions, the system state, and which task is the most important. By

considering the set of system states, we can predict under what circumstances a give task will be most important.

6.1. Policies with Statics Rankings

Assume that a given system requires that, to meet the system goal, all tasks active in the system must be ranked according to some criteria known at task activation. Further assume that this ranking, once established within an active task set, does not change until the composition of the task set changes. We term rankings that have this property *static rankings*. Several well-known traditional scheduling policies (e.g., priority-driven, nearest deadline first, and rate monotonic) have this property of static ranking.

The rank r for a task is based on the ranking criteria; for a priority-driven system this criteria is the priority, for rate monotonic it is the period of the task, and for nearest deadline first it is the deadline. Importance functions emulating these algorithms must use this ranking criteria as a parameter; moreover, at every point in time, the rank imposed must be maintained by the importance functions. Thus, for all time t ,

$$r_i > r_j \Rightarrow I_i(t) > I_j(t) \quad \forall (i, j \in T \text{ at time } t) \quad \text{Eq 6.1}$$

This is the *defining property* for all scheduling policies based on static ranking.

The proof that a particular importance function set meets this defining property is trivial. For a priority-driven policy, the set of importance functions as given in Eq 5.4 meet this property since, for all time t ,

$$p_i > p_j \Rightarrow I_i(t) > I_j(t) \quad \forall (i, j \in T \text{ at time } t) \quad \text{Eq 6.2}$$

For rate monotonic, the priority is the inverse of the period, so the property holds in this case as well. For the nearest deadline first policy, the ranking is determined by the nearness of the deadline. Hence, for all time t ,

$$\frac{1}{d_i - t} > \frac{1}{d_j - t} \Rightarrow I_i(t) > I_j(t) \quad \forall (i, j \in T \text{ at time } t) \quad \text{Eq 6.3}$$

$$d_i < d_j \Rightarrow I_i(t) > I_j(t) \quad \forall (i, j \in T \text{ at time } t) \quad \text{Eq 6.4}$$

Notice that this defining property does not state that all deadlines are met. The fact that, if all deadlines can be met, the nearest deadline first policy will meet them, is a consequence of the ranking and not in itself a defining property; the nearest deadline first policy is only a single element in the set of policies that guarantee that all deadlines are met.

6.1.1. Determining Completion Time

For policies that have this static ranking property, it is more interesting to ask questions about the schedules imposed by the sets of importance functions. For example, we can ask when a particular task will complete, or under what set of conditions will a task miss its deadline. In general completion time is a difficult aspect to predict; for this discussion we assume knowledge of the complete task set and the attributes of the tasks within.

Assume that a task set T has cardinality n , and that the tasks within T have known arrival times. Let a_i be the arrival time for task i . Without loss of generality we can order the tasks in T such that the tasks are numbered from highest to lowest ranking. Hence, task i has ranking greater than or equal to task j if $i > j$. Tasks with equal rank are ordered by arrival; otherwise, arrival times have nothing to do with the ordering of the tasks in T . Assume the importance function set for task set T is given by

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ r_i, & \text{if } (t \geq a_i) \end{cases} \quad \text{Eq 6.5}$$

for all $r_i \geq 0$.

For a given task j we seek the completion time c_j . There may be some tasks of higher importance that have arrived (become active) before a_j , and there may be tasks that arrive after a_j that are more important than task j . We can identify these more important tasks as those having indices less than j . Those active tasks that are more important than task j at time a_j will complete before task j can begin. Let $p_{j,1}$ be the earliest time task j can start given that no tasks arrive after a_j :

$$p_{j,1} = \max_{1 \leq i \leq j-1} (c_i | (a_i \leq a_j) \wedge (c_i < a_j + w_j)) \quad \text{Eq 6.6}$$

This states that task j can start no sooner than the greatest completion time of the more important active tasks.

But some tasks may arrive between a_j and $p_{j,1}$ plus the amount of work left to complete task j . During this period task j is subject to preemption by some higher ranking tasks, thus possibly delaying the completion time of task j . We call this a “vulnerable period.” Since we must consider this, let $p_{j,2}$ be defined as follows:

$$p_{j,2} = \max_{1 \leq i \leq j-1} (c_i | (a_j \leq a_i \leq p_{j,1} + w_j |_{p_{j,1}}^{\infty})) \quad \text{Eq 6.7}$$

This states that task j can finish no sooner than the greatest completion time of any tasks arriving within the vulnerable period. We use the term $w_j |_{p_{j,1}}^{\infty}$ (as defined in Appendix A) since this value reflects the amount of work left to do on task j after the time $p_{j,1}$. By considering the vulnerable period above, a new vulnerable period is created. To consider this new vulnerable period as well, define $p_{j,3}$ as follows:

$$p_{j,3} = \max_{1 \leq i \leq j-1} (c_i | (a_j \leq a_i \leq p_{j,2} + w_j |_{p_{j,2}}^{\infty})) \quad \text{Eq 6.8}$$

Continuing this chain of logic through iteration k :

$$p_{j,k} = \max_{1 \leq i \leq j-1} (c_i | (a_j \leq a_i \leq p_{j,k-1} + w_j |_{p_{j,k-1}}^{\infty})) \quad \text{Eq 6.9}$$

Since there are n tasks in the set T , there can be at most n periods of vulnerability. Hence:

$$p_{j,n} = \max_{1 \leq i \leq j-1} (c_i | (a_j \leq a_i \leq p_{j,n-1} + w_j |_{p_{j,n-1}}^{\infty})) \quad \text{Eq 6.10}$$

Note that once $p_{j,k} = p_{j,k+1}$ for some k we do not need to calculate any more periods of vulnerability, thus we can make the assignment $p_{j,n} = p_{j,k}$. Task j will complete at time c_j , where c_j is given by

$$c_j = p_{j,n} + w_j |_{p_{j,n}}^{\infty} \quad \text{Eq 6.11}$$

6.1.2. Meeting Deadlines

Since nearest deadline first is a static ranking, the result given in Eq 6.11 also applies for schedules imposed using importance functions emulating the nearest deadline first algorithm. Such a set of importance functions is given in Eq 5.3. We know from [LIU73] that, if deadlines can be met for a given task set, they will be met using the nearest deadline first policy. However, Liu and Layland show this for a set of *periodic* tasks by proving that nearest deadline first will schedule tasks to meet deadlines if the *utilization factor* (the sum over all tasks of the ratios of work required to length of period) for the task set is 1 or less. Unfortunately, the utilization factor proof in [LIU73] only holds for periodic task (a counterexample: task 1 has arrival time $a_1 = 5$, work required $w_1 = 10$, deadline $d_1 = 15$, and task 2 has $a_2 = 15$, $w_2 = 10$, $d_2 = 25$; the utilization factor is 2, yet the task set is feasible).

To show that the nearest deadline first algorithm will meet all deadlines for an aperiodic task set if there exists any schedule which can meet all deadlines, we must show that the completion time from Eq 6.11 for each task is always less than or equal to the task's

deadline; that is, for each task i with deadline d_i , $c_i \leq d_i$. We prove this within the importance abstraction by using the property, given by Eq 6.4, for the importance function set used to emulate the nearest deadline first algorithm. We also need the condition under which any schedule will meet all deadlines. For a schedule to meet every deadline in the task set the schedule must ensure that the following is true for all points in time:

$$\text{AND } \left(\sum_{i=1}^j w_i \Big|_t^\infty \leq \max(d_j - t, 0) \right) \quad 1 \leq j \leq n \quad \text{Eq 6.12}$$

This is actually a set of conditions, all of which must be true. Consider $t = 0$. For $j = 1$, the amount of work done on task 1 over all time must not exceed its deadline. For $j = 2$, the amount of work done on task 2 in addition to the work done on task 1 must not exceed the deadline d_2 . For $j = n$, where $n = |T|$, the amount of work done on all n tasks must not exceed the deadline of task n .

Theorem 1

Given a task set T for which there exists some schedule that meets all deadlines, then a schedule imposed by the nearest deadline first algorithm will also meet all deadlines.

Proof

Assume the tasks of task set T are scheduled by a set of importance functions for which Eq 6.4 is a property. Let T be ordered such that $d_1 \leq d_2 \leq \dots \leq d_n$. Let task k be the lowest indexed task for which $c_k > d_k$, where c_k is the completion time and d_k is the deadline for task k .

There are two cases. First, if there is no idle time between time 0 and time c_k , then the sum of all of the work done on all tasks over that interval is the length of the interval and equals c_k . Therefore:

$$\sum_{i=1}^n w_i \Big|_0^{c_k} = c_k \quad \text{Eq 6.13}$$

Since the property given in Eq 6.4 holds for this task set, only the tasks whose deadlines are d_k or prior are serviced over the interval 0 to c_k ; we may rewrite Eq 6.13 as:

$$\sum_{i=1}^k w_i \Big|_0^{c_k} = c_k \quad \text{Eq 6.14}$$

Also, these tasks are run to completion before task k is completed, so we can replace $w_i \Big|_0^{c_k}$ with w_i :

$$\sum_{i=1}^k w_i = c_k \quad \text{Eq 6.15}$$

But the k^{th} condition of Eq 6.12, for $t = 0$, states:

$$\sum_{i=1}^k w_i \Big|_0^{\infty} \leq \max(d_k - 0, 0) = d_k \quad \text{Eq 6.16}$$

Since $w_i \Big|_0^{\infty}$ equals all of the work required by the task, this expression in Eq 6.16 can be replaced by the value w_i :

$$\sum_{i=1}^k w_i \leq d_k \quad \text{Eq 6.17}$$

By substitution of Eq 6.15 into Eq 6.17, we arrive at $c_k \leq d_k$, a contradiction of our initial assumption that $c_k > d_k$.

For the second case, if there is at least one gap of idle time between time 0 and time c_k , let t_g be the time when the last gap ends so that on the interval t_g to c_k there is no idle time. The work done over that interval must sum to the length of the interval:

$$\sum_{i=1}^n w_i \Big|_{t_g}^{c_k} = c_k - t_g \quad \text{Eq 6.18}$$

Since Eq 6.4 holds, no tasks of index greater than k will be serviced during this interval, so we can change the upper limit of the summation. Also, since each task with index k or less will finish before time c_k , we can replace the expression $w_i \Big|_{t_g}^{c_k}$ with $w_i \Big|_{t_g}^{\infty}$:

$$\sum_{i=1}^k w_i \Big|_{t_g}^{\infty} = c_k - t_g \quad \text{Eq 6.19}$$

The k^{th} condition of Eq 6.12, for $t = t_g$, yields:

$$\sum_{i=1}^k w_i \Big|_{t_g}^{\infty} \leq \max(d_k - t_g, 0) \quad \text{Eq 6.20}$$

By substitution of Eq 6.19 into Eq 6.20, we arrive at:

$$\begin{aligned} c_k - t_g &\leq d_k - t_g \\ c_k &\leq d_k \end{aligned} \quad \text{Eq 6.21}$$

Again, we find the contradiction.

Therefore, if there exists a schedule which can meet all deadlines in a task set, then the schedule imposed by the importance functions which emulate the nearest deadline first algorithm will also meet all deadlines. Since the importance functions and the algorithm impose the same schedule, then the result holds for the nearest deadline first algorithm as well.

■

For rate monotonic, each task is instantiated exactly once during the task's period. This instantiation must complete before its period expires and the new instantiation is created. We can therefore think of each instantiation of a task as a separate task, and the end of the period as that task's deadline. In this sense rate monotonic is similar to the nearest deadline first algorithm where the deadline d_i is given by $d_i = a_i + T_i$, for T_i the period for task i .

6.1.3. Meeting Deadlines with Arbitrary Arrivals

Theorem 1 assumes that the task set T has a constant cardinality n and is known *a priori*. In a system where the task set T cannot be known *a priori*, and where the cardinality is not known to be a constant (i.e., there may be arbitrary future arrivals), we cannot prove that all deadlines will be met. We can, however, create a test which will identify as early as possible when a task will miss its deadline.

Let tasks be requested at arbitrary times such that the request time for task i is less than or equal to the arrival time for task i ; that is, $req_i \leq a_i$. Index the tasks such that, for all tasks $i, j \in T$

$$i > j \Rightarrow (req_i < req_j) \vee ((req_i = req_j) \wedge (a_i < a_j)) \quad \text{Eq 6.22}$$

Thus the tasks are indexed by when they are requested.

We need to define a few functions for convenience. Let $D:T \rightarrow N$ be a function that takes a task and returns a natural number representing the task's current order with respect to deadline nearness. If task i has the j^{th} nearest deadline, then $D(i) = j$. Let $D':N \rightarrow T$ be the inverse function which, given a natural number j , returns the task index whose deadline is currently the j^{th} nearest. Let $n(t)$ be a function which returns the cardinality of the set T at time t . The condition for meeting all deadlines for the task set T at time t is:

$$\text{AND } 1 \leq j \leq n(t) \left(\sum_{i=1}^{D(j)} w_{D'(i)} \Big|_t^\infty \leq \max(d_j - t, 0) \right) \quad \text{Eq 6.23}$$

This condition is similar to the condition given in Eq 6.12. This condition states that, at some time t and for all tasks j from 1 to the current cardinality of the task set T , the sum of the work required by all tasks whose deadlines are priori to task j must be less than or equal to the amount of time between the current time and task j 's deadline. We define the term *overload* to be the state of the task set at time t such that Eq 6.23 is not true.

Theorem 2

Let T be an arbitrarily large task set containing tasks with arbitrary request times. The nearest deadline first algorithm will meet all deadlines if any algorithm can meet all deadlines.

Proof

Assume that a system requests work on tasks at arbitrary time such that the size of the task set is not known *a priori*. Assume that task k is requested at time req_k , and at that time an overload occurs such that some task m cannot meet its deadline using the nearest deadline first algorithm. At time req_k we can construct a task set T_k that

includes all tasks requested from time 0 to time req_k . Let these tasks be indexed according to the nearness of their deadlines such that $i < j \Rightarrow d_i < d_j$. By application of Theorem 1 we know that no algorithm can meet all deadlines if the nearest deadline first algorithm cannot meet all deadlines.

■

6.1.4. Meeting Critical Deadlines, with Arbitrary Arrivals

One of the problems with a pure nearest deadline first algorithm is that the tasks are not otherwise ranked in the presence of missed deadlines such that the most critical tasks are given preference at the expense of the least critical. The importance abstraction can easily express this bilevel ranking, where the nearest deadline first policy is augmented by considering a criticality measure associated with each task. Let us call this new for of nearest deadline first the nearest critical deadline first (NCDF). From the representation of the NCDF policy within the importance abstraction we seek the conditions under which a given task k will meet its deadline, and from that prove that NCDF maximizes a quantity based on the criticality of the tasks serviced.

Let each task i have two attributes: the deadline d_i and a criticality p_i . Assume that the criticality p_i is an element of L , where L is the set of natural numbers in the range MINCRIT to MAXCRIT. To construct a set of importance functions which will implement the NCDF policy we first define a few auxiliary functions for convenience. Define the function $Over: \{T\} \times time \rightarrow Boolean$ as:

$$Over(T, t) = \text{AND}_{1 \leq j \leq |T|} \left(\sum_{i=1}^{D(j)} w_{D(i)} \Big|_t^\infty > \max(d_j - t, 0) \right) \quad \text{Eq 6.24}$$

The function $Over$ returns one if the task set T will not meet all deadlines at time t , zero otherwise. Note that this is a functional representation of the conditions from Eq 6.23. Define $Crit: L \rightarrow \wp(T)$ as a function that takes the criticality level from the set L and returns

the subset of T that share this criticality level. Finally, we define a function $InMostCrit: T \times time \rightarrow Boolean$ that takes a task and a time and returns one if the task is in the set of tasks whose deadlines will be met because they are among the most critical at that time, and returns zero otherwise. The function body is:

```

InMostCrit(i, t) {
   $T' = \emptyset$ 
  for  $k = MAXCRIT$  downto  $MINCRIT$  do
    for each  $j \in Crit(k)$  do
      if not Over( $T' \cup \{j\}$ , t)
        then
           $T' = T' \cup \{j\}$ 
        endif
      endfor
    endfor
  return ( $i \in T'$ )
}

```

Now for the importance functions:

$$\forall (i \in T), \quad I_i(t) = \begin{cases} 0, & \text{if } (t < a_i) \\ (d_i - t)^{-1}, & \text{if } ((d_i > t \geq a_i) \wedge InMostCrit(i, t)) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.25}$$

Given a task k with an importance function defined as above, we seek the conditions under which this task k will meet its deadline. Since we are now considering a task set with arbitrary future arrivals, we cannot predict *a priori* that task k will meet its deadline; rather, we can show the conditions necessary at certain points in time for task k to meet its deadline. At time req_k , task k is schedulable if the following is true:

$$\left(\sum_{i=1}^{D(k)} \left(w_{D(i)} \Big|_{req_k}^{\infty} \right) (p_{D(i)} \geq p_k) \right) \leq d_k - req_k \quad \text{Eq 6.26}$$

We must check this condition not at time req_k but every time a request for service is made, hence:

$$t = req_j | \overset{\text{AND}}{(req_k \leq req_j \leq d_k)} \left(\left(\sum_{i=1}^{D(k)} \left(w_{D^{(i)}} \Big|_t^\infty \right) (p_{D^{(i)}} \geq p_k) \right) \leq (d_k - t) \right) \quad \text{Eq 6.27}$$

This expression states that, for each time t that a new task arrives between the request of task k and task k 's deadline, the following must be true: the sum of the work remaining for tasks whose deadlines are nearer than task k 's and whose criticality is at least as great as task k 's must be less than or equal to the difference between task k 's deadline and the time we are considering.

Biyabani *et al* explore this kind of bilevel ranking in [BIYA88]. They offer a new semantic for the term *guarantee* that reflects the uncertainty of the future task set composition. They state that at request time a task is guaranteed to meet its deadline if (1) it is among the most critical tasks in the current task set, and (2) the arrivals of subsequent tasks do not cause this task to leave the set of the most critical tasks. The system guarantees that the most critical tasks will meet their deadlines; however, we cannot predict which tasks will be in the set of most critical tasks.

We constructed the importance functions so that only the most critical tasks are serviced to completion. When the system presents more tasks than can be serviced without missing a deadline, some tasks must be pruned. The condition *InMostCrit* is used within the importance functions of Eq 6.25 to do this pruning. We can quantify how well the goal of meeting most critical deadlines is being met by summing the criticality values for all tasks whose deadlines have been met by time t . Define the quantity *CritCount*(t) as:

$$CritCount(t) = \sum_{i=1}^{n(t)} \left(w_i \Big|_{a_i}^{d_i} \geq w_i \right) p_i \quad \text{Eq 6.28}$$

When the work done on a task i is greater than or equal to the work required, the criticality value of task i is added to the criticality count *CritCount*. Because the NCDF policy is

greedy, we expect the *CritCount* for the schedule produced to be optimal among all policies. The following lemma supports a theorem that proves that NCDF is optimal with respect to maximizing this quantity.

Lemma 1

Any task set that is schedulable by the nearest deadline first (NDF) policy is also schedulable by the NCDF policy

Proof

Let T be a task set that is schedulable by NDF. Thus, by Eq 6.23 we know that, for all time t , the following is true:

$$\text{AND}_{1 \leq j \leq n(t)} \left(\sum_{i=1}^{D(j)} w_{D(i)} \Big|_t^\infty \leq \max(d_j - t, 0) \right) \quad \text{Eq 6.29}$$

Since the only difference in NDF and NCDF is the presence of the condition *InMostCrit*, as long as *InMostCrit* is true for some task i over all time t , then task i will be scheduled by both algorithms at exactly the same time, for exactly the same duration, and having exactly the same completion time. Let task k be a task schedulable by NDF but not by NCDF. Thus, *InMostCrit*(k, t) must not be true for some time t . This implies by Eq 6.24 that

$$\sum_{i=1}^{D(k)} w_{D(i)} \Big|_t^\infty > d_k - t \quad \text{Eq 6.30}$$

But from Eq 6.29 we know that

$$\sum_{i=1}^{D(k)} w_{D(i)} \Big|_t^\infty \leq d_k - t \quad \text{Eq 6.31}$$

This is a contradiction.

■

Theorem 3

The NCDF policy maximizes the criticality count *CritCount* at time t among all scheduling policies.

Proof

Assume that there exists some scheduling policy A that, at some time t , produces a schedule that has a higher value for $CritCount$ than NCDF. Let T_A be the set of tasks scheduled by policy A by time t , and T_{NCDF} be the set of tasks scheduled by NCDF. If these tasks are equal then their $CritCount$ values must also be equal and thus we have a contradiction.

If the task sets are not equal, then the set of tasks chosen by policy A must contain some tasks not chosen by NCDF. For the quantity $CritCount$ of T_A to be higher than that for T_{NCDF} , policy A either scheduled more tasks or instead scheduled tasks of greater criticality. By Theorem 1 we know that the task set T_A can be scheduled by NDF. By Lemma 1 we know that any task set schedulable by NDF is also schedulable by NCDF. Therefore, policy A could not have scheduled more tasks than NCDF; instead, to have a higher value for $CritCount$, policy A must have scheduled different, more critical tasks.

Since, at every point in time, NCDF chooses the most critical task with the nearest deadline, any more critical tasks chosen by policy A , and therefore schedulable by both NDF and NCDF, would have also been chosen by NCDF. Thus policy A could not have scheduled more critical tasks than NCDF, and a contradiction results.

■

6.1.5. Heterogeneous Task Set

Consider a task set that contains some tasks that are only deadline-driven and some tasks that are only priority-driven. Because the priority-driven tasks do not have a time constraint, most policies schedule the deadline-driven tasks first and use any remaining processor cycles to service the priority-driven tasks. Policies of this type are easily constructed within the importance abstraction by using the following importance functions: Let T_d be the subset of T that are deadline-driven tasks and T_p be the subset of T that are priority-driven tasks. Let p be equal to MAXCRIT. The importance functions for both types of tasks are given by:

$$\forall (i \in T) \quad I_i(t) = \begin{cases} (d_i - t)^{-1} + p, & \text{if } (i \in T_d \wedge a_i \leq t < d_i) \\ p_i, & \text{if } (i \in T_p \wedge a_i \leq t) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.32}$$

Since the importance of a deadline-driven task is always higher than the importance of any priority-driven task, Theorems 1 and 2 from the previous sections still hold. A characteristic of schedules produced using this set of importance function is that priority-driven tasks are always deferred until there are no deadline-driven tasks in the set to be serviced. Thus, as a consequence of trying to meet the deadlines of the tasks of subset T_d the priority-driven tasks must wait until there are no active deadline-driven tasks.

Consider a system that must meet all deadlines as well as attempt to minimize the average response time to the priority-driven tasks. If there is no stated benefit from servicing the deadline-driven tasks sooner rather than later, as long as the deadline is met if it can be met, then we want a schedule that defers deadline-driven tasks to the last possible moment. Unfortunately, deferring deadline-driven tasks without *a priori* knowledge of future task arrivals may indeed cause some deadlines to be missed where not deferring the tasks (as with NDF and NCDF) would have met the deadlines. Consequently there must be restrictions on the task set in order to explore a policy that uses procrastination of deadline-driven tasks to reduce the response time for priority-driven tasks.

Clearly, the most conservative restriction is to require a fixed size task set that is known *a priori*. Let each of the n tasks in T be indexed thus: tasks 1 through m are elements of T_p and are ordered by increasing arrival times, and tasks $m+1$ through n are the elements of T_d and are ordered by increasing deadlines. To keep the procrastination of deadline-driven tasks from causing some task's deadline to be missed, the latest possible starting time for a given task i such that it can still meet its deadline must be determined. Define s_i to be this latest possible starting time:

$$s_i = \min_{i \leq j \leq n} (d_j - \sum_{k=i}^j w_k) \quad \text{Eq 6.33}$$

The restriction of a fixed size task set known *a priori* can be relaxed to allow arbitrary arrivals with conditions placed on when the request for service for each task is made. Assume that the tasks are now indexed by their request times such that $i < j \Rightarrow req_i < req_j$. The restriction must ensure that, if any two tasks' deadlines are sufficiently close together, then the tasks must be requested appropriately. Recall that $D'(i)$ returns the index of the task whose deadline is the i^{th} nearest. If the difference between the deadlines of tasks $D'(i+1)$ and $D'(i)$ is less than the quantity $w_{D'(i+1)}$, then it is possible for task $D'(i)$ to be deferred in such a way as to interfere with the meeting of task $D'(i+1)$'s deadline. Both tasks can be taken into account if the task whose deadline is later is known about at the same time as or before the task whose deadline is nearer. Specifically, the request times for tasks $D'(i)$ and $D'(i+1)$ must be ordered such that:

$$d_{D'(i+1)} - w_{D'(i+1)} < d_{D'(i)} \Rightarrow req_{D'(i+1)} \leq req_{D'(i)} \quad \text{Eq 6.34}$$

Rewriting Eq 6.33 to reflect indexing tasks by request time order, the latest starting time for some task i is given by:

$$s_i = \min_{D'(i) \leq j \leq |T_d|} (d_{D'(j)} - \sum_{k=i}^j w_{D'(k)}) \quad \text{Eq 6.35}$$

In either case, a set of importance functions for a procrastination policy is:

$$\forall (i \in T) \quad I_i(t) = \begin{cases} (d_i - t)^{-1} + p, & \text{if } (i \in T_d \wedge s_i \leq t < d_i) \\ p_i, & \text{if } (i \in T_p \wedge a_i \leq t) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.36}$$

6.1.6. Heterogeneous Task Sets with Critical Deadlines and Arbitrary Arrivals

We can combine the conditions from the importance functions of Eq 6.25 and Eq 6.36 to form a set of importance functions that provide guaranteed service to the most critical deadline-driven tasks while minimizing the average response time for tasks that are

priority-driven. Assuming the restrictions on the request times for the task set as given in Eq 6.34, the importance functions are:

$$\forall (i \in T) \quad I_i(t) = \begin{cases} (d_i - t)^{-1}, & \text{if } (i \in T_d) \\ & \wedge (s_i \leq t < d_i) \\ & \wedge InMostCrit(i, t) \\ p_i, & \text{if } (i \in T_p \wedge a_i \leq t) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.37}$$

In the nearest deadline first policy processor idle time occurs only after the deadlines of all of the active tasks are met. With a heterogeneous task set, the idle time is used to service the priority-driven tasks. When the deadline-driven tasks are deferred until the last possible moment, the priority-driven tasks are serviced sooner, thus moving the idle time in between the servicing of tasks from T_p and tasks from T_d . The final variation on the nearest deadline first policy presented here observes that, although there may be no benefit from servicing deadline-driven tasks earlier than later, there is no benefit from waiting to serve them while idle time exists. We construct a set of importance functions that implement a policy that (1) meets the deadlines for tasks in T_d , (2) prunes the least critical deadline-driven tasks when necessary, (3) reduces the response time for tasks in T_p , and (4) eliminates processor idle time if any task is active.

Define the function $Active:\{T\} \rightarrow Boolean$ to take a task set and return the value one if the set has any tasks which have arrived but for whom service is not completed, and return value zero otherwise. The set of importance functions is:

$$\forall (i \in T) \quad I_i(t) = \begin{cases} (d_i - t)^{-1}, & \text{if } (i \in T_d) \\ & \wedge (s_i \leq t < d_i) \\ & \wedge InMostCrit(i, t) \\ p_p, & \text{if } (i \in T_d \wedge a_i \leq t < s_i \wedge \neg Active(T_p)) \\ p_p, & \text{if } (i \in T_p \wedge a_i \leq t) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.38}$$

Schedules produced using these importance functions will service deadline-driven tasks in criticality order during what would have been idle time until either some priority-driven task becomes active or the current time equals the latest possible start time for this task.

Since servicing tasks from T_d during the idle time will affect the latest possible start time, the term s_i can be made into a continuous function $s_i(t)$:

$$s_i(t) = \min_{D(i) \leq j \leq |T_d|} (d_{D'(j)} - \sum_{k=i}^j w_{D'(k)} \Big|_t^\infty) \quad \text{Eq 6.39}$$

Replacing s_i with $s_i(t)$ in Eq 6.38 will constantly update the latest possible start time. As this time is made later, the priority-driven tasks are given longer service times before being preempted for the deadline-driven tasks. This further reduces the average response time for tasks in T_p .

6.2. Projections

In Section 5.3. we constructed an example that showed that importance functions could be used to express scheduling relationships based on continuously updated variables. The design of the importance functions given in Eq 5.12 through Eq 5.15 was based on the safety limits of the system. As a consequence the safety of the system is easily proven.

Given a set of importance functions and a system state, system designers can project which task in the set will be most important by applying the parameters that make up the system state to the importance functions. This suggests the following equation:

$$M (\text{set of importance functions} \times \text{system state}) = \text{the most important task} \quad \text{Eq 6.40}$$

The system designer can also derive the conditions under which one of the tasks of the task set will be the most important task. This requires determining the set of system states for which the given task's importance function evaluates to a greater value than the importance functions of any other task in the task set.

Consider the importance functions given in Eq 5.12 through Eq 5.13:

$$\begin{aligned} \text{Define: } \Delta_1(t) &= |F(t) - X| \\ \Delta_2(t) &= |V(t) - Y| \end{aligned} \quad \text{Eq 6.41}$$

$$I_1(t) = \begin{cases} \Delta_1(t)y, & \text{if } \Delta_1(t) \leq x \\ xy, & \text{otherwise} \end{cases} \quad \text{Eq 6.42}$$

$$I_2(t) = \begin{cases} \Delta_2(t)x, & \text{if } \Delta_2(t) \leq y \\ xy, & \text{otherwise} \end{cases} \quad \text{Eq 6.43}$$

$$I_3(t) = \begin{cases} xy + 1, & \text{if } (\Delta_1(t) > x) \text{ or } (\Delta_2(t) > y) \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 6.44}$$

Here, the values of $F(t)$ and $V(t)$, expressed through $\Delta_1(t)$ and $\Delta_2(t)$, represent the system state, and hence determine which of the tasks are most important. We can determine the conditions under which each task will become most important by examining the interactions of the various of the functions. When both $\Delta_1(t) \leq x$ and $\Delta_2(t) \leq y$, task 1 becomes most important if $I_1(t) > I_2(t)$:

$$\Delta_1(t)y > \Delta_2(t)x \quad \text{Eq 6.45}$$

$$\Delta_1(t) > \Delta_2(t)\frac{x}{y} \quad \text{Eq 6.46}$$

In terms of $F(t)$:

$$|F(t) - X| > \Delta_2(t)\frac{x}{y} \quad \text{Eq 6.47}$$

$$X - \Delta_2(t)\frac{x}{y} > F(t) > X + \Delta_2(t)\frac{x}{y} \quad \text{Eq 6.48}$$

By similar derivation, task 2 is most important when $V(t)$ is bounded as follows:

$$Y - \Delta_1(t)\frac{y}{x} > V(t) > Y + \Delta_1(t)\frac{y}{x} \quad \text{Eq 6.49}$$

Task 3 is most important if either $\Delta_1(t) > x$ or $\Delta_2(t) > y$ since these conditions represent when the system is not operating safely. In this example the system the conditions for safe operation is precisely the conditions task 3 to become most important. In systems where safety is essential, the importance functions offer an opportunity to use the expression of the safety conditions directly in the scheduling process.

7 Implementing the Importance Abstraction Efficiently

The scheduler within the importance abstraction logically consists of a function M that returns the most important task at the moment the function M is evaluated. In the general case, importance functions can be arbitrarily complex. To ensure the service of the most important task at every point in time, the function M must be evaluated at every point in time. This assumption serves a purpose for the theoretical analysis of sets of importance functions, but for real systems, an implementation of such a scheduler would be impractical.

Although the importance abstraction places no restrictions on the complexity and composition of the set of importance functions profiling the task set for a particular system, it is clear that certain sets of importance functions may have properties which lead to an efficient implementation of the scheduler. These properties form two main classes: those sets of importance functions for which discrete evaluation points can be determined, and those sets of importance functions for which discrete evaluation points must be assigned.

7.1. Discrete Evaluation Points

There are two ways to determine discrete points in time for importance function evaluation: (1) find the points of intersection of the importance functions, and (2) determine when the parameters to the importance function may change. Using the intersection method may be more efficient in terms of reducing the number of evaluation points; keying on the

parameters may be easier to implement if the changes in the state or value of the parameters can be signaled, as with an interrupt.

With the intersection method we begin with the highest-valued function at this point in time. By pairwise evaluation we can determine when the next intersection point will occur, and as a consequence which function will be the next highest-valued function, since after this point of intersection another task will become more important than the currently most important task. The importance function of the new most important task is used in the pairwise evaluation to determine the next point of intersection, and hence the next scheduling point. Unfortunately, in the general model importance functions can be arbitrarily complex, and finding intersection points can be quite difficult. In fact, for functions of degree four or greater, it is impossible, in general, to find the intersection points of any two given functions. Even degree three is difficult, although there exists a closed form expression for finding the roots.

The other method for determining evaluation points is to use changes in the parameters to the functions to signal reevaluation of the functions. Many of the system characteristics which may be used as parameters to the importance functions change at discrete times as the result of an “event.” If these discrete times can be known *a priori*, then these times can be built into the scheduler. If the events are signaled by the system, then the scheduler can use this signal. An important example is the set of parameters whose values are determined after a system interrupt. If it can be shown that the importance functions will maintain a stable relative ranking until one of a set of identifiable events occurs, then the scheduler can use that set of event signals to determine the scheduling point.

It is worth noting that there is an efficient implementation for each of the examples in Chapters 5 and 6. The scheduling algorithms for which static rankings exist are particularly easy since the discrete events necessary to awake the scheduler are the arrival

and departure of tasks. The example in Section 5.3. uses continuously updated variables—in the theoretical domain these variable are updated infinitely often. Since the sensor devices take an analog input and convert it to a digital reading, the devices are actually reporting readings at a discrete rate. Since the importance functions given for this example use these sensor values, the scheduler need only be invoked when any one of the values change.

7.2. Approximations

For some situations it may not be possible to discover simplifying aspects of the set of importance functions. Consequently, in this case the scheduler must evaluate every function in the set of importance functions as often as is possible. Unfortunately, it may be impossible to assure that the most important task is being scheduled at every point in time, and thus in this case the implementation only approximates the importance abstraction.

The value of the function at its time of evaluation approximates the value of the function until the next time the function can be evaluated. Furthermore, if there are no parallel evaluations, the values will be the result of evaluations of each function at a different time. Fortunately, this is a common problem in real computing systems, so the solutions are not esoteric.

The degree of imprecision tolerable by a specific system is system-dependent. By system examination the worst-case time between evaluations can be determined. If the system can tolerate the possibility of the wrong task being serviced for at most the worst-case amount of time, then the approximations are adequate for the system. Returning to the novel example of Section 5.3., we can consider invoking the scheduler only after any one of the continuously updated variables changes by a set amount. The analysis of the system suggested in Chapter 6 must take these approximations into account.

If the system cannot tolerate the degree of imprecision inherent, the importance abstraction can be implemented in dedicated hardware. If a digital processor is used, the approximations still exist but the worst-case time between evaluations may be reduced to a tolerable level. An analog device may be used, eliminating the need for approximations, but this introduces a problem with how often to sample. Again, some degree of precision will be lost as the continuous functions are represented and evaluated by an inherently discrete system.

8 The Communication Subsystem

Each task in a system has some inherent importance, and when a task endeavors to communicate with another task, the communication between the two tasks also has an importance. This importance is derived from how important it is for the sending task to convey its information and how important it is for the receiving task to receive it.

We use the term *communication* to denote a transfer of data between two (or more) physically separate tasks. This communication is initiated when a sending task issues a request to the communication server. The request has as one of its parameters the data to be transferred; this data is bundled into a *message* for use by the communication server. The receiving task also issues a request; it asks the communication server to notify it when a message arrives for it. It is what happens within the communication server, and the relationships among the importance values of the tasks using the communications and how the communications themselves are treated, that are the topics of this chapter.

One of the advantages to using a local area network for communication is the economic impact of replacing point-to-point wiring with a common communication medium. A disadvantage that follows directly is that the network becomes a shared resource that must be managed and whose access must be arbitrated. Viewed separately, the communication server is a “system” as defined in Chapter 4, and the tasks are the requests made to it by the users. Within the system, however, there are several other distinct places where the system model can again be applied: at the message level, where the messages compete for buffers and processing; at the internetwork level, where the constituent packets of a message compete for route-related resources; and at the medium access control level,

where the stations compete for placing their packets onto the physical medium. In general there can be arbitrarily many such points of contention, and at each of these some discrimination policy must be in place to determine the order of processing. Although it would seem that all of the discrimination policies within a communication subsystem should work toward a common master policy, in practice this is seldom the case.

In this chapter we present the common approaches currently employed to provide some form of discrimination of important versus unimportant communication. We next briefly review the standard communication architecture and provide more detail on several of the layers implementing the data transfer services. We then consider how to apply the concepts of using importance functions within the communication subsystem, and discuss the issues raised and the advantages presented.

8.1. Discrimination Techniques

A *discrimination technique* is the method by which different levels of service can be applied to communication requests. Message discrimination techniques can be divided into two general approaches, as shown in the taxonomy given in Figure 8.1. These approaches are *priorities* and *levels of service*. Priorities are values assigned to messages for use in ordering the messages at decision points. The priority value may be *static*, that is, fixed over a message's entire lifetime, or it may be *dynamic*, and change with varying conditions within the system. A level of service specifies the message's privileges, such as the amount of time allotted to service messages of this level, permission to bypass flow control constraints, or permission to preempt other messages.

8.1.1. Priorities

Prioritization is the most popular method for discrimination in a network. A priority is a mechanism by which a message has a relative value assigned to it for use during

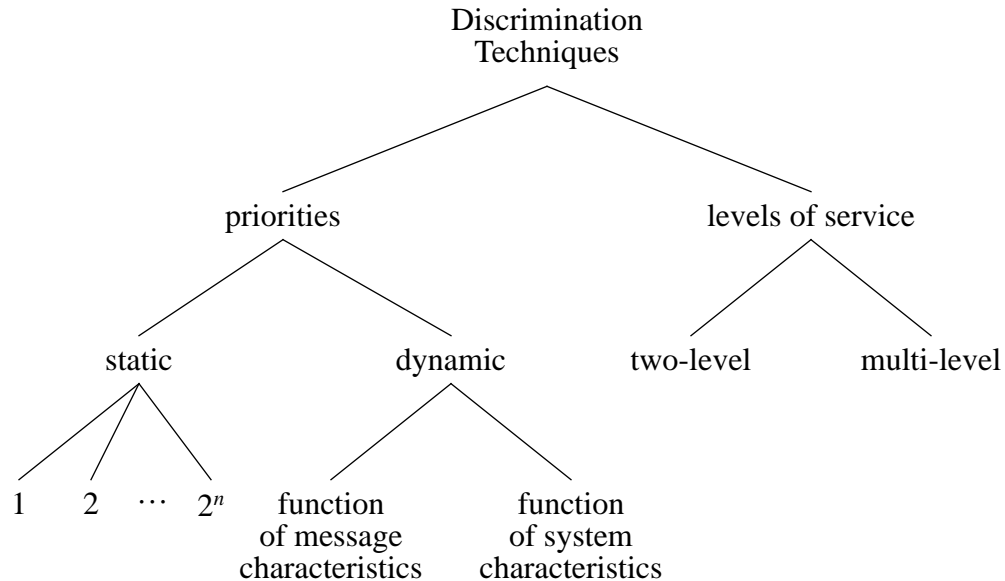


Figure 8.1 — Taxonomy of Discrimination Techniques

competition for resources. This value, therefore, imposes some ranking on those messages competing for service. The server chooses the highest priority message each time it can make such a choice. Within a distributed computing environment, this satisfies, at least nominally, the users' concerns that some communications are more urgent than others.

Several observations, however, pervade the use of priorities. First, if the architecture is layered according to functionality, as has become the universal approach, there is possibly a set of one or more queues at each interface between layers where each layer may dictate its own priority scheme. For example, there may be a medium access control service with eight priorities for providing some ordering to all messages attempting to access the medium, an internetworking service that has no priority structure, and a reliable end-to-end service with a simple two-value priority for providing some ordering to the protocol processing within the transport layer—yet there may not be a clear mapping among these layers. Using the common token ring local area network as an example, the

two priority levels of this transport layer must somehow be mapped (through a nondiscriminating network layer) into a specification of time values expressed in milliseconds which limits the token rotation time around a physical ring. No such mapping exists, yet the performance seen by each level of priority at the transport layer is completely dependent on how this mapping is done. In general, the meanings of priority values remain consistent only within each layer.

Second, locally generated priorities may affect the way in which the messages are handled by nonlocal agents. Consequently, the priority scheme must be globally administered to ensure that each priority level has the same meaning at each processing point. Priorities can only be effective if every participant agrees with and adheres to the meanings of the various priority levels.

Third, no communications protocol in common use completely avoids the problem of *priority inversion*, where a message of lower priority prevents a message of higher priority from being served. Consider a medium access control layer where no preemption of messages is allowed while they access the medium. Once a message is granted access, no other message, regardless of its priority, can be serviced until the current message completes. If the second message is actually of higher priority than the first message, a priority inversion occurs. (We examined the effect on priority inversion and performance of having a preemptable MAC service in [STRA91].) In general protocols trade some priority inversion for reduced complexity, efficiency, and performance.

Fourth, priority schemes in most extant standards are static—that is, once assigned, the priority value is unchangeable. In many implementations of medium access control protocols it is impossible to change the priority of a message once that message has been enqueued for transmission. Yet static priority is not responsive to the dynamic nature of the environment, nor does it represent time constraints appropriately. Messages that spend a

significant portion of their transfer time in queues within stations may benefit from a dynamic priority scheme. For example, Gaitonde *et al.*, in [GAI90], explored changing priorities on certain messages as those messages aged in the communication subsystem, and found this to improve service.

Finally, the additional protocol processing required to use priorities may cause significant overhead. Employing prioritization mechanisms often wastes bandwidth as the stations on the network try to determine which among them has the highest priority messages. Peden has shown that average delay increases and overall throughput decreases when a priority scheme is implemented [PEDE88]. The need to order messages must outweigh this drawback for priorities, or any discrimination technique, to be effective.

Consequently, in current practice there is no overall consistent end-to-end view of priority. Protocol standards are developed by different groups than those who design and develop the operating system and the task scheduling scheme. Even within the communication subsystem the standards for each layer are developed by different groups with different objectives. Yet clearly the service characteristics of the tasks using the communication subsystem are directly affected by how each component of the communication subsystem handles requested communications. The only way to make statements about task service times is to have a consistent approach to scheduling service at all layers.

8.1.2. Levels of Service

Prioritization of messages represents one way to provide different levels of service to various users: by affecting the ordering of messages for service some messages will have quicker response times (the delay between the request for the communication and the acknowledgement that the request has been completed). In general, however, providing the

user with various levels of service encompasses more than just message ordering. Each message must be processed by the service providers within the communication subsystem. Certain constraints are placed on the processing of the messages, including the amount of time allotted for processing these messages, the number of outstanding messages allowed by the flow control procedures, or the rate at which messages may be transmitted.

Communication protocols endeavor through offered levels of service to guarantee certain aspects of the message exchanges. Ferrari [FERR90] compares these guarantees to a contract established between the service user and the service provider: if the user meets certain conditions concerning its service requests, the communication subsystem will provide the level of service required.

One approach to providing several levels of service in a communication subsystem starts with two basic levels: a *normal* service and an *expedited* service. In the Transmission Control Protocol (TCP, [DARPA81a]) and the ISO Transport Protocol Class 4 (TP4, [ISO8073]), the normal service is for common communication, while the expedited service bypasses flow control constraints as well as assuming a higher priority in processing. By ignoring flow control constraints the expedited messages may overwrite normal messages waiting in message buffers; however, no expedited message will be delayed while waiting for buffers to become free. Along these lines we can consider adding a new level of service called *preemptive*. This level allows a service that not only bypasses flow control but preempts messages even as they are being processed. By analogy, a fire truck is “expedited” since it may ignore traffic signals, while a fire truck with a plow on the front is “preemptive” since, with the plow, it can also push cars out of the way.

Dempsey *et al.*, in [DEMP92], explore the idea of allowing a user-specified degree of reliability such as may be useful in time-constrained communications where progress is more important than completeness. The user indicates what density of errors (the number

of lost packets per any set of n contiguous packets) is permissible without the transmission losing its meaning; such service is thought to be useful for audio and video data transmission.

Time-division multiplexing is another place where levels of service can be established. An example of this is the timed-token approach in the Fiber Distributed Data Interface (FDDI) standard [ANSI86]. In FDDI, a certain class of messages, called the *synchronous* class, requires at least some minimum amount of service each time period. A second class of messages, called *asynchronous*, may utilize the time left over. For messages that require service at precisely the same time each period, new enhancements to FDDI provide a class of service called *isochronous*.

As observed with priorities, there is no consistent end-to-end view of offering levels of service. Between any two layers of the communication subsystem, including the tasks that use the communication services, there is no general rule for mapping one layer's levels of service into the levels of service of the other layer. This is compounded by the fact that each layer has a set of functions specific to that layer, and these functions may be reflected in what levels of service are offered. Again, a consistent approach is required before any statement about the service can be made.

8.2. The ISO Reference Model

In 1984 the International Organization for Standardization (ISO) presented the Open Systems Interconnect (OSI) Reference Model [ISO7498], shown in Figure 8.2), and this model has since become the means by which the division of functionality and service are described within a communication subsystem. From bottom to top, the *physical layer* defines how bits are represented on the physical medium. The *data link layer*, of which the *medium access control* (MAC) and the *logical link control* (LLC) are sublayers, governs

the framing of data bits into units called frames, or packets, and arbitrates station access to the physical medium. The *network layer* provides the mechanism for connecting several network segments to form an internetwork such that the packets can be routed across segment boundaries. The *transport layer* ensures reliable, end-to-end delivery of arbitrarily large messages; these messages are segmented into bounded-size packets which are error controlled. The *session layer* uses synchronization points to provide dialogue control. The *presentation layer* addresses disparity in data representations. The *application layer* offers services to the communication user built upon the functionality of the six layers below and that are specific to that user's needs.

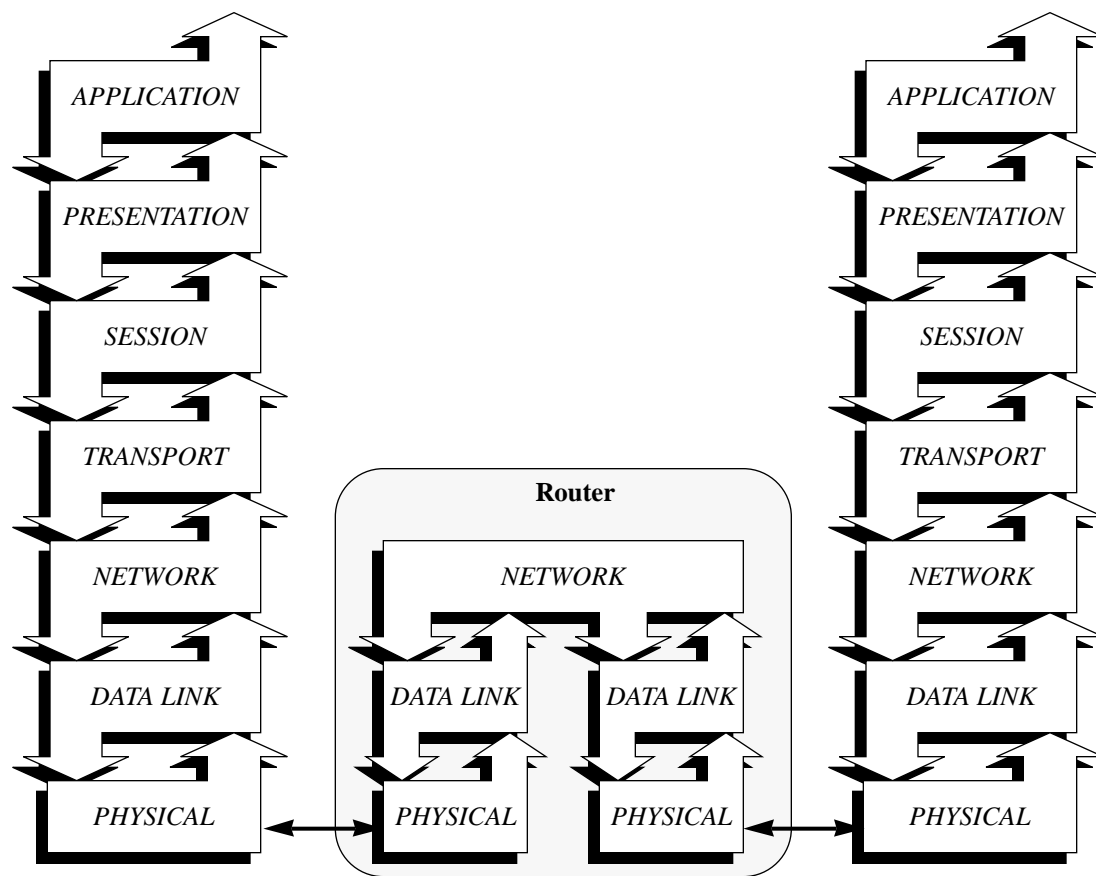


Figure 8.2 — The OSI Reference Model

For this chapter we examine primarily the data delivery services of the transport layer and below since this portion of the layered reference model represents the minimum functionality that is required for a robust data delivery service over a general network topology. In this section we discuss in more detail the functions of the MAC, network and transport layers, especially the current approaches to providing a means for discrimination of certain data in favor of certain other data.

8.2.1. Media Access Control Layer

The protocol imposed by the medium access control layer within a station governs the access by that station to the shared physical medium. Stations are distributed about the physical medium and are provided access to this medium by the MAC layer. The set of MAC implementations around the physical medium work collectively to impose some policy for access. There is a queue of requests for access at each station, as shown in Figure 8.3; the MAC implementations in each station exchange queue information to determine which station may next satisfy a request.

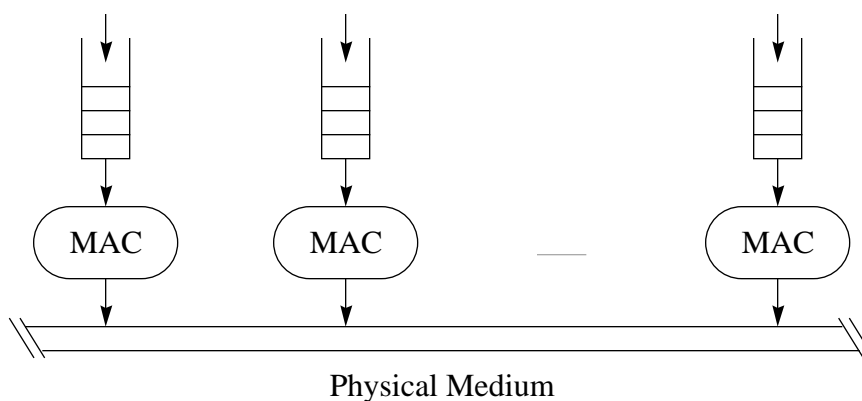


Figure 8.3 — Medium Access Control

Since each station's MAC protocol must work in concert with all other MAC protocols to provide an access policy, some mechanism independent of the data transfer must allow these MAC protocols to communicate. Ethernet [METC76], and its standardized counterpart, IEEE 802.3 [IEEE85c], passively monitor the medium to determine when access can be granted. Since this method can produce collisions, backoffs and retry mechanisms are included within the protocol. Other protocols are more active in granting access: IEEE 802.4 Token Bus [IEEE85a], IEEE 802.5 Token Ring [IEEE85b], ANSI Fiber Distributed Data Interface (FDDI [ANSI86]), and the SAE High Speed Ring Bus (HSRB [SAE87]) all use special frame types called *tokens* to grant access permissions. The Dual Queue Dual Bus (DQDB [IEEE89]) uses each of its two busses to carry access requests for the other bus. Two other access methods include register insertion, where messages are inserted onto the medium via buffers called registers, and slotting, where slots for transmitting are assigned to stations.

Arbitrating access to the physical medium is often augmented by some discrimination scheme. Discrimination at the MAC layer generally requires (1) knowledge of what packets are enqueued at this station, and (2) what priority level is the highest among all other stations. How this information is used distinguishes the various methods of medium access.

Ethernet and IEEE 802.3 explicitly state that no priority mechanism is to be used since each station is a passive participant. As long as no collisions (two or more stations attempting to access the medium at the same time) take place, access is granted in FIFO order; when collisions occur, the binary exponential backoff scheme perturbs the order of access, possibly producing a LIFO order for some period of time.

Token-passing protocols use a special frame called a *token* as a "permit" for transmission. Stations using token passing protocols are generally organized into a logical

ring, whether or not a physical ring topology is present. When a station sees the token and meets certain requirements, that station may capture the token and commence transmission. When required to cease transmission, the station passes the token to a logical neighbor that then may capture it.

Token-passing protocols are divided into two major groups: *timed token* and *token reservation*. Timed token protocols, notable among them IEEE 802.4 Token Bus and ANSI FDDI, circulate a token in logical ring order such that the each station is granted permission to transmit packets for a bounded amount of time. Each level of service will be given a target token rotation time which specifies that a token granting permission to transmit packets at this level will arrive before this target rotation time has elapsed. Due to this assertion timed token protocols are often termed “deterministic.”

Token reservation protocols, such as IEEE 802.5 Token Ring and SAE HSRB, use the token both for granting permission to transmit as well as for polling the stations to determine which among them has the highest priority packet. When the token is captured, it becomes the header for the packet being transmitted. The packet makes a complete tour of the stations on the ring, during which each station may bid on the next priority level based on the packets within that station. The token is reissued by its current captor at the highest bid, which is the priority of the highest priority packet enqueued at all of the stations. In this manner packets are (nearly) ordered by their priority.

8.2.2. Network Layer

The network layer has the responsibility of directing packets from one network segment, through devices called *routers*, to their destination network segment and station. This is accomplished by the cooperation of the network layer protocols in the stations across the internetwork. A source station places an address in a network layer packet,

endowing it with enough information to direct its handlers (i.e., the routers) to deliver it to the destination station in the destination network segment. Some addresses carry specific routing information, while others depend on routing databases or routing algorithms to guide the packets through the internetwork. In general, however, the address pair consisting of the destination station and network segment must uniquely identify the destination station among all stations in all networks.

Routers are the principle agents of the network layer. Typically a network segment will consist of all of the stations attached to a single physical medium or a series of bridged media. One or more of these stations may be routers physically attached to this segment as well as to others, and participating in all of them, as seen in Figure 8.4. Traffic from one segment destined for any other segment attached to the router can travel through that router. Any MAC address translation required (say, from Ethernet to FDDI) is performed by the router.

Some network layer protocols provide a service that is connectionless while some establish and maintain a connection between each endpoint and all of the routers along the path. The Internet Protocol (IP [DARPA81a]) is an example of the former; X.25 [CCITT84] is an example of the latter. An interesting new idea is presented in the Xpress Transfer Protocol (XTP [PEI92]), where the network layer is combined with the transport layer into a *transfer* layer. Here the network layer functionality exhibits behavior of both the connectionless and connection-oriented approaches since, while logically separate, an XTP connection and the path supporting the connection share the same packet-switched mechanisms for their maintenance.

Depending on the approach used to provide the network layer functionality, several issues emerge. The routers along the path represent potential points of congestion and contention for the routers' resources. Connection-oriented approaches focus on dedicating

buffer space up front so that contention for that space is eliminated. Connectionless approaches instead depend on a roughly uniform demand on its buffers; higher layer recovery techniques are assumed for buffer overruns. Yet neither approach directly addresses packet processing or end-to-end bandwidth reservation as resources. Interestingly, some solutions have come from transport layer research.

Flow control at the transport layer governs the use of buffers at the endpoints of a connection. When congestion occurs anywhere along the path, and data is lost, the error control procedures will detect it. The flow control procedure can use the error reports to

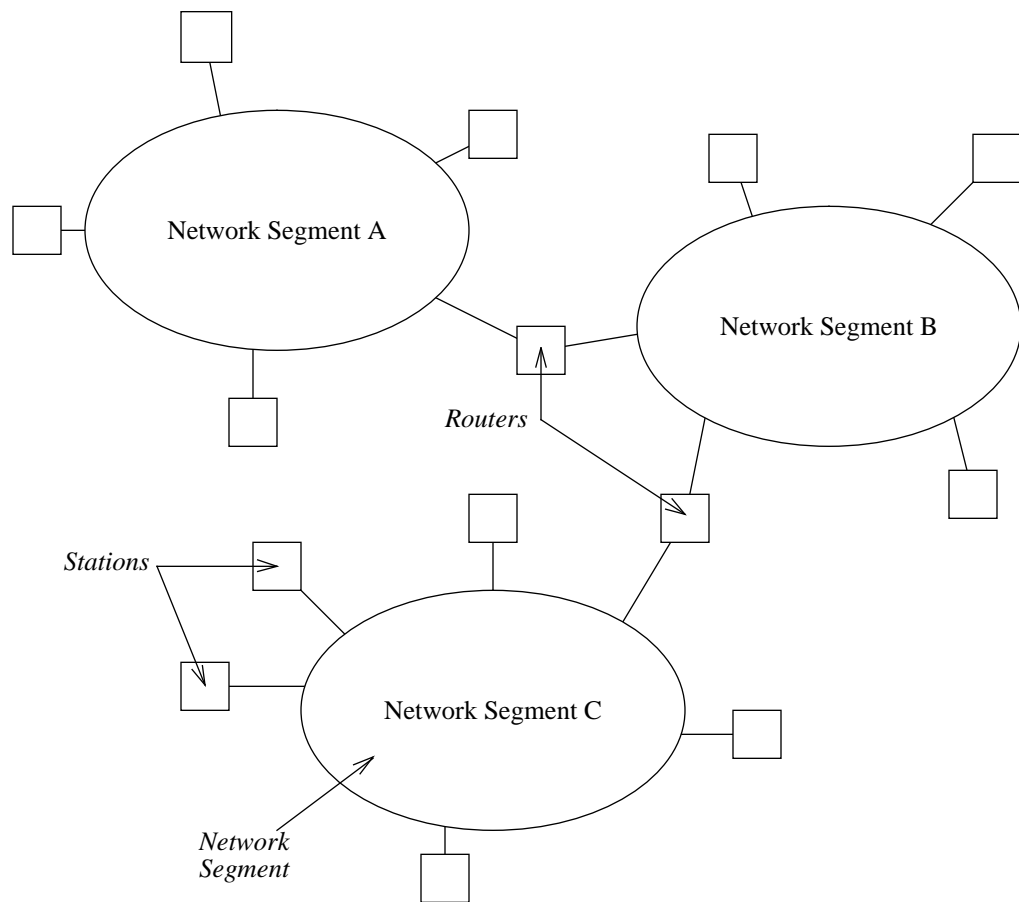


Figure 8.4 — An Internetwork: Stations, Routers, and Network Segments

adjust the number of packets flowing from the sending endpoint. TCP takes this approach. NETBLT [CLAR87] uses a technique called *rate-based flow control*, where two parameters, *burst* and *rate*, regulate the flow of packets: a burst of packets is sent at a specified rate. VMTP [CHER88] also uses rate-based flow control, but instead of burst and rate, it uses an *interpacket gap*. XTP applies rate control independently from flow control: the rate is not deduced from the error reports at the endpoints; rather, it is derived via packet exchanges where the maximum rate and burst values for the path are found. This maximum rate represents the fastest speed at which packets can be injected onto this path according to the processing capacity along the path. In this manner XTP addresses route control as a network layer issue rather than an end-to-end transport layer issue.

Although not prohibited, most network layer services do not provide discrimination even though route processing presents a point of contention. Since each router along the path must employ a MAC protocol to provide access to the medium, packets are ordered by the policies of the MAC layer. However, a router may serve disparate network segments, each using its own brand of MAC protocol. Priorities and levels of service may get lost in the translation. Typically no packet ordering or levels of service are distinguished at this layer. XTP, however, is an exception. XTP's discrimination mechanism, the *sort* field, is used to order packets within each router along the path due to the nature of the transfer layer architecture.

8.2.3. Transport Layer

A transport protocol provides reliable, end-to-end delivery of arbitrarily large messages by segmenting the message into one or more packets and employing the services of the network layer, and through the network layer, the MAC layer. Upon a communication request, the transport server segments the message from the request into one or more packets; these packets are enqueued for the network and MAC layer services. A robust

transport service provides end-to-end flow control to reduce buffer overruns, and error control to ensure a complete and in-order delivery of the message.

Flow control is usually provided by using a sliding window technique where at most a set number of packets are outstanding at any time. The window represents a contiguous set of packets; as the receipt of a packet is acknowledged, the window may advance. Error control detects when packets have not been acknowledged, either through use of timers or negative acknowledgements or both. Packets considered lost are retransmitted. It is also the duty of the error control procedures to detect duplicate packets, as may happen if a packet is retransmitted but was not really lost.

Often the transport layer service will allow the user to distinguish between the urgency of its data. Both TCP and TP4 provide a two-level discrimination scheme (*normal* and *push* for TCP, *normal* and *expedited* for TP4) that uses a logically separate channel for the pushed or expedited service. VMTP uses four levels of service: *background*, *normal*, *important*, and *urgent/emergency*. XTP has a 32-bit *sort* field used for discrimination. Packets are serviced in ascending *sort* value order. In this manner, deadlines placed in the *sort* fields of the packets will cause the packets to be serviced in nearest deadline order, although there is no provision for expired deadlines. To the author's knowledge, this *sort* field represents the largest priority space provided in any reasonably well-known transport layer protocol to date.

8.2.4. Summary of Issues

As we have seen, the service time for a task is directly related to the processing required within the task. When this processing requires requests to subsystems, a domain boundary is met with regard to scheduling. It is essential that the importance of the request to the task, as well as the importance of the task to the system, be conveyed in the request

for service from the subsystem. In this chapter we have been examining a communication subsystem as a specific example of this general problem.

Typically, functionality within the communication subsystem is modularized into layers, where each layer may provide some means of service discrimination. The challenge is to provide a uniform and consistent mechanism for:

1. carrying discrimination information between domains and layers
2. mapping one layer's mechanisms onto another's
3. providing a means for end-to-end analysis of the service time characteristics
4. maintaining the ability to provide layer-specific levels of service without using a different paradigm for discrimination

Below we consider the importance abstraction as a possible approach to this challenge.

8.3. Using Importance Functions

Importance functions provide a method for describing an ordering for a collection of tasks for every point in time based on how important each task is to the system at each point in time. The product is a schedule for the set of tasks. When communication between tasks takes place, the communication itself becomes a component of completing the task requesting or receiving the communication. Therefore, the two communicants must provide some importance information to the communication subsystem so that the communication being processed may reflect how important it is.

Since the message is a manifestation of the communication, the importance of the communication can be imposed onto the message by assigning the message some discrimination value. When the message is segmented, each packet must also be endowed with a discrimination value to allow it to compete for access to the medium and for processing within routers. In this section we address some of the issues involved in using importance functions in these places. We also look at some other uses for importance

functions within the communication subsystem. Since the subsystem is itself a processor, and the protocol procedures are really tasks to that processor, then importance functions can also be used to schedule the protocol processing. Also, importance functions can be used to help avoid congestion by considering the paths through routers as schedulable entities.

8.3.1. Messages and Importance Functions

When a task makes a request for communication it must provide, along with the message, all of the addressing and service parameters required by the subsystem. Among these parameters should be the information about how important this message is. Since the communication is a component of the processing of the task, the message's importance should be a reflection of the task's importance. One way to do this is to make known to the communication subsystem the task's importance function so the message can be ranked in the same way as the task. However, this approach may be overly simplistic since the aspects of a system that make a communication important may not necessarily be the same aspects on which the task bases its importance.

The message's importance function must reflect not only what makes the task important to the system but also what makes the message important to the task. A very important task could have unimportant communication requests, or an unimportant task could have a communication request that is very important to it. In both cases, the importance of the message must reflect its importance to the accomplishment of the system goal which, in this case, is a weighted function of how important the task is as well as how important the communication is to the task.

Consider first a system where tasks have constant importance functions such as those representing priority in Eq 5.5. Assume that each communication request is essential to the task requesting the communication. In this case the importance function of each task

can be assigned directly to each message since each component of a task is processed at the same importance value. Thus, for task i and message msg requested by task i , the importance of the message msg is given by:

$$I_{msg}(t) = I_i(t) = p_i \quad \text{Eq 8.1}$$

If the communication requests within the tasks could themselves be assigned an importance, then we can use a bi-level importance function, where the importance value of the message is a function of the importance value of the task and the criticalness of the communication:

$$I_{msg}(t) = f(p_i, crit_{msg}) \quad \text{Eq 8.2}$$

For example, assume that there are n levels of criticalness of communication. The importance function for each message would be:

$$I_{msg}(t) = p_i n + crit_{msg} \quad \text{Eq 8.3}$$

This importance function ensures that the most critical messages of the most important tasks are served preferentially.

When a task has a deadline by which it must complete its processing or the task becomes useless to the system (or even detrimental), a communication requested by the task also has a deadline of at least that applicable to the task. If a message is outstanding after the task's hard deadline is missed, the message is no longer worth processing. The communication may have a deadline prior to the task's deadline to allow for additional processing after the communication request is complete. Importance functions for messages with deadlines are similar to those given in Section 6.1.2. More generally, the importance function for the message could be a function of the importance of the task, the message's deadline, and the message's criticality:

$$I_{msg}(t) = f(I_i(t), d_{msg}, crit_{msg}) \quad \text{Eq 8.4}$$

Of course, the importance function of a message may be based on any system parameter or attribute that can be known by the communication subsystem, so Eq 8.4 could be a function of additional parameters.

Requests are ordered at the interface between the tasks using the communication subsystem and the subsystem itself according to the importance functions of the requests. Since this interface handles both send requests and receive requests, both sets of requests are competing; the interface handles the most important request at each point in time. Once the request is accepted, the message uses its importance function to contend for processing time. Since we are assuming a transport-based subsystem, the processor segments the message and prepares each constituent packet for the requests to the network layer. This processor also reassembles received packets into the whole message, so outgoing messages compete with incoming messages here as well.

8.3.2. Packets and Importance Functions

At the interface and within the transport processing the importance function has been associated with the message without having to be placed into a structure designed for transmission. Consequently, the importance function can be arbitrarily large and complex, as allowed by the generalized tasks of earlier chapters. When the message is segmented into packets, however, the discrimination information must be attached to each packet even while the packet is being transmitted over the physical medium. Conventionally a discrimination mechanism, such as a priority, is placed within a field within the packet structure, usually the header. Importance functions in their general form do not lend themselves to being placed into a fixed-size field within a packet. One solution would be to allow an arbitrary size discrimination mechanism within the packet structure so that the

whole importance function could be included within. Although this is certainly the most general solution, most modern networks attempt to minimize variable length fields. Also, recall that packets have a maximum size; it is possible that the importance function would be so large that it would not fit within an entire packet. Consequently we must consider some alternative strategies.

Instead of carrying both the function and its parameters, if every packet used the same function then only the parameters to that function would be needed. Each packet would carry the parameters, and at every decision point, the parameters would be applied to the function for the packet's importance. This is what happens with conventional priorities, where the function is actually an implicit identity function. The drawbacks to this scheme are that there can only be one function and the number of parameters must be limited. The first drawback can be remedied by making the single function a composite of several different functions, one for each class of messages. As for the second, the number of parameters is a restriction for the same reason that prevents carrying the whole importance function in each packet: the space required for the parameters must be limited. This solution is useful, therefore, if the number of parameters can be fixed and require a reasonable amount of space.

Another approach makes the first parameter be a function pointer that specifies which of several functions to use. With the function specified, the parameters that follow are specific to that function, and therefore take up less space than would the set of all parameters to a composite function. Again, a limitation is that only as many parameters as can fit within the packet may be used for any given function. Also, the set of all functions must be available at every decision point along the packet's path, meaning that copies must be kept at all routers and endpoints. However, this solution presents a high degree of flexibility while conforming to the physical constraints of packet sizes.

8.3.3. Stations and Importance Functions

In local area networks each station on a local network segment must vie for the right to transmit on that shared medium. As we have seen, the protocols for arbitrating this access are varied. Some medium access control protocols allow contention to be resolved through a series of attempts and collisions. Other protocols, however, enforce stricter access control, where the stations are ranked according to some scheme, and hence an ordering of service is produced.

In some protocols access is granted to stations in a round robin manner, with each station gaining some slot within which to transmit. Recalling Eq 5.8, and assuming that there are m stations on the local network segment, then we can construct an importance function for station s , $1 \leq s \leq m$, that provides slotted service:

$$I_s(t) = \sin\left(bt + \frac{2\pi s}{m}\right) + d \quad \text{Eq 8.5}$$

Station s gets $1/m^{\text{th}}$ of the period, which is $2\pi/b$ time units long.

Often a station is granted access according to how important the most important packet enqueued at that station is. Thus the importance function for the station is based on the importance functions of the set of packets within that station. Let I_{pkt_s} be the set of importance functions for packets at station s , then the importance function for station s is:

$$I_s(t) = f(I_{pkt_s}) \quad \text{Eq 8.6}$$

The function f is usually the maximum. Hence, each importance function is evaluated at time t and the maximum of the values is then the importance of the station:

$$I_s(t) = \max\left(I_{pkt_s}|_t\right) \quad \text{Eq 8.7}$$

8.3.4. Decision Points

One of the tenets of the importance abstraction is that the most important task is selected for processing at every point in time. In a communication subsystem, the tasks are the messages and their constituent packets. At each processing point the protocol processor chooses the most important message to process, but these processing points, or decision points, occur at discrete places throughout the path between the two communicants. At each of those decision points the importance functions can be used to ensure that the most important message is processed at the decision points. However, from the message's point of view there are periods of time during the transfer when the message is "in flight" and its importance function is inaccessible. During these times of inaccessibility the processing order established at the last decision point is maintained, even if another message becomes most important.

Decision points are largely an artifact of the distributed nature of a communication subsystem. Messages cannot be reordered while they are on the physical medium. Furthermore, the use of the physical medium for sharing station state information, such as which station has the highest priority message, implies that decisions are arrived at over a period of time during which the decision may become incorrect. The importance functions, like any discrimination mechanism, suffer from the fact that making decisions in a communication subsystem happens at discrete times; the discussion from Chapter 7 on relaxing the principle of the most important task being processed at every point in time applies here as well.

8.3.5. Other Uses for Importance Functions

Messages are the most common objects of discrimination, but there are other things within the communication subsystem that can be viewed as a system. Consider the paths

between two communicating endpoints. Each path has associated with it a rate at which the all of the protocol processors, i.e., routers and endpoints, along the path can process the flow of messages. The maximum rate along a path is the rate of the slowest protocol processor along the path. As a router is used for more than one path, the rate afforded by that router must be divided among all of the paths through that router. Consequently, a router is said to be *congested* when the rate of processing available to any one of the paths through that router becomes intolerably low.

XTP provides a mechanism by which a new path between two endpoints can be cut when the old path between them is unusable, including when the path becomes overly congested. Part of the information available to the endpoints is the rate at which packets can be forwarded on the path; this rate information is derived through normal packet exchanges, and serves to drive the rate control protocol procedure within XTP. We cite XTP here as evidence that path information such as the processing rate can be known to endpoints of the communication.

Consider an internetwork where multiple paths between any two endpoints exist, and the cost for switching paths is negligible. A set of importance functions can be used to rank the paths for use, where the most important path is the path that could offer the best service, i.e., the highest processing rate. Assume the routers dole out fractions of their rates as paths are created through them, and reclaim those fractions as paths through them are released. The rate of any path, therefore, is a function of time, $rate(t)$. A simple set of importance functions to determine which path from Host A to Host B is the best to use at any time is:

$$I_{A \rightarrow B}(t) = rate_{A \rightarrow B}(t) \quad \text{Eq 8.8}$$

Another parameter that helps determine which path is best to use is the number of buffers each router has available along the path. Assume that a router does not dedicate buffers to a particular path, but rather makes all buffers available in a first come, first served manner. A measure of buffer availability over the path can be derived by taking the minimum of the ratios of buffers in the router to paths through the router for each router along the path. Hence a path can be characterized by the router with the worst *buffer-per-path* (bpp) ratio. Since this value may also change with time, we will use the value $bpp(t)$. Now the importance function can be some function of these two parameters:

$$I_{A \rightarrow B}(t) = f(\text{rate}_{A \rightarrow B}(t), \text{bpp}_{A \rightarrow B}(t)) \quad \text{Eq 8.9}$$

The function f is determined by how much each parameter should figure into the importance of using the path.

8.3.6. Importance Functions and Levels of Service

Importance functions are ranking mechanisms, designed to allow what is important to the system to be reflected in the order in which the tasks are serviced. Importance functions, therefore, differentiate between the service given to the messages only by changing the order of processing; they do not also provide for bypassing constraints as may be allowed by various levels of service. There is no notion within the importance functions themselves for a service that can ignore, for example, flow control; such distinctions are not within the scope of the importance abstraction. The importance functions do, however, provide a mechanism for describing how important certain individual packets in a message stream are, and through that levels of service can be established.

As cited previously, Dempsey *et al.* [DEMP92] describe a service where the error recovery procedures are governed by the density of errors allowable by the application. For audio data, there can be several lost packets before a degradation of service is noticeable

by a human, so long as the errors are not too dense. Consider a set of importance functions where the parameters are carried by the packets. Assume that among these parameters is an error density measure reporting the number of known dropped packets out of the m packets preceding this one. (The value m is application-defined, reflecting the locality of the density measure.) Since each router along the path knows how many packet it had to drop as well as which packets did not arrive from the previous router, this density value is updated by each router along the path. The importance of a particular packet increases as the density of errors over the previous m packets approaches the threshold of acceptable packet loss. Hence, the connection, as represented by the aggregate of all packets that successfully traverse the network, is afforded a level of service based on the density of errors that can be withstood.

Consider another example. Assume that there exists some passive network monitor that broadcasts network statistics periodically. A communication subsystem can offer levels of service according to how much bandwidth is guaranteed to the connection. A set of importance functions associated not with packets or messages but with the connections themselves can take as a parameter the network utilization statistic. As the utilization approaches the sum of the promised bandwidths, the importance functions respond by reducing the importance of, and hence the service given to, unguaranteed connections.

8.3.7. Analysis

Recall from Chapter 4 that the relationship between the importance functions and the amount of work done to a particular task is given by:

$$w_i \Big|_{t_1}^{t_2} = \int_{t_1}^{t_2} (M(I_T) = i) dt \quad \text{Eq 8.10}$$

This equation holds for any tasks in any system—the amount of work done to the task is directly related to the amount of time that task is the most important task in the system. If we think of the communication subsystem as a system, then the relationship given in Eq 8.10 should apply to the tasks of the communication subsystem.

When a task within a station has a communication request to make to the communication subsystem, that request carries with it the message that requires transfer. We consider the message as a task of the communication subsystem and, by following the task through its stages of processing, develop the relationship between the sets of importance functions associated with the message and its constituent packets, and the work done toward completing the communication request.

Assume that a communication request i is made to the communication subsystem. Request i is for the transfer of message msg_i from this station to a remote station, possibly employing some intermediate nodes for routing. The communication subsystem processes the request i by (1) buffering message msg_i , (2) segmenting message msg_i into n constituent packets $pkt_{i,j}$, $1 \leq j \leq n$, (3) employing a packet delivery service to access the medium and deliver each of the packets to the next station along the path, (4) receiving acknowledgements and, if necessary, retransmitting lost data, and (5) indicating success or failure to the user.

Processing steps (1), (2), (4), and (5) above are performed on the message msg_i at the source station. From the point of view of the message msg_i the work done on the message in this station is the amount of time the message is the most important message in the station. Assume that I_{msg} is the set of importance functions for all messages in this station. The work done on msg_i is given by:

$$w_{msg_i} \Big|_0^\infty = \int_0^\infty (M(I_{msg}) = msg_i) dt \quad \text{Eq 8.11}$$

Note that the lower bound of 0 can be replaced with the time that request i was issued, but since message msg_i does not exist in the communication subsystem before the time of the request, the integral from 0 to the request time will be 0 anyway.

Although Eq 8.11 calculates the amount of work done on message msg_i in the source station, one cannot calculate precisely the total amount of work needed for any given message due to the unpredictable nature of losses within the communication subsystem. A message that requires retransmissions requires more total processing time. The upper bound on the integral therefore is infinity, but since practical transport protocols have some mechanism for bounding the length of time retransmissions are attempted, the upper bound can be set to a worst case time.

Since the work done on an message includes segmenting the message into one or more packets, and the delivery and acknowledgement of each packet, then we can examine the work from the point of view of a packet. For this discussion, we assume an internetworking topology as shown in Figure 8.4. In step (3) the message msg_i is segmented into n packets, $pkt_{i,j}$, for $1 \leq j \leq n$. The packet delivery service then transfers each of the packets to the destination station. To transmit packet $pkt_{i,j}$, the source station must vie for access to the medium, and packet $pkt_{i,j}$ must be the most important of all packets to transmit. Assume that each station in the internetwork is granted permission to transmit based on the importance values of the packets enqueued, as would be the case with token-passing networks. The amount of time a station s is granted permission to transmit is given by:

$$w_s \Big|_0^\infty = \int_0^\infty (M(I_{net}) = s) dt \quad \text{Eq 8.12}$$

where I_{net} is the set of importance functions for each of the m stations on the local network segment.

Assume that there are h stations along the path, including the two endpoints, and each station is granted transmission access to the medium based on the importance of the packets that station has enqueued. Two conditions must be met for packet $pkt_{i,j}$ to be transmitted: the packet must be the most important packet in station s , and station s must have the most important packet enqueued among all stations. Thus the work done on packet $pkt_{i,j}$ over all stations from 1 to h is given by:

$$w_{pkt_s} \Big|_0^\infty = \sum_{s=1}^h \int_0^\infty [(M(I_{pkt_s}) = pkt_{i,j}) (M(I_{net}) = s)] dt \quad \text{Eq 8.13}$$

where I_{pkt_s} is set of all importance functions for packets within station s .

The above equation would be accurate if, at the precise moment one packet became most important, the packet delivery service could immediately begin work on it. However, this is not the case with typical packet deliver services in real networks, since none to our knowledge have the ability to preempt one packet for another. Consequently, there is a lag between when a station has the most important packet in the local network segment, and when the station can begin to transmit that packet.

Since constituent packets of a message are essentially pipelined through the internetwork from the source station to the destination station, the work on each constituent packet is done in a concurrent fashion. Consequently, the work done on request i is not the sum of the work done on the message msg_i and each of the packets $pkt_{i,j}$, $1 \leq j \leq n$, but rather it is the sum of the time spend doing *any* work on the various parts of the request.

Assume there is no need for a retransmission and the only acknowledgement is a packet called pkt_{ack} . Also, let the source station be station 1 and the destination station be station h . The amount of work done on request i is given by summing all of the time that at least some processing of some component of the request was being performed.

$$\begin{aligned}
\lambda_{req_i} \Big|_0^\infty &= \sum_{\substack{1 \leq s \leq h \\ 1 \leq j \leq n}} \int_0^\infty [(M(I_{msg}) = msg_i) (s = 1) \\
&\vee (M(I_{pkt_s}) = pkt_{i,j}) (M(I_{net}) = s) \\
&\vee (M(I_{msg}) = msg_i) (s = h) \\
&\vee (M(I_{pkt_s}) = pkt_{ack}) (M(I_{net}) = s)] dt
\end{aligned}
\tag{Eq 8.14}$$

Expression Eq 8.14 states that the amount of work required by the request req_i is the sum of the work done at each station on the message or a constituent packet. After the message is segmented, there can be more than one station doing work toward delivering the message, therefore the amount of time spent on the message is not the sum of the work done at *each* station, but rather the sum of the work done at *any* station. Consequently, when any one component is the most important, and thereby receives work, a boolean expression indicating this will have the value of 1. By taking the “or” of all of these boolean expressions the equation sums the amount of work when at least one component is being worked on. Here, the expression is the sum over all stations, $1 \leq s \leq h$, and all packets, $1 \leq j \leq n$, of any work done at the station on this request. At the first station ($s = 1$), the message is processed if it is the most important message; at the first station and all others, the packet is processed if both the packet within the station is most important among all packets in the station, and the station is most important among all stations on the local network segment. At the destination station ($s = h$), the message is processed by reassembling the constituent packets into the message. The work on the request concludes as an acknowledgement packet is sent back to the sending station.

9 Extended Example

In this chapter we offer an example of a distributed system as viewed from one particular computer system. Communication services for the distributed system are provided by a communication subsystem as described in the last chapter. We define three types of tasks for the particular computer system of interest: a file server, a video server, and an alarm message server. All three tasks are designed to take requests from clients and send the clients the appropriate messages in response.

Once we describe the tasks and give their various definitions, we give a set of system requirements on how the tasks must be ordered for service. We develop a set of importance functions that ensure that the requirements are met. Once the tasks are represented by their respective importance functions, we can ask questions about how the system will perform under various conditions.

Next we describe the types of messages each task generates; since each task's sole purpose is message generation, the importance functions of the messages are incidentally identical to the importance functions of the tasks. Within the communication subsystem, however, the packets that comprise the messages have different constraints, and therefore must have different importance functions for use in ordering them for service. We give this set of importance functions. Finally, we give an expression for the length of time that the call used to send messages will take as a function of the type of message to be sent.

It is interesting to note that this set of tasks is heterogeneous with respect to the attributes that are considered during task scheduling. Some of the tasks are priority driven, some are periodic with real-time constraints, and some tasks must be serviced in order of

arrival. Although this example is contrived, the types of tasks are not unusual. However, attempting to find a single traditional scheduling algorithm that can satisfy all of the ordering requirements and task constraints is not intuitively obvious. Given that at least some of the tasks are periodic, rate monotonic scheduling seems to be necessary, but this policy does not adequately satisfy the needs of all of the other tasks. Furthermore, we pose and answer questions within the importance abstraction; these same questions would be difficult to pose with a more traditional scheduling approach.

9.1. Task Definitions

We make the assumption that the interface to the communication subsystem employs the transport layer data transfer mechanisms. Since the transport layer provides the mechanisms for a reliable transfer, we assume that a send request will block, awaiting the confirmation of the delivery of the message, according to the reliability semantics required of the application making the send request.

The send request `send(to_whom, type, message, ImpParms)` has four parameters. The first is the address of the recipient of the message. The second is the type of message, for use in determining the reliability semantics required. The third parameter is the message itself (normally the message is represented by a buffer descriptor but for this example we will suspend such realism). The last parameter, `ImpParms`, is used for passing importance information to the communication subsystem.

The file server task `file_server`, shown in Figure 9.1, is instantiated when a remote client system sends a message to this system requesting a file. A global flag `request_present` signals that the communication subsystem has received a message intended for the `file_server` task. When this occurs the `file_server` task issues a receive request to the communication subsystem and thereby collects the necessary

```
task file_server {
    receive(&from_whom, &filename);
    sent = false;
    to_whom = from_whom;
    size = get_buffer(filename, buffer);
    ImpParms = null;
    while (size > 0) {
        send(to_whom, "file", buffer, ImpParms);
        size = get_buffer(filename, buffer);
    }
    send(to_whom, "file", END, ImpParms);
    sent = true;
}
```

Figure 9.1 The `file_server` Task Definition

information for the transfer. This information includes the requesting client's address (`from_whom`) and the name of the file (`filename`). Note that the name of the file is the entire message from the requesting client.

The `file_server` task then retrieves the file using the `get_buffer` procedure call. Each buffer of data is then used as the message parameter in a send request. Buffers are retrieved and sent until the end of file is reached, at which time the `END` message is sent to indicate that the transfer is complete.

The task used to transfer video data is the `video_transfer` task, shown in Figure 9.2. Video data has an inherent timeliness wherein the data is useful until a deadline, and thereafter the data is useless. In this example we make several assumptions about this video transfer. First, we assume that video frames are being generated at a fixed rate of 30 per second. Second, we assume a data compression technique is used that takes six frames of data and constructs a "bundle" of the six frames at the remote client. Half of this bundle is the basis information and the other half is the change information from that basis for the six frames. Third, we assume that there is a 600 ms delay between the generation of a frame

bundle and the playback of the constituent frames. As a consequence, three bundles can be in the pipeline between their generator and their playback device. Fourth, we assume a transfer time estimate of 50 ms. This transfer time is used to help determine the deadline for sending the bundle such that there is time for the bundle to be delivered to the playback device. Finally, we assume that the video bundles are not buffered deeper than one bundle, so if another bundle is generated before the previous one has been retrieved from the buffer, the bundle is overwritten.

The task `video_transfer`, shown in Figure 9.2, loops forever, getting a bundle and sending the bundle. A global flag called `bundle_present` indicates when a bundle has been generated. This bundle is timestamped with its generation time, `bundle.gen_time`. When the bundle is retrieved using the `get_video_bundle` call, the variable `bundle_read` is set to true and `sent` is set to false. These help keep track of what has been done within this task.

```

task video_transfer {
    transfer_time = 50;
    delay = 600;
    do forever {
        bundle_read = false;
        get_video_bundle(bundle);
        bundle_read = true;
        sent = false;
        d_send = bundle.gen_time + delay - transfer_time;
        gen = bundle.gen_time;
        ImpParms = {gen, d_send};
        if (time() <= d_send) {
            send(to_whom, "video", bundle, ImpParms);
        }
        sent = true;
    }
}

```

Figure 9.2 The `video_transfer` Task Definition

The variable `d_send` is the generation time plus the delay minus the assumed transfer time. The time in `d_send` is therefore the deadline by which the send request must be given to allow the bundle time to be transferred to the playback device. If the current time in `time()` is less than or equal to the deadline to send in `d_send`, then the bundle is sent, otherwise the task drops this bundle for missing its sending deadline and gets another bundle to work on. Note that a decision was made in the design of this task to make the task wait as long as possible to send a bundle even at the expense of not retrieving a fresh bundle.

The `send_alarm` task, shown in Figure 9.3, is invoked when an alarm condition occurs within this computer system whose existence must be made known to a group of other computer systems in the distributed system. In this system there are two alarm conditions, a level 1 alarm and a level 2 alarm, where level 1 is more urgent than level 2. When one of the alarm conditions occurs, the global flag `alarm_present` signals that notification of this alarm condition must be made: the `send_alarm` task is thus instantiated. The `get_alarm` call retrieves the information about the alarm condition, namely the group to notify, the alarm message, and the level of the alarm condition. The send request then is used to notify the group of recipients of the alarm condition.

```

task send_alarm {
    get_alarm(&group_to_whom, alarm_msg, level);
    current_time = time();
    sent = false;
    ImpParms = {level, arrival_time};
    send(group_to_whom, "alarm", alarm_msg, ImpParms);
    sent = true;
}

```

Figure 9.3 The `send_alarm` Task Definition

We can use the task definitions given to estimate the amount of work required by each task. Here we assume that system calls account for a substantial amount of the processing time, so the length of these calls is used in determining the estimated work requirements. We use len_{sys_call} to represent the amount of time required by the system call. Since the amount of time required for a `send` system call depends on the error correction semantics required by the type of message being sent, we use $len_{send}(type)$ to represent the length of time required for sending this type of message.

For the `file_transfer` task, the work starts when a request for a file is received. There are a number of `get_buffer` system calls and `send` system calls, then one last `send` system call to indicate the end of the file. If $buffer_size$ is the size of the buffer used in `get_buffer`, and $file_size$ is the size of the file, then the following shows the work required by the `file_transfer` task:

$$w_{file} = len_{receive} + \left\lceil \frac{buffer_size}{file_size} \right\rceil (len_{get_buffer} + len_{send}(file)) + len_{send}(file) \quad \text{Eq 9.1}$$

Since the `video_transfer` task loops forever, the work required for it cannot be measured. However, the work required from the generation of a bundle until the generation of the next bundle can be calculated, as follows:

$$w_{video} \Big|_{gen}^{gen+200} = len_{get_video_bundle} + len_{send}(video) \quad \text{Eq 9.2}$$

Finally, the work required by the `send_alarm` task is given by:

$$w_{alarm} = len_{get_alarm} + len_{send}(alarm) \quad \text{Eq 9.3}$$

9.2. Task Level Scheduling

In determining the importance functions for each of these tasks, the system specifications require that the following be met:

1. Sending a level 1 alarm message is the most important task in the system.

2. Level 1 alarm messages must be ordered first come, first served.
3. A video bundle must be sent while it is still useful unless a level 1 alarm causes this deadline to be missed.
4. All level 2 alarms are equally important.
5. Sending a level 2 alarm message is more important than sending a video bundle as long as the video bundle has at least 225 ms before its deadline, otherwise the video bundle is more important.
6. Retrieving a video bundle is increasingly important as the generation of a new bundle approaches, but it is never more important than sending either a level 1 or level 2 condition.
7. Serving the file transfer request is strictly a background task.

The following importance functions are designed to meet these requirements. For the file transfer we define a function that assigns the importance value of 1 to the task when either there is a request for a file transfer present or the current request has not been completely fulfilled, and 0 otherwise:

$$I_{file}(t) = \begin{cases} 1, & \text{if request_present or not sent} \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 9.4}$$

Since file transfer is strictly background, the importance value 1 forms the basis to which other importance function values will be compared.

The importance function for the `video_transfer` task must represent the timeliness of the video data, both while the data is awaiting retrieval and, after retrieval, while it is waiting to be sent. The `video_transfer` task must issue the send request for a bundle while it is still possible for that bundle to meet its deadline. Consequently, the deadline for sending the bundle, that is, `d_send`, takes into account the estimated transit time, and the importance function uses this to help order video transfers according to which transfer more urgently needs service in order to meet its sending deadline. Also, when a bundle is waiting to be retrieved, the importance function must express the urgency of retrieving this bundle before the next bundle becomes available. The following importance function meets these requirements:

$$I_{video}(t) = \begin{cases} 2 + \frac{2(t - \text{gen})}{d_send - \text{gen}}, & \text{if bundle_read and not sent} \\ 2 + \frac{(t - \text{gen})}{200}, & \text{if video_present and not bundle_read} \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 9.5}$$

Since there may be more than one `video_transfer` task activated, the task with the nearest deadline should be serviced first. Hence this importance function increases monotonically in value from 2 to 4 as the video transfer task awaits sending a bundle. Once the deadline `d_send` is missed, the task cannot base its importance function on sending the bundle, and instead must concentrate on reading a fresh bundle. If the bundle has not yet been retrieved, the function increases monotonically from 2 to 3 awaiting bundle retrieval. Since bundles are generated every 200 ms, the effective deadline for the bundle currently waiting to be read is its generation time plus 200 ms. Once this deadline is passed, the task bases its importance on the age of the new bundle waiting to be retrieved. When there is nothing to do (the read bundle has been sent and no new bundle has been generated) the task has an importance value of 0.

The importance function for the `send_alarm` task must ensure that the sending of a level 1 alarm message is the most important task in the system, and that all of the `send_alarm` tasks sending level 1 messages are ordered according to alarm arrival. In addition, the importance function for this task must also ensure that requirement 3 above is satisfied; that is, sending a level 2 alarm is more important than servicing a video bundle unless the video bundle is within 225 ms of its deadline, in which case servicing the task with the video bundle is more important. The importance function below offers one solution to these requirements.

$$I_{alarm}(t) = \begin{cases} 5 + \frac{t}{arrival_time}, & \text{if level} = 1 \text{ and not sent} \\ 3, & \text{if level} = 2 \text{ and not sent} \\ 0, & \text{otherwise} \end{cases} \quad \text{Eq 9.6}$$

Sending a level 1 alarm message will always be the most important task in the system, as per requirement (1). Multiple level 1 alarm messages will be sent in a first come, first served manner since the importance function monotonically increases with the age of the `send_alarm` request. This fulfills requirement (2). Since all tasks sending level 2 alarms have the same importance value, no level 2 alarm message is ever more important than another, as stated in requirement (4). Sending a video bundle is less important than sending a level 2 alarm message until the bundle's age is within 225 ms of its deadline to send, at which point the importance of sending the video bundle becomes greater than the importance of sending a level 2 alarm message. This satisfies requirement (5). Retrieving a video bundle is never more important than sending a level 1 or level 2 alarm message since the maximum value of its importance function less than or equal to the minimum importance value for either level alarm message, as required by (6). Requirement (7) is met since the maximum importance value attainable by a file transfer task is strictly less than the minimum importance value attainable by any other type of task.

At this point we can ask various questions about how the system will behave given the importance functions for the various tasks. The following questions are just samples of some of the things system designers may want to know; the point is that we can ask questions regarding when certain tasks will receive service by examining the importance functions for the tasks. Some answers we derive are in the form of the conditions that are necessary for the situation described in the question to occur; others are limits or other values associated with the system.

Since the `video_transfer` task has real-time constraints on the delivery of the video data, a natural question is “Under what conditions will a `video_transfer` task fail to retrieve a ready bundle?” We know by examination that the `video_transfer` task’s importance function will always have a value greater than any `file_transfer` task. The question, then, becomes “Under what conditions is $I_{video}(t) > I_{alarm}(t)$?” By examination it is obvious that $I_{video}(t) \leq I_{alarm}(t)$ as long as the `video_transfer` task is waiting to retrieve a ready bundle. The answer is that the `video_transfer` task will miss a bundle if there are n_{alarm} `send_alarm` tasks over a 200 ms period, where n_{alarm} is given by:

$$n_{alarm} = \left\lceil \frac{200 - (len_{get_video_bundle} + len_{send}(video))}{len_{get_alarm} + len_{send}(alarm)} \right\rceil \quad \text{Eq 9.7}$$

Similarly we can ask “Under what conditions will a `video_transfer` task miss a deadline to send?” The answer now must take into consideration the fact that $I_{video}(t)$ is greater than $I_{alarm}(t)$ if the `send_alarm` task has a level 2 alarm and the `video_transfer` task has a bundle to send and has waited longer than 225 ms. In order to make the deadline to send, the `video_transfer` task must start the send call within 550 ms of the bundle’s generation. Consequently, there must be more than n_{alarm} `send_alarm` tasks handling any alarm message over the first 225 ms, and m_{alarm} `send_alarm` tasks handling level 1 alarm messages over the second 225 ms. The values of n_{alarm} and m_{alarm} are given below:

$$n_{alarm} = \left\lceil \frac{225 - len_{get_video_bundle}}{len_{get_alarm} + len_{send}(alarm)} \right\rceil \quad \text{Eq 9.8}$$

$$m_{alarm} = \left\lceil \frac{225 - len_{get_video_bundle}}{len_{get_alarm} + len_{send}(alarm)} (\text{level} = 1) \right\rceil \quad \text{Eq 9.9}$$

It may be useful to know the physical limitations on the number of `video_transfer` tasks that can be simultaneously running. If no more important tasks are running, the answer is the number of `video_transfer` tasks that can be executed within 200 ms:

$$n_{video} = \left\lfloor \frac{200 - (len_{get_video_bundle} + len_{send}(video))}{len_{get_video_bundle} + len_{send}(video)} \right\rfloor \quad \text{Eq 9.10}$$

To answer “Under what conditions will a `send_alarm` task servicing a level 2 alarm receive service?” we look at each task’s importance function to determine what conditions lead to $I_{alarm}(t)$, `level = 2`, having the highest value; since a `send_alarm` task processing a level 2 alarm always has an importance value of 3, we can say that the `send_alarm` task j will be most important if, for the task set T at some time t :

$$\max(I_i(t) |_{t} < 3) (\forall i \in T, i \neq j) \quad \text{Eq 9.11}$$

The `file_transfer` task is, by inspection, always less important than any other type of task. For the `video_transfer` task, however, there are two cases. First, if `bundle_read` and `not sent` are both true, then:

$$I_{video}(t) < 3 \quad \text{Eq 9.12}$$

$$2 + \frac{2(t - \text{gen})}{d_send - \text{gen}} < 3 \quad \text{Eq 9.13}$$

$$t < \text{gen} + 225 \quad \text{Eq 9.14}$$

The second case is if `video_present` and `not read` are both true:

$$I_{video}(t) < 3 \quad \text{Eq 9.15}$$

$$2 + \frac{t - \text{gen}}{200} < 3 \quad \text{Eq 9.16}$$

$$t < \text{gen} + 200 \quad \text{Eq 9.17}$$

So the conditions for a `send_alarm` task processing a level 2 alarm to be most important in the presence of either `file_transfer` tasks or `video_transfer` tasks is:

$$\begin{aligned} & ((t < \text{gen} + 225) \wedge \text{bundle_present} \wedge \text{not read}) \\ \vee & ((t < \text{gen} + 225) \wedge \text{video_present} \wedge \text{not sent}) \end{aligned} \quad \text{Eq 9.18}$$

If any level 1 alarms are present, the level 2 alarm will be preempted. If another level 2 alarm is present, there is no guarantee that the `send_alarm` task j will be the one to receive service since requirement (4) says all level 2 alarms are equally important.

9.3. Message Level Scheduling

Each task must supply some importance information with the message when it makes a send request. As discussed in the previous chapter, the importance of the communication request and its manifestation, the message, are largely dependent on the importance of the task issuing the request: the importance of the message should be some function of the importance of the task issuing the request to send the message. In general the importance of the message also depends on how important the communication is to the task, not just how important the task is.

We have constructed tasks here whose sole purpose is to service various requests for sending messages: the file server task responds to the file requests, the video server supplies a stream of video frames, and the alarm server responds to alarm conditions. The importance values of the messages generated, therefore, are directly related to how important the tasks are. Because of how tightly the tasks are coupled to the sending of messages, it follows that we can use the tasks' importance functions for the importance functions of the messages generated by the tasks.

9.4. Packet Level Scheduling

The messages are given to the communication subsystem for processing. This processing for each message includes segmentation of the message into packets, sending the packets across the internetwork to the destination station, reassembling the packets into the original message, and acknowledging receipt if required. When the message is segmented into packets, the packets are sent from the source station through various routers along the path to the destination station. Each packet must contend with all other packets at each station for both buffer space and processing time. The most important packets are preferred; the least important packets are dropped if buffer space is unavailable. We must therefore design a set of importance functions for the packets such that their importance values reflect the importance of their contents.

In this example the file transfer messages are segmented into as many packets as are required. In keeping with the notion that file transfer is strictly background, these packets are given an importance function that returns the lowest value among importance functions for packets with other types of contents. The video transfer bundle is segmented into two packets, one holding the basis information for the six frames, and the other one holding the change from the basis for the six frames. The content of the former packet is called the *basis*, and the content of the latter packet is called the *delta*. The basis packet is more important than the delta packet. Furthermore, it is necessary that the density of lost basis packets be reduced. We include in the importance functions a parameter called `dropped` that holds the number of basis packets dropped over the last ten basis packets sent.

Each alarm message fits into a single packet. The level 1 alarm packet is the most important packet in the system. The level 2 alarm packet fits in the middle of the importance space, as did the task that generated it.

Below we give the importance function that meets the need for ordering the packets:

$$I_{pkt}(t) = \begin{cases} 1, & \text{if } type = \text{file} \\ 2, & \text{if } type = \text{delta} \\ 3, & \text{if } type = \text{level 2 alarm} \\ 4 + \frac{\text{dropped}}{10} - \frac{2(d-t)}{200}, & \text{if } type = \text{basis} \\ 6, & \text{if } type = \text{level 1 alarm} \end{cases} \quad \text{Eq 9.19}$$

These data-bearing packets are not the only packets associated with the various communications, however. According to the error control semantics for each type of communication, there may be positive or negative acknowledgements. In this example, all of the messages require a positive acknowledgement except the video bundle message; reports about basis and delta packets comprising the bundle are generated only if the packets are known to be missing. Since acknowledgement packets are as important as the packets that they acknowledge, the importance functions associated with the acknowledgement of each packet type are virtually identical to the importance functions for the packets themselves, with a few minor exceptions. Below we give this importance function:

$$I_{pkt}(t) = \begin{cases} 1, & \text{if } type = \text{file} \\ \left. \begin{array}{l} 2, \text{ if } (t < d - 50) \\ 0, \text{ otherwise} \end{array} \right\}, & \text{if } type = \text{delta} \\ 3, & \text{if } type = \text{level 2 alarm} \\ \left. \begin{array}{l} 4, \text{ if } (t < d - 50) \\ 0, \text{ otherwise} \end{array} \right\}, & \text{if } type = \text{basis} \\ 6, & \text{if } type = \text{level 1 alarm} \end{cases} \quad \text{Eq 9.20}$$

Now that we know the importance function for each packet, we can invoke Eq 8.14 to give an expression for the length of time required for a send call according to the type of message being sent:

$$len_{\text{send}}(\text{type}) = w_{\text{msg}_{\text{type}}} = w_{\text{msg}_{\text{type}}}\Big|_0^{\infty} \quad \text{Eq 9.21}$$

Given that h is the number of hops in the path between the server and the client, s designates the station, n is the number of packets comprising a message of this type, I_{net} is the set of importance functions for the stations on a given network segment (here each station is as important as its most important packet), and I_{pkt_s} is the set of importance functions for the set of packets present at station s , the work done on a message of a particular type is given by:

$$\begin{aligned} w_{\text{msg}_{\text{type}}}\Big|_0^{\infty} &= \sum_{\substack{1 \leq s \leq h \\ 1 \leq j \leq n}} \int_0^{\infty} [(M(I_{\text{msg}}) = \text{msg}_{\text{type}}) (s = 1) \\ &\vee (M(I_{\text{pkt}_s}) = \text{pkt}_{\text{type},j}) (M(I_{\text{net}}) = s) \\ &\vee (M(I_{\text{msg}}) = \text{msg}_{\text{type}}) (s = h) \\ &\vee (M(I_{\text{pkt}_s}) = \text{pkt}_{\text{ack},\dots}) (M(I_{\text{net}}) = s)] dt \end{aligned} \quad \text{Eq 9.22}$$

This expression states that the amount of work required by a message of a certain type is a sum of the boolean conditions. This expression, like Eq 8.14, takes into account the fact that work on the message is really the sum of the work done on each constituent packet anywhere in the system—the amount of work required for the message is the “or” of any work done throughout the system on any of the constituent packets.

10 Conclusions

The *importance abstraction* is a general framework for expressing scheduling policies. The framework is general in that the scheduling algorithm does not vary with the policy—the scheduler chooses the most important task at every point in time. Each task has associated with it a function that profiles that task’s importance to the system over time. The function, called an *importance function*, reflects the task’s importance by taking as parameters all of the task attributes and system characteristics that may cause the task to become more or less important to the system. As the system’s characteristics change over time, the task that is most important to the system may change as well.

The importance abstraction is a new way to express scheduling problems. It places the emphasis on individual tasks and what makes them important to the system, rather than fitting a task set onto a well-known algorithm in order to use its analytical results. Since the importance abstraction expresses the scheduling problem in terms of what tasks are most important, a wide range of problems can be presented under a unified abstraction and analyzed using similar tools.

Traditional scheduling algorithms are easily emulated within this abstraction by creating importance functions that cause particular tasks to become most important at precisely the same instant that the scheduling algorithm would have chosen that task for service. In addition to these traditional scheduling policies, novel scheduling policies can also be easily expressed. These novel scheduling policies include scheduling heterogeneous task sets and tasks that are dependent on continuously updated variables as parameters.

Since the scheduling policies are expressed in terms of sets of functions, these sets may be manipulated and analyzed using mathematical techniques. In addition to the flexibility and intuitiveness of expressing scheduling requirements in terms of functions, functional analysis can now be employed to help answer questions about the schedules and how the system would respond under various circumstances. The scheduling problem is therefore moved from the traditional algorithmic domain to the functional domain, and mature analytical tools can be employed.

Even though the importance abstraction relies on evaluating every importance function at every point in time, there are certain classes of importance function sets that allow us to relax this requirement. Scheduling policies whose importance function representations belong to these classes can be implemented in an efficient manner using functions to drive the scheduling.

10.1. Summary of Work

The goal of this research was to set forth a method for describing scheduling problems using a function-based technique. To gauge its usefulness we had to show that this new method expressed both traditional and novel scheduling policies, that it was conducive to analysis in that questions may be posed and answered about the schedules produced, and that this method, with possible restrictions, could be implemented efficiently. We have shown that the importance abstraction can express a representative group of traditional scheduling policies. We have analyzed a set of policies called static rankings which include nearest deadline first, and proved several facts about nearest deadline first. We continued by relaxing certain restrictions on the nearest deadline first policy, hence creating new policies, and also proved or developed expressions for some of their properties. We also explored the implementability of the importance abstraction, suggesting restrictions that would aid in reducing the work for the scheduler.

We then examined an example system, the communication subsystem. The importance abstraction was found to be useful in describing discrimination policies within the subsystem. The importance abstraction can also aid in unifying the discrimination policies across layer boundaries, and across the boundary between the user and the communication subsystem.

We concluded this work with an extended example of a distributed system wherein one computer system in particular had three types of message server tasks. We designed importance functions for these tasks that were able to meet the system specifications and, using these importance functions, posed and answered questions concerning the conditions under which certain events would occur. We note that the set of tasks were heterogeneous in nature, and scheduling with traditional policies would have been more difficult.

The importance abstraction represents an new approach to expressing scheduling policies that is intuitive and conducive to analysis. Emphasis was placed on the individual characteristics of each task, yet the scheduling algorithm remains constant. We have demonstrated the usefulness of the importance abstraction through the analysis of a traditional scheduling policy, through the examination of issues within the communication subsystem, and through an extended example where tasks, messages, and packets are all scheduled using the same approach.

10.2. Contributions

This dissertation makes six points of contribution. First, we have developed a general framework for the expression and analysis of scheduling. Within our framework we can express a wide range of scheduling policies. Since we cast the scheduling constraints and conditions into a set of functions, we can emulate traditional scheduling policies simply by constructing functions that take as parameters the same attributes that the traditional

scheduling policies use. Within the aegis of the importance abstraction we can express such scheduling policies as rate monotonic and least slack time. Since each task is assigned an importance function, heterogeneous task sets are scheduled using a consistent approach. For example, minimizing response times for non real-time tasks while meeting deadline for deadline-driven tasks requires special adaptations of the rate monotonic policy; within the importance abstraction, no special mechanisms are employed to handle this heterogeneous task set.

Second, the importance abstraction allows one to consider the problem of scheduling tasks by focusing on what makes each task important to the system. By answering the question “Under what conditions should this task be the most important task in the system?” the construction of the importance functions is a more intuitive exercise than discovering the algorithm that both fits the general need of the system and produces the desired scheduling results. In this respect, non-traditional (novel) scheduling policies can be employed.

Third, once the scheduling problem is expressed in terms of a set of functions, analysis of scheduling using mathematical tools and techniques follow. By expressing the scheduling policy in terms of functions, we move from an algorithmic analysis to functional analysis. In the functional domain we can decouple the actual problem from its representation, and use the proof techniques of mathematics to attain scheduling results.

Fourth, we explore the issues involved in using the importance abstraction as a tool for efficiently implementing scheduling policies. The framework is based on continuous evaluation of each importance function, yet computer systems are by nature discrete machines. Either there is inherent in the set of importance functions obvious discrete evaluation times, or there will be a degree of imprecision introduced. The granularity of the imprecision, if significant, must be accounted for within the analysis.

Fifth, we have applied this framework to an application: the communication subsystem. In doing this we have shown that the importance abstraction is a general approach to scheduling, even in applications where scheduling is not necessarily a traditional concern.

Finally, we have shown by extended example that the importance abstraction provides a consistent mechanism by which scheduling concerns can cross domain boundaries. In this example we examine tasks within a system attached to a distributed system. We pose and answer questions using the framework provided by the importance abstraction, and show that end-to-end scheduling concerns can be addressed since all levels of scheduling employ the same approach.

10.3. Future Research

Although we have shown the usefulness of the importance abstraction as outlined above, there are aspects of the importance abstraction that require further research. Several of these are enumerated here.

The *defining property* as described in Chapter 4 is the necessary and sufficient conditions for a set of importance functions to impose a schedule that meets the system goal. A defining property can be used to determine if any given set of importance functions is a member of the equivalence class of sets of importance functions that meet the system goal. We have not yet explored the issues involved in producing the defining property from a given system goal. We will seek to characterize the system goals for which defining properties can be ascertained. In some cases the sufficient conditions will produce an adequate test; we seek to characterize these instances as well.

We make the claim that the importance abstraction is general in that it can emulate all traditional scheduling policies. We will try to prove coverage. It has been suggested that

this approach cannot handle the paging scheduling policies for which anomalies occur. Whether the importance abstraction provides complete coverage is still an open question.

We have shown how to express the work done to a task even if there are several concurrent processors. We seek to apply the importance abstraction to multiprocessor and multiple resource problems.

In the importance abstraction, the complexity of the scheduling problem is moved from the algorithm into the functions. Although we only showed importance function representations of polynomial-time scheduling problems, the importance functions can just as well express non-polynomial-time scheduling problems. It would be interesting to reprove results about NP-complete scheduling problems using this new framework.

We have examined only preemptable tasks here. With non-preemptable tasks, the greedy solution is often a pitfall. Since the importance abstraction is at its heart a greedy algorithm, we seek to explore how to schedule non-preemptive tasks within this framework.

Bibliography

- [ANSI86] American National Standards Institute, "FDDI Token Ring Media Access Control Standard," *Draft proposed Standard X3T9.5/83-16, Rev. 10*, February 1986.
- [BERN71] Bernstein, A. J. and Sharp, J. C., "A Policy-Driven Scheduler for a Time-Sharing System," *Communications of the ACM*, Vol. 14, No. 2, pp. 74-78 (February 1971).
- [BIYA88] Biyabani, S. R., Stankovic, J. A. and Ramamritham, K., "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama, pp. 152-160 (December 6-8, 1988).
- [CCITT84] Comité Consultatif International de Télégraphique et Téléphonique, "The X.25 Packet Layer Protocol," 1984.
- [CHEN88] Cheng, S., Stankovic, J. A. and Ramamritham, K., "Scheduling Algorithms for Hard Real-Time Systems — A Brief Survey," in **Hard Real-Time Systems**, Stankovic, J. A. (ed.), IEEE Computer Society Press, August 1988.
- [CHER88] Cheriton, D. R., "VMTP: Versatile Message Transaction Protocol, Protocol Specification Preliminary Version 0.7," Computer Science Department, Stanford University, February 22, 1988.
- [CLAR87] Clark, D. D., Lambert, M. L. and Zhang, L., "NETBLT: A Bulk Data Transfer Protocol," Network Information Center RFC 998, SRI International, March 1987.
- [CONW67] Conway, R. W., Maxwell, W. L. and Miller, L. W., **Theory of Scheduling**, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1967.
- [DARPA81a] Postel, J., ed., "Internet Protocol - DARPA Internet Program Protocol Specification," RFC 791, USC/Information Sciences Institute, September 1981.
- [DARPA81b] Postel, J., ed., "Transmission Control Protocol - DARPA Internet Program Protocol Specification," RFC 793, USC/Information Sciences Institute, September 1981.
- [DEMP92] Dempsey, B. J., Strayer, W. T., and Weaver, A. C., "Adaptive Error Control for Multimedia Data Transfers," *Proceedings of the International Workshop*

on Advanced Communications and Applications for High Speed Networks, Munich, Germany, March 16-19, 1992.

- [DHAL78] Dhall, S. K. and Liu, C. L., "On a Real-Time Scheduling Problem," *Operations Research*, Vol. 26, No. 1, pp. 127-140 (January-February 1978).
- [FERR90] Ferrari, D., "Client Requirements for Real-Time Communication Services," *IEEE Communications Magazine*, Vol. 28, No. 11, pp. 65-72 (November 1990).
- [GAIT90] Gaitonde, S. S., Jacobson, D. W. and Pohm, A. V., "Bounding Delay on a Multifarious Token Ring Network," *Communications of the ACM*, Vol. 33, No. 1, pp. 20-28 (January 1990).
- [GARE79] Garey, M. R. and Johnson, D. S., **Computers and Intractability, A Guild to the Theory of NP- Completeness**, W. H. Freeman and Company, New York, 1979.
- [GOOD88] Goodenough, J. B. and Sha, L., "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," Technical Report CMU/SEI-88-SR-4, Carnegie-Mellon University Software Engineering Institute, March 1988.
- [IEEE85a] Institute of Electrical and Electronics Engineers, "IEEE Standard 802.4 Token-Passing Bus Access Method and Physical Layer Specifications," 1985.
- [IEEE85b] Institute of Electrical and Electronics Engineers, "IEEE Standard 802.5 Token Ring Access Method and Physical Layer Specifications," 1985.
- [IEEE85c] Institute of Electrical and Electronics Engineers, "IEEE Standard 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications," 1985.
- [IEEE89] Institute of Electrical and Electronics Engineers, "IEEE 802.6 Proposed Standard: Distributed Queue Dual Bus Metropolitan Area Network," November 30, 1989.
- [ISO7498] International Organization for Standardization, "Information Processing Systems - Open Systems Interconnection - Basic Reference Model," *Draft International Standard 7498*, October 1984.
- [ISO8073] International Organization for Standardization, "Information Processing Systems - Open Systems Interconnection - Transport Protocol Specification," *Draft International Standard 8073*, July 1986.
- [JENS85] Jensen, E. D., Locke, C. D. and Tokuda, H., "A Time- Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the Real-Time Systems Symposium*, pp. 112-122 (December 3-6, 1985).

- [KLEI75] Kleinrock, L., **Queueing Systems-Volume 1: Theory**, John Wiley & Sons, New York, 1975.
- [LEHO87] Lehoczky, J. P., Sha, L. and Strosnider, J. K., "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings of the 1987 IEEE Real-Time Systems Symposium*, San Jose, California, pp. 261-270 (December 1-3, 1987).
- [LIU73] Liu, C. L. and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61 (January 1973).
- [LOCK86] Locke, C. D., "Best-Effort Decision Making for Real-Time Scheduling," Dissertation (Computer Science Report No. CMU-Computer Science-86-134), Carnegie-Mellon University Department of Computer Science, May 1986.
- [LOCK88] Locke, C. D. and Goodenough, J. B., "A Practical Application of the Ceiling Protocol in a Real-Time System," Technical Report CMU/SEI-88-SR-3, Carnegie-Mellon University Software Engineering Institute, March 1988.
- [METC76] Metcalfe, R. M. and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Vol. 19, No. 7, pp. 395-404, (July 1976).
- [PEDE88] Peden, J. H. and Weaver, A. C., "The Utilization of Priorities on Token Ring Networks," *Proceedings of the 13th Conference on Local Computer Networks*, Minneapolis, Minnesota, pp. 472-478 (October 10-12, 1988).
- [PEI92] Protocol Engines, Inc., "XTP Protocol Definition, Rev 3.6," PEI 92-10, January 1992.
- [RUSC77] Ruschitzka, M. and Fabry, R. S., "A Unifying Approach to Scheduling," *Communications of the ACM*, Vol. 20, No. 7, pp. 469-477 (July 1977).
- [SAE87] Society of Automotive Engineers, "SAE AS4074.2 High Speed Ring Bus, Final draft Standard," June 1987.
- [SHA86] Sha, L., Lehoczky, J. and Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proceedings of the 1986 IEEE Real-Time Systems Symposium*, New Orleans, Louisiana, pp. 181- 191 (December 2-4, 1986).
- [SHA87] Sha, L., Rajkumar, R. and Lehoczky, J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Technical Report CMU-Computer Science-87-181, Carnegie-Mellon University, Computer Science Department, 1987.
- [SHA90] Sha, L. and Goodenough, J. B., "Real-Time Scheduling Theory and Ada," *IEEE Computer*, Vol. 23, No. 4, pp. 53-62 (April 1990).

- [SPRU88] Sprunt, B., Lehoczky, J. and Sha, L., "Exploiting Unused Periodic Time For Aperiodic Service Using The Extended Priority Exchange Algorithm," *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama (December 6-8, 1988).
- [STRA91] Strayer, W. T., "A Study of Preemptable vs. Non-Preemptable Token Reservation Access Protocols," *Computer Communication Review*, Vol. 21, No. 2, pp 71-80 (April 1991).
- [TOKU87] Tokuda, H., Wendorf, J. W. and Wang, H., "Implementation of a Time-Driven Scheduler for Real- Time Operating Systems," *Proceedings of the 1987 IEEE Real-Time Systems Symposium*, San Jose, California, pp. 271-280 (December 1-3, 1987).
- [TOKU89] Tokuda, H., Mercer, C. W. and Ishikawa, Y., "The ARTS Distributed Real-Time Kernel and its Toolset," Report, 1989.
- [WEND88] Wendorf, J. W., "Implementation and Evaluation of a Time-Driven Scheduling Processor," *Proceedings of the 1988 IEEE Real-Time Systems Symposium*, Huntsville, Alabama, pp. 172-180 (December 6-8, 1988).

Appendix A

In Chapter 2 we discussed an inherent attribute of a task, called the task *length*. The task length is the amount of time required for a processor to complete the task, including securing any additional resources, scheduling this and all other active tasks, and other associated latencies, such as context switching. Here we expand upon the discussion of the task length by providing a notation and an algebra for task lengths.

Define w_i as the amount of processing time required by task i . For some tasks it may be possible to calculate or estimate the amount of processing time required; for others it may not be possible to know the processing time requirement until the task finally completes. The quantity w_i has a definite value; however, that value may not be known *a priori*.

Define the amount of work done on task i over the interval (a, b) as $w_i|_a^b$. Since the work on a task cannot exceed the time allotted for that work,

$$w_i|_a^b \leq b - a \quad \text{for } (b \geq a). \quad \text{Eq A.1}$$

For the degenerate case¹ of $b < a$, $w_i|_a^b = 0$.

Assume that task 1 is processed until completion. Then, $w_i = w_i|_0^\infty$. Furthermore, the sum of the work done to a task before some point in time t and the work done after that point t is the total work done to the task:

$$w_i = w_i|_0^\infty = w_i|_0^t + w_i|_t^\infty \quad \text{Eq A.2}$$

¹. The normal case is assumed for the rest of this discussion.

When a task i has a deadline d_i , then

$$w_i \Big|_a^b \leq \min(d_i, b) - a \quad \text{Eq A.3}$$

Consider two tasks, i and j , receiving work over some interval a to b ,

$$w_i \Big|_a^b + w_j \Big|_a^b \leq b - a \quad \text{Eq A.4}$$

For some set of tasks T receiving work over the interval a to b ,

$$\sum_{i: \tau_i \in T} w_i \Big|_a^b \leq b - a \quad \text{Eq A.5}$$

For tasks i and j considered over the intervals a to b and a to d respectively,

$$w_i \Big|_a^b + w_j \Big|_a^d \leq \max(b, d) - a \quad \text{Eq A.6}$$

Considering these two tasks over the intervals a to b and c to b ,

$$w_i \Big|_a^b + w_j \Big|_c^b \leq b - \min(a, c) \quad \text{Eq A.7}$$

For task i over the interval a to b and task j over the interval c to d , the following is true:

$$w_i \Big|_a^b + w_j \Big|_c^d \leq \max(b, d) - \min(a, c) \quad \text{Eq A.8}$$

The quantity $w_i \Big|_a^b$ is either (1) known, (2) derived, (3) assigned, or (4) unknown. At time t , $w_i \Big|_0^t$ is *known* since the amount of work done for task i at time t is known at time t . Also at time t , $w_i \Big|_t^\infty$ is *derived* as $w_i - w_i \Big|_0^t$. If it is determined that from time a to time b task i will get 3 time units of work (for $b - a \geq 3$) then $w_i \Big|_a^b = 3$ is assigned. Otherwise, $w_i \Big|_a^b$ is unknown. If $w_i \Big|_a^c$ is known, derived, or assigned, then

$$w_i \Big|_a^b - w_i \Big|_a^c = w_i \Big|_c^b \quad \text{for } (b \geq c) . \quad \text{Eq A.9}$$