Slotted Priorities:
Supporting Real-Time Computing Within General-Purpose Operating Systems

by

Gregory Bollella

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1997

Approved by:

Kevin Jeffay

F. Donelson Smith

Peter Calingaert

James P. Gray

ABSTRACT

Slotted Priorities:
Supporting Real-Time Computing
Within General-Purpose Operating Systems

Recent advances in network technologies, processor capabilities, and microcomputer system hardware, coupled with the explosive growth of the Internet and on-line data access, have created new demands on personal computer operating systems and hardware. In large part, these demands are for the ability to acquire, manipulate, display, and store multimedia data. The computational processes that successfully acquire and display multimedia data necessarily have deadlines. That is, the computation must be complete before a specified point in time. Currently, no general-purpose operating systems support such real-time processes. We have developed a software architecture, called slotted priorities, that defines a way to add support for real-time computation to existing general-purpose operating systems for uniprocessor machine architectures. The slotted priorities architecture shares the resources of a computing system between a general-purpose operating system and a real-time kernel. Software components called executives manage how an instance of a resource is shared. Executives ensure that the RTK can gain access to the resource at precise times. The resulting operating system will be able

to guarantee that certain computations will *always* complete before their stated deadline. The modifications to the general-purpose operating system are modest.

The architecture is comprised of a resource model, an execution model, and a programming model. The resource model is a classification of resources according to characteristics relevant to the sharing of the resources between the real-time kernel and the general-purpose operating system. The execution model defines how real-time tasks acquire the processor. The programming model defines how programmers write and think about real-time programs for an implementation of the slotted priorities architecture. Finally, we develop a feasibility test which can determine if a set of periodic real-time threads will all meet their deadlines when executed on a system implementing this architecture.

We describe an implementation of the architecture and a set of experiments that validate the implementation. Two real-time demonstration applications were built and executed on the test implementation. Results and analysis of those applications are also presented.

ACKNOWLEDGMENTS

I would like to thank my wife, Paula Keith, and daughter, Alex, for all of their support and patience.

The members of my committee have all been tremendously helpful and deserve thanks for all of their effort. Kevin Jeffay, my principal advisor, has been very helpful by guiding me through the real-time literature and being a willing sounding board for all of my ideas. Jim Gray helped me with some of the basics, Peter Calingaert provided encouragement to excel in all areas. Don Smith helped me see my work from a practical viewpoint. Don Stanat helped my effort with his experience, often offering wise advice. Jim Anderson's research in wait-free objects simplified the description of part of my architecture.

My IBM managers during this project, April Singer, Diane Pozefsky, and Mark Pozefsky, all helped me by providing time when I needed it to tackle some of the bigger portions of this work in contiguous chunks. Mark Pozefsky, my second line manager at IBM during the implementation portion of this work, provided equipment and encouragement. Many of my co-workers at IBM helped by allowing me to explain to them some of the ideas in this work and then providing useful suggestions. Thanks to Dave Ogle for reading and providing comments on parts of this work.

Also, a special thanks to Yap Chua and Charles Winton. They provided encouragement and guidance in the early part of my computer science career.

CONTENTS

1 Introduction **1**
1.1 Motivation . . . 1
1.2 An Overview of the SP Architecture . . . 10
1.3 Thesis Statement . . . 15
1.4 Contributions . . . 16
1.5 Preview of Subsequent Chapters . . . 17

2 Related Work **21**
2.1 Introduction . . . 21
2.2 Systems that Mix Real-Time and Non-Real-Time Tasks . . . 22
2.2.1 Integrated Processor Scheduling . . . 23
2.2.2 Real-Time Mach . . . 26
2.2.3 Processor Capacity Reserves . . . 28
2.2.4 Rialto . . . 31
2.2.5 An IPC Mechanism for Continuous Media . . . 32
2.2.6 Summary . . . 34
2.3 Scheduling Work . . . 34
2.3.1 Lottery Scheduling . . . 35
2.3.2 Proportional Share Resource Allocation . . . 36
2.3.3 Rate-Based Execution . . . 37
2.3.4 Summary . . . 38
2.4 Extensible Kernels . . . 39
2.4.1 SPIN . . . 39
2.4.2 ExoKernel . . . 40
2.4.3 Microkernels and Virtual Machines . . . 41
2.4.4 Summary . . . 43
2.5 Time Division Multiplexing . . . 43
2.6 OS Emulation . . . 44
2.6.1 Virtual Machines . . . 45
2.6.2 OS/2 . . . 46
2.6.3 RTX 4.1 . . . 47
2.7 Summary . . . 49

3 The Slotted Priority Architecture **51**
3.1 Introduction . . . 51
3.2 A High Level Description of the SP Architecture . . . 53
3.3 Two Fundamental Requirements of an SP Implementation . . . 56
3.4 Realization of the Two Requirements . . . 60
3.5 Arguing that an Implementation of the SP Architecture Can Meet Guarantees . . . 64
3.6 Execution Model . . . 64
3.7 The Executive . . . 68
3.8 Resource Model . . . 70
3.8.1 Partitionable Resources . . . 78
3.8.2 Slotted Resources . . . 80
3.8.2.1 Preemptible Resources . . . 83
3.8.2.2 Non-Preemptible Resources . . . 84
3.8.2.3 Internally Triggered Non-Preemptible Resources . . . 90
3.8.2.4 Externally Triggered Non-Preemptible Resources . 92
3.8.2.5 A Periodic Process Driving an Externally Triggered Non-Preemptible Resource . . . 95
3.8.2.6 A Stochastic Process Driving an Externally Triggered Non-Preemptible Resource . . . 95
3.9 Programming Model . . . 98
3.9.1 Video Display Example . . . 99
3.9.2 Disk Write Example . . . 105
3.10 Intra-thread Communication . . . 108
3.11 Summary . . . 111

4 Analysis of the Slotted Priority Architecture CPU Executive **113**
4.1 Introduction . . . 113
4.2 Prior Work Related to this Analysis . . . 116
4.2.1 The Liu and Layland Model of Periodic Tasks . . . 116
4.2.2 Cyclic Executives . . . 118
4.2.3 Interrupt Overhead Cost Accounting . . . 119
4.2.3.1 Relationship of SP to the Interrupt Handler Work . 122
4.3 Determining Feasibility in an SP Implementation . . . 122
4.3.1 Relaxing Assumptions . . . 125
4.3.2 Reducing Computation Complexity . . . 128
4.4 Summary . . . 137

5 An Implementation of the Slotted Priority Architecture **139**
5.1 Introduction . . . 139
5.2 Hardware Description . . . 140
5.2.1 Periodic Interrupt Generators . . . 144
5.2.2 Use of Hardware Task Switch Mechanism . . . 145
5.3 Description of the Software Modules . . . 147

5.3.1 The Interrupt Enable/Disable Virtualization Component . . 147
5.3.2 The CPU Executive 150
5.3.3 The Real-Time Kernel 153
5.3.3.1 The Real-Time Thread Scheduler 155
5.3.3.2 The RTK Admission Control Function 155
5.3.3.3 The RTK Real-Time Thread Services Library . . . 156
5.3.3.4 The RTK Personality Neutral Server 156
5.3.4 Other Device Executives 157
5.4 Experiments 158
5.4.1 Requirement $\mathcal{A}$ Experiments 159
5.4.1.1 Slot Length Variation 160
5.4.2 Requirement $\mathcal{B}$ Experiments 167
5.4.2.1 Baseline Experiment 171
5.4.2.2 Loop Execution Time Variation with Calls to the Display Library 176
5.4.2.3 Loop Execution Time Variation with Loop Execution in Two Slots 179
5.4.2.4 Loop Execution Time Variation Caused by GPOS Execution 179
5.4.2.5 Loop Execution Time Variation with Loop Execution in Two Real-Time Minor Cycles 183
5.4.3 Arguing that this Implementation of the SP Architecture Meets Guarantees 187
5.4.4 SP Overhead 190
5.4.5 GPOS Slowdown 192
5.4.6 Summary of Experiments 195
5.5 Discussion 196
5.5.1 New Implementation Design Considerations 197
5.5.2 Minor Cycle Lengths 200
5.6 Summary 202

6 Application Demonstrations **204**
6.1 Introduction 204
6.2 Character Display Demonstration 205
6.2.1 Shared Resources 207
6.2.2 Real-Time Kernel 209
6.2.3 Real-Time Threads 210
6.2.4 Non-Real-Time Threads 212
6.2.5 Analysis 212
6.2.6 Empirical Results 215
6.2.7 Summary 215
6.3 Network Audio Demonstration 216
6.3.1 Shared Resources 219
6.3.2 Real-Time Kernel 222

6.3.3 Real-Time Threads 222
6.3.4 Results 224
6.3.5 Network Audio Demonstration Summary 226
6.4 Chapter Summary 227

7 Implications and Summary **228**
7.1 Introduction 228
7.2 Further Research 228
7.2.1 An Alternate Real-Time Thread Model 230
7.2.2 RTClock Frequency Discrepancy 232
7.3 Summary 233
7.4 Conclusion 236

Bibliography **238**

LIST OF FIGURES

1.1 Major and Minor Cycles 13

2.1 Emulation of Operating Systems in VM 46
2.2 Emulation of DOS in OS/2 48

3.1 An Operating System View of the Execution Model 54
3.2 Major and Minor Cycles 67
3.3 Executive Organization 71
3.4 Resource Model Classification 72
3.5 An Obvious Approach to Multiplexing 74
3.6 The SP Architecture Approach to Multiplexing 77
3.7 Sequence of Commands and Responses for a Network Send Operation 82
3.8 Video Display Example 100
3.9 Code for Video Display Example 101
3.10 Disk Write Example 106
3.11 Code for Disk Write Example 107

4.1 L is a multiple of MC 133
4.2 L is *not* a multiple of MC 136

5.1 Some Important Motherboard Circuits. 143
5.2 Relationship of the Major Software Modules 148
5.3 Slot Lengths (Quiet System) 162
5.4 Cumulative Percent of Slots From Nominal (Quiet System) 163
5.5 Slot Lengths (Busy System) 164
5.6 Cumulative Percent of Slots From Nominal (Busy System) 165
5.7 Major Cycle Lengths (Busy System) 168
5.8 Cumulative Percent of Major Cycle Lengths From Nominal (Busy System) 169
5.9 Baseline Experiment 173
5.10 Variation in Execution Times for the Baseline Experiment 174
5.11 Variation in Completion Times for the Baseline Experiment 175
5.12 Variation in Execution Times when the Loop Uses the Display Device 177
5.13 Variation in Completion Times when the Loop Uses the Display Device 178

5.14 Variation in Execution and Completion Times for a Loop that Executes Across Two Consecutive Slots 180
5.15 Variation in Execution Times for a Loop that Executes in Two Consecutive Slots 181
5.16 Variation in Completion Times for a Loop that Executes in Two Consecutive Slots 182
5.17 Variation in Execution Times when the GPOS is Active 184
5.18 Variation in Completion Times when the GPOS is Active 185
5.19 Loop Execution in Two Consecutive Real-Time Minor Cycles Experiments 186
5.20 Variation in Execution Times for a Loop that Executes in Two Consecutive Real-Time Minor Cycles 188
5.21 Variation in Completion Times for a Loop that Executes in Two Consecutive Real-Time Minor Cycles 189

6.1 Real-Time Thread's Computation Loop 211
6.2 Network Audio Demonstration Diagram 218

7.1 Typical Real-Time Thread Loop 230

LIST OF TABLES

3.1 Minor and Major Cycle Lengths for Four Instances of 'Process Data' 104

5.1 Minor Cycle Lengths used in Requirement $\mathcal{B}$ Experiments 171
5.2 Execution time in seconds for an IBM MicroKernel build. 191
5.3 Measured overhead in percent of all processor cycles consumed. . . 192
5.4 Percent Slowdown of a User Loop 193
5.5 Percent Slowdown of a User Loop (continued) 194

6.1 SP and Real-Time Thread Parameters 213
6.2 Evaluation of Feasibility 214
6.3 Notes Missed in Network Audio Demonstration 225

LIST OF ABBREVIATIONS

API	Application Programming Interface
ASCII	American National Standard Code for Information Interchange
CPU	Central Processing Unit
CRK	Co-Resident Kernel
CP	Control Program
CSMA	Carrier Sense Multiple Access
DMA	Direct Memory Access
DOS	Disk Operating System
EDF	Earliest Deadline First
EISA	Extended Industry Standard Architecture
FIFO	First In First Out
FSM	Finite State Machine
FPT_{nrt}	Fraction of the Processor Time for Non-Real-Time Tasks
FPT_{rt}	Fraction of the Processor Time for Real-Time Tasks
FTP	File Transfer Protocol
GDT	Global Descriptor Table
GPOS	General Purpose Operating System
GUI	Graphical User Interface
IEC	Interrupt Enable/Disable Component
IO	Input Output
ISA	Industry Standard Architecture

ITC	Intra-Task Communication
ITDS	Integrated Time-Driven Scheduler
LAN	Local Area Network
LDT	Local Descriptor Table
LRTF	Least Remaining Time First
MC	The Length of a Major Cycle
mc_{nrt}	The Length of a Minor Cycle for Non-Real-Time Tasks
mc_{rt}	The Length of a Minor Cycle for Real-Time Tasks
MHz	Millions of Cycles Per Second
MVDM	Multiple Virtual DOS Machines
OS	Operating System
OS/2	IBM's Operating System for Intel-based PC Class Computers
PC	Personal Computer
PCR	Processor Capacity Reservation
PIC	Programmable Interrupt Controller
RAM	Random Access Memory
RBE	Rate Based Execution
RTClock	Real Time Clock
RTK	Real-Time Kernel
SJF	Shortest Job First
SCSI	Small Computer System Interface
TSS	Task State Segment
UDP	User Datagram Protocol
VDM	Virtual DOS Machine
VGA	Video Graphics Adapter
VM	Virtual Machine
YARTOS	Yet Another Real-Time Operating System

Chapter 1

Introduction

1.1 Motivation

Recent advances in network technologies, processor capabilities, and microcomputer system hardware, coupled with the explosive growth of the Internet and on-line data access, have created new demands on personal computer (PC) operating systems and hardware. In large part, these demands are for the ability to acquire, manipulate, display, and store multimedia data. The temporal characteristics of multimedia data (typically video and audio) are fundamentally different from all other types of data for which general-purpose computer systems have been designed, primarily because, for a particular instance of their use, multimedia data are perishable in time. For example, a particular sequence of bytes comprising a video frame is usable for only a few tens of milliseconds in an application such as video conferencing.

Two other unique characteristics of multimedia data streams are also relevant. First, in some cases, multiple streams such as audio and video, may have to be synchronized to be useful. Second, for live interactive multimedia streams received from a network, buffering alone cannot be used to smooth out variations in arrival times of the data packets from the network. This is because the latency created by buffering is unacceptable to humans in two-way communication, not because of the buffering itself, but because of the amount of buffering one typically must do in practice. These temporal characteristics of multimedia data streams place constraints on executing the tasks[1] that manipulate multimedia data streams. Primarily, the task, in concert with its environment, must be able to guarantee its timely completion or else the data's usefulness will degrade. For example, incoming compressed video frames must be decompressed and displayed approximately every 33 milliseconds, or the quality of the displayed video will degrade.

In 1973 Liu and Layland published their seminal work in the scheduling of tasks with real-time execution constraints [24]. Their model is called the periodic real-time task model. The two defining temporal characteristics of periodic real-time tasks are (1) the tasks are invoked once in every interval of time of a fixed length, known as the period, and (2) the i^{th} invocation of a task must complete its computation before the end of the i^{th} period.

At an abstract level, the properties of periodic real-time tasks are exactly the properties that tasks handling multimedia data streams require. Consider a periodic real-time task that receives, decompresses, and displays video frames arriving from a network. With a period of 33 milliseconds, this task would, given the timely

[1] In this work three terms are used to refer to an instance of a program executing on a computing machine. "Task" is used in the context of describing previous work that used that term. "Process" is used when referring to the complete program in a general sense. "Thread" is used when referring to a portion of a program with certain characteristics or requirements.

arrival of video frames from the network, correctly display the video data at 30 frames per second without distortion. Unfortunately, however, no commercial successful operating systems for PC class computer systems support periodic real-time tasks without auxiliary hardware.

Another useful application of periodic real-time tasks is in the area of process automation. Control of robotic machines in manufacturing lines has long been an area where traditional real-time operating systems are employed. A general-purpose operating system that could correctly control these robotic machines would have significant value to industry.

This paper develops a practical method to guide additions to general-purpose operating systems (GPOS) for PC class computer systems to support periodic real-time tasks.

General-purpose operating system design has developed over four decades in an environment where the characteristics of the data manipulated by the computer system have not included a temporal aspect. The correctness of the hardware and system and application programs have not been defined for data that lose value in time. Note that this is different from *response* time (a type of temporal requirement) that has been much studied. Used often as a quality measure for interactive programs response time is measured from some event until the completion of the task initiated by that event.

After the initial problems of transmission, storage, and manipulation of data were solved in early computer designs, the emphasis shifted to focus on the problem of fairly allocating the resources of a computer system among its various tasks. Important milestones were interrupts, scheduling and multitasking, virtual memory, instruction and data caching, pipelining, and direct memory access (DMA). These features of operating systems and computing devices all increase a computer system's overall efficiency but add variability to the length of time that any particular task requires to complete its computation. This variability precludes supporting periodic real-time tasks, because they have deadlines that may not be met if the amount of time they need to finish varies too widely.

Early computer system designs used the processor to interact directly with attached devices, resulting in an interaction paradigm known as programmed I/O. Then, and now, attached devices were three orders of magnitude slower than the processor [3]. Interrupts, first developed in the IBM Stretch [3], relieved the processor from continuous interaction with attached devices. An interrupt is a signal from an attached device that causes the processor to switch from its current task to a sequence of instructions (the interrupt handler or device driver) that interacts with the signalling device. Thus, the processor executes the instructions of the interrupt handler only when the device is ready to interact with the processor; this allows the processor to execute instructions for other activities while the device is busy with its internal operations.

An important point to note here is that devices operate asynchronously with respect to the processor and each other. As a result, the times when interrupts arrive at the processor, and hence how frequently interrupt handlers execute, cannot be precisely predicted. Thus, a design trade-off with interrupts is that one can no longer predict how many processor cycles will be consumed by interrupt handlers in a given interval and how many will be available for other tasks. This is a problem for periodic real-time tasks, because they must be able to finish their computation before their next period begins.

As interrupts were developed, it became clear that a processor relieved of interaction with devices was useful only if it could do other work. The notion of multitasking, with its requirement of task scheduling, became important. Multitasking describes an operating system that simultaneously multiplexes the execution of more than one user program. Scheduling refers to the way the operating system decides which program to execute next.

Many scheduling algorithms were developed, analyzed, and implemented, but eventually a single design dominated and is used in some form in many general-purpose operating systems. It comprises a set of round-robin priority queues with priority promotion. Each queue holds tasks of the same priority, and tasks from the queue with the highest priority are serviced by allowing each task in the queue a fixed amount of time (quantum) on the processor in rotating order among all tasks in that queue. Tasks in the queue with the next highest priority are serviced in the same manner only when the higher priority queue is empty. As tasks spend more time in lower priority queues, their priority slowly increases so that they are eventually serviced. A design trade-off here is that the time any particular task may wait in the queue system depends on the number of tasks in the system and the task's initial priority. This wait time can be predicted only by using stochastic methods with assumed distributions of task service requirements and task invocation times. Thus one cannot predict for how long any particular invocation of a task will wait in the queue system before completing its computation. This is not a liability when using a general-purpose operating system since a heavily loaded computer system can be expected to take longer to finish any particular task. That is, in a GPOS one typically only cares about average performance. However, not being able to predict the completion time of a task means that a system cannot guarantee completion times as required by periodic real-time tasks.

Another design milestone in the development of operating system theory was the principle of locality first proposed by Denning [6]. This principle simply states that it is highly likely that the next logical instruction to be executed in a task will be close, in memory location, to the previously executed instruction. The principle of locality allows the efficient implementation of paging. Paging was first implemented on the ATLAS [1]. In a paged system memory pages no longer actively in use are moved from memory to backing store, providing physical memory free for other tasks. From paging came the idea of virtual memory, analyzed by Denning [7]. Virtual memory extends the logical address space available to tasks by multiplexing the use of physical memory. Again, a design trade-off with paging and virtual memory is a decrease in the the ability to predict when any particular task will finish, since bringing in a page from disk to resolve a page fault introduces considerable variability into the execution time of a task.

Another efficiency enhancement, direct memory access (DMA) allows an attached device to transfer data between itself and memory, and between itself and other devices, without much interaction with the processor. The processor must set up the parameters of the transfer but need not attend to the transfer of each word. Because of how devices typically behave it is more efficient to allow the device using DMA to have priority access to the system bus. This prevents the processor from accessing memory during DMA transfers. This results in a more efficient system overall but makes the completion time of a task less predictable because the length of time the processor must wait for access to the system bus to fetch instructions and data from memory is more variable.

Instruction and data caching increases the efficiency of computer systems by keeping recently used instructions and data in memory with low access time, rather than in relatively slower main memory. Efficiency is increased because main memory access time is slower than average instruction execution time but cache memory access time is more closely matched to instruction execution time. Because cache memory is much more expensive than main memory, caches are typically much smaller than main memory. This requires keeping a limited number of instructions and data in the cache and replacing cache lines, according to a particular replacement policy, when a cache miss occurs. Using instruction and data caches again increases the variability in task completion times.

The unpredictability of task completion time is also increased by the unrestricted use of the interrupt enable/disable mechanism, which is current practice in some PC systems. This is because during the time when interrupts are disabled, the operating system may not receive the interrupt from the system timer when the timer expires. Instead it is received later when (1) interrupts have been enabled, and (2) the interrupt from the system timer obtains priority, among any other pending interrupts, to interrupt the processor.

For example, consider a process that has issued a call to the GPOS to sleep for n milliseconds. If, at the end of the n milliseconds, interrupts were disabled by some privileged but non-kernel code, the GPOS would be unable to gain control to awaken the sleeping process. In other executions of that process, however, the sleeping process might be awakened on time giving it a different execution time. The interrupt enable/disable mechanism, unfortunately, is used in many PC systems to ensure mutual exclusion of critical sections within the operating system kernel. In some cases, interrupts may be disabled for rather long periods of time because, for example, the critical section being protected is a search through a linked list of events with no practical upper bound on the number of elements.

Any practical method for adding support for periodic real-time tasks to a GPOS would have to address the variability in task completion time introduced by interrupts, scheduling and multitasking, virtual memory, DMA, caching, RAM refresh[2], and the unrestricted use of the interrupt enable/disable mechanism. One approach

[2]RAM refresh is an artifact of the realization of memory by memory cells that indicate data values by storing a voltage potential in a capacitor. Capacitors are not perfect machines and therefore leak the voltage. In time the voltage in the memory cell will be insufficient for the read lines to determine the data value of the memory cell. RAM refresh periodically restores the voltage potential to its initial value so that subsequent reads occur without error. To do this, it uses the bus and therefore the bus is unavailable to the CPU or other devices.

(not the one used in this work) would be to modify the source code of the GPOS to reduce this variability sufficiently and to add a new scheduler (with or instead of the existing scheduler) that supports the scheduling of periodic real-time tasks and non-real-time tasks.

Another approach, proposed by Shaw [33], would be to determine the "dilation" effect that interrupt handlers have on task execution time. If attached devices were designed so that the minimum interarrival times between interrupts to the processor and the length of execution times of the interrupt handler for the device were bounded, then their maximum dilation effect on task execution time could be calculated. This approach does not consider other factors, such as page misses, DMA, and the unrestricted use of the interrupt enable/disable mechanism, that contribute to variability in task execution time.

This work develops another approach, named the slotted priorities (SP) architecture. Instead of modifying the GPOS, SP adds another simple operating system kernel (a real-time kernel, RTK) that provides the basic services required by periodic real-time tasks and multiplexes the execution of the GPOS and its tasks with the execution of the real-time kernel and its tasks in alternate intervals.

Both the length of the intervals and their starting points must be precise. SP addresses the unrestricted use of the interrupt enable/disable mechanism for mutual exclusion by requiring virtualization of the interrupt enable/disable mechanism. This virtualization allows the system to create precise intervals, using the timers available on the hardware, in which the two kernels and their respective tasks execute. Without virtualization of the interrupt enable/disable mechanism, the interrupt that marks the beginning of the interval in which the real-time kernel and its tasks execute could be delayed, which could cause a real-time task to miss its deadline.

SP also disallows interrupts, virtual memory, and may disallow DMA (depending on the design of the system bus) while the computer system is executing in the interval belonging to the real-time kernel and its tasks. The real-time kernel (RTK) should also provide periodic real-time task scheduling, a task admission control algorithm, and various libraries of real-time utilities.

In addition to defining the SP architecture, this work also presents the details of an implementation of the SP architecture on an Intel i486 hardware platform, the results of a number of experiments to show that the implementation of the SP architecture meets the requirements defined in the architecture, and two application demonstrations to highlight portions of the SP architecture implementation.

1.2 An Overview of the SP Architecture

At a high level, the slotted priorities architecture shares the resources of a computing system between a general-purpose operating system and a real-time kernel. Software components called executives manage how an instance of a resource is shared.

The slotted priorities architecture consists of:

- an execution model that explicitly defines how the CPU is multiplexed between the GPOS and the RTK

- a taxonomy of resources classed by characteristics of the resource and the services required in an executive to ensure that the executive correctly shares the resource between the GPOS and the RTK

- for each class in the taxonomy, a description of the services required in an executive that can correctly share the resources in that class

- a programming model that includes guidelines for writing real-time tasks to execute on a system which implements the architecture

Executives manage the sharing of a particular resource between the GPOS and the RTK. Examples of resources are the CPU, memory, display, and network device. Executives ensure that the RTK can gain access to the resource at precise times. Various types of executives are needed in any particular implementation of the SP architecture. For example, the executive needed to manage sharing of the display differs from the executive needed to manage the sharing of the CPU. For any resource in a particular class in the resource classification, a single executive model can be used to construct the actual executive.

There are two main classes in the resource classification: partitionable and slotted. Partitionable resources contain numerous identical units, all of which may be in simultaneous use. Examples of partitionable resources are main memory, direct access storage device (DASD) blocks, TCP/IP ports, and display pixels. Partitionable resources are shared by separating the identical units into two disjoint sets. The GPOS and its tasks use the units in one set exclusively and the RTK and its tasks use the units in the other set exclusively.

Slotted resources contain a single unit that must be multiplexed between the two OS kernels. An example of a slotted resource is the processor. The execution model describes the multiplexing between the GPOS and its tasks and the RTK and its tasks. The processor is in the preemptible subclass of the slotted class, because almost all computer systems have hardware support that allows preemption of the processor. Preemption saves the current state, installs a new state, and resumes the execution of the processor on another sequence of instructions. The hardware support for preempting the processor reduces the complexity of sharing the processor between the GPOS and the RTK. Other slotted resources that are not typically provided with a preemption capability fall in other subclasses of the slotted class. The executives that manage their sharing are even more complex than the executive for sharing the processor.

The execution model defines how real-time tasks acquire the processor. Time is viewed as an infinite series of discrete points starting at time $t = 0$. A time unit is the interval between two points. Time units are divided into two classes: real-time time units and non-real-time time units. A real-time minor cycle is a contiguous sequence of mc_{rt} real-time time units and a non-real-time minor cycle is a contiguous sequence of mc_{nrt} non-real-time time units. A major cycle, MC, is an interval made up of mc_{nrt} time units followed by mc_{rt} time units. Thus the length of a major cycle is $MC = mc_{nrt} + mc_{rt}$ time units.

Major cycles occur one after another forever. The GPOS and its non-real-time tasks execute within successive non-real-time minor cycles, and the RTK and its real-time tasks execute within successive real-time minor cycles. Figure 1.1 illustrates the relationships between the major cycle and the two types of minor cycles.

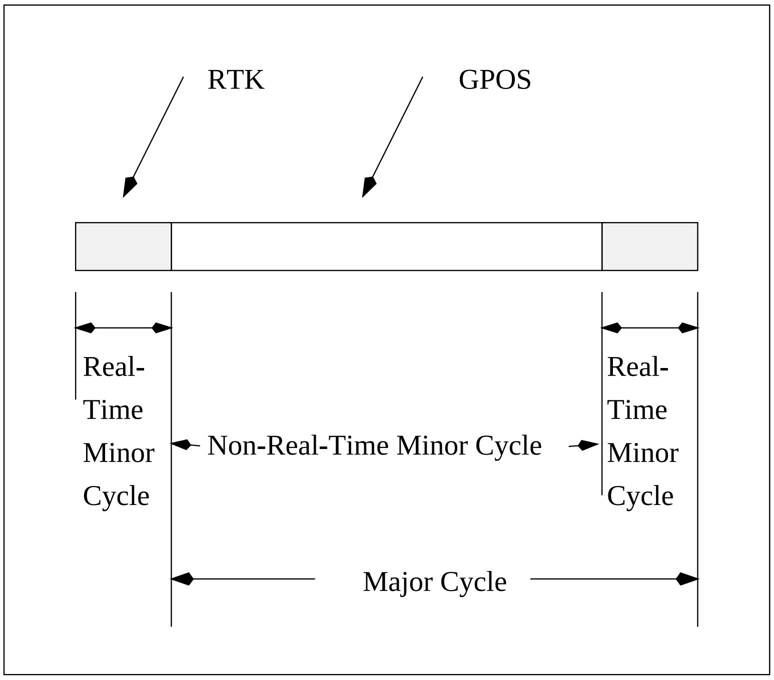

Figure 1.1: Major and Minor Cycles

Within the execution model, the processor executive must allow precise minor cycle lengths to be created. This will typically be accomplished with an interrupt from a system timer. If the SP architecture is being implemented on a computer system whose interrupt enable/disable mechanism has unrestricted use, the processor executive must ensure that the interrupt that marks the minor cycle boundaries is not delayed because of disabling of interrupts by the GPOS.

The SP architecture does not specify, nor depend on, particular values for the lengths of the minor cycles. Consider a set of periodic real-time tasks that use 50 percent of the processor cycles, including the overhead of the RTK and any variability in the start of the real-time minor cycle. In this situation, the SP architecture supports any values for the lengths of the minor cycles, as long as the two minor cycles have equal lengths (50 percent for the RTK and 50 percent for the GPOS). Thus, the minor cycle lengths of $mc_{rt} = 5$ ms and $mc_{nrt} = 5$ ms or $mc_{rt} = 500$ ms and $mc_{nrt} = 500$ ms would be valid.

There are however, practical considerations. There is a trade-off between the overhead incurred by switching between the minor cycles and the granularity at which the real-time tasks can specify their periods. For example, if an implementation limited minor cycle lengths to at least 30 ms then no real-time task could specify a period of less than 60 ms. There may also be practical limitations on the maximum length of a real-time minor cycle, such as existing device drivers or devices that are not robust when isolated from each other during the real-time minor cycle. Also note that shorter real-time minor cycles reduce the system's efficiency. Thus determining minor cycle length should be a dynamic function of the RTK that is adjusted for efficiency and practical considerations as real-time tasks are created.

1.3 Thesis Statement

Given the utility afforded by general-purpose operating systems to their users and the demand for processing video and audio data, there is a clear need to support periodic real-time tasks within general-purpose operating systems. The thesis of this work is one approach to merging the utility of general-purpose operating systems with support for real-time computation.

> *The slotted priority architecture defines a way to modify general-purpose operating systems to schedule, manage, and execute periodic real-time tasks. The resulting modified system will be able to guarantee that all periodic real-time tasks active on the system will meet their stated deadlines. Periodic real-time tasks may reserve resources, such as direct access storage device access and network access, and the modified system will guarantee that the resources are available when needed by the reserving tasks with a reasonable maximum latency. For contemporary operating systems the modifications to the original general-purpose operating system are modest.*

The thesis has been proved by building a prototype implementation of the architecture within a general-purpose operating system and then analyzing and testing the implementation by executing periodic tasks doing multimedia processing. We mathematically derived an analysis to determine the feasibility of a set of periodic real-time tasks executing on an implementation of the SP architecture. A feasible set of real-time tasks was created and executed on the implementation to see if any task missed its deadline. Additionally, test cases were run to determine if the minor cycles started with bounded latency and if the variability of real-time thread completion times was reasonably small.

1.4 Contributions

This work makes several contributions to the existing body of knowledge concerning the fair and efficient sharing of the resources of a computing machine.

Slotted Priority Architecture: an architecture to add support of periodic real-time tasks to general-purpose operating systems.

Execution Model: a detailed description of how periodic real-time tasks may share the CPU with non-real-time tasks. This includes deriving a feasibility condition that, if satisfied, ensures that all periodic real-time tasks will meet their deadlines and a demonstration of how the model can be used to solve some nontrivial real-time problems.

Resource Model: a taxonomy of classes covering the physical devices typically attached to a computing machine. All members of a particular class can be shared between the GPOS and the RTK with a similar executive.

1.5 Preview of Subsequent Chapters

Chapter 2 examines other work that merges real-time support with general-purpose operating systems, explores the issues surrounding the joint scheduling of real-time tasks and non-real-time tasks, and describes extensible kernels. Also reviewed are time division multiplexing, and operating systems that emulate other operating systems. Since the SP architecture is conceptually an addition to a GPOS, this chapter includes work in operating systems theory that provides for the dynamic addition of function to operating system kernels—the so-called extensible kernel school of OS design. The SP architecture's guidelines for construction of a CPU executive multiplexes the CPU into two virtual CPUs in much the same manner as time division multiplexing uses a single physical link for two, or more, communication streams. Therefore Chapter 2 examines the relation to time division multiplexing. The SP system can also be viewed as two separate operating systems sharing the same physical hardware. Two other systems, VM and OS/2, that also do this are examined.

Chapter 3 presents the complete SP architecture. Two requirements necessary for any implementation of the SP architecture are derived from first principles and their realization in an actual system is discussed. Once an SP system is implemented, one must know the criteria for determining that that implementation can actually meet the performance requirements of real-time tasks. This argument, described in Section 3.5, has two parts. The first part is to determine by analysis and experimentation if a particular implementation of the SP architecture meets the two requirements stated in Section 3.3. The second part is to use the feasibility analysis presented in Chapter 4 to determine if a particular set of periodic real-time tasks is feasible when executed on an implementation of the SP architecture. Once these two parts are complete, an implementation of the SP architecture will be able to execute that set of real-time tasks so that no task ever misses its deadline.

The three models that constitute the SP system are then described. The execution model describes how the CPU is shared between the GPOS and the RTK. The resource model classifies resources by characteristics that determine how they are shared, and the programming model describes how a programmer would write a real-time program for execution on an SP implementation.

Chapter 4 is a complete analysis and proof of the feasibility condition for a set of periodic real-time tasks executing on an SP implementation. Unlike task scheduling on a GPOS, real-time task scheduling includes the notion of a saturated processor. This is the point at which adding another real-time task causes some real-time tasks to miss their deadline. Thus, for a particular set of real-time tasks, and a scheduling policy, a feasibility analysis will show whether a schedule can be determined in which all tasks meet their deadlines. For this reason an implementation of the SP architecture must include a computationally tractable feasibility analysis to compute the feasibility of the set of real-time tasks as new real-time tasks request admission to the system. The Liu and Layland task model is adapted for use in the SP architecture.

Chapter 5 describes an implementation of the SP architecture and a set of experiments that verify that the implementation meets the two fundamental requirements presented in Chapter 3. The test implementation of SP was accomplished on a Model-95 PS/2 IBM MicroChannel Intel 80486 based computer. The implementation used only those circuits normally found on the 'AT' motherboard[3]. The set of experiments showed that the implementation satisfied the two fundamental requirements described in Chapter 3 and quantitatively showed the slowdown of the GPOS when the SP system was added. Also, the issues concerning implementing the SP system on any computer are described in detail.

Chapter 6 describes two applications built for the SP implementation. The pixel display application has two real-time tasks that loop continuously and compute an artificial value and display a pixel in each iteration of the loop. This computation and display must be completed in each period. The two real-time tasks have different periods. The values of the periods and costs of the two real-time tasks, as well as system parameters, are used in a sample calculation of the feasibility of this set of real-time tasks.

The second application is a network audio demonstration. This application receives packets of musical notes sent over a token ring network from a non-real-time machine running the OS/2 operating system and plays the musical notes on the speaker of the test SP implementation. There is minimal buffering of the packets of notes, yet the SP system accomplishes its guarantees. In both

[3]The AT motherboard is the set of circuits available on the original IBM Model AT computers and now the de facto standard on all Intel-based personal computers (PCs) produced by various manufacturers.

demonstrations, the GPOS was fully active and we show that the real-time threads always complete before their deadlines.

Chapter 7 summarizes the this work and presents conclusions. It also discusses the future direction of this work.

Chapter 2

Related Work

2.1 Introduction

Chapter 1 briefly described the SP architecture and the problem it solves. This chapter outlines related work that solves the same problem by different approaches. Additionally, this chapter presents work from which certain aspects of the SP architecture derive and which, at a high level, resembles the SP architecture. With the exception of Real-Time Mach each work presented in this chapter addresses only part of the problem solved by the SP architecture. Real-Time Mach addresses the whole problem but uses a fundamentally different approach.

Section 2.2 presents various attempts at mixing the execution of real-time tasks and typical GPOS tasks in workstation operating systems. Most designs center around the problem of scheduling this mix of tasks in the context of a GPOS. But supporting real-time tasks in a GPOS transcends simple scheduling. Issues such as ensuring predictable computation times once scheduled by controlling DMA, virtual memory, the interrupt enable/disable mechanism, and the execution of interrupt handlers must be addressed. If real-time tasks are to be supported, they must be able to access all required resources when they need them; so simply modifying the scheduler in a GPOS will not suffice. Nonetheless, scheduling is important and Section 2.3 describes two pure scheduling algorithms that could be employed to schedule real-time and non-real-time tasks.

Another possible approach is to let applications define their own schedulers (among other things) and download these operating system 'extensions' into the kernel at runtime. Section 2.4 presents work addressing operating system architectures that allow the dynamic addition of function to the operating system kernel. Section 2.5 compares the allocation of CPU cycles in SP to time division multiplexing, a well-known technique in networking used to share physical communications links. Section 2.6 describes two operating systems that emulate other operating system.

2.2 Systems that Mix Real-Time and Non-Real-Time Tasks

Many commercial operating systems claim to support real-time computation. However, those claims are usually based on a much looser definition of real-time computation than the definition used here. We define a real-time task as a task that has declared a deadline and that is considered to have failed completely if it has not completed its computation before its stated deadline.

Broader definitions of real-time range in meaning from the ability of an operating system to begin a task with low latency from some defined event, to the ability to process data as quickly as a human operator can input it, to the existence of a highest priority class available to user tasks. These looser definitions of real-time are useful in their own systems. However, even if the system faithfully implements its own specifications, it cannot be used to schedule and execute periodic real-time tasks while providing guaranteed completion times. This section reviews some research based on definitions of real-time that are closer to the definition used in the SP architecture.

2.2.1 Integrated Processor Scheduling

Nieh et al. developed the concept of integrated processor scheduling because the real-time scheduler supplied with System V Release 4 UNIX could not effectively schedule a set of tasks which included both traditional timesharing tasks and tasks for managing multimedia (which required real-time support) [27]. Integrated processor scheduling is so named because it integrates the scheduling of conventional and real-time tasks within a single system.

Measurements completed on the System V Release 4 (SVR4) UNIX scheduler showed conclusively that the SVR4 priority based scheduling algorithm could not schedule both applications with deadlines and typical GPOS applications [27, 28]. The SVR4 real-time scheduler allows real-time tasks to be scheduled at the highest priority class in the system to allow them to receive processor cycles as needed. Specifically, the real-time scheduler incorporated in UNIX System V Release 4 could not meet its stated claim of being able to efficiently and fairly schedule real-time and non-real-time tasks and the scheduler allowed priority inversion to occur. Measurements showed that multimedia tasks executing in the SVR4 and scheduled with the SVR4 real-time scheduler had unacceptable latencies. In addition SVR4 may lock up when attempting to execute multimedia tasks. The lock-up had three causes. The SVR4 scheduler incorrectly identified a batch type of job as interactive and raised its priority. The incorrect identification occurred because the batch job executed many `sleep` calls to wait for its forked child processes to complete. The SVR4 scheduler incorrectly identified the windowing subsystem displaying video as compute intensive and reduced its priority. The priority of the process generating video frames increased because it slept waiting for new frames.

Thus, due to all three causes, many more frames were generated than the windowing system could process. This effect is an instance of the priority inversion problem, which is the observation that a low priority task can lock a resource and then be swapped out, because a higher priority task has become ready to run, while still holding the lock. If the higher priority task attempts to access the locked resource it will block. If this occurs when many medium priority tasks become ready to run, the medium priority tasks will run and block the low priority task so that the resource remains locked. Thus, the high priority task may be starved indefinitely, while medium priority tasks continue to run.

Additionally, although this work did successfully tune task priorities in the experimental system so that latencies and response times from the experimental mix of applications were reasonable, it was argued that the priority values were useful only for that particular mix of applications and input data. Small differences in the tuning values, the application mix, or the input data could have caused radically different latencies and response times [27].

Integrated processor can fairly schedule a mix of tasks with deadlines and traditional priority-scheduled tasks because it does not confuse urgency and importance. It does not give higher priority to real-time tasks just because they have deadlines, but rather allocates processor cycles fairly among all tasks active in the system, both conventional and real-time.

The scheduler operates by first giving processor cycles to all real-time tasks in an earliest deadline first order until all have received their fair share of processor cycles. A fair share is determined by weighing the required processor demand of the real-time tasks against the workload of the non-real-time tasks. The conventional tasks are then scheduled with a round-robin scheduler. One criterion in developing this scheduler was that the application programmer would not have to supply information, such as execution cost, about a particular real-time task that is difficult or impossible to obtain.

However, a real-time task may receive its fair allocation of processor cycles and still not complete in time. In this case, the scheduler notifies the task of the impending missed deadline but does not allocate more processor cycles to the real-time task, because this may cause other real-time tasks to miss their deadlines or significantly lengthen the time conventional tasks need to complete their computation.

While it is clear that integrated processor scheduling can allocate processor cycles fairly to a mix of real-time and typical GPOS applications, the results show that the completion times of the real-time tasks vary significantly. In a completely quiet system, the standard deviation of the completion times of the real-time tasks is about 10 percent of the mean (average completion time is 112 ms and standard deviation is 9.75 ms); on a busy system, the standard deviation of the completion times of the real-time tasks is about 30 percent of the mean (mean completion time 177 ms and standard deviation 48.3 ms) [27]. This is undesirable, because the largest possible completion time would have to be used to determine the share given to the real-time tasks, thus reducing the amount of processor cycles available for real-time tasks. By contrast, the SP implementation, as discussed in Chapter 5, provides real-time task completion times whose standard deviation is only about 0.03 percent of the mean on a busy system. Also, if the effect of CPU cycle stealing through DMA, of interrupt handlers, and of the unbounded delays induced by unrestricted use of the interrupt enable/disable mechanism is not addressed, implementing integrated processor scheduling on a typical GPOS may not guarantee an upper bound on task completion time. These issues are discussed further in Section 3.3.

2.2.2 Real-Time Mach

The work on Real-Time Mach attempts to develop a real-time version of the Mach Kernel [36]. Real-Time Mach addresses the same issues as the SP architecture but uses a different approach. All of the subsystems of the Mach kernel are modified to create reasonable upper bounds on the cost of using the services of these subsystems.

Real-Time Mach adds real-time thread management, an integrated time-driven scheduler (ITDS), real-time synchronization, and memory resident objects to the standard Mach kernel. Processes can create real-time and non-real-time threads. Real-time thread creation requires parameters to define the thread's timing characteristics, such as the period (both periodic and aperiodic threads are supported), maximum cost, and deadline, and whether the deadline is hard or soft. Real-Time Mach adds real-time synchronization to solve the priority inversion problem by using priority inheritance [32]. Real-time synchronization thus establishes an upper limit to the amount of time a high priority process may be blocked by a lower priority process that has locked a resource the higher priority process needs.

Externally Real-Time Mach resembles an implementation of the SP architecture because a real-time process in either system sees a unified application programming interface (API) with which the process can create real-time threads and reserve resources. However, although they appear similar, their internal differences are significant. In some sense the design of Real-Time Mach follows an obvious course: modify a GPOS kernel sufficiently to support the scheduling of tasks with deadlines. In doing so, most of the GPOS kernel had to be modified or at least analyzed to ascertain the effect on real-time threads. A new scheduler to support the mix of real-time and non-real-time threads had to be designed and implemented, and the memory subsystem of the kernel had to be redesigned.

The SP architecture takes the opposite approach: it considers the real-time and non-real-time requirements to be so different that ultimately it is less costly, in terms of the programming task, to avoid merging them and to develop a generic approach that can be used with any GPOS.

2.2.3 Processor Capacity Reserves

Processor capacity reservation (PCR) is an abstraction and mechanism, added to Real-Time Mach, to allow the user to control allocation of processor cycles among a number of tasks reserving processor cycles [25]. A new kernel abstraction, the reserve, is also introduced. The reserve tracks the reservation and measures the processor cycles used by tasks with reservations. The reserve is not bound to a process but rather to a thread of execution, so that in microkernel operating systems and windowing systems like X-windows, processor cycles accumulate both when a process is executed and when servers are executing on behalf of the task that reserved processor capacity. PCR solves the problem of scheduling tasks with real-time requirements and conventional time-sharing tasks.

The work defines four issues necessary for the reservation strategy to perform correctly. The system must:

1. provide some means for application programs to specify their processor requirements.

2. evaluate the processor requirements of new programs to decide whether to

admit them.

3. schedule programs consistently with the admission control policy.

4. accurately measure the computation time consumed by each program to ensure that programs do not overrun their reservations.

The task model is that of periodic tasks. PCR handles issue one by measuring the processor time used by a process during an interval. This time is then divided by the length of that interval to give the percentage of processor time required by this application or its rate. This rate, or processor specification, is the criterion by which tasks are scheduled.

The second issue, admission control, is performed by summing the utilizations of all of the real-time processes when using dynamic priority assignment. The work notes problems with this approach and states that fixed priority assignment could also be used, but that in general fixed priority scheduling cannot schedule 100 percent of the processor. For example, a process that requires 30 percent of the processor may require 30 ms every 100 ms or 300 ms in 1000 ms, two very different conditions. Thus two of the following three variables are required to specify the task's processor requirements, a utilization percentage, a computation time, and the length of an interval in which the computation must occur. The three variables are related by the following expression: $\rho = \frac{C}{T}$ where C is the computation time, and T is the interval in which the computation must occur. This analysis allows straightforward application of the results presented by Liu and Layland [24], for the purpose of determining the feasibility of the real-time task set and is what they use for admission control.

The third issue requires that the scheduler must be consistent across all resource management policies in the system and that the scheduler and admission control policy must agree with respect to task ordering, preemption, and measurement.

Issue four is the most complex. Consider a server process with a number of threads, such as the server used for a graphical user interface (GUI). The threads of the server often operate on behalf of some application process, so the CPU cycles used by those server threads should be charged to the task on whose behalf they operate. Thus the accounting of CPU cycles cannot be done simply on a per process basis.

To deal with this problem the authors introduce the concept of an activity which is an abstraction that includes all activities performed within a computer system on behalf of some particular process. Activities, rather than processes, reserve processor capacity. A measurement mechanism accurately measures the processor time accumulated by each activity. Activities that have not yet consumed their reserve have a higher priority than activities that have.

The measurement mechanism is driven by an auxiliary timer board. The system described by Mercer et al. [25] does recognize that interrupt handlers artificially increase the amount of processor time a scheduler attributes to a particular task, because their measurement mechanism specifically excludes the execution time of interrupt handlers. The processor capacity reserve system does not, apparently, provide a mechanism to control the inherently high priority of interrupt handlers. Thus, if many interrupt handlers execute during an activity's period, too few processor cycles may remain to complete the activity. Like RT-Mach, PCR is an instance of modifying the complete operating system to support real-time tasks.

2.2.4 Rialto

The Rialto system being developed at Microsoft Research attempts to incorporate into their operating system high-level reasoning and dynamic inheritance of scheduling attributes, across boundaries (such as modules and RPC) traditionally closed to such inheritance, to support real-time computation [22, 21]. Rialto defines and uses programming and system abstractions that make it easier to manage computer system resources. Each resource providing a service can define how much resource it needs to accomplish that service. For example, a module which implements a disk read operation would specify a set of resources, such as CPU time, bus bandwidth, and memory, that it needs to actually accomplish the read. The values would be for the worst case. A process can use these data to reason about its temporal requirements. A resource planner negotiates resource usage between processes and resources, eventually accepting or rejecting a request from a process for a set of resources. A mix of real-time and non-real-time activities is supported in this system since processes not requiring real-time access to resources can negotiate completion times with long deadlines. However, such negotiations are arguably unnatural.

This resource model closely resembles the programming model of the SP architecture, in that resource use has an associated cost that is considered part of the processor demand of the real-time task. However, SP does not expose these costs to the tasks but incorporates them as it calculates the cost of the real-time thread. Real-time tasks in SP simply state their requirements. As long as the utilization of the resources requested is below 100 percent after the request is considered and the resulting schedule of real-time tasks is feasible, the request is granted. Negotiation could be added to the SP architecture and, in fact, the **Rialto** abstractions could be readily accommodated in an implementation of the SP architecture. Issues surrounding the sharing of attached physical devices are not addressed.

2.2.5 An IPC Mechanism for Continuous Media

The preceding sections have discussed work addressing the modification of complete operating systems. The next sections describe work which modifies only part of the operating system, and is therefore closer to SP.

In some of the preceding discussion, the method of accommodating real-time computation begs the question of how much of the GPOS must support real-time computation. The answer is all: a real-time task must be able to access any required resource within a known upper bound on latency. SP largely avoids this question, because it separates real-time computation from non-real-time computation via the CPU executive. Even in SP, however, resource executives must ensure that their managed resource is available when a real-time task attempts to access it.

Anderson et al. consider OS support for continuous media [11]. They address the issue of which OS function should be modified to accommodate real-time access by noting that resource conflict occurs during user/kernel domain switches and mapping (context) switches. The work tries to minimize the latencies caused by these switches by adding a new process and scheduler structure and a new process-to-kernel communication mechanism. They note that in the UNIX asynchronous I/O mechanism, ten domain switches and two mapping switches are required to read a block of data. The authors make a case for reducing the amount of overhead incurred when processes cross domain boundaries in typical UNIX implementations [11]. They have designed two mechanisms that help reduce the latencies that arise from switching: split-level scheduling with synchronization and memory-mapped streams.

Split-level scheduling with synchronization minimizes user/kernel interaction by using several schedulers (one kernel scheduler and a scheduler for each user address space) rather than only one, and by allowing more than one lightweight process in a single user address space. The kernel scheduler decides which user address space should execute, and the user level scheduler in the indicated user address space decides which lightweight process to execute. The kernel and user schedulers communicate via shared memory and minimize the number of domain and mapping switches.

Memory-mapped streams reduce the number of domain switches by providing a queue for kernel work requests. These requests are placed on the queue by the lightweight processes, and removed and acted upon by the kernel. Using this queue reduces the number of domain switches.

Anderson's design attempts to modify some pieces of a typical GPOS (UNIX) to provide the timing and scheduling support required by real-time tasks. The issues of interrupt disabling and interrupt handler execution are not addressed.

2.2.6 Summary

Several systems claim to support a mix of real-time and non-real-time computation. From their descriptions one can infer that (1) modifying the scheduler of a GPOS without modifying other OS components is insufficient; (2) the crucial details of handling interrupts and the unrestricted disabling of interrupts within the GPOS must be considered in any modification of a GPOS or system design; and (3) priority scheduling alone cannot schedule a mix of real-time and non-real-time tasks.

2.3 Scheduling Work

This section describes methods of supporting real-time tasks in a GPOS by defining new scheduling models. Although the problem of mixing real-time and non-real-time tasks transcends scheduling, it is nonetheless instructive to examine some work that addresses the scheduling issue in isolation. We consider three new models: lottery scheduling, proportional share allocation, and rate-based execution.

These algorithms address the issue of mixing real-time and non-real-time processes in the context of a GPOS, but none accounts for latencies at scheduling boundaries due to the disabling of interrupts or for the effect of interrupt handlers executing below the visibility of the scheduler.

2.3.1 Lottery Scheduling

In lottery scheduling, the CPU is scheduled by randomly selecting a process through a lottery [37]. Processes obtain a number of 'tickets'. The share of a resource that a process eventually receives is proportional to the number of tickets a process holds. A random number generator is employed to select the winning ticket and the ready process holding the winning ticket is given access to the resource. A number of implementations of the ready queue are given.

The authors state that this scheduling mechanism is responsive and flexible and can fairly allocate processor cycles among real-time and non-real-time processes. Responsiveness is the scheduler characteristic that measures its ability to respond to changes in the ratio of allocated tickets. Fairness is defined as the measure of the actual ratios of execution time to ticket allocations among tasks. Flexibility explains that the scheduler accepts changes in relative ticket allocation among tasks. For example, suppose that a lottery is held every millisecond and there is one real-time task, with a period of 30 milliseconds and a cost of two milliseconds, and 28 non-real-time tasks. The real-time task should then get two tickets and all of the non-real-time tasks should get one ticket each. In every 30 milliseconds, then, the real-time task will probabilistically receive two milliseconds of processor time.

Lottery scheduling solves the priority inversion problem by allowing ticket transfers. When a real-time task blocks, it can transfer its tickets to the task for which it is waiting, which increases that task's ability to obtain access to the resource.

2.3.2 Proportional Share Resource Allocation

Lottery scheduling is a form of proportional share allocation. Other relevant work in this area includes an algorithm that successfully mixes real-time and non-real-time scheduling so that all processes in the system progress at precise, well-defined, uniform rates [34]. The algorithm is earliest deadline first, but it supports services of both the GPOS and real-time tasks. It is essentially a cross between processor sharing and typical periodic real-time scheduling.

Every process is assigned a weight from which a share of the processor is calculated. If a process's share of the processor is s, then in any interval t, that process is guaranteed to receive $st \pm \epsilon$, where $0 \leq \epsilon \leq \delta$ for some constant δ. Since all processes progress at their defined rates, no distinction is made between real-time and non-real-time processes or their threads. The eventual goal of this work is to use proportional share allocation for all resources, including interrupt handlers and I/O buses. Using the example of the previous section, the real-time task should be assigned a weight such that its share would be $0.0\overline{66}$. Then in each interval of 30 milliseconds, the real-time task would receive two (30×0.066) milliseconds ($\pm\epsilon$) of processor cycles.

2.3.3 Rate-Based Execution

Jeffay and Bennett propose a model, *rate based execution* (RBE), in which processes state their timeliness requirements via a rate expressed in activities per time unit [19]. For example, a process may require enough resources to process one video frame every 33 ms. The model is general enough to encompass most traditional periodic and sporadic task models, as well as some proposed multimedia computation models. It also allows the integration of non-real-time activities with real-time computation.

The model defines a triple (x, y, d), where x is the number of events that must be processed in an interval of length y, and d specifies the desired maximum elapsed time between the arrival of an event and the time at which its processing is complete. When $x = 1$ and $d = y$, the RBE model is identical to the Liu and Layland periodic task model [24]. RBE has been implemented on YARTOS (Yet Another Real-Time Operating System) and Real-Time Mach.

Although RBE supports non-real-time and real-time activities, its focus is different from the SP architecture. The SP architecture develops a methodology for adding support for real-time computation to existing GPOSs whereas RBE defines an abstraction for process execution. So, for example, one could use RBE in an implementation of the SP architecture where RBE handles the scheduling and SP ensures that real-time tasks obtain the number of processor cycles granted to them by the RBE scheduler.

2.3.4 Summary

Traditional operating system scheduling theory addresses the issue of fairness: does the particular algorithm allocate resources in a manner such that all processes receive a fair share of the scheduled resource in a timely manner? In scheduling real-time tasks, however, fairness is irrelevant. If the algorithm schedules real-time tasks correctly and the task set is feasible, then every process receives what it requires to meet its deadline.

When an algorithm attempts to schedule both real-time and non-real-time processes, the fairness issue becomes relevant and is more important than in an algorithm that schedules only non-real-time processes. This is because, informally, the real-time tasks execute to completion before their deadline and thus consume processor cycles that could have been used for the non-real-time tasks. This condition requires that the scheduling algorithm be careful about allocating the resource to the non-real-time tasks.

The three algorithms presented in this section accomplish this goal efficiently and correctly. They are important because they can be implemented in a traditional operating system and are thus promising for adding support for real-time computation to GPOSs. However, we do not yet know how much of the original GPOS would have to be modified to incorporate these new scheduling models.

2.4 Extensible Kernels

The SP architecture proposes a configuration where two separate operating system kernels share the physical devices of a computing system, but it does not define when the second kernel and its supporting systems become operational. Systems have been proposed and implemented that allow the dynamic, run-time extension of a GPOS kernel. Such systems might allow the addition of a real-time kernel, or its functions, at run-time.

The current implementation of SP is not this general, i.e., it is linked statically with the GPOS kernel and they are both loaded at boot time. Although the real-time kernel does not become active until initiated through I/O from the operator, the interrupt virtualization mechanism is active from the initial load of the kernels. The virtualization mechanism could be implemented to become active immediately before the RTK begins operation, which would allow the dynamic addition of an SP implementation to a GPOS kernel. However, such a system would be more sophisticated than current extensible kernel architectures.

This section looks at two architectures that represent possible approaches to dynamically extending an operating system kernel to support periodic real-time tasks.

2.4.1 SPIN

The SPIN project attempts to facilitate the addition of services in an operating system by viewing function and procedure calls as events that can be dynamically bound to various handlers [29]. The basic SPIN kernel consists only of device access, dynamic linking, and event handling. All other typical GPOS services, such as threads and virtual memory, are provided by applications as needed. Although not explicitly mentioned, one assumes that an application could install various task schedulers dynamically, which would enforce different scheduling policies. So, for example, a lottery scheduler and its associated necessary run-time data structures could be installed.

One can imagine an implementation of the SP architecture being installed dynamically as long as the necessary virtualization of the interrupt enabling/disabling mechanism already existed within the basic SPIN kernel. Minimally, all attempts to enable or disable interrupts should already be function or procedure calls, which would allow the dynamic binding of new handlers.

2.4.2 ExoKernel

The ExoKernel proposes an operating system architecture that exports the capabilities of the physical hardware and provides only protection; the goal here is to separate protection from management [8]. For example, the ExoKernel protects (prevents access by unauthorized processes to) a component such as a frame buffer, but does not attempt to manage it, i.e., it does not maintain a free list, etc. The ExoKernel architecture was developed because applications can optimize use of hardware (since they know intimate details about their operation) and because typical GPOSs attempt to provide general resource management at the expense of certain classes of applications. All operating system services are provided by an operating system library linked with each application.

The ExoKernel securely exposes the hardware by *secure bindings*, which protect applications from interfering with one another while enhancing performance by decoupling authorization from use. The ExoKernel also provides revocation actions that require it to visibly revoke resources from applications. Visible revocations allow the applications to manage application-level resources effectively because the application-level libraries are aware of how resources are allocated. Results of experiments show that the particular implementation of the ExoKernel is significantly faster than a typical GPOS for operations such as an exception dispatch (a source of overhead in traditional operating systems) [8]. Applications can also perform scheduling for themselves as well as for other applications.

Since the ExoKernel architecture exports an interface just above the physical hardware, it seems likely that an implementation of the SP architecture could easily be constructed within the ExoKernel. An application would define the minor cycles and dispatch real-time tasks appropriately.

2.4.3 Microkernels and Virtual Machines

Another approach to dynamically adding function to an active kernel is the creation of virtual machines. One example is the Fluke (*Flux μ-kernel Environment*) operating system architecture [9] which enhances OS modularity, flexibility, and extensibility. The Fluke architecture allows deep hierarchies of virtual machines to operate efficiently. It is a nested process architecture, similar to a virtual machine architecture. However, it does not incur the performance degradation of virtual machines when they occur in layers.

The Fluke architecture is specifically designed to avoid several problems:

- the requirement of having to emulate all instructions because sensitive information, such as privilege level, leaks into user-accessible registers
- performance penalties that increase exponentially with stacking depth
- only parent-child interprocess communication

Instead of the traditional virtual machine architecture, where each virtual machine exports a hardware interface, in this work each virtual machine exports a software interface designed specifically to efficiently support deep hierarchies. The virtual machines, called nesters, nest within each other. Each exposes the same interface, thereby allowing modular composition of operating system services. For example, the demand paging nester need not be in the hierarchy at the base level. It can be at a level where only other virtual machines and applications that require demand paging have access to demand paging routines and therefore incur their cost. The performance degradation experienced with deep hierarchies is reduced by allowing microkernel primitives to be used directly by processes without having the process's parent involved in the operation. That is, the child process imports microkernel operations directly from the microkernel rather than from its parent process.

Since Fluke is implemented by a microkernel at the hardware layer, implementing SP in the Fluke environment would involve much the same issues as implementing SP in a typical GPOS. The interrupt enable/disable mechanism would have to be virtualized and an interrupt handler would have to be installed to create the minor cycles needed by SP.

2.4.4 Summary

The question posed in this section is: can real-time support be added to a operating system kernel by adding function via provided extension mechanisms? We claim that it can be done only by adding basically all of the functions of the operating system, as Real-Time Mach did. This approach, although feasible, requires tremendous effort and in the end must address the same issues that SP addresses to be successful. These issues are (1) processor cycle stealing of interrupt handlers, (2) precise control over the starting time of intervals in which real-time processes execute, and (3) ensuring the availability of resources when real-time processes require access.

2.5 Time Division Multiplexing

The SP architecture CPU executive manages the GPOS and the RTK by interleaving their executions. One may view this alternate execution as dividing the CPU's capacity in time. Paradigms to facilitate the efficient use of a communications link, such as a telephone network's trunk lines, have long included the notion of time division multiplexing [35]. Consider a physical communications link with bandwidth β and a number of distinct logical channels each of which carries a conversation on behalf of a link user. Time division multiplexing allows all of the logical channels to share the physical link. If n channels are multiplexed, each receives $\frac{\beta}{n}$ units of bandwidth. Access to the physical link by each channel is allowed only in a specified time interval, each channel sending on the physical link during its own interval.

The SP architecture specifies that the RTK and GPOS share the CPU in a manner similar to how time division multiplexing multiplexes a number of logical channels over a single physical communications link. The executions of the RTK and the GPOS occur in minor cycles and this sharing is managed by the CPU executive, which is similar to the controllers that divide use of the physical communications channel.

2.6 OS Emulation

It is fairly common to find operating systems emulating other operating systems. This is typically done to provide some level of backward compatibility, i.e., allow users to execute programs which were written for an operating system other than the dominant one running on the system. Two examples are given below. However, note that neither host operating systems nor the emulated operating systems claim to have any hard real-time properties.

2.6.1 Virtual Machines

The term virtual machine typically refers to software that emulates a particular hardware system, but need not be executed on any particular hardware. There are different ways this capability might be used to one's advantage. One can envision many different virtual machines sharing a single hardware system. For example, a virtual PS/2, a virtual DecStation, and a virtual Macintosh might all simultaneously execute on a single hardware system and within each virtual machine an appropriate operating system and its associated processes would execute. Thus, with a single hardware system an enterprise might accommodate many users with diverse requirements.

Another use of virtual machines might be to provide each user with the illusion of a complete single user machine while actually executing many identical virtual machines, each with its own user, on a single hardware system. This is the use provided by IBM's Virtual Machine (VM/370) [15, 16]. VM/370 is a micro-kernel architecture, a small kernel interacting directly with the physical hardware and exporting the 360 architecture. The VM/370 micro-kernel is called the control program (CP). VM/370 builds a complete virtual hardware system and operating system for each user and controls all access to the hardware via calls from the virtual operating system to the CP.

The CPU executive in SP virtualizes the underlying hardware somewhat differently. It isolates, in time, various functions to provide two different uses of the same hardware. In contrast, a virtual machine provides the illusion of complete machines to be used as the executing operating system decides. Figure 2.1 shows the relationship of virtual machines to their hardware system.

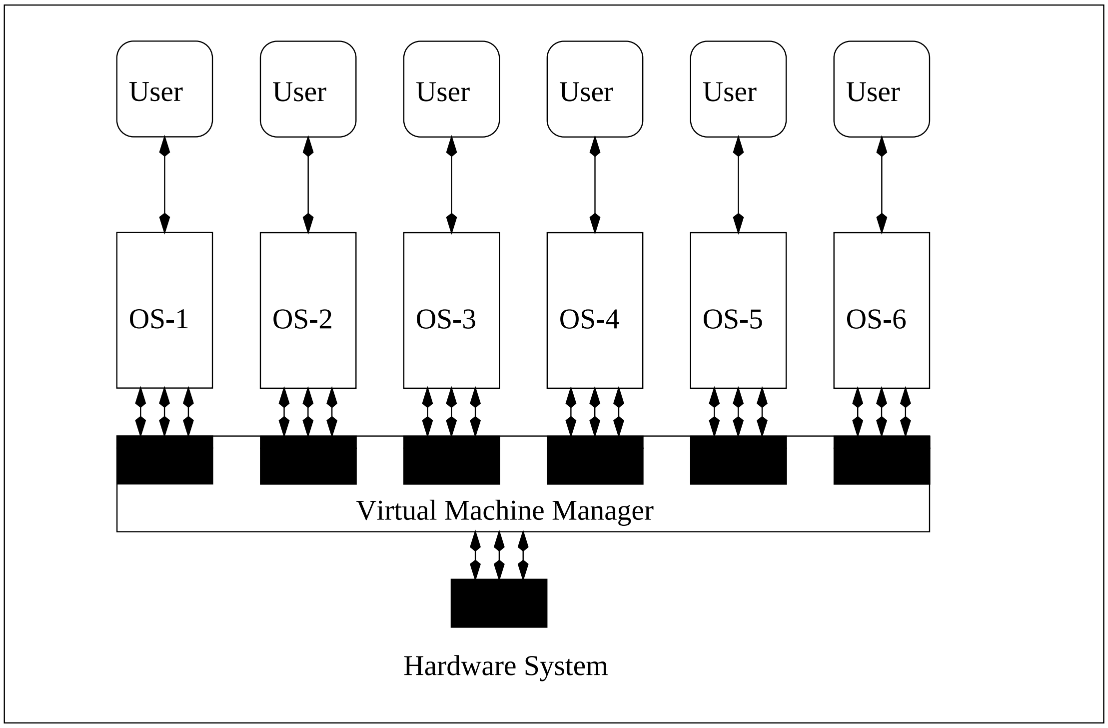

Figure 2.1: Emulation of Operating Systems in VM

2.6.2 OS/2

IBM's OS/2 operating system provides the ability to execute programs written for DOS or Windows in an environment that emulates the original operating system. Almost all programs written DOS or Windows can run in the simulated environment without modification. OS/2 provides this environment by creating a virtual DOS environment (VDM) for each DOS program that is executed, using the multiple virtual DOS machine (MVDM) technology [5]. The MVDM is called a kernel and is scheduled by OS/2 as a single OS/2 task; hence, it is subject to all of the unpredictability of an OS/2 task.

SP can be viewed as two operating systems executing on a single hardware system and therefore is similar to executing DOS within OS/2. The difference is that in OS/2 the DOS environment is scheduled by the OS/2 scheduler and receives CPU cycles like any other OS/2 process. In SP, the GPOS scheduler does not know that the RTK is executing in the real-time minor cycles. Figure 2.2 shows how the interrupts and traps are received directly by OS/2, then issued to the appropriate device drivers. The device drivers interact with a set of virtual device drivers associated with the MVDM, which are shared among the instances of DOS.

2.6.3 RTX 4.1

VentuCom currently advertises a commercial product, RTX 4.1 [4], that adds support for a particular type of real-time computation to a general-purpose operating system. The product is described as a real-time extension to Windows NT. Windows NT provides a hardware abstraction layer (HAL) which virtualizes hardware resources. The programming interface of RTX 4.1 is called the Real-Time Application Programming Interface (RTAPI). RTX 4.1 does not support periodic real-time threads nor does it have a real-time thread scheduler. RTX 4.1 only guarantees that for an interrupt identified by RTAPI calls to the extension, an associated

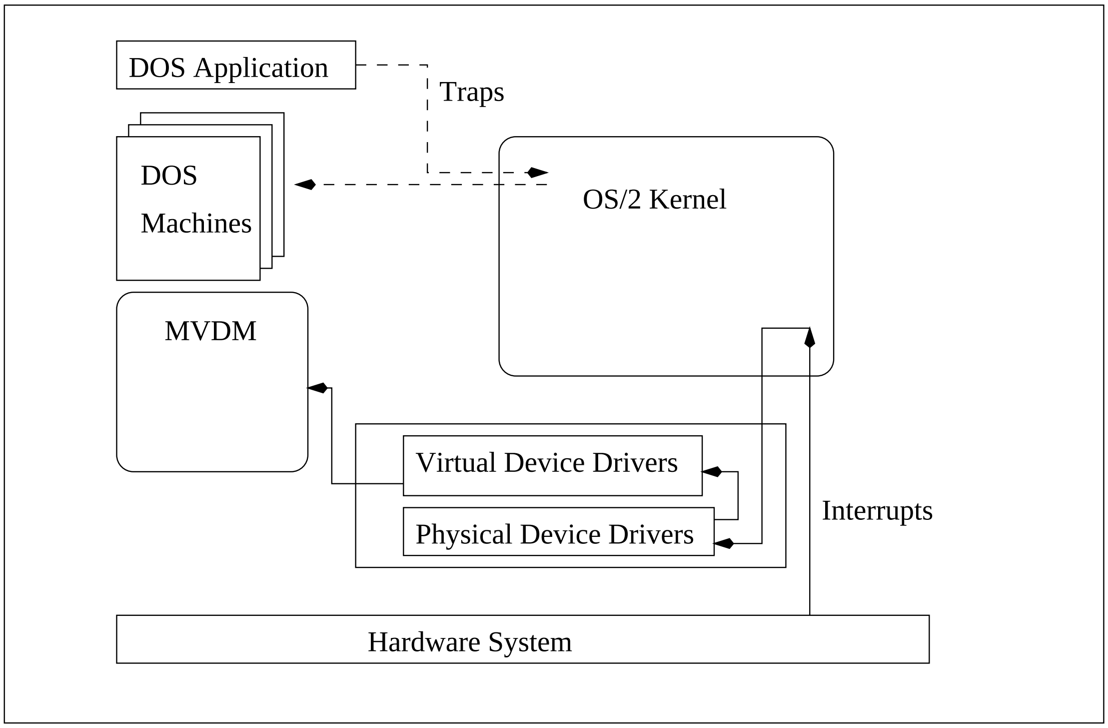

Figure 2.2: Emulation of DOS in OS/2

interrupt handler will begin execution within a defined time after the interrupt occurs.

2.7 Summary

The chapter describes work related to the design of SP. SP mixes the execution of real-time and non-real-time tasks on a single hardware system. Other approaches to this problem include Real-Time Mach, Processor Capacity Reserves, Rialto, and various scheduling schemes.

Mixing execution of the two task types transcends scheduling. In a real computer system, just modifying the scheduler to accommodate real-time tasks is insufficient. Any resource used by a real-time task must be available, with a reasonable upper bound, when the real-time task requires it. This extends to all hardware and software subsystems, including memory subsystems, disk drives, network adapters, virtual memory subsystems, and file systems.

This requirement also applies to SP, but SP avoids the issue by clearly separating real-time portions from non-real-time portions of tasks. For example, the Mach virtual memory subsystem must be modified, creating Real-Time Mach, so that no stochastic delay occurs when an application requests memory. Compare that to SP where memory is partitioned and there is no virtual memory for real-time tasks executing within the RTK. The time to perform memory allocation for these tasks is therefore deterministic.

Although more than scheduler modification is needed within a GPOS to support real-time computation, it is nonetheless useful to look at schedulers such as lottery scheduling and proportional share allocation that claim to support both types of tasks. Again SP avoids the issue. Instead of requiring a scheduler to support both task types it schedules task types independently: non-real-time by the GPOS scheduler and real-time by the RTK. Thus two simpler, and well understood, schedulers may be employed. For example, in the implementation described in Chapter 5, the RTK scheduler is a straightforward earliest deadline first scheduler—a well known approach.

Can a GPOS that supports real-time computation be created by adding function to a system in which one can extend the kernel? Yes, if all of the functions needed by the real-time tasks are modified to provide deterministic access times.

Chapter 3

The Slotted Priority Architecture

3.1 Introduction

The SP architecture describes a possible modification of general-purpose operating systems to schedule, manage, and execute periodic real-time threads. It comprises three components: the execution, resource, and programming models. An implementation of the SP architecture includes, among other constructs, executives that share resources between the GPOS and the RTK.

The execution model describes the operation of the processor executive, which enables processor multiplexing between the GPOS and the RTK. The resource model classifies resources according to characteristics that determine the structure of an appropriate executive. The programming model defines the interface presented to programmers who write periodic real-time threads to be executed on an implementation of the SP architecture. It also defines the interaction between the real-time and non-real-time threads of a program.

Section 3.2 presents the overall design of the SP architecture. Section 3.3 derives the two requirements that must be met by any implementation of the SP architecture. Section 3.4 describes how the two requirements might be realized. Section 3.5 describes how one can argue that an implementation of the SP architecture will be able to guarantee that a set of feasible periodic real-time threads will always meet their deadlines. Sections 3.6 through 3.9 describe the execution, resource, and programming models that comprise the SP architecture.

In the SP architecture, executives share resources between real-time and non-real-time threads. When the SP architecture is implemented on a particular computing machine, an executive will control access to each resource associated with the machine and the GPOS used by real-time threads. The execution model, described in sections 3.6 and 3.7, is the executive that shares the CPU, a resource, between the RTK and the GPOS.

Section 3.8 describes the resource model, which has eight classes. A resource falls into a particular class based on characteristics that determine the structure of an executive that can correctly manage how that resource is shared.

Section 3.9 describes the programming model. Two examples are given to show how an application programmer might write code for two different problems. The problems can be solved only by real-time threads. The first example shows the data decompression and display stage of a postulated video pipeline from a stochastic network to the display device. The second example shows a stage in a postulated manufacturing assembly line in which final details of a manufactured object are recorded to disk.

3.2 A High Level Description of the SP Architecture

Central to the SP architecture is the sharing of the CPU between the GPOS and the RTK. Figure 3.1 illustrates an operating system view of the execution model defined by the SP architecture. The GPOS and the RTK share the CPU by executing in alternate intervals called real-time minor cycles and non-real-time minor cycles. Within the execution model, the SP architecture specifies the structure of three components:

- the method by which the CPU is shared between two operating system kernels (the GPOS and the RTK)
- the component that manages the sharing, the CPU executive
- a requirement to ensure precise minor cycle lengths and bounded real-time thread execution times

The CPU executive determines the start of the two types of minor cycles from a sequence of interrupts from a hardware timer and invokes its dispatcher to dispatch either the GPOS or the RTK scheduler. The architecture also requires that the interrupts marking the start of the minor cycles not be delayed indefinitely by any action of the GPOS, GPOS device drivers, or attached devices themselves.

The SP architecture suggests useful functions for the RTK to provide: a real-time thread scheduler, libraries of various real-time system calls, an interface to the GPOS and applications used to create and control real-time threads, and an

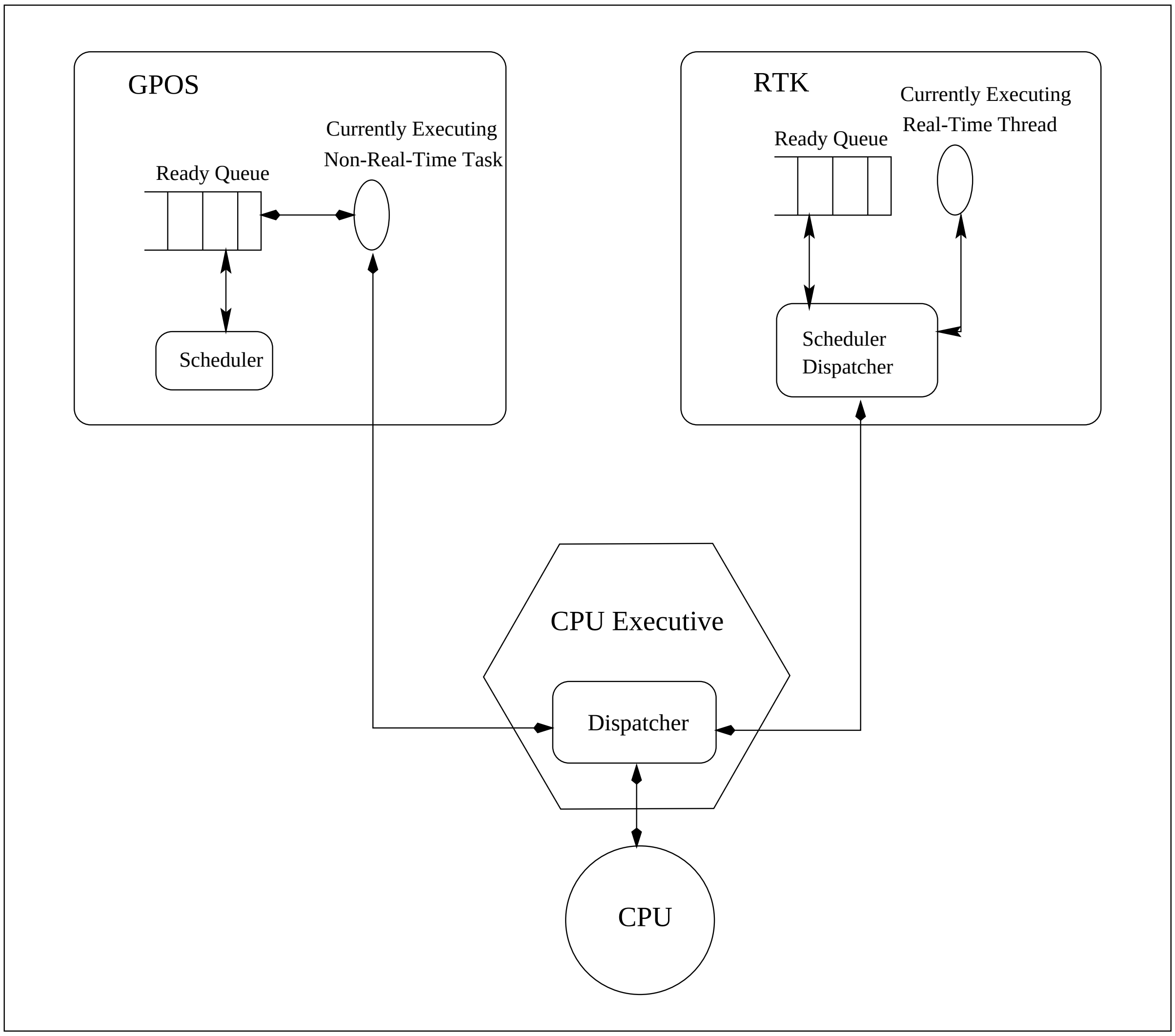

Figure 3.1: An Operating System View of the Execution Model

interface to the GPOS for non-real-time services that real-time threads could use. Currently the architecture has specified, and gives proven feasibility conditions for, earliest deadline first scheduling in the RTK. However, other schedulers could be employed when appropriate feasibility conditions are derived and proven. The architecture in the programming model specifies that the RTK and GPOS together support queues by which real-time threads can send and receive requests for basic non-real-time services provided by the GPOS.

The resource model was designed to help the SP architecture provide useful guidelines for the construction of executives that share resources. The alternative would be the impossible task of providing details on every one of the overwhelming number of devices and device interfaces available today. Similar executives can be used to share, between the RTK and GPOS, resources that have similar characteristics. For example, sharing a resource similar to the CPU (preemptible with state saving and restoration) could be managed by an executive similar to that described in the execution model.

The programming model suggests a paradigm of how programmers should write and think about real-time programs written for an implementation of the SP architecture. Real-time threads are threads of traditional GPOS processes that are scheduled by the RTK and execute in real-time minor cycles. The implementation supports the creation of periodic real-time threads via operating system calls to the GPOS and RTK. These real-time threads then execute in a continuous loop with one wait statement per loop.

An implementation also supports and requires the reservation of resources. Each real-time thread reserves a resource it intends to use within its continuous loop. How a resource is used is independent of its class in the resource model. The programmer does not have to explicitly reserve the CPU; rather it is reserved as a result of the call to create a real-time thread. Any other resource is simply reserved, and the underlying executive and execution model ensure that an instance of that resource is available, within some reasonable upper bound, as needed by the real-time thread.

3.3 Two Fundamental Requirements of an SP Implementation

Two necessary conditions must be true before an implementation of the SP architecture can guarantee that all real-time threads will meet their stated deadlines. A real-time thread has a cost of c time units and a period of p time units. The explicit requirement is that a real-time thread execute for c time units on the processor in each interval $[(k-1)p, kp], \forall k > 0$. However, this definition is not precise enough.

Assume a computing machine on which the CPU operates at fifty million cycles per second, (20 nanoseconds per cycle). A real-time thread with a cost of 200 ms implies that the thread requires 10 million CPU cycles to complete. It therefore requires 10 million CPU cycles in each interval as defined above.

In general-purpose operating systems, the amount of CPU time the scheduler assigns to a thread cannot be used to determine the number of CPU cycles actually used in executing the thread's instructions while it was assigned to the CPU.

There are two reasons for this condition. First, interrupt handlers operate below the visibility of the operating system scheduler. Typical schedulers allocate time on the CPU to threads in fixed quanta during which interrupt handlers may also execute in response to interrupts from hardware devices. Typically, the scheduler code does not account for the execution of interrupt handlers. Thus, although the scheduler has allocated a complete quantum to the thread, in fact, the thread has not received all of the CPU cycles implied by the length of the quantum.

Second, typical computing machines provide DMA, which may cause the CPU to wait for access to the internal memory bus. During this wait period, the thread receives no CPU cycles with the same result as in the interrupt handler case: the quantum expires, yet the thread has not received a quantum's worth of CPU cycles.

Thus, the fundamental requirement given above can now be restated: if a system claims to guarantee that, given a feasible schedule, periodic real-time threads will complete execution before their stated deadlines, it must ensure that each real-time thread receives the number of CPU cycles implied by the magnitude of its cost within each interval $[(k-1)p, kp], \forall k > 0$.

This fundamental requirement has straightforward and profound implications. When the real-time thread scheduler assigns a real-time thread τ to execute on the CPU for an interval of time $[t_0, t_1]$ where $t_0 < t_1$, of length $L > 0$, there must exist a function or method that can precisely determine the minimum number of CPU cycles used to execute the instructions of τ. If L must be larger than some reasonable δ, such as a real-time scheduling quantum, then a function F or method can exist on a real computing machine. The SP architecture defines how this can be accomplished on a computing machine within the context of a GPOS.

The problem is not that interrupt handlers steal cycles from threads, but that the interrupt handler's frequency and length can be determined only stochastically. Consider a system that 1) enforced a minimum interarrival time between invocations of any interrupt handler, 2) set an upper bound on the length of execution, in CPU cycles, of interrupt handlers, and 3) limited the length and frequency of intervals used by the DMA subsystem. For such a system, ignoring other concerns, it is possible to derive a function equivalent to F. However, in a GPOS it is difficult, if not impossible, to enforce a minimum interarrival time between interrupts and to enforce a maximum execution time for interrupt handlers.

Another fundamental difficulty arises when attempting to mix execution of real-time and non-real-time threads within a typical GPOS, specifically a system that uses a preemptive, quantum-based, round robin scheduler. Assume a system that mixes execution of real-time and non-real-time threads by executing the two types of threads in alternate quanta. Now, consider the quantum in which a real-time thread must finish execution to meet its deadline. If the start of this quantum is arbitrarily delayed, then the thread may miss its deadline. Hence, no guarantee can be made that a real-time thread will finish before its stated deadline if the start of a quantum may be delayed by an arbitrary amount. A second fundamental principle can now be stated: if a system executes real-time threads in regular, periodic intervals, then the system must ensure that the intervals begin when they are scheduled to begin or that they begin with a delay that does not exceed some defined upper bound.

In summary, two requirements must be considered when attempting to implement the SP architecture. There must be a method to determine how many CPU cycles are dedicated to a real-time thread executing in the intervals for real-time threads, and the intervals in which real-time threads execute must begin no later than some known upper bound. More formally:

Requirement $\mathcal{A}$ If a system interleaves the execution of real-time and non-real-time threads in alternate intervals and the intervals in which real-time threads execute are scheduled to begin every l time units, then it must ensure that the intervals begin at times t where $kl \leq t \leq kl + \epsilon$,$\forall k \geq 0$ (where ϵ is a small, implementation dependent integer value).

Requirement $\mathcal{B}$ For $L > \epsilon$, (for a suitable ϵ) for which the real-time thread scheduler has assigned a real-time thread, τ, to be executed on the CPU, there must be a function or method by which the minimum number of CPU cycles available to execute the instructions of τ can be determined.

These two conditions are necessary for any implementation to claim that it implements the SP architecture. Note that requirements $\mathcal{A}$ and $\mathcal{B}$ are necessary but not sufficient to reason about the real-time performance of real-time threads. One still needs a task model, scheduler, kernel, etc. The analysis in Chapter 4 describes the necessary and sufficient conditions for determining the feasibility of a set of real-time threads that will execute on an implementation of the SP architecture.

3.4 Realization of the Two Requirements

The realization of requirement $\mathcal{A}$ stated in Section 3.3 requires that the signal used to transfer control from the GPOS to the RTK must never be disabled. In a typical system this signal is an interrupt and typical OS implementations disable interrupts for two reasons. First, most operating systems do not support the nested execution of interrupt handlers and so they disable interrupts during their execution. Second, many OS implementations use the disabling and enabling of interrupts to enforce mutual exclusion in critical sections within the operating system kernel. Thus an implementation of the SP architecture must be able to selectively disable interrupts, so that the GPOS cannot disable the interrupt used to switch between the GPOS and the RTK. However, the interrupt may need to be disabled by the RTK at certain points or by modifications to the GPOS made by the implementer of the SP architecture.

Our method for realizing requirement $\mathcal{B}$ can be best understood by considering a simple CPU with no attached devices, caches, interrupts, DMA, pipelining, traps, or RAM refresh. In this simple system we have only memory, a bus, and a CPU. All CPU cycles in this imaginary system are dedicated to executing instructions; nothing else shares the CPU. Moreover, assume no random variables in calculating the number of CPU cycles required to complete a particular instruction. Thus, the rate at which the sequence of instructions executes can be calculated precisely, and the system would have the function described in Section 3.3; i.e., for any interval of length L, the number of CPU cycles available for execution of a specified sequence of instructions can be determined. Although such a system may be useful for specialized applications, it is not useful for general-purpose computing because it can do no I/O and would perform poorly.

However, if a typical general-purpose computing machine could be precisely and periodically converted into such an imaginary, simple system, during those intervals, a function F or method as described in Section 3.3 could exist. These intervals could be called the real-time intervals. This is the essence of the SP execution model: to operate the computer system such that periodically, precisely, and for reasonable length intervals, the minimum number of CPU cycles available to execute a particular sequence of instructions can be precisely determined.

So far, this hybrid system can only execute CPU instructions during the special intervals when it is acting as a simple imaginary system. (The system is behaving as a normal computing system with a GPOS during the other intervals.) Without interacting with attached physical devices during the special intervals, executing CPU instructions alone is not very useful. So the next addition to the hybrid system is controlled access to attached devices.

Section 3.8 describes model SP uses for resources in detail. For now it is sufficient to say that for each instance of a resource, the SP architecture requires that an executive multiplex access by the GPOS and the RTK to that resource. Essentially, the executive ensures that any object required for access to the resource is available in the real-time intervals in which real-time threads have reserved that resource. One can then compute the cost, in time units, of accessing the resource, since no stochastic quantities are involved. And if the cost can be computed, the access to the device can be considered as part of a real-time thread, and the real-time scheduler can schedule the combination of the real-time thread and the instructions used to access the device, thus guaranteeing completion before the deadline.

The imaginary system disallows attached devices, caches, interrupts, traps[1], DMA, and pipelines. Since real systems have all of these items, an implementation of the SP architecture must accommodate their existence. The items fall into two categories. The first category contains those items that would invalidate a function F as described in Section 3.3—interrupts, traps, and DMA. For example, an interrupt handler may, for a relatively large interval, dedicate all CPU cycles to a sequence of (the handler's) instructions that is not the sequence belonging to the thread assigned to the CPU. The second category contains those items that increase variation in the number of CPU cycles needed to complete any particular instruction—caches and pipelines.

An implementation of the SP architecture requires either that interrupts be disabled during real-time intervals or that the host system be able to enforce a minimum interarrival time between the execution of any two interrupt handlers from the same source and to enforce a strict upper bound on the number of CPU cycles any interrupt handler may consume in a particular invocation. Because

[1]In this work an interrupt is considered to be a signal generated in the CPU by hardware that transfers control from the currently executing sequence of instructions to a defined interrupt handler. These signals must occur asynchronously with respect to the executing sequence of instructions. Traps are considered similar to interrupts, but their occurrence is synchronous with respect to the executing set of instructions.

most general-purpose computer systems lack the latter capability, the interrupts must be disabled during real-time intervals.

Essentially the same requirements are placed on traps and DMA, but a typical general-purpose computer system can constrain their activity. In the implementation described in Chapter 5, DMA is constrained by the bus design. In that design, the IBM MicroChannel^{TM}, no device is allowed to hold access to the bus for more than 7.8 microseconds if another device requests it. Thus active DMA can steal only a bounded number of CPU cycles from the scheduled thread and hence is constrained.

The programming model of the SP architecture ensures that some traps, such as page faults, do not occur. Other traps under programmer control, such as divide-by-zero, must be restricted by programming practice. SP accommodates the existence of caches and pipelines by noting that their effect is to reduce the actual number of cycles used by a sequence of instructions of a real-time thread. Thus, as long as the cost of a real-time thread is calculated without including any optimizations from caching and pipelining, the result may be that the real-time threads use less of the CPU than scheduled. The SP implementation returns this time to the GPOS for use by non-real-time threads.

3.5 Arguing that an Implementation of the SP Architecture Can Meet Guarantees

The SP architecture is a set of methods and guidelines for modifying a GPOS and a hardware computing system to support periodic real-time thread execution. This section describes how to structure the argument that a set of real-time threads, when executed on a particular implementation of the SP architecture, will all meet their deadlines.

The argument has two steps. The first step, deferred until Chapter 4, is to determine if the real-time threads meet the assumptions and conditions of a feasibility test. If so, the real-time threads have a feasible schedule. The second step of the argument is to show that a particular system implements the two fundamental requirements given in Section 3.3. This second step is argued based on the results of experiments on the implementation and by an analysis of the source code of the target GPOS and hardware specifications of the target hardware platform.

An example of this argument is given in Chapter 5, which describes the experiments performed on the implementation and analyses of the GPOS source code and hardware specifications of the target system to verify that that implementation meets these requirements.

3.6 Execution Model

The execution model defines how real-time threads acquire the CPU. The CPU executive described by the execution model must enforce precise minor cycle lengths, typically with an interrupt from a system timer. If the SP architecture is being implemented on a computer system with unrestricted use of the interrupt enable/disable mechanism, the executive must ensure that the interrupt that marks the minor cycle boundaries is not unduly delayed due to the disabling of interrupts by the GPOS.

An implementation accomplishes this function with a virtualized interrupt enable/disable mechanism. This mechanism is hardware-specific and each implementation of the SP architecture must be designed to best accomplish the virtualization. The SP architecture only specifies that minor cycles lengths must be precise. Basically, the interrupt virtualization mechanism must protect the interrupt that marks the minor cycle boundaries when the GPOS disables interrupts. This implies that an interrupt can occur during intervals in which the GPOS has requested that no interrupts occur.

Consider the interrupt that marks the beginning of a real-time minor cycle. Two conditions need to be considered. In the first condition, the GPOS is in a critical section and has disabled interrupts to enforce mutual exclusion. As long as no code is executed in the RTK or its real-time threads that accesses the object being protected in the GPOS critical section, then no fault will occur.

In the second condition, the GPOS has disabled interrupts during execution of an interrupt handler, and communication between it and its device is paused. That is, the interrupt handler cannot execute instructions to interact with its device until the the real-time minor cycle is complete. In practice this pause may cause problems because the interrupt handler or the device has not been designed to accommodate such pauses. However, at the expense of some device inefficiencies, we assert that interrupt handlers and their devices can be modified to operate correctly in this environment. Note that the interrupt virtualization mechanism allows the RTK and its threads to execute within critical sections of the GPOS and within the execution of the GPOS device drivers. Careful consideration in the construction of the executive is demanded.

The real-time and non-real-time minor cycles are time constructs and are used by the CPU executive to share the CPU between the RTK and the GPOS. On some hardware platforms it is difficult to arrange for precise interrupts to occur at intervals of varying length (in this case two different lengths, mc_{nrt} and mc_{rt}), but relatively easy to arrange for precise interrupts to occur at intervals of a fixed length. Thus, if it is difficult to obtain precise interrupts at varying lengths, another construct related to the passage of time must be defined. This construct, a *slot*, is defined to be a constant number of contiguous time units. The slot length is determined at the time the system starts and does not change. A slot may contain either real-time time units or non-real-time time units, but not both. Thus, minor cycle lengths are an integral number of slot lengths, so the length of a slot must be less than or equal to the length of the shortest minor cycle. The existence of slots is not crucial to the architecture but is strictly implementation dependent. Figure 3.2 illustrates the relationships between the major cycle, the two types of minor cycles, and slots.

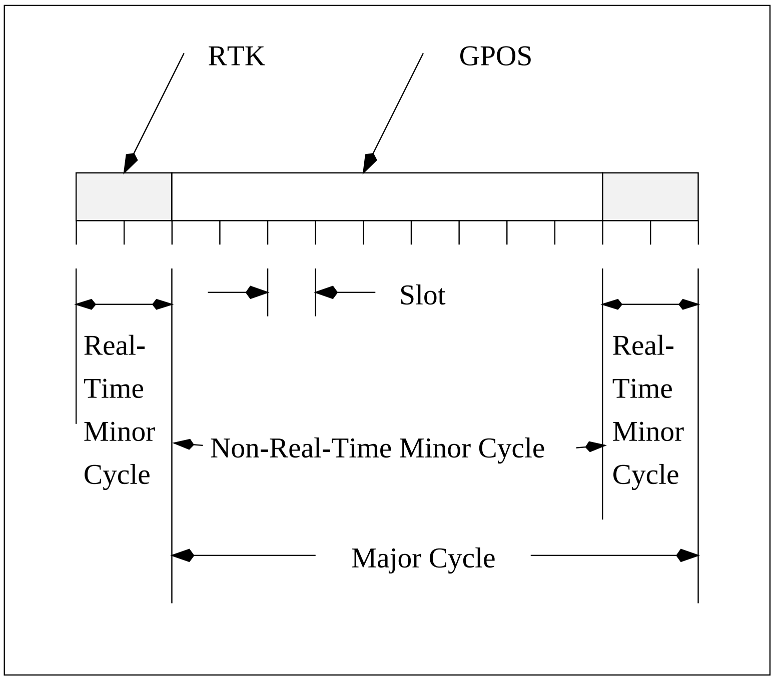

Figure 3.2: Major and Minor Cycles

The low-level dispatcher that runs at each slot boundary handles the switch between real-time minor cycles and non-real-time minor cycles. At each execution either (1) no switch occurs, (2) the real-time kernel state is saved and the GPOS state is installed in the CPU, or (3) the GPOS state is saved and the real-time kernel state is installed in the CPU.

3.7 The Executive

An implementation of the SP architecture includes, among other constructs, executives that share resources between the GPOS and the RTK. An executive is required for each instance of a resource that is to be shared between the GPOS and the RTK. The design, structure, and operation of the executive are driven by the resource model class in which the resource resides.

As an example consider partitionable resources (described more fully in Section 3.8). A partitionable resource is composed of many identical units which can be separated into two disjoint sets, one set each for the RTK and the GPOS. Executives for partitionable resources are relatively simple, because they only need to partition the units of the resource into two sets, allocate one to the GPOS and one to the RTK, and manage any requested changes to the partitioning.

The pixels of the display device are a partitionable resource. Consider a typical windowing system such as the presentation manager of the OS/2 operating system. If a periodic real-time thread displays live video, the display executive can arrange to allocate the pixels of one window created and managed by the presentation manager to the real-time thread by sending the coordinates of two corners of the window to the thread. The real-time thread then simply writes appropriate data to the frame buffer for the given coordinates and the video appears within the window boundaries. When the user attempts to relocate the window, the executive first informs the real-time thread to halt writing to the frame buffer, allows the window to be relocated, and then sends the new coordinates to the real-time thread and indicates that the real-time thread can resume writing data to the frame buffer.

Executives for slotted resources are more complicated. Slotted resources, which are described fully in Section 3.8 are resources composed of a single unit which must be shared between the RTK and the GPOS in time. One example of a slotted resource executive is the processor executive described previously. For other slotted resources, the executives must ensure that the real-time threads have a reasonable and bounded wait-time to gain access to the resource. This is accomplished differently for the various slotted resource classes (see Section 3.8).

The executive must disallow access to the resource by the GPOS long enough before the time when a real-time thread will require the resource so that at that point any use of the single functional unit by the GPOS is complete. For example, consider a disk drive. Although many disk adapter cards allow multiple work requests to queue on the card itself, the card's interaction with the device driver usually requires using a single instance of shared memory into which the device driver writes commands for the disk and reads responses from the disk. The executive then must ensure that the shared memory is not currently in use by either the device driver or the disk when a real-time thread requires access to the disk[2].

Although the SP architecture does not differentiate between software and hardware resources, resources are typically physical devices. Figure 3.3 shows the relationship between the traditional organization of physical devices, their device drivers, and the operating system in both a typical GPOS and an implementation of the SP architecture. Figure 3.3b illustrates the relationship of the executive to the device, device driver, and operating system.

3.8 Resource Model

The resource model allows the SP architecture to provide useful guidelines for constructing executives to multiplex resources without having to provide details on every available device and device interface. One constraint placed on this work by the author was that the host GPOS be modified as little as possible. Obviously, real-time threads can be supported if major portions of the GPOS are rewritten. However, rewriting operating systems is difficult under normal circumstances. Attempting to include support for real-time threads complicates the effort significantly. The view taken in this work is that separating concerns, non-real-time and real-time, reduces the overall complexity of the programming effort. The GPOS

[2] Many issues arise in the actual implementation of an executive for a disk drive, which is not covered here, such as whether or not the disk device driver is to be modified. If so, constructing the executive is simpler but the executive must still ensure reasonable and bounded waiting time to access the disk. Some slotted resource devices, such as the IBM 16/4 token ring adapter, may require the executive to ensure that any operation started by a real-time thread completes before permission to access the device is returned to the GPOS.

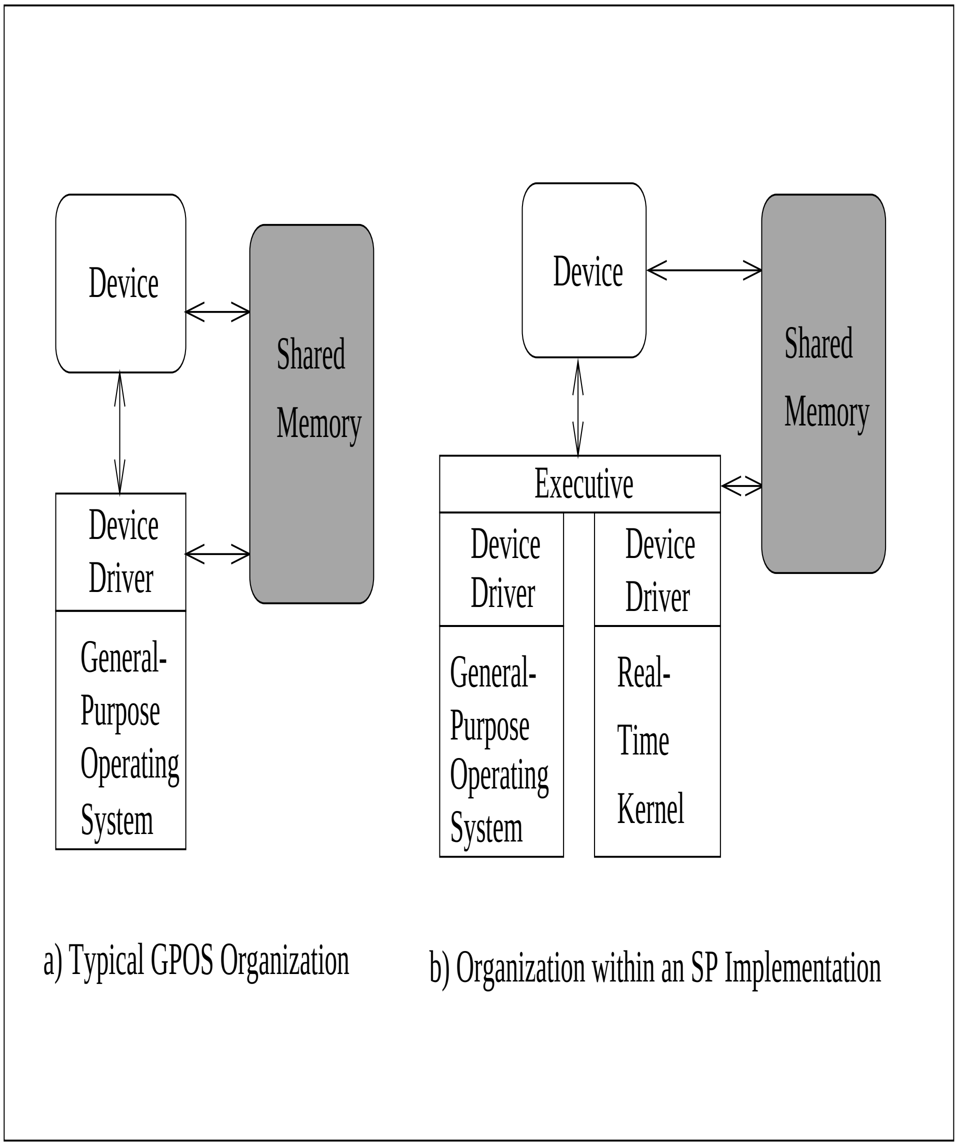

Figure 3.3: Executive Organization

needs to be modified in only the most straightforward manner and the RTK can be constructed in isolation, giving thought to the issues of real-time threads and their requirements. The result of this constraint is the resource model described here.

We develop a hierarchy of resources based on the characteristics of resources that require different capabilities in an executive that can correctly share resources between the GPOS and the RTK. The resource hierarchy is shown in Figure 3.4. All resources are divided into classes: partitionable and slotted.

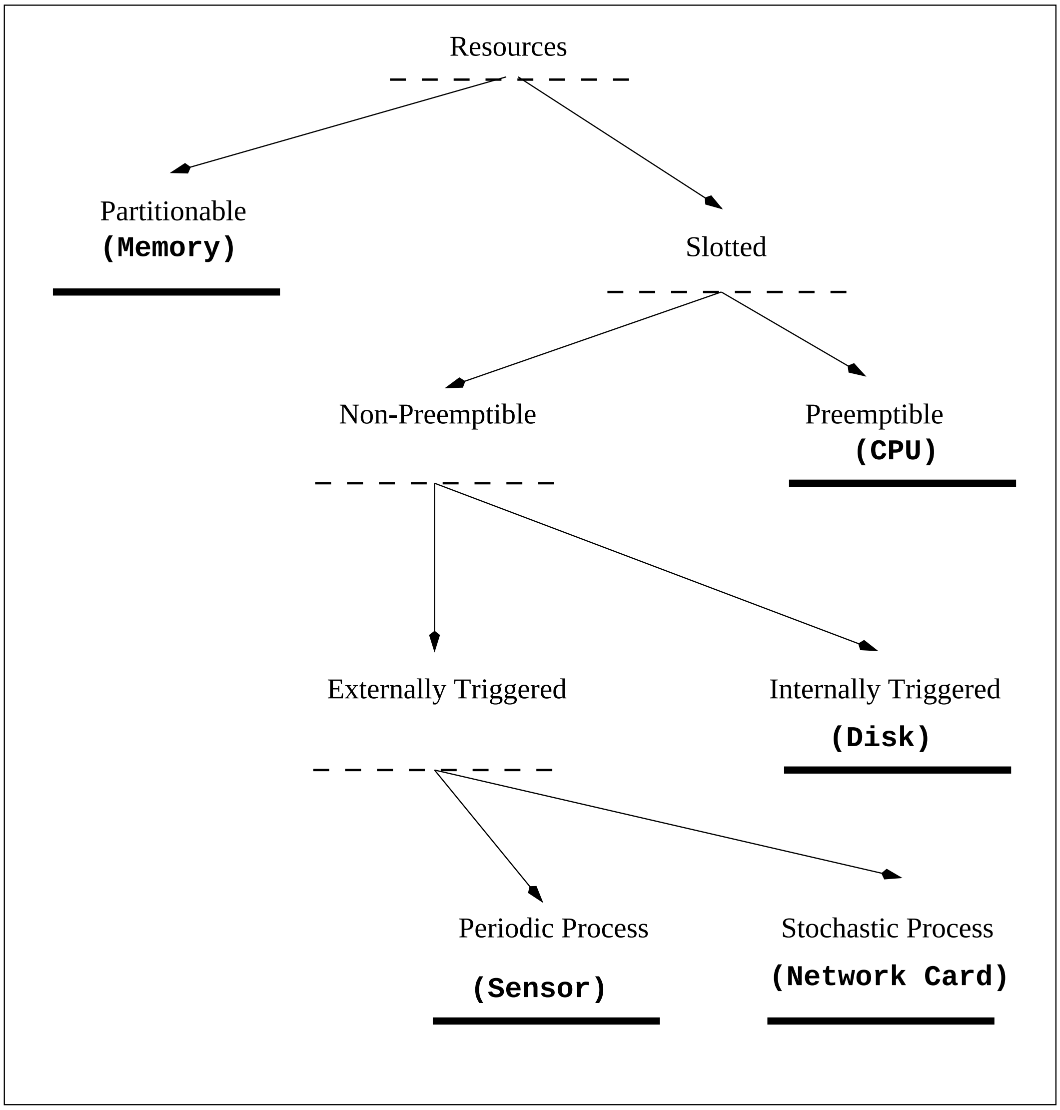

Figure 3.4: Resource Model Classification

To motivate the need for the complexity of the resource model we begin with an example that illustrates the subtle reasoning that encourages further subdividing the slotted class. We argue that a simple, obvious approach to sharing slotted resources other than the processor actually results in an unacceptable level of variability in the minor cycle lengths.

Consider Figure 3.5. The approach to multiplexing access to the physical devices illustrated there is that the implementation would modify the GPOS device drivers to provide an operation to the RTK that would allow it to insert the work request of a real-time thread at the head of the device queue. The resulting delay until the work request was done could be calculated as the amount of time to queue the request plus the maximum time required for any work request for this device to complete. The design is appealing and simple, but problematic.

First, three requirements of this design are not immediately obvious.

1. The operation that removes a work item from the queue and gives it to the device *must* operate on an interrupt thread. If the dequeue operation occurs on a user or kernel thread, it will be subject to the variations imposed by the GPOS scheduler. That is, its completion time would be a function of the other threads currently executing on the system. So the dequeue must occur on the interrupt thread created by the device when it interrupts for the completion of the previous request.

2. The operation used by the RTK to place a work item at the head of the queue must be protected by a real interrupt disable and not a virtualized interrupt disable. This is simply a mutual exclusion issue: since the RTK and GPOS are both accessing the queue, it must be protected.

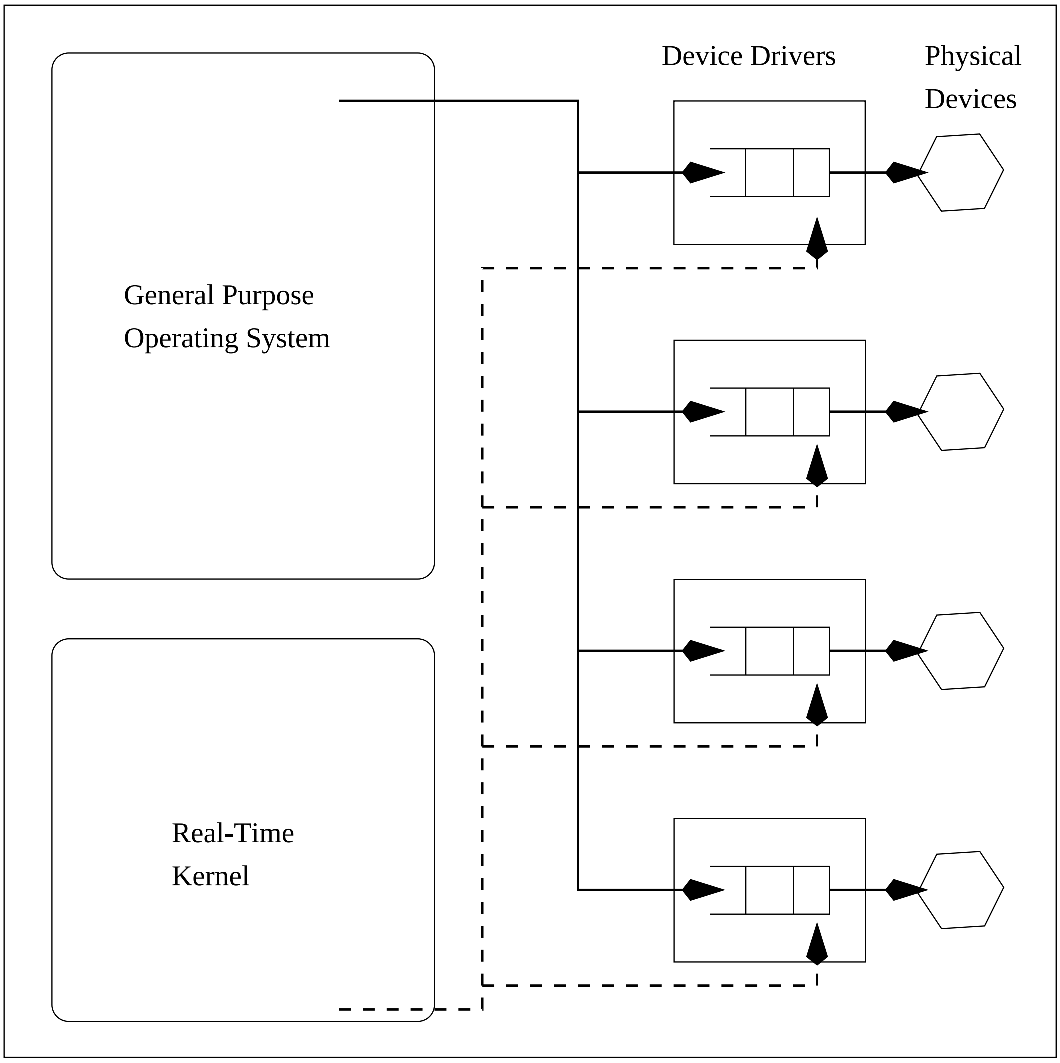

Figure 3.5: An Obvious Approach to Multiplexing

3. The interrupt multiplexor circuit on the computer system must provide dynamically alterable priorities among the input lines from devices. Unless the priorities of the interrupt lines from the devices into the multiplexing circuit can be altered to a higher priority, busy devices may keep a lower priority device from starting on the real-time thread work request assumed to be at the head of its queue.

Now, consider that these three requirements are met and imagine a computer system with a provision for attaching sixteen devices through two cascaded interrupt controllers, such as is the case on the popular PC motherboard architecture. If the interrupt controllers and devices are in a particular state, a device attached to the slave interrupt controller may have to wait through 64 interrupt handlers before it can dequeue the work request at the head of its queue. This is because device priorities may rotate after completion of an interrupt. So the slave must wait for all devices on the master to be serviced before it is serviced if:

- the device to which a real-time thread queued a request just completed an interrupt (making its priority now the lowest), and
- this device is on the slave controller, and
- the slave controller was the last device to interrupt the master controller (making the slave controller the lowest priority device on the master controller), and
- all other devices raise their interrupts simultaneously (or nearly so).

Now the slave controller can get one of its devices serviced and the device trying to dequeue the real-time request moves up one priority level. If all of the devices on the master controller now raise their interrupts again, the slave controller will have to wait again for all of the devices of the master controller to be serviced.

This cycle can continue six more times before the device attempting to dequeue the real-time request can be serviced. All in all, the device would have waited for 64 interrupt handlers to execute before its own interrupt handler could run. Since a typical execution time for device interrupt handlers is about 100 microseconds, the real-time request may have to wait 6.4 milliseconds in the device queue before being dequeued. Therefore, a real-time thread using the example device would have to have its cost inflated by 6.4 milliseconds in order for the real-time thread scheduler to perform a correct feasibility analysis.

We assume typical real-time thread execution times to be in the hundreds of microseconds, so the few milliseconds of inflation would raise the cost by an order of magnitude above the real cost. This amount of inflation is considered unacceptable. The three previous conditions are required to ensure a bounded waiting time for access to the device (or else the waiting time would be unbounded). But further analysis shows that, although bounded, the waiting time is unacceptably large.

Figure 3.6 illustrates the approach of the SP architecture to multiplexing slotted resources other than the processor. The three requirements for the obvious approach described above are not required for the SP approach. The SP architecture allows the executive for the resource/device to ensure that access to the resource incurs no waiting time. It does this by scheduling all accesses to the resource such that any serializable portion of the resource is idle when a real-time thread requires access.

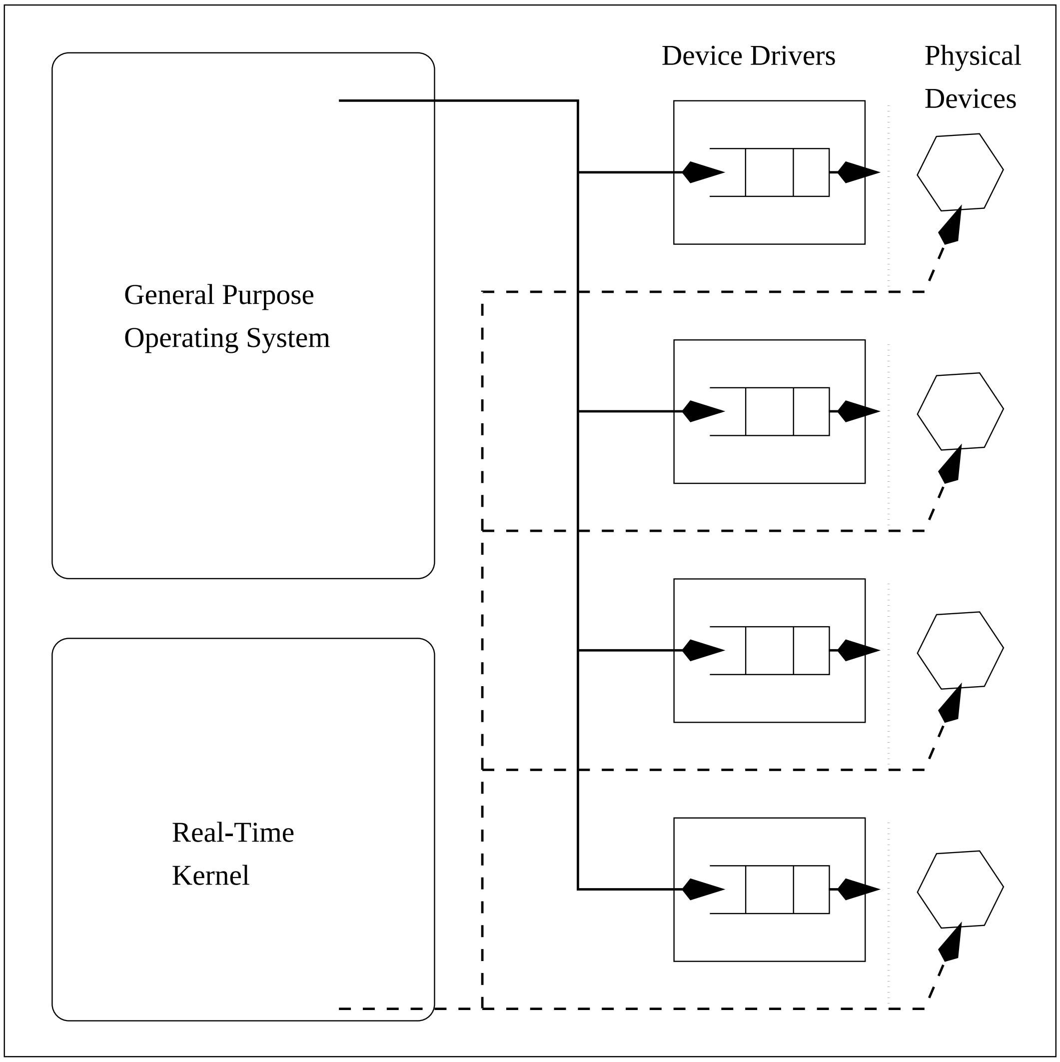

Figure 3.6: The SP Architecture Approach to Multiplexing

The resource model defines the parameters that separate devices into distinct classes. These parameters have so been determined that the resources in a particular class can be managed by similar executives. The SP architecture requires an executive for each instance of a resource existing on a computing machine which is used by a real-time thread (that is, the real-time thread intends to use the resource sometime in its execution). The resources are classified in a way that guides implementers in constructing executives that should require only minimal changes to the low-level physical device drivers already on the system. In some cases those device drivers need no modification at all.

Also, though not explicitly stated in Section 3.6, it is worth noting here that the real-time minor cycles can begin during the execution of the GPOS device drivers. This is necessitated by requirement $\mathcal{A}$: if the intervals in which real-time threads execute are scheduled to begin every l time units then the CPU executive must ensure that the intervals begin at times t, where $kl \leq t \leq kl+\epsilon$, $\forall k \geq 0$. The device drivers should continue to function correctly, since when they resume, no detectable change in system state has occurred, only time has passed. Section 3.8.1 describes partitionable resources in detail and Sections 3.8.2 through 3.8.2.6 describe the subclasses of the slotted resource class.

3.8.1 Partitionable Resources

The primary characteristics of resources in this class are that each comprise many identical units that are simultaneously usable. Examples of instances of this resource type are, memory, video display pixels, IP ports, and disk blocks. In each example the resource is composed of identical units that may be used by either the GPOS or RTK without modification.

The executives for this class of resources are the easiest to construct. The issue here is that real-time threads must have exclusive access to the subset of units assigned to the RTK. This is because the real-time and non-real-time threads are not synchronized. For example, the pixels of the display device can be partitioned and a subset assigned to the RTK. The windowing subsystem of the GPOS must be informed, by some means, that the set of pixels assigned to the RTK are not available to non-real-time threads in the GPOS. The partition is now established. However, repartitioning may occur over time as the window is moved around the display, changing the set of pixels that should be assigned to the RTK. The executives created for this class are, in some sense, static. Repartitioning may occur over time, as in the case of a window of pixels assigned to the RTK, but once the partitioning is established, and until another repartitioning occurs, the RTK and GPOS use only the pixels in their respective static subsets.

Note that in some cases even though the resource is partitioned in an implementation, and the RTK has exclusive access, the RTK must access the units of the resource through yet another resource that may fall in a different class. For example, disk blocks need to be partitioned between the RTK and GPOS; otherwise the file system of the GPOS would need extensive modification because the sharing would occur within the GPOS file system. This is because the file system assumes it has exclusive access to all blocks on the disk (ignoring the fact that some disk blocks may be unused and that blocks may be partitioned between different file systems; we are considering only the set of blocks known by the file system).

Therefore in this case, use of some of the disk blocks by real-time threads would have to be coordinated, within the file system, with use by non-real-time threads. This implies that the disk blocks would be partitioned between the RTK and the GPOS by the file system and not a resource executive. However, the further implication is that the whole file system would have to be modified to accommodate the sharing. To avoid this, the SP architecture specifies that the disk blocks be partitioned by a disk block executive which allows exclusive access to a disjoint set of disk blocks by the real-time threads[3]. However, this simple partitioning is not sufficient to allow the RTK access to the disk blocks which occurs through the device driver associated with the disk. The disk device driver would, typically, fall into another class within the resource classification and require its own executive.

3.8.2 Slotted Resources

Some resources must be multiplexed dynamically in time, because the communication method between the driver and the resource/device is non-reentrant and requires mutual exclusion over certain intervals. This typically results from memory mapped I/O ports or command blocks. If the port or command block is being used by the GPOS, it is unavailable for use by the RTK. As an example of a driver-to-device non-reentrant communication method, consider an imaginary network device card which is accessed via four memory mapped blocks of memory

[3]In practice this can easily be accomplished by having a non-real-time program create an unused file within the GPOS file system of a size required by the RTK and then passing the list of disk blocks to the RTK to use as required.

into which the device and its device driver place commands and responses[4]. Furthermore, assume a finite state machine that defines sequences of commands and responses between the device and its device driver.

Figure 3.7 shows a sequence of device driver interactions that might occur when the device driver intends to send a packet into the network. The parenthesized phrases are actions that are taken by either the device or the device driver, and the phrases with arrows are commands or responses for the driver to the device (right pointing arrow) or from the device to the driver (left pointing arrow).

Note the points in the figure identified by '(X)' and '(Y)'. Between these points the command block is in use. If the commands were issued by the GPOS, the command block would not be available to an SP executive. In fact, without modifications to the device driver code, an SP executive should avoid using the device at any point in this sequence between '(X)' and '(Z)'[5], because the device driver and the device make assumptions about each other's state based on past interactions. Another device driver attempting to interact with the device could invalidate these assumptions, resulting in error in either the device or the device drivers.

There are six subclasses of the slotted class, of which two are further subdivided. Since actual instances exist for only leaves of the resource hierarchy tree, only four of the subclasses contain resources. The slotted class hierarchy will be briefly described here and each subclass will be described in detail in its own section

[4]This example adapter closely resembles the IBM 16/4 Token Ring network adapter.

[5]A point worth noting here is that executives for network adapter cards are the most difficult type executive to construct.

```
Device Driver                                   Device
-------------------------------------------------------
(X)(Writes send Command to Command Block)
A Command is in the Command Block -->
                                            (Reads command)
                                            (Writes response to response block)
                                            <-- The Command has Been Read
(Reads response)(Y)
                                            (Writes response to response block)
                                            <-- Ready to Send
(Reads response)
(Writes location of Data to Command Block)
A Command is in the Command Block -->       (Reads command)
                                            (Writes response to response block)
                                            <-- The Command has Been Read
(Reads response)
                                            (Writes response to response block)
                                            <-- The Data have Been Read
(Reads response)
                                            (Writes response to response block)
                                            <-- The Data have Been Sent
(Reads response)(Z)
```

Figure 3.7: Sequence of Commands and Responses for a Network Send Operation

below. The slotted class hierarchy is shown in Figure 3.4.

The first division in the slotted class hierarchy separates those resources that can be preempted from those that cannot. The CPU is an example of a preemptible resource, the typical direct access storage device is an example of a non-preemptible resource. The preemptible class is a leaf.

The non-preemptible class is subdivided into resources that are internally triggered and resources that are externally triggered. In internally triggered non-preemptible resources, all interactions begin in the GPOS and its associated processes. In externally triggered non-preemptible resources, some interactions are initiated by a process outside the computing machine (such as a packet arriving from the network). The externally triggered class is subdivided into those resources for which the triggering process is periodic (a leaf) and those resources for which the triggering process is stochastic (another leaf).

3.8.2.1 Preemptible Resources

The only example of a resource in this class of which we are aware is the CPU. Preemptibility has been a requirement of CPUs since the very earliest designs. This feature typically facilitates the use of attached devices that operate at much slower (relative) speeds than the CPU. The SP resource model considers this class as a leaf, because for a resource with this capability, no other issues affect construction of its executive. In fact, the executive, although complex, is the most straightforward of all executives for slotted class resources. This is because the hardware system has been designed to ease the transfer of control from one sequence of instructions to another asynchronously (with respect to the execution of programs) by activating an electrical input into the device. Thus, a system can gracefully suspend and resume program execution at well defined points.

Executives for the preemptible class, such as a CPU, are considerably more difficult to construct than executives for the partitionable class. As required of all executives, the executive constructed for this class must gain access to the resource at precise intervals. In this case this amounts to a process switch from some process in the GPOS to the CPU executive. One way to provide this context switch is via an interrupt to the CPU from a hardware timer. The code that executes in response to this interrupt can be considered to be the executive. The executive decides to execute either a process in the GPOS or the RTK according to the execution model described in Section 3.6. To ensure that this interrupt occurs on time GPOS access to the interrupt enabling/disabling mechanism must be virtualized.

3.8.2.2 Non-Preemptible Resources

Two primary characteristics drive the structure of an executive for a non-preemptible resource:

- the resource cannot be conveniently preempted
- sequences of GPOS driver/resource interactions exist during which the RTK cannot access the resource

Such sequences must be atomic with respect to the GPOS driver and the resource. The key problem designing an executive for non-preemptible resources is to identify the points between sequences at which the RTK can access the resource and to manage the resource so that such a between-sequence point exists when the resource is needed by a real-time thread. The details of how an executive can accomplish this function are described in Sections 3.8.2.3, 3.8.2.5, and 3.8.2.6.

The programming model, described in Section 3.9, specifies two modes of interaction with these resources: real-time and non-real-time. The real-time mode allows a real-time thread to ensure that its operation with the resource completes during its execution.

Note that the real-time mode continues to guarantee that the real-time thread will complete before its deadline. The non-real-time mode only guarantees that the call to queue the request to the resource completes while the real-time thread is executing. Of course, the real-time thread will complete before its deadline, but the operation for the resource may complete at any time after the request is queued. The non-real-time mode simply queues the work request for the GPOS via a non-blocking, real-time queue, which is required to ensure that the queue operation completes in bounded time.

The real-time mode works as follows: the real-time thread includes all instructions executed from the time the request is made in the real-time thread until the physical device has issued a start command to commence processing the real-time request. The time necessary for the device to complete the request is also considered part of the real-time thread, but does *not* require CPU cycles. Thus, in the real-time minor cycles that occur during this virtual portion of the real-time thread, the RTK is free either to schedule other real-time threads that do not require this resource or to return to the GPOS.

When the device completes the request, the original real-time thread regains control at the start of the next real-time minor cycle, just after the call site. The real-time thread scheduler accommodates access to the device by considering the call to be a blocking call with a certain cost. This cost includes several parts:

- the cost of executing instructions from the call site to the point at which the RTK device driver completes the request to the device
- the time the resource uses to processing the request
- and the cost of executing instructions within the RTK device driver up to the return to the statement after the call in the real-time thread

The RTK scheduler may also avoid complexities by scheduling the complete real-time thread, both actual and virtual portions, as though they both really used CPU cycles. In this case real-time minor cycles may occur in which the scheduled real-time thread is blocked waiting for the resource to finish processing the request. The low-level dispatcher can then simply return to the GPOS.

Another issue to consider is how the resource executive avoids an unbounded wait in the RTK device driver. Typical GPOS device drivers accept requests from threads for work the device is to perform. If these requests arrive faster than the device can complete them, the device driver queues the requests. Thus, in a GPOS, the completion time of a request can be determined only stochastically.

There are two possible models for avoiding this unbounded wait and replacing it with either a zero wait or a bounded wait. In the bounded wait model, the RTK device driver interacts with the GPOS device driver and places the request from the real-time thread at the head of the queue. Thus, the longest wait will be the time the device needs to complete a single, largest possible request, plus the longest time that an interrupt from the device may have to wait for service. This wait is included in the cost of the real-time thread, because the real-time thread scheduler must know the cost of all of the real-time threads so that it can perform the feasibility test and set the sizes of the minor cycles.

This model requires significant modification of the existing device drivers and may significantly increase the variability in how long a thread requires to complete.

The other model is for the RTK device driver to interact (minimally) with the GPOS device driver before the request from the real-time thread, and to have the GPOS device driver stop issuing requests to the device. Thus, when the request arrives from the real-time thread, the physical device and its GPOS device driver are at a point in their interaction sequences at which an RTK device driver can access the device, so the device can start the real-time work request immediately.

The two models differ in the length of the virtual portion of the real-time thread. The first model requires that the virtual portion be the length of time the device needs to complete a single, largest possible request, plus the time an interrupt may have to wait for service plus the length of time the device needs to complete the request from the real-time thread. The second model requires the virtual portion to be the length of time the device needs to complete the request from the real-time thread. It allows the RTK scheduler to be more efficient, at a cost of restricting GPOS requests to the resource at appropriate times. However, neither model changes the way the RTK scheduler accomplishes its work; the models are transparent to the RTK scheduler.

That the resources in this class cannot be preempted complicates the design of an executive that can correctly multiplex access to the resource between the GPOS and the RTK. A trade-off in the design of executives for these resources is where to merge the work requests from both the GPOS and the RTK into a single stream. The closer to the resource this occurs, the lower the variability in real-time thread cost that can be achieved by the system but the efficiency of the GPOS decreases. The decreased efficiency in the GPOS with respect to this resource results from the lower amount of coordination between the GPOS and the RTK about the use of this resource. The resource executive may have to disallow access to the resource by the GPOS for longer periods of time in advance of use of the resource by a real-time thread. Merging requests closer to their source increases programming effort and real-time thread cost variability but better GPOS efficiency.

Because a preemptible resource can be stopped precisely and with fine granularity (in time) the two request streams do not merge. Rather, the real-time requests preempt the resource's operation on non-real-time requests. To accomplish this with a non-preemptible resource might be possible, but only with a very thorough understanding of the interaction sequences between the resource and its software driver and the ability to determine, at any point, the states of the resource and the software driver. In addition, the RTK, on behalf of one of its real-time threads, would need to retain control of the resource until the interaction sequence between the real-time thread and the resource was complete[6]. The advantage of this design is that it requires no modifications to the software driver. Its disadvantage is that the RTK must rapidly poll the resource to determine when it can safely take control.

Another possible design requires a minimal modification of the software driver that allows the executive to restrict the flow of work requests from the GPOS to the resource. The executive would then stop the software driver early enough from starting a new interaction sequence that when the real-time thread required the resource, it would be available. How, then, does the executive know when the resource will be needed? We postulate a real-time thread scheduler with a lookahead feature. The scheduler allocates all of the CPU cycles in the real-time minor cycles to appropriate real-time threads for the next given interval of time. At each scheduling decision, the scheduler looks ahead and compares its list of resources reserved for real-time threads to the actual real-time threads that have been allocated CPU cycles in the lookahead interval. Note that each resource has a specified time interval after which it is available to start a sequence of interactions

[6]This may require extending a real-time minor cycle.

with a new driver, assuming that requests from the GPOS are stopped at or before the interval begins. If the time to stop requests for a resource is approaching (because that resource has been reserved by a real-time thread scheduled in the lookahead interval), the scheduler informs the resource executive to stop requests from the GPOS. This is the recommended design of executives for non-preemptible resources. Its advantage is that it requires no polling of the resource and only minimal modifications to the GPOS software driver.

A final executive design is to modify the GPOS software driver to provide an RTK entry point that places the real-time thread work request at the head of its queue. As mentioned previously, this method can significantly increase the variability in the cost of a real-time thread and may require changes to other GPOS software drivers and to hardware programming. However, it may be the only option for externally triggered resources.

3.8.2.3 Internally Triggered Non-Preemptible Resources

For the resources in this class, all requests originate from within the GPOS, unlike externally triggered resources where some requests arise from processes outside the computing machine. The unique characteristic of internally triggered resources is that the SP executive and GPOS device driver can completely control the scheduling of the communication medium used by the resource and the GPOS device driver. For an externally triggered resource this is not possible, since the communication medium may become busy due to an event that neither the SP executive nor the GPOS device driver can control. As a result, the SP executive and GPOS device driver cannot schedule of the use of the communication medium between the GPOS device driver and the resource.

A direct access storage device is one type of internally triggered resource. Although it will interrupt the processor when it has completed a request, the request originated from the GPOS on behalf of one of its threads. Therefore, the SP executive, along with the GPOS device driver, could have prevented this interrupt by preventing the originating request from proceeding from GPOS to resource. This design works well for any internally triggered resource, because no matter how obscure the interaction sequence is or whether one can determine the states of the software driver and resource, by stopping requests from the GPOS, the executive knows that the resource will eventually complete all of its pending requests and be ready to start an interaction from the RTK. The more the executive knows about the interaction sequence and the easier it can correctly determine the states of the software driver and the resource, the better it can interact with both. Better interaction here means that the executive can stop the flow of requests from the GPOS closer to the point at which the resource will be required by a real-time thread. This is a better method than adding the real-time request to the head of the resource queue, because it does not expand the variability in the cost of the real-time threads by increasing the contention for the resource queue between the GPOS and the RTK. Moreover, it requires no modification to other software drivers and hardware.

Different techniques can be used to cut off the flow of requests to the physical device. For example, the physical device itself may allow the executive to set the device's state to busy for diagnostic purposes. Once this flag is set the GPOS device driver will stop issuing requests to the resource and pending requests will complete. Another technique is to modify the GPOS device driver slightly to check a flag before issuing a request to the device. The executive then sets this flag at the appropriate time based on an indication from the real-time thread scheduler analyzing its lookahead interval. Although this technique requires access to the source code and build environment of the GPOS device driver, it is valid for all of the GPOS device drivers considered here. This technique should also be valid for all physical device drivers in the internally-triggered class. A GPOS device driver is the only component in a computing system that delivers work to an internally triggered device. With access to the source code and build environment of the device driver, coding a flag that controls whether or not the driver issues a command to the device is straightforward.

3.8.2.4 Externally Triggered Non-Preemptible Resources

The externally triggered resources, such as network adapters and keyboards, are problematic. Use of the communication medium between the GPOS device driver and the resource can be generated by external processes, such as arrivals of network packets and keystrokes. At times, even though the executive tries its best to ensure that the communication medium between the GPOS device driver and the resource is free when a real-time request arrives, the medium may instead be in the middle of an interaction sequence originated by a request from an external process. For example, consider a network device whose medium includes a memory block shared between the device and the GPOS device driver. When a packet arrives, the network device places a command into the shared memory block and interrupts the CPU. Placing the command into the shared block initiates a sequence of interactions between the device and the GPOS device driver. Until this sequence ends the RTK device driver is denied access to the communication medium of the device.

The characteristic variation of the interarrival times of events from these external processes forms the basis for the decomposition of this class. If the external process generates events whose interarrival times theoretically do not vary, then the resource is said to have a periodic process. An executive can easily manage such a resource, because it can predict the use of the communication medium between the resource and the GPOS device driver. Thus the SP executive can ensure access of the RTK device driver to the resource with a bounded delay. If, however, the external process generates events whose interarrival times vary stochastically, the resource is said to have a stochastic process. In such a case the executive has great difficulty managing the resource because the communication medium between the GPOS device driver and the resource is unpredictable, so the SP executive cannot guarantee access to the resource with a bounded delay. The delay is unbounded because when a real-time minor cycle begins the communication medium may be busy, disallowing access by the RTK. Therefore, the only option is for the RTK to return control to the GPOS, so that it can complete its sequence of interactions with the resource. But the same situation may occur as the next real-time minor cycle begins, now due to an interaction sequence started by another externally generated event.

These coincident events may continue, thus denying the RTK access to the resource. Note that no modification to the GPOS device driver will alleviate this problem and allow the real-time thread to use the communication medium to issue the real-time request directly to the resource. One solution is to have the RTK device driver simply queue the real-time request in the GPOS device driver's queue. However, this method, as explained earlier, can cause significant variation in the cost of a real-time thread.

It is worth restating here that the minimal modification constraint means that the design of the executive must embody an essentially complete device driver for the physical device. Such an RTK device driver will interact directly with the physical device's interface and must adhere to its rules. Physical devices rarely accommodate multiple device drivers, so their interfaces provide little support for the interleaving of interaction sequences from multiple device drivers. Thus, for now the best solution for real-time access to, and completion of requests to, externally triggered, non-preemptible, stochastically driven resources is to enter the real-time request into the GPOS device driver queue and absorb the attendant increase in cost variability.

3.8.2.5 A Periodic Process Driving an Externally Triggered Non-Preemptible Resource

If the all of the processes generating events to an externally triggered resource do so at precise intervals, then the resource falls in the periodic process class. Even when many processes all generate periodic events, the pattern of arrivals is still deterministic, although it is no longer periodic. These resources still fall into this class. An example of such a device is a sensor that collects temperature data and generates a report every 30 seconds ($\pm$ some known variation).

The executive for a resource in the periodic process class must have a schedule of when requests will arrive from the external processes at the resource and how much execution time each request will need to complete. Using this schedule, the executive can calculate the longest busy interval of the resource R_b. When the first real-time thread reserves the resource, the real-time thread scheduler adds R_b to the reservation cost, because a real-time request from the real-time thread may occur at the beginning of the resource's longest busy interval.

3.8.2.6 A Stochastic Process Driving an Externally Triggered Non-Preemptible Resource

This last class is the most difficult for an SP implementation to manage. Unfortunately, many useful resources, such as network adapters and keyboards, fall into this class. Their events, for example the arrival of a packet from the physical network link or a user pressing a key on the keyboard, initiate a transaction from the physical device to the GPOS device driver. In no case could any action within the computing machine regulate when the transaction begins because the machine cannot control events outside itself. Although the ability of the physical device to actually generate a signal to the GPOS device driver could be disabled, the physical device would still be busy and its interface would probably block the start of a second transaction until the pending transaction was handled. Handling the pending transaction might include the ability to reset the adapter, losing the pending data. However, since the executive lacks information about the state that exists between the GPOS device driver and the physical device, arbitrarily resetting the physical device is problematic.

There is no way to ensure that the RTK device driver can communicate with a resource in this class when a real-time thread requires it. In fact, for some high speed network adapters, events can arrive faster than the GPOS device driver can move the data to buffers, resulting in lost data. The higher level protocols acknowledge this possibility of loss and assume that the sender will retransmit; however, the effect on a real-time thread in the SP system would be dramatic if the lost packet were intended for that thread. However, the SP architecture does not address device utilization overruns. For resources in this class, the resource itself and its first-level interrupt handler should be considered as part of the stochastic process that is beyond control of the executive and RTK device driver. Device utilization overrun also affects the GPOS, but, of course, no guarantees are broken, since the GPOS does not guarantee completion time.

Two categories of recommendations can be fruitfully discussed at this point:

recommendations for constructing devices and recommendations for constructing the interfaces between devices and their controlling software.

The best possible functional addition to a device previously falling into the stochastic process class is one that completely preempts and restores an arbitrary sequence of interactions between the GPOS device driver and the resource. Thus, the executive can the interact with the device just as it does with the CPU. At any point the executive can preempt the active sequence and start a real-time sequence for a real-time thread. Since the completion time of the transaction would have an upper bound, feasibility could be determined correctly. Although this asks much of device designers, there is a move to employ general purpose processors as the controlling entity in adapter devices and with modest design effort this recommendation may be realizable.

Another alternative would be to allow the abnormal termination of any active sequence of interactions between the GPOS device driver and the resource. This would accomplish the same result with some performance degradation, since the processes involved in the terminated sequence would eventually restart it, thereby wasting the processing already completed but relying on higher level software to retry. Note that this method would not work for keyboard devices because asking the user to retry is not acceptable. This design would also require that the device and controlling software be able to prevent a situation where so many interaction sequences are terminated that no useful work is accomplished. This situation is analogous to operating system virtual memory thrashing and CSMA physical network media collisions.

A third design is to modify the interface between the resource and the GPOS device driver so that the executive can ensure that the resource can begin a given sequence at the next available time and operate on only one sequence at a time. In this case, even though the executive might not be able to start a real-time thread's sequence at a particular point, the amount of time the transaction would have to wait would be limited. This design would, again, allow the correct calculation of feasibility.

The remaining option is to install a separate, dedicated device for exclusive use by the RTK and the real-time threads.

3.9 Programming Model

The programming model defines how programmers write and think about real-time programs for an implementation of the SP architecture. Real-time threads are threads of traditional GPOS processes. The SP architecture defines how periodic real-time threads are created via GPOS system calls. These real-time threads then execute a continuous loop with one period-expiration wait statement per loop. The SP architecture also defines and requires that resources be reserved for real-time threads in the real-time mode. Each real-time thread reserves the real-time resources it intends to use within its continuous loop. How it uses each resource is somewhat independent of the resource's class in the resource model. The underlying executive and execution model ensure that the resource can communicate freely with its device driver, within some reasonable upper bound, when required by a real-time thread. The CPU is the exception because it is reserved not by the programmer but by the call to create a real-time thread. One other slight difference among the use of resources is that in the reservation call the programmer must specify how much resource is required in units particular to the resource. The RTK uses this value to calculate how many CPU cycles the real-time thread needs (the thread's cost) and how long the device will require to complete the request.

3.9.1 Video Display Example

Consider a requirement for a real-time thread to receive video data from an external communications network, process each incoming data buffer, and display those data on the video display device (see Figure 3.8). The code fragment in Figure 3.9 demonstrates how this function might be implemented in an SP implementation and represents the box labeled 'ProcessData' in Figure 3.8.

This example illustrates the use of a resource (the network device) in the stochastic externally-triggered non-preemptible resource class. One operational mode, as described in the resource model, is for the executives to manage resource sharing by polling. Polling adds no more variability to the real-time performance of threads using resources in this class above that already induced by the stochastic process generating the events.

In Figure 3.8 the ellipse labeled 'Network Device Executive' watches the incoming packets on the 'Network Device.' If one is destined for a real-time thread,

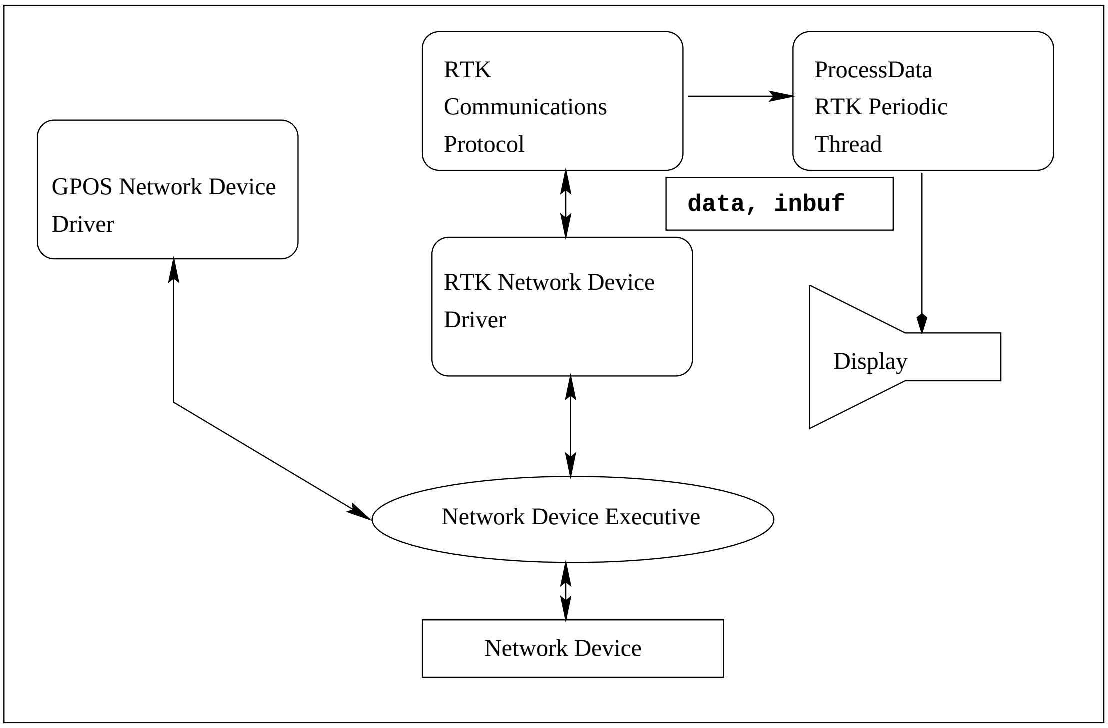

Figure 3.8: Video Display Example

```
main ( )  {
   double p = 33.0; /* period in ms */
   double c;  /* assume (c) cost = 4.0 ms */
   RTThread * pProcessData;
   static shared char inbuf[INBUFSIZE];
   char outbuf[OUTBUFSIZE];
   /* share inbuf with the communications protocol */
   ShareMemoryWithCommStack ( READ, inbuf );
   /* compute execution time of ProcessData thread*/
   c = ComputeRTCost(ProcessData);
   /* create the real-time thread */
   pProcessData = CreateRTThread (c,p,ProcessData);
   /* if the real-time thread was created, start it */
   if ( pProcessData != NULL )
      startRTThread ( pProcessData );
   /* wait until the real-time thread quits */
   WaitOnRTThread ( pProcessData );
}
RTThread ProcessData ( ) {
while ( true ) {
   if ( data ) {
      /* read and decompress data from the network */
      for(i=0;i<INBUFSIZE;i++,j+=compressionFactor)
         Uncompress ( inbuf[i], outbuf[j] );
      /* display the video data */
      for(i=0;i<OUTBUFSIZE;i++)
         display[i] = outbuf[i];
      data = false;
      }
   } /* wait until the next period */
   WaitForPeriodExpiration ( );
}
```

Figure 3.9: Code for Video Display Example

it calls the 'RTK Network Device Driver,' which communicates with the 'Network Device' to read the packet. The packet is then passed to the 'RTK Communications Protocol' to strip headers, etc; and thence is written into 'inbuf.' The buffer 'inbuf' is set up as shared memory for the 'RTK Communication Protocol' and 'ProcessData.' The 'Network Device Executive,' 'RTK Network Device Driver,' and 'RTK Communication Protocol,' execute in response to an interrupt from the 'Network Device,' The 'ProcessData' threads execute once every 33 ms and are scheduled by the real-time thread scheduler. The shared variable 'data' serializes access to 'inbuf.' During each period the 'ProcessData' thread checks for new data ('data' equal to true) and uncompresses it, displays it, and sets 'data' to false. If a packet arrives and 'RTK Communication Protocol' finds 'data' equal to true, the packet is discarded. If 'data' equals false, 'RTK Communication Protocol' writes the data into 'inbuf.'

This example also shows the disparity between the stochastic driven process and the deterministic process. Because the stochastic events occur unpredictably, packets will be discarded and the quality of the display will degrade. Although buffering can be used to some degree to smooth out this discrepancy, it increases latency, which may be undesirable at times.

It is now useful to examine how 'ProcessData' would be scheduled in an implementation of the SP architecture. To make things more interesting, consider four instances of 'ProcessData,' each of which is displaying a separate video stream. To start, consider how these four instances might be scheduled in a traditional real-time system such as a cyclic executive.

A cyclic executive is a static execution schedule for a fixed set of threads, so organized that all threads meet their deadlines. The schedule is made up of major and minor cycles. A minor cycle is a constant time period in which a fixed sequence of threads, or subthreads (logical pieces of threads), execute. A major cycle is a repeating sequence of minor cycles.

Assume the four instances of 'ProcessData' are labeled A, B, C, and D, and that two background processing threads, labeled E1 and E2, do required background processing. Assume also that the cost of each instance of 'ProcessData' is 4 ms and that E1 and E2 have costs 4 ms and 5 ms. The period of each instance is 33 ms. A minor cycle in a cyclic executive might look like (A—E1—B—E1—C—E1—D—E2). The cost of this minor cycle is 33 ms; and if it is continuously repeated, each instance of 'ProcessData' would receive its 4 ms of processor time in each period.

Now consider how the four instances might be scheduled in an implementation of the SP architecture. The instances would be created serially and the RTK would set up the real-time minor cycle and non-real-time minor cycle lengths, mc_{rt} and mc_{nrt}, after each creation as shown in Table 3.1.

The extra 0.5 in the minor cycle length accommodates (possible) known variation in the start of the real-time minor cycles and the execution of the RTK at the start of each real-time minor cycle. After the second instance is created, each instance will execute in alternate real-time minor cycles. After the third instance is created, each instance will execute in every third real-time minor cycle. After

Active Instances	mc_{rt}	mc_{nrt}	MC
A	4.5	28.50	33.00
A and B	4.5	12.00	16.50
A, B and C	4.5	6.50	11.00
A, B, C, and D	4.5	3.75	8.25

Table 3.1: Minor and Major Cycle Lengths for Four Instances of 'Process Data'

the fourth instance is created, each instance will execute in every fourth real-time minor cycle. At all times each instance executes once in every 33 ms as required.

Another reasonable approach to setting the minor cycle values as each instance was created would be to increase the real-time minor cycle length by 4 ms and decrease the non-real-time minor cycle length by 4 ms. After the four instances were created the values would be $mc_{rt} = 16.5$ and $mc_{nrt} = 16.5$. These values show better utilization of the CPU, because the RTK needs to execute only at the start of a real-time minor cycle (except for some small scheduling actions as each instance calls the `Wait` function). However, the real-time minor cycle is rather long and the GPOS would lose efficiency as various devices and their device drivers waited for the end of the real-time minor cycle.

This example illustrated the use of a resource in the stochastic, externally-triggered, non-preemptible resource class in an appropriate manner for its class. The network device and some of its low level controlling software are considered as logical parts of the stochastic process generating events. The next example illustrates the use of a internally-triggered non-preemptible resource, a disk drive, in real-time mode.

3.9.2 Disk Write Example

Consider a requirement of a real-time thread to read data from the network, process them, and write them to disk while guaranteeing that the data bits are on the disk media before the end of each 100 ms period. Figure 3.10 illustrates the relationship between modules of this application. Figure 3.11 shows a code fragment, similar to the previous example, that might implement this function in an application running on an SP system. This rather contrived example still shows how a real-time thread would satisfy the requirement that the write to disk must happen before the deadline of the real-time thread.

Two differences from the previous example are that the disk is reserved and that data are written to the disk instead of to the display. The disk is an internally-triggered non-preemptible resource. The resource model encourages the executive to stop the requests from the GPOS before the real-time thread uses resource so that the communication medium between the RTK device driver and the resource is free when accessed by the RTK device driver.

In this case, assume that the communication medium needs up to 2 ms to complete any pending interaction sequence, so the lookahead interval of the real-time thread scheduler should be set to 2 ms. If this example is the only real-time

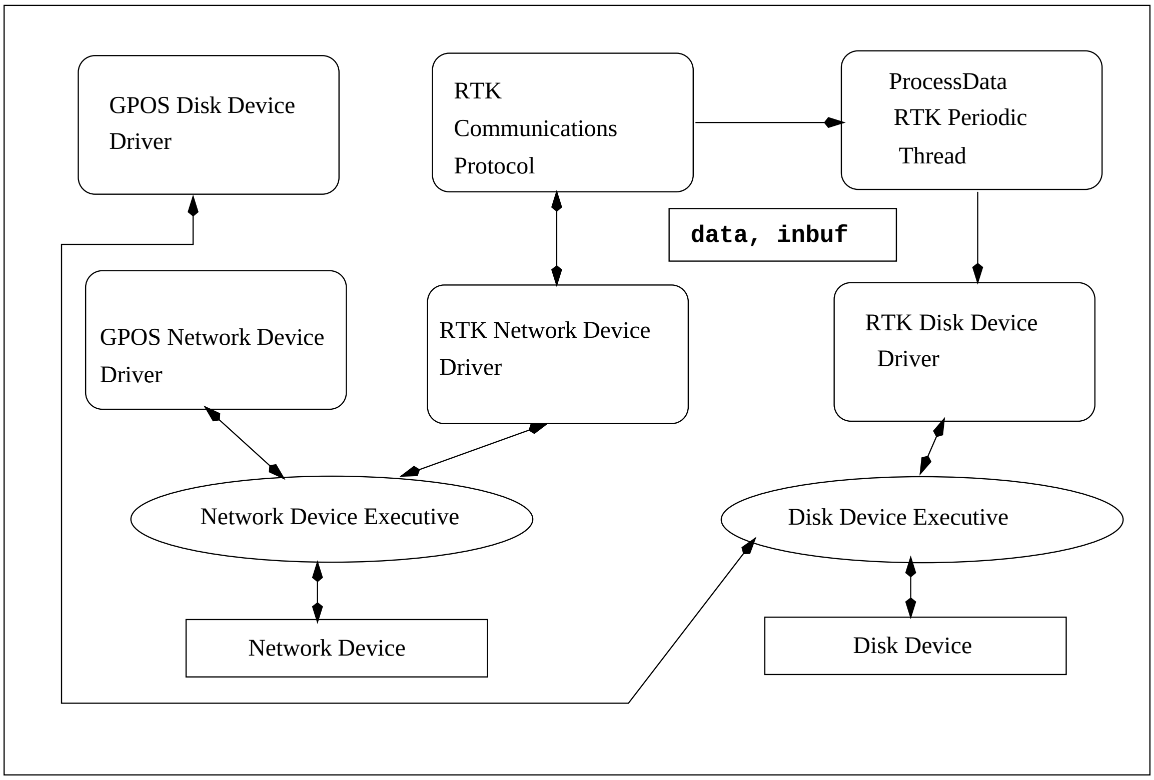

Figure 3.10: Disk Write Example

```
main ( )  {
   long rc = 0;
   double p = 100.0; /* period in ms */
   double c;  /* assume (c) cost = 28.0 ms */
   RTThread * pProcessData;
   static shared char inbuf[INBUFSIZE];
   char outbuf[OUTBUFSIZE];
   /* share inbuf with the communications protocol */
   ShareMemoryWithCommStack ( READ, inbuf );
   /* reserve the disk for a write of OUTBUFSIZE */
   rc = ReserveRTResource ( RTDisk,
     DiskWrite, OUTBUFSIZE, WriteRTDataDisk );
   if ( rc == 0 ) {
      /* compute execution time of ProcessData thread*/
      c = ComputeRTCost(ProcessData);
      /* create the real-time thread */
      pProcessData = CreateRTThread (c,p,ProcessData);
      /* if the real-time thread was created, start it */
      if ( pProcessData != NULL )
         startRTThread ( pProcessData );
      /* wait until the real-time thread quits */
      WaitOnRTThread ( pProcessData );
   else
     exit ( error );
}
RTThread ProcessData ( ) {
while ( true ) {
   if ( data ) {
      /* read and process data from the network */
      for(i=0;i<INBUFSIZE;i++,j++)
         Process ( inbuf[i], outbuf[j] );
      /* write the data */
       WriteRTDataDisk ( OUTBUFSIZE, outbuf );
      data = false;
      }
   } /* wait until the next period */
   WaitForPeriodExpiration ( );
}
```

Figure 3.11: Code for Disk Write Example

thread in the system, the real-time thread scheduler would send a signal to the disk executive 2 ms before each period begins, telling it to set the flag in the GPOS disk driver so that the GPOS device driver will not begin any new sequences of interactions with the disk device. The 'ProcessData' thread checks for data from the network device as in the 'Video Display Example,' processes any it finds before writing it to the disk with a call to `WriteRTDataDisk`.

Assume that the `WriteRTDataDisk` will take up to 23 ms to complete and that the other actions of `ProcessData` take 5 ms. The CPU executive would set the real-time minor cycle to $mc_{rt} = 28.5$ and the non-real-time minor cycle to $mc_{nrt} = 71.5$ with a major cycle of $MC = 100$. Although the real-time minor cycle is rather long, the CPU executive can return to the GPOS while `ProcessData` is blocked waiting for the disk to complete.

3.10 Intra-thread Communication

The SP architecture expects a particular structure for programs that use an implementation's real-time and non-real-time capabilities. The structure is driven by the observation that the amount of computation and I/O that must actually be accomplished in real time is small, well-constrained, well-defined, and localized. These properties are considered with respect to the overall program. An example would be a program to receive, control, and display live audio and video data from a network. The video should be displayed in a graphical user interface window which would be created and controlled by the GPOS's window manager.

The audio should be played on the speaker. The threads of the program that received, processed, and displayed the live video and audio should be programmed as real-time threads with appropriate parameters.

The program, however, should also provide the user with non-real-time features, such as threads that control the placement of the live video window on the display screen, and threads to monitor user-activated window controls to modify the video tint, brightness, and contrast, or the audio volume. Additionally, the real-time threads might generate data that could be handled by GPOS services, such as accounting data that should be written to log files. Such activities need not be handled in the real-time threads because they have no inherent deadlines.

The problem is how the real-time and non-real-time portions of the program could interact. For example, the real-time thread that processed the incoming video might also control brightness, if the real-time thread would have to incorporate any change in the brightness setting into its processing. Also, the real-time thread processing the video might need to detect and log framing errors while continuing to display the video. The log information may be moved to disk at any point, but the real-time and non-real-time threads must communicate easily.

Although many mechanisms for this intra-thread communication are possible, their essential component, viewed from the real-time thread, is an abstract wait-free queue [31]. Thus, the real-time thread would queue a work request to the queue and that queue operation would have a bounded completion time. The thread that dequeues and accomplishes the work request could exist in the GPOS, in a RTK device driver, or in the non-real-time portion of the program. The GPOS could be modified to support these queues directly, allowing it to define the structure and type of work requests and make them available to real-time threads.

Another approach would be to implement the queues in RTK device drivers that would export a non-real-time interface to the real-time threads. For example, the RTK disk device driver could export a `WriteNRTDataDisk` call that had a bounded completion time. The call would be an implementation of the wait-free queue operation. The RTK disk device driver would then write the data to disk in a non-real-time mode.

Another option would be for the GPOS to support the creation of the queues between the real-time and non-real-time threads of a program. This option is the most flexible, because all of the GPOS services are available to the non-real-time threads, and because the programmer defines the structure and type of work requests. Using the disk write example again, the real-time thread would queue a work request, a non-real-time thread in the program would dequeue it and determine that this was a work request to write data to the disk. The non-real-time thread would then call the GPOS disk services for file creation and write.

The non-real-time thread could also modify attributes of the video display, in response to user input, by changing the value of a global variable in the address space of the real-time thread. Some serialization would be required, because the non-real-time thread could change the value of the variable when the real-time thread was partially through processing the video data of one packet. This could happen if the real-time thread was executing during two, or more, subsequent real-time minor cycles. The easy solution is to have the real-time thread make a copy of the value at the start of processing and use that value to process a complete video packet. If the non-real-time thread then changed the value, the new value would not be used until the real-time thread started processing the next packet of video data.

3.11 Summary

The SP architecture defines three models—execution, resource, and programming—that together help an implementer modify a GPOS to support periodic real-time threads that require a guaranteed completion time. The execution model defines the sharing of the CPU (classed as preemptible in the resource model) between the GPOS and the RTK by the time division multiplexing of the execution of the RTK and GPOS. The resource model classifies the typical resources found on a computing machine (memory, display, network adapters, etc.) by the structure of the executive that manages how the resource is shared between the RTK and the GPOS. Thus, the resources in each class can use similar executives to manage the sharing between the GPOS and the RTK. The programming model defines the system calls a programmer uses to carry out real-time computation on an SP system.

Any implementation of the SP architecture has two requirements:

- There must be a minimum number of CPU cycles in any interval in which real-time threads execute.

- There must be an upper bound on the start time latency of the real-time intervals.

Typically these two requirements can be met by 1) virtualizing interrupts and 2) selectively disabling interrupts.

Chapter 4

Analysis of the Slotted Priority Architecture CPU Executive

4.1 Introduction

In Chapter 3 we noted that a system that supports periodic real-time tasks must be able to determine whether adding a new real-time task would prevent other real-time tasks from meeting deadlines. A GPOS need not address this because tasks executed by a GPOS do not have to meet deadlines and it is expected that the addition of new tasks will increase the average completion time for all tasks. However, periodic real-time tasks must be completed before their deadlines so one must determine, before additional real-time tasks join the system, whether the system will be able to meet its deadlines. This requires a feasibility analysis, also called admission control. This chapter derives an algorithm to test the feasibility of a set of periodic real-time tasks executing on a system implementing the SP architecture.

Scheduling real-time tasks with completion time deadlines is fundamentally different from scheduling non-real-time tasks. Typically, GPOSs use schedulers, such as first in first out (FIFO), shortest job first (SJF), or least remaining time first, (LRTF) [30]. The quantitative measures of system performance (such as task completion times) for these schedulers are computed using assumed stochastic distributions for the service requirement of each task and for the interarrival times of tasks at the ready queue. These stochastic methods estimate the average time a task waits in the queue [23]. For GPOSs this is the best that can be done, since the actual interarrival times and service requirements of its tasks cannot be predicted. Given that estimated average and a predicted service time, one can compute the expected completion time of any task: the average time required from when the task becomes ready to when it completes its computation. However, this expected completion time is useful only for analyzing a particular system during its design phase to predict its capacity under various loads or during off-line system tuning. It is cannot be used while a system is in operation to predict if any particular task will complete its computation before a given deadline[1]. However, a system that supports real-time tasks as defined here must be able to do just that.

Two separate issues are involved in scheduling real-time tasks: the scheduling policy, such as earliest deadline first (EDF), and admission control or feasibility analysis. In GPOSs and their schedulers, admission control is typically not an issue, since all new tasks are admitted to the system. However, as the number

[1] Note that none of the queuing theory analyses were intended to be used during the operation of a system to predict the actual values of metrics for particular tasks but rather were intended to be used during the design phase of the system or for off-line for operational tuning. During the design phase capacities of the various components of the system, such as the CPU, DASD, etc., are changed and the analysis run and various metrics, such as average wait time, are compared. The intent of this activity is to determine the combination of components that minimize and maximize, appropriately, the various metrics, including the total cost of the system.

of tasks increases, the system performance, as measured by, for example, average waiting in the queue, decreases. This is not acceptable, since real-time tasks must all meet their deadlines or the system is considered to have failed. Admission control (feasibility analysis) determines whether a set of real-time tasks with their costs, periods, and deadlines, operating under a particular scheduling policy will all meet their deadlines in each of their periods. This section develops a feasibility analysis for real-time tasks executed on an implementation of the SP architecture.

Such a feasibility analysis is fairly straightforward. Many of the existing analyses of real-time task systems can be modified slightly to accommodate the fact that these tasks are allowed only a fraction of the CPU, as opposed to traditional real-time systems where such tasks are assumed to have all of the CPU. The modifications are simple because the SP architecture requires that the lengths of the two types of minor cycles (and, therefore, the major cycle) be constant.

Section 4.2 reviews previous analyses. The first, the Rate Monotonic Priority Assignment, is the seminal work in analyzing real-time tasks and has essentially defined the fundamental task model.

A second task model and execution scheme, the cyclic executive, is described in Section 4.2.2. This model defines a set of rules to facilitate the static, off-line scheduling real-time task modules such that all tasks, when dispatched according to the calculated schedule, will meet their deadlines.

Work on accounting for the overhead of interrupt handlers is presented in Section 4.2.3. Although seemingly unrelated, it is very useful in analyzing real-time tasks executing on an implementation of the SP architecture.

4.2 Prior Work Related to this Analysis

The problem of assessing the feasibility of real-time tasks has been well studied. The seminal work on defining real-time tasks, along with two algorithms for assigning priorities, is by Liu and Layland [24]. The SP system uses this model for the tasks which it specifically supports. The cyclic executive model described in Section 4.2.2 also has some useful features.

4.2.1 The Liu and Layland Model of Periodic Tasks

A task, $\tau_i = (c_i, p_i)$ is characterized by a cost, c_i, and a period, p_i. The cost of a task is the time required to execute its instructions on a dedicated uniprocessor, and is assumed to remain constant. Task τ_i becomes ready to run, or is invoked, at the beginning of each interval of length p_i, i.e., at times $t = kp_i$ for $k = 0, 1, 2, \ldots$. A task is considered to have missed its deadline if the k^{th} invocation of task τ_i has not completed execution at time $t = (k + 1)p_i$.

Two algorithms for scheduling real-time tasks are specified and proven optimal. The rate monotonic algorithm is a static priority assignment method, and the earliest deadline first algorithm is a dynamic scheduling method. For either algorithm, the tasks are considered independent, that is, the individual invocations of a particular task do not depend on the invocation or completion of any other task. Also, any nonperiodic tasks in the system are special, initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and lack hard, critical deadlines.

Liu and Layland show that the rate-monotonic assignment algorithm is optimal in the following sense. If a feasible priority assignment exists for some task set, the rate-monotonic priority assignment is feasible for that set. The EDF algorithm is optimal in the sense that it will schedule any set of real-time tasks as long as they do not require more than 100 percent of the processor time.

The rate-monotonic priority assignment simply assigns priority to tasks by period length. Tasks with shorter periods receive higher priority. The utilization of a set of m tasks is given by:

$$U = \sum_{i=1}^{m} \frac{c_i}{p_i} \tag{4.1}$$

Liu and Layland show that the rate-monotonic priority assignment will result in a feasible schedule if:

$$U \leq m(2^{1/m} - 1). \tag{4.2}$$

For large m: $U \simeq \ln 2 \approx 0.67$.

At any time during execution, the currently executing task must be the task with the highest priority. Thus, the dispatcher must be preemptive since a task with a higher priority might become ready before the current task is finished[2]. The rate-monotonic scheduling algorithm requires a preemptive dispatcher and

[2] A note on the difference between a dispatcher and a scheduler in a real system. A scheduler is the task that decides the priority of each task in the system. A dispatcher is the task that, from a set of ready tasks (i.e., tasks that are awaiting execution) selects the next task to execute based on the priority set by the scheduler.

we say that the rate-monotonic scheduling algorithm is a preemptive scheduling algorithm.

The earliest deadline first (EDF) scheduling algorithm is dynamic, in the sense that a task's priority may vary from request to request. Simply stated, the algorithm schedules, from a set of ready tasks, the task with the closest deadline. Liu and Layland show that a set of m real-time tasks can be scheduled using EDF if and only if

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1 \tag{4.3}$$

Liu and Layland also give feasibility conditions for an alternate model in which some tasks are scheduled by the rate monotonic algorithm and the rest by the earliest deadline first algorithm. Jeffay and Stone solve this alternate model exactly (see Section 4.2.3).

4.2.2 Cyclic Executives

A cyclic executive is the execution model used most often to schedule hard real-time tasks in embedded systems. A cyclic executive is a dispatcher that continually executes a schedule for a fixed set of periodic tasks. The schedule is made up of major and minor cycles. A task in a cyclic executive is generally characterized by the triple (c_i, p_i, d_i), where c_i is the cost, p_i is the period, and d_i is the deadline. The schedule is so organized that all tasks meet their deadlines. For a set of n tasks the constraints on the length m of a minor cycle are [2]:

- $m < d_i$ for $i = 1$ to n.

- m must be greater than the the longest task or subtask.

- m must divide the major cycle M.

- $m + (m - gcd(m, p_i)) \leq d_i$ for $i = 1$ to n.

Some portions of the schedule do not specify the execution of any periodic task. In these regions aperiodic tasks, such as non-real-time, may execute. The schedule is prepared off-line and cannot be changed while the system is executing. It has been shown that finding a schedule is NP-hard for one processor [10]. Although the cyclic executive does allow the execution of aperiodic and periodic tasks, it fits poorly into a general-purpose operating system because it is static.

The SP architecture borrows notation and concepts from the work on cyclic executives. However, in the SP architecture, a minor cycle is either a time interval that contains only real-time tasks (if any real-time tasks are ready to run) or a time interval that contains only non-real-time tasks. In addition, the SP model permits dynamic adjustment of the relative size of the real-time and non-real-time intervals and there is no static schedule. Scheduling is all done on-line.

4.2.3 Interrupt Overhead Cost Accounting

In a real-time system that manages physical devices with interrupts and interrupt handlers, the feasibility analysis must consider the effect of executing interrupt handlers on the total execution time of a particular real-time task. Interrupt handlers are executed below the scheduler's awareness and affect how long a real-time task takes to complete its computation. The amount of time depends on the frequency and duration of interrupt handler execution. The SP architecture eliminates the effect of interrupt handlers on feasibility by disallowing their execution in real-time minor cycles.

Another approach is to account for the time the interrupt handlers execute, i.e., make that time visible to the scheduler. Jeffay and Stone solve the feasibility conditions exactly for the alternate model given by Liu and Layland. In the work by Jeffay and Stone the effect of execution of interrupt handlers is taken into account when deciding the feasibility of a set of periodic real-time tasks [20]. The difficulty is determining the amount of CPU time the interrupt handlers consume because their interarrival times and costs are known only stochastically.

The Jeffay and Stone model considers two types of tasks, interrupt handlers and application tasks (periodic real-time tasks). An application task T is a pair (c, p), where c is the maximum amount of time required to execute task T and p is the interval between invocations of task T. An interrupt handler I is a pair (e, a), where e is the maximum amount of processor time required to execute the interrupt handler I and a is the interval between occurrences of I.

The feasibility analysis then derives the amount of CPU time consumed by the interrupt handlers. A function $f(t)$ is specified to be the CPU time the interrupt handlers consume in the interval $[0, t]$. It is shown that for a set of n application tasks $(c_1, p_1), \ldots, (c_n, p_n)$ and m interrupt handlers $(e_1, a_1), \ldots, (e_m, a_m)$, the amount of CPU time consumed by the m interrupt handlers in an interval of length

l is:

$$\forall l > 0, f(l) = \begin{cases} 0 & \text{if } l = 0 \\ f(l-1) & \text{if } f(l-1) = \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \\ f(l-1)+1 & \text{if } f(l-1) < \sum_{i=1}^{m} \left\lceil \frac{l}{a_i} \right\rceil e_i \end{cases} \tag{4.4}$$

The application tasks will then be feasible if and only if for all $L, L > 0$,

$$L - f(l) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \tag{4.5}$$

Equation 4.5 states that L units of work are available to both application tasks and interrupt handlers and $L - f(l)$ units of work are available to the application tasks alone. Thus, if the applications need fewer units of work than are available in every interval the application set is feasible. Jeffay and Stone also show that Equation 4.5 is equivalent to the Liu and Layland feasibility condition, for their alternate model, as Equation 4.6 is equivalent to Equation 4.7, which they also show.

$$\forall L \geq 0, L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \tag{4.6}$$

is equivalent to

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1 \tag{4.7}$$

Jeffay and Stone also show that one need not look at all Ls to determine if a particular task set is feasible; in fact one need only look at a polynomial number of Ls.

4.2.3.1 Relationship of SP to the Interrupt Handler Work

The SP architecture proposed in this paper is a special case of the model developed by Jeffay and Stone. In an implementation of the SP architecture the GPOS can be considered as a single interrupt handler with cost mc_{nrt} and period MC. Since MC is constant with respect to an analysis, the recursive function, $f()$ in Equation 4.4, can be expressed in a closed form when analyzing the CPU executive in an SP implementation.

4.3 Determining Feasibility in an SP Implementation

We start with a simple analysis that requires assumptions about the relationships between the costs of the real-time tasks and the length of the real-time minor cycle and between the periods of the real-time tasks and the length of the major cycle. Our goal is to provide a simple computational method to determine appropriate lengths of the real-time minor cycle and the major cycle for SP implementations, assuming that currently executing tasks meet the assumptions, or can be made to, by choosing appropriate cycle lengths. This simple analysis also serves as a starting point for the subsequent, more general analysis and illustrates that feasibility for the simple case is a function of processor utilization, as in the Liu and Layland work.

The analysis first imposes rather strict assumptions and considers a case in which feasibility is a function of processor utilization, similar to the analysis given by Liu and Layland [24]. The assumptions are then relaxed in the analysis of Section 4.3.1: feasibility then requires a test for all intervals, but covers the general case where the costs and periods of the real-time tasks are unrelated to the lengths of the minor cycles. Finally, in Section 4.3.2, the number of intervals that have to be checked to determine feasibility is limited so that the computation of feasibility is tractable.

The goal of the analyses is to show that a set τ of n tasks, when executed on an implementation of the SP architecture, is feasible: it is possible to schedule the tasks so that each invocation of each task finishes at or before its deadline.

Time is viewed as an infinite series of discrete points starting at time $t = 0$. The distances between these discrete points are divided into two classes: real-time time units and non-real-time time units. The assumptions for the initial analysis are:

For a set, τ, of m real-time tasks, $\tau = ((c_1, p_1), \ldots, (c_m, p_m))$ where:

1. mc_{rt} divides c_i, $\forall i = 1$ to m.
2. $mc_{rt} + mc_{nrt}$ divides p_i, $\forall i = 1$ to m.
3. The tasks are scheduled by the EDF algorithm.
4. The system starts with a non-real-time minor cycle.

Theorem 1: τ is feasible if and only if

$$\sum_{i=1}^{m} \frac{c_i}{p_i} \leq \frac{mc_{rt}}{mc_{rt} + mc_{nrt}} \tag{4.8}$$

The necessity of Equation 4.8 is obvious since, as discussed in Section 4.2.3, the left hand side is the utilization of the real-time tasks in the system and it is clear that one cannot schedule a set of tasks that require a utilization greater than the fraction $\frac{mc_{rt}}{MC}$.

The sufficiency of Equation 4.8 is now proved by contradiction. Assume Equation 4.8 is true but yet τ is not feasible, that is, $\exists$ a time t_d such that at t_d some task does not meet its deadline. Since a task can only overflow at the end of its period and $mc_{rt} + mc_{nrt}$ divides all periods, then t_d is a multiple of $mc_{rt} + mc_{nrt}$. Choose a point $t < t_d$ which is the greatest of:

- 0
- the end of the last idle period prior to t_d
- the last time a task with a deadline after t_d executed

This time t is also a multiple of $mc_{rt} + mc_{nrt}$. Of course 0 is a multiple of $mc_{rt} + mc_{nrt}$. Note that, given the assumption that mc_{rt} divides all of the c_i, either a single real-time task executes in any real-time minor cycle or no real-time tasks execute in a real-time minor cycle. Thus, the end of the last idle period is at the end of a real-time minor cycle, as is the last time a task with a deadline after t_d executed. Since the system starts with a non-real-time minor cycle, times at which the ends of the real-time minor cycles occur all divide $mc_{rt} + mc_{nrt}$.

The amount of work required in the interval, $[t, t_d]$, is

$$\sum_{i=1}^{m} \left\lfloor \frac{t_d - t}{p_i} \right\rfloor c_i \tag{4.9}$$

If some task misses its deadline at t_d then the following must be true.

$$\sum_{i=1}^{m} \left\lfloor \frac{t_d - t}{p_i} \right\rfloor c_i > \left\lfloor \frac{t_d - t}{mc_{rt} + mc_{nrt}} \right\rfloor mc_{rt} \tag{4.10}$$

The right hand side of Equation 4.10 is the amount of processor time available to real-time tasks in the interval, $[t, t_d]$. Since $t_d - t$ is a multiple of $mc_{rt} + mc_{nrt}$ we can drop the floor on the right hand side of Equation 4.10 and since $x \geq \lfloor x \rfloor \, \forall x$ we can drop the floor on the left hand side of Equation 4.10. We now have

$$\sum_{i=1}^{m} \frac{t_d - t}{p_i} c_i > \frac{t_d - t}{mc_{rt} + mc_{nrt}} mc_{rt} \tag{4.11}$$

Dividing through by $t_d - t$, we get

$$\sum_{i=1}^{m} \frac{c_i}{p_i} > \frac{mc_{rt}}{mc_{rt} + mc_{nrt}} \tag{4.12}$$

This is a contradiction of Equation 4.8 and thus proves the sufficiency of Equation 4.8.

4.3.1 Relaxing Assumptions

Assumptions in Section 4.3 are somewhat restrictive. All the assumptions (except the assumption that an earliest deadline scheduler is used to schedule the real-time threads) can be relaxed at the cost of a feasibility test with greater time complexity.

We start by showing:

Lemma 1: For all $l, l \geq 0$,

$$\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}) \tag{4.13}$$

is the greatest lower bound on the number of real-time processor units in the interval $[t, t+l]$.

Proof: For all $t, t \geq 0$, in the interval $[t, t+l]$, at least $\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt}$ time units are available for executing real-time tasks and at least $\left\lfloor \frac{l}{MC} \right\rfloor mc_{nrt}$ time units are available for executing non-real-time tasks. Of the remaining

$$l - \left(\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \left\lfloor \frac{l}{MC} \right\rfloor mc_{nrt} \right) = l - \left\lfloor \frac{l}{MC} \right\rfloor MC = l \bmod MC \tag{4.14}$$

processor units in the interval, at most mc_{nrt} of these can be non-real-time units. Thus, if $mc_{nrt} \geq l \bmod MC$, then at least $l \bmod MC - mc_{nrt}$ additional processor units must be available for executing real-time tasks. Therefore the greatest lower bound on the number of real-time processor units in the interval $[t, t+l]$ is

$$\left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}) \tag{4.15}$$

Theorem 2: A set of periodic tasks $\tau = \{(c_1, p_1), c_2, p_2), \ldots, c_n, p_n)\}$ can be scheduled on an SP implementation with real-time minor cycle length mc_{rt} and non-real-time minor cycle length mc_{nrt} if and only if for all $L \geq 0$:

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \leq \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \tag{4.16}$$

where $MC = mc_{nrt} + mc_{rt}$.

Proof: ($\Leftarrow$) A set of tasks can be scheduled only if for all $l, l \geq 0$, the amount of processor time available to real-time tasks in the interval $[0, l]$, is at least as big as the work requested by the invocations of tasks with deadlines in $[0, l]$.

In $[0, l]$, each real-time task will require $\left\lfloor \frac{l}{p_i} \right\rfloor c_i$ units of processor time to ensure no invocation of that task misses its deadline in the interval $[0, l]$. Thus the work requested by all invocations of tasks with deadlines in $[0, l]$ is $\sum_{i=1}^{n} \left\lfloor \frac{l}{p_i} \right\rfloor c_i$.

By Equation 4.13, the amount of processor time available to real-time tasks in the interval $[0, l]$ is at least Equation 4.13. Thus, since Equation 4.13 is a greatest lower bound, τ can be scheduled only if

$$\sum_{i=1}^{n} \left\lfloor \frac{l}{p_i} \right\rfloor c_i \leq \left\lfloor \frac{l}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, l \bmod MC - mc_{nrt}) \qquad (4.17)$$

Note that no assumptions are made (or needed) about whether or not time 0 corresponds to the start of a real-time or non-real-time minor cycle.

($\Rightarrow$) To show sufficiency of Equation 4.16, it suffices to show that if a task system τ satisfies Equation 4.16 for all $L, L > 0$, then a deadline-driven scheduler will succeed in scheduling τ. This is shown by contradiction.

Assume that for all $L, L > 0$, τ satisfies Equation 4.16 but yet a real-time task misses a deadline when scheduled according to a deadline driven algorithm. Let t_d be the earliest time at which a deadline is missed and let t be the greater of:

- the end of the last interval in a real-time minor cycle during which which no real-time task executed (or 0 if all processor units in every real-time minor cycle up to time t_d have been consumed), or,

- the latest time before t_d at which an invocation of a real-time task with deadline after time t_d executes (or 0 if such an invocation does not execute prior to T_d).

By the choice of t, no invocation of a real-time task with deadline after t_d executes in the interval $[t, t_d]$. If deadline-driven scheduling is performed, then the processor demand in the interval $[t, t_d]$, is $\sum_{i=1}^{n} \left\lfloor \frac{(t_d - t)}{p_i} \right\rfloor c_i$. Moreover, at least

$$\left\lfloor \frac{t_d - t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, (t_d - t) \bmod MC - mc_{nrt}) \tag{4.18}$$

real-time processor units are available for real-time tasks in $[t, t_d]$. Since a deadline is missed at time t_d, it follows that

$$\sum_{i=1} n \left\lfloor \frac{t_d - t}{p_i} \right\rfloor c_i > \left\lfloor \frac{t_d - t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, (t_d - t) \bmod MC - mc_{nrt}). \tag{4.19}$$

However, this contradicts the assumption that τ satisfies Equation 4.16 for all L. Hence, if τ satisfies Equation 4.16 then a deadline-driven scheduler will succeed in scheduling τ. It follows that satisfying Equation 4.16 for all $L, L > 0$, is a sufficient condition for feasibility.

4.3.2 Reducing Computation Complexity

A feasibility computation using Equation 4.16 requires evaluation for all positive integers up to the point where one can trust that all possible permutations of real-time task invocations and periods and minor cycles have occurred. The time complexity of this computation is exponential in the number of periods, since such a point is, in general, the product of the periods of each of the real-time tasks. In certain special cases, though, the value of such a point is small and the time required for the computation is reasonable. An example of such a special case is when all of the periods are the same, $I = p_1 = p2 = \ldots = p_n$. In this case

Equation 4.16 would have to be evaluated only for the integer values $0 \leq I$ to determine if the task set is feasible. Other special cases, where Equation 4.16 is computationally reasonable, include task sets where all of the periods are multiples of some integer or when there are relatively few tasks.

This section proves that the evaluation of Equation 4.16 need only be done for integer values of L *that are multiples* of any of the periods up to a computed value B, which is often much smaller than the product of the periods.

To start, define the total utilization, U_T, of the SP system as:

$$U_T = \sum_{i=1}^{n} \frac{c_i}{p_i} + \frac{mc_{nrt}}{MC} \tag{4.20}$$

This is the fraction of processor time required by the real-time tasks plus the fraction of processor time dedicated to the GPOS.

Theorem 3: Let τ be a task system as given for Theorem 2 with $U_T < 1$. Let

$$B = \frac{mc_{nrt}}{1 - U_T} \tag{4.21}$$

and let $P = \{kp_i | kp_i < B \wedge k \geq 0 \wedge 1 \leq i \leq n\}$ be the set of nonnegative multiples, less than B, of the periods of the real-time tasks. τ will be feasible if and only if for all L, $L \in P$:

$$\left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \tag{4.22}$$

Proof: ($\Rightarrow$) The necessity of Equation 4.22 for all L, $L \in P$ is a direct consequence of Theorem 2.

($\Leftarrow$) The sufficiency of Equation 4.22 is shown in two parts. First, it is shown that Equation 4.22 need hold at only the multiples of the periods for the task to be feasible. Second, it is shown that only values of L less than B need to be considered.

Let $Q = \{kp_i | k \geq 0 \wedge 1 \leq i \leq n\}$. Choose $t, t' \in Q$, such that no $r \in Q, t < r < t'$ exists. Let ϵ be an integer such that $0 \leq \epsilon < t' - t$. It follows that for all i, $1 \leq i \leq n$, $\left\lfloor \frac{t}{p_i} \right\rfloor = \left\lfloor \frac{(t+\epsilon)}{p_i} \right\rfloor$. If Equation 4.22 is satisfied at t, then adding ϵ to t on the rhs preserves the inequality:

$$\left\lfloor \frac{t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, t \bmod MC - mc_{nrt}) \geq \sum_{i=1}^{n} \left\lfloor \frac{t+\epsilon}{p_i} \right\rfloor c_i$$

The next step of the proof is to add ϵ to each instance of t on the lhs, giving

$$\left\lfloor \frac{t+\epsilon}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, t + \epsilon \bmod MC - mc_{nrt}) \geq \sum_{i=1}^{n} \left\lfloor \frac{t+\epsilon}{p_i} \right\rfloor c_i \tag{4.23}$$

However, some explanation is in order with regard to why the inequality is preserved. Clearly, $\left\lfloor \frac{t+\epsilon}{MC} \right\rfloor mc_{rt} \geq \left\lfloor \frac{t}{MC} \right\rfloor mc_{rt}$, but two cases need to be examined when ϵ is added to t in $\mathrm{MAX}(0, t \bmod MC - mc_{nrt})$:

Case 1:

$$\mathrm{MAX}(0, t + \epsilon \bmod MC - mc_{nrt}) \geq \mathrm{MAX}(0, t \bmod MC - mc_{nrt})$$

and Equation 4.23 holds.

Case 2:

$$\mathrm{MAX}(0, t + \epsilon \bmod MC - mc_{nrt}) < \mathrm{MAX}(0, t \bmod MC - mc_{nrt}).$$

However, if this is true, then

$$t \leq kMC < t + \epsilon \text{ for integer } k \geq 1$$

and ϵ is large enough to cause $t + \epsilon$ to be greater than at least the next multiple of MC. But then,

$$\left\lfloor \frac{t+\epsilon}{MC} \right\rfloor mc_{rt} = \left\lfloor \frac{t}{MC} \right\rfloor mc_{rt} + mc_{rt}$$

and since

$$0 \leq \mathrm{MAX}(0, t \bmod MC - mc_{nrt}) \leq mc_{rt} - 1$$

then

$$\begin{aligned}
\left\lfloor \frac{t+\epsilon}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, t + \epsilon \bmod MC - mc_{nrt}) &\geq \\
\left\lfloor \frac{t}{MC} \right\rfloor mc_{rt} + mc_{rt} + \mathrm{MAX}(0, t + \epsilon \bmod MC - mc_{nrt}) &\geq \\
\left\lfloor \frac{t}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, t \bmod MC - mc_{nrt}) &\geq \sum_{i=1}^{n} \left\lfloor \frac{t}{p_i} \right\rfloor c_i
\end{aligned}$$

and Equation 4.23 holds.

Thus if Equation 4.22 holds at the multiples of the p_i, then it holds at all values between those multiples. So it suffices only to consider the positive multiples when using Equation 4.16.

Next we show that only those multiples of the p_i less than B need to be considered when using Theorem 2 to determine feasibility of a task set. Consider the following function:

$$g(L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i - \left(\left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \right) \tag{4.24}$$

By Theorem 2, τ will be feasible if and only if $g(L) \leq 0$ for all $L \geq 0$. $g(L)$ is bounded from above by the function:

$$h(L) = (U_T - 1)L + mc_{nrt}$$

To see this, first show

$$g(L) \leq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i - \left(L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \mathrm{MIN}(L \bmod MC, mc_{nrt}) \right) \tag{4.25}$$

from Equation 4.24. To see this we examine two cases.

Case 1: L in this case is a multiple of MC. Thus:

$$\mathrm{MAX}(0, L \bmod MC - mc_{nrt}) = 0$$

and

$$\mathrm{MIN}(L \bmod MC, mc_{nrt}) = 0$$

Since

$$\begin{aligned} L &= \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} + \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} \\ L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} &= \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} \\ L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \mathrm{MIN}(L \bmod MC, mc_{nrt}) &= \\ \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} &+ \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \end{aligned}$$

Thus, Equation 4.25 follows when L is a multiple of MC.

As an example of case 1 consider, as depicted in Figure 4.1, a system with $L = 16$, $MC = 4$, $mc_{rt} = 1$, $mc_{nrt} = 3$

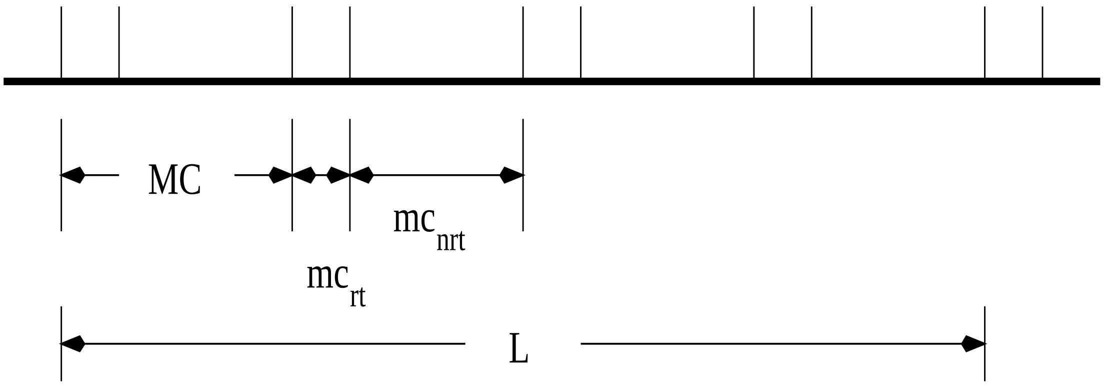

Figure 4.1: L is a multiple of MC

$$\begin{aligned}
L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} &= \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \\
16 - \left\lfloor \frac{16}{4} \right\rfloor 3 &= \left\lfloor \frac{16}{4} \right\rfloor 1 + \mathrm{MAX}(0, 16 \bmod 4 - 3) \\
16 - (4 \times 3) &= (4)1 + 0 \\
4 &= 4
\end{aligned}$$

Case 2: L is *not* a multiple of MC

To see that Equation 4.25 holds we must show that

$$\begin{aligned}
&L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \mathrm{MIN}(L \bmod MC, mc_{nrt}) \\
&\quad \leq \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt})
\end{aligned} \tag{4.26}$$

Algebraic manipulation and replacement are applied, noting that $MC = mc_{nrt} +$

mc_{rt}.

$$L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \mathrm{MIN}(L \bmod MC, mc_{nrt})$$
$$\leq \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt})$$
$$L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt}$$
$$\leq \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt})$$
$$L - \left(\left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} + \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} \right)$$
$$\leq \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt})$$

$$L - \left((mc_{nrt} + mc_{rt}) \left\lfloor \frac{L}{MC} \right\rfloor \right)$$
$$\leq \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt})$$
$$L - \left(MC \left\lfloor \frac{L}{MC} \right\rfloor \right)$$
$$\leq \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt})$$

Since $x \bmod y = x - y \left\lfloor \frac{x}{y} \right\rfloor$, $y \neq 0$

$$L \bmod MC \leq \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt})$$

Thus for case 2 it remains to show that

$$\mathrm{MAX}(0, L \bmod MC - mc_{nrt}) + \mathrm{MIN}(L \bmod MC, mc_{nrt}) \geq L \bmod MC$$

If $0 \leq L \bmod MC \leq mc_{nrt}$ then

$$\mathrm{MAX}(0, L \bmod MC - mc_{nrt}) = 0$$

and

$$\mathrm{MIN}(L \bmod MC, mc_{nrt}) = L \bmod MC$$

Thus

$$\text{MAX}(0, L \bmod MC - mc_{nrt}) + \text{MIN}(L \bmod MC, mc_{nrt}) = L \bmod MC$$

If $mc_{nrt} < L \bmod MC < MC$, then

$$\text{MAX}(0, L \bmod MC - mc_{nrt}) = L \bmod MC - mc_{nrt}$$

and

$$\text{MIN}(L \bmod MC, mc_{nrt}) = mc_{nrt}$$

Thus

$$\text{MAX}(0, L \bmod MC - mc_{nrt}) + \text{MIN}(L \bmod MC, mc_{nrt}) = L \bmod MC$$

Therefore, Equation 4.26 is true. As an example of case 2 consider, as depicted in Figure 4.2, a system with $L = 19$, $MC = 4$, $mc_{rt} = 1$, $mc_{nrt} = 3$.

$$\begin{aligned}
L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} &\geq \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \text{MAX}(0, L \bmod MC - mc_{nrt}) \\
19 - \left\lfloor \frac{19}{4} \right\rfloor 3 &\geq \left\lfloor \frac{19}{4} \right\rfloor 1 + \text{MAX}(0, 19 \bmod 4 - 3) \\
19 - (4 \times 3) &\geq (4)1 + 0 \\
7 &\geq 4
\end{aligned}$$

Now, starting with Equation 4.25

$$g(L) \leq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i - \left(L - \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} - \text{MIN}(L \bmod MC, mc_{nrt}) \right)$$

then rearrange

$$g(L) \leq \sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i - L + \left\lfloor \frac{L}{MC} \right\rfloor mc_{nrt} + \text{MIN}(L \bmod MC, mc_{nrt})$$

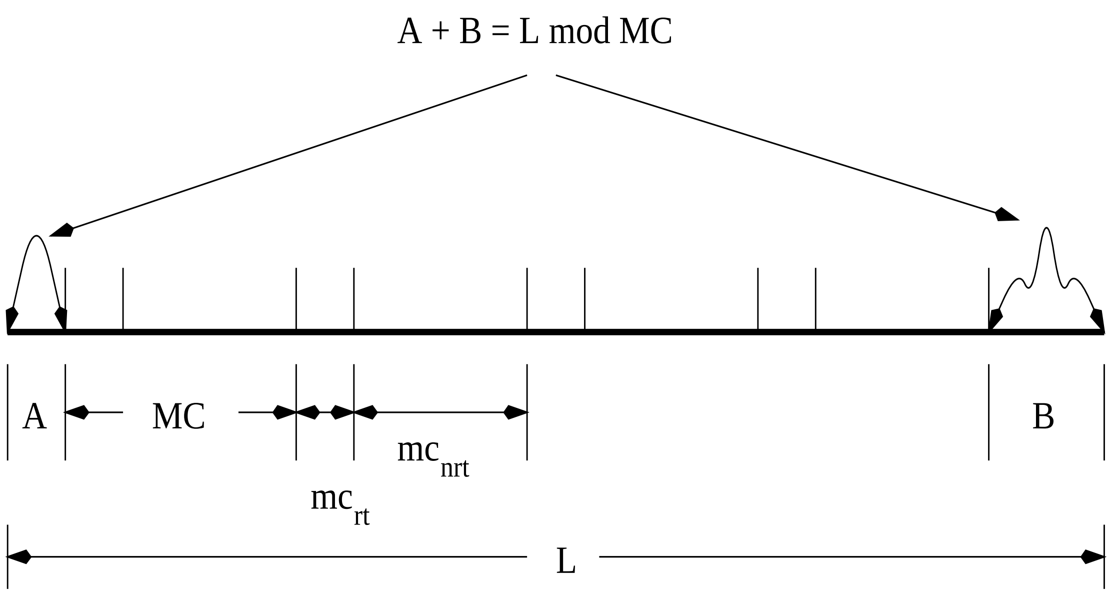

Figure 4.2: L is *not* a multiple of MC

dropping the floors preserves the inequality

$$g(L) \leq \sum_{i=1}^{n} \frac{L}{p_i} c_i - L + \frac{L}{MC} mc_{nrt} + \text{MIN}(L \bmod MC, mc_{nrt})$$

rearranging

$$g(L) \leq L \sum_{i=1}^{n} \frac{c_i}{p_i} - L + L \frac{mc_{nrt}}{MC} + \text{MIN}(L \bmod MC, mc_{nrt})$$

noting that $\text{MIN}(L \bmod MC, mc_{nrt}) \leq mc_{nrt}$

$$g(L) \leq L \sum_{i=1}^{n} \frac{c_i}{p_i} - L + L \frac{mc_{nrt}}{MC} + mc_{nrt}$$

factoring out L and rearranging

$$g(L) \leq L \left(\sum_{i=1}^{n} \frac{c_i}{p_i} + \frac{mc_{nrt}}{MC} - 1 \right) + mc_{nrt}$$

substituting U_T

$$= (U_T - 1)L + mc_{nrt}$$

Thus, it is seen that $g(L) \leq h(L)$ for all $L \geq 0$. $h(L)$ is a linear function in L with slope $(U_T - 1)$ and an L intercept at the point $L = B = \frac{mc_{nrt}}{(1-U_T)}$. Since $U_T < 1$, $h(L)$ has a negative slope and thus for all $L \geq B$, $g(L) \leq h(L) \leq 0$. Thus τ will be feasible if and only if $g(L) \leq 0$ for all $L \in P$. This proves Theorem 3.

4.4 Summary

The analysis of the SP CPU executive presented in this section is derived from the seminal work of Liu and Layland [24] and the work by Jeffay and Stone [20]. Liu and Layland defined the fundamental characteristics of real-time tasks and proved for their model that the feasibility of a task set is a function of processor utilization when scheduled by an EDF scheduler.

The initial analysis presented here shows that a simple test for feasibility based on utilization can be used for a task set an SP implementation, given certain assumptions about the relationships between the minor cycle lengths and the costs and periods of the real-time tasks. In the general case, where the assumptions about the relationships between periods and costs are removed, the feasibility test can no longer be a simple function of processor utilization but must be a set of tests, one for each interval of integer length up to some value. In some systems that value can be rather large, since it may be the product of all of the periods. However, the analysis continues to show that all intervals must be checked, or rather, only those intervals which are a multiple of one of the periods and less than a computed value that is typically much smaller than the product of all the periods.

Chapter 5

An Implementation of the Slotted Priority Architecture

5.1 Introduction

The previous chapter presented a derivation of a feasibility analysis that can be used to determine if a set of periodic real-time threads executing on a system implementing SP can be scheduled by an earliest deadline first scheduler so that all of the threads meet their deadlines. This chapter describes the details of an actual implementation of the SP architecture that employs EDF scheduling. This chapter also presents the results of a set of experiments used to verify the implementation and show its effect on GPOS performance.

To accomplish the implementation, the system must be able to precisely execute the RTK at the beginning of each real-time minor cycle. Moreover, the RTK must be able to determine the minimum number of processor cycles available to execute real-time threads in any real-time minor cycle. The first issue is that the enabling and disabling of interrupts has to be virtualized. This is because the unrestricted use of disabled interrupts to, for example, enforce mutual exclusion in the GPOS kernel can cause unpredictable delays in executing the RTK. The second issue is that during the real-time minor cycles, interrupts must be selectively disabled, because the feasibility analysis assumes that all CPU cycles in a real-time minor cycle, except for the cycles used to execute the RTK itself, are available to real-time threads.

Section 5.2 describes the hardware system used for the implementation. An implementation of the SP architecture is very specific to its hardware system as well as to its GPOS. Some of the software modules described in Section 5.3 are specific to the GPOS, in this case the IBM MicroKernel. Others are generic and could be easily ported to an implementation using any GPOS. Section 5.4 introduces a set of experiments that show that this particular implementation of the SP architecture meets the correctness criteria established by the architecture defined in Chapter 3. Section 5.5 describes issues that should be considered when when attempting to port an implementation of the SP architecture from a particular hardware/GPOS combination to another combination. In addition, Section 5.5 describes issues to consider when choosing a minor cycle length.

5.2 Hardware Description

The implementation described here was constructed on an IBM PS/2 Model 95 workstation which uses the Intel 80486DX2 (i486), as the main processor with a 20 nanosecond CPU internal cycle time (50 MHz CPU internal bus clocked with a 25MHz external clock signal). The execution times of instructions in the i486 instruction set vary from one cycle to 329 cycles with the average being about five cycles[1]. The DMA controller operates at 25MHz and can transfer a single byte in approximately 740-900 nanoseconds, depending on whether the operation is a read or a write to memory and whether the memory page is present [14]. In burst mode, the DMA controller can transfer data between $300 + (320 + 160) \times n$ and $300 + (320 + 280) \times n$ bytes/ns, where $n =$ the number of bytes in the burst. This model has no level-2 cache, and its first level cache is a 8KB, 4-way, write-through, internal cache [14]. This particular machine has 32 megabytes of 80 ns main memory and an IBM MicroChannel Bus. During a burst, an adapter can continue to hold access to the bus for up to 7.8 microseconds after an adapter with a higher arbitration level has raised the bus request line [12]. The microprocessor has a high priority arbitration level and may thus have to wait only 7.8 microseconds for access to the bus while another adapter finishes a burst request.

The motherboard contains two programmable interrupt controller (PIC) circuits [13, 18], one programmable system timer circuit, and a Real-Time Clock Plus Battery Backed CMOS Ram (RTClock) with a periodic interval timer [13, 26].

The system timer, the RTClock, and the PICs are sources of hardware interrupts to the i486 CPU. The system timer provides a number of different modes that determine when and if an interrupt will be generated. In the mode used by

[1]Repeat `REP` instructions perform a simple action, such as moving a word of memory from one address to another, for a variable number of times. The number of times the simple action is repeated is controlled by the programmer. With a single repeat instruction the programmer can thus move large regions of memory. A potential problem is that if the processor architecture does not allow interrupts between the actions of a repeat instruction, long, unpredictable, and irrevocable delays may occur before an interrupt may be handled. The i486 architecture allows interrupts between the actions of repeat instructions [17].

the IBM MicroKernel, the timer generates an interrupt when the value of a two-byte countdown register reaches zero. The countdown resumes when either a new value is written into the countdown register or a control word is written to the circuits control register. The timer frequency is 1.193 MHz, which decrements the countdown register each 838.223 nanoseconds. The MicroKernel uses this timer as its fundamental hardware timer and derives all other time measurements from it.

Figure 5.1 illustrates connections among the CPU, the system timer (8254), the Real-Time Clock, and the programmable interrupt controllers (8259A). The two PICs are connected to the CPU interrupt line in a cascade. The slave PIC's interrupt line is connected to the number 2 input line of the master PIC, whose interrupt line is connected to the CPU's interrupt input. When a device attached to an input line of the slave PIC raises its interrupt line, the slave PIC, according to its internal algorithm for multiplexing interrupts, will raise its interrupt line, which is input to the master PIC. At the appropriate time the master PIC will raise its interrupt line into the CPU, and when the CPU acknowledges the interrupt, the master PIC will transfer the appropriate codes to the CPU to indicate which device is requesting service.

The PICs multiplex interrupts from peripheral devices onto the single interrupt line of the i486. In addition, the PICs provide a means of assigning priorities to each incoming interrupt line as it maps each line to a physical device. The priority assignment feature is important for this implementation of the SP architecture, because the line used by the timer that defines slots (see Section 3.6) is given the

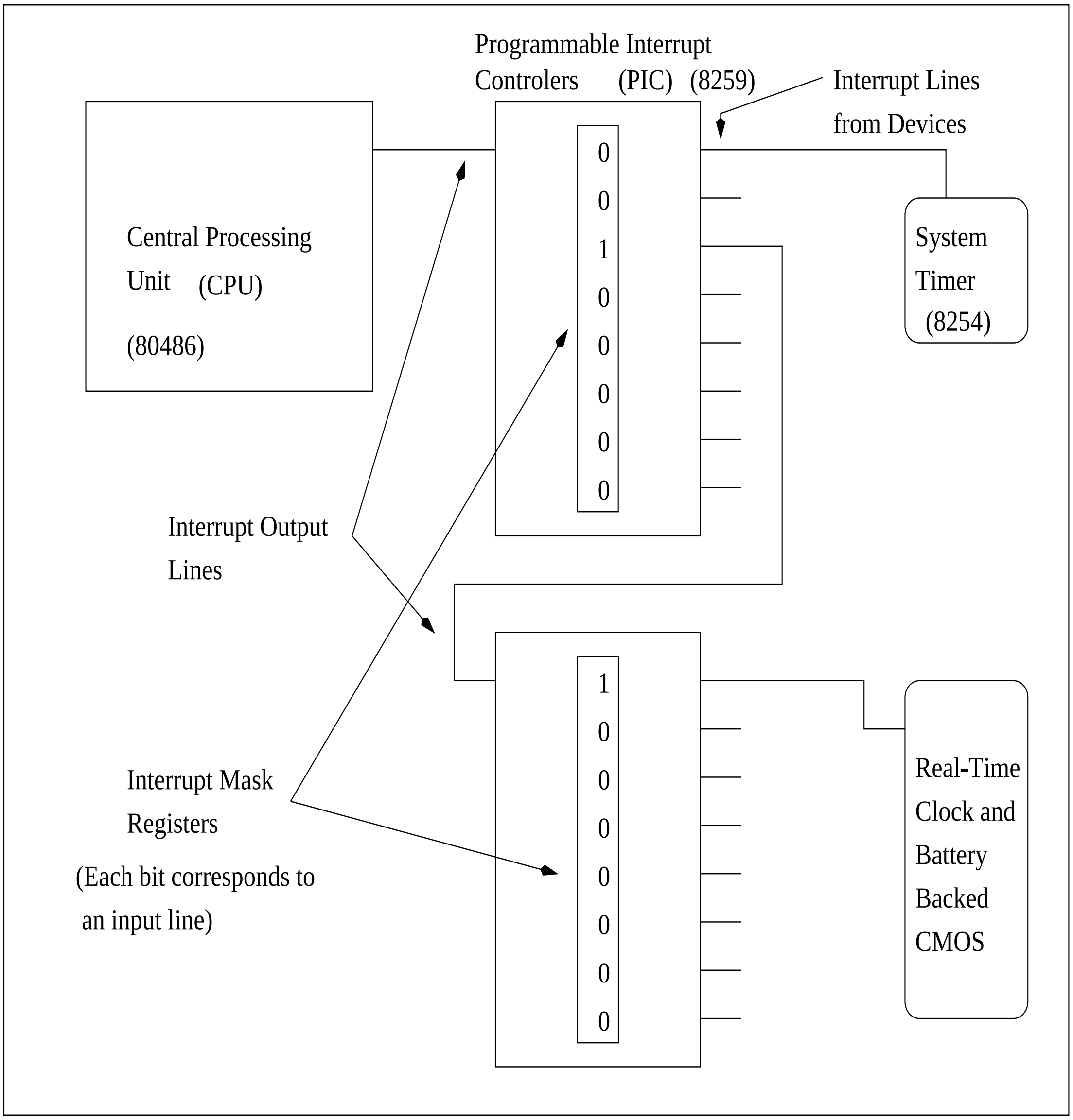

Figure 5.1: Some Important Motherboard Circuits. The bit values in the interrupt mask registers indicate the state where only the RTClock interrupt is allowed to pass through the PICs to the CPU. A bit value of '1' allows the interrupt through and a value of '0' stops the interrupt at the PIC.

highest priority. This priority assignment decreases the variability in slot lengths, since the SP timer will interrupt the CPU before any other device that becomes ready at the same time. Also, each PIC has a bit mask which can enable and disable each line, and therefore each device, individually.

The RTClock provides a programmable interval interrupt function. The interval may be set to one of 16 values from 30.5 microseconds to 500 milliseconds. The RTClock generates an interrupt that defines the start of each slot.

The host operating system is the IBM MicroKernel using OSF/1 as the dominant personality[2]. The host operating system was chosen primarily because the kernel source code and a build environment were readily available.

5.2.1 Periodic Interrupt Generators

The version of the IBM MicroKernel used in this implementation of the SP architecture did not use the RTClock for periodic interrupts. Instead the implementers of the MicroKernel chose to use the system timer and schedule variable interrupts as necessary to process events on the event queue. This minimized the changes necessary to the IBM MicroKernel, since the SP implementation could use the RTClock independently[3].

[2]The IBM MicroKernel is a version of the Mach Operating System developed at Carnegie Mellon University. The dominant personality refers to the operating system implementation with which the user interacts.

[3]IBM's OS/2 operating system, on the other hand, uses the RTClock for its timing services, but it does not use the system timer. So to implement SP in OS/2 one would have to use the system timer for its periodic interrupts or share the RTClock. The latter solution would require more changes to the host kernel. A host kernel that used both the RTClock and the system timer would need to be modified to allow SP to share one of the sources of interrupts.

5.2.2 Use of Hardware Task Switch Mechanism

Another issue that affects an SP implementation on this hardware is that the IBM MicroKernel does not use the i486 hardware task switch mechanism. The i486 provides hardware support for task or context switches. If an operating system opts to use the hardware support for task switching, each task is allocated a task state segment (TSS) which is a segment of memory with a descriptor in the global descriptor table (GDT). A descriptor is an 8-byte entry, in one of the descriptor tables, that contains information about a memory object. There are segment, interrupt gate, and task gate descriptors. Information in the descriptors includes a pointer to the location in memory of the memory object, its privilege level, an indicator of whether the descriptor is globally or locally accessible, and a segment present bit. A task gate descriptor points to a TSS. To initiate the hardware task switch mechanism the program simply includes a jump instruction with the offset of the task gate descriptor in the GDT as the operand to the jump instruction. For example the single assembler instruction `ljmp 0x20,0` when executed by the RTK low-level dispatcher causes the hardware task switch mechanism to perform a task switch from the RTK to the IBM MicroKernel.

Instead, the implementers of the MicroKernel chose to use their own mechanism to do task switching in software. The basic data structure involved in an i486 task switch is the TSS. Although the IBM MicroKernel does not use the i486 hardware task switch mechanism, it does have to provide a TSS for every task, because the i486 obtains the address for a kernel stack from the TSS any time a user level task is interrupted. Moreover, the TSS contains a bit mask for I/O ports which the i486 uses to determine if a particular task has permission to access an I/O port. Since the IBM MicroKernel had already set a TSS for each task and for itself, it required only a few changes to initialize a number of fields in the TSSs for the SP implementation. The implementation's use of the TSS switch tasks required that locks be placed around a few lines of code within the IBM MicroKernel. These lines of code set the task register and the page directory base register (PDBR) in the i486. The task register points to the TSS of the currently executing task and the PDBR holds the physical address of the page table directory for the currently executing task; together they constitute the hardware's view of the executing task. If the SP interrupt handler executed while this view is inconsistent, the state of the MicroKernel task saved in a TSS might be inconsistent; when the SP dispatcher switched back to that TSS, the IBM MicroKernel would crash. An interesting point, showing an unexpected advantage of the SP architecture's separation of real-time and non-real-time threads, is worth noting here. In most cases when the IBM MicroKernel crashes the RTK does not. In fact, even when the IBM MicroKernel intentionally traps to the debugger the RTK continues to execute unabated. This is because the trap handler for the MicroKernel simply executes and transfers control to the debugger. The CPU executive continues to switch between minor cycles. The debugger executes in the non-real-time minor cycles and the RTK and its tasks execute in the real-time minor cycles.

5.3 Description of the Software Modules

This implementation of the SP architecture consists of an interrupt enable/disable virtualization component, a real-time kernel (RTK), a CPU executive, a network adapter device executive, a speaker executive for the built-in audio output device, and a display executive for a VGA display. The real-time kernel comprises a real-time thread scheduler (earliest deadline first), an admission control function, an RTK *personality-neutral server*, and an RTK real-time thread library component.

The name, *personality-neutral server*, has its roots in the architecture of the MicroKernel which is derived from the Mach Operating System. In these systems personality neutral refers to services that are independent of the particular operating system server that executes above the MicroKernel. These services provide the operating system interface to user processes of that particular operating system. The CPU executive includes an interrupt handler as well as logic that manages how the CPU is shared between the GPOS and the RTK. Figure 5.2 shows the relationship among the major software modules for this implementation of the SP architecture. The following sections describe each of the major components in Figure 5.2 in greater detail.

5.3.1 The Interrupt Enable/Disable Virtualization Component

The first and most important function required by any implementation of the SP architecture is a predictable, periodic interrupt vectored to the CPU executive

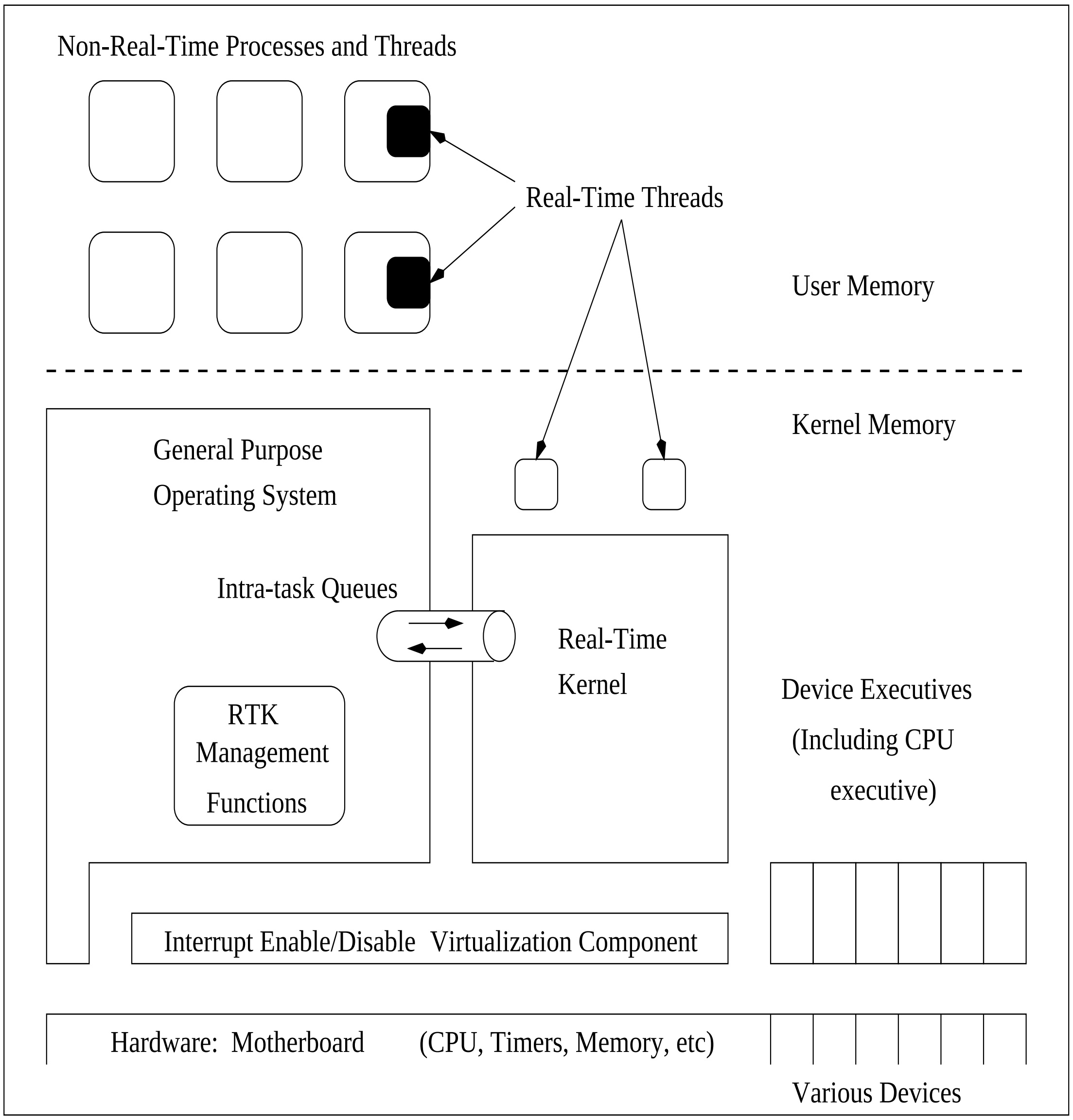

Figure 5.2: Relationship of the Major Software Modules

interrupt handler. The CPU executive interrupt handler is the first code that executes in response to the interrupt that marks the start of a minor cycle or slot, if slots are necessary. In this particular implementation, the source is the RTClock. The RTK and the GPOS guarantee that the interrupts will execute the CPU-executive interrupt handler with a bounded (and small) latency after the exact moment of the interrupt. The RTK real-time thread scheduler uses this bound on the latency as a constant when calculating feasibility. In this implementation, empirical measures show that the latency does not exceed 330 microseconds. This implementation guarantees interrupt latency by disallowing the IBM MicroKernel access to the interrupt bit (the IF bit) in the CPU flags register. This bit determines whether the CPU responds to the INT line from the set of PICs.

All requests from the IBM MicroKernel to change the state of the IF bit are handled by the SP virtualized interrupt enable/disable component (IEC). The IEC ignores IBM MicroKernel requests to disable interrupts. That is, it leaves the IF bit in the CPU flags register set, but sets the interrupt masks in the PICs so that only the interrupt from RTClock used by SP can get through to the CPU. Since only SP uses the RTClock, the IBM MicroKernel can proceed as normal, since none of the interrupts that affect its operation will be allowed. By never disabling the RTClock interrupt the CPU executive interrupt handler will be executed at each slot boundary. When the IBM MicroKernel requests that interrupts be enabled, the IEC sets the interrupt masks in the PICs so that all normal MicroKernel interrupts (plus the RTClock interrupt) are enabled. The overall effect is to isolate the IBM

MicroKernel from the hardware, while SP still has access to a periodic timer.

The IEC implementation required that the IF bit be cleared (i.e., that all interrupts be disabled) to enforce mutual exclusion within the IEC itself. These critical sections are approximately 200 CPU cycles at most. On the hardware used, each cycle is 20 nanoseconds. This means that the RTK itself disables the IF bit for no more than 4 microseconds, which can delay the interrupt from the RTClock. There are also critical sections between the GPOS and the RTK. These two types of critical sections contribute to the variability in slot lengths.

Note that in Figure 5.2, part of the GPOS extends around the IEC down to the hardware. This illustrates the critical sections in the GPOS that have to be serialized with the RTK. To do this, the GPOS must call the hardware interrupt enable/disable mechanism directly.

5.3.2 The CPU Executive

The CPU executive includes a function to manage the lengths of the minor cycles and an interrupt handler that is the first code executed in response to an interrupt from the RTClock that begins each slot. The values of mc_{rt} and mc_{nrt} are set by the admission control function of the RTK in multiples of the slot length. The CPU executive implements minor cycles by counting slots and switching between the GPOS and the RTK when appropriate. When the RTClock interrupts, the interrupt handler executes the following logic:

- If the previous slot and the next slot are both in a non-real-time minor cycle, the interrupt handler signals end-of-interrupt and immediately returns to the GPOS.
- If the previous slot was in a non-real-time minor cycle and the next slot is in a real-time minor cycle, the interrupt handler switches to the RTK and indicates that this call begins a new real-time minor cycle.
- If the previous slot and the next slot are both real-time minor cycles, the interrupt handler switches to the RTK and indicates that the RTK is within a real-time minor cycle.
- If the previous slot was in a real-time minor cycle and the next slot is in a non-real-time minor cycle, the interrupt handler switches back to the GPOS.

Slots are artifacts of the hardware used in this implementation of the SP architecture. The preferred implementation would generate interrupts at minor cycle boundaries. However, the Mod-95 lacks the hardware necessary for generating precise interrupts that have such widely varying values[4]—anywhere from a few microseconds to hundreds of milliseconds—and that can be reprogrammed after each interrupt. The RTClock can actually generate interrupts in the appropriate range. However, attempts failed to get it to work correctly, as determined by a relatively large variability in minor cycle lengths, while reprogramming it after each interrupt, Thus we chose to program the RTClock with a single value and

[4]The values should be able to accommodate reasonable, possible lengths of minor cycles.

create the minor cycles out of these fixed intervals (slots).

The details of how the interrupt handler is started are interesting. At the location reserved for the RTClock interrupt, the interrupt vector table has a descriptor called a CPU task descriptor. It causes the CPU to invoke special functions when it is accessed and includes a pointer to a CPU Task State Segment (TSS). When the CPU detects this type of descriptor during an interrupt, the complete state of the currently executing process is saved in the task's TSS. The task state from the TSS pointed to by the descriptor is loaded into the CPU, and execution begins at the instruction pointer for the just loaded task. Even though this type of interrupt costs more than others, it is implemented this way to minimize changes to the GPOS and to provide complete separation between the GPOS and the RTK.

This implementation of a CPU executive also required modification of the first level interrupt handler of the IBM MicroKernel. When a hardware interrupt occurs, the IEC is called, which sets the interrupt masks in the PICs and IF bit in the CPU flags register so that the RTClock can interrupt the CPU. This allows the CPU executive interrupt handler to execute within the context of the device interrupt handlers in the IBM MicroKernel. If a particular interrupt from the RTClock occurs during the execution of a MicroKernel device interrupt handler and the CPU executive should start a real-time minor cycle, it does so and the real-time minor cycle is maintained for its full length, delaying the execution of the device interrupt handler. We have found that this design does not cause non-recoverable failures on devices such as disks, although it may contribute to the general overhead imposed on the MicroKernel by the implementation, because the MicroKernel interrupt handlers and their associated device are not interacting optimally.

5.3.3 The Real-Time Kernel

The real-time kernel is an umbrella term for functions required by the real-time threads:

- scheduling the set of real-time threads
- managing the attributes, such as the periods, of the real-time threads
- providing services, such as a library of video display calls, to the real-time threads

The RTK is logically equivalent to the GPOS for its functions and is thus shown in Figure 5.2 next to the GPOS. It is smaller than the GPOS because it includes less function. The SP architecture was designed so that the type of computation and I/O that a non-real-time task could likely require of a periodic real-time thread was constrained and well-defined. Therefore, the function that a real-time thread should require of the RTK would be much less than that required of the GPOS by non-real-time threads.

The RTK in this implementation of the SP architecture employs an earliest-deadline-first scheduling algorithm. In theory, an EDF scheduler begins executing a new real-time thread at each instant when a real-time thread with a deadline closer than the currently executing thread becomes ready to run. However, in practice the scheduler will be able to make these scheduling decisions only at defined times. In this implementation of the SP architecture, these scheduling points occur whenever the RTK executes, i.e., at the start of each real-time minor cycle and at the start of each slot within a real-time minor cycle. Thus, at these points the RTK makes a scheduling decision as to which real-time thread should execute next (when this execution of the RTK is finished). Because the scheduler is EDF, the next thread will be the real-time thread with the closest deadline. In the case of a tie, the first thread in the RTK ready queue is chosen. In addition to the scheduling decision, the RTK collects accounting data on the current set of real-time threads and itself each time it executes.

The RTK uses the hardware task switch mechanism to dispatch the real-time threads. The RTK switches to a real-time thread by a long jump to a global descriptor table entry that contains a task descriptor for the real-time thread.

Note that some of the RTK is implemented in the GPOS. The RTK is actually just another real-time thread, and thus can communicate, in a non-real-time mode, with the GPOS. The admission control function (feasibility analysis) of the RTK is in the GPOS. Since the new real-time thread need not be accepted in real time and the feasibility analysis can be lengthy, it is more properly implemented in the GPOS.

5.3.3.1 The Real-Time Thread Scheduler

The scheduling policy used in this implementation is EDF, in which the scheduler makes a scheduling decision at two points: when the RTK executes at the start of each slot and any time that a real-time thread calls its `WaitForPeriodExpiration` function. The granularity of the scheduler is thus equal to the length of a slot. For many of the experiments and application demonstrations, the slot length was just under 1 millisecond. Note that in practice, the feasibility analysis will have to add to each real-time thread's cost the cost of executing the RTK.

5.3.3.2 The RTK Admission Control Function

Any real-time system must restrict the activation of real-time threads because if demand for "real-time" CPU cycles is excessive, some real-time threads will miss their deadlines. Admission control is an implementation of the feasibility analysis associated with the particular scheduling policy currently in use. This implementation uses the feasibility analysis presented in Chapter 4. When a new real-time thread requests admission into the real-time ready queue, the admission control function first determines if the current values of the minor cycle lengths are appropriate. For example, if the current values are $mc_{rt} = 2$ ms and $mc_{nrt} =$ 200 ms and the requesting thread has a period of 50 ms, the current values of minor cycles lengths are clearly inappropriate and the feasibility analysis is certain to fail. In these cases the admission control function calculates appropriate values for mc_{rt} and mc_{nrt} and then determines feasibility of all real-time threads with the cost and period of the requesting real-time thread included.

If the minor cycles have appropriate lengths, the admission control function also computes a feasibility analysis that includes the cost and period of the requesting real-time thread, but uses the current values of the minor cycle lengths. If the set of real-time threads, including the new real-time thread, is feasible the requesting thread is accepted. If the new set of threads is not feasible, the admission control function computes new values of the minor cycle lengths and performs the feasibility analysis again. If, after two tries, feasibility is not obtained the request is rejected.

5.3.3.3 The RTK Real-Time Thread Services Library

The real-time threads managed by the RTK will require services to perform real-time I/O; communicate with the non-real-time portion of their creating task; communicate with the GPOS for non-real-time access to I/O devices; and carry out other services, such as blocking and timing calls, provided by the RTK itself.

5.3.3.4 The RTK Personality Neutral Server

In this implementation, the non-real-time task that creates a real-time thread does so with a call to `CreateRTThread`, which is implemented in the RTK real-time thread services library. As part of completing the `CreateRTThread` call, the services library calls the personality neutral server function of the RTK for function provided by the MicroKernel. This function is the ability to copy the text of a real-time thread from user memory space into kernel memory space, because in this implementation, all real-time threads execute in kernel memory space.

Having the real-time threads execute in kernel memory space is a convenience, not a necessity. There is fundamentally no reason why the RTK could not transfer control to code that is logically in user space. As the personality neutral server transfers code, which will become a real-time thread, from user space to GPOS kernel space, it communicates a number of parameters to the RTK: the logical address of the thread's start, the period of the thread, and the cost in CPU time required by the thread during one period. The RTK real-time thread scheduler uses these values to determine when to execute the thread and, by the RTK real-time thread services component to set up a TSS for this thread. Each real-time thread is written as a C function that takes two parameters. These parameters are pointers to structures that contain pointers to RTK exported functions and data. The real-time threads in this implementation can access a real-time screen output library and RTK administrative data. The RTK real-time thread services component sets up the stack provided to the real-time thread with these parameters before the real-time thread's first execution. The current implementation has no protection and real-time threads can corrupt the GPOS kernel, since the pointers give them unrestricted access to the GPOS kernel's logical address space.

5.3.4 Other Device Executives

Three other device executives are used on the hardware platform in this implementation. They are all described in greater detail in Chapter 6 but are mentioned here for completeness.

- The network adapter falls into the non-preemptible, externally triggered, stochastic process resource class of the SP architecture resource model.
- The speaker device falls into the partitionable resource class of the SP architecture resource model.
- The display device falls into the partitionable resource class of the SP architecture resource model.

5.4 Experiments

This section presents the description and results of some quantitative experiments involving an implementation of the SP architecture. The primary purpose of the experiments is to validate the implementation by determining whether it realizes the two fundamental requirements originally presented in Section 3.3:

Requirement $\mathcal{A}$ If a system interleaves the execution of real-time and non-real-time threads in alternate intervals and the intervals in which real-time threads execute are scheduled to begin every l time units, then it must ensure that the intervals begin at times t where $kl \leq t \leq kl + \epsilon$, $\forall k \geq 0$.

Requirement $\mathcal{B}$ For $L > \epsilon$ (for a suitable ϵ) for which the real-time thread scheduler has assigned a real-time thread, τ, to be executed on the CPU, there must be a function or method by which the minimum number of CPU cycles available to execute the instructions of τ can be determined.

Two sets of experiments were conducted. To satisfy requirement $\mathcal{A}$, this implementation must ensure that the real-time minor cycles begin with bounded latency after their scheduled start time. This is shown by measuring the lengths of consecutive slots. If there were zero variability in the slot initiation times, all of the measured lengths would be identical. If the start of a slot were delayed, it would be longer and the subsequent slot would (likely) be shorter.

To satisfy requirement $\mathcal{B}$, there must be a function or method for determining the minimum number of CPU cycles available to execute the instructions of the real-time thread. In other words, the elapsed execution time of an instance of a particular periodic real-time thread (given that any conditional paths through the code of the thread are considered) varies only slightly under all conditions of system loading. This second notion of requirement $\mathcal{B}$ was used to design the experiments to test requirement $\mathcal{B}$.

5.4.1 Requirement $\mathcal{A}$ Experiments

If a system to support the execution of real-time threads correctly interleaves the execution of real-time and non-real-time tasks, then the intervals in which real-time threads execute must begin at times t where $kl \leq t \leq kl + \epsilon \, \forall k \geq 0$. Our SP architecture implementation does correctly interleave the execution of real-time and non-real-time tasks. The minor cycles are defined in terms of slots delineated by interrupts from the RTClock. How long the slots last is defined by RTClock programming. It is assumed that the RTClock generates interrupts precisely at the times necessary to create the slot interval. A test will show if the CPU and low-level interrupt handler code handles the interrupt from the RTClock with a maximum upper bound on latency. The results of the experiments indicate that there appears to be an upper bound on the maximum delay for an interrupt from RTClock of about 300μs, the longest slot length deviation measured. This delay is due to the critical sections in the GPOS that must be serialized with the RTK by the real interrupt enable/disable mechanism, i.e., not the IEC. Although for some of these experiments the GPOS was forced to idle by being held in the kernel debugger, some kernel threads were still executing. The debugger thread was generally blocked on input from the debugger terminal, so the kernel was executing its idle loop. The GPOS kernel timer was still generating interrupts, so the GPOS scheduler ran at each interrupt to detect any threads ready to run, etc. Also, the critical section around some of the code that resets the GPOS timer caused some of the variance.

5.4.1.1 Slot Length Variation

Three experiments determined if the implementation met requirement $\mathcal{A}$. The first experiment measured the lengths of slots of over 10 million RTClock interrupts when the GPOS was essentially inactive[5]. The purpose was to determine the maximum variation in the length of a slot under ideal circumstances. The expected slot length was 976.562 microseconds (1,165 ticks of a timer at a rate of 1 tick every

[5]The GPOS was held in a trapped state with the kernel debugger, and the network adapter connection to the token ring was removed.

838.223 nanoseconds). As shown in Figure 5.3, the distribution of measured slot lengths was from 1,134 to 1,169 timer ticks (939.2 μs - 979.88 μs), primarily in the interval 1,154-1,157 (967.31 μs-969.82 μs). A constant shift of approximately 10 μs less than the expected value was found in all all of the experiments on the system. We conjecture that the RTClock is being clocked by the motherboard timing signal of 1.193MHz instead of the designed input for the RTClock of 1.048576MHz. This would cause the RTClock to produce intervals about 1.1 percent shorter than design. Our measurements agree with this discrepancy, but the issue is still under investigation.

Figure 5.4, another view of the data presented in Figure 5.3, illustrates that 99.6 percent of all measured slot lengths were within 8 μs of the expected value. Given an assumed clock frequency error of about 8 μs, it can be said that 99.6 percent of all measured slots should be within 0 μs of the expected value given the programming of the RTClock (that is, the RTClock should generate an interrupt every 976.562 microseconds).

The second experiment is identical to the first, except that the GPOS is now forced to be very active by building a new version of the MicroKernel, specifically by compiling a kernel source file and linking the resulting object file with all other object files of the GPOS kernel. The procedure for doing this requires a number of different processes to be active and imposes a significant I/O and computational load on the system. The result is a greater variation in slot length, as illustrated by Figures 5.5 and 5.6.

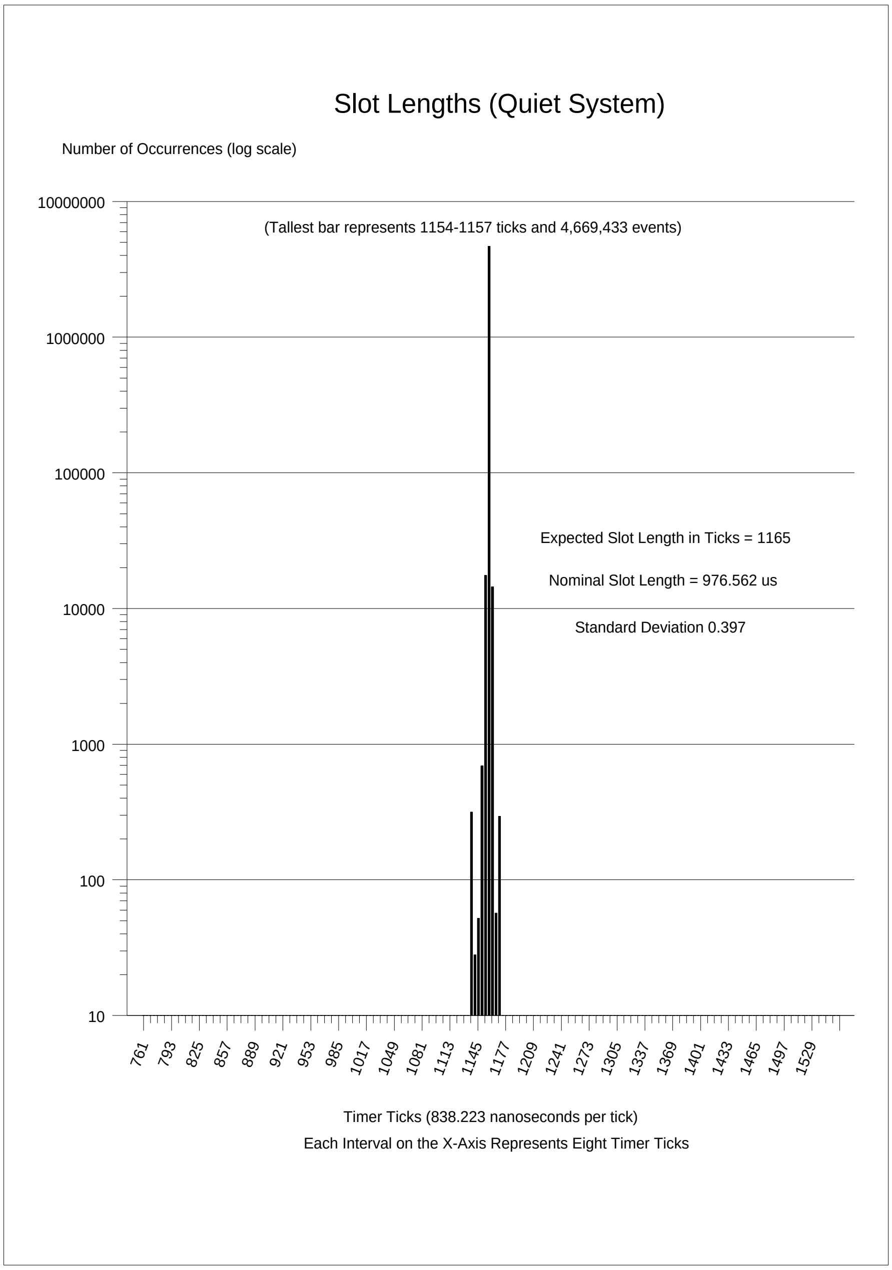

Figure 5.3: Slot Lengths (Quiet System)

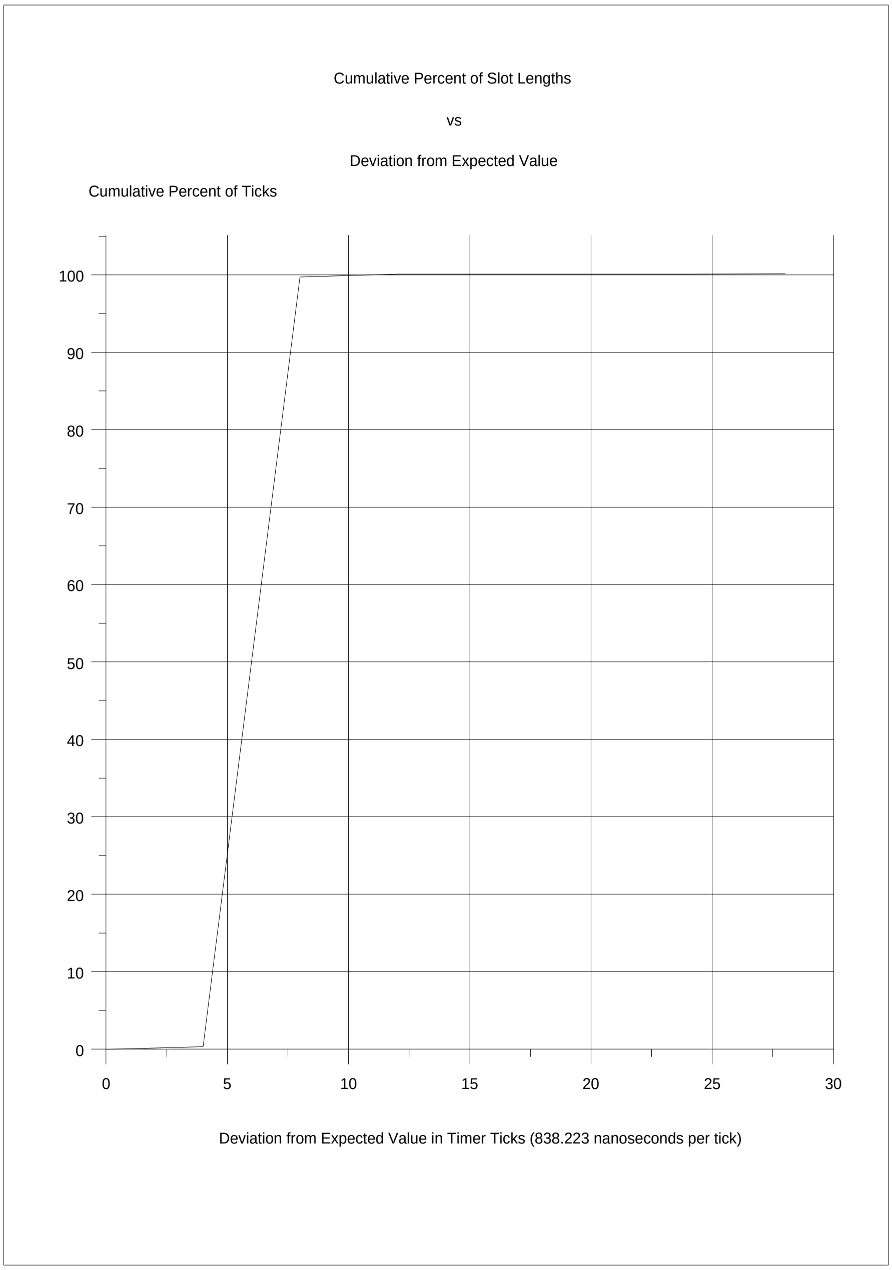

Figure 5.4: Cumulative Percent of Slots From Nominal (Quiet System)

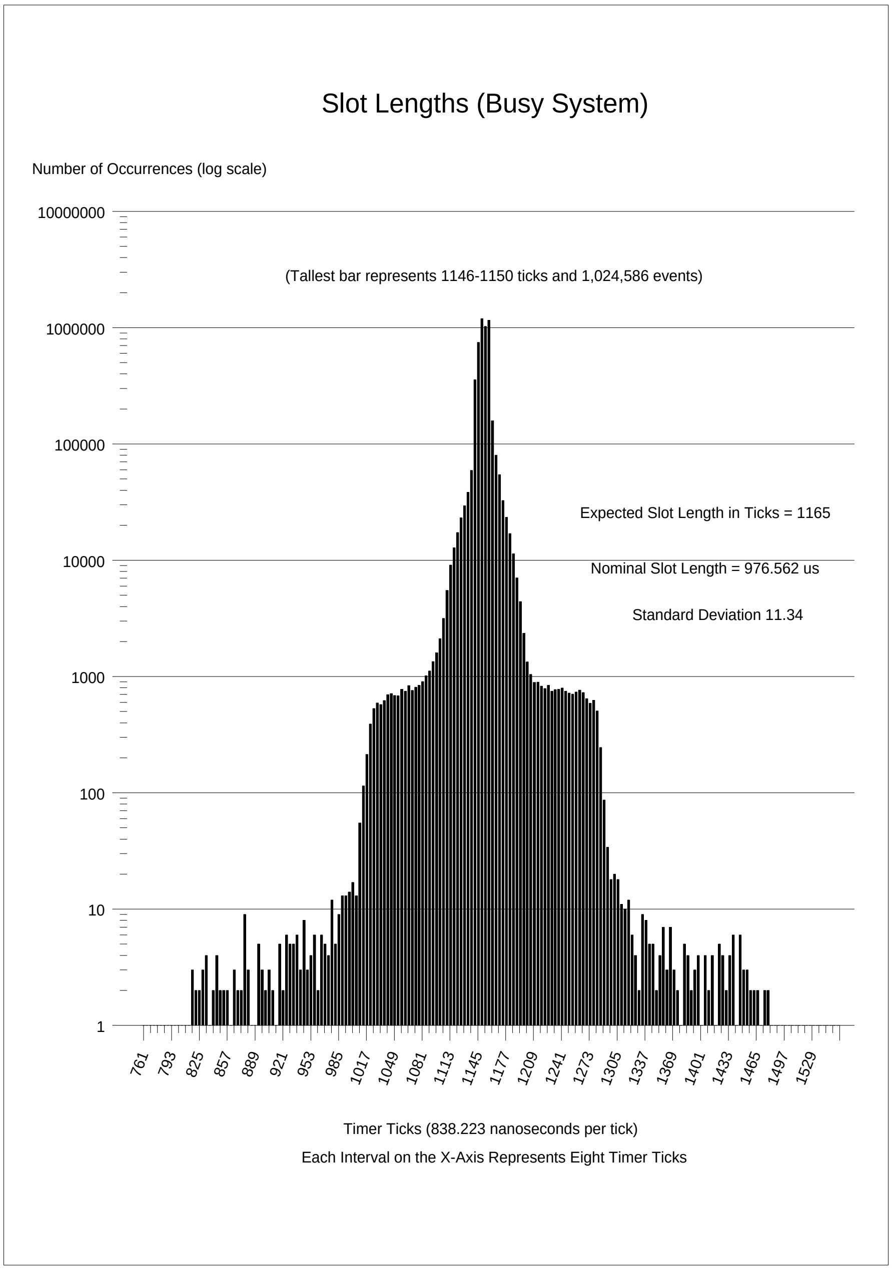

Figure 5.5: Slot Lengths (Busy System)

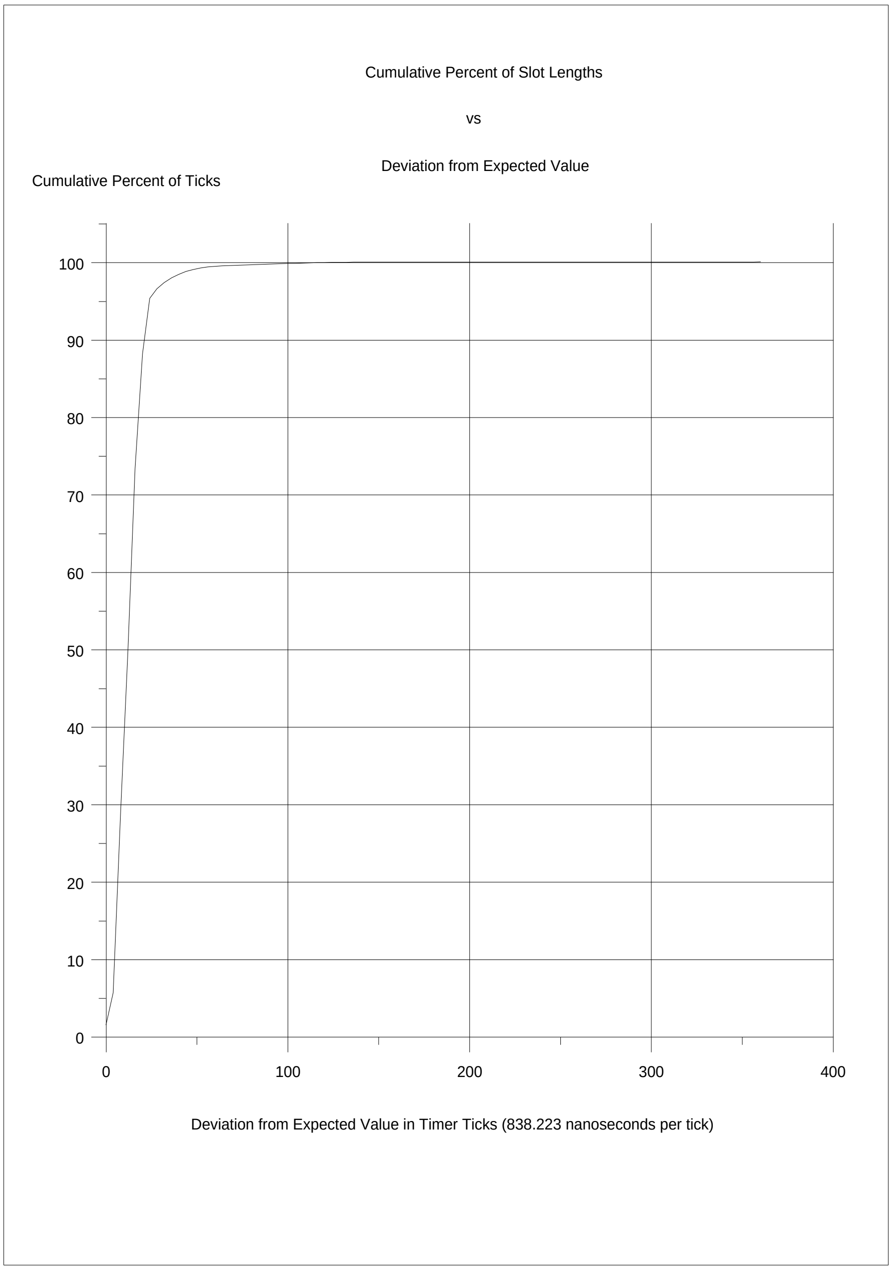

Figure 5.6: Cumulative Percent of Slots From Nominal (Busy System)

Slot lengths were measured from 802 to 1,485 timer ticks (672.25 μs - 1,244.76 μs) a deviation from the expected value of -304 μs to +268 μs. This variation occurs because the critical sections in the GPOS that are serialized by calls to the real interrupt enable/disable mechanism (not the virtualized one) take much longer to execute and because the GPOS is performing DMA between the disk and memory and DMA has priority for use of the bus. The cache on the machine has sixteen byte lines and the longest protected critical section is 523 bytes. This requires 33 main memory accesses to load the code of the largest critical section in the worst case. The processor may be locked out from the bus for up to 7.8 μs. Including the time required for memory accesses, this means the critical section may take 346 μs to load into memory. This time is calculated as follows:

- 33 memory accesses times 7.8 μs wait time for each access equals 257 μs.
- Compute the memory access time for each cache line: 400 ns (worst case memory access time) times 4 words (16 bytes per cache line) plus 300 ns to access the system bus transfer equals 1.9 μs to load each cache line.
- 1.9 μs times 33 cache lines is 63 μs.
- 63 μs plus 257 μs is 320 μs.

Note that this time does not include time to execute the instructions of the critical section. However, this execute time would overlap with the memory access time and the bus waiting time, so an accurate value would be hard to determine.

The delays in this experiment correlate well with this analysis. Figure 5.6 presents the data of the second experiment differently, illustrating that 99.8 percent of the slot lengths were within 77 μs of the value one would expect, given the programming of the RTClock (i.e., the RTClock should generate an interrupt every 976.562 microseconds).

The third experiment measured major cycle lengths in a busy system with a load generated as in experiment two. The real-time minor cycle length is 2 slots and the non-real-time minor cycle length is 8 slots, which means the major cycle length is 10 slots or 9.76 milliseconds. The postulated clock crystal frequency problem mentioned earlier in this section appears in these measurements, also, most of the cycle lengths were about 70 μs (83 ticks), less than the expected values would indicate. Figure 5.7 shows a histogram of the measured major cycle lengths that compares well with the histogram of slot lengths in a busy system. Figure 5.8 presents the data of the third experiment differently and shows that 99.8 percent of the major cycle lengths were within 67.1 μs (80 timer ticks) of the the value one would expect given the programming of the RTClock; that is, the RTClock should generate an interrupt every 976.562 microseconds, and the number of slots in a major cycle.

5.4.2 Requirement $\mathcal{B}$ Experiments

The next set of experiments proved that the implementation satisfies requirement $\mathcal{B}$. This implementation uses time to measure the progress of the real-time threads

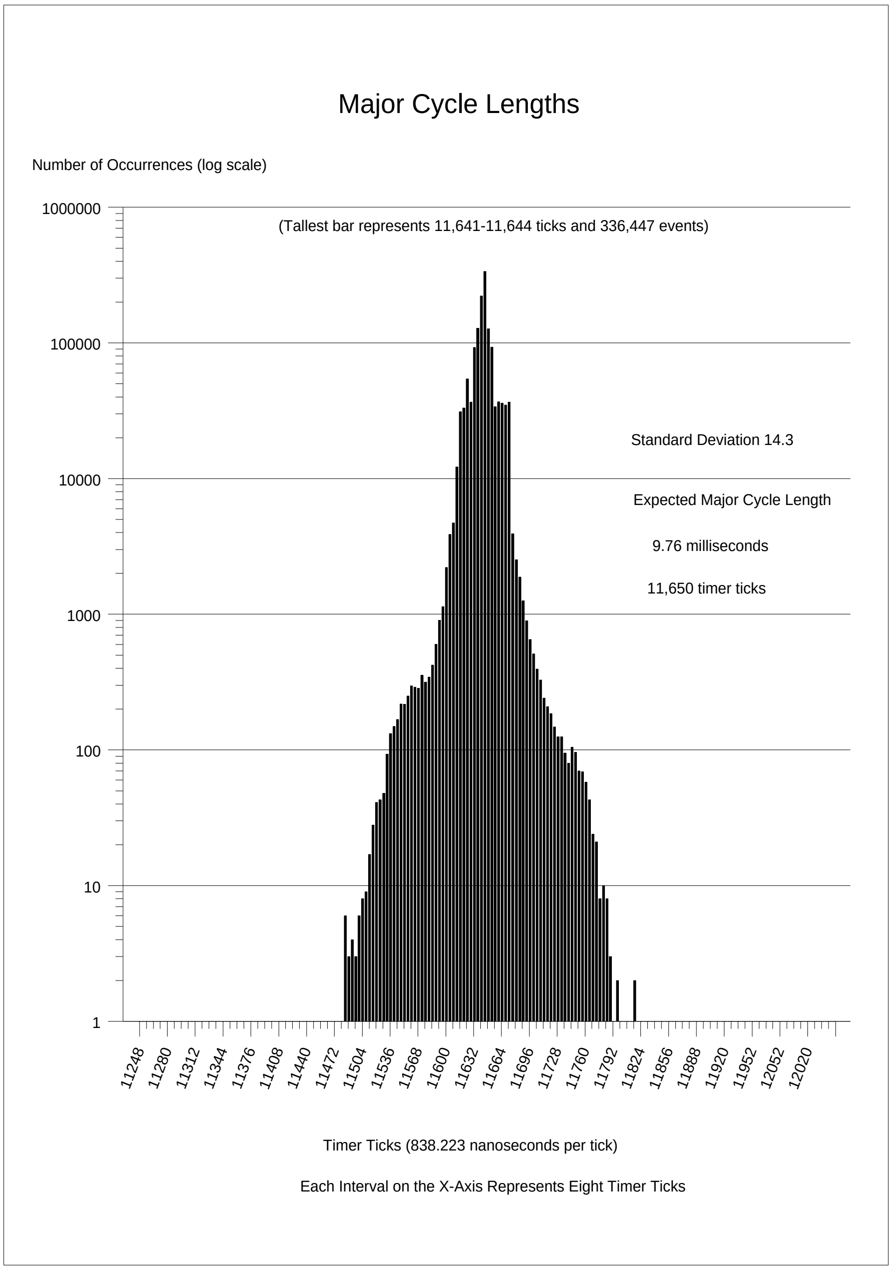

Figure 5.7: Major Cycle Lengths (Busy System)

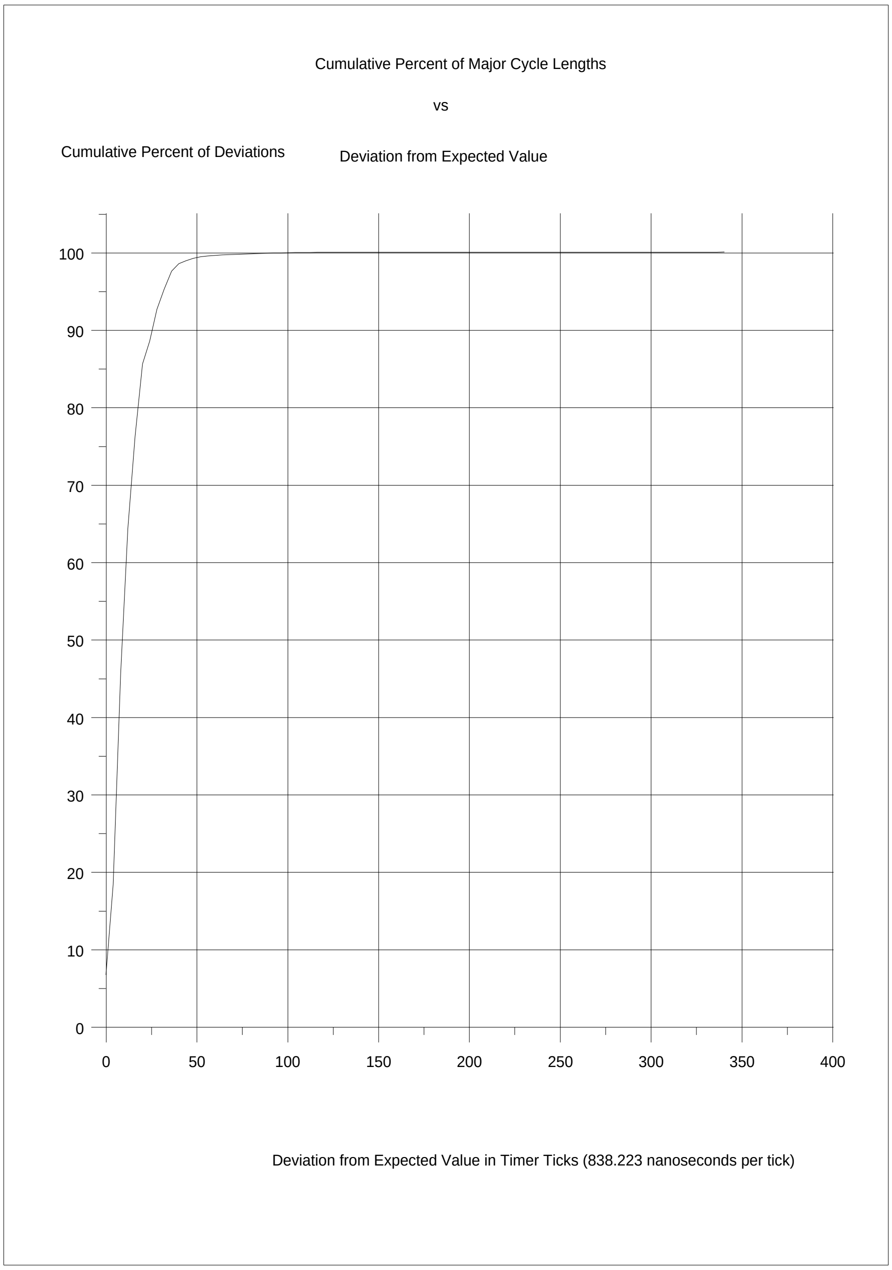

Figure 5.8: Cumulative Percent of Major Cycle Lengths From Nominal (Busy System)

and therefore measures intervals as time rather than CPU cycles. The experiment uses a real-time thread that executes a tight loop for a specified number of iterations, which determines the number of CPU cycles the loop consumes. If the function between elapsed time and CPU cycles consumed were linear, i.e., if a constant number of CPU cycles were available for real-time thread execution in the interval in which the loop executed, then the elapsed time from the start of the loop until the specified number of iterations were complete would be constant for all executions of the loop.

The effect of the instruction cache is constant in these experiments. This is reasonable, since the real-time minor cycles have no interrupts. The effect of the instruction cache will be the same at the start of each real-time minor cycle for each experiment because the test loop is small enough to fit in the cache and the real-time minor cycles are relatively far apart (so the cache will be cold in certain experiments). In the experiments where the GPOS is not executing, the cache will likely contain the test loop instructions at the start of each real-time minor cycle. In the experiments where the GPOS is executing, the cache will likely not contain the instructions of the test loop at the start of each real-time minor cycle. If the implementation satisfies requirement $\mathcal{B}$, then the time needed to execute the loop should remain fairly constant.

The following experiments all run the programmed loop 1000 times and record the elapsed time used for each run. The graphs plot the measured time (y-axis) against the run number (x-axis). The granularity of the timer is again 838.222

nanoseconds. In each pair of graphs the first (Execution Time) shows the execution time of the loop and the second (Completion Time) shows the execution of the RTK and the test loop. The logic performed by the RTK depends on whether the slot marks the beginning of a real-time minor cycle or a non-real-time minor cycle, or is within either a real-time or a non-real-time minor cycle (see Section 5.3.3).

Table 5.1 shows the values of the minor cycles used in all of the requirement $\mathcal{B}$ experiments.

5.4.2.1 Baseline Experiment

The baseline experiment attempts to eliminate all sources of variability not controlled by the SP implementation to determine how small the variability in real-time thread execution time can be. Additionally, as the sources of variability are progressively added, the additional variability is clearly seen in the results. For the baseline case, the loop is set for a number of iterations that will make it execute in less time than the length of a slot plus the time the RTK executes at the start of each slot (which removes the inherent variability in the RTK), the GPOS is held

Cycle	MC	mc_{rt}	mc_{nrt}
Slots	20	5	15
Milliseconds	19.531	4.883	14.648

Table 5.1: Minor Cycle Lengths used in Requirement $\mathcal{B}$ Experiments

almost[6] completely quiet by using the debugger, and the loop has no instructions that use any I/O device.

Figure 5.9 illustrates the interval in which the programmed test loop executes, where the RTK executes relative to where the loop executes, the slot boundaries, and starting and stopping points of the measured intervals of execution time and completion time.

Graph 5.10 shows the 1,000 points of measured execution time (see Figure 5.9) for the baseline case. The apparent horizontal lines are actually a sequence of points (each point is marked with a +). Note that the lines are just a little less than 1 microsecond (actually 838.222 nanoseconds) apart. This is the resolution of the timer used to measure the execution and completion interval lengths. The difference between the longest and shortest measured execution times is 2.5 microseconds or 3 timer ticks. This variability, about 0.36 percent, probably comes from the unaddressed variability in the MicroChannel bus, such as use of the bus for RAM refresh.

Graph 5.11 shows the 1,000 points of measured completion time(see Figure 5.9). Here the difference between the longest and shortest measured completion times is 3.3 microseconds or about 4.6 percent. The additional variability comes from the execution of the RTK: the RTK must perform a number of tasks, such as deciding on the type of the current slot, deciding which real-time thread to execute

[6]Note that interrupts are enabled while in the debugger; the debugger is simply a module in the kernel, since the debugger code has to respond to keyboard input from the auxiliary terminal via the serial port. Thus with interrupts enabled, the GPOS device drivers are free to respond to any requests from devices.

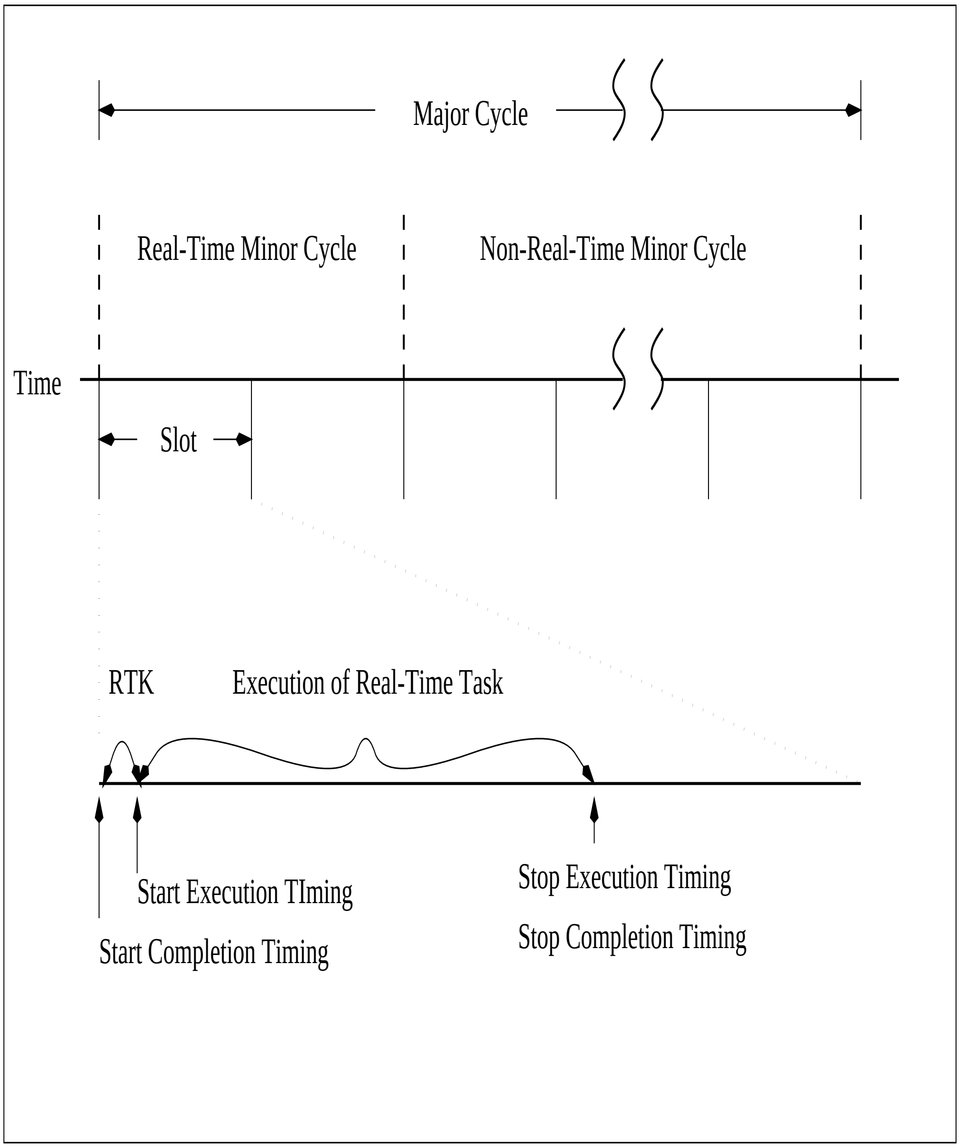

Figure 5.9: Baseline Experiment

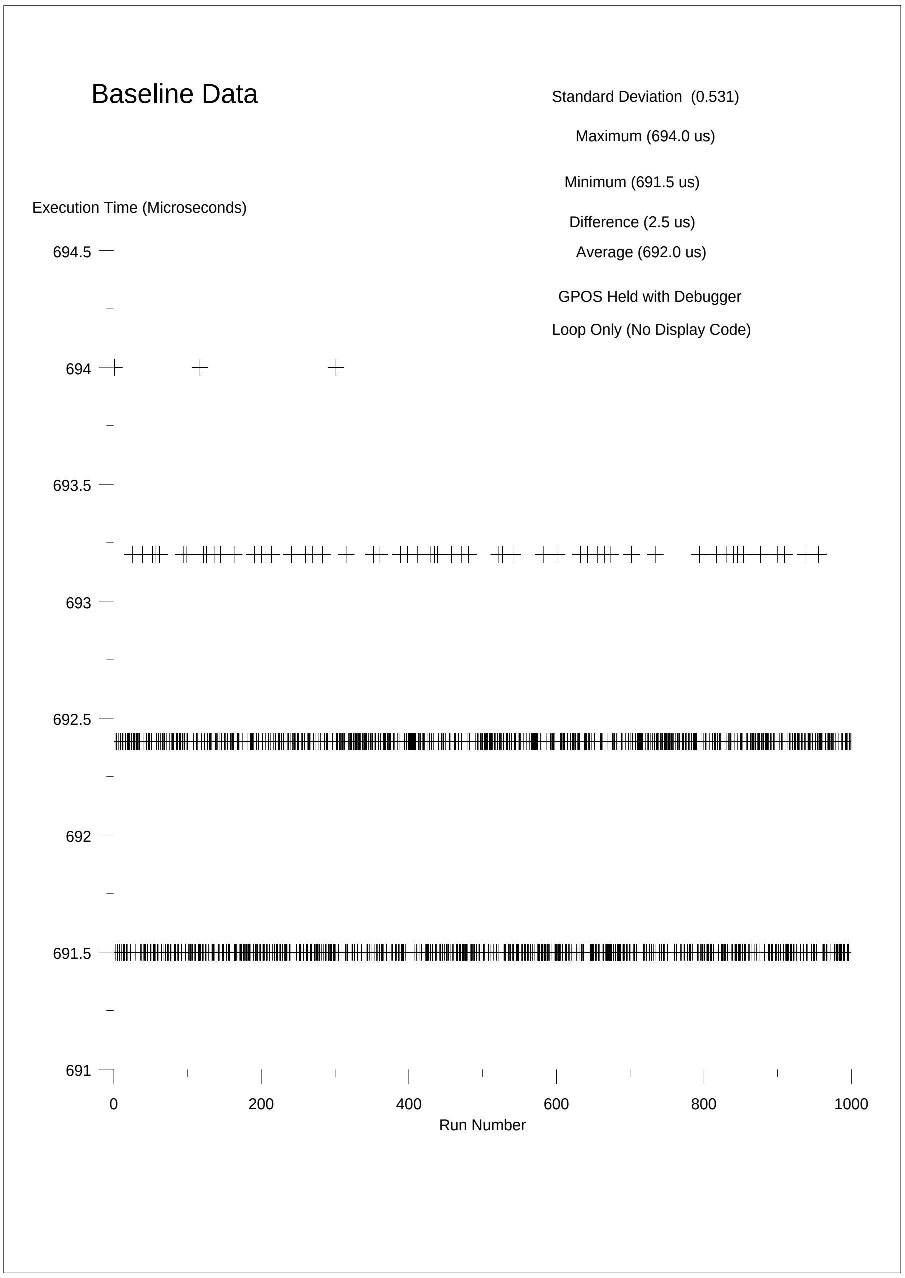

Figure 5.10: Variation in Execution Times for the Baseline Experiment

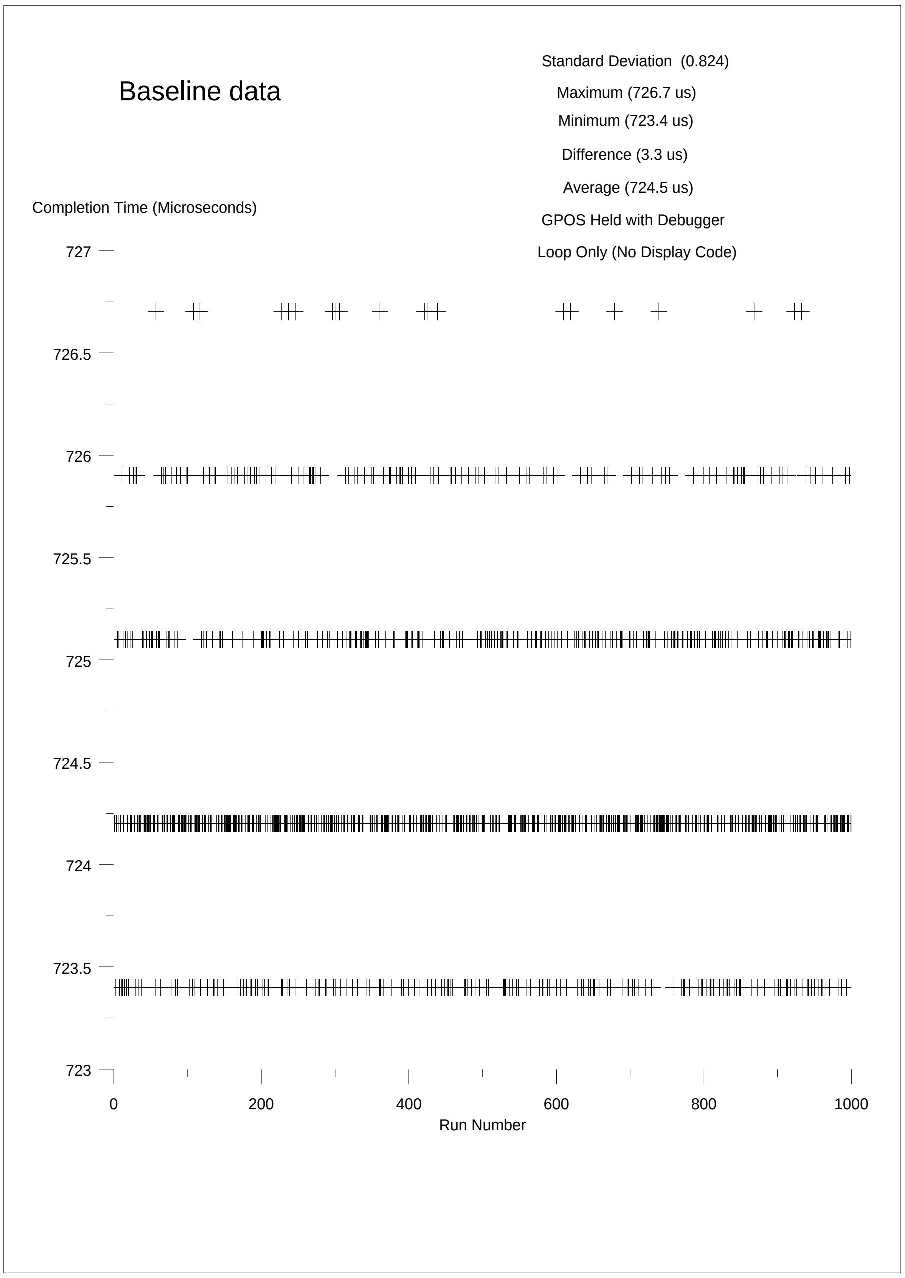

Figure 5.11: Variation in Completion Times for the Baseline Experiment

if this slot is in a real-time minor cycle, and accumulating various administrative statistics. For any implementation of the SP architecture, this variability will have an upper bound which must be determined experimentally. The upper bound on RTK execution time was found to be 90 μs by subtracting the shortest execution time from the longest completion time of the experiment with the greatest variability, which was the last experiment that measured execution and completion times when the test loop executed in two subsequent real-time minor cycles.

5.4.2.2 Loop Execution Time Variation with Calls to the Display Library

In this set of experiments, the only change from baseline is that at the end of each run the test loop now writes characters to a display device, the console, using memory mapped IO. The test loop simply writes a byte of data to a particular memory location. However, since there is a real physical device on the receiving end of the system bus, one would expect somewhat more variability. As in the baseline case, the time needed to execute the loop is less than a slot and the GPOS is still held with the debugger. Although the differences between the longest and shortest measured execution and completion times are about the same as the baseline, Graphs 5.12 and 5.13 show that considerably more of the runs had longer measured times. That is, variation was more common: the standard deviation (SD) in the execution times of the baseline experiment is 0.531, compared to an SD for this experiment of 0.574.

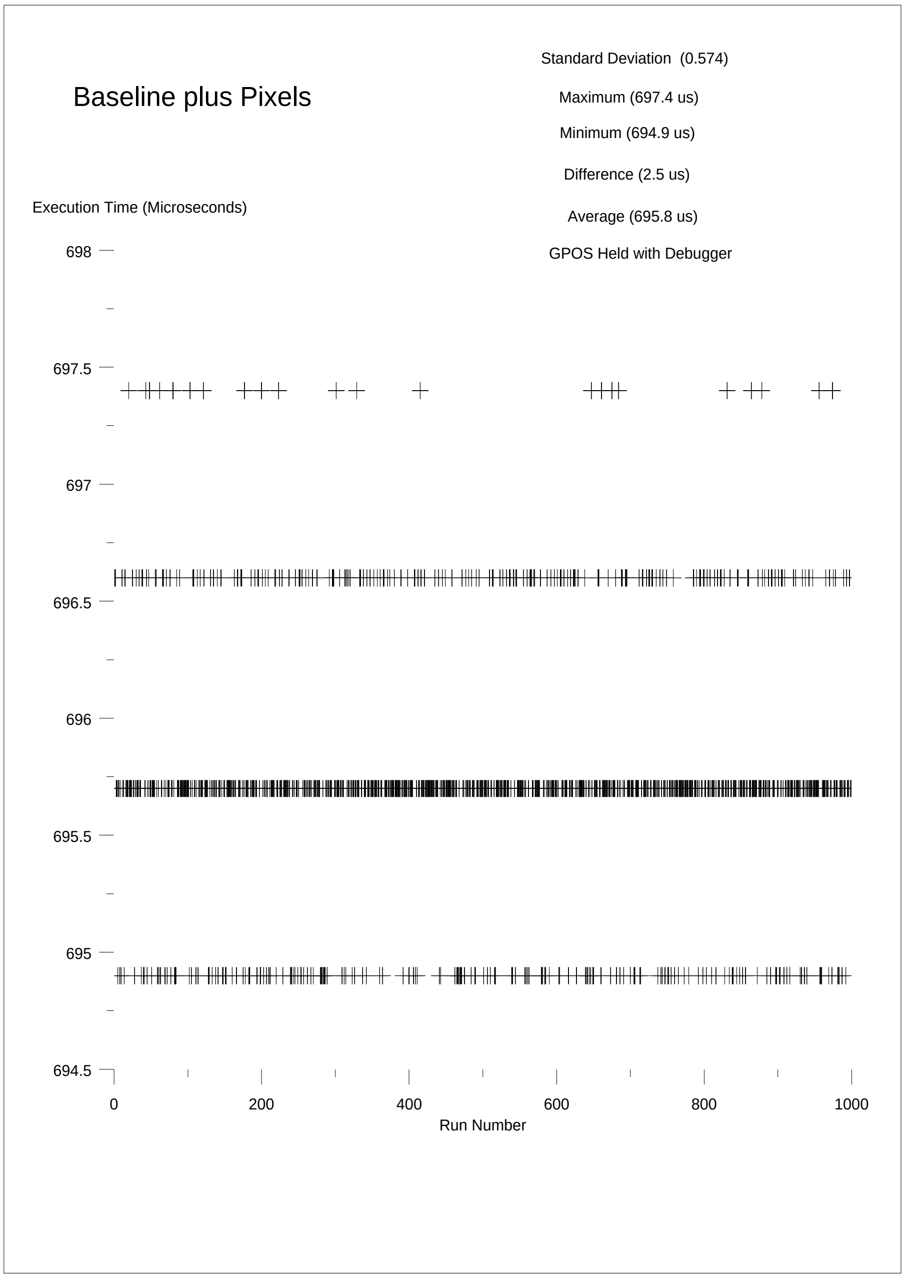

Figure 5.12: Variation in Execution Times when the Loop Uses the Display Device

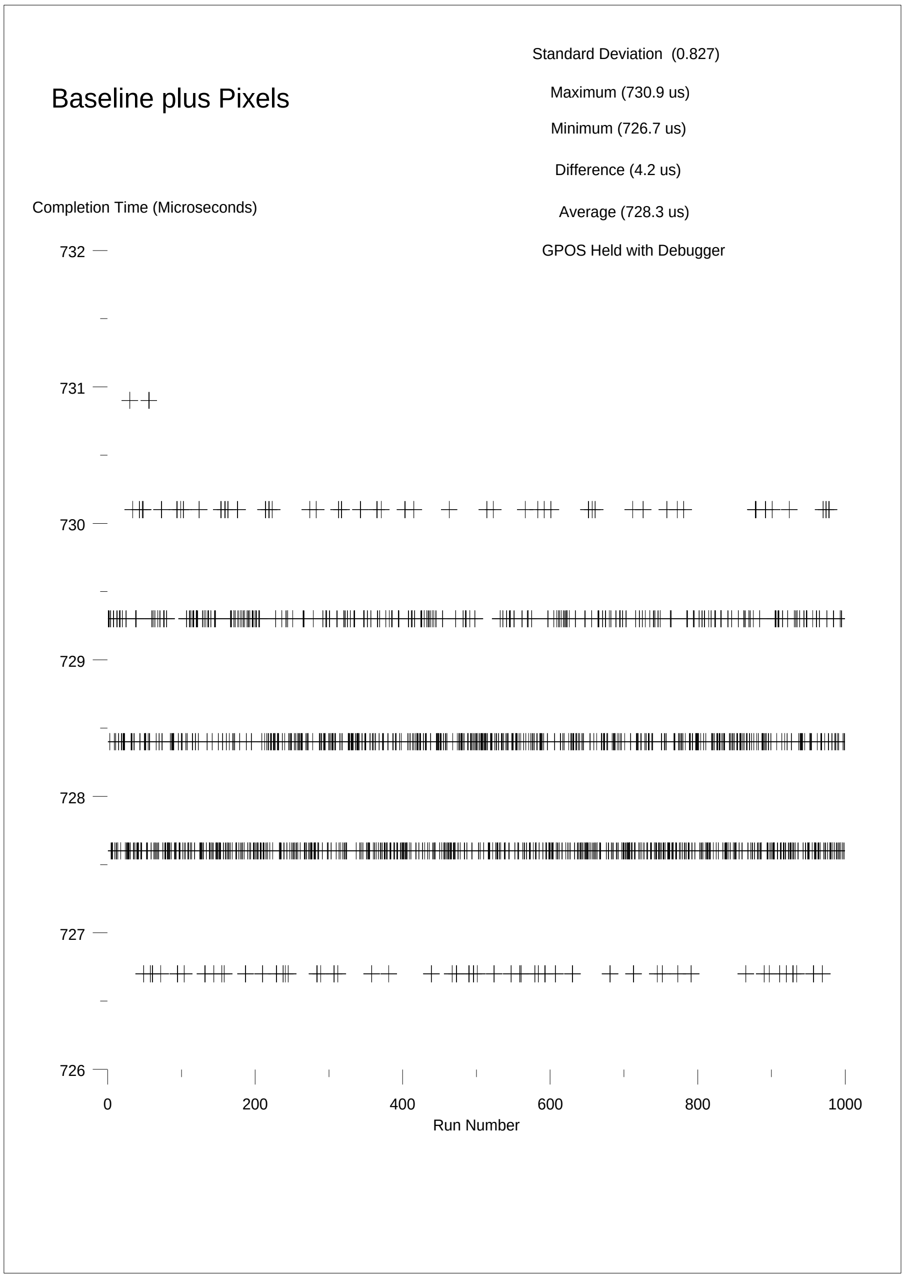

Figure 5.13: Variation in Completion Times when the Loop Uses the Display Device

5.4.2.3 Loop Execution Time Variation with Loop Execution in Two Slots

The next experiments add invocations of the RTK and the CPU executive interrupt handler to the measured times. Figure 5.14 shows the measured test loop, execution time, and completion time intervals. The change is that sufficient iterations are programmed in the test loop to cause it to execute across a slot boundary but within a real-time minor cycle. At first glance, this additional invocation of the RTK would seem to add about as much variability as a single invocation of the RTK. However, the CPU executive interrupt handler, i.e., the mechanism that responds to the interrupts from the RTClock that defines slots, executes within an interval measured by the experiment. Thus, additional variability arises in the measured interval not just from the RTK and CPU executive interrupt handler, but also from the off-CPU timer chip, the programmable interrupt controller chips, and the low-level interrupt dispatching function of the CPU. The experiment tests whether these additional components introduce more variation in the loop execution and completion times. Graphs 5.15 and 5.16 indicate that the variability is only slightly above the baseline.

5.4.2.4 Loop Execution Time Variation Caused by GPOS Execution

This set of experiments includes GPOS execution, i.e., the GPOS is not held inactive with the debugger. This experiment particularly examines the effect of DMA started by non-real-time tasks in non-real-time minor cycles that continues during real-time minor cycles. This effect appears only at the start of the real-

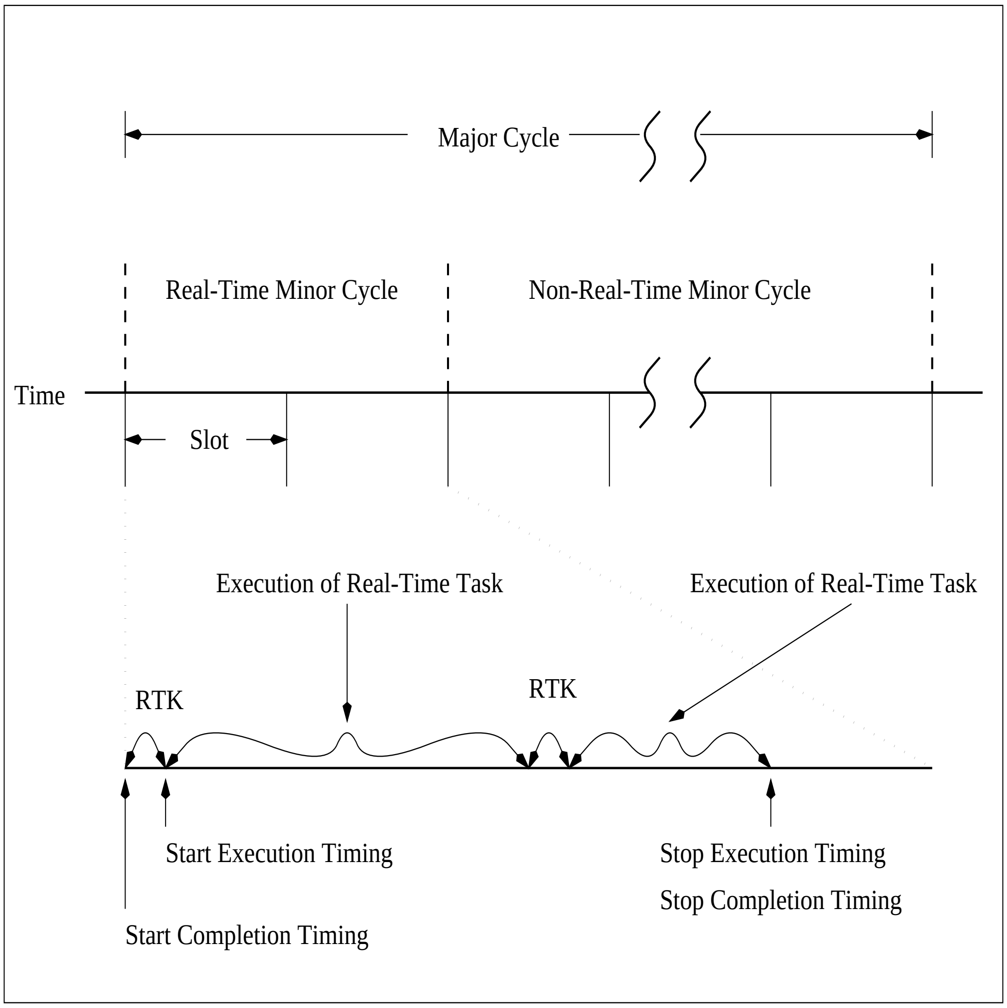

Figure 5.14: Variation in Execution and Completion Times for a Loop that Executes Across Two Consecutive Slots

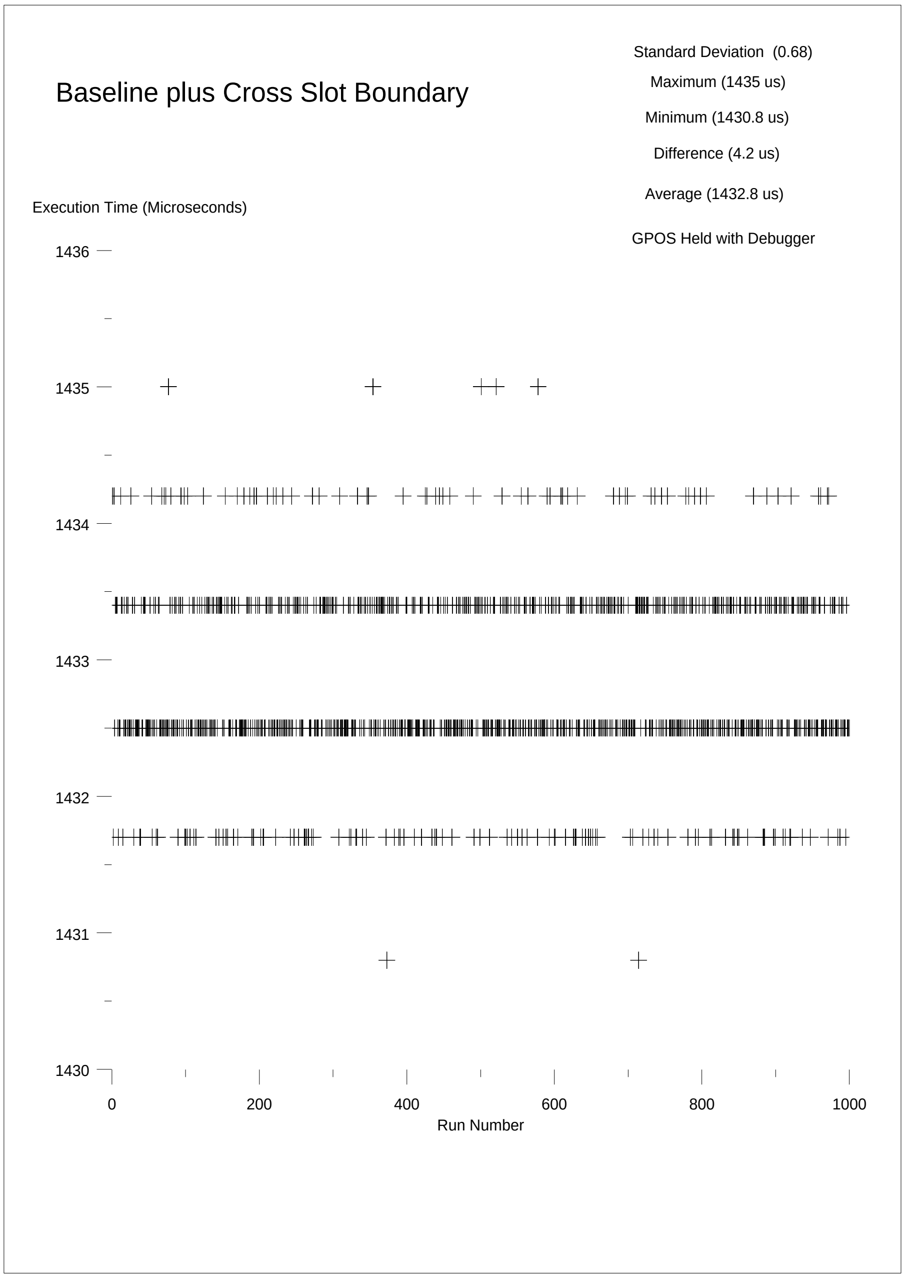

Figure 5.15: Variation in Execution Times for a Loop that Executes in Two Consecutive Slots

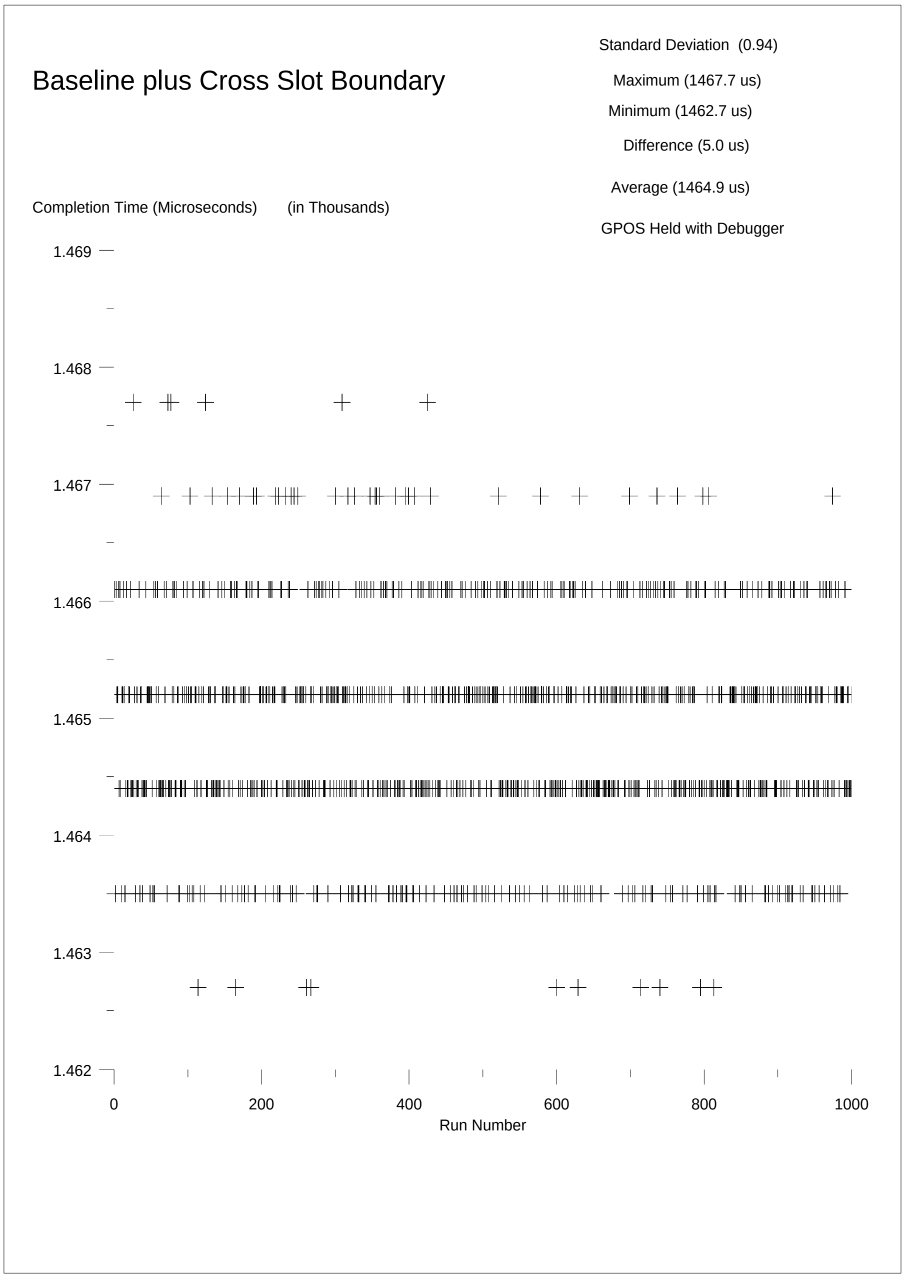

Figure 5.16: Variation in Completion Times for a Loop that Executes in Two Consecutive Slots

time minor cycle during RTK execution because the instruction cache probably does not contain the instructions of the RTK. The effect is not prominent in the loop execution time because the loop is small enough to fit completely in the instruction cache and no other task, thread, interrupt handler, or trap handler executes during loop execution (so loop instructions stay in the cache).

The MicroChannel bus allows active DMA to stall the CPU, as is typical in most architectures. This increases the time to access memory (since the bus is used by the DMA process) and therefore increases the time needed to execute the test loop. This implementation of the SP architecture does not attempt to control the DMA process. Figures 5.17 and 5.18 show that the effect of DMA is evident in completion time but not execution time. This is just an artifact of the size of the test loop and availability of the instruction cache. The completion time measurements start at the beginning of the slot and the execution time measurements start at the point at which the loop begins execution. The average execution time of the RTK in this experiment is about 44 μs or about 20 percent longer than its execution in the baseline experiment.

5.4.2.5 Loop Execution Time Variation with Loop Execution in Two Real-Time Minor Cycles

This set of experiments increases the number of iterations of the test loop so that it requires about six slots to execute. Since a real-time minor cycle is only five slots long, test loop execution will span a non-real-time minor cycle and will include six executions of the RTK. Figure 5.19 illustrates the measured

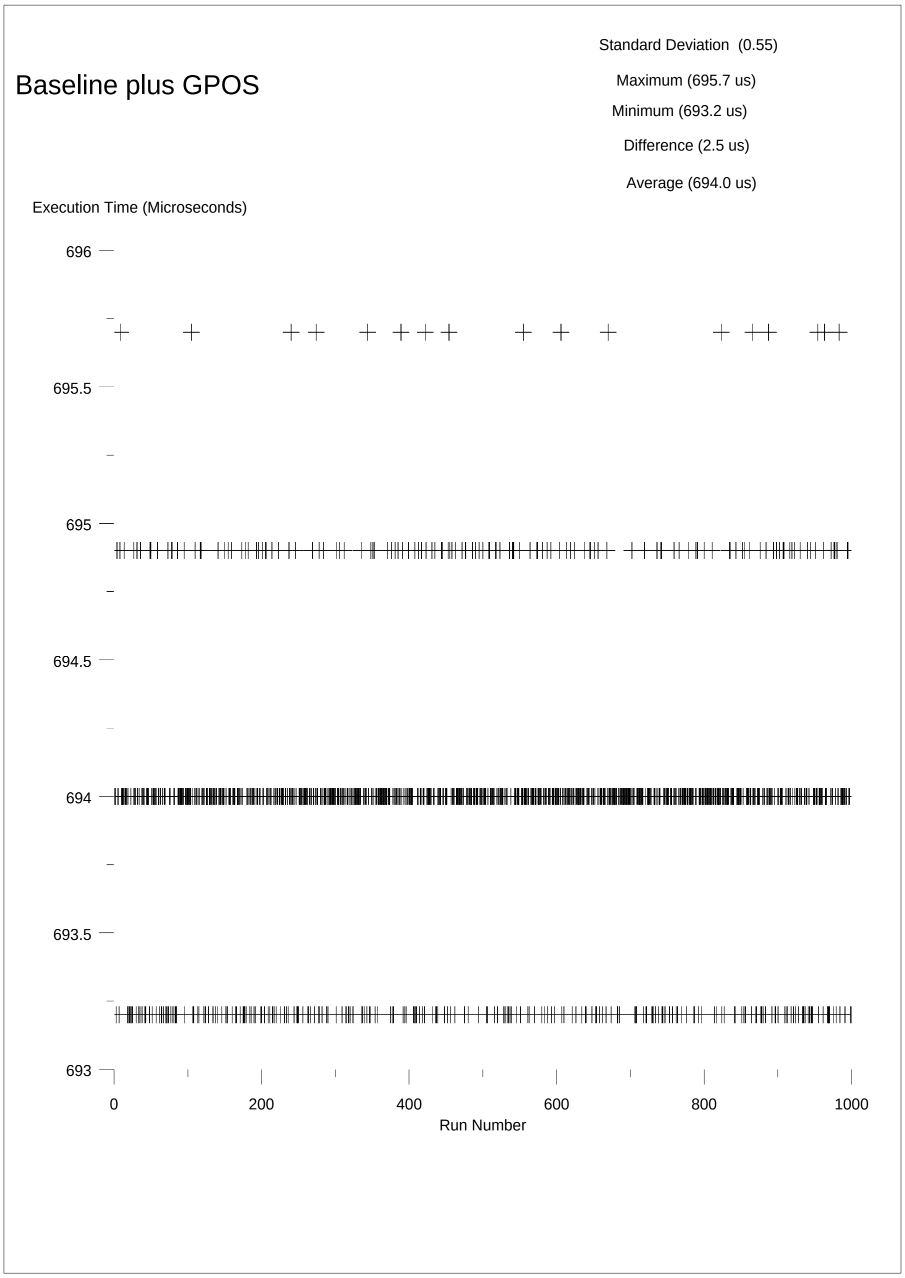

Figure 5.17: Variation in Execution Times when the GPOS is Active

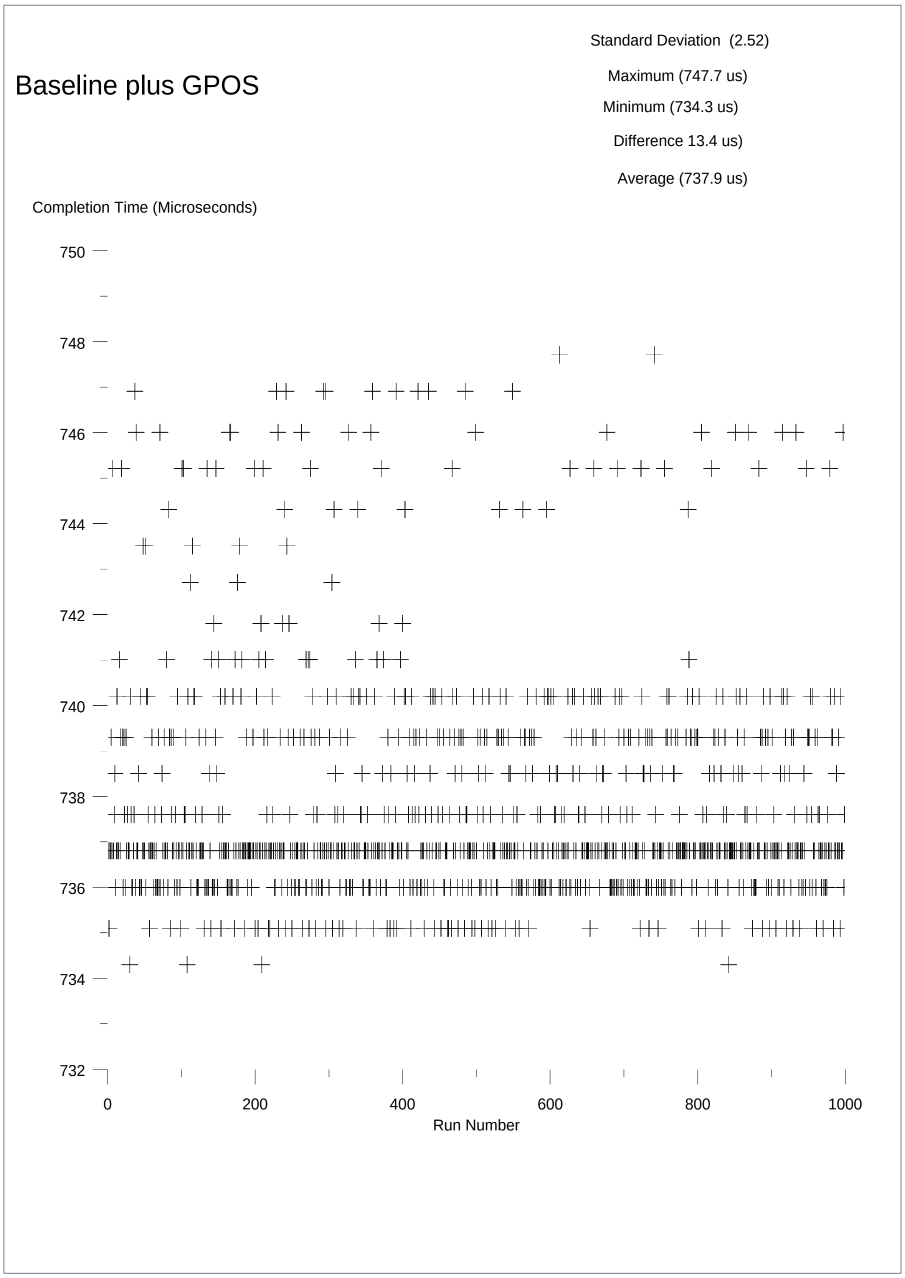

Figure 5.18: Variation in Completion Times when the GPOS is Active

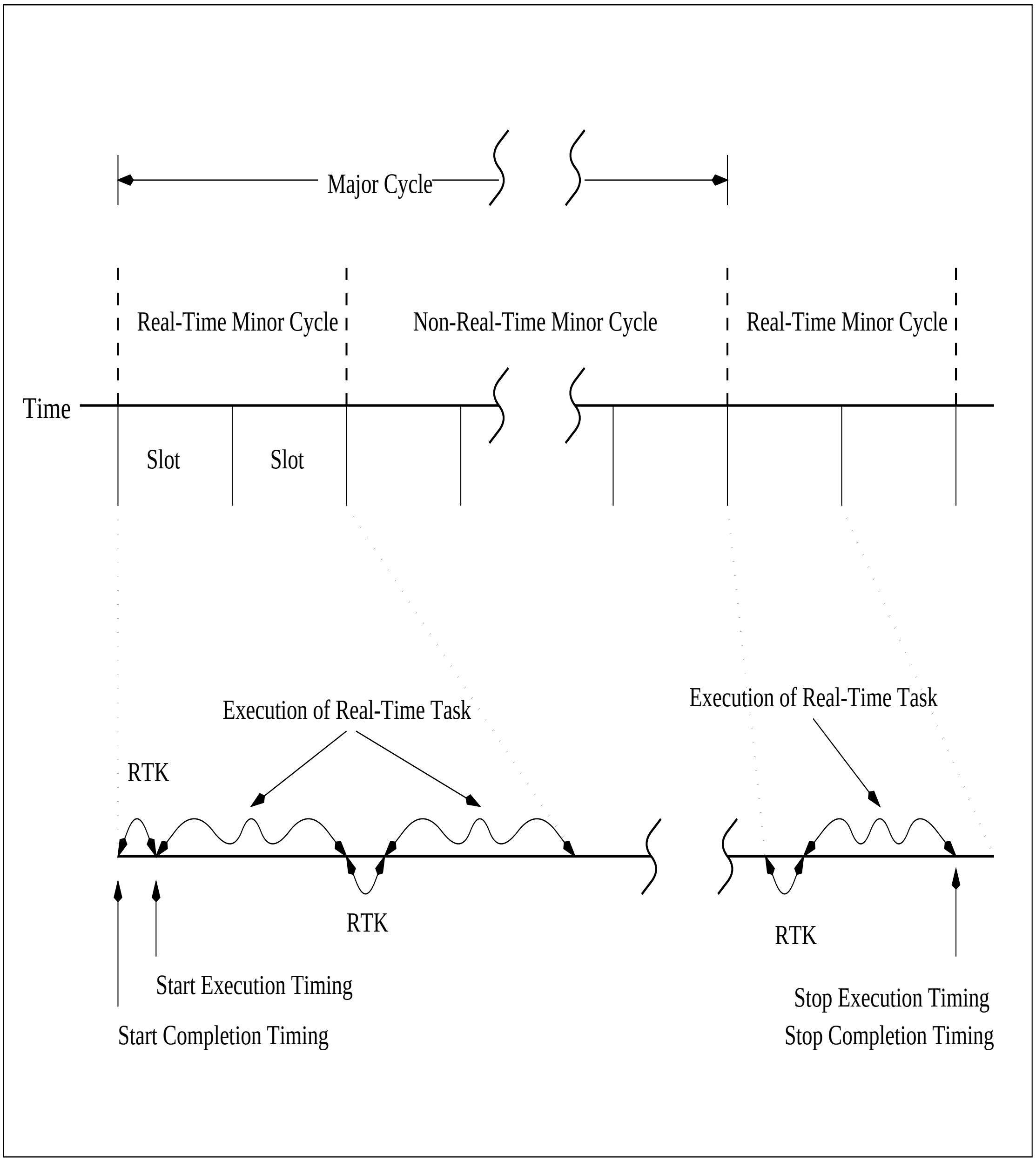

Figure 5.19: Loop Execution in Two Consecutive Real-Time Minor Cycles Experiments

intervals. Figures 5.20 and 5.21 show that the average execution and completion time required to execute the test loop is now about 19 milliseconds. The shortest and longest measured times differ by about 0.058 milliseconds, or about 0.3 percent. The three points that contribute to that difference, at about 19.6 milliseconds on each graph, can be predicted from Figure 5.6. On that graph, about 99.8 percent of the slot lengths are within 0.058 milliseconds of nominal. It appears that the length of the last slot in the non-real-time minor cycle is one of the longer slots and, therefore, the first slot of the second real-time minor cycle begins late. This effect is not seen when the the test loop executes in two subsequent slots, because the GPOS does not execute between the slots of a real-time minor cycle, and therefore the critical sections in the GPOS that must be serialized with the RTK do not delay the second, and subsequent, slots of a real-time minor cycle, causing the higher values seen in the requirement $\mathcal{A}$ experiments. This experiment illustrates two important points about slot length variation: the errors are *not* cumulative and they tend to cancel out. If these conditions were not true, the variability in this experiment would be rather large.

5.4.3 Arguing that this Implementation of the SP Architecture Meets Guarantees

To show that a particular implementation of the SP architecture can guarantee that real-time threads will all meet their deadlines, the subject implementation must meet the two requirements restated in Section 5.4. The results of our experiments demonstrate that this implementation does meet them. Chapter 6 presents

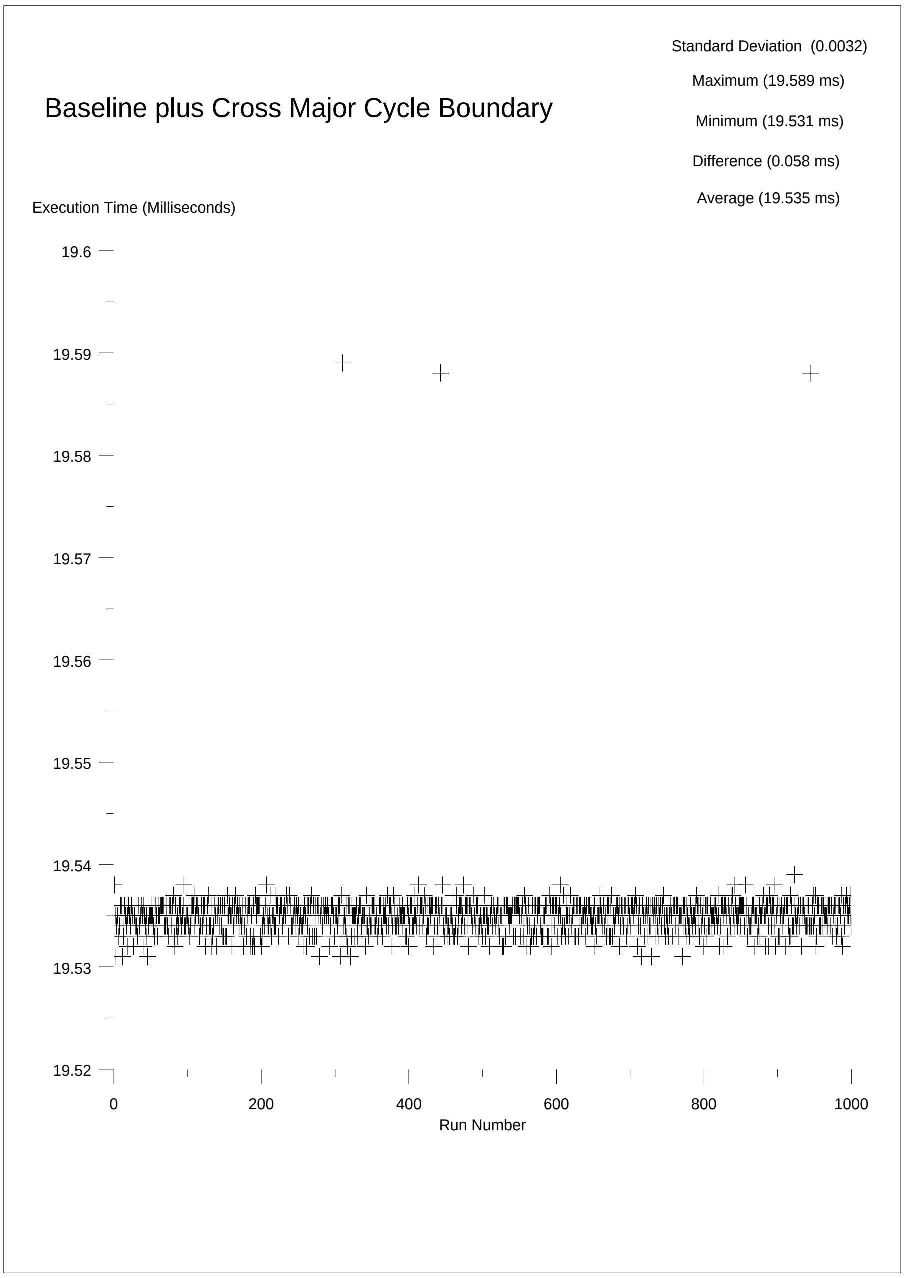

Figure 5.20: Variation in Execution Times for a Loop that Executes in Two Consecutive Real-Time Minor Cycles

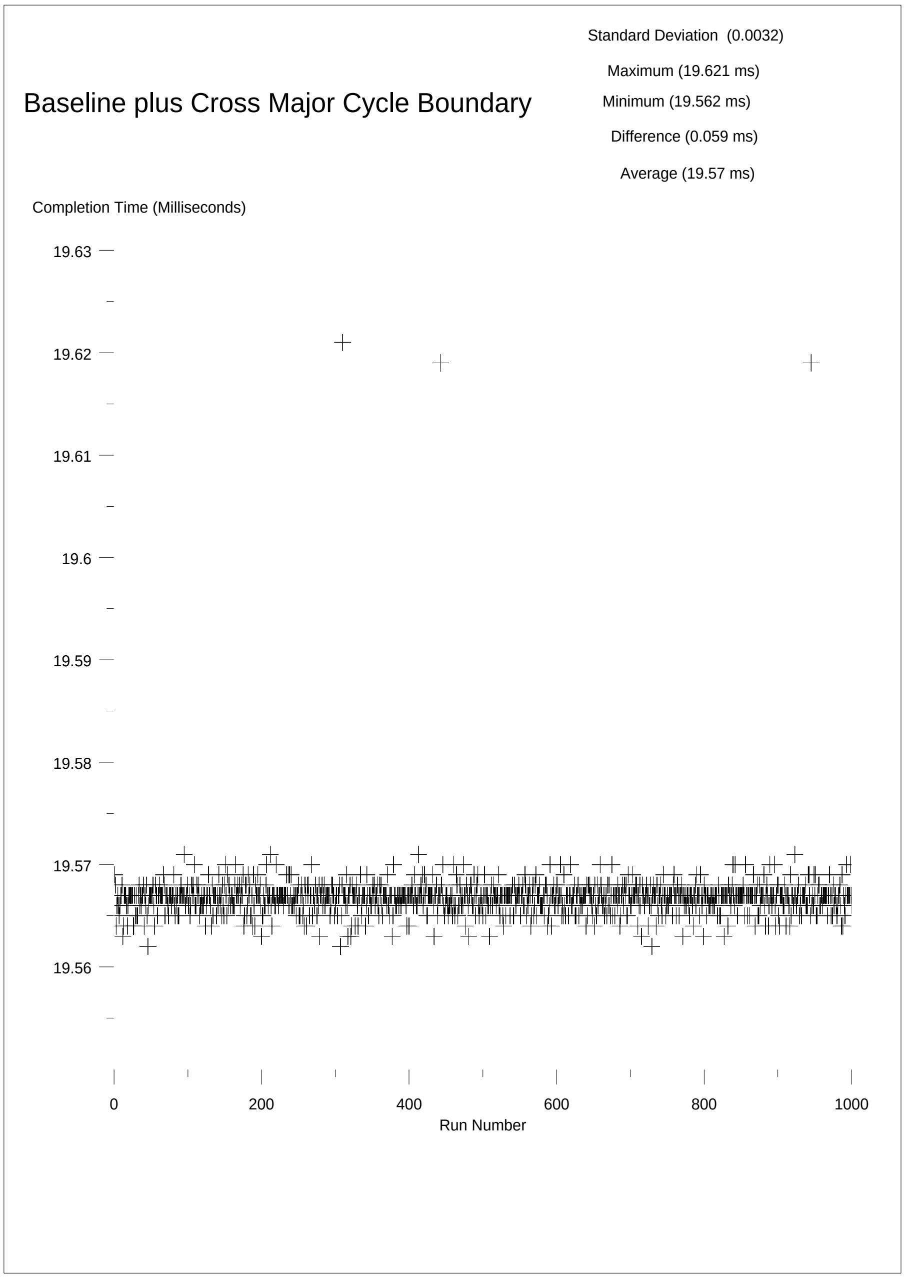

Figure 5.21: Variation in Completion Times for a Loop that Executes in Two Consecutive Real-Time Minor Cycles

an analysis of a set of real-time threads which provides additional proof that a feasible set of real-time threads will all meet their deadlines when executed on this implementation.

5.4.4 SP Overhead

The experiments presented in this section quantify the overhead introduced when the components of this implementation of the SP architecture are executed. The experiments are designed so that a constant computational demand is executed on the system and the time required for its completion is recorded. As various components of the SP architecture implementation are introduced into the system, the constant computational demand is executed in the new environment and the time recorded. The recorded times indicate the amount of overhead introduced by each component.

Table 5.2 shows the times obtained by measuring the time used to execute a constant load (a build of the IBM MicroKernel) under varying conditions. The first experiment refers to the time required for the load when the MicroKernel has no modifications for the SP implementation. The second experiment refers to the CPU executive interrupt handler, which was responding to interrupts from the RTClock but immediately returning to the GPOS. The third experiment refers to the situation where the CPU executive interrupt handler switches to the RTK for the slots in the real-time minor cycle but finds no real-time threads ready to execute. The fourth experiment is the same as the third but with real-time threads ready to execute.

Build Time (slot = 488μs, mc_{rt} = 2 slots, mc_{nrt} = 24 slots)	
Experiment	Elapsed Time (seconds)
Unmodified IBM MicroKernel	180
Interrupt enable/disable virtualization	184
CPU executive interrupt handler execution	201
Two real-time threads executing	216

Table 5.2: Execution time in seconds for an IBM MicroKernel build.

Table 5.3 shows that the interrupt enable/disable virtualization mechanism increases the execution time about 2 percent. This value is calculated from the data in Table 5.2 and is the increase in load execution time caused by the interrupt enable/disable virtualization mechanism, compared to the time to execute the load by the unmodified MicroKernel. Although this figure may seem high at first, it is reasonable for this implementation. The IBM MicroKernel enables and disables interrupts often (about 900,000 times in the first two minutes of power up). An additional function call was added to each call, along with the logic needed to virtualize the interrupt bit for the GPOS.

The interrupt processing in the RTK takes another 6 percent. This includes at least two task switches for each interrupt, one to start the RTK executing and another to switch back to the task in the GPOS. The percent increase in the CPU executive interrupt handler execution in Table 5.3 is the overhead, above

Overhead (slot = 488μs, mc_{rt} = 2 slots, mc_{nrt} = 24 slots)	
Component	Percent
Interrupt enable/disable virtualization	2.17
CPU executive interrupt handler execution	8.46
Two real-time threads executing	6.94
Total of all SP implementation code	16.67

Table 5.3: Measured overhead in percent of all processor cycles consumed.

the interrupt enable/disable virtualization component overhead. The two real-time threads executing row in Table 5.3 is the overhead imposed by the real-time threads over that of the two previous rows. And the last row in Table 5.3 is the total overhead introduced by the implementation.

5.4.5 GPOS Slowdown

In these experiments (see Tables 5.4 and 5.5) a constant computational demand is executed five times on the system while the SP implementation is executing a constant real-time load, and the time required for each trial to complete is recorded. The real-time load is increased in each experiment from 0 to almost 50 percent. The recorded times indicate the amount of slowdown introduced by the SP implementation. For all of the experiments, slots are 976.562 μs, the major cycle length is 20 slots for all experiments, the real-time minor cycle length is 5 slots for experiments 1-5, 10 slots for experiment 6, and 15 slots for experiment 7.

GPOS Slowdown				
(slot = 976.562μs, MC = 20 slots = 19.531 ms)				
Experiment	1	2	3	4
mc_{rt}, mc_{nrt} (slots)	5,15	5,15	5,15	5,15
RT-thread Percent	0	0.296	2.96	5.92
User Loop Time	25,672,355	25,778,910	26,791,177	27,892,439
	25,672,423	25,776,856	26,791,369	27,870,389
	25,672,445	25,778,010	26,791,168	27,881,933
	25,672,451	25,776,791	26,785,379	27,892,443
	25,672,403	25,776,828	26,785,289	27,892,457
Average	25,672,415	25,777,479	26,788,876	27,885,932
Percent Increase	NA	0.41	4.35	8.62
Factor Increase	NA	1.0041	1.043	1.086
Percent Overhead	NA	0.112	1.16	1.99

Table 5.4: Percent Slowdown of a User Loop

As real-time threads in the system use more CPU, non-real-time tasks will take longer to complete, since the CPU is now shared and the real-time threads get as much CPU as they need to complete on time. In this implementation, overhead is associated with each slot, the execution of the RTK, the time required by the hardware task switch mechanism, and the CPU executive interrupt handler.

GPOS Slowdown			
(slot = 976.562μs, MC = 20 slots = 19.531 ms)			
Experiment	5	6	7
mc_{rt}, mc_{nrt} (slots)	5,15	10,10	15,5
RT-thread Percent	11.84	23.7	47.7
User Loop Time	30,371,156	37,141,288	66,498,582
	30,391,223	37,141,187	66,461,377
	30,328,074	37,146,016	66,530,311
	30,359,462	37,146,161	66,461,631
	30,336,140	37,111,096	66,530,196
Average	30,357,213	37,137,149	66,496,419
Percent Increase	18.25	44.7	159
Factor Increase	1.182	1.447	2.589
Percent Overhead	3.56	7.19	13.68

Table 5.5: Percent Slowdown of a User Loop (continued)

Tables 5.4 and 5.5 show the percent increase in wall-clock time expended by a user task executing in the GPOS as the percent of CPU used by a real-time thread increases. The third row in each table gives the measured percent of CPU used by a real-time thread. The real-time thread uses 10 times more CPU in experiment 3 than in experiment 2 and 2 times more in each subsequent experiment after experiment 3. The next five rows contain the measured times (in timer ticks, 838.222 nanoseconds) taken by the user (non-real-time) task to complete in five trials. The average of the five trials is also given. The percent increase row in the tables shows the increase in percent of the user task over the time needed when no real-time threads were executing. The factor increase row in the tables expresses the percent increase as a factor.

Experiment 7 gives an execution time increase factor of 2.589, with 47.7 percent of the CPU used by a real-time thread. One would expect a factor increase of about 1.91. So, given the 2.589 factor increase, the percent CPU used by the real-time thread, the RTK, and the task switches is 61.38. This implies that the RTK and task switches use 13.68 percent of the CPU under these conditions. Note that the RTK overhead percentage increases linearly with the percent increase of CPU used by the real-time thread. This is expected since the overhead is incurred for each slot; if the real-time thread took twice as much CPU, it would execute in about twice as many slots.

5.4.6 Summary of Experiments

The experiments described in this chapter show that the implementation meets the two fundamental requirements. The slot length experiments measured over 10,000,000 slot lengths and found that all were within 360 microseconds of the expected length. The experiments that measured the real-time thread test loop showed that for tasks that execute in the real-time minor cycles, time, rather than

CPU cycles, can be used to measure the progress of tasks. This is because in any interval comprised of real-time minor cycles, the minimum number of CPU cycles dedicated to executing the instructions of real-time threads can be determined. A set of experiments that were run to determine the effect of adding the SP system to the IBM MicroKernel showed that the GPOS slows down as real-time tasks use more of the CPU.

From these results one can conclude that the variations in major cycle lengths and task execution times are reasonable. One can also conclude that the maximum delays in lengths or extensions of execution times are upper bounds, as supported by an analysis of the source code and hardware specifications that agrees well with the empirical observations. The analysis of the source code concluded that the largest critical section between the RTK and the GPOS would require 33 memory accesses by the CPU to fill the cache and that bus contention between the 33 memory accesses, plus the memory access time itself, would result in the critical section requiring 320 μs to complete. Most importantly, given the two previous conclusions, a feasible set of periodic real-time threads executing on this implementation will all meet their deadlines.

5.5 Discussion

This chapter describes one implementation of the SP architecture on a particular hardware platform and with a particular GPOS. Investigators need to be know the essential issues, such as:

- What types of resources are typically found on the particular hardware platform?
- What kinds of executives will have to be constructed?
- What are the sources of periodic or variable interrupts on the hardware?
- Which of those, if any, does the GPOS use when applying SP to a particular hardware and GPOS combination?

A particular GPOS implies a particular set of device driver implementations. Also, in this particular implementation of SP the relative sizes of the minor cycles are important. Both these issues are discussed.

5.5.1 New Implementation Design Considerations

This section describes some of the thought necessary in the design phase of a project to implement the SP architecture on a new hardware/GPOS combination.

The SP architecture requires a predictable source of interrupts from the hardware to create the minor cycles. If the device creating the interrupts can dynamically vary the time between successive interrupts, slot implementation will not be necessary. The intervals should be programmable from a few microseconds to several hundred milliseconds. If the hardware lacks a dynamic capability, fixed intervals (slots) can be used with additional overhead. In the fixed interval case, minor cycles are created out of some number of fixed intervals. If the hardware lacks a predictable timer, or the GPOS has monopolized all the hardware timers and timers cannot be shared between the GPOS and the RTK, a separate timing board must be used.

Since a particular SP implementation must virtualize the interrupt enable/disable mechanism, having a programmable interrupt controller (PIC) considerably simplifies coding the virtualization modules. The essential feature is the ability to mask the various interrupt lines from physical devices so that the interrupts from physical devices can be selectively stopped at the PIC. This is easier than than allowing the CPU to recognize virtualized interrupts and having the virtualization code record the interrupt and then delay the actual processing of the interrupt handler. When the GPOS requests that interrupts be enabled, the virtualization module can restore the mask so that the PIC will raise the interrupt line to the CPU when any line from the physical devices indicates an interrupt. The coding of the virtualization module is significantly complicated without a mask at the PIC.

If the hardware supplies a task switch mechanism and the GPOS uses all, or part of it, the CPU executive will have to be coded with critical sections around such use. Typically, the GPOS will have disabled interrupts around its use of the task switch mechanism, but since the virtualization module has virtualized interrupts, with respect to the GPOS, the interrupt from the predictable timer used to define a minor cycle can still occur when the GPOS uses the task switch mechanism. If the CPU executive also uses the hardware task switch mechanism to switch to the RTK, a critical section will be violated and the system will become corrupted. Thus, it is important when implementing SP architecture to correct for any resources, such as the task switch mechanism, that are shared between the GPOS and the RTK. In the case of critical sections of code, the options are either not to share the code or to use the real interrupt enable/disable mechanism.

If the GPOS provide system calls to copy code into the kernel, this call can be used to implement a `CreeteRTThread` system call. Real-time threads can then be copied into the kernel and executed in the address space of the RTK. Alternatively, the real-time threads may be executed in user space but the two requirements the SP architecture imposes on an implementation must still be met. For example, the memory in which the real-time threads exist will have to be excluded from the paging mechanism (usually called 'pinning the memory').

The device drivers of a particular GPOS may need to be modified when executives for the devices are created, because there is no clean way to allow an executive to access the particular device when isolated from the GPOS device driver or when the GPOS device driver is not robust enough to handle exclusion from the device during the real-time minor cycles. One issue that can be problematic is the robustness of the device drivers when they are interrupted by a real-time minor cycle. Typically, GPOSs are not coded to allow nested interrupts although hardware support for such interrupts is generally available. Thus, the existing device drivers will have been coded and tested in an environment where they are not interrupted. Implementing the SP architecture requires that minor cycles begin on time, even if they occur while a GPOS device driver is executing. The issue here is not that new critical sections need to be handled, but that the drivers may be unable to recover after being interrupted by a real-time minor cycle, because they have entered a new state. Since the device driver does not recognize the new state, it will interact with the device as though it were in its previous state. When the device responds as dictated by its new state, the device driver may not expect that particular response.

5.5.2 Minor Cycle Lengths

The admission control function of the RTK sets values of mc_{rt} and mc_{nrt} to ensure that enough CPU cycles are available to the set of real-time threads that they all meet their deadlines. But for a given set of real-time threads, many sets of possible values for mc_{rt} and mc_{nrt} exist that will ensure sufficient CPU cycles. This section addresses which set of values the admission control function should choose and how that decision is made.

Consider a set, $\mathcal{T}$, of n periodic real-time threads, as described by Liu and Layland, such that thread τ_i has cost c_i and period p_i [24]. We know that $\sum_{i=1}^{n} \frac{c_i}{p_i}$ is the CPU utilization of the real-time threads. Thus, we know that the following must hold (it is necessary but not sufficient) before the feasibility analysis will succeed:

$$\frac{mc_{rt}}{MC} \geq \sum_{i=1}^{n} \frac{c_i}{p_i}$$

But this is not the whole story. Suppose $\sum_{i=1}^{n} \frac{c_i}{p_i} = 0.5$. Then the ratio $\mathcal{R} = \frac{mc_{rt}}{MC} = 0.5$ would be realized when $mc_{rt} = 500$s and $MC = 1000$s. However, if one real-time thread has a period of 100 seconds, the feasibility analysis would not indicate that the thread set was feasible. So as a first step, the length of the MC must be bounded from above by $\min(p_i)$ for $1 \leq i \leq n$, which would indicate that $MC = 100$s and $mc_{rt} = 50$s in the example. Although $\mathcal{R}$ is the same 0.5, the feasibility analysis would indicate that the real-time thread was feasible (given that the cost of the real-time thread was less than 50s).

Now consider how one might determine a minimum length for the major cycle. Let O be the overhead introduced by some of the components of an SP implementation:

- the execution of the RTK at the start of each real-time minor cycle (and, if slots were used, at the start of each slot within a real-time minor cycle)
- the cost of switching among the GPOS, RTK, and the real-time threads
- the cost of handling the interrupt that marks the minor cycle, or slot, boundaries.

Note that the overhead of the interrupt enable/disable virtualization mechanism (IEC) is specifically not included in O, because it is within the GPOS and independent from the minor cycle lengths.

Now, consider the effect of allowing MC to be about the same value as O. The overhead, not including the IEC, of the SP implementation is 50 percent. We suggest $\frac{O}{MC} \leq 0.01$. In this implementation the average execution time of the RTK is about 36 μs and, including the other overhead factors, assume $O = 40$ μs. This would indicate that $MC \geq 4$ ms.

If only one real-time thread is in the system it may be possible to make the length of the real-time minor cycle equal to the cost of the real-time thread plus the overhead, and the length of the major cycle equal to the period of the real-time thread. This may also be possible for a set of real-time threads whose periods are identical, such as the reception and display of a video stream. Although many real-time threads may be involved in the reception and display, they would all have the same period—the interval between video frames.

5.6 Summary

An implementation of the SP system was constructed on an IBM PS/2. The GPOS was a version of the IBM MicroKernel with OSF/1 as the dominant personality. The interrupt disabling system was virtualized and a real-time kernel which used an earliest deadline first scheduler was built. A set of experiments showed empirically that the implementation met the two requirements stated in Chapter 3. The first set of experiments measured the lengths of consecutive slots to determine if the interrupt from the RTClock was initiating the start of the slots at precise intervals. The second set of experiments measured the execution time of a test loop to determine the amount of variability in execution time for threads executing in the real-time minor cycles.

Based on these experiments, one can conclude that the variations in major cycle lengths and task execution times are primarily due to instruction caching and bus contention induced by DMA. In any event, the variations are reasonable and upper bounds on slot deviation and execution time within real-time minor cycles can be proven, as shown by analyzing the source code and hardware specifications and comparing that analysis with empirical observations. Most importantly, given the two previous conclusions, a feasible set of periodic real-time threads executing on this implementation will all meet their deadlines.

Chapter 6

Application Demonstrations

6.1 Introduction

From the implementation description and experimental results presented in the previous chapter, one can conclude that a feasible set of periodic real-time tasks executing on that implementation would meet their deadlines. This chapter carries that work further. First, it empirically validates a feasibility analysis on a set of real-time threads and, second, it demonstrates the programming paradigm and the use of the system. Two real-time applications were designed and programmed on the implementation. The first displayed a character on the display device in each period of a set of real-time threads. The second real-time application received an audio stream from the network and played the audio on the system speaker. All of the real-time threads of the two applications met their deadlines.

In addition, this chapter also introduces the notion of a sporadic real-time task with another audio application demonstration. This alternate demonstration shows how the SP architecture can support other models of real-time computation and how resources are shared as defined by the SP architecture. The character display demonstration described in Section 6.2 illustrates sharing of the display, the processor, and system timers. The network audio demonstration, described in Section 6.3, illustrates the use of sporadic real-time tasks and sharing of the network adapter card. Both demonstrations execute on the implementation described in Chapter 5.

6.2 Character Display Demonstration

This application demonstrates empirically that a successful feasibility analysis of two real-time threads implies that the threads will meet their deadlines even though the GPOS is heavily loaded. Additionally, the demonstration uses two GPOS non-real-time threads programmed to display characters in the same manner as the real-time threads, but on different horizontal lines. These two non-real-time threads illustrate the load placed on the GPOS and the effects of real-time vs. non-real-time scheduling.

Before a character is displayed at location (x, y), the character at location $(x-1, y)$ is erased. The visual effect is that of a character moving from the left edge of the display horizontally toward the right edge of the display. When the previous character displayed reaches the right edge of the display the current character is displayed at the leftmost position of the same line and the previous character is erased. The visual effect is that the character instantaneously restarts at the leftmost position after reaching the rightmost position.

In the application, two instances of this display process are active (on separate lines). Each character is displayed by its own real-time thread and thus the speeds at which the characters are displayed may differ. The speed at which a character moves across the display is controlled by the period of the real-time thread that displays the character. Since 70 (out of a possible 80) horizontal character positions are used for each display line, a character controlled by a real-time thread with a period of 100 ms will move 10 horizontal positions every second and take 7 seconds to cross the display. In addition to displaying the character, each real-time thread has an artificial computational load which consists of a loop with addition operations as its body. The number of iterations of the loop determines its computational load.

This application demonstration uses three shared resources: the CPU (a preemptible resource) the video display terminal (a statically partitionable resource) and a timer chip (also a statically partitionable resource). The slot length was 976.562 microseconds and the major cycle length was 33 slots (32.2 ms). The real-time minor cycle length was 2 slots (1.95 ms), so the non-real-time minor cycle length was 31 slots (30.3 ms). The test ran for 48 hours. During this time various loads were run on the GPOS, including network transfers of large files and compilations of the GPOS kernel, and no real-time thread ever missed its deadline. In contrast, the two GPOS non-real-time display threads missed deadlines often.

6.2.1 Shared Resources

The SP model specifies that any resource used by both the GPOS and the RTK must be shared such that a real-time thread scheduled by the RTK will have the resource when needed. In this demonstration the CPU, the video display, and the 8254 programmable interval timer were the three resources used by both the GPOS and the RTK. These are among the simplest of resources to share, because the actions of their executives are straightforward. As explained in Section 3.8.2, the SP model considers the CPU a dynamically preemptible resource. This means that at well defined points, the complete state of the resource can be saved and subsequently restored with no effect on the thread using the resource except for the passage of time. For example, the primary requirement of the CPU executive is that it can control and preempt the CPU at well-defined times by the virtualizing the interrupt enable/disable mechanism and use of the RTClock.

The display is an example of a partitionable resource. This type of resource, described in Section 3.8.1, is the easiest to build an executive for. Its individual units are character locations in the display memory. In the system used for this implementation each character is represented by two memory-mapped bytes of main memory (although the memory address is not really main memory but memory on the display adapter card). The starting address, s, of the memory-mapped display represents the character in the lower left corner of the display. The values written to bytes with even addresses, $s+(2k), k=0,1,2,\ldots$, define the character to be displayed at the location (x,y) whereas the values written to bytes at odd addresses, $s + (2k + 1)$, define the *attributes* of the character at location (x, y). Attributes include foreground and background color, intensity, and whether the character is flashing.

For this demonstration, the character buffer locations of the display were statically partitioned into two sets: Those for the lower four rows of the display were assigned to the GPOS, and those for the remaining rows were assigned to the RTK. This partitioning was accomplished with a UNIX utility called `window` running on the GPOS. This utility partitions the display into a number of windows which the user of the system can use independently. Two windows were created: window 1 was the lower four rows of characters and window 2 was the remaining rows. GPOS threads all used the lower window; when the RTK started, it wrote directly to the memory-mapped bytes for the characters of the remaining rows. However, the `window` utility could be switched to the window reserved for the real-time threads, allowing non-real-time threads to overwrite the display[1] Although this demonstration used the standard 25x80 character display, the technique could easily be generalized to a bit-mapped, graphical, windowed user interface. For example, one could modify the X-windows server to create real-time threads to display specific data.

The second shared resource used by this demonstration was the 8254 programmable interval timer, which is also a partitionable resource. The chip, or its

[1]With other utilities, however, it would be possible to define the display size to the GPOS as just the lower four rows, which would preclude the overwrite problem. This alternate method was not investigated.

emulation in some larger circuit, has three registers which can be used for timing. The GPOS (IBM MicroKernel) used only one register for its timing needs. The GPOS sometimes used this register to produce a waveform that would be routed to the speaker. Because the GPOS also used this register when it needed to create a sound at the speaker, it already had code to use a different register for its timing needs when the primary register was used for the speaker. We exploited this fortunate situation by using the first register for SP timing and statically requiring that the GPOS always use the second. This was done by modifying the source of the GPOS so that it always detected that the speaker was in use. This is an instance of partitioning. The RTK thread services timing routines used the timer to provide real-time threads and the RTK with submicrosecond timing granularity.

6.2.2 Real-Time Kernel

The real-time kernel in this demonstration consisted of an earliest deadline first (EDF) scheduler and a set of library functions. The library functions provide real-time threads with calls to set the character and attributes of a given display character buffer location, to measure and convert time, and to block until their next invocation. The RTK maintains data structures for each real-time thread including the thread's period and deadline (in units of slots) and an indicator of whether the thread is ready to run or is blocked. Since each slot is 976.562 microseconds the granularity with which the RTK can perform activities within a real-time minor cycle is approximately once every millisecond. At the beginning of each real-time minor cycle, the RTK schedules the ready real-time thread whose deadline is closest. If a real-time thread finishes before the end of a real-time minor cycle, the RTK schedules the real-time thread with the closest deadline from among the ready real-time threads.

6.2.3 Real-Time Threads

This demonstration system used two real-time threads, which were processes with respect to the CPU architecture. Each thread's CPU state is represented in a task state segment (TSS), as defined by the Intel i486 architecture. In addition to a TSS, the RTK also maintains a number of data structures for each real-time thread that record thread parameters and state variables such as a thread's period, deadline, a blocked/ready indicator, and a missed deadline counter. The period is usually set at the time of thread initialization, although it can be changed at any time. The deadline is set at the end of the current period. The blocked/ready indicator is used by both the thread itself and the RTK. When a thread has finished the work that it is programmed to accomplish in a period, it calls a system call to suspend its execution; this call sets the indicator to blocked. At each invocation of the deadline scheduler in the RTK, the blocked/ready indicator for each thread is checked and the thread with the nearest deadline among the ready threads is dispatched. At each invocation of the RTK, the blocked/ready indicator of any thread whose deadline has expired is set to ready. If the RTK detects that a thread's period has expired and the blocked/ready indicator for that thread is *not* set to blocked, the RTK increments the missed deadline counter for that thread.

The real-time threads performed the computation shown in pseudocode in Figure 6.1. The real work of these threads occurs on lines 3, 5, and 6. Line 5 calculates a location (row, column) for the next character to be displayed. Thus it is an artificial load that varies and that simulates the computation a real real-time video display thread might have to accomplish. Line 6 calls the real-time display library to update the values of the particular character. Lines 1, 4, 7, 8, and 9 call the real-time timing library and are used to accumulate the data displayed in some of the graphs in Chapter 3. Line 10 sets the blocked/ready indicator in the thread's RTK data structures to blocked and jumps to the RTK. This indicates that the thread has completed its required work for a particular period. In this demonstration each real-time thread displays one character in each period.

```
   BEGIN
1      Get Starting Time
2      Do Forever
3         For Character Location
4           Start Execution Timing
5           Compute Character Values
6           Set Character Values at Location
7           Stop Execution Timing
8           Stop Completion Timing
9           Save Timing Values
10          WaitForPeriodExpiration
         End For
      End Do
   END
```

Figure 6.1: Real-Time Thread's Computation Loop

The attributes (cost, period) of the real-time threads in this demonstration can be varied dynamically; many combinations were considered and run. One such set was $\tau_0 = (1 \text{ ms}, 33 \text{ ms})$ and $\tau_1 = (2 \text{ ms}, 250 \text{ ms})$. The CPU utilization of τ_0 is $\frac{1}{33} \times 100 = 3.03$ percent and for $\frac{2}{250} \times 100 = 0.8$ percent for τ_1. For τ_0 the statement on line 5 was configured so that the loop took 1 ms to complete and for τ_0 it was configured such the loop took 2 ms to complete. The period value in the RTK data structure for τ_0 was set to 33 ms and to 250 ms for τ_1.

6.2.4 Non-Real-Time Threads

The non-real-time threads of this demonstration were very similar to the real-time threads. They were actual MicroKernel threads whose pseudocode was identical to that of the real-time threads. The only difference in actual code was in line 10 which called routines in the MicroKernel used to block threads. Note that the non-real-time threads used the real-time library calls for displaying characters and obtaining timings, so that they were as similar to the real-time threads as possible. Although using the real-time library calls biased the results, it only favored the non-real-time threads, since their use of real-time calls only increased their own predictability. The experiment tested whether any threads missed their deadlines because of the computation of line 5.

6.2.5 Analysis

In this section the analysis in Chapter 4 is used with the actual parameter values of the real-time threads to determine if the threads are feasible for certain real-time and non-real-time minor cycle lengths. The feasibility condition is re-stated here:

A set of periodic tasks $\tau = \{(c_1, p_1), (c_2, p_2), \ldots, (c_n, p_n)\}$ can be scheduled on an SP implementation with real-time minor cycle length mc_{rt} and non-real-time minor cycle length mc_{nrt} if and only if for all $L \geq 0$:

$$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i \leq \left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt}) \tag{6.1}$$

where $MC = mc_{nrt} + mc_{rt}$. Table 6.1 shows the values of the SP parameters and the real-time thread parameters. The cost values c_0, c_1 were set experimentally by adjusting the load of each thread's computational statement.

Section 4.3.2 showed that only certain values of L need to be considered in an analysis. We first calculate the upper bound on the values of L we need to consider. The utilization of the system (U_T) is:

$$U_T = \sum_{i=1}^{n} \frac{c_i}{p_i} + \frac{mc_{nrt}}{MC}$$

$$U_T = \left(\frac{1}{33} + \frac{2}{250}\right) + \frac{23}{25} = 0.0383 + 0.92$$

$$U_T = 0.9583$$

Next, find B:

$$B = \frac{mc_{nrt}}{1 - U_T}$$

$$B = \frac{23}{1 - 0.9583}$$

MC	mc_{rt}	mc_{nrt}	c_0	c_1	p_0	p_1
25	2	23	1	2	33	250

Table 6.1: SP and Real-Time Thread Parameters

$$B = 551.5$$

Thus, the values of L that must be considered when using Equation 6.1 to determine the feasibility of the real-time threads in this real-time demonstration application are multiples of the thread's periods, 33 and 250, up to 551. Table 6.2 shows the results of the feasibility analysis. The row labeled with the left-hand-side of Equation 6.1 is the demand for CPU cycles, expressed as time, by the real-time threads in the interval L. The row labeled with the right-hand-side of Equation 6.1 is the minimum number CPU cycles, expressed as time, available for the real-time threads in the interval L.

The results indicate that Equation 6.1 is true for all required values of L, so this set of real-time threads is feasible when executed on an implementation of the SP architecture with the minor cycle lengths given in Table 6.1. Therefore, all invocations of these two threads should meet their deadlines.

L	33	66	99	132	165	198	231	250	264
$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i$	1	2	3	4	5	6	7	9	10
$\left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt})$	2	4	6	10	12	14	18	20	20
L	297	330	363	396	429	462	495	500	528
$\sum_{i=1}^{n} \left\lfloor \frac{L}{p_i} \right\rfloor c_i$	11	12	13	14	15	16	17	19	20
$\left\lfloor \frac{L}{MC} \right\rfloor mc_{rt} + \mathrm{MAX}(0, L \bmod MC - mc_{nrt})$	22	26	28	30	34	36	38	40	42

Table 6.2: Evaluation of Feasibility

6.2.6 Empirical Results

The character display demonstration ran for various durations, up to 50 hours, with different types of non-real-time threads loading the CPU which included a compilation and linking of the MicroKernel itself (which is alternately I/O bound and CPU bound). During the demonstration we also transferred large files using FTP (file transfer protocol), which is I/O bound. In all cases the two real-time threads always met their deadlines—the expected result. The non-real-time threads that duplicated the real-time thread's computation often missed deadlines.

6.2.7 Summary

The character display demonstration used two real-time display threads, two GPOS display threads, and three shared resources. The shared resources were the CPU (a preemptible resource) the video display terminal (a partitionable resource) and a timer chip (also a partitionable resource). The CPU was shared by an SP executive that multiplexed execution of the GPOS and its threads with the RTK and its threads. The display was partitioned by allocating some of the character buffer locations to the RTK for it to re-allocate to real-time threads and some of the buffer locations to the GPOS for it to re-allocate to non-real-time threads. The RTK employed an earliest deadline first scheduler to schedule the real-time threads.

The real-time threads performed an artificial computation for each character displayed. That computation and the display of a character had to be completed before a deadline. Two non-real-time threads, scheduled by the GPOS, also attempted to finish computing in intervals corresponding to the periods of the real-time threads. The analysis determined that the real-time threads were feasible and the empirical data confirmed that assertion.

6.3 Network Audio Demonstration

The second real-time application demonstration uses three real-time threads to receive, process, and play a continuous audio stream arriving from a network and originating on another computer using the OS/2 operating system. This demonstration further validates the architecture and scheduling analysis by using a resource from the non-preemptible, externally-triggered, stochastic process class—the class for which executives are hard to build.

Four resources are shared between the RTK and the GPOS:

- the network adapter, an externally triggered stochastic process resource
- UDP ports, a partitionable software resource
- the CPU, a preemptible resource
- the system speaker and its controlling 8254 timer chip, another partitionable resource

Figure 6.2 shows a functional diagram of the components of the demonstration. A second general-purpose machine sends a series of UDP packets containing a description of musical notes to be played on the speaker across the token ring to the test SP implementation system. The sender and the network are dedicated to this demonstration and the packets arrive at the receiver at predictable intervals. The network adapter's function in this demonstration is to receive from the network the UDP datagram containing audio data. The RTK network device executive uses the UDP port numbers to determine if a particular UDP message is for a non-real-time thread in the GPOS or a real-time thread in the RTK. The system speaker and its controlling circuit create the correct sound indicated by the data in the UDP datagrams.

On the SP implementation machine, the token ring device driver is not modified but shared by placing intercept code in the low-level interrupt handler, the piece of code that initially handles all interrupts and eventually invokes the appropriate device driver.

The application uses three real-time threads. The first executes with each invocation of the RTK at the start of each slot within a real-time minor cycle and polls the network adapter during real-time minor cycles. The second executes at the start of every slot and polls the speaker to see whether the current note has been played for its indicated duration. If so, it turns off the speaker. The third thread starts a note playing in the speaker if the previous note has finished. More details on the operation of these real-time threads are given below.

The application's correctness is determined by listening to the notes since any that are missed or played late are easily identified. The results of running the application with four different combinations of GPOS and network loadings are given in Section 6.3.4. Together they indicate that the real-time threads meet

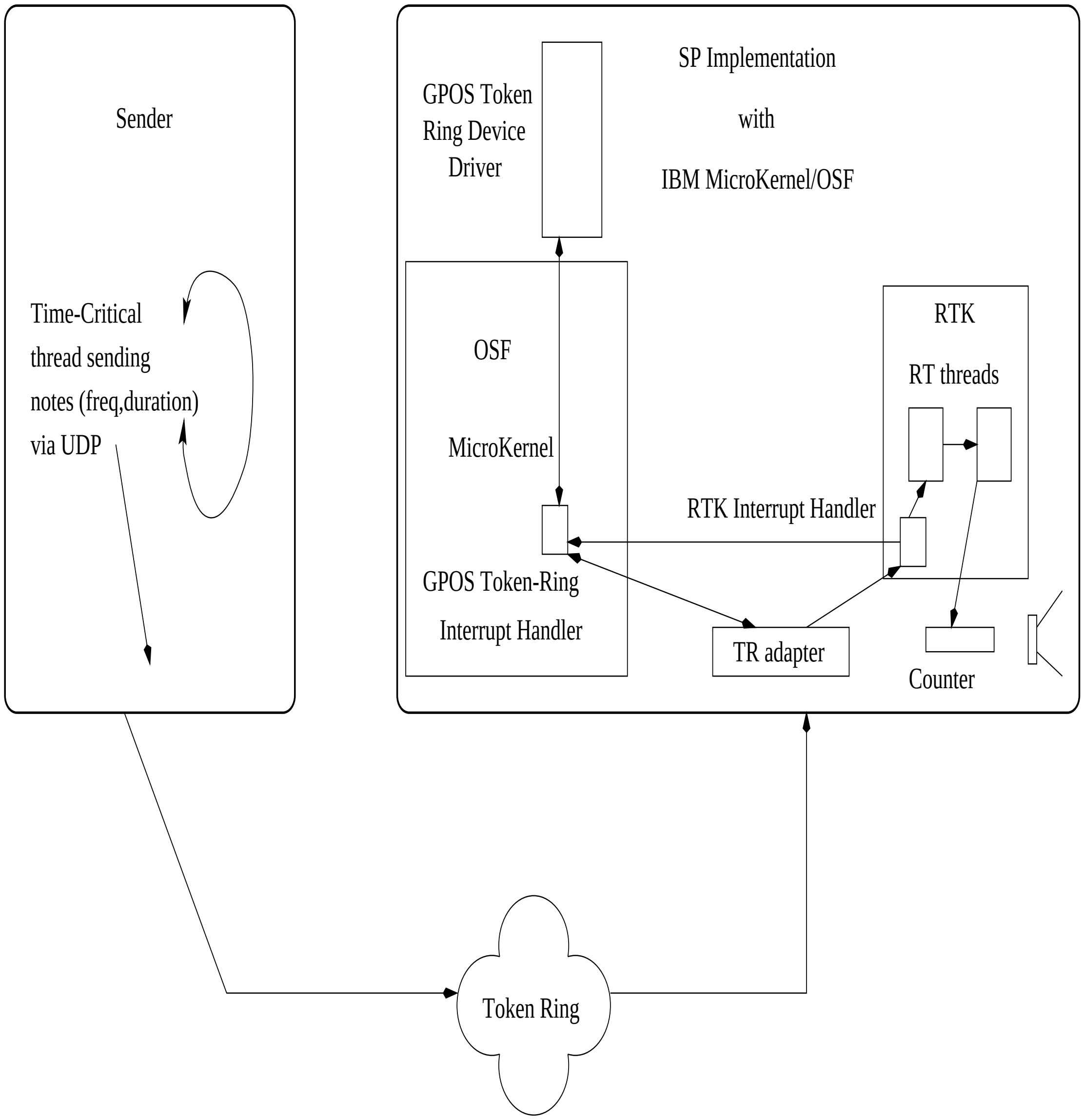

Figure 6.2: Network Audio Demonstration Diagram

their deadlines in all circumstances except one. Deadlines are missed only because during an interaction sequence between the GPOS token-ring device driver and the token-ring device, the GPOS token-ring device driver is not scheduled on the CPU by the GPOS scheduler for about a second. Since the device driver and device are in the middle of a sequence that cannot be interrupted, the RTK device driver cannot interact with the device to receive incoming audio packets. This results in either a missed note or a note played late. That the GPOS device driver is not scheduled correctly is an artifact of how the GPOS is constructed[2]. In such cases, the SP architecture recommends the addition of a separate device dedicated to the RTK and the real-time threads or the complete re-write the device driver so that all device specific code that is needed by the real-time threads runs as real-time threads.

6.3.1 Shared Resources

This application uses four shared resources: the CPU, the network adapter, the 8254 timer chip (in its role as speaker frequency generator), and UDP ports. The CPU is shared by using the same executive as in the character display application.

The User Datagram Protocol (UDP) is a transport level protocol used extensively in the Internet with implementations of the Internet Protocol (IP). A UDP port identifies the particular process waiting to receive a particular datagram among all processes waiting for UDP datagrams. Each machine has a range

[2]In subsequent versions of the IBM MicroKernel the model was changed to allow correct scheduling of the token-ring device driver.

of UDP port numbers. This demonstration shared the available UDP ports by partitioning. A subset of the ports was allocated to the RTK to dispense to any real-time threads wishing to use UDP; the rest remained with the IP implementation in the GPOS. Real-time threads in the RTK opened a UDP socket indexed by a UDP port. The RTK kept track of the UDP ports for which a real-time thread had opened a socket.

The 8254 timer chip is also a partitionable resource, but in this application it was shared slightly differently than it was in the character display application. The register in the 8254 used for timing in the character display application was also used to make the speaker produce a sound of a particular frequency. The value in the register was considered the duration of an interval, measured in timer ticks (approximately 838 nanoseconds), between pulses sent to the speaker. Thus to get a 60Hz tone from the speaker, a value of 19,889 would be placed in the register. This value is derived as follows:

$$\begin{aligned}\text{value} &= \frac{1}{60} \div 838 \times 10^{-9} \\ &= \frac{10^9}{60 \times 838} \\ &= 19,889\end{aligned}$$

As in the character display application, the GPOS must always use the second register on the 8254 for its timing needs.

The most difficult instance of resource sharing in the application is the network adapter. This resource—non-preemptible, externally-triggered, stochastic process—is the most difficult to share because its immediate availability to a real-time thread cannot be guaranteed. This is because an external, nonperiodic event may make it busy independently of any action the GPOS or RTK can control. Thus, when the network adapter receives an incoming network packet and a sequence of interactions between the adapter and the GPOS device driver begins, the RTK is locked out until that sequence is complete.

This effect is exacerbated by the structure of device drivers in the early pre-release of the IBM MicroKernel used in this experiment. The network adapter device driver executes in user space and is subject (unintentionally, since this was a development release) to scheduling delays. When the GPOS is loaded and the network is busy, the network adapter device driver may take seconds to complete a transaction with the network adapter. In this case the RTK must wait, since the network adapter will not begin any other transaction until it has received an end-of-interrupt (EOI) from the device driver.

The executive for the network adapter is a modification of the low-level interrupt handler code. The executive checks to see if the current interrupt is from the network adapter; if so, it calls an additional network adapter device driver in the RTK before invoking the GPOS network adapter device driver. The RTK device driver looks on the adapter to see if this interrupt is for an incoming packet. If it is, the driver looks at the packet to determine if it is for a thread in the RTK, as indicated by the partitioning of the UDP ports. If the incoming packet is a UDP packet and has a port number used by a real-time thread, the RTK device driver removes the packet from the adapter and returns to the low-level interrupt handler, which then invokes the GPOS device driver which subsequently throws away the UDP packet[3]. Note that the RTK interrupt handler runs in response to an interrupt from the network adapter even during a non-real-time minor cycle. This is typical of an executive for a non-preemptible, externally-triggered, stochastic-process resource, because work destined for real-time threads may arrive at any time from the external environment.

6.3.2 Real-Time Kernel

The RTK in this application is essentially the same as the one used in the character display application with one important difference: for efficiency, two real-time threads were coded to run at each invocation of the RTK to poll the network adapter and speaker. Their cost is *very* small, only hundreds of nanoseconds. They were implemented in this manner to save task switch time, when the RTK switches to a real-time task, since their periods should be about as long as the interval between invocations of the RTK.

6.3.3 Real-Time Threads

This application contains three real-time threads modelled after the Liu and Layland task model [24]. (An alternate model for these real-time threads is described

[3] An obvious optimization here would be to modify the low-level interrupt handler to determine if the RTK device driver removed the packet and, in that case, simply issue an end-of-interrupt to the device and return, rather than call the GPOS device driver. However, this optimization would not affect the correctness.

in Section 7.2.) The three real-time threads are:

- a thread to poll the network adapter during real-time minor cycles (since the network adapter interrupt is disabled)
- a thread to poll the speaker and turn it off when the current note has been played for its given duration
- a thread to start playing the next note on the speaker if the previous note has completed

The first two threads are not executed within the SP system as traditional periodic threads which typically are executed in real-time minor cycles. Instead, because these two threads have inherently small periods (high frequency) and small costs, they are implemented as part of the RTK.

The first real-time thread in this application polls the network adapter for receive interrupts during real-time minor cycles. During real-time minor cycles interrupts from all devices, except the timing circuit for the slots, are disabled. Thus, were a UDP packet to arrive during a real-time minor cycle, it would not be received until the next non-real-time minor cycle, when all interrupts are again enabled and the GPOS device driver can run. This thread has a period of one slot-time (976 microseconds) and executes at each invocation of the RTK, and determines if the network adapter is signalling that it has received a packet. If so, the thread calls the RTK network adapter device driver which checks if the packet is for a real-time thread and, if so, removes it from the adapter. Thus, this thread allows the system to avoid the delay that could occur if a packet of notes arrived during a real-time minor cycle. This real-time thread can be considered as part of the RTK network adapter device driver and its cost can be accounted for in RTK cost.

The second real-time thread is similar to the first. This thread also executes at each invocation of the RTK to determine if the current note playing at the speaker has finished, and if so, to turn off the speaker.

The third real-time thread, with a period of 30 milliseconds, starts the next note playing on the speaker. If no notes remain to be played it reads the next packet of notes from the communication socket buffer. If the buffer is empty, then the network has delayed the packet, the sending process on OS/2 has experienced a scheduling delay, or the GPOS device driver has had a scheduling delay. These conditions are expected, since no timeliness guarantees are made by either the network or OS/2 and as explained earlier, the GPOS device driver is subject to delays, causing the network adapter to appear busy when needed by a real-time thread.

6.3.4 Results

Table 6.3 shows the results of four tests run with this application in four different environments. The two binary variables were the GPOS and the network, run under high utilization (Active) or low utilization (Quiet) conditions. To force the GPOS to high utilization, a large compilation was run, and for a high utilization network, the system was connected to the production ring at an IBM software development facility. We listened to the audio output to determine if notes were missed or delayed. As the table shows, no notes were missed in three of the four environments.

Notes are missed when a discarded network packet happens to be a packet of notes. Since the sending process does not resend the packet, the audio stream is interrupted and notes are deleted.

Notes can be delayed if the network adapter is busy when the last note of the buffer has been played, preventing the real-time thread from obtaining the next packet of notes. The packet of notes has not been discarded from the adapter, but a scheduling delay in the GPOS device driver holds the adapter in a busy state. When the adapter becomes idle, the real-time thread can read the packet off the adapter and put it in the socket buffer, but the note will be played late.

If the GPOS is active and the network is quiet, no notes are missed or delayed, because the only packets on the network are the packets of notes from the OS/2 machine. That no notes are delayed when the GPOS is active demonstrates that

GPOS	Network	
	Active	Quiet
Active	Missed and Delayed Notes	No Missed or Delayed Notes
Quiet	No Missed or Delayed Notes	No Missed or Delayed Notes

Table 6.3: Notes Missed in Network Audio Demonstration

the notes are played and received by real-time threads without scheduling delays. That no notes are missed in this case demonstrates that because the network traffic is low, the GPOS device driver can remove packets fast enough to avoid discarding packets.

When the GPOS is quiet and the network is active, no notes are missed or delayed. That no notes are missed demonstrates that although the network traffic is high, the GPOS device driver can remove packets from the network adapter fast enough that no packets are discarded. That no notes are delayed demonstrates that the real-time threads are executed at the appropriate times.

The feasibility analysis for the three real-time threads in this demonstration is straightforward and the set of threads is feasible.

6.3.5 Network Audio Demonstration Summary

This application demonstrates that an SP implementation can share diverse resources and guarantee real-time execution of threads using those resources. The CPU, network adapter, 8254 timer chip (in its role as the speaker frequency generator), and UDP ports were the shared resources. Packets of notes were sent from a machine running the OS/2 operating system to the SP test implementation over a token ring LAN. Three real-time threads cooperated to read the notes from the network adapter, start the notes playing on the speaker and stop the sound from the speaker when the current note had played for its indicated duration. All three threads were modelled, in the application, as traditional periodic threads. However, during this work an alternate model and an extension to the SP architecture were developed (see Section 7.2.1). The application was run in three different experiments where the GPOS, the network, or both were busy. In the first two no notes were missed or delayed. However, due to the construction of the GPOS network adapter device driver, notes were missed and delayed when both the GPOS and network were busy.

6.4 Chapter Summary

This chapter describes two real-time applications implemented on the test SP implementation described in Chapter 3. The applications shared various resources with the GPOS and allowed the real-time threads of the applications to complete their computation before their deadlines, even when the system was at high utilization. The first application consisted of two real-time threads that computed a dummy load and then displayed a character on the display device. The second application received packets of musical notes from a token ring network, sent by a non-real-time machine, and played the notes on a speaker. Both applications demonstrated that the test SP implementation can reliably execute real-time threads so that they always meet their deadlines, which validates the analysis in Chapter 4 and shows how applications might be written and how they execute.

Chapter 7

Implications and Summary

7.1 Introduction

In this chapter we present starting points for continuing the investigation into supporting real-time computation within GPOSs and a summary of all of the previous chapters. Section 7.2 describes those starting points. Section 7.2.1 describes in detail how an alternate real-time task model might be supported by the SP architecture. Section 7.2.2 describes the discrepancy we believe to exist in the hardware used for the implementation and demonstrations. Section 7.3 is the summary. We draw the primary conclusion from this work in the final section.

7.2 Further Research

Useful and direct implementation extensions of this work include:

1. Build device executives for other devices.

2. Design and implement real-time utilities that real-time threads can use in the real-time kernel.

3. Implement the SP architecture in other hardware and GPOS combinations for testbeds.

Useful and direct research extensions of this work include:

1. Extend the investigation of varying minor cycle lengths while holding the ratio $\frac{mc_{rt}}{MC}$ constant to provide system configuration parameters that could be used for tuning an implementation for efficiency.

2. Implement the SP architecture on a variety of other hardware and GPOS combinations to determine its robustness and to illustrate possible useful changes.

3. Implement the SP architecture within the extensible kernel model of operating system design to provide easier implementation.

4. Investigate the use of real-time task models other than this periodic model to provide more general support for real-time computation (see Section 7.2.1).

5. Investigate the use of other real-time thread schedulers in the RTK to provide more general real-time computation.

6. Extend the investigation of varying minor cycle lengths while holding the ratio $\frac{mc_{rt}}{MC}$ constant to better understand the relationship between real-time computation models and shared CPUs.

7. Examine classes of non-real-time applications, such as communications congestion control algorithms, graphical user interfaces, network management

utilities, to see if they are more efficient when redesigned using the real-time thread scheduling paradigm.

8. Design interfaces for attached physical devices that support multiple, simultaneous sequences of interactions from multiple device drivers.

9. Design interfaces for attached physical devices that allow efficient preemption of sequences of interactions.

The next section looks at another real-time thread model.

7.2.1 An Alternate Real-Time Thread Model

To look at future research and extensions of the basic SP model, consider the generic pseudocode shown in Figure 7.1 and the possible variation in intervals between consecutive calls to the `Compute` statement when executed on a typical GPOS (without SP).

```
BEGIN
   Do Forever
      Compute
      SleepValue = ComputeNewSleepValue ( )
      Sleep ( SleepValue )
   End Do
END
```

Figure 7.1: Typical Real-Time Thread Loop

Define a new minor cycle called a sporadic minor cycle which may occur in either a real-time minor cycle or non-real-time minor cycle. A sporadic thread is defined as a tuple, $s_i = (e_i, q_i)$, where e_i equals the processor units required by sporadic thread s_i in an interval of no shorter than q_i. Require that for all i, $q_i > mc_{rt}$ to ensure that no sporadic thread will be invoked more than once in any real-time minor cycle. Sporadic threads must be accounted for in the feasibility analysis of the periodic real-time threads so for each sporadic thread, add a periodic real-time thread $\tau_k = (c_k, p_k)$, where $c_k = e_i$ and $p_k = MC$. Thus any sporadic thread will be invoked only once in a real-time minor cycle and its use of processor cycles will be accounted for in the feasibility analysis. The feasibility analysis is not affected by invocations of sporadic threads in the non-real-time minor cycle.

Now consider the two periodic real-time threads of the network audio demonstration application that start and stop the notes played on the speaker. Since notes are played for varying amounts of time, periodic real-time tasks would not be as efficient since they have to poll for the correct times to start the notes playing. Instead, the real-time threads could be modelled as a single sporadic thread shown by the following pseudocode.

```
BEGIN
   Do Forever
      If ( No_More_Notes )
      Then
             Notes = Read_Note_from_Socket_Buffer ( )
             i = 0;
      End
      Start_Note_Playing_on_Speaker ( Notes[i].Pitch )
      Sleep ( Notes[i].Length )
      i = i + 1;
   End Do
END
```

Since an implementation of the SP architecture can precisely and accurately execute the RTK and its threads, only a small modification to the RTK would cause it to invoke itself when the sleep times for the sporadic threads expire. Thus the SP architecture can accommodate sporadic threads, and when they become ready to execute, begin their execution with low latency. One class, rate-based congestion control, of computer communication protocols might benefit from precise execution. Research into these issues is currently ongoing at IBM and the University of North Carolina at Chapel Hill.

7.2.2 RTClock Frequency Discrepancy

As mentioned in Section 5.4.1.1, slots have been measured about 1.0 percent less than is indicated by the documentation of the RTClock. We conjecture that this discrepancy is caused by using the motherboard timing frequency of 1.193 MHz to clock the RTClock rather than the designed input frequency of 1.048576 MHz. Our measurements agree with a computed interval length generated by the RTClock if it were clocked with the motherboard frequency.

Three issues for further research become apparent:

- to investigate the typical design parameters for personal computer class machines to determine the actual tolerances in timer outputs one might expect to find.
- design a methodology for accommodating the known tolerances in a manner transparent to the programmer of real-time threads.
- establish an industry-wide standard reporting framework for personal computer manufacturers to document the tolerances found in their products.

7.3 Summary

This work had several goals:

1. Present an architecture that defines a way to add support for real-time tasks to a general-purpose operating system with minimal modification.

2. Provide a means for proving that an implementation of the architecture was correct.

3. Provide guidelines for implementing the architecture.

4. Provide guidelines for implementing executives.

5. Derive feasibility conditions for determining whether a set of periodic real-time threads, as described by Liu and Layland, executing on an implementation of the architecture and scheduled by an earliest deadline first scheduler would all meet their deadlines.

6. Using the programming model, show example programs written to solve real-time application problems.

The architecture was presented in Chapter 3. Two fundamental requirements were derived from first principles. A general methodology for realizing the requirements and a method for arguing that an implementation has realized those requirements are given. Three models of the architecture—the execution model, the resource model, and the programming model—were described.

The execution model specifies the interaction between the CPU and the two operating system kernels as was shown in Figure 3.1. The GPOS and the RTK share the CPU by executing in alternate intervals called real-time minor cycles and non-real-time minor cycles. Within this model, the SP architecture specifies the design of three concepts

- the method by which the CPU is shared between two operating system kernels, (the GPOS and the RTK)
- the component that manages the sharing, the CPU executive;
- a requirement to ensure precise minor cycle lengths and bounded real-time thread execution times.

The CPU executive determines when the two types of minor cycles begin from a sequence of interrupts from a hardware timer and invokes its dispatcher to dispatch either the GPOS or the RTK scheduler. The architecture also requires that the interrupts that mark the start of the minor cycles are not indefinitely delayed by any action of the GPOS, the GPOS device drivers, or the attached devices.

The resource model assures that the SP architecture can provide useful guidelines for constructing executives that share resources. The alternative would be the impossible task of providing details on every one of the overwhelming number of devices and device interfaces available today. Fortunately, similar executives can manage the sharing, between the RTK and GPOS, of resources with similar characteristics. For example, an executive similar to the executive described in the execution model can manage the sharing of a resource similar to the CPU; that is, one supporting preemption with state saving and restoration.

The programming model defines how programmers write and think about real-time programs in an implementation of the SP architecture. Real-time threads are threads of traditional GPOS processes that are scheduled by the RTK and execute in real-time minor cycles. The implementation supports the creation of periodic real-time threads via operating system calls to the GPOS and RTK. These real-time threads then execute in a continuous loop with one wait statement per loop.

An implementation also supports and requires the reservation of resources. Each real-time thread reserves a resource it intends to use within its continuous loop. How a resource is used is somewhat independent of its class in the resource model. The CPU does not have to be explicitly reserved by the programmer, but is reserved by the call to create a real-time thread. The underlying executive and execution model ensure that an instance of any reserved resource is available, within some reasonable upper time bound, when needed by the real-time thread.

Feasibility conditions are derived and proved in Chapter 4. These are computationally tractable expressions that allow an implementation to determine if a real-time thread requesting admission into the active set of real-time threads will cause any of the real-time threads to miss their deadlines. Thus, the implementation can either expand the real-time minor cycle or refuse admission.

Chapter 5 describes an implementation of the architecture and presents the result of a series of experiments conducted to determine if the implementation correctly realized the two requirements stated in Chapter 3.

Chapter 6 empirically validates a feasibility analysis on a set of real-time threads and demonstrates the programming paradigm and the use of the system. Two real-time applications were designed and programmed on the implementation. The first displayed a character on the display device in each period of the real-time threads. The second received an audio stream from the network and played the audio on the system speaker. All of the real-time threads of the two applications met their deadlines. This chapter also introduces the notion of a sporadic real-time task and section 7.2.1 shows how the SP architecture can support other models of real-time computation. Initially, the audio real-time application used periodic real-time tasks but it was noted that the function of the demonstration could have been better implemented with sporadic real-time tasks. Thus, two demonstrations highlight the correct execution of periodic and sporadic real-time tasks and show the sharing of resources (beyond the sharing described in Chapter 5 for the experimental test loops) as defined by the SP architecture.

7.4 Conclusion

In Chapter 1 we stated the following thesis:

The slotted priority architecture defines a way to modify general-purpose operating systems to schedule, manage, and execute periodic real-time tasks. The resulting modified system will be able to guarantee that all periodic real-time tasks active on the system will meet their stated deadlines. Periodic real-time tasks may reserve resources, such as direct access storage device access and network access, and the modified system will guarantee that the resources are available when needed by the reserving tasks with a reasonable maximum latency. The architecture methodology also provides that modifications to the original general-purpose operating system will be modest.

We have described such an architecture; shown that the architecture can be implemented on typical personal computer hardware; shown that the completion times of a feasible set of real-time threads can be predicted; shown that real-time threads can reserve resources such as the speaker and network adapter; and shown that all of these tasks can be accomplished with minimal modifications to the original general-purpose operating system.

We conclude that the thesis has been proven.

BIBLIOGRAPHY

[1] Anonymous. Atlas, a new concept in large computer design. *Commun. ACM 3* (June 1960), 367–368.

[2] Baker, T. P., and Shaw, A. The cyclic executive model and ada. *The Journal of Real-Time Systems 1*, 1 (June 1989), 7–25.

[3] Brooks, F. P. Personal communication. Advanced Architecture Class, 1990.

[4] Carpenter, B., Roman, M., Vasilatos, N., and Zimmerman, M. The rtx real-time subsystem for windows nt. In *The USENIX Windows NO Workshop Proceedings* (Aug. 1997), pp. 33–38.

[5] Deitel, H., and Kogan, M. *The Design of OS/2.* Addison-Wesley, Reading, MA, 1991.

[6] Denning, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[7] Denning, P. J. Virtual memory. *ACM Comput. Surv. 2*, 3 (Sept. 1970), 153–189.

[8] Engler, D. R., Kaashoek, M. F., and Jr., J. O. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles* (Dec. 1995).

[9] Ford, B., Hibler, M., Lepreau, J., Tullmann, P., Back, G., and Clawson, S. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 1996), pp. 137–151.

[10] Garey, M., and Johnson, D. *Computers and Intractability.* Freeman, San Francisco, California, 1979.

[11] Govindan, R., and Anderson, D. P. Scheduling and IPC mechanisms for continuous media. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Oct. 1991).

[12] Heath, C., and Rosch, W. L. *The MicroChannel Architecture Handbook.* Simon and Schuster, Inc., New York, NY, 10023, 1990.

[13] IBM. *The IBM Personal System/2 Hardware Interface Technical Reference*, 1 ed., 1988.

[14] IBM. *The IBM Personal System/2 Model 95 XP 486 Technical Reference*, 4 ed., 1992.

[15] IBM Corporation. *IBM Virtual Machine Facility /370 Planning Guide (GC20-1801-0)*, 1972.

[16] IBM Corporation. *IBM Virtual Machine Facility /370 Release 2 Planning Guide (GC20-1814-0)*, 1973.

[17] Intel Corporation. *Intel486^{TM} Microprocessor Family Programmer's Reference Manual.* Mt. Prospect, IL, 1993.

[18] Intel Corporation. *Peripheral Components.* Mt. Prospect, IL, 1993.

[19] Jeffay, K., and Bennett, D. A rate-based execution abstraction for multimedia computing. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video* (Apr. 1995).

[20] Jeffay, K., and Stone, D. L. Accounting for interrupt handling costs in dynamic priority task systems. Not Published.

[21] Jones, M. B. Adaptive real-time resource management supporting modular composition of digital multimedia services. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Nov. 1993).

[22] Jones, M. B., Leach, P. J., Draves, R. P., and Barrera, J. S. Support for user-centric modular real-time resource management in the rialto operating system. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video* (Apr. 1995).

[23] Little, J. D. A proof of the formula $l = \lambda w$. *Operations Research 9*, 3 (May-June 1961), 383–387.

[24] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM 20*, 1 (Jan. 1973), 46–61.

[25] Mercer, C. W., Savage, S., and Tokuda, H. Processor Capacity Reserves for Multimedia Operating Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems* (May 1994).

[26] Motorola. *Motorola Semiconductor Technical Data.* Motorola Design-Net 602-244-6591.

[27] Nieh, J., Hanko, J. G., Northcutt, J. D., and Wall, G. A. Svr4unix scheduler unacceptable for multimedia applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Nov. 1993).

[28] Nieh, J., and Lam, M. S. Integrated processor scheduling for multimedia. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video* (Apr. 1995).

[29] Pardyak, P., and Bershad, B. N. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 1996), pp. 201–212.

[30] Peterson, J. L., and Silberschatz, A. *Operating System Concepts.* Addison Wesley, Reading, Massachusetts, 1983.

[31] Ramamurthy, S., Moir, M., and Anderson, J. H. Real-time object sharing with minimal system support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing* (May 1996), ACM, pp. 233–242.

[32] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Softw. Eng. 39*, 9 (Sept. 1990), 1175–1185.

[33] Shaw, A. C. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng. 15*, 7 (July 1989), 875–889.

[34] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S. K., Gehrke, J. E., and Plaxton, C. G. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium* (Dec. 1996), pp. 288–299.

[35] Tanenbaum, A. S. *Computer Networks.* Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1981.

[36] Tokuda, H., Nakajima, T., and Rao, P. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of USENIX Mach Workshop* (Oct. 1990), pp. 73–82.

[37] Waldspurger, C. A., and Weihl, W. E. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 1994).