Operating Systems Mechanisms For Continuous Media

Ramesh Govindan

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94709

Support for digital audio and video (*continuous media*) as part of the human/computer interface is an important direction for computer systems research. There are various ways to incorporate continuous media (CM) in computer systems; in the *integrated* approach, CM data is handled by user-level programs on general-purpose operating systems such as Unix or Mach.

Integrated CM applications handle data at high rates, with strict timing requirements and often in small "chunks". Conventional operating systems support for program execution and local communication may be non-optimal for such applications. In particular, conventional mechanisms for process scheduling and inter address space communication can add significant overhead for some CM programs. *User/kernel interactions*, by which user-level programs invoke system functions, are partly responsible for this overhead.

We describe new mechanisms for process scheduling and CM stream communication between virtual address spaces. These mechanisms, *split-level scheduling* and *memory-mapped streams*, reduce or eliminate user/kernel interactions by using user/kernel shared memory for exchanging scheduling and communication information. We demonstrate that, compared with conventional mechanisms for process scheduling and stream communication, these new mechanisms can reduce overhead by up to a factor of four.

ACKNOWLEDGEMENTS

A number of people have been instrumental in shaping this dissertation and influencing the direction of my career.

To David Anderson, my thesis advisor, I am immeasurably grateful. He has instilled in me a research discipline that, left to myself, I might not have inculcated. His constant emphasis on effective communication has greatly improved my writing and verbal presentation skills. His patience with my sometimes monstrous stupidity and ineptness still continues to baffle me. Those competitive Friday afternoon tennis games helped sharpen my tennis game considerably. Thank you, Dave.

Prof. Domenico Ferrari was responsible for introducing me to the DASH group in my early days as a graduate student. He has been generous with his time whenever I needed advice. His patient and thorough reading of this dissertation also helped improve the quality of the presentation.

Prof. David Wessel has always maintained a great interest in my work with continuous media. His interest in Indian classical music provided a focal point for some interesting discussions.

A number of colleagues have contributed to my intellectual development, either directly with advice or help or indirectly as role models. Among them: Ramon Caceres, Riccardo Gusella, Srinivasan Keshav, Puneet Kumar, Steve Lucco, Mark Moran, Stuart Sechrest, Shin-Yuan Tzou, Dinesh Verma, Robert Wahbe, and Hui Zhang. Members of the ACME group have directly or indirectly contributed to this dissertation: Eric Barr, Pamela Chan, Cindy Hertzer, George Homsy, Yukkei Hui, Yoshi Osawa, and Kyoji Umemura.

I am very deeply indebted to my grandfather S. Lakshminarasimha Sastry and to my mother Indira for the gift of music. Music provided much needed solace during trying times and helped preserve my sanity in this sometimes difficult world.

A number of people have supported and guided me during these years as a graduate student. Ramach Gurumoorthy and Sekhar Raghavan have been kind and caring friends. Sekhar and Ramach, thanks for those long and animated discussions that brightened many an evening. I'll miss those wonderful Sunday evening music sessions with Ramach and Swaminathan Gopalswamy. M. T. Raghunath and I go a long way back; I shall long cherish his wacky sense of humor and his warm personality. Ravikumar Ramachandran and Sridhar Srinivasan, thanks for those great dinners and juicy gossip sessions. Shanthi, my wife of two months, thanks for those lovely letters and phone calls that kept me going for the last year.

Finally, I'd like to thank my parents and my brother whose incredible support from across the oceans kept me going for these years. Without them, this work would not exist.

TABLE OF CONTENTS

1. **INTRODUCTION** 1
   1.1. Motivation 1
   1.2. Thesis Statement and Contributions 3
   1.3. Thesis Overview 3

2. **INTEGRATED CONTINUOUS MEDIA** 5
   2.1. What is Integrated Continuous Media? 5
      2.1.1. Hardware and Software Framework 5
      2.1.2. The Integrated Continuous Media Framework 5
   2.2. Requirements of Integrated CM Applications 6
      2.2.1. Example: The CM File Playback Application 7
      2.2.2. Example: Audio Teleconferencing 7
      2.2.3. Delay and Throughput Requirements of Integrated CM Applications 7
   2.3. Survey of Integrated CM Systems 8
      2.3.1. End-to-end scheduling 8
         2.3.1.1. The CM-Resource Model 9
            2.3.1.1.1. Describing CM Workload and Delay 10
            2.3.1.1.2. Sessions and Compound Sessions 10
         2.3.1.2. The Producer/Consumer Paradigm 12
         2.3.1.3. The Sun Approach 12
      2.3.2. CM Storage 13
         2.3.2.1. Structural Issues in CM File Storage 13
         2.3.2.2. Real-time Storage and Retrieval of CM files 13
      2.3.3. CM I/O 13
      2.3.4. Network Communication 14

3. **OPERATING SYSTEM SUPPORT FOR INTEGRATED CM APPLICATIONS** 15
   3.1. Reexamining OS Policies and Mechanisms 15
   3.2. Operating Systems Policies 15
      3.2.1. Deadline/Workahead Scheduling 16
   3.3. Operating Systems Mechanisms 17
      3.3.1. Mechanisms For Process Scheduling and Stream Communication 17
         3.3.1.1. CM Task Structure Examples 18
         3.3.1.2. Conventional Approaches for Process Scheduling and Message Transfers 20
         3.3.1.3. Domain Switches And Mapping Switches 20
   3.4. New Mechanisms for Process Scheduling and Stream Communication 21

4. **SPLIT-LEVEL SCHEDULING AND SYNCHRONIZATION** 22
   4.1. Overview of Split-Level Scheduling 22

4.2. Client Interface to the Split-Level Deadline/Workahead Scheduler 22
4.2.1. LWP Creation and Deletion 22
4.2.2. LWP State Manipulation 23
4.2.3. LWP Synchronization 24
4.3. Implementation of the Split-level Deadline/Workahead Scheduler 24
4.3.1. Terminology and Notation 24
4.3.2. User/Kernel Interface 25
4.3.2.1. System calls 25
4.3.2.2. User-interrupts 26
4.3.2.3. User/Kernel Shared Memory Interface 27
4.3.3. ULS Implementation 27
4.3.4. KLS Implementation 28
4.3.5. Implementing Synchronization 30
4.3.6. Extensions to Split-Level Scheduling 30
4.3.6.1. Different policy for workahead processes 30
4.3.6.2. Adding support for non-real-time policies 31
4.3.6.3. Different CPU scheduling policy 31
4.3.7. UNIX Implementation of SLS 31

5. MEMORY-MAPPED STREAMS 34
5.1. Motivation 34
5.2. Interface to Memory-Mapped Streams 35
5.3. Implementation of Memory-Mapped Streams 37
5.3.1. User/Kernel Interface 37
5.3.1.1. User-kernel shared memory 37
5.3.1.2. System Calls and User-interrupts 38
5.3.2. Library Implementation 39
5.3.3. Kernel Implementation 41
5.3.4. UNIX implementation of MMS 41

6. SHARED MEMORY CONCURRENCY CONTROL 44
6.1. User-mode Critical Section Violations 44
6.2. Preventing User-interrupts During Process-level Execution 44
6.3. Concurrency Control Between User-mode and Kernel-mode Execution 47
6.3.1. Preemption Masking 47
6.3.2. Multi-word Reads 48
6.3.3. Multi-word Writes 48
6.3.4. Read Followed By Related Write 49

7. PERFORMANCE OF SPLIT-LEVEL SCHEDULING AND MEMORY-MAPPED STREAMS 51
7.1. Scheduling Paths and Their Costs 51
7.1.1. Thread Scheduling Path Costs 52
7.1.2. LWP Scheduling Path Costs 52
7.1.3. SLS Scheduling Path Costs 53
7.1.4. Discussion 53
7.2. I/O Paths and Their Costs 53

7.2.1. Read/Write System Call I/O Path Costs 54
7.2.2. Asynchronous I/O Path Costs 54
7.2.3. MMS Path Costs 54
7.2.4. Discussion 55
7.3. Performance Under Synthetic Workloads 55
7.3.1. Methodology 56
7.3.2. Simulation Results 56
7.3.2.1. Varying the Number of Processes 56
7.3.2.2. Varying the Message Rate 57
7.3.3. Varying Process Delay Bounds 57
7.3.4. Varying I/O Buffer Sizes 59
7.3.5. Varying the Number of VASs 60
7.3.6. Varying workahead policy 61

8. RELATED WORK 63
8.1. Efficient Process Scheduling 63
8.2. Efficient Stream Communication 64
8.3. OS Support for CM 64
8.4. Related Trends in Operating Systems 65

9. CONCLUDING REMARKS 66
9.1. Thesis Contributions 66
9.2. Summary of Results 66
9.3. Future Work 67

LIST OF FIGURES

1.1. An integrated CM application 2
2.1. The CM file playback application 6
2.2. An audio teleconference 8
2.3. Different portions of an integrated CM system 9
2.4. Message arrival time and workahead 11
3.1. Deadline/workahead scheduling 17
3.2. The ACME server task structure 18
3.3. The mixer task structure 19
4.1. Split-level scheduling 23
4.2. Two parts of a split-level scheduler 25
4.3. Illustration for Claim 4.1 28
4.4. Illustration for algorithm in Section 4.3.3 29
4.5. UNIX implementation of SLS 32
5.1. Memory-mapped stream interfaces 35
5.2. The MMS data part 39
5.3. Illustration of KU stream transfer 42
5.4. UNIX Implementation of MMS 43
6.1. User-mode critical section violations 45
6.2. Virtual user-interrupt masking 46
6.3. Critical section 47
6.4. Multi-word reads 49
6.5. Missed I/O notifications 50
7.1. Varying the number of processes 57
7.2. Varying message rate 58
7.3. Varying process delay bounds 59
7.4. Varying I/O buffer sizes 60
7.5. Varying the number of VASs 61
7.6. Varying workahead policy 62

LIST OF TABLES

4.1. Client interface to the split-level scheduler 24
4.2. User/kernel interface for SLS 26
5.1. Client interface to MMSs 37
5.2. The MMS user/kernel interface 40
7.1. Scheduling path costs 54
7.2. I/O path costs 55

Chapter 1

INTRODUCTION

The topic of this dissertation is operating systems mechanisms for digital audio and video (*continuous media*). We consider operating systems (OSs) that provide protected virtual address spaces for user program execution. We argue that mechanisms in these OSs may be non-optimal for user programs that handle CM data (CM programs). In particular, CM programs may incur high overhead when implemented using conventional approaches for process scheduling and CM data transfer between address spaces. We propose new mechanisms for process scheduling and CM data communication and show that they can reduce overhead significantly for CM programs, compared to conventional approaches.

The research area of this work is Operating Systems. It is also related to Multimedia Systems, an emerging discipline which discusses software and hardware issues in supporting diverse media types (still images, animation, digital audio, full-motion video and so on) in computer systems.

The first chapter briefly motivates the dissertation, states its thesis and its contributions, and gives an overview of its structure.

1.1. Motivation

Historically, the addition of new elements to the human/computer interface has necessitated hardware and software changes in computer systems. For instance, the addition of pointer-based interaction required the development of workstations with a bitmapped display and a mouse. Significant additions were needed in terms of software: network-transparent window servers [2, 4] and application-side user-interface toolkits [3] evolved gradually and with considerable discussion. However, this software placed few new requirements on operating systems or networks; it could be layered on existing systems such as MS-DOS and UNIX [1].

The development of CM I/O devices (*e.g.*, digital-to-analog convertors) and the trend towards faster processors and high-speed networks will soon make it feasible to incorporate CM into the human/computer interface in an integrated manner. In an *integrated CM system*, CM data is handled in the same hardware and software framework as discrete media (such as text and graphics). This framework is characterized by a computing environment distributed on an internetwork of LANs and WANs. Hosts on this internetwork run general-purpose operating systems (such as UNIX or Mach). Applications may communicate discrete media over the network; to support such applications, the software framework has elements like network file servers, network-transparent window systems and so on.

Integrated CM applications (like the video file playback application in Figure 1.1) have real-time requirements that are different from those of discrete media applications. For instance, an integrated CM application may need guaranteed minimum throughput, depending on the data rate of the CM stream that it handles. This requirement can range from tens of Kbps to tens of Mbps. The application may also need a bound on the end-to-end delay of the CM stream (in Figure 1.1, this is the time taken for a chunk of video data to be moved from the disk to the display). The required bound can range from a second down to tens of milliseconds.

While attempting to integrate CM into the human/computer interface, researchers have encountered the following questions. The answers to these questions have required (or will require) changes or additions to the existing discrete media framework.

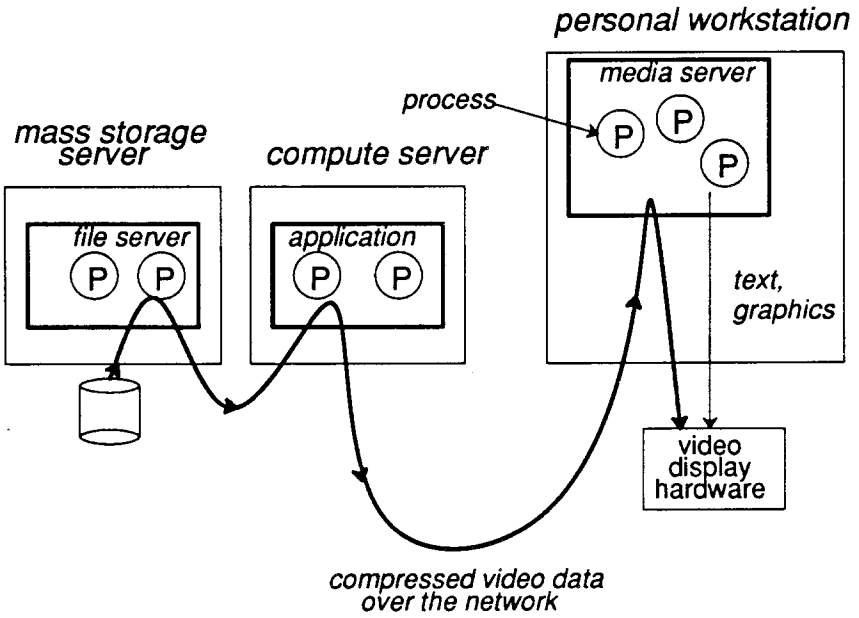


Figure 1.1. An integrated CM application.

In this application, a video file is retrieved from a file server and transmitted on the network to a compute server. The compute server performs some processing on the video data and then sends it to the media server, which outputs the data on the local display.

(1) What parametrization of user requirements is general enough (sufficient for a large range of applications) and accurate enough (gives adequate information so that resources can be used efficiently) for CM? Should an integrated CM system attempt to provide performance guarantees to CM applications (the hard real-time approach)? Or, should it only schedule resources intelligently and gracefully degrade when resource limits are reached (a soft real-time approach)? In Figure 1.1, the file traverses different hardware components (the disk, the file server CPU, the network etc.). How do these different components collaborate to ensure that application requirements are met end-to-end?

(2) CM files are characterized by high display rates and large sizes. What are appropriate disk architectures for storing these large files? What file system data structures permit efficient manipulation (sharing, composition) of CM files? What are good file layout policies and disk head scheduling algorithms for reading CM files from disks at the display rate?

(3) In the existing discrete media framework, window servers provide network-transparent, concurrent access to I/O devices. What are the appropriate abstractions for accessing CM I/O devices in a similar manner? How should such a media server (Figure 1.1) provide synchronization between different media (*e.g.*, lip synch, video sub-titling)?

(4) How can existing packet switched networks transport CM data in real time? What are the network issues (*e.g.*, congestion control, routing) that arise in this new framework? What are appropriate transport-level protocols for CM data? How are such protocols efficiently implemented?

(5) In the integrated CM framework, user programs can handle CM data. How can operating systems satisfy the real-time processing requirements of such programs? Are conventional operating systems mechanisms still appropriate for CM programs?

Existing and ongoing work has attempted to answer some of the questions posed in items (1) through (4) (Chapter 2). This work attempts to answer some of the questions raised by item (5).

1.2. Thesis Statement and Contributions

Some CM programs consist of multiple processes. A CM process may communicate CM data with a process in another address space or with the kernel (we call this *CM stream communication*). Our thesis is that conventional mechanisms for process scheduling and stream communication are inefficient for CM programs, and that new techniques can significantly improve performance.

The contribution of this thesis is a new set of efficient mechanisms for process scheduling and stream communication on general-purpose OSs. These new techniques arise from a reevaluation of conventional approaches in light of CM program requirements.

Conventional approaches to process scheduling:
With conventional approaches, process scheduling is either implemented entirely in the OS kernel or at the user level. The former requires a kernel intervention (*e.g.*, a system call) for each process switch. With the latter, it is not possible to prioritize a process in one address space relative to processes in other address spaces.

New approach to process scheduling:
This approach splits the functionality of scheduling between user and kernel levels, to simultaneously achieve cheap process switches and global process prioritization. User-kernel shared memory conveys scheduling information between the two levels; this avoids kernel intervention whenever possible.

Conventional approaches to stream communication:
Stream communication is implemented as a sequence of data transfers between user address spaces or between user and kernel address spaces. Conventional approaches require kernel intervention for each data transfer; that is, each data transfer operation incurs one or more system calls. The kernel interface allows both synchronous (relative to user programs) and asynchronous data transfers.

New approach to stream communication:
Stream communication is implemented at the user level using shared memory between the communicating address spaces. Communication is asynchronous relative to user programs and the use of shared memory reduces or eliminates system calls.

This dissertation explores these new approaches, validating the thesis by design, implementation and performance evaluation.

1.3. Thesis Overview

Chapter 2 describes the integrated approach in greater detail. It surveys work in different aspects of designing an integrated framework for CM, sampling research projects to see how they answer some of the questions posed in Section 1.1.

Chapter 3 discusses operating system related issues for an integrated CM framework. It motivates the thesis stated in Section 1.2.

Chapter 4 and Chapter 5 describe the design and implementation of new mechanisms for process scheduling and stream communication respectively. Both these techniques use user/kernel shared memory to communicate information. Chapter 6 describes efficient user/kernel shared memory concurrency control mechanisms.

Chapter 7 presents the results of a series of simulations to study the performance of these mechanisms. It describes CM workload characterization and an experimental study of the effect of different factors on the scheduling and communication costs for CM workloads.

Finally, Chapter 8 presents related work in the area and Chapter 9 offers some concluding remarks.

Chapter 2

INTEGRATED CONTINUOUS MEDIA

In this chapter, we describe the *integrated* approach to incorporating CM into computer systems. In this approach, CM data is handled in the same hardware and software framework as other (discrete) data. Section 2.1 lists the properties of an integrated CM system. Section 2.2 discusses the requirements of integrated CM applications. These requirements differ from those of discrete media applications. This difference affects parts of the existing software framework such as network communication, operating systems, and so on. Section 2.3 surveys work on adding CM functionality to these parts.

2.1. What is Integrated Continuous Media?

We first describe our model of the existing framework for handling discrete media data (*e.g.*, text and graphics). We then list desirable properties of an integrated CM framework and compare it with other approaches to incorporating CM.

2.1.1. Hardware and Software Framework

In our model of the existing hardware framework, general-purpose computing is performed on a collection of personal workstations linked by an internetwork of LANs and WANs. The internetwork may also contain other hosts; file servers for storage, gateways for internetwork communication, compute servers for special-purpose processing and so on.

Each host runs a general-purpose operating system (OS) such as UNIX [RiT74] or Mach [ABB86]. In such systems, the OS kernel runs in privileged mode and provides multiple, protected *virtual address spaces* (VASs) for user program execution. A thread of control in a VAS is called a *process*. Each VAS may contain one or more processes. The term process does not imply a particular implementation technique; processes may be implemented entirely at the user level or by the kernel. *Task* denotes an instance of user program execution; a task consists of the VAS in which the program executes, the processes in that VAS and other resources (*e.g.*, port IDs, file descriptors) allocated at runtime to the program.

In our model of the existing software framework, application tasks may reside on different nodes and communicate over the internetwork. A standard protocol suite is used for network communication (e.g, the Internet protocol suite [Tan81]). Network file servers (*e.g.*, NFS [SGK85]) provide distributed data storage and retrieval. Network-transparent window systems (*e.g.* X11 [ScG86] or NeWS [SSS87]) manage the human/computer interface elements such as the display, the keyboard and the mouse.

2.1.2. The Integrated Continuous Media Framework

If the framework described in Section 2.1.1 has the following additional properties, we say that it supports *integrated continuous media*:

(1) CM data is handled in hardware by primary memory, the I/O subsystem, and networks, and in software by the OS and user programs. DVI [Gre92] is an example of a system that provides this functionality.

(2) Users can run multiple applications concurrently, with no adverse effects from contention for hardware resources.

(3) Application tasks can communicate CM data across the network.

(4) The essential elements (*e.g.*, network-transparent window systems, network file access) of the framework described in Section 2.1.1 are used for CM data storage, processing, communication and I/O as well.

Other approaches to incorporating audio and video in computer systems are possible. In one approach, audio and video data is in analog form. However, audio and video storage and communication are under computer control. Examples include VOX [ABL89] and IMAL [LuD87]. In another approach, CM data is in digital form and may be communicated over a digital communication network, but does not pass through the main memory of the computer. Examples include Pandora [Hop90] and the Xerox Etherphone [ZTS89].

In the integrated CM framework, CM data can be manipulated algorithmically in the same way as other data. Relative to these other approaches, an integrated CM system is therefore more flexible and general. Moreover, hardware may be simpler, since no separate disks or networks are required for CM data.

2.2. Requirements of Integrated CM Applications

Applications that handle digital audio and video may have stringent real-time requirements. For instance, CM data must be produced, processed, and consumed at fixed rates (up to 40 Mbps for uncompressed full-motion video). To understand the range of real-time requirements (Section 2.2.3) of integrated CM applications, we describe two such applications (Sections 2.2.1 and 2.2.2).

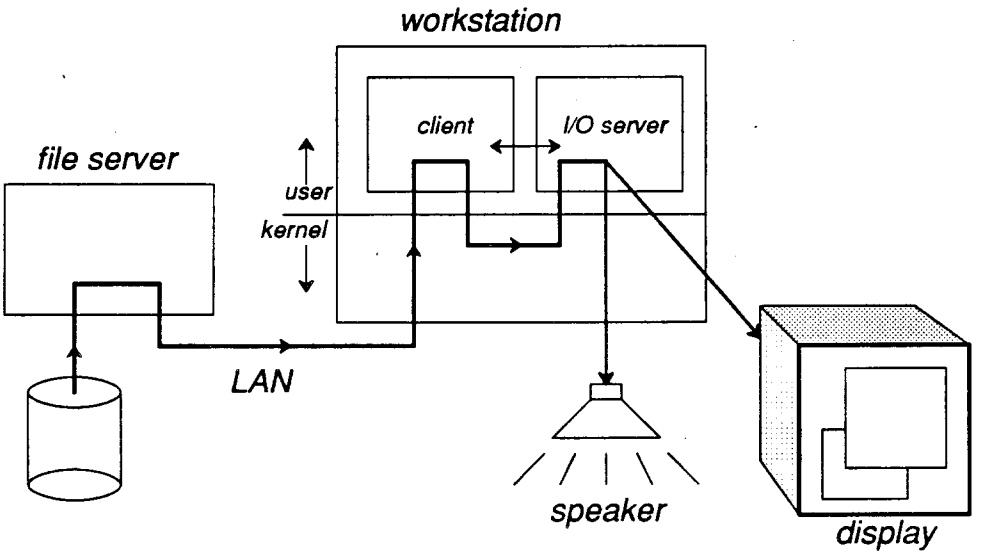


Figure 2.1. The CM file playback application.

This picture shows a program that implements playback of a file containing audio and video data. The program retrieves the file from a network file server, does some processing on the data, and sends the data to a I/O server. The latter outputs the video to the display device and the audio to the speaker.

2.2.1. Example: The CM File Playback Application

Figure 2.1 illustrates an application playing back a file containing a video sequence and its associated audio. An I/O server provides network-transparent, concurrent access to CM output devices (speaker and video display) and to CM input devices (microphone and video camera, not shown). This functionality is similar to that provided by the X11 server for discrete media I/O devices such as the display, the keyboard and the mouse.

A network file server transmits the file as a CM *stream* (a sequence of bytes of CM data) across the LAN. A client of the I/O server may do some processing (*e.g.* audio volume scaling) on the stream. The client task sends the CM stream to the I/O server. The latter separates the audio and video data and sends each component to its respective output device (speaker, display). More generally, the client task may execute on a different workstation from the I/O server.

2.2.2. Example: Audio Teleconferencing

An important class of audio applications enabled by integrated CM is audio teleconferencing. Each participant in a teleconference must hear a sum of the inputs of other participants. The conference may also take input from one or more files; all participants hear file inputs. Similarly, a "transcript" of the conference might be recorded to a file.

In general, a conference has *N* audio sources and *M* audio sinks. Each sink receives a linear combination of the sources, with arbitrary volume coefficients. Some common examples of audio teleconferences are:

N = 2, M = 2: a telephone conversation between two people.
N = 1, M = large: a radio broadcast.
N ~ 10, M ~ 100 : a panel discussion with an audience.
N ~ 10, M ~ 10: a chamber music rehearsal.

When an audio teleconference is implemented in the integrated CM framework, conference sources and sinks can be either I/O servers or file servers distributed across the network. Other hosts on the network may be used to run a third component of the teleconferencing application, "mixers". A mixer combines audio streams from conference sources or from other mixers and distributes the results to one or more sinks or mixers. Mixers, I/O servers and file servers exchange audio streams over network connections.

There are many digital formats for audio, ranging from telephone quality to hi-fi quality. Source and sinks may be able to generate or accept data in more than one format, and mixers may be able to convert between formats.

Given sources and sinks, the audio teleconferencing application computes a configuration of mixers that satisfies the requirements of the conference. Figure 2.2 shows one such configuration for some sources and sinks.

2.2.3. Delay and Throughput Requirements of Integrated CM Applications

Integrated CM applications may require guaranteed minimum throughput. For example, the file playback application (Section 2.2.1) requires that data be read from the disk file and transferred to the DAC at a minimum average data rate. This rate is determined by the data representation.

The data rates for digital audio can vary from 64 Kbps (for telephone quality audio) to 1.4 Mbps (for 44 KHz stereo CD-quality audio) and 1.536 Mbps (for 48 KHz stereo DAT quality audio) [Wat88]. Uncompressed full-motion NTSC video requires a data rate of 40 Mbps. DVI edit-level video [Gre92] compresses this to about 1.2 Mbps. The MPEG standard [Fox91] for motion video compression achieves a maximum data rate of 1.5 Mbps for NTSC quality video and associated audio. However, analogous future technology for HDTV video may produce data rates in the tens of Mbps.

In the file playback example (Section 2.2.1), the CM stream traverses a number of devices from source to sink (disk, file server CPU, network, client CPU and display). At each of these, the CM stream may incur some queueing and processing delay. We call the cumulative delay from

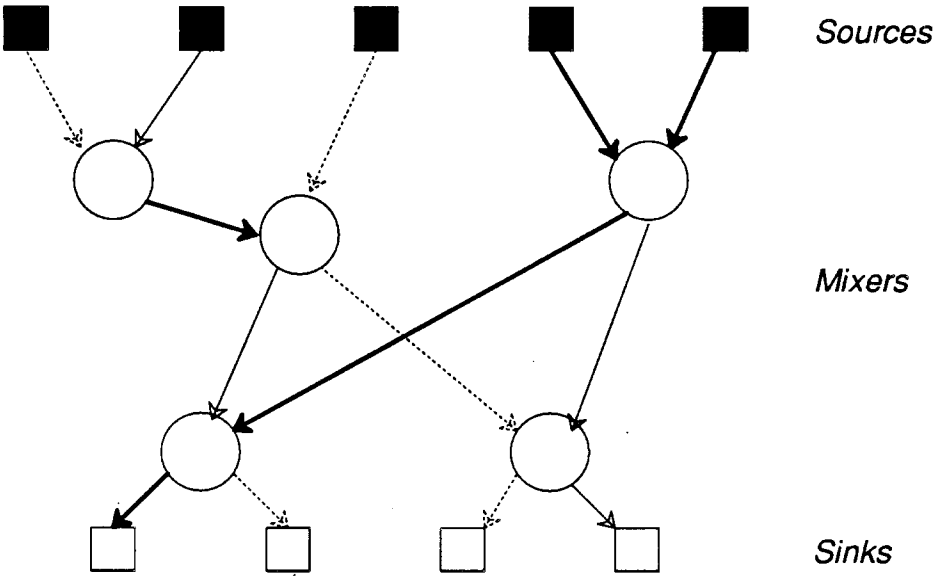


Figure 2.2. An audio teleconference.

A particular configuration of mixers in an audio teleconference might look like this. Arrows represent network connections carrying audio data. A dashed arrow represents low-quality audio data, a thin arrow medium quality and a solid arrow high quality.

source to sink the *end-to-end delay*.

Integrated CM applications may also require bounds on the end-to-end delay of the CM stream. Delay requirements depend on the nature of the application. File playback may be able to tolerate delays in the range of 1-2 seconds. However, audio teleconferencing (Section 2.2.2) requires an end-to-end delay in the range of 100-200 milliseconds [Coh78]. For applications such as distributed music rehearsal (a special case of audio teleconferencing), the required delay may be 20 milliseconds or less [Loy85].

2.3. Survey of Integrated CM Systems

To realize an integrated CM framework, we need to satisfy the four properties described in Section 2.1.2 in the existing discrete media framework. Equivalently, we need to satisfy properties (1) and (4), and satisfy the delay and throughput requirements listed in Section 2.2.3. Figure 2.3 shows portions that should be added to or changed in the existing discrete media framework for this purpose. In this section, we survey existing and ongoing work on: end-to-end scheduling, CM storage, CM I/O, and network communication. The fifth portion, processing and local communication, is the subject of this dissertation and is discussed in subsequent chapters.

2.3.1. End-to-end scheduling

We say that an integrated CM framework solves the *end-to-end scheduling* problem if it can satisfy the throughput and end-to-end delay requirements of integrated CM applications. There are two parts to the problem. The application's requirements must be conveyed to all devices (disks, networks, CPUs and so on) on the path of the CM stream from source to sink. Then, device schedulers must each satisfy the minimum throughput requirement and collectively satisfy the

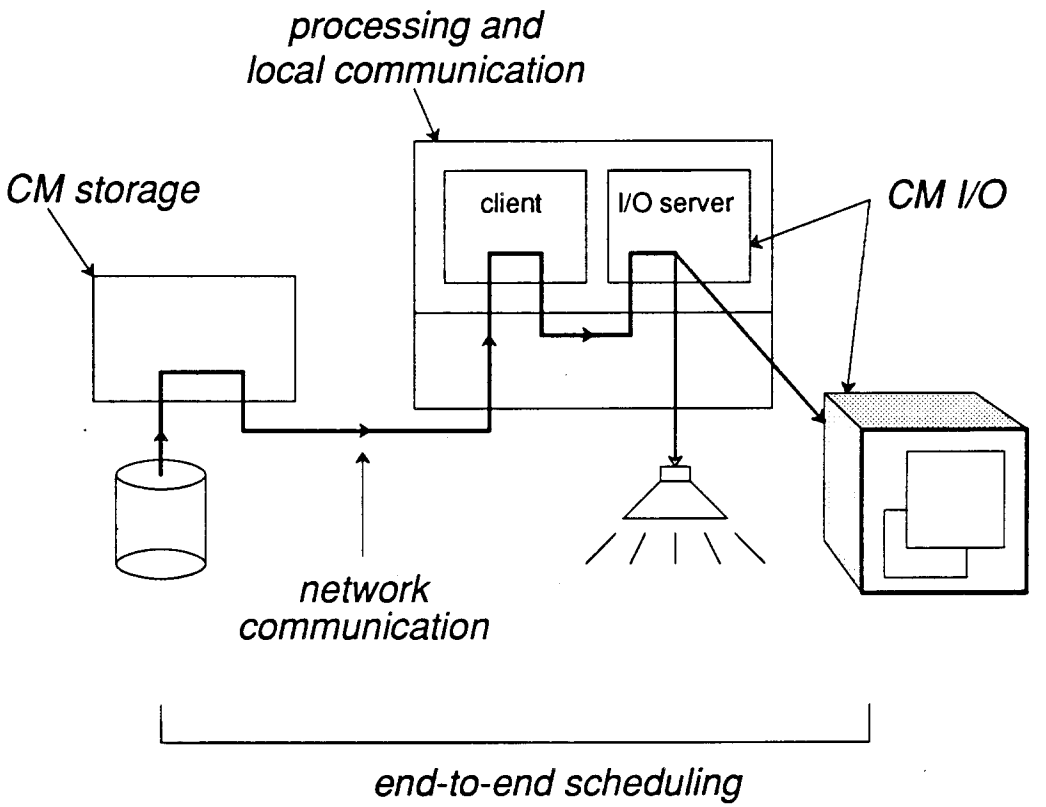


Figure 2.3. Different portions of an integrated CM system.

Taking the file playback example, we highlight five different portions of the software framework described in Section 2.1.1 that are affected by the need to integrate CM.

end-to-end delay requirement.

One approach to this problem reserves device capacity and "guarantees" that an application's requirements will be met over its lifetime (Sections 2.3.1.1, 2.3.1.2). The approach described in Section 2.3.1.3 does not reserve device capacity, but detects and gracefully degrades from device capacity overload.

2.3.1.1. The CM-Resource Model

To guarantee performance levels for the duration of an application's execution, the shared components, such as CPU, file system, and network, may support "reservations". An application specifies its workload and the component reserves part of its capacity to provide the application with a performance guarantee.

To formalize the reservation of component capacity, a model for expressing workload and processing is needed. In this section, we briefly describe the *CM-Resource model* [And]. In this model, the set of system components that handle CM data is decomposed into a set of *resources*. In general, a resource corresponds to a schedulable hardware device and its accompanying software driver. For example, a CPU and its scheduler might comprise a resource. Resources may also be more complex: a local area network (which includes multiple interface devices,

concurrent operation, and multiple scheduling mechanisms) might be treated as a single resource.

The CM-resource model assumes that work is assigned to resources in discrete units called *messages*, typically representing a segment of CM data. Each message has a well-defined *arrival time* at which it is available for handling by a resource and *completion time* at which the handling is finished.

The flow of CM data consists of linear simplex *streams* of messages that pass through one or more resources. Data is generated by a *source resource* (a disk, digitizer, or compression unit), then processed by a sequence of *handler resources* (networks, CPUs, etc.) and finally consumed by a *sink resource* (disk, decompression unit, etc.). A message's completion time in one resource is its arrival time at the next resource. Many of these simplex data streams may exist concurrently, even within a single application. Therefore this scheme encompasses many CM applications: file playback, audio teleconferencing.

2.3.1.1.1. Describing CM Workload and Delay

Each data stream flowing across an interface defines an *arrival process* into the downstream resource. To describe message arrival, the CM-resource model uses *linear bounded arrival processes* (LBAPs). An LBAP has the following parameters:

M = maximum message size (bytes)
R = maximum message rate (messages/second)
W = workahead limit (messages)

that, for all $t_0 < t_1$, satisfies $N_I(t_0, t_1) \leq R\,|t_1 - t_0| + W$, where $N_I(t_0,t_1)$ denotes the number of messages arriving at an interface I in the time interval $[t_0,t_1)$.

The long-term data rate of an LBAP is MR bytes per second. The parameter W allows short-term violations of this rate constraint, modeling programs and devices that generate "bursts" of messages that would otherwise exceed the constraint. These bursts consist of messages that have arrived "ahead of schedule"; they do *not* reflect burstiness in the underlying data stream. The extent to which arrivals are ahead of schedule is quantified by the *workahead* $w(t)$ of an LBAP, defined as

$$w(t) = \max_{t_0 < t} \left\{ 0, N(t_0, t) - R|t - t_0| \right\}.$$

Intuitively, $w(t)$ is the largest "message excess" (relative to R) during any time interval ending at t. More concretely, $w(t)$ is a function that increases by 1 on each message arrival, decreases with slope $-R$ otherwise, and remains nonnegative (see Figure 2.4).

To parameterize the delay between two interfaces in the system, we take workahead into account. For a given LBAP, let $m_0 \cdots m_n$ denote the sequence of messages, and let $a_0 \cdots a_n$ denote their arrival times. The *logical arrival time* $l(m_i)$ of a message m_i is defined as $l(m_i) = a_i + w(a_i)/R$ Intuitively, $l(m)$ is the earliest time message m could have arrived if workahead were not allowed (see Figure 2.4; note that the logical arrival times of consecutive messages are separated by at least $1/R$.)

We define the *logical delay* $d(m)$ of a message m between two interfaces I_1 and I_2 to be the difference between the logical arrival times of m at those interfaces.

2.3.1.1.2. Sessions and Compound Sessions

Prior to using a resource, an application must create a *session* with the resource. Each message handled by a resource is associated with a particular session. A session has the following parameters:

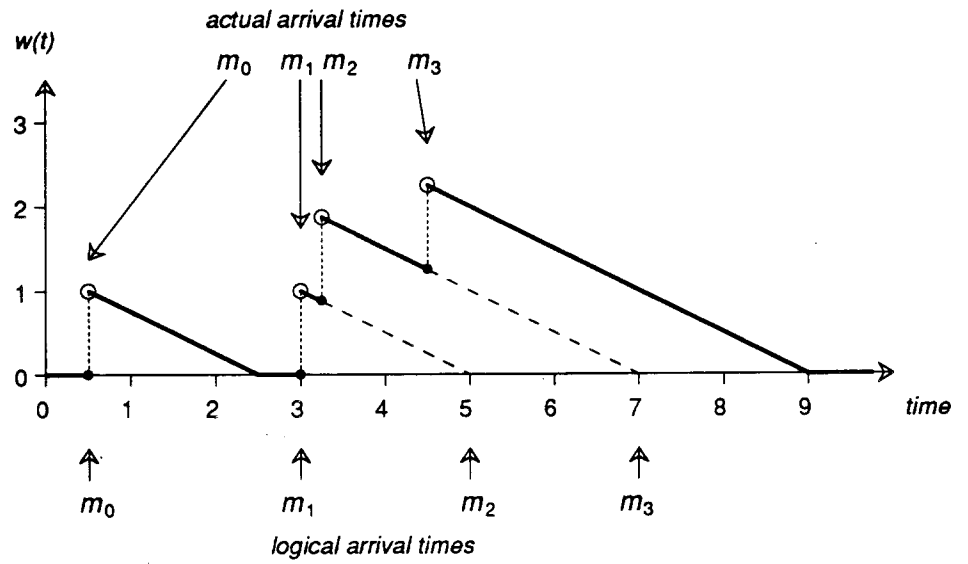


Figure 2.4. Message arrival time and workahead.

The workahead function $w(t)$ for an LBAP with R = 0.5 and message arrivals at times 0.5, 3.0, 3.25, and 4.5. The corresponding logical arrival times are 0.5, 3.0, 5.0, and 7.0.

M = maximum message size (bytes)
R = maximum message rate (messages/second)
W_{in} = input workahead limit (messages)
W_{out} = output workahead limit (messages)
D = maximum logical delay (seconds)
A = minimum actual delay (seconds)

The application must ensure that the arrival process at the input interface obeys the LBAP parameters M, R and W_{in}. The resource must ensure that the arrival process at the output interface obeys the LBAP parameters M, R and W_{out}. D is an upper bound on the logical delay, between the input and output interfaces of the resource, of any message associated with the session. A is a lower bound on the actual delay.

A session is an agreement between the application and the resource. The resource guarantees that it will obey the delay bounds and the output workahead limit. The application guarantees that it will not exceed the workload parameters.

Each resource has an associated software module that exports three functions: `reserve()`, `relax()` and `free()`. To establish a session with a resource, the application first calls `reserve()`, giving the session's message size and rate. If the session can be accepted, `reserve()` returns the minimum possible logical delay bound D_{min} for the session, and makes a resource reservation sufficient to provide this bound. The application then decides (see below) on a specific delay bound $\overline{D} \geq D_{min}$. It calls `relax()` to alter the existing reservation by changing the delay bound to $\overline{D}$. `Free()` deletes an existing session.

Consider a situation where CM data traverses a linear sequence of resources. For example, in the file playback application of Section 2.2.1, data originates from a disk, traverses a CPU, a network, and another CPU, and is then consumed by the display or the speaker. In the CM-

resource model, such a situation is represented as a "compound session" consisting of sessions with each of the resources involved. A *compound session S* is a sequence of sessions $S_1 \cdots S_n$ in which the output interface of S_i is the input interface of S_{i+1}.

The CM-resource model defines a two-phase protocol for establishing compound sessions. In the first phase, the application calls `reserve()` at each resource S_i. Based on an economic approach to end-to-end delay allocation, the application then decides how to divide its target end-to-end delay among the resources S_i. In the second phase, the application calls `relax()` at each resource from S_n to S_1, giving each resource its share of the delay. Anderson [And] discusses this algorithm in greater detail.

2.3.1.2. The Producer/Consumer Paradigm

Another approach to the end-to-end scheduling problem uses the *real-time producer/consumer paradigm* [JeS90]. A CM application may be modeled as a directed graph in which the vertices represent processes and the edges represent message communication channels. Each channel defines a producer/consumer relation between two processes. The application specifies the minimum rate of messages on the channel.

A pair of interconnected processes adheres to the real-time producer/consumer paradigm if a message produced on the connecting channel is consumed before the next message is sent. In other words, the emission of messages by the producer defines ticks of a discrete time clock; if the paradigm is obeyed, the consumer appears as fast as the producer. Jeffay [Jef] describes a decision procedure for determining whether, for a given set of processes and processing resources, all pairs of interconnecting processes are guaranteed to adhere to the paradigm. The input to the decision procedure includes the worst case message processing time on each channel (the actual delay in Section 2.3.1.1).

This approach has been used in the construction of a desktop audio and video conferencing system on top of a real-time operating system called YARTOS [JSS91]. The kernel provides two basic abstractions: *tasks* and *resources*. A task represents a thread of control; a task may need to access one or more resources during its execution. An application workload is specified as a set of tasks and resources. The kernel then guarantees, using the real-time producer/consumer paradigm, that the delay requirements of all tasks are met and that no shared resource is accessed simultaneously by more than one task.

2.3.1.3. The Sun Approach

Sun Microsystems' HRV project [HBJ91] takes a different approach to the end-to-end scheduling problem. In their approach, an application specifies its delay and throughput requirements, but no attempt is made to reserve resources for the application's lifetime. Instead, their approach attempts to satisfy each application's requirements until device capacity overload is encountered. At that point, their resource schedulers negotiate with applications to arrange for graceful degradation from overload.

An application constructs an end-to-end schedulable entity using two abstractions: *transducers* and *conduits* [NoK91]. A transducer is similar to a source or sink resource in Section 2.3.1.1, while a conduit represents a handler resource. An application may compose conduits in serial and in parallel to obtain compound conduits (similar to compound sessions).

Applications specify their delay and throughput requirements in terms of a time interval (T) during which some number of samples (N) are to be transported from source to sink transducers. The operating system and network communication system then attempt to satisfy this requirement using *software phase-locked loops*. This is a mechanism for time-regulation of conduits which adjusts the rate of transport on the conduit based on an error signal. This signal is generated by comparing the actual number of samples transported in a time interval over the compound conduit with the desired number N.

2.3.2. CM Storage

The throughput requirements of integrated CM applications have two implications for CM storage system design. High data rates mean that CM files can occupy significant disk space. File system organization must permit efficient sharing of CM files. Section 2.3.2.1 discusses these structural issues. Moreover, applications may concurrently retrieve multiple files from disk. Section 2.3.2.2 surveys work in real-time storage and retrieval of CM data from disk.

2.3.2.1. Structural Issues in CM File Storage

Structural issues for CM files (sharing, parallel composition, annotations, etc.) have been addressed in the Xerox Etherphone system [TeS88], the Sun Multimedia File System [StL89], and the Northwestern Network Sound System [RKD85].

In the Etherphone system, a sequence of continuously recorded video frames or audio samples is called a *strand*. Editing applications may create *ropes*: a rope describes a collection of related strands and how they are synchronized relative to each other. Ropes permit serial or parallel temporal composition of strands. A rope can be implemented without duplicating the constituent strands. A browsing application may annotate a collection of strands; in the Sun Multimedia File System, annotations can be stored as part of a rope.

2.3.2.2. Real-time Storage and Retrieval of CM files

In the Continuous Media File System (CMFS) [AOGar], applications may store or retrieve CM files in "sessions". Each session has a guaranteed minimum data rate. Multiple sessions, perhaps with different data rates, may coexist. CMFS also handles non-real-time traffic concurrently with real-time sessions.

Consider a schedule of disk reads that cyclically reads a set of blocks for each session in progress. Such a schedule is *feasible* if the playback time of the blocks read for each session is greater than the worst case time to perform the schedule. The existence of a minimal such schedule that fits the available buffer space is used to test for session acceptance. After a session is accepted, CMFS guarantees an application a minimum data rate for reading the file.

If adequate buffer space is available, CMFS can work ahead (Section 2.3.1.1) on a session. If CMFS has worked ahead on all sessions, it has some "slack time" before which reads for sessions should restart (to avoid session starvation). This slack time is used to service non-real-time disk requests. In the absence of such requests, CMFS uses the available slack time to compute a feasible schedule that adds to session work ahead.

Other work has investigated more restricted versions of the problem. Abbott *et al.* [Abb84] and Park and English [PaE91] address performance issues without guaranteeing minimum data rates. Yu *et al.* [YS89] discuss the layout of interleaved data streams with different data rates on a compact disk for guaranteed-performance playback. Gemmell and Christodoulakis [GeC92] describe a file system supporting multiple audio channel playback with concurrent non-real-time traffic. The channels must have the same (constant) data rate and must start at the same time. Finally, Rangan and Vin [RaV91] describe a system that combines disk input and display-device output for multiple data streams. They study admission control under the assumption that sessions have equal data rates.

2.3.3. CM I/O

In the existing discrete media framework, window systems like X11 and NeWS provide network-transparent, concurrent access to discrete media I/O devices. Property (4) of the integrated CM framework (Section 2.1.2) implies that it is desirable to access CM I/O devices as well in a network-transparent, concurrent manner. Examples of work in this area include ACME [AGH91], the DEC Audio Server [AHL91], and MuX [RBK92].

The ACME client-server protocol is based on the following abstractions:

- *Ropes, Strands, and CM Connections:* A *strand* is a stream of audio or video data encoded in a byte stream. Each strand has a type representing the encoding scheme. Multiple strands (say, an audio and a video stream) may be interleaved in a byte-stream *rope*

(different from the rope described in Section 2.3.2); the interleaving scheme defines the type of the rope. A *CM connection* is network connection used to convey a strand or rope.

- *Logical Devices:* A *logical device* (LDev) is an abstract CM I/O device. There are four types of LDevs: *VWins* (video output), *VCams* (video input), *listeners* (audio input), and *players* (audio output). LDevs have various attributes according to their type; for example, a VWin has the attributes of a graphics window: position, size, and stacking order. Clients can *map* LDevs to physical I/O devices. Multiple LDevs may be mapped to a single physical device; in the case of players, the server is responsible for "mixing" the respective outputs. The LDevs associated with the strands of a rope are grouped into a *compound logical device* (CLDev).
- *Logical Time Systems:* LDevs and CLDevs can be associated with a *logical time system* (LTS). All strands in an LTS are played (or generated) in synchrony, even if they come from different sources. The ACME server ensures that the strands start playing at the same time and remain in lockstep. The client may also start, stop, or alter the speed of an LTS, affecting the component strands uniformly.

The DEC Audio Server defines abstractions similar to CM connections, LDevs and CLDevs. Synchronization is performed using command queues associated with the CLDev; a command specifies a particular operation on an LDev or CLDev. Queues allow for sequential processing of commands without client-server round-trip communication. The server also maintains a client-addressable name space of audio clips; it stores and retrieves these directly from the storage device.

MuX is an extension to the X11 window system to support network-transparent, concurrent CM I/O. In addition to strands, ropes, CM connections, logical devices and logical time systems, MuX also provides the *sequence* abstraction. A *sequence* is a time-line description that allows parallel and serial composition of CM and discrete media I/O activity.

2.3.4. Network Communication

In the framework described in Section 2.1.1, applications communicate over the network using a standard protocol suite. Property (4) of an integrated CM framework implies that it is desirable to modify the Internet suite of protocols for CM data communication across networks. This section briefly reviews some work in this area.

SRP [AHS90] is a resource reservation protocol for guaranteed-performance communication on the Internet. Using the CM-resource model as the basis for reserving network resources (network interfaces, gateways etc.), SRP sets up a compound session associated with the connection of a particular IP-based protocol (*e.g.*, a TCP connection). Data sent on this connection is then transmitted according to the throughput and delay requirements of the compound session.

The Tenet real-time protocol suite [BaM91] is a connection-oriented suite of network and transport protocols for real-time wide area communication. The Tenet approach provides for the establishment of *real-time channels* that guarantee minimum throughput and bounded network delay. The basis for resource reservation is similar to the CM-Resource Model; the traffic parametrization is different.

The Stream Protocol (ST) is an internet layer connection-oriented protocol for real-time conferencing applications. ST allows higher level protocols to set up *streams* [Top90]. A stream is a multi-way connection spanning all participants of a conference. Applications specify stream characteristics such as average and burst throughput, round-trip delay, delay variance and error rate. Gateways and networks select multicast routes and perform network resource allocation. Streams may be modified by addition or deletion of endpoints or by network failures.

Chapter 3

OPERATING SYSTEM SUPPORT FOR INTEGRATED CM APPLICATIONS

This chapter discusses how integrated CM affects the "processing and local communication" portion of the existing discrete media framework (Figure 2.3). Property (2) of an integrated CM framework requires that users be able to run multiple CM applications concurrently. Therefore, multiple CM tasks (tasks whose processes handle CM data) may execute concurrently on a workstation. For example, Figure 2.1 shows a client task and an I/O server task executing on the same workstation. These tasks may also communicate CM streams between user VASs or between user and kernel VASs. In Figure 2.1, the client task sends a CM stream to the I/O server and the latter writes two streams to the kernel.

Section 3.1 motivates reexamination of OS **policies** and **mechanisms** for integrated CM. Section 3.2 argues that CPU scheduling policies in general-purpose OSs may be non-optimal for scheduling CM tasks. It describes *deadline/workahead scheduling*, a CPU scheduling policy designed for CM. Section 3.3 argues that conventional mechanisms for process scheduling and inter-VAS stream communication can add significant overhead to CM task execution. This motivates the design of new process scheduling and stream communication mechanisms, the subject of this dissertation.

3.1. Reexamining OS Policies and Mechanisms

General-purpose operating systems incorporate design principles that are contrary to the needs of integrated CM application processing and communication:

- The *request/reply* paradigm (the basis of centralized systems as well as RPC-based and object-oriented distributed systems) may be non-optimal for stream-oriented CM communication.
- Assumptions about delay tolerance of data accesses leads to the use of buffering and large messages. These can add queueing or packetization delay to CM processing.
- Scheduling policies in current systems have the goals of fairness, maximum system throughput, and fast interactive response. CM applications have real-time requirements that may conflict with these goals.
- Communication protocols may provide reliable data transport. Some CM applications can tolerate unreliable delivery of CM data.

These principles impact the design of policies and mechanisms in these general-purpose OSs. These policies and mechanisms may need to be reexamined in light of the real-time requirements of integrated CM applications.

3.2. Operating Systems Policies

Anderson's solution [And] to the end-to-end scheduling problem (Section 2.3.1.1) divides an application's end-to-end delay among the different system components that handle CM data. One such component is a CM task (*e.g.*, the client task in Figure 2.1). Within a CM task, it is convenient to handle separate CM streams in separate processes; the CPU scheduler can then schedule each process according to its throughput and delay requirements.

Most CPU scheduling policies in general-purpose OSs do not allow applications to "reserve" CPU capacity. CPU overload may occur; such overload can violate the throughput requirements

of CM processes. Most OSs lack mechanisms for overload detection and graceful degradation.

Even if CPU capacity reservation were supported, CPU scheduling policies in general-purpose OSs could still be non-optimal for CM process scheduling. Most such policies use the following criteria: 1) fast response for interactive processes, 2) high throughput for background processes and 3) fairness. For instance, the round-robin policy is often used to satisfy fairness. Thus, the UNIX time-slicing policy [LMK89] assigns processes to one of many different priority levels; at a given level, processes are scheduled round robin.

One or more of the above criteria may be contrary to CM process delay requirements. Suppose we are given a set of CM processes, each with its delay and throughput requirements. Suppose further that there exists a scheduling policy which will satisfy these requirements (*i.e.*, the set of processes is *schedulable*). Results in real-time systems show that policies which use preemption on quantum expiration (*i.e.*, use round-robin scheduling) may reduce the schedulability of such a set of processes (*i.e.*, may not be able to satisfy its real-time requirements) [LiL73, TNR90].

3.2.1. Deadline/Workahead Scheduling

The *deadline/workahead* CPU scheduling (DWS) policy is designed for CM tasks [And]. Each CM task has one or more processes. In DWS, processes are classified as either *real-time* or *non-real-time*. There are two classes of non-real-time processes: *interactive* (those requiring fast response) and *background* (those requiring high throughput).

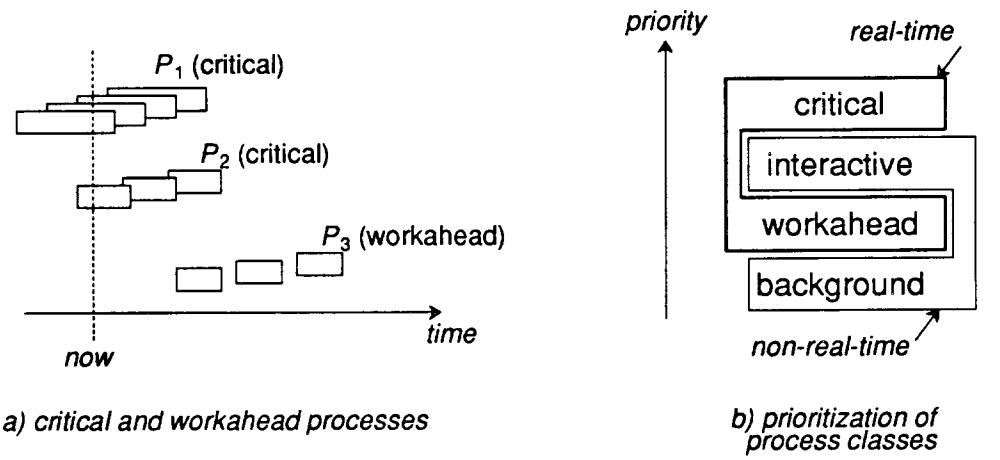
CM tasks reserve sessions (Section 2.3.1.1) with the CPU scheduler's resource manager. We assume that for each session S there is a real-time process P_S that does all the work for S, and no other work. P_S handles a sequence of messages arriving asynchronously (say, on a network connection), and sleeps whenever no messages are available. The input interface for S is defined by message arrivals; in the case of a network connection, a message arrival occurs when the network interface requests a receive interrupt. Message completion occurs when P_S makes a call indicating that it has handled the packet; this call either changes the priority of P_S or puts it to sleep if there are no more messages.

The *delay bound* of a real-time process is the delay bound of its associated session. The *deadline* of a message is the sum of its logical arrival time (Section 2.3.1.1) and the delay bound of the process handling that message. At any instant, a real-time process may have one or more unprocessed messages. The *critical time* of a real-time process is the earliest logical arrival time among all unprocessed messages. The *deadline* of a real-time process is the earliest deadline among all unprocessed messages. At a given time t, a real-time process is called *critical* if it has an unprocessed message m with $l(m) \leq t$ (*i.e.*, m's logical arrival time has passed). Real-time processes that have pending work but are not critical are called *workahead* processes. A workahead process becomes critical when its critical time equals the current time.

The DWS policy can be summarized as follows (Figure 3.1). Critical processes have priority over all others, and are preemptively scheduled according to earliest deadline. Interactive processes have priority over workahead processes, but are preempted when those processes become critical. Background processes have lowest priority.

Non-real-time processes are scheduled according to an unspecified policy, such as the UNIX time-slicing policy. This policy may also move a process between interactive and background. The policy for workahead processes is also unspecified. One possible policy chooses a workahead process P, perhaps with the earliest deadline or the most work available. P is then run for a full quantum (say, 100 times the system call plus context switch time) even if its deadline advances beyond that of another workahead process. Such a policy is designed to reduce context switch overhead.

The DWS policy is appropriate for CM process scheduling for a number of reasons. It supports CPU capacity reservation; this can prevent CPU overload and "guarantee" process throughput requirements. For any set of schedulable processes whose delay bound is equal to the message interarrival time, the earliest-deadline policy can meet each process' delay bound; DWS uses earliest-deadline scheduling as a heuristic to improve the schedulability of CM workloads. The DWS policy includes timeliness as a criterion for real-time processes; CM processes may express their throughput and delay requirements in terms of message deadlines. DWS also



a) critical and workahead processes

b) prioritization of process classes

Figure 3.1. Deadline/workahead scheduling.

In the deadline/workahead scheduling (DWS) policy, each real-time process has a queue of pending messages. In example a), each message is shown as a rectangle whose left edge is its *logical arrival time* and whose right edge is its *deadline*. P_1 and P_2 are *critical* because they have a pending message whose logical arrival time is in the past. Processes are prioritized as shown in b). Critical processes are executed earliest deadline first; policies for other classes are unspecified.

allows processing of messages that have arrived before their logical arrival time; CPU schedulers may efficiently schedule workahead processes (see the policy above), for example, to reduce context switch overhead.

3.3. Operating Systems Mechanisms

Section 3.1 suggests reexamining OS mechanisms for integrated CM applications. These mechanisms may add processing overhead to CM tasks. This can happen when functionality is inappropriately partitioned across protection domains. For instance, if process scheduling is implemented in the OS kernel, every process switch requires kernel intervention. Real-time processes in DWS change their deadlines at every message completion (Section 3.2.1); each such deadline change incurs kernel trap overhead.

Section 3.3.1 argues that process scheduling and inter-VAS CM stream communication mechanisms can add significant processing overhead to integrated CM. Recent work demonstrates that this overhead is not decreasing in proportion to increasing processor speeds.

3.3.1. Mechanisms For Process Scheduling and Stream Communication

In DWS (Section 3.2.1), each real-time process handles a sequence of CM messages. After handling a message, the process changes its deadline (a scheduling operation). CM tasks may communicate CM streams between user VASs or between a user VAS and a kernel VAS. Inter-VAS CM stream communication can be decomposed into a series of CM *message transfer* operations.

Section 3.3.1.1 describes the process structure of two CM tasks, and illustrates how these tasks may perform frequent process scheduling and message transfer operations. Section 3.3.1.2 lists two conventional approaches for process scheduling and message transfer. Section 3.3.1.3

argues that CM tasks may incur high overhead when implemented using these approaches.

3.3.1.1. CM Task Structure Examples

Figure 3.2 shows the real-time processes in a typical ACME (Section 2.3.3) server task. This task has two types of real-time processes: *network I/O* and *device I/O* processes. Network I/O processes perform I/O to and from CM connections. A network output process reads a block of messages from a CM connection and writes it to an internal strand buffer. Before writing the data it may process it in some way; for example, it may do volume scaling of audio streams or split a rope into its constituent strands. Device I/O processes perform I/O to and from CM devices. A device output process gathers data from one or more strand buffers, perhaps combines them (*e.g.*, by summing audio samples), and writes them to the device. A device input process reads data from the device and writes a copy to the internal buffer, to be read later by a network input process.

Figure 3.3 shows the real-time process structure of the mixer task (Section 2.2.2). Each mixer task has three kinds of real-time processes. *Reader* processes copy data from input network connections to memory buffers. They may do some processing (*e.g.*, type conversion) on the stream. *Adder* processes read data from one or more buffers, do the mixing, and write the results to other buffers. *Writer* processes copy data from buffers to the output CM connections. They may also perform some mixing or format conversion.

The rationale for the process structures of these tasks is as follows. Streams on different CM connections may tolerate different delays and different workahead limits; for instance, stream

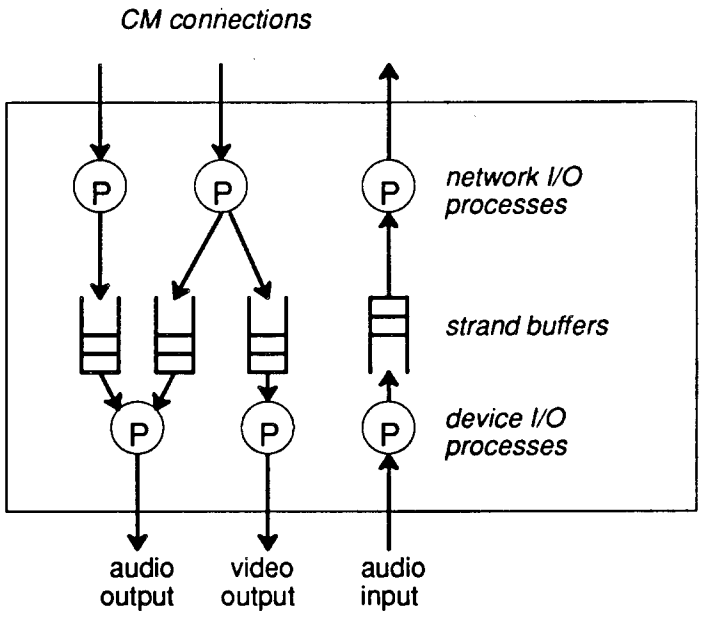


Figure 3.2. The ACME server task structure.

The ACME server is structured as a set of LWPs. Network I/O processes perform message transfers from CM connections and may do some processing on the message. Device I/O processes perform message transfers to CM I/O devices.

c (Figure 3.3) may be an input from a file and stream *b* may be from a live participant. Assigning one process to each network connection (*e.g.*, reader or writer processes in the mixer and network I/O processes in ACME) enables the CPU scheduler to differentiate between streams with different delay bounds and workahead limits. For a similar reason, each I/O device in ACME is assigned one process. A similar argument can be made for having multiple adder processes in the mixer task. Suppose streams *c* and *d* (Figure 3.3) are input from a file but *a* and *b* are input from live participants. The reader tasks of *c* and *d* can work ahead on their streams. If both reader tasks have worked ahead, then it is beneficial to work ahead on mixing those streams also. For that reason we assign an adder process to mixing *c* and *d*.

Each real-time process in ACME and the mixer performs at least one scheduling operation per CM message. A real-time process may also perform one inter-VAS message transfer per message. For example, the network output process does a kernel to user message transfer on message arrival and changes its deadline on message completion. A writer process does a user to kernel message transfer and a deadline change on message completion. An adder process does a deadline change on message completion.

If the number of processes is high (*e.g.*, the number of concurrent CM streams is high), then the frequency of process scheduling and message transfer operations is high. This frequency is high for CM tasks handling low end-to-end delay CM streams. Such streams tend to have high message rates; one hundred messages/sec represents a packetization delay alone of 10 ms.

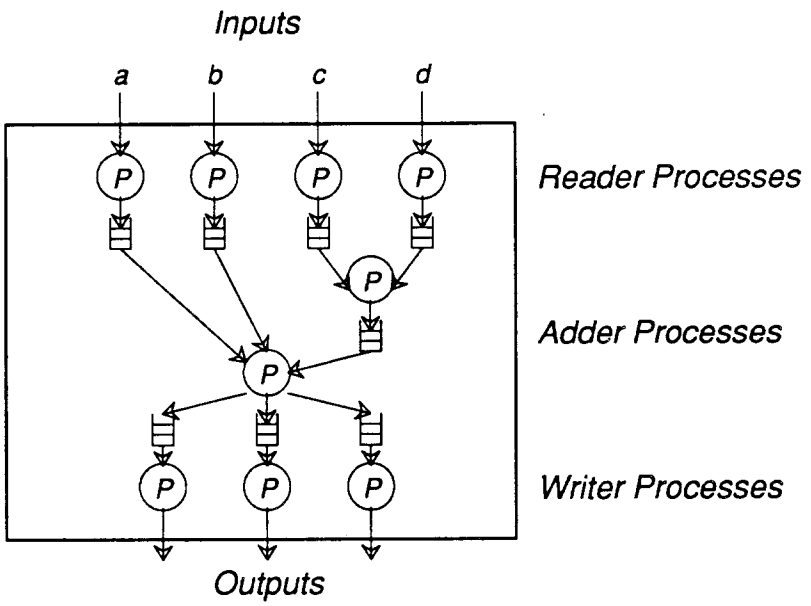


Figure 3.3. The mixer task structure.

Real-time processes in a mixer are of three types: reader, writer and adder. The reader and writer processes perform I/O from network connections to sources or sinks. The adder processes perform the intermediate steps of the mixing computation.

3.3.1.2. Conventional Approaches for Process Scheduling and Message Transfers

In existing general-purpose OSs, tasks use one of two conventional mechanisms for process scheduling and message transfer.

- *Threads and read/write system calls:* A *thread* is a kernel-implemented process. The kernel exports a system call interface for thread management and maintains thread execution state. Thread switching is done completely within the kernel. Threads perform message transfers using system calls. In Mach, for example, a thread calls `send_msg()` for a user-to-user message transfer.
- *Lightweight processes (LWPs) and asynchronous message transfer:* LWPs are implemented entirely at the user level. A library implements LWP creation, deletion, state manipulation and switching. A blocking system call (*e.g.*, on an I/O operation) by a LWP suspends all activity in the VAS. To avoid that, LWPs use a kernel interface that provides asynchronous, non-blocking message transfers. The QIO facility in the VMS operating system provides such an interface [LeE89]. With such an interface, a message transfer typically involves a system call to initiate the transfer followed by an *asynchronous event* (*e.g.*, a UNIX signal [LMK89]) to signal transfer completion.

When implemented using these approaches, CM tasks may incur the overhead of *user/kernel interactions* (by which user programs access system functions for process scheduling and message transfer). User/kernel interactions are of two types: synchronous (with respect to user programs) user-to-kernel system calls and kernel-to-user asynchronous events. Each user/kernel interaction incurs one or more *domain switches*. A *domain switch* is defined as a crossing of the user/kernel protection boundary; a system call involves two domain switches and a UNIX signal three (one for a kernel upcall to user space to execute the signal handler and two for a signal return system call that resets signal masks). A user/kernel interaction may also incur a *mapping switch* between different user VASs.

With these conventional approaches, a single scheduling or message transfer operation may incur more than one user/kernel interaction. For example, an asynchronous kernel-to-user message transfer in UNIX may incur up to nine domain switches and two mapping switches. A non-blocking `read()` system call returns immediately if no messages are pending. A `SIGIO` signal is delivered when a message arrives. The `select()` system call chooses the file descriptor on which the message has arrived and another `read()` does the actual transfer.

3.3.1.3. Domain Switches And Mapping Switches

Domain switches are expensive relative to procedure calls. A null system call (which incurs two domain switches) takes about 34 μsecs on SunOS 4.1 for the SPARCstation 1+. There are two components to system call overhead: the kernel trap and preparing the processor to execute a procedure call to a higher-level language OS routine. This latter call preparation involves vectoring from trap entry point to the appropriate exception handler, managing machine state (*e.g.*, machine registers and kernel stack pointers) and saving/restoring registers used during the call.

A system call has indirect costs as well. Because a system call involves a cross-domain interaction, the system call handler must copy and check parameters to guard against application errors. Moreover, inter-procedural optimizations (*e.g.*, avoiding register saves/restores) are difficult for system calls [Kar89]. Finally, kernel execution during a system call may incur more cache and TLB misses than user-level execution. Agarwal *et al.* [AHH88] show that caches perform poorly for OS memory references for two reasons: 1) OS code and data structures are larger than user code and data structures; they occupy more cache space and bringing the working sets into the cache needs more cache misses and 2) OS code loops have fewer iterations than user code loops. Clark and Emer [CIE85] argue that TLBs perform poorly for similar reasons.

With architectural trends towards RISC processors, system call costs have not decreased in proportion to increasing processor speeds [ALB89]. RISC processors such as the Sun SPARC [CCC90] and the MIPS R2000 [Kan87] have more than 64 registers, so call preparation costs are greater. Moreover, RISC processors have added new features, such as the register windows in SPARC, that reduce kernel trap performance. In SPARC, the hardware ensures that one register window is available for the trap handler on exceptions. This handler has to ensure that another

window is available for the system call handler (to avoid recursive window overflow traps). This requires possibly saving (and later restoring) user register windows, and additional copying of system call parameters.

User VAS mapping switches are also expensive. In SunOS 4.1 for the SPARCstation 1+, the cost of suspending one UNIX process and switching to another is about 140 μsecs.

The direct cost of a mapping switch is the time taken to change the current processor memory map. However, a mapping switch has a significant indirect cost component. A mapping switch results in a change in program locality; the indirect cost is the cost of repopulating cache and TLB contents in this new locality. Depending on the cache parameters, this cost may be in the tens or hundreds of microseconds [MoB89].

The indirect costs of mapping switches are not decreasing in proportion to increasing processor speeds [MoB89]. Processor speeds are improving, but memory access times have not been increasing proportionately. Thus, cache-miss penalties are becoming relatively greater.

3.4. New Mechanisms for Process Scheduling and Stream Communication

The previous section makes the following arguments:

(1) Some CM tasks may perform frequent process scheduling and message transfer operations.

(2) With conventional approaches, each process scheduling or message transfer operation may incur one or more user/kernel interactions. A user/kernel interaction requires at least one domain switch and possibly a mapping switch.

(3) Domain switches and mapping switches are expensive. The cost of these components does not seem to be decreasing with increases in processor speeds.

This dissertation argues that *new process scheduling and inter-VAS stream communication mechanisms can significantly reduce overhead (up to a factor of four) for some CM application workloads.* These new mechanisms are:

- *Split-level scheduling and synchronization.* In this approach each user VAS contains multiple lightweight processes (LWPs). The scheduler is partitioned into user-level and kernel-level parts, which communicate via shared memory. The information in shared memory is used to correctly prioritize LWPs in different VASs and to minimize user/kernel interactions.
- *Memory-mapped streams.* A memory mapped stream (MMS) is a shared-memory FIFO used for communicating CM data between user and kernel address spaces. Once the MMS has been set up, no explicit kernel requests are needed to transfer data, and a minimal number of user/kernel interactions are needed for producer/consumer synchronization and I/O initiation.

Chapter 4

SPLIT-LEVEL SCHEDULING AND SYNCHRONIZATION

This chapter describes *split-level scheduling* (SLS), a LWP scheduler implementation technique that correctly prioritizes LWPs in different VASs while reducing or eliminating user/kernel interactions. Section 4.1 gives an overview of the technique. Section 4.2 lists the client interface to a split-level scheduler for the deadline/workahead scheduling policy (Chapter 3). Finally, Section 4.3 describes the implementation of split-level scheduling, and discusses a related mechanism for LWP synchronization.

4.1. Overview of Split-Level Scheduling

A split-level scheduler consists of two parts: a per VAS *user-level scheduler* (ULS) and a single *kernel-level scheduler* (KLS) (Figure 4.1).

The ULS runs the highest-priority LWP in its VAS. In a VAS, multiple LWPs share a single thread. An LWP sleeps or changes its priority by calling the ULS for that VAS. At that time, the ULS checks whether its VAS still contains the globally highest-priority LWP; this is done by examining an area of memory shared with the kernel. If so, the LWP context switch is done without kernel intervention. Otherwise, the ULS executes a kernel trap to transfer control to the KLS.

The *kernel-level scheduler* (KLS) schedules the VAS with the globally highest-priority LWP, preempting the currently executing VAS if necessary. LWP priorities are communicated to the KLS through user/kernel shared memory. Shared memory is used to determine the VAS with the highest-priority LWP. The KLS is unaware of the existence of LWPs (it is not involved in the creation of a LWP, for instance), but helps in scheduling them correctly.

While split-level scheduling can be used with many scheduling policies, we focus on its implementation for the deadline/workahead (DWS) policy described in an earlier chapter. For simplicity, we consider only the scheduling of real-time processes. In Section 4.3.6, we briefly discuss how split-level scheduling works for non-real-time processes and for other CPU scheduling policies.

4.2. Client Interface to the Split-Level Deadline/Workahead Scheduler

A user-level library provides the client interface to the split-level deadline/workahead scheduler. The interface functions may be grouped into: LWP creation and deletion, state manipulation, and synchronization. We discuss each group in turn. The interface is summarized in Table 4.1.

4.2.1. LWP Creation and Deletion

A deadline/workahead scheduled LWP P has three scheduling parameters: a fixed *delay bound* (see Chapter 3), a *critical time* C_P (the logical arrival time of the next message for the LWP) and a *deadline* D_P (C_P plus the delay bound).

An application task creates a LWP by calling the function

```
create_LWP(
        PROCEDURE proc,
        TIME delay,
        TIME critical_time,
)
```

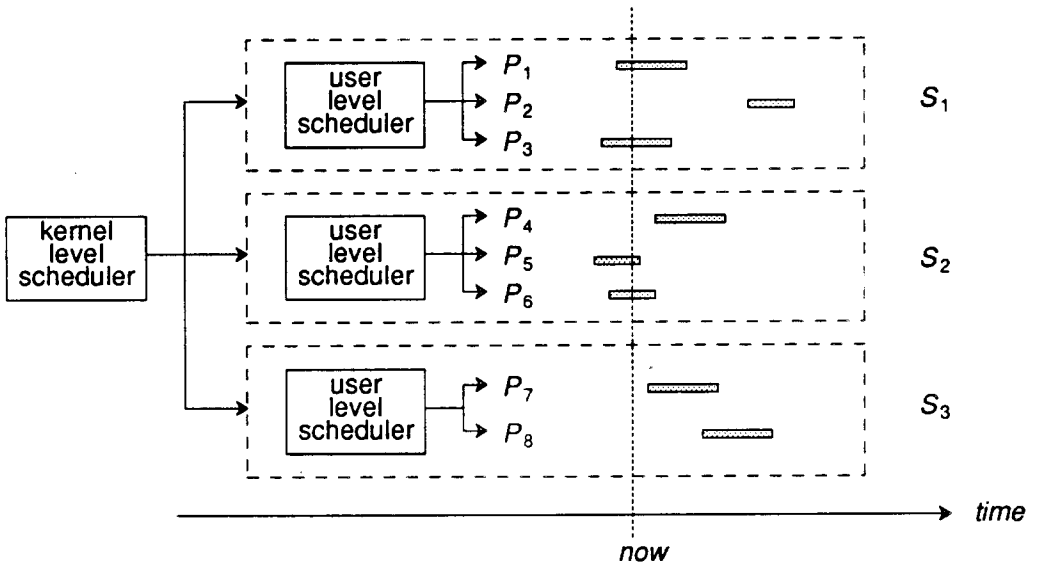


Figure 4.1. Split-level scheduling.

Using *split-level scheduling*, the kernel-level scheduler decides which user VAS should execute, and each VAS has a user-level scheduler (ULS) that manages the LWPs in that VAS. In this example, the KLS chooses VAS S_2 to run because it has the globally highest-priority LWP. The ULS in that VAS executes P_5, which has this deadline (the deadline of a LWP is the right endpoint of the corresponding rectangle). User/kernel interactions can often be avoided: in this example, if P_5 yields then the context switch to P_6 (the next earliest deadline) can be done without a kernel call.

The newly-created LWP P executes the function `proc` and has a delay bound specified by `delay`. C_P is set to `critical_time` and D_P to C_P plus the delay bound. If C_P is less than the current time, the LWP is made runnable, otherwise it is suspended. A suspended LWP becomes runnable at `critical_time`.

`Destroy_LWP()` releases all resources held by the calling LWP.

4.2.2. LWP State Manipulation

LWPs may dynamically change their deadline, wait for a specified time period to elapse or wait for I/O completion. Each function call in this group alters the scheduling parameters of the calling LWP and may cause the LWP to become suspended.

After processing a CM message, an LWP calls

```
time_advance(TIME critical_time);
```

the argument is the logical arrival time of the next message. C_P is set to `critical_time` and the LWP remains runnable.

To suspend execution until a specified time, an LWP calls

```
timed_sleep(TIME critical_time);
```

create_LWP()	set a new LWP executing a specified function
destroy_LWP()	delete and release resources held by calling LWP
time_advance()	change scheduling parameters of caller
timed_sleep()	suspend caller until specified time
IO_wait()	suspend caller until I/O becomes possible on specified descriptor
mask_LWP_preemption()	start critical section
unmask_LWP_preemption()	end critical section

Table 4.1. Client interface to the split-level scheduler.

The client interface to the split-level scheduler for the DWS policy contains functions for creating and destroying LWPs, for manipulating LWP state and for mutually exclusive access to short critical sections.

after that time it becomes runnable and C_P is set to the current time. This may be used by processes that do time-based output with no device synchronization (*e.g.*, slow video) or for rate-based flow control.

To wait for I/O to become possible on a given I/O descriptor, an LWP calls

```
IO_wait(DESCRIPTOR iodesc, TIME critical_time);
```

the I/O descriptor may represent a file, a socket, an I/O device or a memory-mapped stream (Chapter 5). The LWP is suspended until data arrives on the descriptor. At that time, the process becomes runnable and its C_P is set to `critical_time`.

4.2.3. LWP Synchronization

Short critical sections in application code are bracketed by `mask_LWP_preemption()` and `unmask_LWP_preemption()`. Between these two calls, the calling LWP cannot be preempted by another LWP in the same VAS.

Applications may use these to ensure mutually exclusive access to shared data structures, provided such critical sections are short. These primitives can also be used to implement sleep-waiting synchronization primitives for longer application-level critical sections.

4.3. Implementation of the Split-level Deadline/Workahead Scheduler

In this section, we describe the user/kernel interface necessary to implement the split-level scheduler (Figure 4.2). We then describe the implementation of the user-level scheduler (ULS) and the kernel-level scheduler (KLS). We defer discussing the implementation of LWP synchronization until Section 4.3.5. Section 4.3.7 discusses issues in implementing SLS on SunOS 4.1 for the SPARCstation 1+. We first list some notation that is used later in this chapter and in subsequent chapters.

4.3.1. Terminology and Notation

The state of a LWP is either *suspended* (waiting for some amount of real-time to elapse or for some I/O operation to complete) or *runnable*. A runnable LWP P is *workahead* if $C_P > T_{now}$ (T_{now} is the current time of day). Otherwise P is *critical*.

In a given VAS, LWP states and scheduling parameters collectively determine the *state* of the ULS. Thus, ULS state may change if an LWP's state changes or its scheduling parameters change.

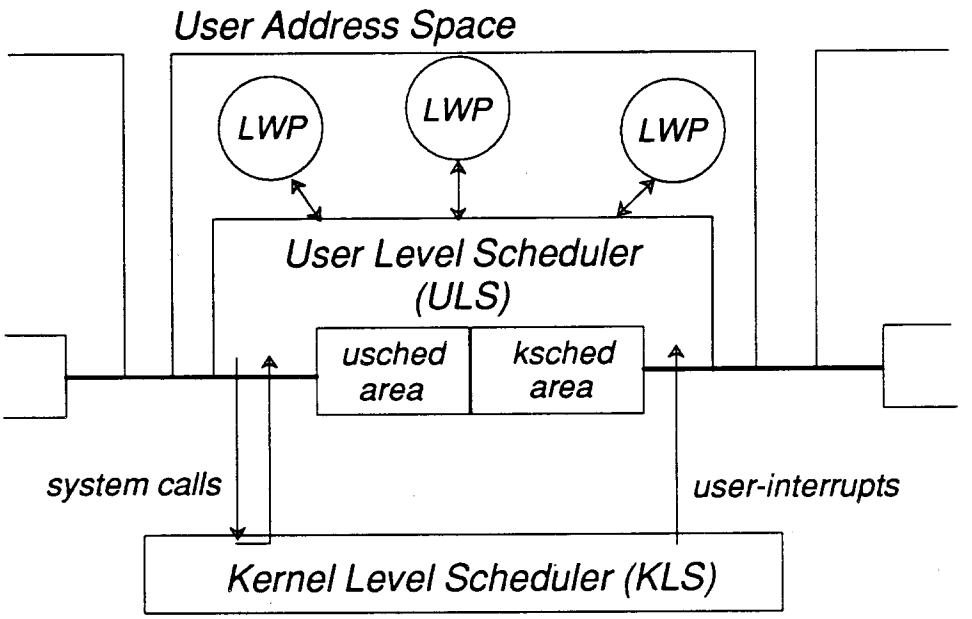


Figure 4.2. Two parts of a split-level scheduler.

The user-level and kernel-level parts of the split-level scheduler communicate using *system calls*, *user-interrupts*, and through an area of shared memory.

The globally highest-priority LWP (as defined by the DWS policy) is denoted by P^*. A^* is the VAS containing P^*. P_A is the highest-priority LWP in a VAS A. If A is the currently executing VAS, $\bar{A}$ denotes the set of VASs not currently executing.

D_A is the minimum of D_P for critical LWPs $P \in A$, or $+\infty$ if there are none. In other words, D_A is the earliest deadline of a critical LWP in A. $D_{\bar{A}}$ is the minimum of D_P for critical LWPs P not in A, or $+\infty$ if there are none.

C_A is the minimum of C_P for workahead LWPs $P \in A$, or $+\infty$ if there are none. In other words, C_A is the earliest critical time of a workahead LWP in A. $C_{\bar{A}}$ is the minimum of C_P for workahead LWPs P not in A, or $+\infty$ if there are none.

4.3.2. User/Kernel Interface

The ULS is implemented as a user-level library that exports the client interface described in Section 4.2. It cooperates with the KLS to schedule LWPs in its VAS. The two parts of a split-level scheduler exchange control and communicate scheduling information in three ways: system calls, kernel upcalls into user space (*user-interrupts*) and shared memory (Figure 4.2). We discuss each of these below. This user/kernel interface is summarized in Table 4.2.

4.3.2.1. System calls

The `init_sls(VIRT_ADDR *user_addr)` call allocates and initializes a block of memory shared read-write between the user VAS and the kernel. This shared memory region is used to convey scheduling information from the ULS to the KLS and vice versa (Section 4.3.2.3). If successful, the call returns the address in user space of the shared memory block. After this

call, the KLS cooperates with the ULS in scheduling LWPs in the VAS according to the deadline/workahead policy.

The `done_sls()` system call unmaps and deallocates the shared memory region from user space. After this call, the KLS cannot globally schedule LWPs in the caller's VAS.

The `register_handler(UINT uinttype, PROCEDURE proc)` system call registers the user-level handler for a specified user-interrupt type (see Section 4.3.2.2).

Finally, the `yield()` system call yields the processor to A^*, the VAS with the globally highest-priority LWP.

4.3.2.2. User-Interrupts

A *user-interrupt* is an upcall from kernel to user space to notify the user VAS of an asynchronous event. The handler for an asynchronous event is registered with the KLS using the `register_handler()` system call. User-interrupts are like UNIX signals except that the handler does not end with a system call to reset the signal mask (hence there is one domain switch rather than three, Section 4.3.7).

Three types of user-interrupts are defined: `INT_TIMER` is delivered when a timer elapses, `INT_IO_READY` is delivered when I/O becomes possible on an I/O descriptor, and `INT_RESUME` is delivered when a user VAS resumes after being preempted.

System calls:	
init_sls()	create user/kernel shared memory
done_sls()	delete user/kernel shared memory
register_handler()	register user-interrupt handler
yield()	yield processor to another VAS
User-interrupts:	
INT_TIMER	timer has expired
IN_IO_READY	I/O is possible on some descriptor
INT_RESUME	VAS returns after being preempted
Usched area of shared memory:	
D_A	earliest deadline of a critical LWP in A
runnable	indicates that some LWP in A is runnable
L	table of suspended and workahead LWPs
WaitingForIO	indicates if I/O descriptor is waiting for I/O completion
T_{next}	the time at which the next timer is to be delivered
Ksched area of shared memory:	
$D_{\bar{A}}$	the earliest deadline of a LWP in some other VAS
T_{now}	the current time of day
ReadyForIO	indicates if previous I/O operation was completed on I/O descriptor

Table 4.2. User/kernel interface for SLS.

The user/kernel interface for a split-level scheduler has three components: system calls, user-interrupts and a region of memory shared read-write between user and kernel. The shared memory region is conceptually divided into a *usched* area (written by the ULS) and a *ksched* area (written by the KLS).

4.3.2.3. User/Kernel Shared Memory Interface

The ULS for each VAS A shares a region of memory read-write with the kernel. This region consists of two parts: the *usched* area and the *ksched* area (see Figure 4.2). The *usched* area is written by the ULS and read by the KLS. It contains the following:

(1) D_A.

(2) The *runnable* flag, which is 1 if there is at least one runnable LWP, and 0 otherwise. In DWS, the policy for scheduling workahead LWPs is left unspecified (Chapter 3). In our description, we assume that the KLS arbitrarily selects a VAS with a runnable LWP when there are no critical or interactive LWPs. The *runnable* flag is used for this purpose. We show in Section 4.3.6 how other workahead policies may be implemented using SLS.

(3) A table L of workahead and suspended LWPs P in A such that $D_P < D_A$. Each entry in the table contains the critical time and deadline of the LWP.

(4) For each I/O descriptor, a *WaitingForIO* flag indicating whether an LWP is blocked on the descriptor. If so, the *usched* area contains the critical time and deadline of that LWP.

(5) T_{next}: the time at which the next `INT_TIMER` user-interrupt should be delivered.

The ksched area, written by the KLS and read by the ULS, contains the following:

(1) T_{now}.

(2) $D_{\bar{A}}$.

(3) For each I/O descriptor, a *ReadyForIO* flag to indicate that data has arrived on that descriptor.

4.3.3. ULS Implementation

A ULS of a VAS A may change state when an interface function is called or a user-interrupt is delivered. Whenever such a state change occurs, the ULS computes P_A according to the DWS policy. If $P_A \neq P^*$ (*i.e.*, if $A \neq A^*$), the ULS calls `yield()`. Otherwise, it does a LWP switch to P_A, if necessary. If the KLS detects that the currently executing VAS $A \neq A^*$, it preempts A. In this section, we describe the data structures and algorithms used in the ULS.

An `INT_TIMER` user-interrupt may cause a non-running workahead LWP or suspended LWP to become critical. Let X be the set of suspended and workahead LWPs P in A such that $D_P < D_A$. Let $T_{critical} = \min(C_P: P \in X)$. Clearly, $T_{critical}$ cannot be less than T_{now}. To reduce the number of `INT_TIMER` user-interrupts, the ULS always sets T_{next} to $T_{critical}$. This policy is correct, as shown in Claim 4.1 (Figure 4.3).

Claim 4.1. *Between T_{now} and $T_{critical}$, changes to P_A cannot be caused by: 1) a suspended LWP becoming critical, or 2) a non-running workahead LWP becoming critical.*

Proof. Suppose, to the contrary, that P_A changes between T_{now} and $T_{critical}$ because a suspended LWP P becomes critical. It must be that $C_P < T_{critical}$ and $D_P < D_A$. However, from the definition of $T_{critical}$, both these conditions cannot be simultaneously true. Hence, P cannot cause P_A to change. A similar argument can be made for the second case. □

The ULS maintains queues of suspended, critical and workahead LWPs. All three queues are sorted earliest-deadline-first. In addition, for each I/O descriptor, the ULS has a pointer indicating which LWP is waiting for I/O completion on the descriptor, if any.

A ULS state change may occur: 1) when an LWP P calls an interface function to change C_P or its state and 2) when the kernel delivers a user-interrupt.

When an LWP R in VAS A calls one of `timed_sleep()`, `time_advance()`, and `IO_wait()`, the ULS inserts R into the appropriate structure (the suspended queue, the workahead or critical queue, and an I/O descriptor respectively), then does the following (see Figure 4.4):

(1) For each LWP P in the workahead and sleep queues such that $C_P < T_{now}$, insert P in the critical queue. For each LWP P waiting for I/O completion for which the

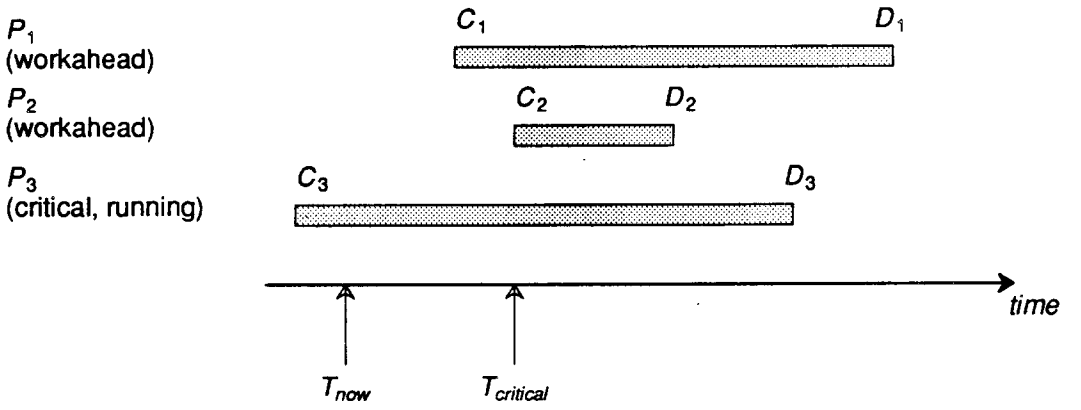


Figure 4.3. Illustration for Claim 4.1.

At a given time T_{now}, the ULS for a VAS A must have a pending `INT_TIMER` user-interrupt for the earliest critical time of a suspended or workahead process $P \in A$ such that $D_P < D_A$. In this example, P_3 is critical and P_1 and P_2 are workahead. If P_3 is still running when C_{P_2} arrives, P_2 becomes critical and must preempt P_3. On the other hand, P_1 cannot preempt P_3 because its deadline is greater. Therefore a timer is needed for C_2 but not C_1.

usched.ReadyForIO flag is set, insert P into the workahead queue (if $C_P > T_{now}$) or the critical queue (otherwise).

(2) Update $usched.D_A$ in the usched area.

(3) Compute $T_{critical}$, using the definition given in Claim 4.1.

(4) If R becomes suspended or is workahead and $D_R < D_A$, update R's entry in *usched.L*. In addition, if R called `IO_wait()`, set *usched.WaitingForIO* for the corresponding I/O descriptor.

(5) If $A \neq A^*$ then call `yield()`, else,

(6) Set $usched.T_{next} = T_{critical}$ (see Claim 4.1). Do a LWP switch to P_A.

The handler for an `INT_TIMER` user-interrupt moves the LWPs P for which $C_P \leq T_{now}$ from the suspended queue to the critical queue. It then executes steps 2, 3 and 6 above. The handler for an `INT_IO_READY` user-interrupt moves all LWPs for which the *usched.ReadyForIO* flag is set to the critical or workahead queue, updates the *usched.L* entry for workahead LWPs P with $D_P < D_A$, and executes steps 2, 3 and 6 above.

An `INT_RESUME` user-interrupt is delivered to a VAS when it resumes execution after having been preempted. Between when the VAS was preempted and T_{now}, an indeterminate amount of time has elapsed. The same is true when the VAS returns from a `yield()` system call. In both cases, the ULS performs steps 1-6 above to update its state.

4.3.4. KLS Implementation

At each instant, the KLS is responsible for running A^*, preempting the current VAS A if necessary. A^* may change if the current VAS A calls `yield()` or if the state of a ULS in $\bar{A}$ changes. The latter can happen in one of two ways: 1) when a suspended or workahead LWP becomes critical or, 2) when a suspended LWP becomes runnable following completion of an I/O operation. The KLS updates $ksched.D_{\bar{A}}$ and $ksched.T_{now}$, and delivers user-interrupts

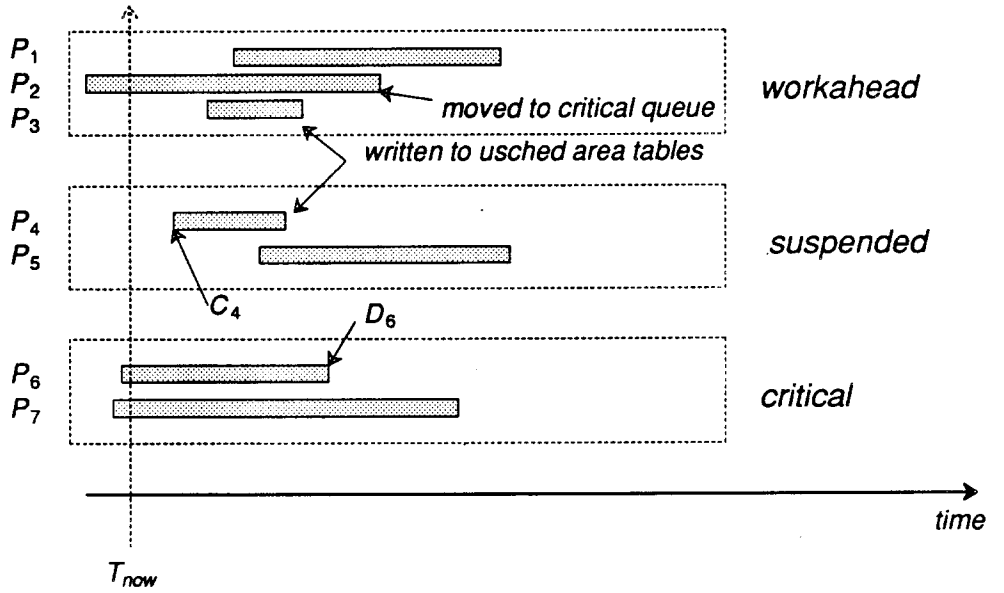


Figure 4.4. Illustration for algorithm in Section 4.3.3.

In this example, a VAS contains LWPs $P_1 \cdots P_7$. The current process, P_4, has called `timed_sleep()`, and the ULS has inserted it in the suspended queue. The ULS then does the following: it moves P_2 to the critical queue, records P_3 and P_4 in the usched area, sets D_A to D_6, and sets a timer for C_4. Finally, it does a context switch to P_6.

appropriately. Below, we discuss KLS data structures and algorithms.

The KLS maintains a linked list of descriptors for all VASs. The descriptor for a VAS is created when it calls `init_sls()`. Each descriptor contains the resources allocated to the VAS, the address of the usched and ksched areas, the I/O descriptors associated with the VAS, a list of pending user-interrupts, and the user addresses of the user-interrupt handlers. The `register_handler()` system call fills in this last field of the descriptor.

A VAS in $\bar{A}$ may change state when a suspended or workahead LWP becomes critical. To detect this change, the KLS sets a timer for $\bar{T}_{critical}$, defined as the minimum of $T_{critical}$ for each VAS in $\bar{A}$ (where $T_{critical}$ is defined in Claim 4.1). To set the timer, the KLS may add $\bar{T}_{critical}$ to a list of pending timers polled on every clock interrupt or may program an interrupt timer to deliver an interrupt at $\bar{T}_{critical}$. $\bar{T}_{critical}$ is computed using *usched.L* for each VAS.

When a VAS A calls `yield()`, the KLS executes the following steps:

(1) Compute A^*, where A^* is that VAS B with the minimum $usched.D_B$, or (if there are no critical LWPs) an arbitrary VAS B whose *usched.runnable* is 1.

(2) Set a timer for $\bar{T}_{critical}$.

(3) Update $ksched.D_{\bar{A}}$ and $ksched.T_{now}$ in A^*.

(4) If $A \neq A^*$, do a context switch to A^*. Arrange to deliver an `INT_PREEMPT` user-interrupt to A^* if it is resuming after preemption.

When a kernel timer goes off, the state of some VAS B in $\bar{A}$ (the current VAS is A) has changed. The KLS takes the following steps:

(1) Recompute D_B and *runnable*, saving these in B's usched area temporarily.

(2) Compute A^* (see `yield()` above).

(3) If $A = A^*$, then update $ksched.D_{\bar{A}}$ (the new value of $D_{\bar{A}}$ is given by D_B) and return.

(4) Otherwise, preempt A, recompute $D_{\bar{A}}$ with respect to A^*, update A^*'s ksched area accordingly and switch to A^*. Arrange to deliver an `INT_PREEMPT` user-interrupt to A^* if it is resuming after preemption.

At every clock tick, the KLS updates the current VAS A's $ksched.T_{now}$ and checks to see if $usched.T_{next}$ has elapsed. If so, it arranges for an `INT_TIMER` user-interrupt to be delivered to A.

When an I/O operation completes on a descriptor, the KLS does the following:

(1) If the I/O descriptor belongs to the current VAS A, set the *ksched.ReadyForIO* flag on that descriptor. If an LWP P is waiting on that descriptor and $D_P < D_A$, arrange to deliver an `INT_IO_READY` to A.

(2) Otherwise, do steps 1 through 4 in the timer expiration case.

4.3.5. Implementing Synchronization

Logically disabling (or *masking*) user-interrupts for short periods of time prevents context switches within a VAS. This technique can be used to implement the `mask_LWP_preemption()` and `unmask_LWP_preemption()` interface functions.

It is desirable to implement user-interrupt masking without user/kernel interactions (*i.e.*, no system calls) in the normal case. This can be done using a technique we call *virtual masking*. Virtual masking is implemented using a *mask level* in the usched area and a *request flag* in the ksched area (the request flag is a bitmap with one flag per user interrupt type).

To mask user-interrupts, the `mask_LWP_preemption()` call increments the corresponding mask level. Whenever the kernel wants to deliver a user-interrupt and finds its mask level nonzero, it sets the corresponding request flag and defers user-interrupt delivery. The `unmask_LWP_preemption()` function decrements the mask level. If this returns to zero and the request flag is set, the appropriate signal handler is called to service the user-interrupt. Chapter 6 discusses virtual user-interrupt masking in greater detail.

4.3.6. Extensions to Split-Level Scheduling

Our description of SLS for the DWS policy can be extended in a variety of ways. In this section, we describe the changes to a split-level scheduler to accommodate a different workahead policy and to incorporate non-real-time workload. SLS can also be adapted to other scheduling policies. We describe a split-level scheduler for preemptive priority scheduling.

4.3.6.1. Different policy for workahead processes

Deadline/workahead scheduling does not specify a policy for workahead processes. In the description of SLS (Section 4.3), we assumed that the KLS arbitrarily chooses a VAS with a runnable LWP when there are no critical LWPs. The policy that the ULS used to schedule workahead LWPs was left unspecified.

A number of workahead policies are possible. The contents of the *usched* and *ksched* areas would change from our description, depending on the policy adopted. For concreteness, we consider a workahead policy that selects the workahead LWP with the globally earliest critical time. Such a LWP is "urgent" in the sense that it has the smallest time to criticality.

To support a split-level scheduler with this policy, we simply need to add a field to the *usched* area containing C_A, and add a field to the *ksched* area containing $C_{\bar{A}}$. The ULS calls

`yield()` if there are no critical LWPs (*i.e.* D_A and $D_{\bar{A}}$ are both $+\infty$) and $C_A > C_{\bar{A}}$. Similarly, the KLS preempts the current VAS A if there are no critical LWPs and $C_{\bar{A}} < C_A$. $C_{\bar{A}}$ is easily computed from *usched.L* of VAS's in $\bar{A}$.

4.3.6.2. Adding support for non-real-time policies

CM tasks may, in general, contain a mixture of process types (real-time, interactive and background processes). SLS can support VASs which contain different process types. In this section, we illustrate how interactive processes can be supported by the split-level DWS scheduler.

We consider the policy for interactive processes that assigns static priorities to each process. Processes are preemptively scheduled and are time-sliced within a given priority level. A process may sleep for a specified period of time or wait for I/O completion.

To add this interactive process scheduling policy to the split-level DWS scheduler, we add two fields to the shared memory region: to the *usched* area, we need to add J_A (the highest priority among interactive LWPs in VAS A) and to the *ksched* area we add $J_{\bar{A}}$ (the highest priority among interactive LWPs in $\bar{A}$). In addition, the list L in the *usched* area also contains a list of sleeping interactive LWPs, their wakeup times and wakeup priorities. For each interactive LWPs suspended waiting on an I/O descriptor, the *usched* area contains its priority on wakeup.

If there are no critical LWPs (*i.e.* D_A and $D_{\bar{A}}$ are both $+\infty$), the ULS schedules the highest priority LWP. It also sets T_{next} (see Section 4.3.3) to the lesser of $T_{critical}$ and $T_{now}+Q_{interactive}$, where $Q_{interactive}$ is the quantum size for interactive processes. The ULS calls `yield()` if (1) $J_A < J_{\bar{A}}$ or (2) $J_A = J_{\bar{A}}$ and each LWP in its VAS at level J_A has already received one quantum.

The KLS "tracks" changes in J_B for VASs B in $\bar{A}$. It sets timers to wake up sleeping interactive LWPs and monitors I/O descriptors that suspended interactive LWPs are waiting on. Whenever some J_B changes such that $J_{\bar{A}} > J_A$, the KLS preempts the current VAS A and schedules A^* (the VAS with the highest priority interactive LWP).

4.3.6.3. Different CPU scheduling policy

SLS is not restricted to deadline/workahead scheduling; it can be adapted to other policies, such as static priorities or usage-based timesharing policies. The policy dictates the contents of ULS/KLS shared memory; in general, the usched area contains the highest priority among runnable LWPs in the address space, while the ksched area contains the highest priority among runnable LWPs in other address spaces. In fact, the description in Section 4.3.6.2 can be adapted to design a split-level scheduler for preemptive priority scheduling with time-slicing at a given priority level.

SLS is applicable for purposes other than CM. Process-control applications (*e.g.*, [AIL86]) have scheduling requirements similar to those of CM.

4.3.7. UNIX Implementation of SLS

We implemented split-level scheduling in SunOS 4.1 (a UNIX-like operating system) for the SPARCstation 1+. One constraint in modifying the kernel was to allow existing non-realtime user programs to execute without recompilation. Changes to the kernel (Figure 4.5) involved adding new code paths (indicated by boxes surrounded by bold lines) or adding to or modifying existing code paths (indicated by shaded areas).

To add a new system call, we place a pointer to the system call handler in the UNIX system call table and implement that handler. The implementation of each system call handler is described in Section 4.3.4.

A CM program begins execution as an ordinary UNIX process. During its initialization phase, the ULS calls `init_sls()` which, apart from creating the shared memory region, marks the calling process "real-time". An unused flag in the `proc` structure (the kernel data structure for UNIX processes) is used for this purpose. When the ULS calls `done_sls()`, this flag is reset.

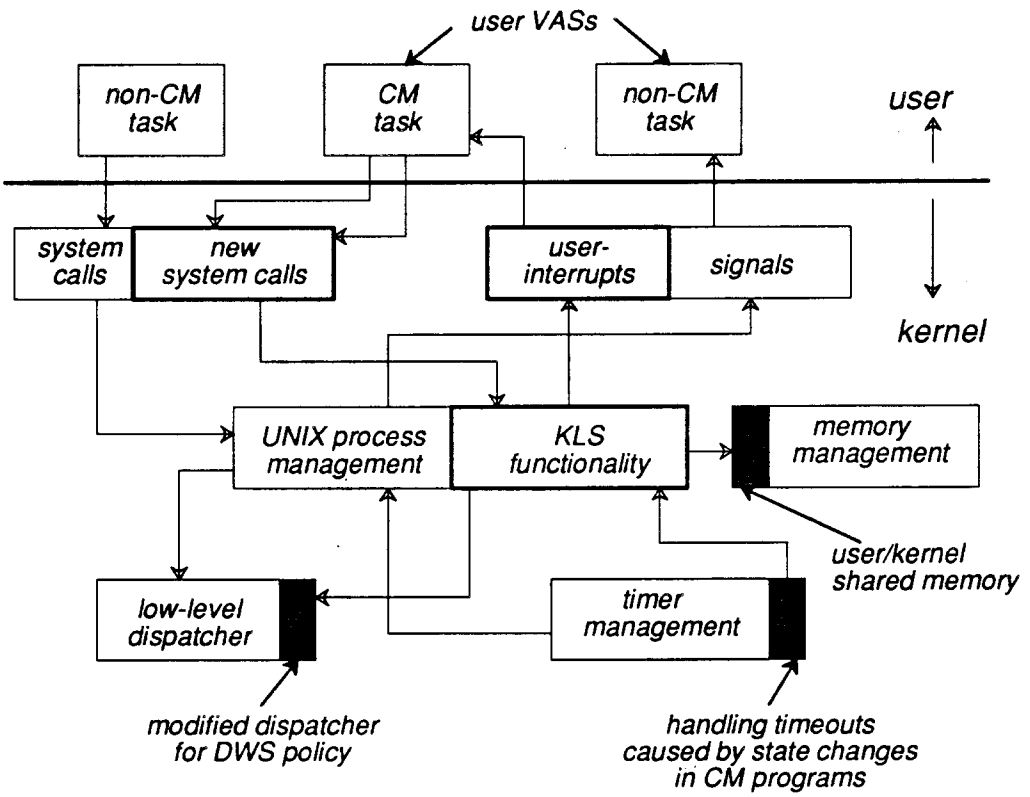


Figure 4.5. UNIX Implementation of SLS.

Implementation of split-level scheduling in UNIX modified or added to a number of different parts of the operating system. The bold boxes represent new functionality and the shaded regions represent additions to existing functions.

The KLS maintains a doubly linked list of real-time process descriptors. Each such descriptor contains a pointer to the `proc` structure, the address of the usched and ksched areas, the I/O descriptors associated with the VAS, a list of pending user-interrupts, and the user addresses of the user-interrupt handlers. The head of the descriptor list is the real-time process containing the globally highest-priority LWP according to the DWS policy.

Real-time processes are similar to ordinary UNIX processes except in one respect: they are scheduled differently. A real-time process is never placed in the UNIX run queue. To satisfy this invariant, a number of kernel functions (trap handlers, signal-related functions, the software interrupt handler that periodically alters UNIX process priorities) now check whether a process is real-time before inserting it into the run queue.

To change the OS CPU scheduling policy to deadline/workahead scheduling, it suffices to alter the low-level dispatcher, `sched()`. In UNIX, `sched()` scans the run queues in priority order to select the highest priority runnable UNIX process. The modified `sched()` first checks the real-time queue and schedules the UNIX process at the head of the queue if it is critical. Otherwise, it checks the UNIX run queues to schedule a runnable non-real-time UNIX process if any (all non-real-time UNIX processes are treated as interactive processes, to simplify the

implementation). Else, `sched()` schedules the UNIX process at the head of the real-time queue; this must contain workahead processes.

For the `init_sls()` system call, the memory management module now exports additional interface functions to create and destroy the shared memory region. The creator function allocates a region of virtual addresses in user space of the specified size, locks the corresponding physical pages in memory and maps those physical pages to a region of virtual addresses in kernel space.

To simplify the implementation, a real-time process is locked into memory when its ULSs call `init_sls()`. This also avoids the problem of priority inversions due to inopportune page faults. As a further simplification, real-time processes are not demand paged but pre-loaded to avoid page faults on startup.

The kernel clock interrupt handler (`hardclock()`) updates the shared memory time-of-day clock and delivers an `INT_TIMER` user interrupt to the currently executing real-time UNIX process if necessary. Kernel timer management also detects sleeping/critical or workahead/critical LWP state changes.

Finally, we modified the UNIX signal mechanism to implement user-interrupts. In SunOS, the signal trampoline code [LMK89] calls `sigcleanup()` after returning from the user-interrupt handler. `Sigcleanup()` modifies the signal mask and restores SPARC register windows [SSS87]. To simplify our implementation of user-interrupts, we assumed that real-time processes do not use register windows. Also, in SLS, the user-interrupt mask is maintained in the usched area (see Section 4.3.5). Therefore, the user-interrupt trampoline code can modify the shared memory interrupt mask at user level and return directly to the interrupted LWP without calling `sigcleanup()`.

Chapter 5

MEMORY-MAPPED STREAMS

This chapter describes *memory-mapped streams*, a mechanism that reduces or eliminates user/kernel interactions for CM stream communication between user and kernel and between user and user VASs. Section 5.1 motivates the need for an efficient mechanism for such communication. Section 5.2 describes the client interface to memory-mapped streams (MMSs) and Section 5.3 discusses the implementation of MMSs in detail.

5.1. Motivation

We use *stream transfer* to denote the simplex communication of CM streams between user and kernel and between user and user VASs. Producers and consumers of CM streams may be user processes or I/O devices (disks, the network, D/A convertors, etc.). There are three types of stream transfers: user to kernel (*UK*), kernel to user (*KU*) or user to user (*UU*). A mechanism for stream transfer has three components:

- *Synchronization:* Producers may wait for data consumption before generating more CM data. A consumer must notify a waiting producer of data consumption. Similarly, producers may need to notify waiting consumers of data production. Such producer/consumer synchronization may also involve initiating I/O device data transfers if either the producer or the consumer is an I/O device.
- *Data location transfer:* Addresses of buffers used to transfer stream data can be determined by producers or consumers. This component communicates changes in data buffer addresses from user to kernel or vice versa.
- *Data transfer:* This component performs actual transfer of data, perhaps by copying or VM remapping.

In our model of CM processing (Chapter 2), each stream transfer consists of a sequence of *message transfers*. As with stream transfers, there are also three types of message transfers (UK, KU and UU). Each message transfer may perform one or more of the components of a stream transfer.

There are two commonly-used techniques for message transfers:

- *Read/write system calls:* A system call is used for each message transfer. In UNIX, a KU message transfer is performed using the `read()` system call. This does data location transfer (the caller indicates the buffer in user space where the data is to be placed) and data transfer (the kernel copies the data to this buffer). It may also do synchronization (the call may block in the kernel if data is not available).
- *Asynchronous, non-blocking I/O:* This technique is similar to the above. However, when a message transfer involves synchronization, the call does not block, but returns immediately. Completion of the message transfer is indicated later by an asynchronous user event (*e.g.*, a UNIX signal).

There are two drawbacks to using these techniques for CM stream transfer. One is reduced flexibility for user programs: a blocking system call suspends all activity (*e.g.*, concurrent user-level LWPs) in a user VAS. The other is poor performance: these techniques require one or more user/kernel interactions for each message transfer (Chapter 3).

MMSs are a class of mechanisms for UK, KU and UU stream transfers. MMSs use shared memory for control and synchronization. MMSs may use any of a number of techniques for data

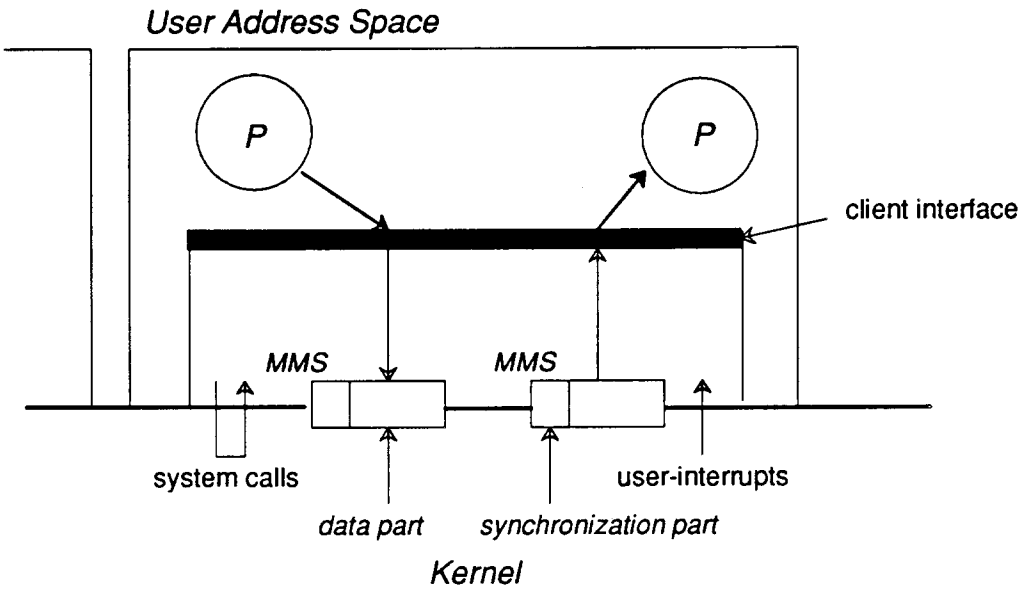


Figure 5.1. Memory-mapped stream interfaces.

The client interface (Section 5.2) to MMSs is provided by a user-level library. This library is implemented using a user/kernel interface (Section 5.3.1) that consists of three elements: system calls, user-interrupts and shared memory.

location transfer; all these use shared memory to hold either the data itself or the data location (with each technique one or more data transfer mechanisms are possible, see Section 5.3.1.1). This combination of shared memory mechanisms reduces or eliminates user/kernel interactions in I/O operations.

There are three types of MMSs (KU, UK and UU) corresponding to the type of stream transfer the MMS performs. Each MMS has a *producer* and a *consumer*. A KU MMS's producer is an I/O device (*e.g.*, disk, D/A convertor) and its consumer is a user process. The situation is reversed for a UK MMS. A UU MMS's producer and consumer are user processes in different VASs.

5.2. Interface to Memory-Mapped Streams

From a CM program's point of view, an MMS is a byte-oriented, sequential, memory-mapped mechanism for sending and receiving CM streams. The client interface to MMSs is provided by a user-level library (Figure 5.1). Table 5.1 summarizes this interface.

An application creates and destroys MMSs using the following library calls:

```
MMS_DESCRIPTOR
MMS_create(
        int     fd,
        int     buffer_size
)

MMS_delete(
        MMS_DESCRIPTOR d
```

```
)
```

`fd` represents the data source or sink. `Buffer_size` indicates the maximum size of a CM stream that may be memory-mapped at any given instant. We now discuss how these parameters are determined.

- *Determining* `fd`: `fd` identifies the operating system abstraction for the source or sink of the CM stream (network connection, disk file, user-user IPC connection, and so on). It also identifies the type of the MMS (KU, UK or UU).

  We do not specify how the CM program obtains this identifier. For UU MMSs, in particular, we do not specify the handshaking steps needed to obtain the identifier for a user-user IPC connection. The socket create-bind-connect paradigm in 4.3BSD UNIX [LMK89] is one possible approach. Once this identifier is obtained, both the producer and the consumer must call `MMS_create()`.

- *Determining* `buffer_size`: Since access to MMSs is sequential and since CM streams may be non-persistent (*e.g.*, a stream generated by an A/D convertor and, finally, consumed by a D/A convertor), it is not necessary to provide buffer space to memory map the entire CM stream. Instead, the delay bound of a stream determines buffer size. In streams that have high end-to-end delay bounds (*e.g.*, a second or more), large buffers may be used. Processes involved in these streams may "work ahead" in these buffers, increasing the overall efficiency and responsiveness of the system. In streams that have low end-to-end delay bounds, this workahead is not possible.

  Essentially, non-workahead streams are those that are part of an inter-human conversation or conference; workahead streams are those in which a storage device sources or sinks data. The workahead status of a process may change dynamically; for example, the ACME audio output process is non-workahead if any of the streams it is currently handling is part of a conversation; otherwise it is workahead.

  A user process may receive a CM stream from an MMS using the following library routines:

```
MMS_canrcv(
      MMS_DESCRIPTOR  d,       // IN
      int             n,       // IN
      POINTER_LIST    *plist   // OUT
);

MMS_rcvdone(
      MMS_DESCRIPTOR  d,
      int             n
);
```

`MMS_canrcv()` checks whether the next `n` bytes of the stream are memory-mapped. If not, it blocks the caller. It returns a list of pointers (`plist`) to blocks that contain those bytes. The caller can then directly access the data. `MMS_rcvdone()` indicates that the caller has "consumed" the next `n` bytes of the stream. The caller cannot consume more of the stream than is memory-mapped at the time of this call.

Symmetrically, a user process sending a CM stream on an MMS uses the following library routines:

```
MMS_cansend(
        MMS_DESCRIPTOR  d,       // IN
        int             n,       // IN
        POINTER_LIST    *plist   // OUT
);

MMS_senddone(
      MMS_DESCRIPTOR  d,
      int             n
);
```

`MMS_cansend()` checks whether `n` bytes may be written to the MMS. If not (this may happen because the MMS buffer is full, for instance), the call blocks until the write is possible. It returns pointers (in `plist`) to regions in memory where the `n` bytes should be written. `MMS_senddone()` indicates that the caller has "produced" the next `n` bytes of the stream. The caller cannot produce more data than the amount of unused memory-mapped MMS buffer at the time of the call.

5.3. Implementation of Memory-Mapped Streams

Section 5.3.1 describes the user/kernel interface required to implement the client interface. Section 5.3.2 discusses the implementation of the client interface when the user-level producers and consumers of MMSs are split-level scheduled LWPs. Finally, Section 5.3.3 details the implementation of the user/kernel interface.

5.3.1. User/Kernel Interface

As with split-level scheduling (Chapter 4), the user/kernel interface for memory-mapped streams consists of three elements (Table 5.2, Figure 5.1): user/kernel shared memory, system calls and user-interrupts. In this section, we discuss each of these elements in greater detail.

5.3.1.1. User-kernel shared memory

An MMS is a region of memory shared read-write between user and kernel (for KU and UK transfers) or user and user (for UU transfers) VASs. This shared memory region consists of two parts: a *synchronization* part and a *data* part.

The synchronization part of an MMS consists of the following fields:

- N_{read}: the number of bytes read from the MMS so far; this is updated by the consumer.
- N_{write}: the number of bytes written to the MMS so far; this is updated by the producer. The buffer is empty when $N_{read} = N_{write}$, and full when they differ by the buffer size.
- `pwait`: a flag maintained by the producer. If true, the producer is awaiting notification of data consumption from the MMS. In the KU case, a system call (see Section 5.3.1.2) is used for this notification. In the UK case, a user-interrupt delivers this notification.
- `cwait`: a flag maintained by the consumer. If true, the consumer is awaiting notification of data production in the MMS.

MMS_create()	set up memory-mapped stream
MMS_delete()	destroy memory-mapped stream
MMS_canrcv()	are the next *n* bytes of the stream memory-mapped?
MMS_rcvdone()	done processing a message
MMS_cansend()	is enough of the MMS memory-mapped for *n* bytes?
MMS_senddone()	done sending a message

Table 5.1. Client Interface to MMSs.

An MMS is a byte-oriented, sequential, memory-mapped mechanism for CM stream transfer. The client interface provides functions for creating an MMS and for sequentially sending or receiving memory-mapped chunks of CM streams.

- B_{pwait}: value for notification of the `pwait` condition, set by the producer. If the data level in the MMS falls below this value and the `pwait` flag is set, the `pwait` notification is delivered to the producer.
- B_{cwait}: value for notification of the `cwait` condition, set by the consumer. If the data level in the MMS is greater than this value and the `cwait` flag is set, the `cwait` notification is delivered to the consumer.

The data part performs shared memory data location transfer. The structure of this part is largely independent of the structure of the synchronization part and determines whether data copying for data transfer can be avoided. Some possible organizations for the data part are (Figure 5.2):

A) Data is passed in pages of physical memory that are statically shared between kernel and user. Data location is implicit. This structure is suitable for CM I/O devices: they may transfer data via DMA from the MMS and avoid a kernel copy.

B) Data is passed in a fixed range of virtual pages that are mapped dynamically to physical pages. Data location is implicit. The kernel can allocate pages from a physical page pool. Once data transfer is complete, it can dynamically map the pages to the data part. This structure is advantageous for page-sized disk transfers if the cost of copying a page is greater than the cost of VM remapping the page.

C) In the most general configuration, the kernel and user share an array of "message descriptors" that contain pointers to blocks of data. Data may be transferred by remapping, by copying, or by copy-on-write. This configuration is suitable for network input: following protocol processing, the kernel need not copy data, but instead simply map the page(s) containing the data to the user VAS and set the descriptors appropriately. Each descriptor contains the start and end addresses of the memory-mapped block of data.

5.3.1.2. System Calls and User-Interrupts

Corresponding to the library routines `MMS_create()` and `MMS_delete()`, there exist system calls:

```
MMS_DESCRIPTOR
MMS_create(
      int          fd,               // IN
      int          buffer_size,      // IN
      VIRT_ADDR    *synch_part       // OUT
      VIRT_ADDR    *data_part        // OUT
      int          *data_part_type   // OUT
);

MMS_delete(
      MMS_DESCRIPTOR  d
);
```

The `MMS_create()` system call allocates and initializes the synchronization and data parts in user VAS. It returns the user addresses of the two parts, the structure of the data part, and a descriptor for the MMS. This information is used subsequently by library routines that access MMSs. The `MMS_delete()` system call deallocates the synchronization and data parts associated with the specified MMS.

The

```
notify(
      MMS_DESCRIPTOR  d
)
```

system call does `pwait` notification in KU stream transfers and `cwait` notification in UK stream transfers. It is used for both types of notification in UU stream transfers (Section 5.3.3).

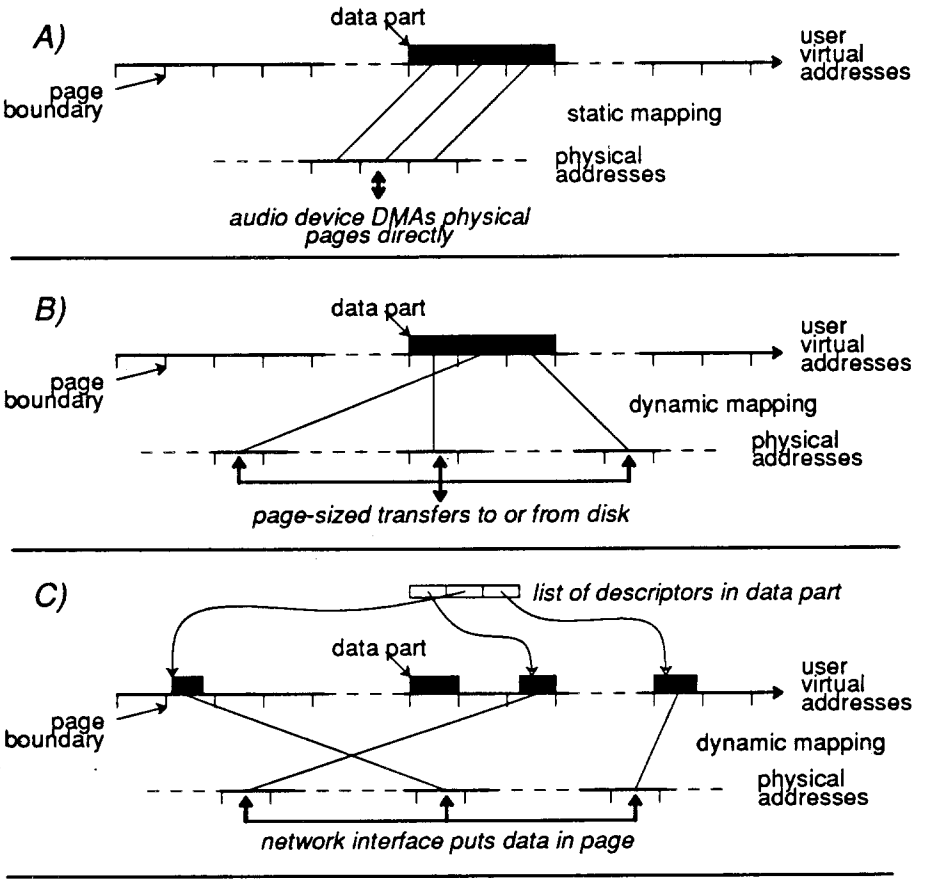


Figure 5.2. The MMS data part.

There are three possible structures for the data part. In A), the data part consists of physical pages statically shared. In B), a range of virtual addresses is shared and the kernel dynamically updates the mappings associated with each virtual page. In C), the data part is an array of descriptors to memory-mapped blocks of CM data.

Similarly, the `INT_NOTIFY` user-interrupt may notify the `pwait` condition in UK stream transfers and the `cwait` condition in KU stream transfers. It does both types of notification in UU stream transfers (Section 5.3.3).

5.3.2. Library Implementation

In this section, we discuss the implementation of the `MMS_canrcv()` and `MMS_rcvdone()` library functions for a user LWP receiving a CM stream on an MMS (Figure 5.3). We assume that the LWP is split-level scheduled; the description will change slightly if user processes are scheduled differently. The implementation of `MMS_cansend()` and `MMS_senddone()` are similar.

Synchronization part of MMS:	
N_{read}	total (in bytes) of stream read from MMS
N_{write}	total (in bytes) of stream written to MMS
pwait	producer is awaiting notification of data consumption
cwait	consumer is awaiting notification of data production
B_{pwait}	water mark for pwait condition
B_{cwait}	water mark for cwait condition
System Call:	
MMS_create()	allocate and initialize synchronization and data parts
MMS_delete()	deallocate synchronization and data parts
notify()	notify pwait for KU transfers, cwait for UK transfers
User-interrupt:	
INT_NOTIFY	notify cwait for KU transfers, pwait for UK transfers

Table 5.2. The MMS user/kernel interface.

The user/kernel interface for MMSs consists of system calls to create and delete MMSs and a system call and a user-interrupt for notification. An MMS itself is a region of shared memory that consists of two parts: a data part and a synchronization part. The latter contains shared information that reduces user/kernel interactions for producer-consumer synchronization.

The algorithms for the two functions are:

```
MMS_canread(d, n, plist) {
    mask_user_interrupts();
    Bwakeup = n;
    WaitingForIO = TRUE;
    w = Nwrite;                              // X
    if (w - Nread < n)
      IO_wait(d, ...);                      // Y, see Chapter 4
    if ((w - Nread < Bpwait) && pwait)
      notify();
    WaitingForIO = FALSE;
    compute plist, the list of memory-mapped stream data blocks
    unmask_user_interrupts();
}

MMS_readdone(d, n) {
    mask_user_interrupts();
    Nread += n;
    unmask_user_interrupts();
}
```

This code executes at user level, so device I/O interrupts cannot be masked (`mask_user_interrupts()` merely inhibits the delivery of `INT_NOTIFY` user interrupts; see Chapter 4). There is a potential race condition if a device inserts some CM data into the MMS between **X** and **Y** in the code above. This race condition is avoided, however, by setting *WaitingForIO*; if data arrival occurs during the critical period, it will simply set the `INT_NOTIFY` request flag and the descriptor's *ReadyForIO* flag. The ULS will check these flags when it unmasks user interrupts in `IO_wait()`, and will awaken the LWP that called `MMS_canread()` if

necessary. The LWP's *WaitingForIO* flag is identical to the MMSs' `cwait` flag.

For high delay bound streams, the LWP may not need to immediately notify a `pwaiting` producer after consuming a message. That is because the producer may have worked ahead on the CM stream. In such cases, if B_{pwait} is set to the size of a non-zero number of messages, the cost of the `notify()` call is amortized over that number of CM messages.

5.3.3. Kernel Implementation

Suppose that in KU stream transfer, a split-level scheduled process *P* receives a CM stream on an MMS *M*. *M*'s producer is a device (*e.g.* the audio input device or a network interface), which repeatedly activates a kernel device interrupt handler to insert CM data into *M* (Figure 5.3). To insert `n` bytes, the handler executes the following steps:

```
if (n bytes cannot be memory-mapped into M) {
        disable device interrupts;
        M.pwait = 1;
      return;
}
insert message into data part;  // this depends on the
                                // structure of the data part
M.N_write += n;
if (P.WaitingForIO)
        if (M.N_write - M.N_read > M.B_cwait) {
                P.ReadyForIO = TRUE;
                if (C_P<T_now and D_P<D_A)
                      deliver INT_NOTIFY user interrupt to VAS
        }

schedule interrupt for next message transfer
```

The logic for a device interrupt handler doing UK stream transfer is similar.

Each device interrupt handler schedules a chain of interrupts to perform message transfers to or from the MMS buffer. This chain of interrupts is stopped when the kernel is waiting for `pwait` (in the KU case) or `cwait` (in the UK case) notification. In these cases, the `notify()` system call restarts the chain of device interrupts.

Finally, we consider the case where LWP *P* is the producer and LWP *Q* is the consumer in a UU stream transfer. *P* and *Q* are in different VASs and communicate using a UU MMS *M*. *P* may call `notify()` when *Q* is `cwaiting`. Similarly, *Q* may call `notify()` when *P* is `pwaiting`. In the former case, the kernel does the following:

```
if (M.N_write - M.N_read > M.B_cwait) {
      Q.ReadyForIO = TRUE;
      if (C_Q < T_now and D_Q < D_A)
          deliver INT_NOTIFY user interrupt to VAS
}
```

The logic for the latter case is similar.

5.3.4. UNIX Implementation of MMS

We implemented UK and KU memory-mapped streams in SunOS 4.1 for the SPARCstation 1+. In our implementation, we assumed that user-level producers or consumers were split-level scheduled LWPs. A user-level library implemented the functionality described in Section 5.3.2. The kernel modifications for the MMS implementation are shown in Figure 5.4.

Adding a new system call is done as described in Chapter 4. We did not need to add new user-interrupts: the `INT_NOTIFY` user-interrupt is the same as the `INT_IO_READY` user-interrupt already supported by the split-level scheduler.

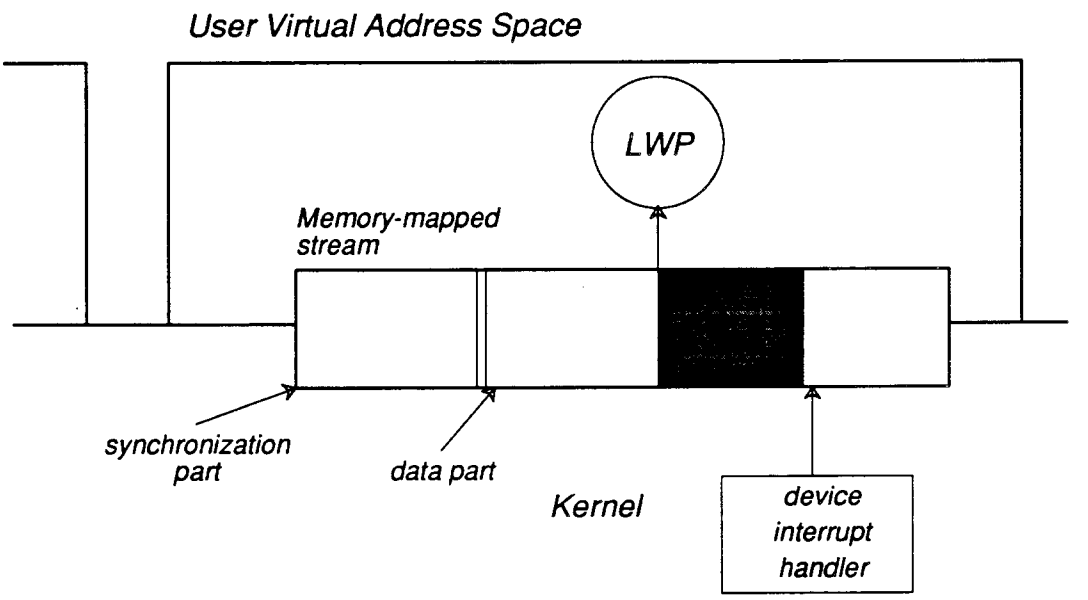


Figure 5.3. Illustration of KU stream transfer.

This figure illustrates a KU MMS, whose consumer is a SLS LWP and whose producer is an I/O device. The device schedules a chain of interrupts to memory map CM stream data into the MMS, and the LWP reads the memory-mapped data directly.

The memory management module now supports additional functionality to allocate and map the synchronization and data parts of an MMS. Our implementation supports the "shared physical pages" data part structure only (structure A in Figure 5.2). This additional functionality was a simple extension to the functionality necessary to set up the shared memory region in SLS.

Finally, the audio and video frame buffer device interrupt handlers now support direct message transfers to and from MMSs, as described in Section 5.3.3. The logic to detect `pwait` and `cwait` conditions and deliver notifications is also included.

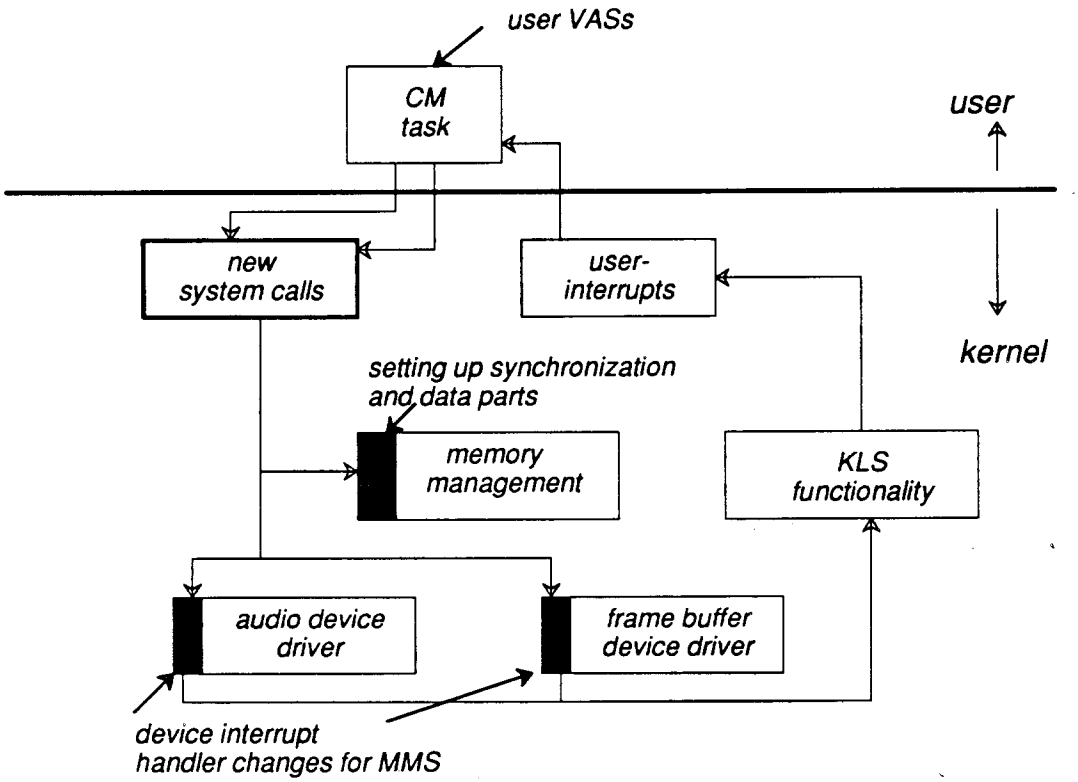


Figure 5.4. UNIX Implementation of MMS.

Our implementation of memory-mapped streams in UNIX modified or added to a number of different parts of the operating system. The bold boxes represent new functionality and the shaded regions represent additions to existing functions.

Chapter 6

SHARED MEMORY CONCURRENCY CONTROL

In split-level scheduling and memory-mapped streams, user/kernel shared memory may be read or written by LWPs, user-interrupt handlers and kernel device interrupts. Conflicting, concurrent reads and writes can produce erroneous results. For example, if $D_{\bar{A}}$ in the ksched area is a multi-word quantity, `time_advance()` may read an incorrect value for $D_{\bar{A}}$. That can happen if an interrupt handler changes $D_{\bar{A}}$ while it is being read.

This chapter discusses concurrency control mechanisms for user/kernel shared memory accesses. These mechanisms fall into two categories: *pessimistic* concurrency control mechanisms prevent concurrent, conflicting accesses to shared memory, while *optimistic* mechanisms detect such accesses and do recovery. Section 6.1 lists two ways in which conflicting, concurrent shared memory reads and writes may occur. Sections 6.2 and 6.3 describe concurrency control mechanisms to handle such accesses.

6.1. User-mode Critical Section Violations

Both user-mode and kernel-mode execution may read or write user/kernel shared memory. A sequence of reads and writes to shared memory from kernel mode is a critical section with respect to user mode reads and writes to shared memory. We distinguish two levels of user-mode execution: the *process* level and the *user-interrupt* level. A sequence of reads and writes to shared memory from either level is a critical section with respect to reads and writes from the other level or from kernel mode.

In this chapter, we assume that there exist pessimistic concurrency control (*i.e.*, *mutual exclusion*) mechanisms for kernel-mode critical sections. In the UNIX kernel, raising device interrupt priority level for the duration of the critical section is an example of such a mechanism.

Our goal is to design efficient concurrency control mechanisms for user-mode critical sections. Mutually exclusive access to user-mode critical sections may be violated in two ways (Figure 6.1):

(1) When a user-interrupt interrupts process-level execution (A in Figure 6.1).

(2) When kernel-mode execution (*e.g.* a kernel device interrupt handler) interrupts user-mode execution (B and C in Figure 6.1).

In the following sections, we describe efficient mechanisms for shared memory concurrency control in these two cases.

6.2. Preventing User-Interrupts During Process-level Execution

When a LWP calls `time_advance()`, that function may change *usched.*D_A and *usched.L* (Section 4.3.3). After `time_advance()` has updated *usched.*D_A and before it changes *usched.L*, a user-interrupt may be delivered to the VAS. This user-interrupt may in turn modify these two fields, leaving the usched area in an inconsistent state.

Virtual user-interrupt masking is an efficient technique for preventing concurrent shared memory accesses by process and user-interrupt execution during critical sections. The technique uses two bitmaps in shared memory to communicate masking information between user and kernel: 1) an *interrupt mask* in the usched area which, if nonzero, masks user-interrupts, and 2) an *interrupt request word* in the ksched area which flags user-interrupts that occur while masking is in effect.

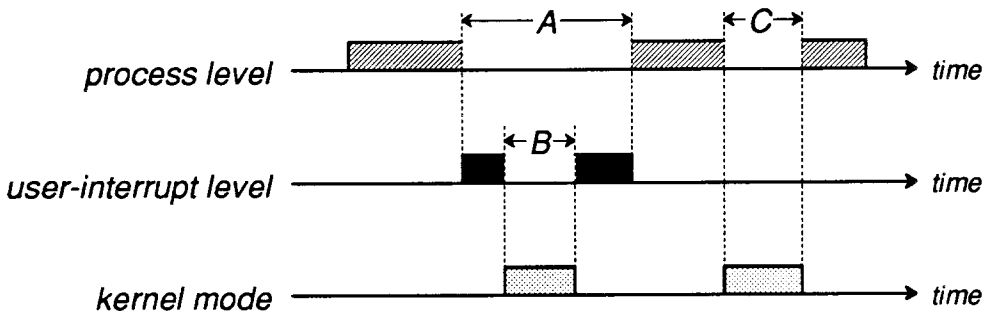


Figure 6.1. User-mode critical section violations.

Process level execution can be interrupted by user-interrupt or by kernel-mode execution (A and C). User-interrupt level execution can only be interrupted by kernel-mode execution (B). These introduce the potential for violating mutual exclusion in user-mode critical sections.

The beginning of a user-mode critical section is marked by a call to `mask_ints()` (Figure 6.2). A critical section ends after the return from `unmask_ints`. When the KLS wants to deliver a user-interrupt to the currently executing VAS, it calls `deliver_ints()`. If the interrupt mask is nonzero, `deliver_ints()` requests a user-interrupt (by setting the request flag). `User_interrupt_handler()` services user-interrupts. If it is called from `unmask_ints()`, `user_interrupt_handler()` services all user-interrupts requested during the period that user-interrupts were masked.

Claim 6.1. *The following assertions hold for all time: 1) user-interrupts do not occur when the interrupt mask is nonzero, and 2) each user-interrupt request is serviced exactly once.*

Proof. We show below that these assertions hold for non-nested critical sections (*i.e.*, from a call to `mask_ints()` until the return from `unmask_ints()`, with no intervening calls to these functions). Since the assertions are trivially true when user-mode execution is not in a critical section, the assertions hold for all time.

Assertion 1 holds because `deliver_ints()` does not deliver a user-interrupt when the interrupt mask is nonzero. To prove that assertion 2 holds, we note that a non-nested critical section (*S* in Figure 6.3) consists of a single execution of section **A** followed by one or more executions of sections **B** and **C** (Figure 6.2).

- During **A**, `deliver_int()` may set a flag in the interrupt request word. Repeated user-interrupt requests do not alter the flag value.
- If a user-interrupt occurs during **B**, the kernel delivers it to the VAS. Then, any user-interrupts that were requested during **A** are also serviced; the `user_interrupt_handler()` resets the corresponding interrupt request flag so that the handler is not subsequently called.
- During **C**, the interrupt mask is set and the `for` loop services any user-interrupts that may have been requested during **A** but which were not serviced during **B**. After a handler is called but before the end of **C**, a user-interrupt may be requested; this is registered in the interrupt request word. Then, sections **B** and **C** are repeated and the interrupt is serviced in the next iteration of the `while` loop.

From the above, any user-interrupt that occurs during **A** is serviced either during **B** or **C**. Moreover, an interrupt that occurs during **B** is serviced immediately. Finally, if an interrupt occurs

```
mask_ints() {      // Process-mode
    interrupt_mask++;
                                        // Start A
}

unmask_ints() {    // Process-mode
                                        // End A
    interrupt_mask--;
                                        // Start B
    while (!interrupt_mask && interrupt_request) {
                                        // End B
        interrupt_mask = 1;
                                        // Start C
        for (all interrupts requested in the interrupt request word)
            user_interrupt_handler();
                                        // End C
        interrupt_mask--;
    }
}

deliver_ints() {   // Kernel-mode
    if (interrupt_mask)
       set appropriate flag in interrupt_request word
    else
       deliver user interrupt to VAS
}

user_interrupt_handler() {   // User-mode
    ... service the interrupt ...
    if (flag is set in interrupt request word)
       reset flag;
}
```

Figure 6.2. Virtual user-interrupt masking.

A user LWP or a user-interrupt handler calls `mask_ints()` before and `unmask_ints()` after a critical section. The KLS calls `deliver_int()` to deliver a user-interrupt to the currently executing VAS. The user-interrupt handler may be called when a user-interrupt is delivered or when `unmask_ints()` finds an interrupt request.

during **C** but after its handler was called, it is serviced during the next **B** or **C**. Thus, assertion 2 also holds.

Finally, suppose critical section *R* is nested inside critical section *S* in Figure 6.3. The `if` conditional in section **B** of *R* always evaluates to false, because the interrupt mask is non-zero. Therefore, *R* only consists of a single execution of **A**. We can easily show that the above assertions hold for nested critical sections as well. □

A user-interrupt may not interrupt user-mode execution while a user-interrupt is being serviced. In our implementation a user-interrupt does not make a system call after executing the handler. Therefore, the kernel must be able to detect when user-mode execution returns to process-level after servicing a user-interrupt. Virtual user-interrupt masking works in this case as well. The kernel sets the interrupt mask before delivering a user-interrupt. When the

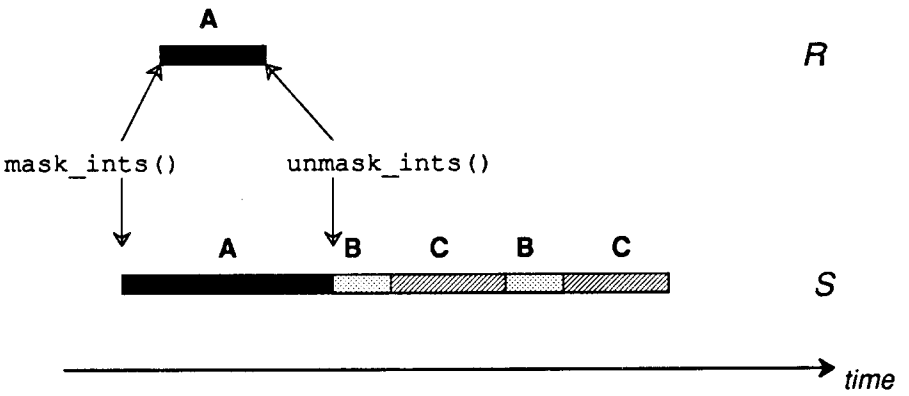


Figure 6.3. Critical section.

A non-nested critical section *S* consists of a single section **A**, followed by one or more sections **B** and **C**. If *R* is nested in *S*, then *R* consists of a single section **A**.

`user_interrupt_handler()` completes, it calls `unmask_ints()` before returning to process level.

6.3. Concurrency Control Between User-mode and Kernel-mode Execution

A kernel device interrupt (*e.g.*, a clock interrupt) may cause user VAS preemption while the latter is in a critical section. Section 6.3.1 describes a pessimistic concurrency control mechanism for this case.

An interrupt handler may read from the usched area (*e.g.*, D_A, to decide whether to preempt the current VAS) or may write to the ksched area ($D_{\bar{A}}$, if some LWP's state was changed by the interrupt handler) while user-mode execution is in a critical section. We distinguish three situations:

- A device interrupt handler may modify a multi-word quantity (*e.g.*, $D_{\bar{A}}$) while a user-mode access is reading the quantity (Section 6.3.2).
- An interrupt handler may read a multi-word quantity (*e.g.*, D_A) while it is being written in user-mode (Section 6.3.3).
- Finally, an interrupt handler may execute between a read and a related write (*e.g.* in `MMS_canread()`, Section 5.3.2).

For each of these cases, Sections 6.3.2, 6.3.3 and 6.3.4 describe optimistic concurrency control mechanisms.

6.3.1. Preemption Masking

A kernel device interrupt may change the state of a LWP in some VAS in $\bar{A}$ (*A* is the current VAS). As a result, *A* may be preempted in a critical section. An `INT_PREEMPT` user-interrupt is delivered to *A* when it resumes execution. That interrupt handler may overwrite shared memory, leaving it in an inconsistent state. To prevent this, we need a *VAS preemption masking* mechanism.

"Virtual" masking can also be used to implement this mechanism, using a *preemption mask* flag in the usched area and a *preemption request* flag in the ksched area. While the mask is

nonzero, the VAS cannot be preempted by another VAS. Upon unmasking preemption, if the ULS finds the request flag set, it calls `yield()`. The correctness proof for virtual preemption masking is similar to that of Claim 6.1.

In summary, virtual masking is used in three different ways: for short critical sections in client code (Section 4.2), for preventing user-interrupts while process-level execution is accessing user/kernel shared memory (Section 6.2) and for masking VAS preemption. The algorithm described in Section 4.3.3 therefore masks both user-interrupts and preemption, but the `mask_LWP_preemption()` function only masks user-interrupts.

6.3.2. Multi-word Reads

User-mode accesses read a multi-word quantity m by reading the constituent words $w_1, w_2, \cdots, w_n$ (in that order, say). Let I be the interval of real time when m is read. Immediately after w_i is read, a kernel device interrupt may modify one or more of $w_{i+1}, \cdots, w_n$. The resulting value v for m may be inconsistent (*i.e.* m may not have had the value v in I).

It is possible to do virtual masking of kernel interrupts, but this has the drawback of requiring a system call to service interrupts that occur while kernel interrupts are masked. A more efficient solution is suggested by Claim 6.2.

Claim 6.2. *Let m be an n-word monotonic quantity. Further, let each word contain k bits. Suppose J is the worst case time to read m twice in succession and the value of m does not change by more than 2^{2k} in J. If two successive reads of m produce the same value v, and K is the interval of time in which these two reads are done, then m had a value v at some instant in K.*

Proof. We assume that m is read least significant word first and m is monotonically non-decreasing. The proof is similar when m is read most significant word first or if m is monotonically non-increasing. Let the n words (least-significant word first) be labeled $a_1, a_2, \cdots, a_n$ respectively (Figure 6.4). Let the first read for m be at times $a_1^1, a_2^1, \cdots, a_n^1$ and the second at $a_1^2, a_2^2, \cdots, a_n^2$. Further, let the values read for a_i be x_i, for each i. By our hypothesis, m changes by less than 2^{2k} in K and m is non-decreasing. Therefore, words a_2 through a_n are unchanged in the interval a_n^1–a_3^2. Since a_1 has the value x_1 at a_1^2, m has the value $x_n \cdots x_2 x_1$ at a_1^2. □

The ULS reads two multi-word variables from shared memory: $D_{\bar{A}}$ and T_{now}. These quantities are both either monotonically increasing or decreasing. Furthermore, for all practical CM applications on 32-bit architectures, these quantities do not change by more than 2^{64} in the time taken to read the quantity twice (a 64-bit microsecond resolution clock wraps around in half a million years, while reading a multi-word quantity twice is on the order of microseconds).

6.3.3. Multi-word Writes

User-mode execution writes a multi-word quantity m (*e.g.*,D_A and T_{next}) by writing the constituent words $w_1, w_2, \cdots, w_n$ in that order, say. If a kernel interrupt occurs immediately after w_i has been written, the interrupt handler may read an inconsistent value for m. As in Section 6.3.2, "virtually" masking device interrupts may be expensive. Our solution exploits specific properties of these quantities to arrive at an efficient mechanism for concurrency control.

For concurrency control, an interrupt handler assumes that the value of m may be inconsistent if it detects that the preemption mask is set (*i.e.* that user-mode execution is in a critical section). If a consistent value m were read, the interrupt handler might perform one of a set of actions depending on the value. However, if the preemption mask is set, our concurrency control mechanism either defers any action or performs the "worst case" action. We illustrate this mechanism for multi-word quantities D_A and T_{next}.

A device interrupt may change the state of a LWP in $\bar{A}$, thus changing $D_{\bar{A}}$ (where A is the current VAS). It then reads D_A to decide whether A should be preempted. If A is in a critical section, the interrupt handler writes $D_{\bar{A}}$ and sets the preemption request flag (the "worst case" action). When user-mode execution services the preemption request, it calls `yield()` only if its VAS does not have the highest priority LWP. Thus the VAS `yield()`s only if it would have been preempted anyway,

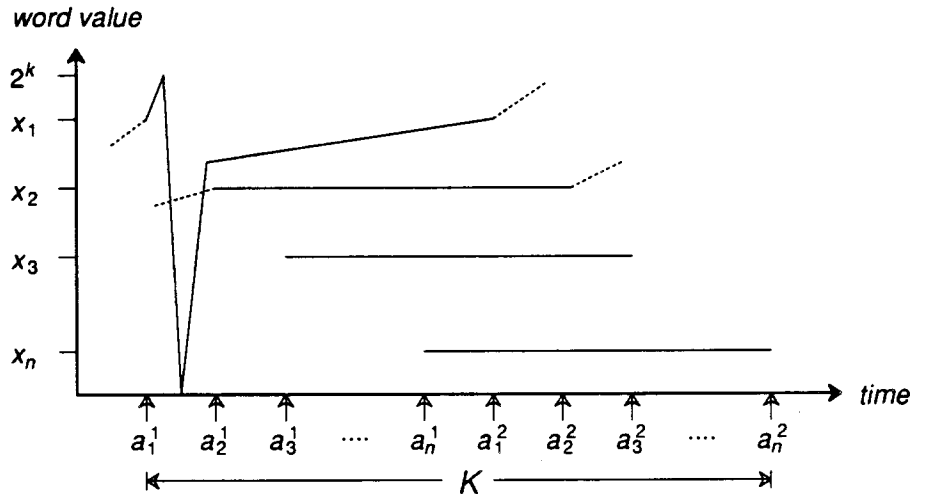


Figure 6.4. Multi-word reads.

Suppose a multi-word quantity m consists of n words $a_1, a_2, \cdots, a_n$. Let m be read twice in succession, with the jth word being read at a_j^1 and a_j^2 respectively. If the successive reads produce the same value, then m had that value at a_1^2.

When the kernel clock interrupt handler reads T_{next} to check if a user-interrupt should be delivered, user-mode execution may be in a critical section. If the preemption mask is set, the interrupt handler does not do the comparison, instead waiting until the next clock tick (*i.e.*, it defers the action). If, however, the clock granularity is large (~ 10 ms or more), or if the previous tick was skipped, the interrupt handler may set the timer request bit and let the ULS service the interrupt (the "worst case" action).

The correctness of the above solutions follows from the correctness of preemption masking and multi-word reads.

6.3.4. Read Followed By Related Write

A read followed by a related write may also cause conflicts, as in KU stream transfers (Section 5.3.2). If enough data is not available in the MMS, `MMS_canread()` requests I/O completion notification (by setting the MMS' *WaitingForIO* flag, Figure 6.5). If *WaitingForIO* is set, the kernel notifies the ULS of I/O completion by setting *ReadyForIO*. The kernel may also deliver an *INT_IO* user-interrupt. This notification may be missed if the I/O completion occurs just before the *WaitingForIO* flag is set, but after the ULS has checked for data availability.

To solve this problem, the ULS sets *WaitingForIO before* checking if sufficient data is present in the MMS. If data arrives after **A** and before **B** or after **B** and before **D**, then it may cause an unnecessary notification (since there may actually be enough data in the MMS that notification is not necessary). However, if data arrives between **B** and **C**, then the notification is registered in the *ReadyForIO* flag. Since interrupts are masked, no user-interrupt is delivered. When the ULS makes an LWP scheduling decision after **C**, it notices that the *ReadyForIO* flag is set and wakes up the LWP.

```
// Wrong
MMS_canread(d, n) {
    mask_ints();
    .....
    if (not enough data in MMS) // Read:
                                // If data completion occurs here, the
                                // notification may be missed
        IO_wait();              // Related write of WaitingForIO flag
    .....
    unmask_ints();
}

// Right
MMS_canread(d, n) {
    mask_ints();
    .....
    WaitingForIO = TRUE;                    // A
    if (not enough data in MMS)             // B
        IO_wait();                          // C
    WaitingForIO = FALSE;                   // D
    .....
    unmask_ints();
}

// On I/O completion, the kernel interrupt handler does the following
    ... add data to the MMS ....
    if (WaitingForIO) {
        ReadyForIO = TRUE;
        .....
    }
```

Figure 6.5. Missed I/O notifications.

A concurrency control mechanism may be necessary for when a user-mode read is followed by a related write. If an I/O operation completes after `MMS_canread()` executes the `if` statement (in the "wrong" version) but before the subsequent `IO_wait()`, then notification of the completion may be missed. The "right" version solves this condition by setting the *WaitingForIO* flag before the `if` statement.

Chapter 7

PERFORMANCE OF SPLIT-LEVEL SCHEDULING AND MEMORY-MAPPED STREAMS

A process state change may be voluntary or involuntary (*i.e.* preemptive), and may involve a process switch or a VAS switch as well. Similarly, a message transfer may involve synchronization and (for KU and UK transfers) I/O initiation.

Sections 7.1 and 7.2 enumerate the possible combinations of the above choices for process state changes and KU and UK message transfers (*i.e.*, I/O operations) respectively. These sections also list the cost of each such combination for three scheduling and I/O alternatives: 1) split-level scheduling and memory-mapped streams, 2) threads using system calls to read and write data, and 3) LWPs using asynchronous I/O.

Using these costs, we simulated the performance of different CM workloads. Section 7.3 presents these simulation results. The results describe the effect of factors such as the number of concurrent CM applications, the number of processes per CM application, process delay bounds, and CM stream message rates on scheduling and I/O costs of CM workloads. They also compare the performance of SLS and MMS with the other two alternatives described above.

7.1. Scheduling Paths and Their Costs

The globally highest-priority process in a system may change if the state of a process changes (from runnable to suspended, from workahead to critical and so on) or its scheduling parameters (critical time, deadline) change (Chapter 4). This change may be voluntary or preemptive and may involve only a process switch or a VAS switch as well. We call each combination of these choices a scheduling *path* (different from a *code path*, which is the sequence of user or kernel code executed for some operation):

(1) *Deadline change*: A process changes its own deadline, but continues to execute as the globally highest-priority process (P^*) in the system.

(2) *Non-preemptive process switch*: A process becomes suspended or changes its deadline so that it no longer has the highest priority. Either action causes a process switch. No VAS switch is incurred on this path; that is, the current VAS A still contains P^*.

(3) *Preemptive process switch*: An external event (I/O completion, timer expiration) awakens a suspended process, which now is P^*. The current process is preempted. No VAS switch is incurred on this path; the current VAS A still contains P^*.

(4) *Non-preemptive VAS switch*: A VAS yields the processor because it finds that all processes are suspended or that some other VAS is A^*.

(5) *Preemptive VAS switch*: The currently executing VAS is preempted in favor of another VAS containing the globally highest priority process.

In this section, we list the cost of each of these paths for the DWS policy for three scheduling techniques: threads, LWPs and split-level scheduled LWPs. For the first two alternatives, we estimated the cost using corresponding paths for preemptive priority scheduling. The costs are measured with user level experiments or by appropriately instrumenting the kernel.

Unless otherwise stated, all our experiments are performed on a 25Mhz diskless SPARCstation 1+ running SunOS 4.1.

7.1.1. Thread Scheduling Path Costs

A thread makes a system call during a *deadline change* path. The cost of this path is the average time taken for this system call.

In addition to the deadline change, a *non-preemptive thread switch* incurs a thread context switch. In our experiment, two threads changed their priorities such that each yielded the processor to the other. The cost of this path is half the average time taken to change one thread's priority.

We conducted the above experiments on a SPARCstation running Mach 2.5. The path costs obtained were unreasonably high, since the Mach 2.5 threads implementation was not optimized. To obtain more realistic estimates for these costs, we conducted the same experiments on a DECstation 3100 running Mach 3.0 (the SPARCstation 1+ and the DECstation 3100 have approximately the same SPEC marks [SSS90]). The results were used as upper bounds for the actual costs of the paths on a SPARCstation. The cost of the corresponding path for SLS LWPs plus the cost of a null system call determined a lower bound for these path costs. We estimated the cost of the path as the mean of its upper and lower bounds.

Preemptive thread switches are triggered by hardware exceptions (I/O completion, timers). The kernel preempts the currently executing thread and switches to the new thread. We were constrained to user-level experiments on the DECstation 3100. For this reason, we assumed that the cost of a preemptive thread switch would be approximately equal to that of a non-preemptive thread switch.

A *non-preemptive VAS switch* is caused by a thread making a system call to yield the processor. We instrumented the SunOS kernel so that each of two SunOS processes repeatedly made a system call to yield the processor to the other. We used half the average time taken for that call as an estimate for this path.

We instrumented the kernel so that a SunOS process repeatedly makes a system call to sleep for a fixed period of time. We then inserted two *probes* (a *probe* is a location in user or kernel code where the current time of day is evaluated): 1) in the kernel, just after a suspended SunOS process wakes up and 2) at user level, upon return from the sleep. The cost of the *preemptive VAS switch* was estimated as the average difference between two probes times.

7.1.2. LWP Scheduling Path Costs

LWPs are scheduled entirely at the user level. In measuring path costs, however, we assume that a LWP scheduler conveys the priority of the highest priority LWP in its VAS to the kernel using a system call, which is necessary for correct global prioritization among LWPs. Threads and split-level scheduling also provide such global prioritization.

There are two components in an LWP's *deadline change* path cost: the cost of changing the LWP's deadline and the cost of the system call to inform the kernel of the change. We estimated the latter as the average time taken for a SunOS `setpriority()` system call, when the caller continues to have the highest priority.

A *non-preemptive LWP switch* incurs a LWP switch in addition to a deadline change. We computed the average time for the LWP switch by having two LWPs repeatedly changing their deadlines so that each one alternately had the earliest deadline.

A *preemptive LWP switch* is caused by an asynchronous user event notification (a UNIX signal). The cost of this path is the sum of the non-preemptive switch cost, the time taken for signal delivery and the time to request signal notification. To measure the second component, we inserted probes at two points: 1) in the kernel, just before a signal is delivered to the user VAS and 2) at user level, upon signal return. The average difference between the two probe times gives the cost of signal delivery. The last component was measured as the average cost of the `setitimer()` system call in SunOS.

The cost of a *non-preemptive VAS switch* is the sum of the LWP deadline change cost and the time taken by a system call to yield the processor. We estimated the latter using the cost of a non-preemptive VAS switch for threads (Section 7.1.1).

A *preemptive VAS switch* involves a VAS switch, a signal delivery to preempt the current LWP, and a system call to request the signal. To measure the cost of the first two components, we inserted two probes 1) in the kernel, just before a preemptive VAS switch and 2) at user level, upon return from the signal. The average difference between the two probe times gives the cost of these components.

7.1.3. SLS Scheduling Path Costs

We measured SLS scheduling path costs using our prototype implementation of SLS and MMS in SunOS for the SPARCstation 1+ (Chapter 4, Chapter 5).

The cost of a *deadline change* in SLS is the average time taken by the `time_advance()` call, when the caller continues to have the earliest deadline.

We measured the cost of a *non-preemptive LWP switch* by changing the deadlines of two LWPs in a VAS so that each yielded the processor to the other. The cost of the path is half the average time taken for the `time_advance()` call.

The cost of a *preemptive LWP switch* is the sum of the times taken to deliver a user-interrupt, to preempt the currently executing LWP, and to switch to the new LWP. We measured the cost of this path by inserting two probes: 1) in the kernel just before a user-interrupt is delivered and 2) in the user VAS after the new LWP begins executing. The average difference between the probe times gives the cost of the path.

We measured the cost of a *non-preemptive VAS switch* by changing the deadlines of two LWPs in different VASs so that each yielded the processor to the other. The cost of the path is half the average time taken for a `time_advance()` by one process.

The cost of a *preemptive VAS switch* is the sum of the times taken to preempt the currently executing VAS, deliver a user-interrupt, preempt the currently executing LWP, and switch to the new LWP. We measured the cost of this path by inserting two probes: 1) in the kernel just before the current VAS is preempted and, 2) in user-space after the new LWP begins executing. The average difference between the probe times gives the cost of the path.

7.1.4. Discussion

Table 7.1 shows the costs for different paths and scheduling alternatives. The *deadline change* and *non-preemptive process switch* costs for SLS are less than half those for the other alternatives. However, SLS VAS switch costs are greater than thread VAS switch costs. A reason for this difference is that, in SLS, scheduling decisions may be made both at user and kernel level on a VAS switch.

CM applications that perform frequent deadline changes and non-preemptive process switches are likely to significantly reduce scheduling costs by using SLS. We study the actual performance improvement under realistic workloads in Section 7.3.

7.2. I/O Paths and Their Costs

In our model for process I/O, each process is allocated a fixed size I/O buffer. This I/O buffer may be shared between user and kernel VASs (MMSs) or may be implemented entirely in the kernel (read-write system calls, asynchronous I/O). Device interrupt handlers directly read from or write to the process' I/O buffer.

An I/O operation may or may not require synchronization with the device handler (*e.g.*, a user process might block if there is no data in the buffer). Similarly, an I/O operation may or may not require I/O initiation. There are four different I/O paths taken by message transfers:

(1) *Data transfer.* This I/O path (read or write) takes place without the caller blocking or initiating I/O from the device.

(2) *Data transfer with synchronization.* The caller blocks during this path. This may result in a process context switch or a VAS context switch. A KU transfer may block if the I/O buffer is empty and a UK transfer may block if the buffer is full.

Scheduling Path Costs (in μsecs)			
Path Description	Threads	Lightweight Processes	Split-level Scheduling
Deadline change	71	63	28
Non-preemptive process switch	111	73	38
Preemptive process switch	111	225	132
Non-preemptive VAS switch	180	204	297
Preemptive VAS switch	188	321	286

Table 7.1. Scheduling path costs.

Table of scheduling path costs for three different alternatives: threads, LWPs and split-level scheduled LWPs.

(3) *Data transfer with I/O initiation*: The caller initiates the next message transfer from the device. We assume that I/O initiation is only necessary when the chain of interrupts scheduled by the device interrupt handler (Chapter 5) has stopped.

(4) *Data transfer with synchronization and initiation*: On this path, the caller initiates I/O and then blocks.

Below we list the cost of each path for a 1-byte I/O message transfer for three alternatives: read-write system calls, asynchronous I/O and MMS.

7.2.1. Read/Write System Call I/O Path Costs

We estimated the cost of a *data-transfer* path for this alternative by the average time taken to read one byte from a non-empty UNIX pipe. To this cost, we add the time taken to initiate I/O on the audio device (see Section 7.2.3) to get the cost for *data transfer with I/O initiation*.

The cost of a *data transfer with synchronization* path includes the costs of the data transfer path and the cost of a non-preemptive thread or VAS switch. We estimate this cost by adding the two components and subtracting the cost of a null system call from the sum. To these costs, we add the time taken to initiate I/O on the audio device to estimate the costs for *data transfer with synchronization and initiation*.

7.2.2. Asynchronous I/O Path Costs

Both the *data transfer* and *data transfer with I/O initiation* paths for asynchronous I/O are similar to the read-write system call case.

However, the *data transfer with synchronization* path is a bit more complicated. The user process first does a non-blocking `read()` which returns immediately. It is notified of I/O completion using a signal (`SIGIO`) after which the user process calls `select()` to find out which I/O descriptor to read data from. Finally, it does a `read()` to perform the data transfer. The cost of this path is the sum of the average times for each of these individual operations. To this sum, we add the cost of a non-preemptive LWP switch or VAS switch. The *data transfer with synchronization and initiation* costs are computed as before.

7.2.3. MMS Path Costs

To measure the costs of the I/O paths for MMSs, we used our prototye MMS implementation. We conducted simple user level experiments for measuring the costs of each path.

The cost of a *data transfer* path is the time taken to do a one byte `MMS_canrcv()`, followed by an `MMS_rcvdone()`, when the device's chain of interrupts is active and data is available. We added the average time taken by the `notify()` call (for the SPARCstation audio device) to this cost, to get the cost of *data transfer with I/O initiation*.

The cost of *data transfer with synchronization* is the sum of the data transfer cost and the cost of a non-preemptive LWP or VAS switch path. To these we add the cost of I/O initiation, to get the cost of *data transfer with synchronization and initiation*.

7.2.4. Discussion

Table 7.2 shows the costs for different I/O paths and I/O alternatives. The *data transfer*, *data transfer with I/O initiation* and *data transfer with synchronization* (process switch case) costs for MMS are two to five times less than the other alternatives. The VAS switch *data transfer with synchronization* costs are comparable for threads and SLS, for reasons described in Section 7.1.4.

CM applications that incur fewer VAS switch *data transfer with synchronization* paths are likely to significantly reduce scheduling costs by using MMS. We study the actual performance improvement under realistic workloads in Section 7.3.

7.3. Performance Under Synthetic Workloads

In this section, we present the results of a series of experiments designed to compare the performance of different techniques for scheduling and I/O. Using synthetic CM application workloads, we compared the scheduling and I/O costs of: 1) threads using read-write system calls for I/O, 2) LWPs using UNIX asynchronous I/O and 3) split-level scheduled LWPs using memory-mapped streams for I/O.

	I/O Path Costs (in μsecs)		
Path Description	Read-Write System Calls	Asynchronous I/O	Memory-mapped streams
Data Transfer	114	114	21
Data Transfer with I/O Initiation	118	118	59
Data Transfer with Synchronization (Process switch)	191	543	59
Data Transfer with Synchronization (VAS switch)	260	735	318
Data Transfer with Synchronization and Initiation (Process switch)	195	547	97
Data Transfer with Synchronization and Initiation (VAS switch)	264	739	322

Table 7.2. I/O path costs.

Table of I/O path costs (for 1-byte I/O operations) for three different alternatives: system calls, asynchronous I/O and memory-mapped streams.

7.3.1. Methodology

To design the experiments, we identified different factors that characterize CM applications on a single computer system:

(1) The number of processes in a single VAS.

(2) The message rates of the CM streams handled by processes.

(3) Process delay bounds.

(4) Process workahead limits.

(5) The number of VASs running CM applications.

(6) Policy for workahead processes.

Factors 1, 3, 5 and 6 are mutually orthogonal. Factors 2, 3 and 4 are related in the following way: If end-to-end delay of a CM stream is low, then process delay bounds may be low, stream message rates may be high, and process workahead limits may be small. We do not consider other factors such as the presence of non-realtime jobs in the system.

We conducted six experiments. In each experiment, one factor was varied over a range of values, while the others were fixed. The workload was chosen so that, at the midpoint of this range, CM message processing utilized about 40% of the CPU.

Although we had implemented SLS and MMS, we chose simulation as the performance comparison methodology for two reasons. First, we lacked a comparable implementation of DWS scheduled threads and kernel support for DWS scheduled LWPs. Second, in our implementation of SLS and MMS, process deadlines have a granularity of 10ms (the clock interrupt period on the Sun SPARCstation). For this reason, process deadlines are constrained to be integral multiples of 10 ms, so the effect of lower process delay bounds cannot be studied. Reducing the clock period would have involved drastic modifications to SunOS.

We used the path costs for scheduling and I/O operations (Sections 7.1 and 7.2) to drive a CM application workload simulator. The inputs to the simulator were the workload description, the scheduling technique to be used, and a time limit for the simulation. The workload was described as a list of processes, each with the following parameters: VAS, delay bound, CM stream message rate, processing time per message, whether the stream is read or written, I/O buffer size and process start time. The output of the simulator was the cost of scheduling and I/O as a percentage of total CPU time and the count of scheduling and I/O paths of each type.

We validated the simulator's performance for SLS by comparing its output in steady state with the scheduling and I/O overhead of a workload on our implementation. The error between the simulator and the real workload was less than 1% for the test cases studied. For threads and LWPs, we validated the simulator by examining simulator traces for simple workloads.

7.3.2. Simulation Results

This section discusses the experiments in greater detail and presents simulation results. For each experiment, we describe the workload used and interpret the results for real CM applications.

7.3.2.1. Varying the Number of Processes

The workload for this experiment consisted of a single VAS with a varying number of processes. Each process handled a 50 messages/s CM stream, had a 12-message I/O buffer, a 250 ms delay bound and a 0.9 ms message processing time. This workload is typical of an ACME server with multiple high delay processes. Figure 7.1 shows how the number of processes affect the scheduling and I/O cost for the different approaches.

As the number of LWPs increases, the overhead for the different approaches also increases. The rate of increase is higher, however, for threads and LWPs than it is for SLS. In SLS, most process context switches are done at user level. The greater the number of processes, the greater is the frequency of user-level switches. With 18 processes, the cost for SLS is less than 5%, whereas the threads cost is 4 times as much.

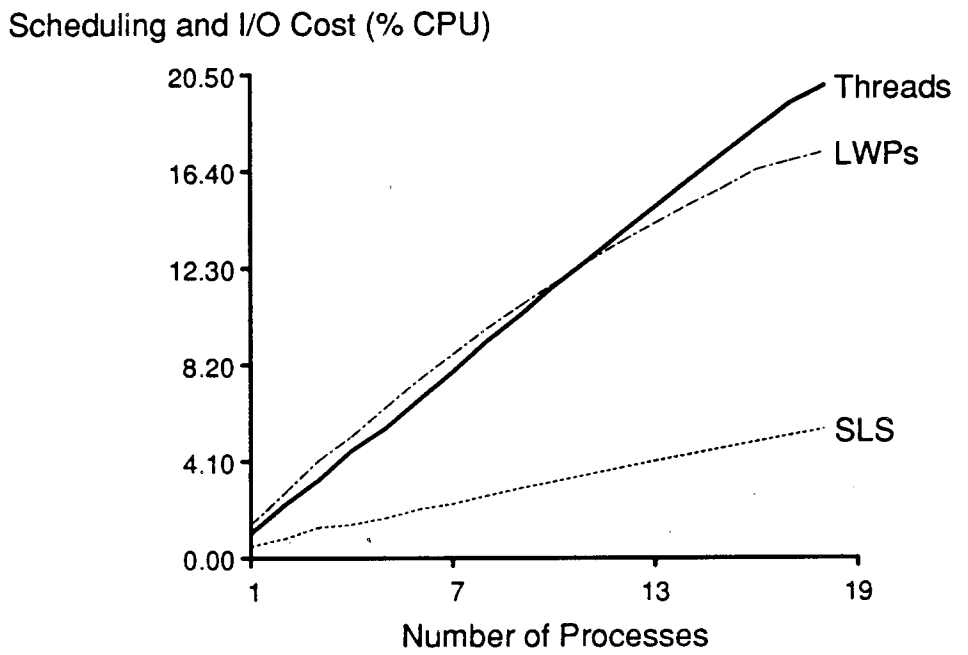


Figure 7.1. Varying the number of processes.

This figure shows the effect of varying the number of processes on scheduling and I/O costs for a single VAS. The rate of increase of scheduling and I/O costs for threads and LWPs is greater than that for SLS LWPs.

Multiprocess workahead CM applications, such as audio mixers mixing file data, may benefit significantly from SLS and MMS.

7.3.2.2. Varying the Message Rate

For this experiment, the workload consisted of a single VAS with five processes. Each process had a 10 message I/O buffer and a 0.9 ms per message processing time. The delay bound for each process varied from 1/2 s for 1 message/s case to 50 ms for the 200 messages/s case. Figure 7.2 shows the scheduling and I/O cost as a percentage of total CPU overhead as a function of CM stream message rate.

As CM stream message rate increases, the overhead for the different approaches also increases. At higher message rates, threads and LWPs have considerable overhead. For instance, at 200 messages/s SLS incurs only 6% scheduling and I/O cost while threads and LWPs cost 4 times as much.

CM applications handling high message rate CM streams may benefit significantly from SLS and MMS.

7.3.3. Varying Process Delay Bounds

For this experiment, our workload consisted of a single VAS with eight processes, each handling 50 message/sec CM streams. Of the eight, some were low-delay and some were high-delay processes. The low-delay processes simulated periodic CM device reads; they had a delay

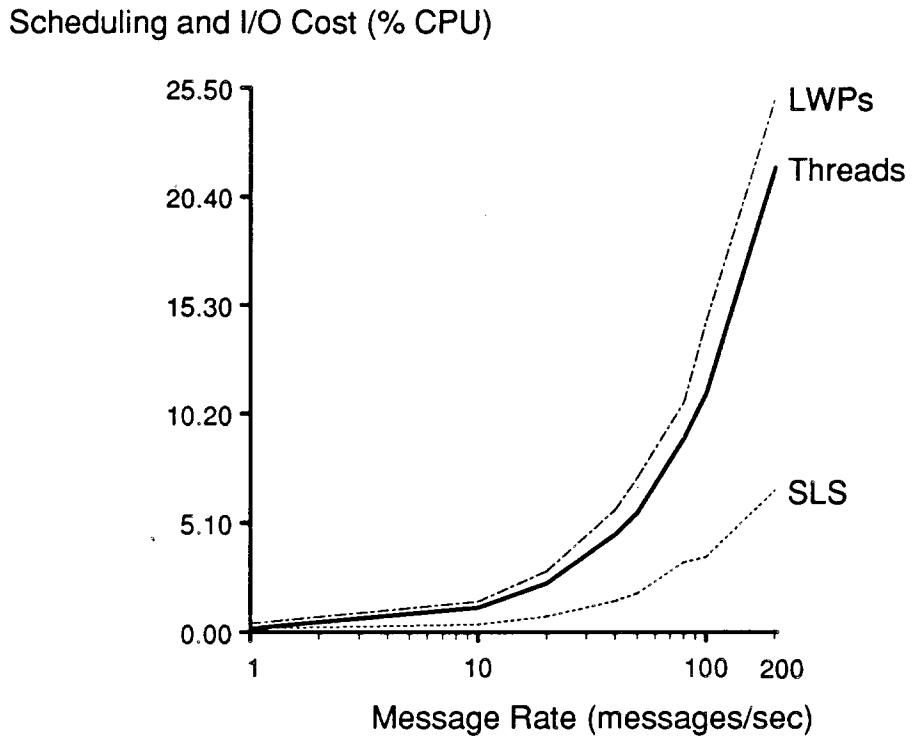


Figure 7.2. Varying message rate.

This figure shows the effect of message rate on scheduling and I/O cost. The cost for threads and LWPs is 4 times that for SLS at high message rates.

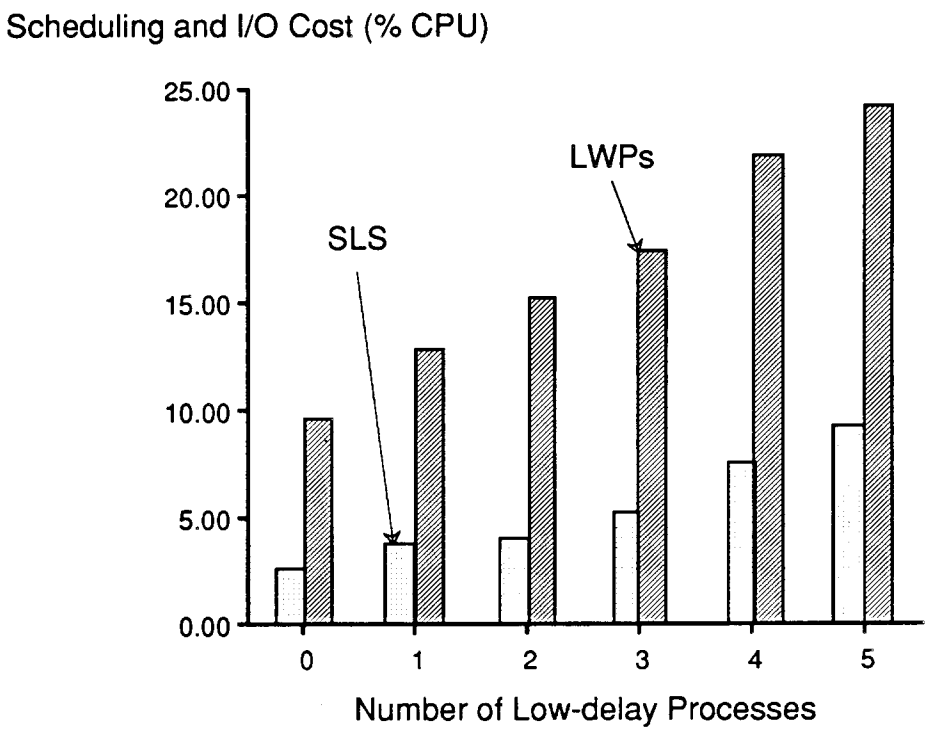


Figure 7.3. Varying process delay bounds.

This figure shows the effect of low-delay processes on scheduling and I/O costs of SLS and LWPs. With low-delay processes, a user-kernel interaction may be incurred on almost every message. Even so, the SLS has three times less overhead compared to LWPs.

bound of 5 ms, a one message I/O buffer and 0.9 ms message processing time. High-delay processes did file or network I/O; they had a delay bound of 250 ms, a 12 message I/O buffer and 0.9 ms per message processing time. This type of workload is typical of ACME servers.

Figure 7.3 shows the scheduling and I/O costs for LWPs and SLS for the above workload. Low-delay processes have small workaheads and when a low-delay process becomes critical, it usually has the earliest deadline. In this case, a user-interrupt is necessary to wake up that process. Thus, low-delay processes may incur more frequent user/kernel interactions, compared to high-delay processes.

This experiment is demonstrates the efficiency of kernel-to-user interaction in SLS. With LWPs, each message in a low-delay stream incurs a system call to set a timer and a signal to wake up the process. In SLS, each message incurs a user-interrupt. The difference in overheads can be pronounced: with 3 low-delay processes, SLS scheduling and I/O cost is 5%, whereas LWPs cost 3 times more. Low-delay applications, like mixers serving live participants, may benefit significantly from SLS and MMS.

7.3.4. Varying I/O Buffer Sizes

The workload for this experiment consisted of a single VAS with eight processes each handling a 50 message/s stream. Each process had a message processing time of 1.5 ms, and a delay bound of 1/2 s. The I/O buffer size (Section 7.2), which represents the workahead limit of

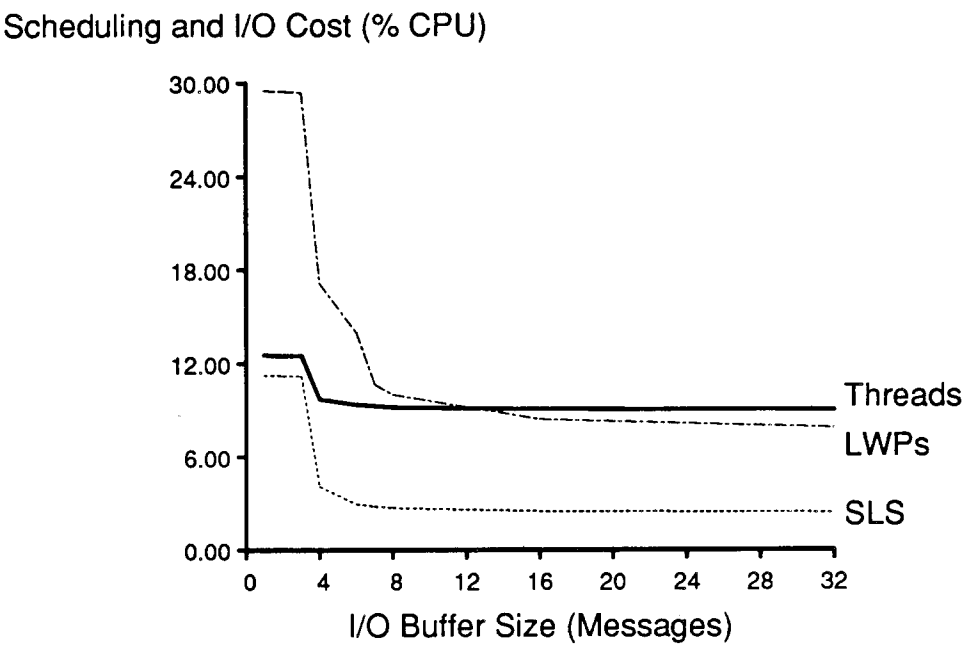


Figure 7.4. Varying I/O buffer sizes.

This figure shows the scheduling and I/O costs for SLS for a varying I/O buffer size. Even with a 4 message I/O buffer, we observe significant reductions in the cost for all approaches.

the process, was varied.

Figure 7.4 shows how I/O buffer size affects scheduling and I/O costs for threads, LWPs and SLS. Even with a buffer size of 4 messages, we observe a significant reduction in cost; for SLS, the cost is nearly 11% for a three message I/O buffer and only 4% for a four message I/O buffer.

Thus, for workahead CM applications, even relatively small I/O buffers can significantly reduce scheduling and I/O cost.

7.3.5. Varying the Number of VASs

The workload for this experiment consisted of twelve processes each handling a 50 messages/s CM stream. The number of VASs for this experiment varied from one to three. The processes were equally divided between the different VASs. For each VAS value, we conducted two experiments: one in which exactly one of each VAS' processes had a low-delay bound (5 ms delay bound, 1 message I/O buffer, 0.9 ms message processing time) and another in which all its processes had high-delay bounds (200 ms delay bound, 10 message I/O buffer, 0.9 ms message processing time). The former case is representative of low-delay CM applications, and the latter of high-delay CM applications.

Figure 7.5 shows the scheduling and I/O costs as a percentage of total CPU time for split-level scheduling. Increasing the number of VASs increases the scheduling and I/O overhead, since a VAS switch is incurred. This increase is less significant for workahead applications, than for low-delay applications. For the latter, kernel-to-user interactions (including expensive preemptive VAS switches) are more frequent (Section 7.3.3).

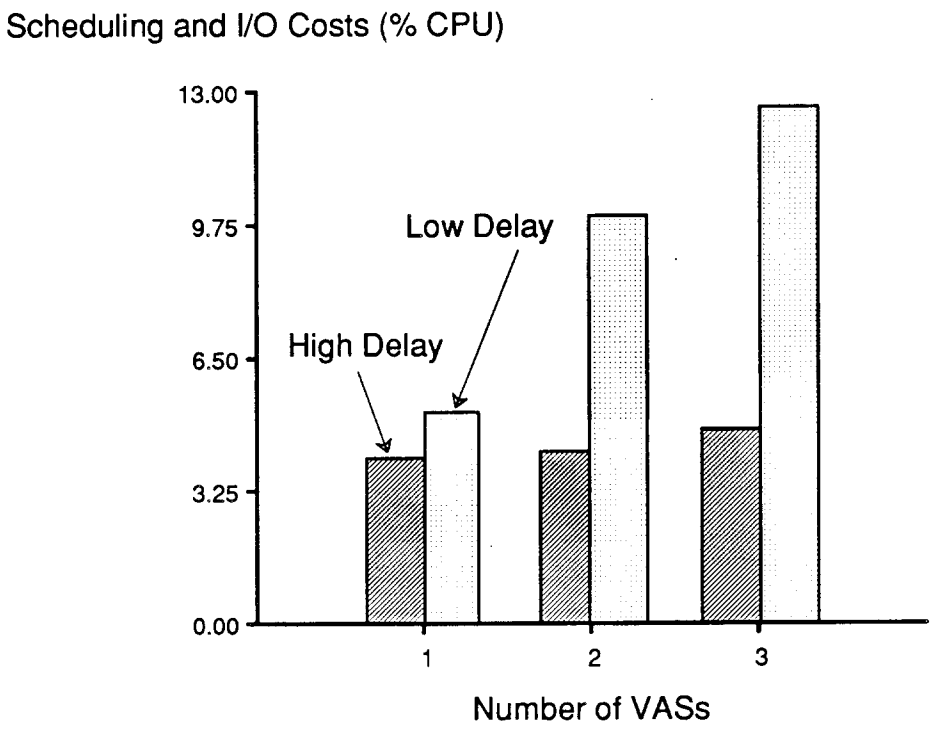


Figure 7.5. Varying the number of VASs.

This figure shows the effect of varying the number of VASs on scheduling and I/O cost for SLS. The cost increases with the number of VASs; the increase is more significant if each VAS has a low-delay process.

This result has implications for distributed CM application structure. To reduce overhead, a single system should have only one or two low-delay applications running; however, multiple high-delay applications may run efficiently on the same workstation.

7.3.6. Varying workahead policy

The DWS policy does not specify how workahead processes are scheduled. This final experiment compares two different policies for split-level scheduling workahead processes. They are:

- *Globally Earliest-deadline*: Workahead processes are scheduled on a globally earliest-deadline basis. This policy attempts to reduce the probability of synchronous user-kernel interactions for processes that can work ahead.
- *Earliest-deadline in VAS*: As long as there are runnable LWPs in a VAS, this policy schedules them earliest-deadline first. A VAS switch is incurred only if a process in some other VAS becomes critical or if there are no more runnable workahead processes in the same VAS.

The workload for this experiment consists of a total of eight workahead LWPs each handling a 50 messages/s CM stream. Each process has a 10 message I/O buffer limit, a 200 ms delay bound and a 0.9 ms message processing time.

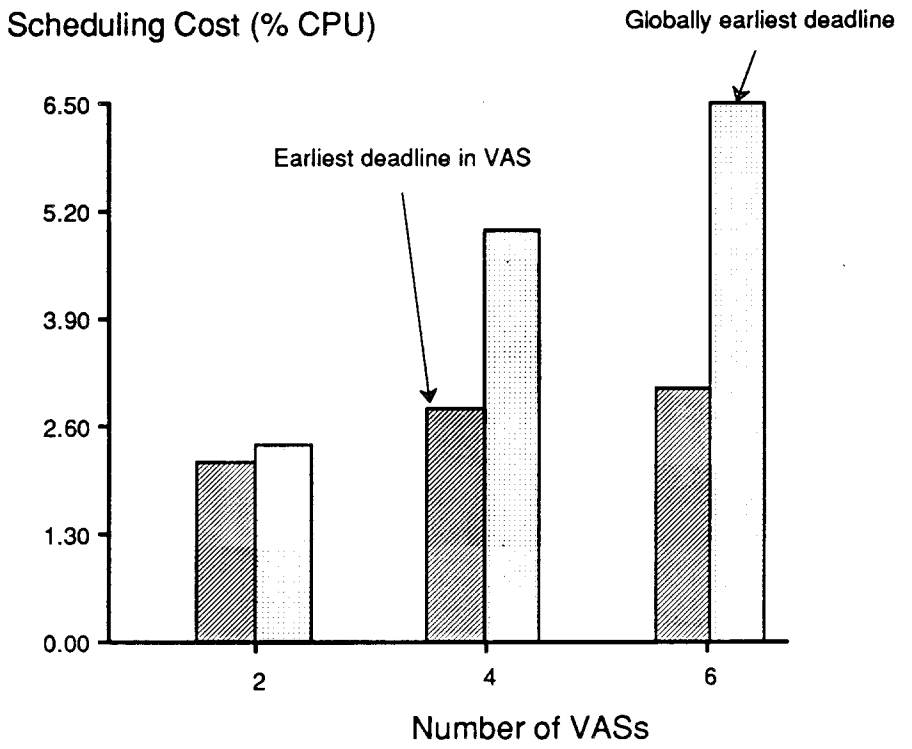


Figure 7.6. Varying workahead policy.

This figure shows the scheduling costs for SLS for two different workahead policies. The earliest-deadline within VAS policy, in which the scheduler avoids a VAS switch as much as possible, consistently outperforms the globally earliest-deadline policy.

Figure 7.6 plots the scheduling cost for the two policies against the number of VASs (if there is only 1 VAS, the two policies have identical cost). This shows that the globally earliest-deadline policy is consistently more expensive than the earliest-deadline within VAS policy.

Thus, the choice of workload policy may have an effect on the performance of SLS. The results indicate that a "greedy" approach to workahead (trying to do as much work in the same VAS as possible) can halve the scheduling cost.

Chapter 8

RELATED WORK

This chapter reviews related work in efficient process scheduling and stream communication mechanisms and describes other work on OS support for integrated CM. It then places this work in the context of recent trends in OS research.

8.1. Efficient Process Scheduling

Two-level process scheduling is not a new idea [SaB75]. However, its performance implications have only been recently investigated. Such investigation has focused on efficient multiprocessor operating system support for user-level management of parallelism. SLS is related to three approaches in this area; Psyche's [SLM90] first-class user LWPs, the XERO OS's threads package, and scheduler activations.

In Psyche, ULSs schedule LWPs on kernel-supported threads. The kernel delivers user-interrupts to notify the ULS of events that affect LWPs in the user VAS, such as blocking cross-domain invocations [MSL91]. Unlike SLS, kernel threads are time-sliced on physical processors. To avoid undesirable interactions between thread scheduling and LWP scheduling, the kernel warns the ULS of impending thread preemption; a ULS can use this warning to avoid acquiring spin locks, for instance. User/kernel shared memory is used to efficiently request timer user-interrupts, to specify a stack on which user-interrupts are to be delivered, and for user-interrupt masking.

User-level LWPs in XERO [IKN91] are managed in a similar manner. User/kernel shared memory is used by the ULS to regulate kernel action when a thread blocks or is preempted. The kernel delivers user-interrupts when a thread returns from preemption. When a thread blocks, the degree of parallelism in the user program may be reduced; in such a case, the kernel creates another thread in user space. If an LWP is spin-waiting on a lock held by a suspended thread, it detects this condition from shared memory (which contains the execution state of threads) and then suspends itself.

In scheduler activations [ABL91], the kernel vectors to the ULS every event (*e.g.*, thread preemption, thread blocking) that affects LWP scheduling. A scheduler activation is an execution context for an event vectored from kernel to user VAS. The ULS uses this execution context to modify LWP data structures, to execute LWPs and so on. The ULS also informs the kernel of changes in the degree of parallelism in its address space. Scheduler activations do not use user/kernel shared memory for communicating scheduling information.

In each of these approaches, the kernel and the ULS scheduling policies are independent. As a result, the approaches cannot correctly prioritize LWPs across threads that may be running in different address spaces but contending for the same processor. They also cannot exploit policy-specific information (*e.g.*, LWP priorities) to reduce user-kernel interactions.

Symunix II [ELS88] also uses user/kernel shared memory for masking interrupts and preemption. In Symunix II, parallel applications are implemented as a collection of UNIX processes communicating through shared memory. Processes use virtual preemption masking while holding short-duration busy-waiting locks and virtual signal masking while updating shared memory. Unlike virtual user-interrupt masking, virtual signal masking requires a new system call to handle pending signals after unmasking interrupts.

FORMULA [AnK91] uses virtual signal masking to implement short critical sections. Heuser [Heu90] describes a mechanism similar to virtual preemption masking for avoiding thread

preemption during short critical sections.

The Synthesis system [PM88] adopts a different approach to reduce the overhead of user/kernel interactions for thread scheduling. This approach uses kernel code synthesis to generate specialized kernel routines for thread context switches. Context switch costs are reduced by saving only part of the context (*e.g.* floating point state is not saved if the process does not perform floating-point computations) and by efficiently traversing process queues (a process performs direct handoff to the next process in the run queue).

8.2. Efficient Stream Communication

Shared memory is used in DEMOS [BHM77] and in VM/370 [Att79] to avoid copying overhead during data transfer. However, this approach does not reduce control overhead; synchronization is necessary for each transfer.

MMSs differ from memory-mapped files [RDH80] in a number of ways. An MMS provides sequential access to possibly non-persistent data. Memory-mapped files provide random access to persistent objects. A CM stream may be larger than a VAS, and an MMS need not contain the entire CM stream; since CM streams are accessed sequentially, a small circular buffer suffices. Since data may be non-persistent, MMS avoid page faults for accessing data, instead relying on explicit producer/consumer synchronization. Also, since access is sequential, data is "released" explicitly and page replacement algorithms are not needed.

Wolf [Wol91] uses a double-buffering technique for KU and UK stream transfers. After each buffer is processed, a user/kernel interaction is necessary. This approach may be inefficient for low end-to-end delay CM streams.

Bershad *et al.* [BAL91] describe a user-level communication mechanism for RPCs. When a client LWP makes an RPC, the ULS switches to another LWP in the same task. The RPC is added to a message queue shared pairwise between the application VAS and the server VAS. On a shared memory multiprocessor, a server LWP may be running on another processor; it accesses this message queue directly, services the request and returns a reply to another shared queue. Client/server synchronization is only necessary when the message queue becomes empty or full. If the message queue is full, for instance, the server does not have enough processing power to handle the requests; the client then *donates* a processor to the server. The shared queue is similar to a UU-type MMS (Chapter 5); in the latter, processor donation is determined completely by the deadlines associated with messages in the MMS.

The stream I/O model in Synthesis [MaP89] provides an alternate approach to efficient stream communication. Kernel code synthesis reduces the cost of system calls to transfer data. Producer/consumer synchronization overhead is reduced by using optimistic synchronization techniques.

8.3. OS Support for CM

The CPU scheduling approach in Synthesis [MaP90] represents an alternative to deadline/workahead scheduling. The Synthesis model is based on rate-control feedback. Processes make no calls to indicate their temporal progress; instead, the kernel adjusts time-slice quanta based on queue lengths. This approach is well-suited to some situations (*e.g.*, audio DSP with little slack CPU time). In general, end-to-end rate-control feedback (Section 2.3.1.3) may be more appropriate for integrated CM.

Nakajima *et al.* [NYM91] describe extensions to Mach for integrated CM. Asynchronous event notifications are preemptively scheduled earliest-deadline first. This mechanism supports development of user-level device drivers for CM devices. The performance implications of these mechanisms for CM applications is not discussed.

Wolf [Wol91] describes a runtime environment for CM communication on the AIX operating system. Threads running in kernel space perform protocol processing of CM data received over the network. The CM stream is subsequently transferred to user space using the mechanism described in Section 8.2. User-level processing of CM data takes place in signal context. This avoids synchronization between user and kernel processes. Relative prioritization between

different signal handlers is not discussed.

8.4. Related Trends in Operating Systems

Our work is related to several directions (such as migration of OS kernel functionality to the user-level, asynchronous communication, and efficient local data transfer) of current OS research.

Modern operating systems such as Amoeba [MRT90], Chorus [RAA], and Mach shift functionality from kernel to user level to improve software structure. In contrast, this work shifts functionality to user level to increase performance.

Most existing operating systems use request/reply communication; examples include UNIX-type system calls, RPC, and object invocation. This paradigm is not well-suited to continuous media (more generally, it may not be well-suited to future distributed systems in which speed-of-light delays dominate throughput limits). MMSs provide efficient local asynchronous communication. Example of related work include the asynchronous RPC proposed by Gifford [GiG88] and the dataflow model of Synthesis [PM88].

In UNIX-type systems, I/O and IPC performance is limited by the overhead of data copying. Systems such as Mach, DASH and Topaz have attacked this problem using techniques such as VM remapping and shared memory [RTY88, ScB90, TzA91]. The MMS mechanism is complementary to this work; it attacks the overhead of control rather than data movement.

Chapter 9

CONCLUDING REMARKS

This chapter concludes the dissertation. Section 9.1 lists its major contributions and Section 9.2 summarizes the major results. Finally, Section 9.3 discusses directions for future work in this area.

9.1. Thesis Contributions

The following are the contributions of this dissertation to research on operating systems support for integrated continuous media applications.

- The recognition that policies and mechanisms in general-purpose OSs may be non-optimal for integrated CM.
- The design and implementation of *split-level scheduling* (SLS), a new scheduler implementation technique that combines the advantages of kernel-implemented threads and user-level LWPs.
- The design and implementation of *memory-mapped streams* (MMS), a new class of mechanisms for asynchronous stream transfer between user and user or user and kernel VASs.
- The evaluation of SLS and MMS performance. Compared with conventional approaches, these mechanisms can reduce scheduling and message transfer overhead for some CM tasks by up to a factor of four.

The next section elaborates on these contributions, highlighting the major results of this dissertation.

9.2. Summary of Results

General-purpose operating systems incorporate design principles (fairness-oriented scheduling, request-reply communication, reliable data delivery) that are contrary to the requirements of CM processing and stream communication. This motivates the reexamination of policies and mechanisms in these OSs.

CPU scheduling policies in general-purpose operating systems may use fast response, high throughput and fairness as the criteria for scheduling processes. These criteria may be contrary to the delay requirements of CM tasks. Deadline/workahead scheduling is appropriate for CM because it incorporates timeliness (in the form of process deadlines) as a criterion.

Some CM tasks perform frequent scheduling and message transfers. With conventional approaches, these tasks may incur significant scheduling and message transfer overhead (between 15% and 25% of total CPU). This overhead is partly due to user/kernel interactions (*e.g.*, system calls and asynchronous events).

SLS divides the functionality of scheduling LWPs between a per VAS user-level scheduler and a single kernel-level scheduler. In such a two-level scheduler, the two parts exchange scheduling information. When a piece of information need not be exchanged synchronously, it is passed through shared memory, thereby avoiding user/kernel interactions.

In MMSs, a region of memory shared between producer and consumer indicates the location of the data to be transferred. Shared memory is also used for producer/consumer synchronization (producer waiting for consumer to read data or consumer waiting for producer to generate

data). These techniques reduce or eliminate user/kernel interactions for message transfers. When combined with SLS, MMSs can further reduce user/kernel interactions for message transfers.

Split-level scheduling is most effective when switches between LWPs within a VAS are more frequent than switches between VASs. Memory-mapped streams are most effective when a VAS contains many LWPs, some of which may work ahead. These properties are satisfied by CM workloads with only one or two tasks containing low-delay processes (*e.g.*, an ACME server task or a mixer task mixing data from a live participant). To best exploit these mechanisms, only one or two tasks with low-delay processes should run on a workstation. CM playback and record applications have only high-delay processes; hence compute servers and file servers may run multiple applications of this type and still benefit from these mechanisms.

These mechanisms are applicable for purposes other than CM. Process control applications (*e.g.*, [AIL86]) have scheduling requirements similar to those of CM. Split-level scheduling could be used with a time-slicing policy for a situation where a VAS contains both interactive and background processes. Memory-mapped streams could be used for access to a sequential disk file or a network stream connection. More generally, the mechanisms may be useful in any situation where the rates of I/O and scheduling operations, and the costs of user/kernel interactions, are high.

A simple example can be devised to show that, on the current generation of uniprocessor architectures, SLS and MMS may actually enable applications like distributed music rehearsal. For instance, suppose a music rehearsal application requires an end-to-end delay of 15 ms. It is not inconceivable that a process handling a stream of this application may need an end-to-end delay bound of (say) 3 ms. By appropriately choosing the kind of computation on the stream, we can show that, while kernel threads and user-level LWPs cannot satisfy this bound, split-level scheduling (and memory-mapped streams) can.

This argument may not hold when processor speeds increase. However, our hypothesis (Chapter 3) would still be valid. In fact, current trends in processor architecture point to a four-fold or greater reduction in scheduling and message transfer costs with SLS and MMS. Both these mechanisms strive to reduce or eliminate user/kernel interactions, and current trends are increasing the relative cost of these interactions. Architecture designers want to improve application performance; with memory access times not improving proportionally with processor speeds, they achieve their goal by increasing register file sizes and cache sizes and thereby exploiting application program locality. However, these methods increase the work to be done during a user/kernel interaction.

9.3. Future Work

All OS processing of CM data must also be scheduled to meet the real-time requirements of integrated CM applications. One obstacle to this goal is unbounded priority inversion: processing of an earlier deadline CM message can be delayed by the processing of a later deadline CM message or by other activity. This can happen in a number of ways. In UNIX, the kernel is non-preemptible, and a long-running system call (*e.g.*, `fork()`) can delay the handling of CM messages. Again, a process in $\bar{A}$ may lock a resource which a process in A may need. Finally, a software interrupt may be handling a CM message when a message with a higher priority arrives. Solutions to these problems exist ([Fur91], [SRL90], [ADH91]). Further work is needed to implement these solutions on general-purpose OSs.

Future research could also examine shared memory multiprocessor operating system support for integrated CM. Such research would first design a multiprocessor CPU scheduling policy which incorporates timeliness and allows workahead. SLS and MMS are directly applicable to shared memory multiprocessor systems. Future work would need to design appropriate concurrency control mechanisms for multiprocessor implementations of SLS and MMS; the assumptions made in Chapter 6 do not hold for shared memory multiprocessors.

With SLS LWPs, inadvertent or malicious faults in a CM task can cause LWPs in other tasks to miss their deadlines. Inadvertent runaway faults (*e.g.*, a LWP setting the preemption mask) may be detected by setting "watchdog" timers. However, detecting malicious or inadvertent

transient (*i.e.*, not runaway) faults is hard; due to lack of adequate knowledge of user-level processing, the kernel cannot verify the correctness of deadlines advertised by the ULS. Future work is needed for efficient mechanisms for fault detection and policies for fault containment.

Future work can investigate the tradeoffs in the design of the data part of MMSs (Section 5.3.1). Given copying and virtual memory remapping costs for different message sizes, such work would determine the optimal structure of the data part for different situations (network input, device input or output etc.). This study would also influence protocol and device driver implementations.

Future work could devise techniques to allow applications to accurately estimate their worst case processing requirements. With the CM-resource model's approach to reserving CPU capacity, CPU utilization can be higher if these estimates are as tight as possible.

BIBLIOGRAPHY

[Abb84] C. Abbott, "Efficient Editing of Digital Sound on Disk", *J. Audio Eng. Soc. 32*, 6 (June 1984), 394.

[ABB86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 81-92.

[AHH88] A. Agarwal, J. Hennessy and M. Horowitz, "Cache Performance of Operating System and Multiprogramming Workloads", *Trans. Computer Systems 6,4* (Nov. 1988), 393-431.

[AIL86] L. S. Alger and J. H. Lala, "A Real Time Operating System For a Nuclear Power Plant Computer", *Proceedings of the 1986 IEEE Real-time Systems Symposium*, 1986, 244-248.

[ALB89] T. E. Anderson, H. M. Levy, B. N. Bershad and E. D. Lazowska, "The Interaction of Architecture and Operating System Design", *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* , Boston, Massachusetts, Apr. 3-6, 1989, 108-120.

[AHS90] D. P. Anderson, R. G. Herrtwich and C. Schaefer, "SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet", Technical Report 90-006, International Computer Science Institute, Feb. 1990.

[ADH91] D. P. Anderson, L. Delgrossi and R. G. Herrtwich, "Process Structure and Scheduling in Real-Time Protocol Implementations", *Proc. Kommunikation in Verteilten Systemen*, Feb. 1991.

[AnK91] D. P. Anderson and R. J. Kuivila, "FORMULA: a Programming Language for Expressive Computer Music", *IEEE Computer*, June 1991.

[ABL91] T. E. Anderson, B. N. Bershad, E. D. Lazowska and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism", *Proc. of the 13th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Oct. 14-16, 1991, 95-109.

[AGH91] D. P. Anderson, R. Govindan and G. Homsy, "Abstractions for Continuous Media in a Network Window System", *International Conference on Multimedia Information Systems*, Singapore, Jan. 1991.

[AOGar] D. P. Anderson, Y. Osawa and R. Govindan, "Real-Time Disk Storage and Retrieval of Digital Audio and Video", *Trans. Computer Systems*, to appear.

[And] D. P. Anderson, "Meta-Scheduling for Continuous Media", *ACM Trans. Computer Systems (to appear)*, .

[AHL91] S. Angebranndt, R. L. Hyde, D. H. Luong, N. Siravara and C. Schmandt, "Integrating Audio and Telephony in a Distributed Workstation Environment", *Proceedings of the 1991 Summer USENIX Conference*, Dallas, TX, Jan. 21-25 1991, 419-434.

[ABL89] B. Arons, C. Binding, K. Lantz and C. Schmandt, "The VOX Audio Server", *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, Ottawa, Ontario, April 20-23, 1989.

[Att79] C. R. Attanasio, "Virtual Control Storage - Security Measures in VM/370", *IBM Systems Journal 18,1* (1979), 93-110.

[BaM91] A. Banerjea and B. A. Mah, "The Real-time Channel Administration Protocol", *Proc. of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991.

[BHM77] F. Baskett, J. H. Howard and J. T. Montague, "Task Communication in DEMOS", *Proc. of the 6th ACM Symp. on Operating System Prin.*, West Lafayette, Indiana, Nov. 16-18, 1977, 23-31.

[BAL91] B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, "User-level Interprocess Communication for Shared Memory Multiprocessors", *Trans. Computer Systems 9, 2* (May 1991), 175-198.

[ClE85] D. W. Clark and J. S. Emer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement", *Trans. Computer Systems 3*, 1 (Feb. 1985), 31-62.

[Coh78] D. Cohen, "A Protocol for Packet-Switching Voice Communication", *Computer Networks 2* (1978), 320-331.

[CCC90] *SPARC RISC User's Guide*, Cypress Semiconductor, San Jose, CA, 1990.

[ELS88] J. Edler, J. Lipkis and E. Schonberg, "Process Management For Highly Parallel UNIX Systems", *Proc. of USENIX Workshop on Unix and Supercomputers*, September 1988, 1-17.

[Fox91] E. A. Fox, "Advances in Interactive Digital Multimedia Systems", *IEEE Computer*, Oct. 1991, 9-21.

[Fur91] B. Furht, *Real-time UNIX Systems: A Design and Application Guide*, Kluwer Academic Publishers, Boston, MA, 1991.

[GeC92] J. Gemmell and S. Christodoulakis, "Principles of Delay-Sensitive Multimedia Data Storage and Retrieval", *ACM TOIS 10*, 1 (Jan. 1992), 51-90.

[Get86] J. Gettys, "Problems Implementing Window Systems in UNIX", *Proceedings of the 1986 Winter USENIX Conference*, Denver, Colorado, January 15-17, 1986, 89-97.

[GiG88] D. K. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Computation", *Trans. Computer Systems 6*, 3 (Aug. 1988), 258-283.

[Gre92] J. L. Green, "The Evolution of DVI System Software", *Comm. of the ACM 35, 1* (Jan. 1992), 53-67.

[HBJ91] J. Hanko, D. Berry, T. Jacobs and D. Steinberg, "Integrated Multimedia at Sun Microsystems", *Proc. of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991.

[Heu90] M. Heuser, "An Implementation of Real-time Thread Synchronization", *Proceedings of the 1990 Summer USENIX Conference*, Anaheim, June 11-15, 1990, 97-105.

[Hop90] A. Hopper, "Pandora – An Experimental System for Multimedia Applications", *Operating System Review 24*, 2 (Apr. 1990), 19-34, ACM SIGOPS.

[IKN91] S. Inohara, K. Kato, A. Narita and T. Matsuda, "A Thread Facility based on User/kernel Cooperation in the XERO Operating System", *Proceedings of the 1991 Compsac*, Tokyo, Japan, 1991.

[JeS90] K. Jeffay and F. D. Smith, "Designing a Workstation-Based Conferencing System Using the Real-time Producer/Consumer Paradigm", *Proc. of the First International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Berkeley, California, Nov. 1990.

[JSS91] K. Jeffay, D. L. Stone and F. D. Smith, "Kernel Support for Live Digital Audio and Video", *Proc. of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991.

[Jef -- K. Jeffay, "The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-time Computations", Ph.D. Thesis, Department of Computer Science, University of Washington. Technical Report #89-09-15.

[Kan87] G. Kane, *MIPS R2000 RISC Architecture*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Kar89] P. Karger, "Using Registers To Optimize Cross-Domain Call Performance", *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating*

Systems , Boston, Massachusetts, Apr. 3-6, 1989, 194-204.

[Lan84] K. Lantz, "Structured Graphics for Distributed Systems", *ACM Trans. on Graphics 3*, 1 (Jan. 1984).

[LMK89] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, 1989.

[LeE89] H. M. Levy and R. H. Eckhouse, *Computer Programming and Architecture: The VAX*, Digital Press, Bedford, MA, 1989.

[LiL73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *J. ACM 20*, 1 (1973), 47-61.

[Loy85] G. Loy, "Designing an Operating Environment for a Realtime Performance Processing System", *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, 1985, 9-13.

[LuD87] L. F. Ludwig and D. F. Dunn, "Laboratory for Emulation and Study of Integrated and Coordinated Media Communication", *Proc. of ACM SIGCOMM 87*, Stowe, Vermont, Aug. 1987, 283-291.

[MSL91] B. D. Marsh, M. L. Scott, T. J. LeBlanc and E. P. Markatos, "First-Class User Level Threads", *Proc. of the 13th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Oct. 14-16, 1991, 110-121.

[MaP89] H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel", *Proc. of the 12th ACM Symp. on Operating System Prin.*, Litchfield Park, Arizona, Dec. 3-6, 1989, 191-201.

[MaP90] H. Massalin and C. Pu, "Fine-Grain Adaptive Scheduling using Feedback", *Computing Systems 3*, 1 (Winter 1990).

[McA88] J. McCormack and P. Asente, "Using the X Toolkit or How to Write a Widget", *Proceedings of the 1988 Summer USENIX Conference*, San Franscisco, June 20-24, 1988, 1-14.

[MoB89] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance", *Proc. Third International Conf. on Architectural Support for Programming Languages and Operating Systems* , Boston, Massachusetts, Apr. 3-6, 1989, 75-84.

[MRT90] S. J. Mullender, G. Rossum, A. S. Tanenbaum, R. Renesse and H. Staveren, "Amoeba: A Distributed Operating System for the 1990s", *Computer 23*, 5 (May 1990), 44-53, IEEE.

[NYM91] J. Nakajima, M. Yazaki and H. Matsumoto, "Multimedia/Realtime Extensions for the Mach Operating System", *Proceedings of the 1991 Summer USENIX Conference*, Dallas, TX, Jan. 21-25 1991, 183-197.

[NoK91] J. D. Northcutt and E. M. Kuerner, "System Support for Time-Critical Applications", *Proc. of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelberg, Germany, Nov. 1991.

[PaE91] A. Park and P. English, "A Variable Rate Strategy for Retrieving Audio Data From Secondary Storage", *Proceedings of the International Conference on Multimedia Information Systems*, Singapore, Jan. 1991, 135-146.

[PM88] C. Pu, H. Massalin and J. Ioannidis, "The Synthesis Kernel", *Computing Systems 1*, 1 (1988), 11-32.

[RaV91] P. V. Rangan and H. M. Vin, "Designing File Systems For Digital Audio and Video", *Proc. of the 13th ACM Symp. on Operating System Prin.*, Pacific Grove, California, Oct. 1991, 81-94.

[RTY88] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Trans. on Computers*, Aug. 1988, 896-908.

[RDH80] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray and S. C. Purcell, "Pilot: An Operating System for a Personal Computer", *Comm. of the ACM 23*, 2 (Feb. 1980), 81-92.

[RBK92] E. Rennison, R. Baker, D. D. Kim and Y. Lim, "MuX: An X-coexistent Time-Based Multimedia I/O Server", *The X Resource 1* (Jan. 1992), 213-231, O'Reilly and Associates, Inc..

[RiT74] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", *Comm. of the ACM 17*, 7 (July 1974), 365-375.

[RKD85] J. M. Roth, G. S. Kendall and S. L. Decker, "A Network Sound System for UNIX", *Proceedings of the 1985 International Computer Music Conference*, Burnaby, B.C., Canada, Aug. 19-22, 1985, 61-67.

[RAA] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard and W. Neuhauser, "CHORUS Distributed Operating Systems", *Computing Systems 1*, 4 , 305-370.

[SGK85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, "Design and Implementation of the Sun Network File System", *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oregon, June 11-14, 1985, 119-130.

[SaB75] A. R. Saxena and T. H. Bredt, "A Structured Specification of a Hierarchical Operating System", *SIGPLAN Notices Notices*, June 1975, 310-318.

[ScG86] R. W. Scheifler and J. Gettys, "The X Window System", *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 79-109.

[ScB90] M. Schroeder and M. Burrows, "Performance of Firefly RPC", *Trans. Computer Systems 8*, 1 (Dec. 3-6, 1989), 1-17. also Proc. of the 12th ACM Symp. on Operating System Prin..

[SLM90] M. L. Scott, T. J. LeBlanc, B. D. Marsh, T. G. Becker, C. Dubnicki, E. P. Markatos and N. G. Smithline, "Implementation Issues For the Psyche Multiprocessor Operating System", *Computing Systems 3, 1* (Winter 1990), 101-137.

[SRL90] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Trans. on Computers 39*, 9 (Sep. 1990), 1175-1185.

[StL89] D. Steinberg and T. Learmont, "The Multimedia File System", *Proc. 1989 International Computer Music Conference*, Columbus, Ohio, Nov. 2-3, 1989, 307-311.

[SSS87a] *NeWS Manual*, Sun Microsystems, Inc., March 1987.

[SSS87b] *The SPARC Architecture Manual*, Sun Microsystems, Inc., Mountain View, CA, Aug. 1987.

[SSS90] *SPEC newsletter benchmark results*, Systems Performance Evaluation Cooperative, 1990.

[Tan81] A. S. Tanenbaum, "Network Protocols", *Computing Surveys 13*, 4 (Dec. 1981), 453-489.

[TeS88] D. B. Terry and D. C. Swinehart, "Managing Stored Voice in the Etherphone System", *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

[TNR90] H. Tokuda, T. Nakajima and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", *Proc. of the USENIX Mach Workshop*, Oct. 1990, 73-82.

[Top90] C. Topolcic, "ST-II", *Proc. of the First International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Berkeley, California, Nov. 1990.

[TzA91] S. Tzou and D. P. Anderson, "The Performance of Message-Passing Using Restricted Virtual Memory Remapping", *Software – Practice & Experience 21*, 3 (March 1991).

[Wat88] J. Watkinson, *The Art of Digital Audio*, Focal Press, Boston, 1988.

[Wol91] L. C. Wolf, "A Runtime Environment for Multimedia Communication", *Proc. of the Second International Workshop on Network and Operating Systems Support for Digital Audio*

and Video, Heidelberg, Germany, Nov. 1991.

[YS89] C. Yu, W. Sun, D. Bitton, R. Bruno and J. Tullis, "Efficient Placement of Audio Data on Optical Disks for Real-Time Applications", *Comm. of the ACM 32*, 7 (1989), 862-871.

[ZTS89] P. Zellweger, D. Terry and D. Swinehart, "An Overview of the Etherphone System and its Applications", *Proceedings of the 2nd IEEE Conference on Computer Workstations*, Mar. 1989.