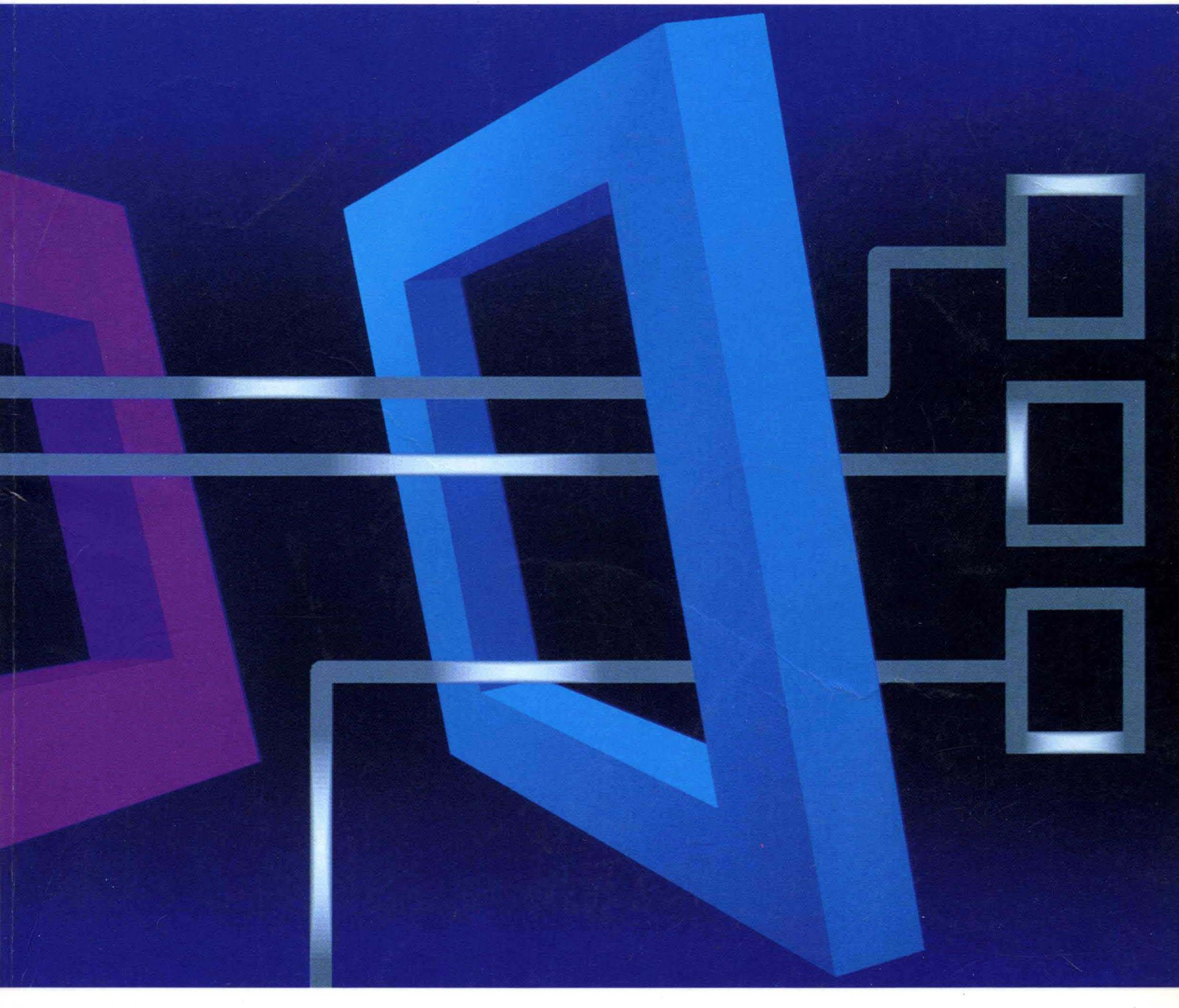
EXPLORING CTOS®



Edna Ilyin Miller, Jim Crook, and June Loy

Exploring CTOS®

Edna Ilyin Miller, Jim Crook, and June Loy

Prentice Hall, Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging-in-Publication Data

```
Exploring CTOS / Unisys Corp.
      p.   cm.
    Includes index.
    ISBN 0-13-297342-1
    1. Distributed operating systems (Computers)  2. CTOS.   I. Unisys
  Corporation.
  QA76.76.063E954  1991
  005.4'3--dc20                                                 90-27131
                                                                    CIP
```

This book was written, edited, and composed using the CTOS-based desktop publishing system, OFIS Document Designer. The system used was a CTOS-based 386 workstateion with VGA graphics. The graphics were created in OFIS Graphics, then integrated into the CTOS Document Designer files. The printer used was an Apple LaserWriter®.

Text is New Century Schoolbook. Helvetica is used for headings. Program listings are set in Courier.

Cover Design: *April Bishop*
Editing: *Carol Collins*
Production Editing: *Antoinette Kohn*
Illustrations: *Jacqueline Mac Millan*
Page Design: *Milena Martin-Arana*

© 1991 by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

The publisher offers discounts on this book when ordered in bulk quantities. For more information, write:

Special Sales/College Marketing
Prentice Hall
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632

All rights reserved. No part of this book may be reproduced, in any form or by any means, without permission in writing from the publisher.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-297342-1

Prentice-Hall International (UK) Limited, *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Simon & Schuster Asia Pte. Ltd., *Singapore*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*

Contents

Figures

Tables

Preface

It has been said that only authors, editors, and those who expect to be acknowledged actually read prefaces, but we are writing one anyway because we have a few things to tell you up front.

Exploring CTOS® is an introduction to an operating system whose name is not widely known, although it is ten years old and well established. The writing of this book was partly prompted by our continuing frustration at reading sober and seemingly well researched articles that discuss at length the difficulties of creating distributed or networked application systems, but never mention CTOS, the only system in which networking is built in and transparent. We tend to share the feelings of one reader of a British publication who, after having read a rare but laudatory article about CTOS therein, wrote a letter to the editor saying:

. . . Hopefully the marketing of CTOS/Open will lead to an end to the networked micro debate and rid us of the awful technology games that advocates have played on corporations throughout the world. After all, if you want to fly from New York to London, you wouldn't choose a Sopwith Camel with an RB-211 strapped to the tail, you'd go 747.[1]

---

1 The Guardian, London, October 17, 1989, Letters Page. Letter from Mike Fitzsimmons, Computer Systems Manager, BBC Broadcasting Research, in response to an article by Jack Schofield that appeared October 8, 1989.

As with all large pieces of software, the people around CTOS have anthropomorphized it. CTOS has a warm, if unusual, personality. It tends to make converts of people from other technical religions, even if in the beginning they regard it with suspicion. First, however, they have to get an introductory acquaintance with it. That is all that this book hopes to provide. We think that computer people in management and marketing, as well as systems people, will be able to follow our description here. Other, more detailed reference documents are available for the programmer who really wants to try out the system.

The book is divided into two parts. Part 1 gives a general overview and some history of CTOS. Part 2 explains a little bit more about what distributed applications are and takes a technical tour of CTOS to support our belief that it is the preeminent platform for such applications.

A note of explanation is in order about the fact that when we describe the history and evolution of CTOS in Chapter 3, we do not name the characters in the drama. The idea here is that the book is about CTOS, not about individuals. Many more people contributed to this system than we could name or even track down at this point. Knowledgeable CTOS people who have helped us by reviewing the manuscript have torn their hair out trying to identify the mentioned players, but our belief is that most readers care less about the names than about the story. (We are, of course, threatening to follow up with a full-length exposé that names names and tells all, but these threats are probably toothless.)

We do want to name some names here, however: many people helped us to put this book together. We are especially grateful to Jon Huie, Irv Kalb, Jeff Seideman, Jay Spitzen, Esq., and Fred Zucker for taking time from other, busy lives to help us reconstruct earlier days of CTOS development. Some very important friends also helped us extensively with old memories, as well as with concrete, current technical information and finally with manuscript reviews: Karen Bedard, David Christie, Drew Hoffman, Jeff Krause, David "Louie" Renaud, Steve Seike, Dave Stearns, and Greg Walsh.

We are also grateful for technical information and manuscript reviews from Joe Altmaier, Stan Balazer, Thomas Ball, Michel Bouckaert, Alan Coleman, Peter Cressman, Kevin D'Souza, Mark Emmerich, Jim Frandeen, Leo Freaney, Christina Gibbs, Dan Gilson, Ed Kaulakis, Andy Little, Carolyn Lowenthal, Gene Lucier, Rakesh Mahajan, John McGinty, Nitin Mehrotra, Alan Merrell, Bert Miller, Val Miller, Margaret Morris, Dave Mullins, Dan Oberrotman, John Quattlebaum, Mike Ribble, Steve Ricossa, Roger Roman, Noah Shaffer, Bill Thomson, Karen Toland, Gijs Van Reijendam, and Bob Walker.

Special thanks go to Tom Germond for his initial legwork and first concept for the book. In the course of many iterations, we have altered it considerably from his plan, but the original themes that he identified still peek through.

Much material in Chapters 6 and 7, which explain interprocess communication and system services, is based on descriptions from an unpublished paper by Patrice Bremond-Gregoire. We thank him for generously permitting us to use it, thus expediting our efforts.

CTOS programmers everywhere will join us in thanking Joe Altmaier and Thomas Ball for allowing us to publish their ServerGen program (Appendix A), a template that gives the inexperienced system service writer a real boost.

All errors, of course, are our own responsibility; but we thank editor Carol Collins for her expert work in helping to ferret out as many as possible. Milena Martin-Arana, Jacqueline Mac Millan, Nettie Kohn, and April Bishop handled the artistic side of the book with patience, humor, and aplomb. We also thank Andrew Keim for his last minute assistance. Linnea de Jaager, Gloria Baker, and William V. Vroman provided personal and professional encouragement and support beyond the call of duty as we struggled to convert a raging ocean of ideas and concepts into a few words on paper.

Finally, EM would like to thank Dave Stearns for being the teacher and mentor he is, and Anna Ilyin McClain and Anita Eagleton for their continuing support and reassurances when the ocean just seemed too deep. JC wants to thank Elizabeth Groom for pushing (an understatement) both CTOS and this engineer in the early days of CTOS, and he would also like to thank his daughter, Charla, for continually demanding equal time! June thanks her family, Greg, Ben, and Jessica, and her colleagues for their support during those hectic weeks as we finalized this book.

Welcome to the CTOS world! We hope you enjoy the trip.

Edna Ilyin Miller
Jim Crook
June Loy

Trademarks

Apple, Appletalk, and Macintosh are registered trademarks of Apple Computer, Inc.

Postscript is a registered trademark of Adobe Systems, Inc.

AT & T and UNIX are registered trademarks of American Telephone and Telegraph Corporation.

BTOS is a trademark of Unisys Corporation.

Bull is a registered trademark of Bull S.A.

Convergent, Convergent Technologies, CTOS, and NGEN are registered trademarks of Convergent Technologies, Inc.

AWS, ClusterCard, ClusterShare, Context Manager, CTOS/VM, DISTRIX, IWS, Shared Resource Processor, SRP, Telecluster, Voice Processor, and X–Bus are trademarks of Convergent Technologies, Inc.

CP/M is a registered trademark of Digital Research, Inc.

Intel and Multibus are registered trademarks of Intel Corporation.

IBM, OS/2, PC/AT, and PS/2, are registered trademarks of International Business Machines Corporation.

Lotus is a registered trademark of Lotus Development Corporation.

Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corporation.

Motorola is a registered trademark of Motorola Corporation.

Oracle is a registered trademark of Oracle Corporation.

Novell and NetWare are registered trademarks of Novell, Inc.

Sun is a registered trademark of Sun Microsystems, Inc.

Unisys is a registered trademark of Unisys Corporation.

Xerox is a registered trademark of Xerox Corporation.

Part 1

An Introduction to CTOS

1

CTOS in Brief

CTOS is different from other operating systems because its architecture is modular and it is easily extended and customized . . . it is optimized primarily as a platform for a particular situation: the modern business and the way most people really do their work.

CTOS® is a protected-mode operating system that has a built-in local area network. It runs on desktop workstations and servers based on the Intel® family of 80*x*86 microprocessors, elegantly exploiting most of the advanced features of these chips.

Although it has not had wide publicity, CTOS has had a steadily growing installed base since its original introduction by Convergent Technologies® in 1980. Today, it runs on almost 800,000 computers around the world. An advanced design to begin with, CTOS has continued to be a technical leader since its inception.

CTOS is different from other operating systems because its architecture is modular and it is easily extended and customized. Flexibility is its hallmark, but it does not attempt to be, in fact, all things to all people. Rather, it is optimized primarily as platform for a particular situation: the modern business and the way most people really do their work.

The Working Model

In a common scenario, individual people work on computers located in their own offices or work areas. They need fast local computation, and also efficient access to resources that are centralized at a server to reduce costs. They need to share information. In other words, they need simple distributed computing.

People also do not normally work sequentially. Instead, they do what is called multitasking: they intersperse work on several ongoing tasks at once. (For example, a person may be referring to a spreadsheet, making corrections in a document, and exchanging electronic mail messages with other people, all at the same time.)

CTOS and the hardware that it runs on are a stable platform optimized for this model of distributed, multitasking computing. Resilient and reliable, it offers real-time, distributed processing that businesses can count on for their mission-critical applications. Because CTOS is multitasking, users can run several applications at the same time, switching from one to the other as needed without necessarily stopping execution. Because it is distributed, users have easy access to networked resources and services.

The CTOS platform efficiently supports not only end users, but also the software developers who write applications for them. It provides developers with an environment in which the necessary networking involved in distributed applications is provided transparently by CTOS. This local area network (LAN) is built-in at the lowest levels of the operating system and hardware. Programs written using the CTOS application programming interface are automatically networked.

CTOS Support for the Model

The technical bases for this good fit between CTOS and its users are its multitasking (multiprogramming) nature and its very fast remote procedure call (RPC).

An RPC is like an ordinary program procedure call, except that it is served by code that resides on another computer, across a network connection from the caller. Ideally the RPC works in such a way that an application can be oblivious to the location where the call is actually served.

Most personal computers and workstations are tied together by add-on networks, so that on these systems, the RPC has been added on at a higher level of software and hardware. These operating systems were also not built from the start with networking in mind, as CTOS was. The result is that their networking is more cumbersome and their RPC is slower.

CTOS supports fast RPC primarily in two ways: it is a message-based operating system in which data is only selectively copied; and its fundamental LAN conforms to the way most users access resources.

A Message-Based Operating System

A system running CTOS has multiple processes, or threads of execution. These processes communicate with each other via small messages through a mailbox-like system. A passed message can point to a larger data item that one process wants to hand to another process. Pointers can merely be exchanged, and large amounts of data do not necessarily have to be copied. (Only pertinent data is copied when messages go across network connections.)

A Simple Cluster LAN

The CTOS cluster links a central server workstation to client workstations. The cluster is implemented through a simple bus topology with RS-422/485 connections, or through TeleCluster™, which connects the workstations over twisted pair (telephone wiring) in a star configuration.

Cluster networking capability is built into the operating system. The client workstations have access to resources at the server, but not the converse.

This arrangement effectively supports the way most people work most of the time. Because the configuration is simple, the CTOS cluster communication software can be highly optimized, and the cluster LAN can provide unusually high performance (throughput) at moderate cost.

Of course, for those who need peer-to-peer access among workstations, this capability can be added in a higher software layer. This easy addition illustrates another important aspect of CTOS: its extensibility.

A Modular, Extensible System

CTOS was designed well before terms such as “message-based” or “multitasking” came into fashion. Its modular, extensible architecture, however, reflects its designers’ innate understanding of the principles that later came to underlie distributed systems.

The CTOS operating system has a very small kernel, or group of primitive operations; but most of the CTOS system environment is made up of modules called system services. These system services have roles that would be part of kernel software in many other operating systems: they handle the file system, various devices, and so on. System services can be loaded dynamically as needed. They communicate with their application program clients and with each other by means of the messages we have described. As a matter of fact, this is the basis of the new micro-kernel architectures.

Because CTOS is modular, it is easily extended or customized. A system service can be written by any experienced programmer and added to the system, or a new system service can be substituted for an existing one. Such a substitution does not require any other alterations as long as the messaging interface is maintained.

Because of the scaffolding that is already present, it has not been hard for CTOS developers to add what one of them calls "bolt-on-beauties" to CTOS as time and technical breakthroughs have gone on. In this way, peer-to-peer networking, the ability to handle a POSIX interface, and many other capabilities have been added, and there will be many more.

The Old Way

There was a time when the demands on an operating system were simple, and it could be designed as a monolithic collection of subroutines that performed commonly needed duties. As time passed and needs grew, more and more subroutines had to be added. To support peripherals and communication, device-specific drivers were written. In fact, specificity was a hallmark of this approach. It was like a wall composed of irregularly shaped stones. Nothing in the structure could be changed or updated without changing the whole thing. A computer running such an operating system had to load all of it, not just the needed parts.

Over time, such a structure became more and more elaborate, rigid, and enormous. The only other choice was to allow the operating system to remain limited and eventually to become too unsophisticated to support users' needs.

Either way, at some point designers would have to grapple with the problems of large-scale kernel rewrites and whole new operating systems that somehow retained backward compatibility. Then they would have to stabilize and debug their new systems all over again.

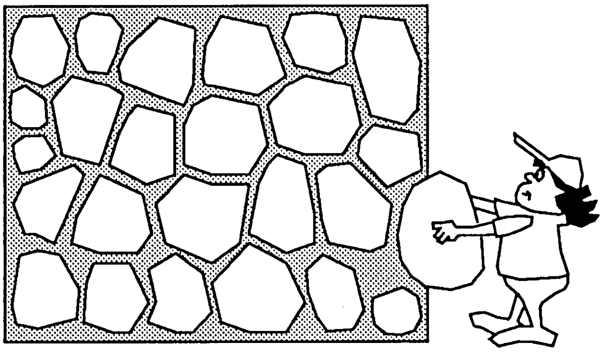


Figure 1-1. The Old Way

A Building Blocks Approach

How much simpler it would be to keep an operating system current if it were made up of separate building blocks with clean, regular interfaces! These blocks could be put together, taken apart, substituted, and reassembled in configurations that included only the needed functions. People other than the original designers could add or substitute blocks of their own. Maybe the blocks would not all have to be in the same place to work together.

CTOS is that building-block operating system. Its parts are separated so that they function independently. The building blocks of CTOS are processes; the mortar is made up of messages and exchanges.

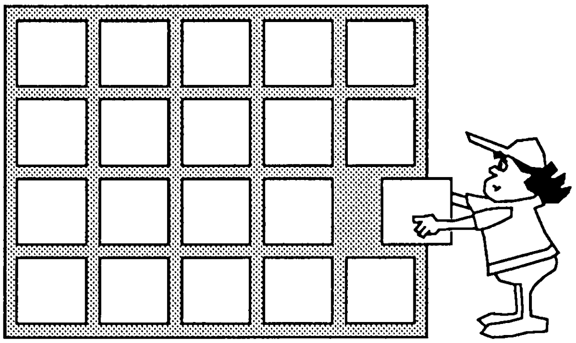


Figure 1-2. The Building Blocks Approach

Processes: the Building Blocks

For years, larger computers with single processors have been doing multitasking. This sleight of hand makes them appear to run more than one program at a time. Most human users naturally do their work in this way. The idea is not new, but small-computer users have only recently been exposed to it.

Multitasking is also called multiprogramming, where more than one program is running at once, sharing the central processing unit over time in some way. Now consider that each program may be composed of more than one process. A process is an independent thread of execution, together with the hardware context (the processor register values) necessary to that thread.

CTOS supports independent invocation and scheduling of multiple concurrent, independent processes. CTOS processes, whether they are application processes or are parts of CTOS itself, are regular building blocks with clean interfaces.

For example, an electronic mail program might have two processes: one allows the user to edit a mail message, while the other monitors incoming mail. Not only does the mail program compete with other programs for use of the processor; the two processes within it compete with each other and with all the processes in all the other programs for processor time. Each must get the processor time that it needs to do a good job for the user.

Process Scheduling

Multiple processes obviously do not really run at the same time on one microprocessor, but it definitely appears so to the user. The different computer systems that offer multitasking use varying mechanisms to simulate this effect.

How does CTOS achieve the "simultaneous execution" of multiple processes and make them all look as if they were running at "normal" speed?

Each process (thread of execution) within CTOS is assigned a priority and is scheduled for execution based upon that priority. The CTOS kernel scheduler performs the scheduling of the processes.

Process scheduling is driven by events. Whenever an event occurs during execution of a process, such as an input/output event (I/O), that process can lose control of the processor. A higher-priority process that is eligible for execution is scheduled for immediate execution. This type of scheduling technique is called event-driven, priority-ordered scheduling.

Messages: the Mortar

In order to do business, individual CTOS processes send messages to each other. This mechanism is called interprocess communication (IPC). Although at the most primitive level, a message can be anything, it is usually a memory address at which some relevant item of information can be found.

A message is passed from one process to another via an exchange. An exchange is like a mailbox; it is a place where processes wait to receive messages or, where messages are deposited to wait to be processed. Each process is allocated an exchange when it is created, and it can ask for more if it needs them.

CTOS is not unique in its use of this message-based model. It is unique in the use of a special type of message, the request for service, which is usually referred to more simply by the term request.

The request is the most common message in CTOS. Requests are specially formatted messages that include a request block header that includes a request code, which identifies the desired service, along with other information that will be needed by the service, such as where to send the response and who is sending the request.

With the help of the CTOS kernel, the request travels transparently to the user or application program across networks to locate any special service.

To make it clear how this works, let's draw a simple analogy between the way people use requests and the way CTOS processes use them.

Suppose Mary needs to have the Acme file copied so that she can take it with her to her next client meeting. She writes the following note (request)

"I need a copy of the Acme file on my desk ASAP!"

and hands it to her administrative assistant, John. John pulls the file from his cabinet and, because there is no copier in the office, decides to send it to a copy center. He asks Fred, who operates the copy center, in a different building, to return the copy to Mary's desk. Fred makes the copy and delivers it to the desk where Mary is waiting.

If Mary and Fred represent CTOS processes and John represents the CTOS kernel, you can think of the interaction described above like this: Mary creates a request that indicates the desired service (copying), some additional information (the name of the Acme file), and the response exchange (her desk) or where she will wait until the service is completed. She hands the request to John (the CTOS kernel), who decides to route the request to Fred at the copy center. Fred does the service and sends the results to the response exchange (Mary's desk).

Mary could have elected to leave the office while the copy was made and to just check her desk occasionally to see if the copy was ready yet. In real life she certainly would have. As a CTOS process, she could also just stop and wait for the file to appear, because she could count on the service being performed very quickly.

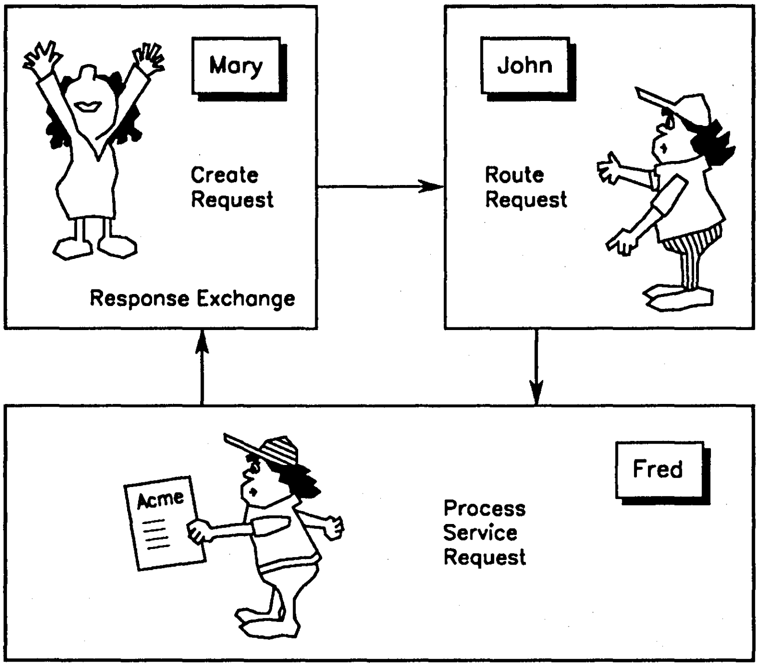


Figure 1-3. An Example Showing Messages and Exchanges

The Roles of Interprocess Communication (IPC)

CTOS is a message-based operating system. In CTOS interprocess communication, exchanges serve as message centers where processes send messages or where they wait or check for messages. Overhead is minimized, because, unless the request must go across the network, only the address of data is passed, not the data itself.

Interprocess communications (IPC) actually has two different roles in the CTOS world. IPC is the means of communication and of transmission of data from one process to another. IPC also allows synchronization of processes (controlling when they stop and start executing relative to each other). Thus, it provides a means for the orderly sharing of resources among processes. We shall discuss these aspects of IPC in Part 2.

The Role of the CTOS Kernel

In any exchange of messages, the CTOS kernel (like John in the example above) is quietly and efficiently involved. At about 4000 bytes, it is tiny, primitive, and powerful, containing only a few vital functions. The CTOS kernel creates processes, assigning their exchanges, among other things. It schedules processes preemptively for execution, based on priorities (0 through 255) that it has assigned to them. It acts as a postal service for communicating processes, delivering messages back and forth between their exchanges. It also controls inter-CPU communication (ICC) on larger servers that have multiple processors.

System Services

Nearly all the other functions that one normally associates with an operating system are actually performed by system service processes. System services manage resources (the file system, communications, and so on) and provide services that are requested by application program processes and by other system service processes. In the analogy we used before, Fred was a system service (copy service).

System services are well-behaved building blocks in the CTOS system. Because of their standard message-based interfaces with the rest of the CTOS world, they can be removed, substituted, and added at will. The "greater CTOS" thus can be efficiently tailored to specific situations.

A system service process receives IPC messages to request the performance of its services. Examples of operating system services include opening or closing disk files or accepting keyboard input.

System services can be linked in with the operating system or can be dynamically installed. In operation, a dynamically installed system service is indistinguishable from a linked-in system service.

The use of system service processes and the formalized interface provided by IPC results in a highly modular environment that increases both reliability and flexibility.

Operating Across a Network

The true beauty of a CTOS system service is that it can operate across a network transparently to the process that requests its services. For example, an application process on one computer can send off a request to a system service to have a certain job performed; but the application does not have to know where the system service resides. If it turns out that the desired system service is not on the local machine, the request is automatically routed across the network to where the service does reside. The response comes back in the same way.

Remember that we said the request procedural interface is designed to make it easy to pass such messages. Remember that in the analogy we used before Mary and Fred did not have offices in the same place (computer). Somehow the operating system set things up so that they did not need to know the exact location (address) of each other's mailbox to exchange messages.

If the CTOS kernel has only essential responsibilities, then how is this routing carried out? By additional system services that specialize in routing the requests. They are called agents, and they are of the class of filter processes, which trap and manipulate messages aimed at other services. We shall see a great deal more of them and of other system services in Part 2.

What Can You Do With This?

This way of doing things makes life easy for developers of distributed applications. First, all the messaging we have talked about is neatly hidden under standard application program interfaces (API) that look just like traditional subroutine calls, so there is no new mechanism to learn unless you want to.

Next, you can write a system service yourself. A new system service can be part of an application or can be an extension to the operating system. A system service is just a program that observes certain rules and makes a few necessary calls when starting and terminating. It does not take very long to understand how system services work or to learn how to write the simplest kind of system service.

Creating a more sophisticated system service as part of your application allows you to place certain program functions in one location on a network and have many users or instances of the program effectively share code across that network. The same version of your application works on any physical configuration, whether it involves one standalone system, a small local area network, or a larger network. There is no such thing as a separate network version of your application.

We briefly mentioned earlier that CTOS has the cluster network built-in. LAN capabilities do not have to be "bolted on": the local network is part of CTOS. The implications here for simplifying the development of distributed applications such as electronic mail programs, for example, are enormous. Part 2 of this book will explore those implications in more detail.

The world in which CTOS lives and plays is moving toward architectures in which software operates in a continuous loop that handles whichever of multiple possible events occurs. These event-loop architectures support the graphical user interfaces of the future. CTOS system services, which are event-loop entities, are ideally suited for this environment.

What Does It Look Like?

CTOS obviously does not exist and perform in a vacuum. In fact, this modular operating system runs on modular hardware. The next chapter gives a quick overview of the physical side of the CTOS world.

2 Physical Systems

Because of its modular design, the CTOS workstation can be easily and quickly configured with as many or as few features as are needed, and features such as disk expansion, graphics, or voice processing can be added as needs and network configurations change.

Modular CTOS runs on modular, intelligent desktop workstations that can stand alone, but that reach their full potential when configured into local- and wide-area networks. The standard CTOS workstation comes in pieces that look rather like groups of medium-size books. Each of these modules supports a special function and set of features: processing, mass storage, graphics, tape backup, and so on.

Easily latched together without tools by an untrained user, the computer has an oblong footprint that allows it to fit easily on a desk, a bookshelf, or other office furniture. Because the monitor and keyboard can be up to 16 feet away from the processor, the workstation can easily fit into an environment in which space is at a premium.

Figures 2-1 through 2-2 show simple workstation configurations.

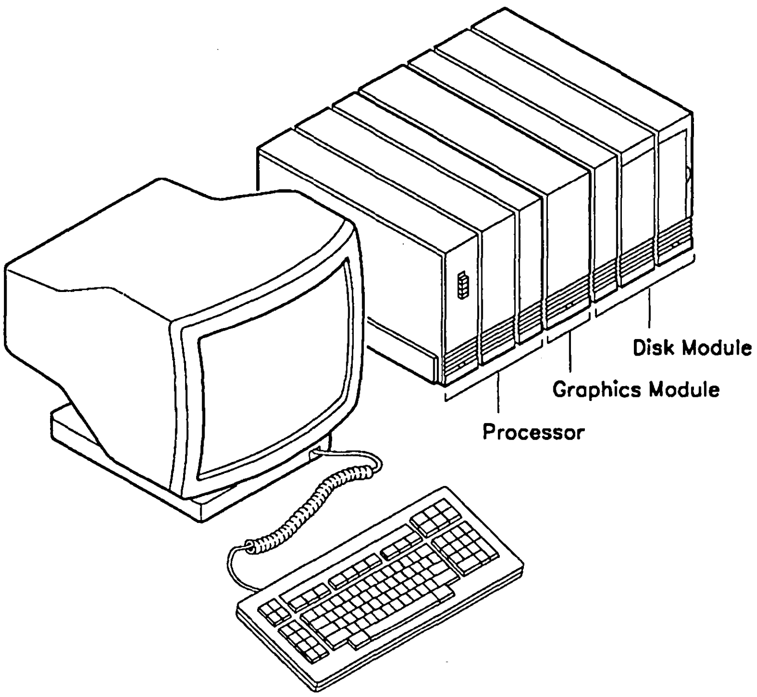


Figure 2-1. A CTOS Modular Workstation

Because of its modular design, the CTOS workstation can be configured with as many or as few features as are needed; and features such as disk expansion, graphics, or voice processing can be added as needs and network configurations change. Disks and special function modules can be easily moved from one system to another as needed. Even processor units can be easily changed for upgrade or repair.

Certain processor models combine commonly needed features into one enclosure: for example, the Intel 80386-based integrated workstation contains hard and floppy disk drives as well as the processor, memory, and power supply. Integrated workstations can take additional cards as well as modules to enhance functionality.

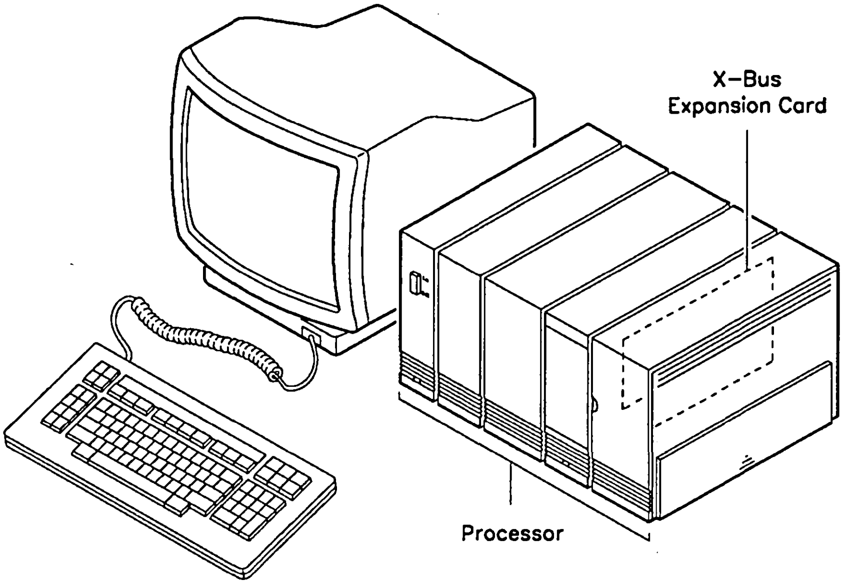


Figure 2-2. A CTOS Integrated Workstation

Many special function modules are available from OEM manufacturers around the world. They include a widely varied set of modules that expand communications functions as well as those that provide basic local functionality such as memory and disk storage.

Table 2-1 lists some of the basic CTOS workstation modules.

Table 2-1. Some Types of Workstation Modules

Module	Description
Processor	Includes 80286 or 80386 processor, system RAM, 2 RS-232 ports, RS-422 or RS-485 port, Centronics-compatible parallel port (bidirectional in some models).
Expandable Processor	Includes 80286 or 80386 processor, system RAM, 2 RS-232 ports, RS-485 port, Centronics-compatible parallel port, and can be expanded with special function cards or modules.
Integrated Processor	Consists of 80286 or 80386 processor module components with SCSI hard and floppy disks and power supply in one module, and can be expanded with special function cards or modules.
Disk Storage	Available in several sizes and types, including floppy/hard disk combination, floppy disk, removable SCSI disk, SCSI hard disk, disk expansion, CD-ROM.
Tape Storage	Provides SCSI quarter-inch streaming tape for backup.
Graphics Controller	Has support for VGA+ compatibility; hardware graphics accelerator; 1024 x 768 pixels.
General Communications	Several modules, some of which contain coprocessors, allow the addition of RS-232 communications ports.
Appletalk®	Allows attachment to Appletalk network.
Ethernet	Allows attachment to Ethernet network.
Token Ring	Allows attachment to Token Ring network.
Voice Processor™	Contains CODEC, DTMF and rotary signaling devices, DTMF tone decoder, etc., for connection to voice and data networks.
FAX	Receives and sends FAX messages.

The X-Bus

What makes this arrangement work is the Extensible Bus (X-Bus™), which provides the mechanical, logical, and power connections between modules. The pins and sockets that allow the X-Bus segment in one module to be attached to the segment in the next module can be seen along the lower edges of the sides of each module. As one module is latched to another, the X-Bus can be extended out to 24 inches.

The system modules are linked to and interact with the workstation processor module via the X-Bus. Figure 2-3 shows this connection.

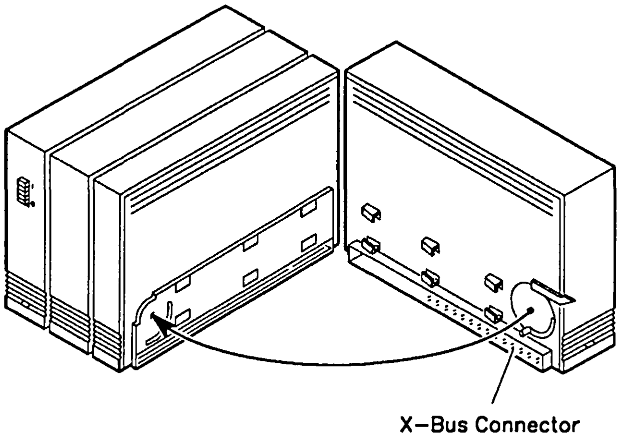


Figure 2-3. Workstation Modules Showing the X-Bus Connection

Cluster and TeleCluster

Workstations can be connected to form a local area network called a cluster. One workstation (usually with many resources, such as disk and tape storage, printers, communications gateways, and so on) is designated as the server workstation. All other workstations on the cluster can use resources at the server, as well as their own.

The server does not need to be a dedicated server used only as a file server as is the case in many other LAN environments: it can also be used as a normal workstation for one of the users on the cluster. Nor do all the workstations on a cluster have to be of the same processor model: more recently acquired workstations continue to work with older ones.

Under this arrangement, some or all cluster workstations can be simply one processor module with no disk at all or a nonexpandable workstation consisting only of a processor/video controller unit, also without a disk. These machines use the disk resources at the server. Diskless workstations reduce the cost of setting up a cluster; but because they retain the sophisticated processing power of the workstation, they remain highly responsive to the individual user. (In fact, a cluster of only four workstations is price competitive with the same number of personal computers on a LAN, and it has more capabilities.) If cluster workstations do have local disks, they can continue to work using their local disks even when the server is not running.

The maximum number of workstations that can be included on one cluster varies with the specific type of processor unit used for the server and with operating system configuration. Because these figures change with new releases, we will not cite maximums here. Suffice it to say that many small companies, and most departments within large companies, are able to include all their employees on one cluster.

When it is desirable to have a very large cluster or to have a great deal of centralized disk storage, larger dedicated servers with multiple, loosely coupled processors can be set up. Such servers are often called Shared Resource Processors.™ The loosely coupled processors that make up a Shared Resource Processor can be dedicated to various functions. The processors run the CTOS operating system and communicate with each other and with the rest of the cluster LAN via the same message-based mechanism described in Chapter 1. In line with the other CTOS computers, the Shared Resource Processor is a modular machine that can accommodate as many specialized processors as required in up to 6 expansion cabinets.

The workstation, CTOS, and the cluster were originally designed at a start-up computer company called Convergent Technologies in 1979. At that time, the decision was made to allow cluster workstations to have access to disks at the server (client-server), but not the converse (peer-to-peer). This arrangement allows greater security for individual workstations than one in which all workstations can access files on all others. (More recently, a peer-to-peer communication capability has been added as part of a higher communications software layer distributed with the CTOS Network software.)

The members of a cluster can be connected to each other in either of two ways: via a standard RS-422 cable connection or via the TeleCluster™ hardware. TeleCluster allows connection via existing building telephone twisted-pair wiring. TeleCluster is far more cost effective than any other method of installing any LAN and is the method of choice where building wiring permits it. Addition of a workstation to the cluster is a simple matter of connecting the cables.

Whatever the physical connections of the cluster, it is very simple to administer because the cluster concept is built into CTOS and is not added on later. A cluster can be set up and maintained by a nontechnical system administrator. Moreover, the same operating environment runs on all hardware, thus simplifying matters for end-users, system administrators, and programmers, alike.

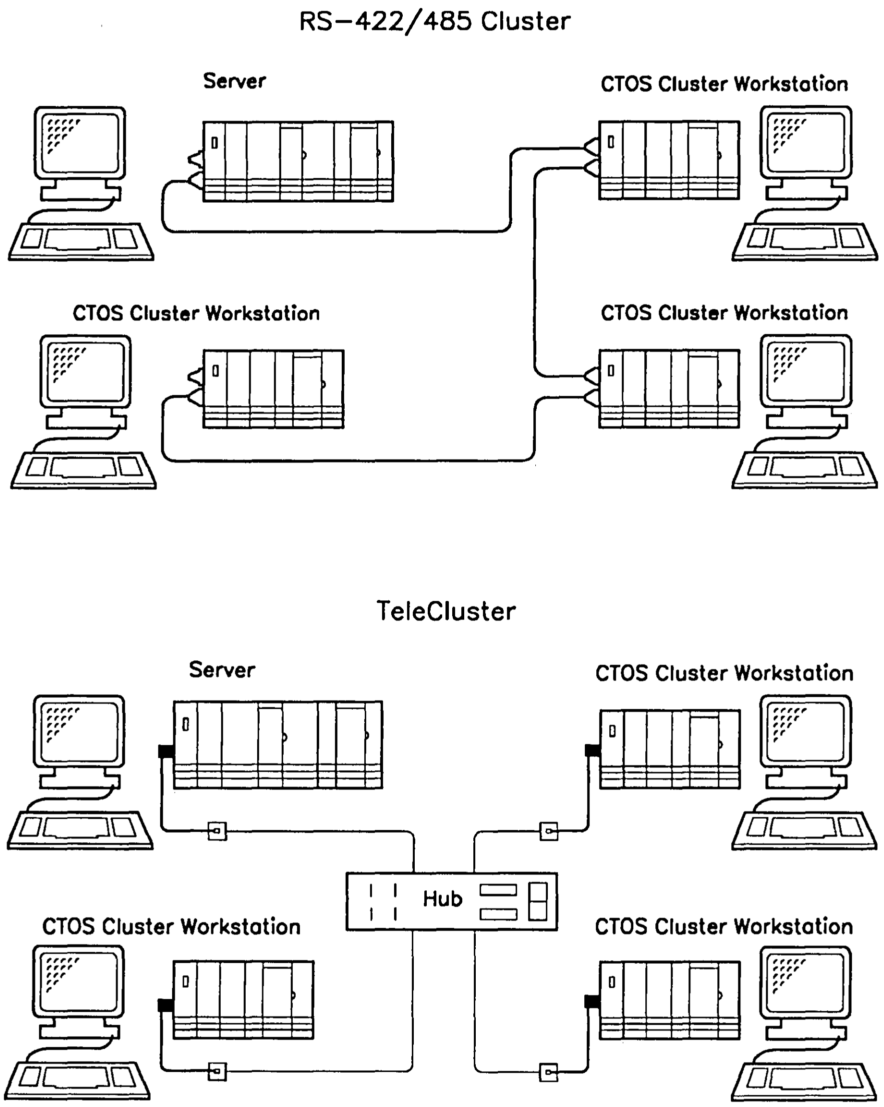


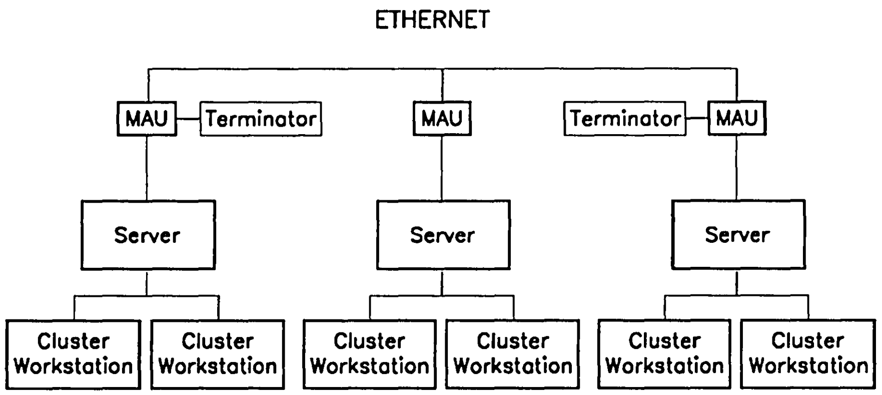
Figure 2-4. Two CTOS Clusters

Larger Networks

Clusters can in turn be connected via the CTOS Network software (sold under various names, such as BNet or CT-Net), which establishes transparent peer-to-peer connections among CTOS server workstations. Each connected server is called a node. The CTOS Network operates as a logical extension of the cluster. Users can simply add the node name to a file specification to access files across the network.

The CTOS Network is media independent and can operate on RS-232, switched, leased, synchronous, and asynchronous lines, as well as X.25 packet-switched networks, Token Ring, and Ethernet (thin, thick, or twisted-pair).

As with the cluster, messages are passed over the CTOS Network transparently to the originating program. Thus an application on a cluster workstation can request a service that is not on that local workstation, and the request can be transparently routed not only to the server of that cluster, but beyond it to other network nodes for service. All this routing activity is transparent not only to the human user of the applications, but also to the application programmer. Through these connections, any resource (file, dataset, printing service, mail, communication service) is available to any cluster workstation.



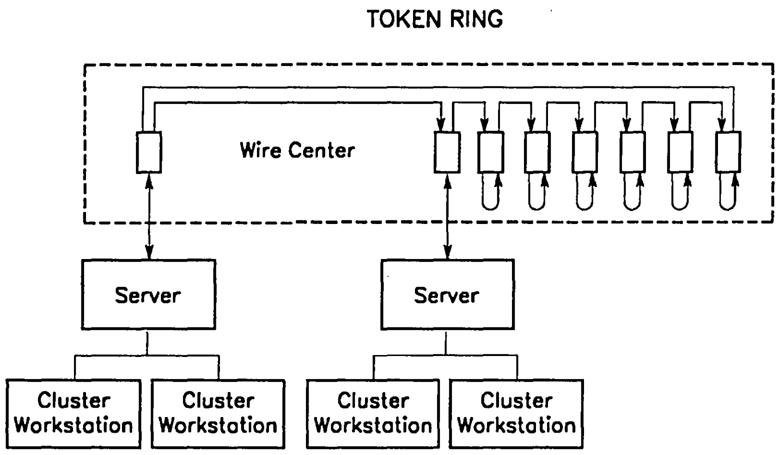


Figure 2-5. Two CTOS Networks

Running MS-DOS Programs

PC Emulator

As we shall see in the next chapter, CTOS systems were never intended as personal computers and were never offered to the general public through retail outlets. As time went on, however, and the IBM® PC and its relatives became popular, more and more application software was written for these personal systems. Eventually, users of CTOS systems wanted to run some of these programs.

CTOS developers responded by creating in several stages the capability of running MS-DOS-based programs under CTOS. On the 80386-based processors, CTOS uses the microprocessor's virtual 8086 mode and its own PC Emulator software to support multiple concurrent instances of MS-DOS,® or "virtual PCs". Use of a VGA monitor with the workstation allows a very high degree of MS-DOS program compatibility. The 80286-based processors can also run PC Emulator software if a PC Emulator coprocessor module is attached.

By comparison, both UNIX® and OS/2™ plan to have in the near future the same sort of capability that CTOS currently has. UNIX provides MS-DOS functionality on an 80386-based workstation with the MS-DOS Merge product, allowing the simultaneous execution of both MS-DOS and UNIX programs. OS/2 currently utilizes the MS-DOS compatibility box, where the system must be switched during run time from OS/2 to MS-DOS and vice versa. In the future, OS/2 will utilize the VM-8086 feature of the 80386 chip to allow the coexecution of several MS-DOS sessions simultaneously, a feature that CTOS has provided for several years.

ClusterCard and ClusterShare

Running MS-DOS-based programs directly on a CTOS workstation was not the only feature users wanted. Some users wanted the power and network capabilities of the CTOS cluster, but already had an investment in simpler MS-DOS oriented hardware. For these users, the ClusterCard™ board and ClusterShare™ software were invented. ClusterCard is an expansion board that fits any standard PC expansion slot and automatically configures itself for the 8-bit PC or 16-bit AT expansion bus. ClusterShare is the software interface. It consists of two parts: a system service that runs at the server and an MS-DOS driver that runs on the PC.

ClusterCard and ClusterShare integrate PCs into the cluster, allowing them to use the CTOS server. The server can provide file, disk, printer, and mail services to the PCs. Because ClusterShare uses the CTOS file system, PCs can use files that are larger than 32M bytes on the CTOS server workstation.

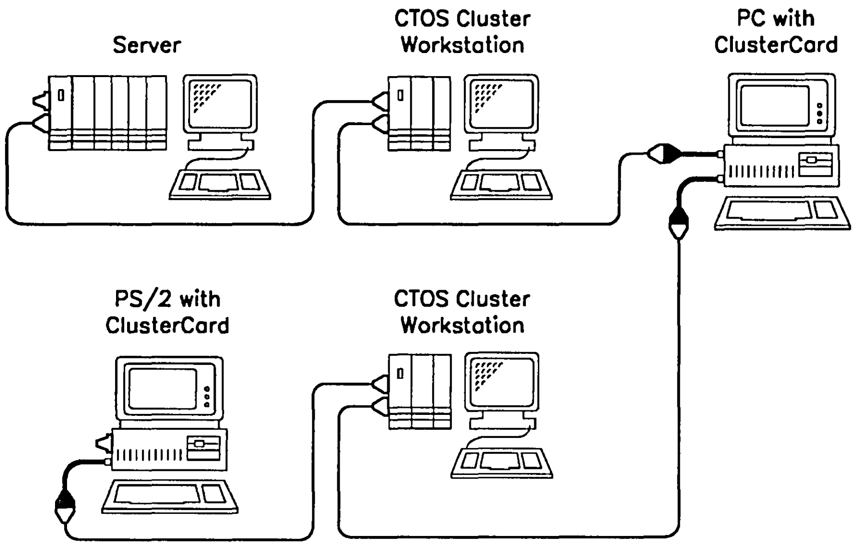


Figure 2-6. A CTOS Cluster with PC, PS/2®, and CTOS Workstations

Why CTOS Is What It Is

CTOS and the CTOS workstations and servers are now at an exciting time in their history. CTOS is evolving from a well-kept secret into a well-known open system. Its developers are confronting thorny technical and philosophical questions as they turn this corner. To understand how CTOS became what it is and to see where it is going, we need to step back into its past.

3

Of History, Religion, and Marketing

The principal designers decided early on to create a message-based operating system with low overhead that would handle multiple processes in as near to real time as possible . . . The designers understood that CTOS would live in a changing technical world. The modular design would allow it to be updated easily in the future, as well as tailored to the special needs of various customers.

As is true of many other start-up computer firms, wild and wonderful tales of exotic personalities and technical derring-do surround the early days of Convergent Technologies. Some are true, some apocryphal; but together the stories convey the feeling of a time that really did exist and really was exhilarating for those who were there.

Convergent® (as it came to be called) was formed by a small group of hardware engineering and marketing people who left Intel Corporation to do so in August of 1979. Convergent culture still retains joking references to designing things on paper napkins, because the concept for the Integrated Workstation (IWS™), Convergent's first hardware product, is supposed to have been sketched on a napkin in a bar as the founders made the decision to go out on their own.

Within a few weeks, the tiny company had hired its first few software engineers. All had extensive experience, but they were from extremely varied backgrounds and had differing and strongly held technical views. No matter what stories are told about the Convergent founders, it is incontestable that they created an atmosphere in which these diverse talents came together and cooperated and learned from each other, with spectacular results.

The First Direction

CTOS has ended up as a potent operating system for distributed business applications. The ideas of the founders, however, had nothing to do with this marketplace. They had been heavily involved in the invention and marketing of the Intel Multibus®, a popular add-on board standard of the time. The associated Intel development systems, called "blue boxes," were hard to use and unreliable.

The original Convergent Technologies product was envisioned as a 16-bit microprocessor-based workstation (the first use of that term) that would be a sleek, easy-to-use replacement for the Multibus and blue-box environment. It was aimed at developers of real-time systems such as test or communications equipment. The company name derived from the convergence of several ripening technologies: the 16-bit microprocessor; a small, built-in hard disk drive; high-quality video; and a truly excellent software development environment coming together in a small desktop system.

The founders allotted one year for the development and release of this new machine: not, as one early developer comments, because there was any rational month-to-month plan, but because they wanted to ship it within that time.

Basic Decisions

Because the founders were from Intel, it may seem obvious that they would base their design for the IWS on the Intel 8086 microprocessor. However, they knew the drawbacks as well as the advantages of the 8086, and the forthcoming Motorola® 68000 was seriously considered. Timing determined the outcome: release of the Motorola chip was delayed, and Convergent went with the 8086 rather than wait several months for a chip that some developers would have preferred to use.

This practical decision was the first of several that did not seem as momentous at the time as they later turned out to be. It was some time later that the IBM PC® was announced and the 8086 became established through it. It is because CTOS workstations and servers still are based on the Intel 80x86 family of microprocessors that they can run several concurrent instances of MS-DOS today.

With hardware design begun, the small company turned to the question of an operating system for the IWS. The principal software architect had been persuaded to leave Xerox® Corporation's Palo Alto Research Center (PARC) to join Convergent. With a strong academic and research background at Harvard University, SRI, and Xerox PARC, he was eager to create a commercially viable product that would use some of his theoretical ideas, yet be pragmatic. Over the next few weeks, several of his former colleagues at Xerox joined him. Affectionately nicknamed the "Xeroids," they contributed strong research abilities and recent academic ideas, as well as coding skill, to the Convergent mix.

The principals were wise enough to realize that other points of view should be included in the software team. Even before all the Xeroids had joined up, Convergent had added experienced software engineers from the "real worlds" of large data-processing systems (SEL) and PBX design (Bell Northern Research), as well a UNIX developer from AT&T® Bell Laboratories.

The first software decision was whether to port an existing operating system to the new hardware. Choices were limited. CP/M® was not nearly powerful enough. UNIX was still considered to be an unreliable academic system; and besides, it could not handle the real-time requirements of the specification. The only other suitable operating system was RMX-80, the Intel blue-box operating system. It was message-based and closer to the real-time design that was needed; but after some serious consideration, the team rejected it. They believed that they themselves could create something better.

A stimulating period of intellectual exchange followed. The developers were not tied to 8-bit technology or to any kind of backward compatibility. Standards were not yet fashionable. They were free to create the most forward-looking system that could be reasonably produced.

Papers from the computer science literature were passed around, and concepts from other areas of experience were unearthed. Everything was discussed intensely in group sessions that included not only the software team, but also hardware and product design engineers, marketing, and the company officers. Participants remember analyzing ideas from the Xerox Alto, Pilot, and the multiprocessing Thoth (developed at the University of Waterloo), among others. Conflicting ideas certainly arose at this time. A strong corporate culture had already taken root, however, and no schisms occurred. This culture was based on the respect each person had for the technical ability of the others, as well as on communication and trust. This factor of trust actually showed up in the design of CTOS later.

It is significant that those who were present often cannot remember which of them came up with any given aspect of the design. In their descriptions, they use the impersonal voice ("It was decided to do so-and-so") and often give credit to others for important ideas. This way of thinking, discussing, reaching consensus, and pulling together toward a goal became a culture that was passed down to later CTOS developers as they appeared. Even if those who did not subscribe to this culture were exceptionally talented, they did not survive in the long term.

The principal designers decided early on to create a message-based operating system with low overhead that would handle multiple processes in as near to real time as possible. This aspect of the design came not only from RMX-80, but also from the exposure of the Xeroids to message-based experiments and from the experience of the PBX designer who had seen it work before. Generally known message-based operating system designs did have the drawback of overhead related to the passing of data. Because they were designing a simple machine with no protection requirements, the team could get around this problem by passing only the memory addresses of data from one process to another.

Another key (and related) decision was to keep the operating system kernel as small as possible and place many traditional operating system functions in separate processes called system services. Other processes (applications or other system services) would send requests to a system service; it would respond with the desired result and a status code. The CTOS kernel would only manage process scheduling and interprocess communication (message passing). The designers understood that CTOS would live in a changing technical world. The modular design would allow it to be updated easily in the future, as well as tailored to the special needs of various customers.

Memory management was considered an application responsibility. There was one address space, a single memory partition. In the early days, CTOS was multiuser: the file system was one user, the human user the other. This concept was discarded before the first CTOS release, but it laid the basis in CTOS data structures for the much later development of multiple memory partitions.

In deciding to create their own operating system and to make it message-based, the design team unwittingly set themselves up for a happy accident: networking.

Networking From Day Two

Although support for networking was not explicitly included in the first hardware design for the IWS, the primary CTOS architect had networking in the back of his mind from the beginning of the project.

Meanwhile, it became evident within a few months after the effort began that the IWS was going to be an expensive machine for customers. Convergent marketing and sales people began to pressure the engineers to reduce the cost of the IWS by removing, for example, the expensive hard disk and using only floppy disks. This suggestion struck horror to the hearts of the designers. Necessity brought forth the invention: the team came up with the notion of placing the expensive resources on one machine and connecting other, less expensive workstations (with or without their own disks) to that central one in such a way that they could all use the resources transparently. The idea for the Convergent cluster was born, yet how to implement it?

There was no time to redesign the hardware, nor to design an elaborate peer-to-peer network of the kind known at Xerox. It fell to the most pragmatic and least academic of the lead software engineers to figure out how to retrofit networking onto hardware that had only one full-word DMA channel left available, and to design how the CTOS message-passing scheme was going to work across the new cluster.

The Request Procedural Interface

Meanwhile, by the second quarter of 1980, the rudiments of CTOS were up and running. Excitement mounted as the team saw that their ideas were going to work well. There was one drawback, though, which revolved around the choice of a message-based implementation. Writing code explicitly to build data into a request block in client data space and to pass messages from process to process was not compatible with existing high-level programming languages or with the way experienced programmers thought. People were accustomed to the idea of subroutine calls. The new system was technically wonderful, but not particularly friendly. To recruit the allegiance of customers' developers, this situation must change.

The solutions to this problem and to the networking problem came in one revolution. The concept of the way the request block was used was ripped up and redone, an event that reverberated through everyone's work. No longer would the programmer explicitly construct the request block.

The new approach was to hide the construction of the request block under a request procedural interface, which looked exactly like a traditional system call, with what appeared to be a function name followed by parameters. Transparently to the programmer, the linker would recognize requests as such. The new request block would be built, not by the programmer in the client's data space, but by the operating system on the client's stack. The request block would include a new header portion that indicated what the rest of the structure contained. In effect, it became self-describing.

Moreover, the CTOS kernel would not simply pass pointers. It would take on a more active role. Operating system tables would understand the new request block format and be able to determine which exchange (and thus, which system service) was the target of the request. A request for a service that turned out not to be local could then be forwarded across the cluster to the server workstation. New system services, called the cluster workstation agent and the cluster server agent, were written to handle routing.

Some ingenious work had to be done to shoehorn cluster communications into the one remaining DMA channel. It turned out to be possible if one byte of the word was outgoing and the other was incoming. (Later revisions of the IWS hardware design corrected this resource problem.) Using inexpensive RS-422 lines at 307Kb, the first cluster did not have blistering speed, yet the team could see that it would work.

The inventions of the request procedural interface and the self-describing request block were the real innovations in the early design of CTOS. All the other concepts were known, although perhaps not widely used up to that time. It was this breakthrough that made CTOS unique and set it up for a second happy accident and a future that was not at all what its creators had expected.

Changing Course

While the designers worked feverishly in cubicles and labs over the winter, the first Convergent salespeople were already on the road looking for the kinds of customers that the founders had envisioned. To everyone's amazement, they did not appear. (In fact, it is safe to say that very few people ever did create custom Multibus hardware and software to run on an IWS.) The salespeople were getting worried. Finally, in the spring of 1980, they got their first really large nibble.

Perhaps not surprisingly, that nibble came from Xerox Corporation, a company that certainly could appreciate advanced design. Xerox, however, wanted to put its own advanced word processor on the IWS for use in office environments. They also put a strong push behind the fledgling efforts toward networking, insisting that it be part of the product. They were interested in becoming an OEM for a distributed office system. (OEM stands for "original equipment manufacturer," but the term has come to denote a company that buys technical products from another company, puts its own name on them, and resells them.)

Convergent responded by pressing forward with the networked design; but the relationship with Xerox did not last. Within a few months, Xerox decided to pursue another course. Xerox had, however, left an indelible mark on Convergent, in both marketing and engineering.

In marketing, the salespeople now understood where to look for OEM customers. They saw to it that Convergent began to design and build its own word processor.

In engineering, the Xerox experience subtly inserted a difference of opinion about design direction. This difference later grew and was not resolved for three years. The question was this one: was the machine really a tiny, powerful, networked minicomputer replacement, or was it a platform for office applications?

There were designers on each side of the question. Nevertheless, at this time there was no overt disagreement between these two philosophical camps. Money was tight, hours were long, everyone was just pulling together to get the product out, and any customer prospect looked good.

The Early Religion

The first CTOS developers never sat down and talked about philosophy or a design "religion" per se: they were too busy creating the product. One of them likes to say that Nature was not designed but debugged into perfection. Another quips that Jesus did not intend to start a new religion but to reform Judaism. In other times and places, they had all seen things done wrong, and they wanted to do things right.

Nevertheless, religious principles did emerge, even if they were not codified. CTOS should be a message-based and real-time multiprocessing operating system. It should stay small, it should be modular, it should have low overhead. It should stay out of the application's way if the application needed to interface with hardware. (This latter principle was to change later.)

A harder principle to describe is that CTOS should be open and trusting of the application writer. The trust among the members of the first team was reflected in the operating system's belief that applications would be voluntarily well behaved. Since all application development was initially done in house or by a few close OEM customers, this policy was successful. Things remained that way for several years, even after CTOS was supporting multiple partitions with concurrently running programs on the 80186 microprocessor, before the 80286 brought memory protection into the picture.

There were two basic tensions in the early development group. One was the differing viewpoints between the more academic and the more pragmatic members. The other was the most important religious principle of all, which one participant expresses as "Ship it!"

The philosophical tension was used constructively. The research-oriented members brought the message-based concept and networking to the product, among other modern ideas. The pragmatists were concerned with performance, creating a redundant and extra-reliable hard disk, using algorithms that were known to work, keeping the cluster concept down to what really could be done with the time and resources available, and generally finding the simpler, faster, and smaller way to do things. Without the academics, nothing would have been new; without the pragmatists, nothing would have been shipped.

In any case, the religion of shipping it was real. Under the deadline pressure of one year, the Ctosians (*see-TOE-zhuns*), as they came to be called in the Convergent vernacular, together built and shipped the IWS with CTOS version 1.0 on it. This achievement was phenomenal in a world where minicomputer and mainframe design cycles commonly consumed several years.

The cluster code was not all implemented in CTOS 1.0, but the basic development system was up and running. The team of 14 people had written 100,000 lines of systems code; created a linker, a loader, an editor, diagnostics, device drivers, an easy, menu-oriented command-line interpreter, and a sophisticated debugger; and ported several compilers. CTOS 1.0 shipped in October 1980, just over a year after the first software engineers were hired.

More Office Applications

In February of 1981, two important events occurred: CTOS 2.0, with cluster code up and stumbling, had its first customer ship; and Convergent signed its first really big contract. A system integrator named C3, Inc. had the courage to propose IWS/CTOS networks in response to a U. S. Coast Guard request for proposal (RFP) that had been written with minicomputers and terminals in mind. Convergent people went through a minor baptism of fire as C3 taught them the art of the government live test demonstration. The Coast Guard was amazed and delighted with the CTOS system, which could do more than they had asked for at a fraction of the cost they expected. It was a deal, and a big one.

The Coast Guard not only wanted the distributed word processor that was under development, but also needed spreadsheets, data bases, and other tools. At Convergent, the end-user orientation now gained strength with the acquisition of this real customer.

Problems with the cluster were quickly worked out, with another CTOS release (3.0) in March of 1981. More large customer prospects began to appear as Convergent went to trade shows and demonstrated its wares. A famous product demonstration of this era was the so-called "kick-the-plug demo," in which, while showing off the new word processor, the salesperson would "accidentally" trip over the IWS power cord, jerking it out and causing the machine to stop dead. As viewers gasped at this seeming disaster, the salesperson would smile and plug in the cord; the system would automatically reboot itself; and when entered, the word processor would replay every keystroke that had been typed up to the point of the interruption. Nothing had been lost. (One visiting engineer who saw this act at the Comdex show actually decided on that basis that he wanted to find a job working on CTOS systems. He was later to join Convergent and participate in a turning point in the life of CTOS.)

Another story from this era concerns the relationship of Convergent and Microsoft® Corporation. Microsoft was developing MS-DOS and associated tools; it was also known as a producer of compilers and office applications. Convergent needed a BASIC compiler but had little ready cash. A trade was worked out in which Microsoft received the source code for Convergent's linker in exchange for some BASIC licenses. This linker, minus its application-swapping technology, which Microsoft engineers deleted, became the linker that MS-DOS developers received from Microsoft. Thus, developers all over the world used a Convergent-written product in writing programs to run under MS-DOS.

Why the Accident Was a Happy One

In 1981 and 1982, Convergent became more and more firmly planted as an OEM supplier of networked office systems that were beyond the state of the art. Convergent's top managers realized that a young company of its type could not hope to do everything well, and so they did not attempt to establish a direct sales force or enter the retail market. Instead they chose to rely upon OEM customers for their marketing force.

Contracts, small and large, were signed; a lower-cost cluster workstation for the office, the AWS,™ was designed and shipped. (AWS seems to have stood for "Advanced Workstation.") Convergent OEMs began selling so many workstations in Europe that software revisions had to be made for native-language support. More application programmers and managers were hired. An unheard-of electronic mail program, based on the distributed capabilities of the operating system, was designed. CTOS itself went through several released versions as it was shaken down and features were added.

The CTOS workstations had arrived in the right place by accident. The accident was a happy one because CTOS was tremendously overdesigned for what most people considered to be office application needs in the early 1980s. (The less sophisticated IBM PC was forming the basis of their impressions.) Yet this excess sophistication and power have allowed CTOS to support unrestricted and easy development of distributed office applications well beyond the state of the art for a decade. Nothing had to be added: it was all built in from the start, with a different market in mind.

A Turning Point

Probably if the early pragmatists in the CTOS group had known that this would be its market, they would not have wanted to implement a message-based, networked architecture just to run word processors. They would have deemed it costly, complex, and wasteful. They did not know, and their work helped to create something that was not of their own world view.

At this time, the division of opinion that had started during the Xerox era was growing. Some customers were primarily developers using the CTOS machines as platforms for such things as telephone systems or interfaces to large typesetting systems. They were in the minority, however. By the middle of 1982, the differences of philosophy surfaced as the team began to deal with a major new idea: multiprogramming.

CTOS had been a multiprocessing system from the beginning, as the operating system, system services, and the single application that was running each had at least one process. The operating system scheduled these processes for execution according to an event-driven, priority-based mechanism to preserve real-time behavior.

Now a young staff engineer wrote an internal paper that proposed running several programs at once. Memory would be divided into several partitions; each program would have a partition. Applications would have the responsibility for staying out of each other's memory space and not doing rude things such as executing busy loops, and CTOS would extend its scheduling and memory management capabilities to handle this situation. To achieve this goal, an old religious tenet had to give way: it would not be appropriate, in most cases, for an application to interact directly with hardware. Etiquette now would require the application to go through CTOS for whatever it wanted.

Everyone agreed that this idea was a good one. The pragmatists, who were setting the primary direction for CTOS at this time, recognized the idea as the workstation equivalent of minicomputer and mainframe schemes in which the user interacted with one partition or context, while other programs could run in other partitions in background. The user's only access to these other partitions was via a batch processing scheme involving job control language (JCL) directives.

The multipartition concept was implemented in exactly this way in CTOS version 7.1 in mid-1982. In an elegant extension of the scheme, it was possible for the JCL programmer to write batch files so that jobs were enqueued to be processed wherever there was an available processor on the network. The user did not, and in fact could not, know on what workstation the job actually was run. The result was returned to the initiating workstation. The basic concept of transparent use of distributed processing power is being discussed as an innovation today. Yet its forerunner was up and running on a commercial system in 1982.

This multipartition scheme was ingenious and ahead of its time, as was so much of CTOS, but it was difficult to use. Certainly the average office user could not be expected to learn JCL. Most such users had trouble distinguishing between the concepts of memory and mass storage. A large contingent among the CTOS and related application developers believed that multipartition should definitely be part of the CTOS world, but that such a multipartition scheme should be entirely open to user interaction and have a new, easy user interface. Arguments began between the two factions over the future direction of their efforts.

None of those who were there seem to enjoy talking about this period. It was the first overtly expressed split in the group that had come so far and achieved so much together. In the end, those oriented toward the more technical, large-system atmosphere, together with the original CTOS architect, had the opportunity to go in a different direction by forming a new Convergent division to design a new, larger system. (Over several years, the division they started became Convergent's UNIX-related group.)

With the departure of these veterans, the influence of the founders' initial marketing aims for the workstation finally faded. There was no longer any question of where CTOS stood in the marketplace: it was a platform for distributed office applications.

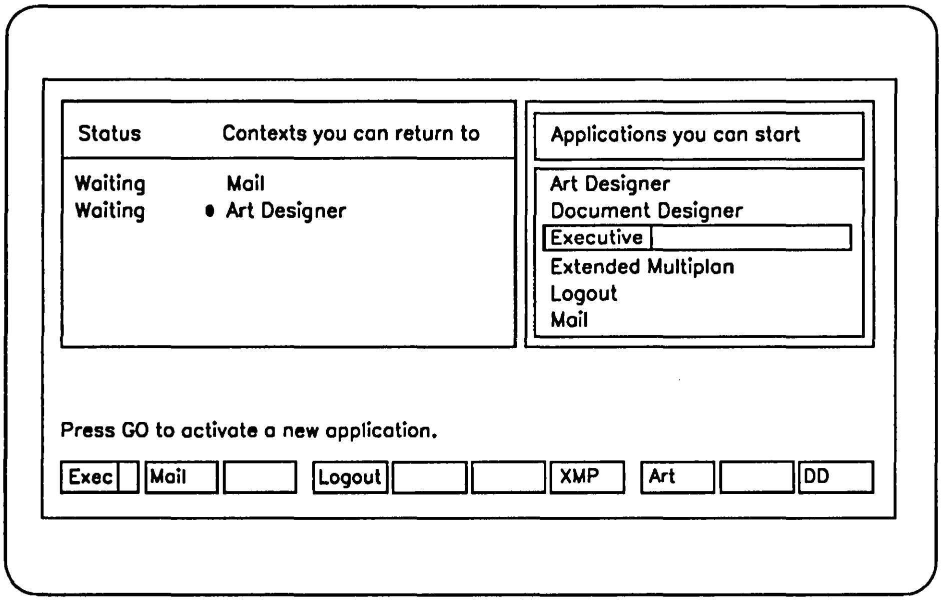
Multitasking for the End User

Those who remained to carry on the development of CTOS in 1983 were its second generation of stewards. (Most of the original members were still at Convergent, but had gone on to special projects of their own.) Most of the new CTOS group had been on the staff from near the beginning, so there was continuity in the culture. Again, they were from diverse backgrounds: two of the original crew from Xerox, one from Data General, one with a background in technical instruments, and so on. They shared a common vision of a multipartition CTOS that would allow any end user to run multiple programs at once, switching back and forth among them at will, interacting with each one directly.

Part of this vision came from a technical stunt pulled off by an engineer who was asked to come up with demonstration software to show to a prospective large customer. This company, whose own customers were largely stockbrokers, wanted users to be able to do word processing while a stock ticker simultaneously ran across the bottom of the video. Basing his demonstration software on the existing word processor, two other programs, and the background batch capability of CTOS 7.1, the engineer was able to show the prospect not just two but three programs on the screen at once. The customer was persuaded. (To this day, no one other than its inventor knows how this demonstration actually worked, because it was not thought by the others to be possible at that time.)

After a pause to rewrite the file system, whose creators had not imagined disks larger than 32 megabytes, the CTOS group went on to implement their vision of multipartition CTOS. The group leader and primary agitator for this viewpoint now created a team to implement a revolutionary new memory management system and user interface. One of the members of that team was the same young man who had been converted by the kick-the-plug demonstration at Comdex. Others were user interface, operating system, and file system specialists.

Context Manager,™ as this new interface was called, was designed so that its use would be intuitively clear to end users (Figure 3-1). On one side of the screen was a list of applications that could be started. On the other was a list of applications that were already running and to which the user could return. The applications could generally continue processing whether they were on the screen or not. A bar cursor could be moved from item to item in these lists. Simple keystroke commands allowed the user to switch contexts directly without opening or closing applications. The computer was going to conform to the user's way of working, not the converse.



Status	Contexts you can return to
Waiting	Mail
Waiting	● Art Designer

Applications you can start

Art Designer
Document Designer
Executive
Extended Multiplan
Logout
Mail

Press GO to activate a new application.

Figure 3-1. Context Manager Interface

The implementation of Context Manager had a clean architecture and little impact on application programmers. CTOS and Context Manager entirely handled context switches: applications did not need to know that they were running in a multipartition environment. In fact, most applications written for single-partition CTOS ran without change under the new scheme.

The emphasis on application politeness now grew. Not only were busy loops taboo: writing directly to hardware would now be cause for CTOS to stop an application when it was not the owner of the video. Few applications were seriously affected by these rules, because most had been well brought up. Fast communications programs that interacted with hardware were affected. These, however, ran perfectly while they owned the video.

All this was implemented in 1983 on a computer that still was based on the Intel 8086 microprocessor, with no special hardware support for saving context states or for memory protection. In the growing PC marketplace, nothing of the sort was yet imagined.

Horizons and Realities: The 80186 and 80286 Chips

While the interactive multipartition CTOS and Context Manager were being developed during 1982 and 1983, another activity was getting started also. Intel was getting ready to release the 80186 microprocessor (which did not have many new features) and simultaneously was publishing specifications for the 80286 chip that was supposed to follow on immediately.

The 8086-based IWS and AWS™ were doing well, paying the bills for further research and development. It was time to design the next-generation workstation, appropriately code-named NGEN®. After a brief tussle with the idea of moving to the Motorola 68000, it was decided that the new machine would be based on the new Intel chips. The 80186 version would be an interim design, but the important version would be based on the 80286. The new machine would have a modular, latch-together hardware design with external, modular power supplies. It would allow the customer to buy only the hardware functionality that was needed, and also to add functionality later.

In the summer of 1983, while most system developers were working on IWS/AWS CTOS and Context Manager, three others holed up in a conference room in a borrowed, empty building to work on the question of what operating system would run on the new machine. Their commission was to focus on the 80286 chip, which would have important new features.

In the Intel 8086 segmented addressing scheme, a 16-bit segment address was left-shifted by four bits and added to a 16-bit offset. The resultant 20-bit address represented a physical memory location. One megabyte (2^{20}) of address space could be handled in this way. Everyone, including Intel, knew that 1Mb would soon be insufficient. (The industry had changed enormously in three years. CTOS developers remember early arguments about whether putting 256Kb on the AWS was overkill.)

In the 80286, the two 16-bit address components, now called selector and offset, would remain, but they would not be added together. Instead, the selector would essentially index into a table and be mapped to an actual memory base location for a 64Kb segment. The offset, as before, identified the address within the segment. This new approach allowed 16Mb of memory to be addressed. It also allowed the implementation of memory protection. The new scheme was called protected mode.

Porting real-address-mode CTOS to the 80286-based NGEN was not a foregone conclusion. An engineering contingent external to the CTOS group tried to show that UNIX should run as native on the NGEN. In a famous internal demonstration, one of the more wizardly Ctosians did some quick coding to show that a version of UNIX hosted on CTOS would in fact have much better performance than native UNIX. Like so many other spur-of-the-moment events, this one was to become a significant influence on the development of CTOS.

Back in their conference room, having turned aside the immediate UNIX issue, the three Ctosians embarked on what some later called the "Summer of Love," mainly for the violent arguments that apparently rattled the walls. At the more conservative end of the spectrum, one wanted a simple port of the existing CTOS that would preserve complete backward compatibility and be out in a relatively short time. At the other end, another wanted immediate implementation of forward-looking features that might take a while to achieve and would cause compatibility issues for IWS/AWS customers. The third, full of ideas in his own right, was also the one who could see both sides and keep everyone talking. In true Ctosian style, the three emerged at the end of the summer with a design on paper.

The best and most painfully laid plans are not always executed. While the three were debating, back in the NGEN design world things were not going so well. Several hitches occurred in attempts to get hardware prototypes running. At the same time, Intel was having great difficulty in meeting its shipping commitments for the 80186 chip, and it postponed the date for the 80286 chip considerably. Convergent had already sold the idea of NGEN to its customers. They, in turn, stopped buying IWS and AWS to wait for it. Start-up style pressures reappeared at Convergent.

Convergent came down out of the design clouds and reverted to its true pragmatic nature. The 80286 project was shelved. Everyone, including our three friends, was thrown into the breach to get the 80186 NGEN debugged and a straightforward port of real-mode CTOS (with Context Manager) up and running. Sleeping bags, pillows, and an unending stream of coffee appeared in the labs, and the job got done. A single multipartition CTOS 9.0 with Context Manager, with a new internal basis for eventual networking beyond one cluster, and with run-time switches allowing it to work on IWS, AWS, and the 80186 NGEN, was released at the very end of 1983. Everyone went home to the Christmas present most wanted: sleep.

The DISTRIX Experiment

NGEN and CTOS 9.0 and its next shakedown release, CTOS 9.1 (mid-1984), were to be the stable basis of the CTOS world for some time. Further minor releases supported specialized new hardware modules (such as streaming tape and the Voice Processor and a new, low-cost, diskless cluster workstation, the CWS.™ TeleCluster (which allowed connecting the cluster via existing twisted-pair building wiring) was invented by a lead Ctosian and an experienced hardware designer, who thereafter referred to themselves as the "Twisted Pair."

Once the crew had recovered from the push to get NGEN out, a new direction emerged within the group. UNIX, no longer just an academic operating system, was being mentioned by some customer prospects. The Ctosians already knew that a hosted UNIX would work better and faster on this hardware than a native one, and a native UNIX would not support the true distributed-system concept. A small group of engineers set off to create such a hosted system at the behest of a customer. The hosted UNIX was to be called DISTRIX™, for Distributed UNIX.

The project started out with more UNIX underpinnings than CTOS. Over the two years of the project, gradually more and more CTOS components replaced the UNIX ones, until only the surface layer was UNIX-like. Two of the developers had been involved in the Summer of Love design, and they drew on this design in creating a new CTOS underlying the UNIX exterior. The main new feature was a conversion of the CTOS 9.0 multipartition scheme, in which partition size was fixed, to a variable-partition mechanism in which the operating system could adjust the sizes of application partitions as needed. Thus, no memory went to waste.

After two years of effort, DISTRIX was released in 1986. Convergent, however, had not taken into account the religious nature of the UNIX community. To these people, DISTRIX was not UNIX because it was mixed with another operating system. It was small, fast, and distributed; but it was not the UNIX they had in their college labs, and it did not do well in the marketplace.

The DISTRIX experiment actually did something for the CTOS world, however. Almost immediately, a version of CTOS that used the variable partitions developed for DISTRIX was issued. Called CTOS II, this version ran on the 80286 chip, which was now finally available, and around which a new NGEN processor had been designed.

CTOS II was really only an intermediate way to get onto the 80286, because it was still a real-mode operating system. It did not exploit the new memory management and protection features of the chip, and memory above 1Mb could not be reached. Regrouping after the DISTRIX excursion, the CTOS designers began again to consider the move to full use of the 80286. Chip chasing, or being the first company to implement new Intel functionality and show it at the Comdex trade show, had become an unwritten part of the religion by now. Other companies had built computers around the 80286 chip (for example, IBM Corporation's PC/AT®), but none of them used its new features to address memory above 1Mb. Convergent could still be first.

First With the Most

As demand increased, Convergent and its OEM customers were writing larger and more ambitious applications. The need to exploit upper memory on the 80286 became pressing.

Efforts now divided into short-term and long-term solutions to the 80286 challenge. In the short term, a new add-on software product called the System Performance Accelerator (SPA) was brought out. SPA allowed caching of user-selected files in memory above the 1Mb limit. It was demonstrated with great success at Comdex in the fall 1985, and it was the first released product on the market to use upper memory.

The second, less flashy but more important short-term effort was the Protected Mode Operating System Server (PMOSS). A system running real-mode CTOS and the PMOSS system service could run its other system services above the 1Mb limit. This step was a great relief to those who were writing and installing more and more system services and larger applications and, as a result, running up against the one-megabyte memory limitation. It also allowed a transitional period for system service writers to begin porting their code to protected mode in parallel while the Ctosians developed a true protected-mode operating system.

Protected-mode CTOS, or CTOS/VM,™ which was the basis for the current CTOS versions, was released in 1987. It grew from CTOS® II, which had been derived from the CTOS underlying DISTRIX, which in turn had its origins in the Summer of Love discussions. Thus, the winding road toward full 80286 support, started in 1982, finally reached its goal.

CTOS/VM on the 80286 allowed full access to the 16Mb address space. It employed the chip's hardware task switch mechanism for rapid process switching. This factor also allowed processing of communications interrupts to be more prompt than on other systems. CTOS/VM used the chip's memory protection mechanism, but not its rings of protection. Protection was not nearly so important to CTOS customers as was the ability to reach upper memory.

It had become part of the religion over the years always to retain backward compatibility. CTOS customers and users should not be wrenched from one system to the next. Convergent hardware had always been especially reliable, and many an IWS was known to be quietly chugging along out in the world. (In fact, there were working IWSs in the Convergent CTOS development group as late as 1987.)

CTOS/VM could not be made to run on the IWS and AWS, designed so many years before. It could be made to support the running of real-mode programs, though. RMOS (real-mode operating system), as this feature was called, was not a separate product, but was (and is) the ability of CTOS/VM to run any older real-mode programs, whether written at Convergent or by other suppliers, without change.

Running real-mode programs along with protected-mode programs on the 80286 required a technical trick. The 80286 processor could switch from real to protected mode easily, but it could not switch back to real mode without a processor reset. Every 80286 developer, not only Convergent, was faced with this problem. Convergent's solution was internally referred to as the "software finger," because it mimicked the finger that pushed the reset button. The solution reset the processor through a special circuit and then skipped most of the boot sequence to get quickly from protected to real mode.

DOS Compatibility

Back in 1984, in the era when arguments about porting CTOS to the PC were common, one of the original Ctosians was assigned to a stopgap project: porting MS-DOS to run hosted on CTOS for a customer whose own customers wanted to run MS-DOS-based applications. This project was difficult, and the result was not entirely satisfactory: MS-DOS applications had an alarming tendency to push the operating system aside, take over the system, and do strange things with hardware. As a result, many MS-DOS programs could not run in this hosted mode. The hardware was simply not the same, and the operating system was multitasking.

A new MS-DOS-compatibility solution was put into place with the 80286 version of CTOS. This solution, the PC Emulator, was both a hardware and a software product. It included an expansion module, called the PC Emulator module, with its own 80186 processor, and two software entities: a BIOS in the module and a system service in the workstation that remapped and handled I/O.

This PC Emulator had much greater application compatibility than did the hosted version of MS-DOS. Perhaps 80 to 85 percent of DOS applications could now run normally. Exceptions were those programs, such as certain games that depended on writing directly to the video map (indirection caused slow performance), copy-protected programs, and certain communications programs that required specialized hardware. Nevertheless, the PC Emulator was real progress toward satisfying a certain group of users.

	1965-1969	1969	1973		1979	1980	1981	1982
Intel				8080	8086		80186	80286
Motorola					68000			
CTOS					Convergent Technologies founded	CTOS 1.0	CTOS 2.0, 3.0 (LAN) Distributed Word Processor	Multiprogramming CTOS (background)
MS/DOS					86-DOS (Seatle Computer Products)		MS-DOS 1.0	
OS/2								
UNIX	Multics research	UNIX for DEC PDP-7	UNIX in C, PDP-11		UNIX Time-Sharing System, 7th Ed.	Microsoft Xenix, 16-bit Berkeley funded for 4.1 BSD		UNIX System III
Apple				Apple II				

(continued)

Figure 3-2. CTOS Through Time

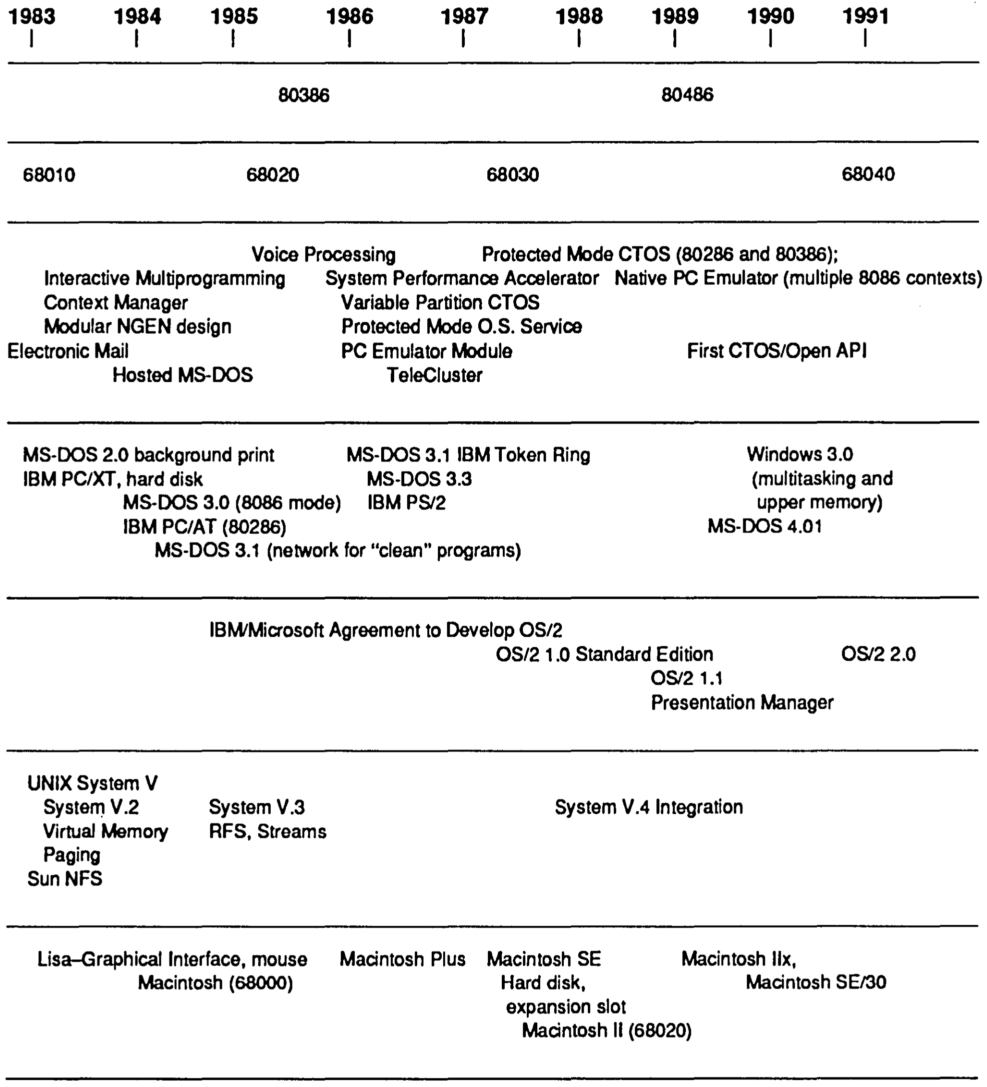


Figure 3-2. CTOS Through Time

First With the Most Again: The 80386 Chip

By 1986, Intel had brought out the long-awaited 80386 microprocessor. Not only did the 80386 allow easy protected-mode-to-real-mode switching in both directions, it also had a virtual 8086 mode in which the chip emulated its 8086 ancestor. In addition, it provided for the first time a large linear address space based on the 32-bit address. Although the segmented architecture (the two-part addresses composed of 16-bit selector and offset) could still be used for backward compatibility, it was now theoretically possible to use a single 32-bit address for the entire memory space (a maximum of 4Gb).

The 80386 capabilities freed designers from the mode-switch problems of the 80286, and CTOS/VM immediately took advantage of this change. More important was the ability to run a virtual 8086 machine. The long struggle for MS-DOS compatibility was largely resolved, because now PC Emulator software could run directly on the 80386. No separate compatibility hardware module was needed. Moreover, multiple virtual 8086 contexts could be run at the same time. Thus, a user not only could run MS-DOS directly on an 80386-based CTOS workstation but could run multiple instances of MS-DOS at one time, switching back and forth among them. With this capability, combined with their inherent networking and ability to link to mainframes, the CTOS workstations suddenly took a quantum leap over what was available for DOS users in the marketplace.

The Big Time

In 1988, while 80386-based development was going on, Convergent reached agreement with its largest OEM customer, Unisys® Corporation, on merging into one entity. (Unisys had a long history with Convergent products, first marketing the IWS and AWS as the B20 Series, and later the NGEN as the B25 Series of workstations.)

Convergent's days as a small company were over, yet contrary to what one might expect, its spirit and technical leadership entered a renaissance. The necessity to negotiate agreements about future design with several large OEMs had started to put a real cramp on Convergent's development style. Merging with one of those OEMs once again freed developers to follow a strong design direction, as well as giving them more resources to do so.

Unisys also offered something that Convergent had never been able to muster alone: extensive marketing and sales resources. It was now possible to think of CTOS as an operating system that could become more widely recognized.

CTOS/Open

By 1988, the marketplace had become consumed with the idea of standards. Ironically, the very systems that were understood to be standard through their wide market acceptance, UNIX and MS-DOS, were becoming technically outdated. Various enhancements to UNIX by different developers caused splintering and factionalism to enter the UNIX picture by 1989. Meanwhile, MS-DOS was waging a stubborn war of resistance against Microsoft's and IBM's newly announced OS/2®, which was supposed to replace it. In addition, Microsoft and IBM produced different versions of OS/2, and the need for custom device drivers for specific hardware complicated the picture.

CTOS had its own problems in the area of standardization. As part of its strategy of selling to OEMs, Convergent had licensed CTOS source code and marketing rights to a number of its OEM customers. These companies had not only developed some of their own hardware, but had also modified the operating system. They had also renamed CTOS as BTOS,™ StarSys, Hero/OS, TNOS, and so on, and had given new names to the workstation hardware as well. In the early 1980's protection of proprietary rights was the watchword; none of these vendors wanted to advertise that alternative solutions were available from other vendors.

One OEM, Bull®, understood the growing pressure for standards in the European marketplace and took the lead in beginning joint engineering projects with Convergent aimed at eventual standardization of the operating system. Unisys followed suit about a year later. In mid-1988, the three companies formed a committee aimed not only at the technical formation of a standard and common release, but also at joint promotion of this standard. This committee worked with other source licensees and some independent software vendors to develop a standard application program interface for CTOS, the CTOS/Open API.

The task of standardization was not as difficult as in the UNIX or other worlds. All the CTOS platforms were based on the Intel chips. The relatively few changes that had been made had utilized CTOS's basic modularity and were therefore simpler to deal with.

The CTOS/Open API for system services was announced at the first CTOS International Convention in Paris, France, in June 1989, sponsored by the Groupe du Standard CTOS, an independent group of end users, value-added resellers, and software companies who wanted to see CTOS established as a standard. The following year, Unisys and Bull announced at the convention that they would use CTOS/Open as the name for their major CTOS marketing initiatives. They also announced the formation of a standards organization: the CTOS/Open Advisory Council. Continuing the work of the earlier committee, this council in turn put forward additions to the CTOS/Open API, and opened its membership to the business community interested in pursuing the comprehensive standard environment demanded by users and application developers.

The people who had left Intel ten years earlier with an idea for the IWS were not there, but they would have enjoyed the show.

Religion and the Future

A technical religion, as we have seen, cannot afford to become rigid. Somehow it must preserve its central philosophy while adjusting itself to the present and the future.

As CTOS moves into its public phase, what has happened to the religion under which it was designed? These days, CTOS development goes forward under a series of eight stated principles. CTOS should be

- Open
- Modular
- Optimized
- Resilient
- Compatible
- Available from multiple sources
- Distributed
- Scalable

In this official list one can see some of the traits of the old, unofficial religion: modularity, distributed nature, small size, strength and speed, openness, and so on. CTOS still is and will be mostly message based. Real-time behavior is not on the list, but it is assumed to be important. Above all, the Ctosians continue to be pragmatists at heart, despite their love of the latest idea. After all the heated arguments are heard, the still-small band of CTOS developers will do the thing that will work best for their users.

A noteworthy event in the life of CTOS is the forthcoming release of a standard graphical user interface. For the first time, CTOS will have a user interface that is not home grown. This step is a response to the marketplace. Now that CTOS has gone public, it must more directly conform to what the public expects.

As resources and horizons for CTOS expand, many exciting ideas are gestating. Whether these ideas, or others not yet conceived, come to fruition in the next few years or not, some things about CTOS and its designers never change. There will always be new influences; there will always be controversy; there will always, in the end, be pragmatism. And there will always be someone impetuously sketching an idea on a paper napkin.

Part 2

CTOS and CTOS Applications

4 Thinking About Distributed Applications

What is a distributed application? It is an application that uses resources such as electronic mail or a data base over a network transparently. It is one that can be divided into pieces, not all of which must reside on the same computer to work together. Distributing an application is a good way to avoid replicating the same code on many machines . . . Through the use of system services and message passing, applications under CTOS have always been distributed, and there has been no need for a change or new approach in designing them.

CTOS is an interesting, unique, and powerful basis for application design. The particularly successful CTOS-based applications have been those that have taken advantage of its unique capabilities. Many of these applications have been designed for systems in which there is a central office or facility communicating with branch offices. They have supported such activities as airline reservation systems, banking, motor vehicle department operations, court reporting, and many government needs.

Before we go on to discuss how such applications are designed under CTOS, we should look a bit more closely at why distributed applications are important and what is needed to support them. Applications do not exist in a pure environment consisting of their creators' ideas about the next neat thing. They are computerized solutions to real problems in a real world. It helps to step back now and then and look at that world.

Distributed Applications

In the real world, nontechnical people are generally trying to do more faster with less all the time. Increasing productivity includes not only doing more things in a shorter time but also doing them at a lower cost. Most nontechnical people are not instinctively attracted to computers, but they have adopted them as necessary tools that free them from routine work and help them to be more creative. Computers also can make more information available and can automate complex tasks.

The acquisition, development, set-up, and maintenance of computer tools costs money: it thus costs money to save money. But we all want it to cost as little as possible. Development of application software should be especially simple and fast. Upgrading should occur at a reasonable cost.

Large Computers

Because they could increase productivity, large, centralized computers with dumb terminals had become a necessity and a fact of life in business by the late 1970s. As with all good solutions, however, large computers introduced new problems. Users had to deal with schedules and glass rooms, massive up and down time, and fluctuating performance. As the computers grew, their flexibility and responsiveness decreased. Users were serving or waiting for the computer a good part of the time. Productivity was not increasing.

Personal Computers

The advent of the microprocessor and the small computer based on it was a boon to users. They did not love computers any more than they had before, but they quickly adopted the desktop computer because it could help them maximize their individual productivity. They might not have tremendous processing power, but they had the tools when they wanted them. Small computers gave users flexibility and control. Once again, they gained time for individual creative work. They did not, however, have access to the work of others nor to information needed by all.

Personal computers sprang up everywhere, and with them came new problems. People who worked together started to use different types and versions of software. They had trouble with file system maintenance and backup. Users could not easily exchange data, and when they did, there were problems with version control. The cost of these problems to organizations increased, and overall productivity was threatened.

Issues in Creating Distributed Applications

The idea of the network and of distributing applications arrived in the larger business computing world as a compromise to keep everyone functioning and productive. The concept was that if all these small computers could be made to communicate with each other and with big computers, users could retain their creative independence and flexibility, while the organization would gain consistency, better exchange and version control, security, and so on. This approach turned out to be a compromise deal in more than one way. Networking and resource sharing had to be retrofitted onto systems that were never designed with them in mind. On PCs, it was necessary to add new dedicated server hardware as well as special boards and software. On multiuser systems running versions of the UNIX operating system, competing versions of complex add-on software were designed by different groups.

In both cases, add-on networking brought with it its own set of problems. Adding a network in the first place was expensive. Furthermore, because networks were not designed in from the beginning, all their administrative workings were exposed and required constant care by technically experienced system administrators. Both factors added to the cost of trying to reach higher productivity.

The result was an improvement over the previous situation, but even after all that work and expense, these networks readily supported only file transfer and certain types of resource sharing. They did not create a platform for true distributed applications.

What is a distributed application? It is an application that uses resources such as electronic mail or a data base over a network transparently. It is one that can be divided into pieces, not all of which must reside on the same computer to work together. Distributing an application is a good way to avoid replicating the same code on many machines. If some parts of an application can be shared, they can be thought of as services and can be placed on one networked computer. All the other members of the network need only run smaller client portions of the application. The user should not have to know which parts of the application are local and which are located across the network. Performance should not become unacceptable as a result of distributing an application.

PCs running MS-DOS, even with added networks, did not easily support the development of distributed applications. Such an application would need to know the topology of its environment: what computers were on the network, on what systems its other components could be found, and how to contact them. If one vendor's application software was to work with that of other vendors, a standardization issue ensued in which vendors had to make treaties (such as the Lotus®-Intel-Microsoft agreement for extended memory) so that their software products could work together.

Microsoft's OS/2, the follow-on to MS-DOS, addresses some distributed application issues by data exchange at the pipe level. An operating system in the early phases of growth, it still requires the network to be added on, with associated administrative burdens and security issues. In addition, replacement of older MS-DOS-based installations with new OS/2 software and associated hardware is costly.

In both the DOS and OS/2 worlds, making an application work across a network requires extensive knowledge of programming for the specific network involved. There are several popular networks. The result is that some programmers can make a very good living by specializing only in the intricacies of one or more networks. Networking an application adds time and cost to a development schedule.

Multiuser time-sharing systems based on UNIX have operated on an extensive electronic mail network for some time. This network, however, primarily supported only file transfer until Sun Microsystems's Network File System (NFS) and AT&T's Remote File System (RFS) appeared in 1987. These add-on features allow transparent file access across the network, but not remote procedure calling (RPC). Differing forms of RPC for UNIX exist, but they are not as fast as that of CTOS. Applications that use this kind of networking are written slightly differently from those that do not. Furthermore, UNIX systems have never been renowned for simplicity of setup and administration, and layers of networking software have not simplified these tasks.

In summary, most existing systems have approached networking and remote procedure calling as add-ons or upgrades, which bring with them expense and complexity. Productivity declines as inconvenience increases.

Distributing Applications Under CTOS

Because CTOS-based systems are message based and were originally overdesigned for the office-application marketplace, they have been able to support increasingly sophisticated distributed applications without encountering design limits. Through the use of system services and message passing, applications under CTOS have always been distributed, and there has been no need for a change or new approach in designing them.

In fact, it is not really possible to write an application under CTOS that is not inherently distributed in nature. Applications do not need to know network topology. They simply make requests, and the operating system takes care of knowing where the service is located and passing requests and responses. The message itself does not differ, whether the service is local or remote. The format of the request block itself enforces standard behavior by all CTOS-based applications. There is no need for external agreements in this area, no need for special network artists.

To design a distributed application under CTOS, one first identifies the separate tasks that will be performed by the application. The second step is to identify which task-performing components could be shared among users. These sharable components can then be written as system services. The balance of the program, usually the user interface, is written to run locally and make what appear to be procedure calls to the system service portions. That is all there is to it.

Because networking and security are inherent in CTOS, there are no add-ons, no special administrative needs, and no costs for these items. Writing a distributed application takes no more effort than writing any other application. There is no cost of writing a duplicate "network version" application. CTOS itself is in no immediate danger of approaching design limits, so there is no anticipated cost for large-scale replacement. All in all, CTOS systems provide an environment of high productivity for the developer, the administrator, and the user.

5
Timekeeper: A CTOS Application

Relate the application architecture to the system architecture, which includes the built-in network. Build your application model on the message-based, distributed CTOS model, the client-server architecture. For those who have worked on other kinds of systems, some rethinking is in order here.

Computers are used for business. CTOS, especially, is a solution for the modern business, where speedy and effective communication between wide-flung parts of a company can mean profits, whereas slow or ineffective communication means a loss.

Modern businesses need shared access to up-to-date data in a real time fashion. Consider the problems of an airline company, where numerous operators all over the world may answer requests for reservation information and then book those reservations, changing a centralized or distributed data base constantly. This can be done by tying the operations into the system with varying levels of distribution: a single, centralized data base on a monolithic computer used for all processing as well as data storage, a distributed data base on multiple centralized sites, or a combination of remote sites and centralized sites. The optimal situation for the end user (attempting to minimize data communications connections while maximizing response time) uses distributed processing, where some processing is done locally with a centralized or even a distributed data base which is only tapped to process transactions.

The problems of the airline company are not unique. The same situation occurs in the banking industry, the stock market, and even in keeping track of sporting events. The problem all these businesses share is that they have multiple operators who need to share data and are spread out over some geographic area. The answer to the problem is distributed processing.

Another problem they all share is the need for the operators to communicate with each other. One might not think this would be the computer's problem. It isn't. But those people are also tied to the computer. They use it to do much of their work. They would also be more effective if they could use it to communicate. The answer to this problem is also distributed processing.

A Look at Some Distributed Applications

CTOS message-based operation, with its integral cluster and effective networking, makes it an ideal platform for the development of distributed applications. To illustrate this in more detail, we are going to explain to you how a distributed application would be developed on CTOS. Along the way, we will pause when necessary to fill in background information about how CTOS works.

The application we will develop will be simple, so that we can complete our description of it in a reasonable number of pages. Before we get to it, however, let's look briefly at the features of two different, real-life, distributed applications that take full advantage of the inherent capabilities of CTOS.

A Unique Application: Reporting Sports Results

Your first thought when a sports tournament comes to mind is certainly not likely to be "how do they manage their computer system?" In fact, however, keeping track of and disseminating information at such an event is quite a job. Several events are usually going on at one time. Scheduled competitions change hourly. Results change with the minute. Competitors need information, the public wants to know, and sportscasters and tournament sponsors scramble to keep it all under control. This is a perfect opportunity for a distributed application.

In such an application, centralized data storage of raw data is essential. So is up-to-the-minute access to that data from multiple locations. Let's look at how one application system solves this problem.

The data base is set up using the ISAM system service and is located on one server workstation. That server can be accessed from cluster workstations in its own cluster and from workstations clustered to other server workstations in the network. PC workstations as well as CTOS workstations can be included in the clusters when a special cluster communications card and ClusterShare software are installed.

An interactive application can be located at each cluster workstation at which blow-by-blow results of the events can be recorded. The interactive application can do all processing locally that requires interaction with the operator and then can send the completed results to the ISAM system service at the server workstation for storage in the data base. The results are thus all stored in one central location.

Scheduling for events can be set up and stored in the data base as well. Again, the interaction with the operator can be done with local processing. The current schedule is stored centrally. From any cluster workstation running an interactive application, anyone interested in the events could find out about the latest schedule changes. Several clusters can be networked together.

Players can also call to find out when they are next scheduled to compete; they hear a digitized recording that tells them when and where they compete. The voice recording is all stored electronically on the computer and played back using the services of the Telephone Manager system service, a device driver for the CTOS Voice Processor module.

In this application, the local workstation actually sends a query regularly to find out what new events may have just been completed and then, using digitized voice processing, verbally announces new score changes or the results of events as they occur. On request, special statistics about results are calculated locally based on raw data retrieved from the data base. The interaction with the user and the calculations are done locally, the raw data is shared across the network. Several of these workstations are located at various locations at the site, or away from it.

In this application, the PCs have a special and interesting role. As mentioned above, they are tied into the network as cluster workstations using a ClusterCard communications card. An MS-DOS based application running on the PC uses ClusterShare software to connect to the CTOS server and to access data in the CTOS ISAM data base. The PC displays the scores and other results as an overlay on the television monitors that are showing events. In this case the PC was chosen because a specific output channel was needed to write to the television display.

This system has been highly effective, providing up-to-the minute information to many different sites. It is fast and responsive. It works the way the people who use it need it to work. It is distributed processing in action.

Interoffice Communication: CTOS Electronic Mail

Nothing could be more representative of the office environment than the memorandum, "memo" for short. And nothing probably wastes more trees or takes more time in the office environment. That's why electronic mail is such a success on all computer networks that offer it.

On CTOS, in an inherently networked environment, electronic mail is a natural. The CTOS electronic mail product is extremely successful and very widely used.

Mail is an interactive application that the user can use to read memos in his in-tray, reply to them, forward them, or file them as necessary. File folders that store messages can be opened and searched for relevant information. Memos created by electronic mail have a standard memo-style format. To send a message, the user simply types in the names of the people to whom he wants the message to be delivered, typing just Reggie Smith or George Robinson on the distribution list line. Remembering complicated addresses or code names is not required. From the local workstation the user can send such memos around the world. They can be sent to users on other networks, other systems, even to a FAX machine.

How does the system work? Through distributed processing, of course.

An interactive application resides on each user's workstation. On the server workstation are two system services, one of which sends and receives messages for each user on the cluster, temporarily filing those messages in a mailbox until the user requests to see his mail, at which point they are read into in the in-tray. This is called the Mail Service. The other service, called the Communications Manager, handles wider network communications, passing messages on to other nodes in the network (other clusters) or to WANs.

The Mail Service also provides the interactive application with the names of all the users known to it (and to the wider network Communications Manager).

Each part of the system performs part of the processing. Each is located at the place in the network where that processing can be performed most effectively and efficiently.

Timekeeper: Our Distributed Application

Keeping these two applications in mind, let's think about some other business solutions. Software should provide a solution to a productivity problem. A simple one to start with is the need to replace the business person's ubiquitous paper To-Do list with something that can go it one better and offer reminders as well. Embroidering on this theme, why not add an appointment calendar that also can issue reminders if the user wants them?

That would be a simple application to serve an individual user, that you can probably go out and buy for whatever computer you are using.

But what can you do with that application in the CTOS world? Suddenly we're talking about the workgroup, about all the other people that the user needs to interact with, and about the business environment. We can think about distributing resources easily. We can think about doing more than one task at once.

Suppose the calendars of all users in the group are centralized. The software can then allow users to check other users' calendars to see when they are free. Extending things a bit, this product can provide automatic calendar checking and meeting scheduling, even looking to see what conference rooms (with appropriate sizes and amenities) are available and choosing the best of them. Then the software can automatically send a message to all meeting participants with the meeting details, let the instigator know who is and is not coming, tell everyone about any changes that are made, and issue reminders to everyone.

Now we are really talking about a distributed application. Let's call this product Timekeeper.

A Framework for the Design

In these few paragraphs, we have already gone through the first few steps in specifying the requirements for a distributed system. In specifying the requirements, the utmost importance is that the system meet the needs of the end user. With our Timekeeper system, because it is simply an avenue for displaying the architectural structure of CTOS, we do not have to delve any deeper in specifying requirements.

Once the requirements have been fully specified, we need to address them with a functional description. We want to keep the process simple by replacing a manual system with a computerized system.

Once the requirements have been specified and we have a functional description of how the requirements are to be satisfied, we get to the design phase. The first question should always be "What are we trying to do here?" Is there an existing product that does all of what we want? Assuming that we want a networked solution, is it appropriate to the CTOS server/client model? If it does not make use of the network at all, if it is only going to serve one user, we probably should look into the DOS marketplace. CTOS is preeminently a platform for distributed applications.

The Design Environment

CTOS has a modular architecture based around a small kernel of primitive operations, to which additional modular system services can be added as needed, where each system service is performing a specific class of functions. With the division of functionality into separate services, we can identify certain characteristics.

- The operating system environment is highly modular, and the modules are independent in that they contain both data and methods.
- The operating system environment incorporates the accepted principle of data hiding within these modules.
- The operating system environment is characterized by data abstraction: the internals of a module can be changed without affecting the business that the module carries on with other modules.
- Components communicate with each other by messages asking for results, but not by telling each other how to carry out the work involved.
- Reuse, rather than reinvention, is emphasized within the operating system environment.

Steps Toward a Design

Keeping in mind the distributed, modular basis of the operating system, we now need to translate our previous functional specification into a design. The real nature of designing applications, no matter what the underlying system, is holistic. Textbooks tout the advantages of top-down design because of the decrease in the cost of integrating various system components; however, top-down design coupled with successive refinement is the primary mode of design within the industry. Because the CTOS operating system environment embraces modular techniques with encapsulation of functionality within the modular components, a top-down, step-wise design process allows for rapid prototyping with minimal functionality, followed by staged phases with ever-increasing functionality.

Starting Simply

The main thing under CTOS is to start with a simple, general version. If you can get this simple version working well, you can always add processes and features in later versions. If, however, you start out with an ornate design involving multiple processes and much functional filigree, you (or your successors?) may end up on permanent debugging detail . . . if, in fact, the application ever works. As one developer says, "Don't get too beautiful with your ideas."

Starting with a simple version is not necessarily simple in itself. It requires that you think ahead about the application's future. What extensions will you need to add later? Leave room and hooks for them. Anticipate that your present "early" version will some day have to handle extended inputs and status codes sent by later versions of the same product running on the same network. In other words, set yourself up early to maintain backward compatibility.

This principle applies to specific areas as well as to the overall design. Start with fewer, more general "umbrella" data formats. Design fewer requests and allow for added subcases within them. Allow in the beginning for requests to be routed by file specification over the network. (Requests are discussed in Chapter 7.) In the user interface, start simply and leave room to build. Reorganized user interfaces in later versions are very difficult for users to accept and learn.

Conversely, as you move to later versions, implement new features in terms of your original model rather than adding more and more specialized code.

In the case of Timekeeper, the specific application that we're focusing on, we should think about the possibility that eventually it could be used organization-wide rather than just in a small workgroup. We are not going to design this capability into the first version, but we need to leave the possibility open.

Relating the Design to the Network

One veteran Ctosian says, "You can't *not* design a networked application. So you musn't write an application without considering the network." Relate the application architecture to the system architecture, which includes the built-in network. Build your application model on the message-based, distributed CTOS model, the client-server architecture.

Although we shall talk about modularity more specifically a little further on, at this point we are already thinking about Timekeeper in network terms. Parts of Timekeeper will be centralized: the data about everyone's calendar, the meeting rooms, the handling of mail messages between users. Other parts will be specific and local to each end user.

User Inputs and Outputs

Now we arrive at what, in standalone systems, is the first step in application design. What does the user expect to put into the application? What will the user get back from it? This basic user I/O is the reason for the application's existence. Note that we are not yet actually designing the user interface, but only identifying what will pass through it.

With Timekeeper, users will input data destined for their own To-Do lists and calendars. They will also input attempts to schedule meetings. They will write notes to each other and input them to Timekeeper's mail message facility.

A Timekeeper administrative user will input all possible meeting rooms and their sizes and amenities, and will maintain this list from time to time. This administrator will also input and maintain the list of users. Users will make some administrative inputs themselves, such as configuring the range of times shown on their calendar days.

In return, Timekeeper will display various views of To-Do lists and Calendars on demand. It will provide desired reminder outputs when duties and appointments fall due. It will respond to users about their attempts to schedule meetings. It will send notifying messages to meeting participants. It may need to notify the administrator of inconsistencies or difficulties it encounters.

Needed Functions

What functions does Timekeeper need to perform in order to use these inputs to provide the desired outputs?

Most obviously, it needs a user interface to receive those inputs and return those outputs.

It needs to store all the data about users' calendars and To-Do lists. The calendar data, at least, should be accessible to all users for inspection and meeting scheduling: the implication is that Timekeeper will need networked data storage.

Finally, it will need a way of checking the current time, comparing it to appointment times on people's calendars and lists, and reminding them of upcoming events or deadlines. Let's call this function the Reminder. You can probably guess that Reminder is going to be a system service.

Division of Labor

Having thought about what the whole application system has to do, we are now at the point of thinking more literally about the division of labor among its parts. There are two issues: client/server division of labor and division of programs into possible multiple processes. Of these two, we are going to take up the first at a simple level here, with more complex issues to come later. We shall defer the issue of multiple processes until after we have discussed processes and interprocess communication further in Chapter 6.

What parts of the application are of value to more than one user in the workgroup? What resources will be shared among users? These are the questions that lead us to define the contents of system services, if any, in an application system. In the case of Timekeeper, we have already seen that calendars will need to be a centralized data resource. A system service often arises as the manager where multiple clients will be making requests for such a shared resource. (This is especially true where the resource is a physical one such as a piece of hardware for data storage or transmission. Prefetching data, control of user contention, and like duties are ideal for a system service.)

We have already suspected that we will need a Reminder system service. Its duties will be to control and update the shared data, compare current time to scheduled events, and return reminder notification to the clients.

What parts of Timekeeper are user specific? Certainly user I/O is not shared. An interactive application program for I/O will be written that will run locally to each user. Instances of this interactive application will be the clients of the Reminder system service, making requests to it to update the centralized data and to remind them of upcoming events.

Where does the To-Do list data go? For simplicity, it goes into the centralized data storage resource along with the calendar data. Although we are not going to have users reading each others' To Do lists, the Reminder service will be handling notification from these lists, so centralization makes sense. It also allows us to use only one data format.

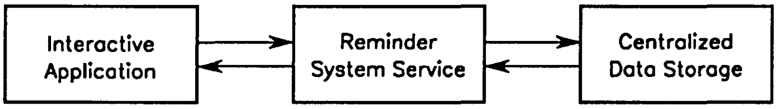


Figure 5-1. Timekeeper's Component Parts

We had also planned to transmit short, electronic text messages among users. The user interface aspect of this facility could be part of the interactive application portion of Timekeeper. What about the handling, routing, and delivery of messages? That part sounds like another system service. (We shall get back to this problem.)

One criterion to use in thinking about division of labor between clients and system services is whether a given functionality must be always "on," always up and running on the system, or whether you can afford to have it go away with the user. The parts that must always be available go into the system services.

At this point the designer should also think about what, if anything, should be placed in libraries and what in system services. In general, system services should handle the event-driven, real-time aspects of processing, while library routines can be written to handle computation. (An example is CTOS Mouse software, in which the real-time Mouse Service reports mouse movements and clicks to the client, while the Mouse library routines handle conversion of this information to whatever coordinate system is being used.) Bear in mind that later revision of a library requires that the application be relinked with the new library and reissued, whereas updates to a system service require reissuing only the system service.

One should be willing to make compromises with the principles outlined above if performance issues intervene. A classic software design trade-off is that of size versus performance. Putting code into a system service makes the application smaller overall. But creating a very pure design in which the system service does everything that is centralized in a very clean way can sometimes have a performance impact (for example, where one character at a time is being fetched and transmitted over the network).

Thoughtful design of user interfaces and modularity in the beginning can save untold grief in later versions. It is worth hashing out all the issues and problems you can think up at this stage.

Using Existing Pieces

In keeping with the modular nature of CTOS, we now look around to see what needed bits of software already exist. We do not want to reinvent too many wheels.

For our centralized data storage, we could write our own networked data base (a lot of work), or we could use the ISAM (Indexed Sequential Access Method) package, an existing CTOS networked data base. If we needed a more elaborate data base we might consider using Oracle®.

For the Reminder system service, actually, we could just use the existing CTOS Queue Manager, which has the ability to check the system time and dispatch messages accordingly. Using the Queue Manager probably is the ideal thing to do, in pure Ctosian terms. However, if we decided to do that, the rest of this book would become decidedly boring, as we would never illustrate the writing of a system service. Therefore, we shall write the Reminder Service from scratch.

As for the delivery of electronic messages, we are going to do the right thing. The CTOS Electronic Mail Service has all the functionality we need for transmitting messages. There is no need for us to create another system service here. We can just write the interactive application to call the Mail Service API.

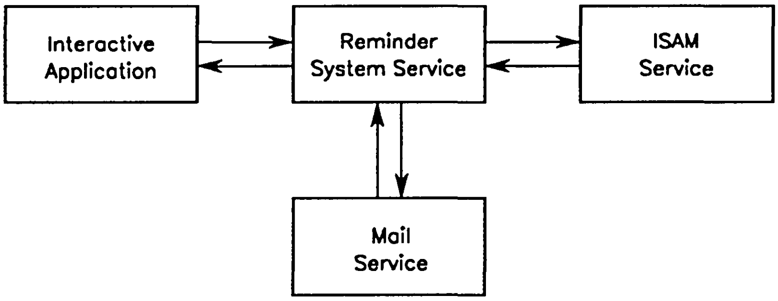


Figure 5-2. Timekeeper Components and Other CTOS Services

Understanding Some Underpinnings

Before we can refine our design and start coding away, we need to discuss two additional conceptual areas. One is the way the CTOS message-based system does its work. The other is the variety of available I/O tools under CTOS.

Chapter 6, therefore, examines messages and CTOS Interprocess Communication (IPC), the underlying architectural concepts which allow processes to communicate, and explains how the request/response model is built on IPC.

Chapter 7 discusses system services, the entities which provide services to multiple clients utilizing the IPC mechanism of Chapter 6, and how they are able to work over the network transparently to their clients.

In Chapters 8 through 11, we look at I/O, ranging from the highest to the lowest level tools, and from disk to wide-area communications.

In Chapter 12 we return with our new found-expertise to choose from this smorgasbord a real-world set of tools with which to write Timekeeper.

6

More About Messages

CTOS has one principal means of interprocess communication (IPC): the mechanism of messages and exchanges. Messages are the basis for consistency and standardization of application behavior under CTOS. Clients and system services, all communicating through the same mechanism, can therefore be changed, substituted, and reassembled in different ways without requiring disruption and recoding of their communication methods.

There are two basic approaches to interprocess communication within any multiprocessing operating system environment: shared memory and message passing. Since CTOS is a networked operating system, shared memory is not possible; hence, messages make CTOS what it is. Its real-time nature, its system services, its transparent networking are built with and on the basic concept of messages.

Messages are the means of communication between processes in both CTOS and applications. A greater level of concurrent processing is possible when a multiprocessing system is message based. Most CTOS processes use messaging to synchronize themselves with the operating system, system services, and/or other processes. Because of this, at any given time in a CTOS system, many different processes may be ready to run, although only one is actually running at a time.

Actually, three kinds of entities compete for the processor under CTOS: processes, device interrupt handlers, and trap handlers. Of these, processes are the most important to us here. To understand how processes communicate with each other, though, we must first describe the process itself.

CTOS Processes

The concept of a process has been described many times and in many ways, but none of the definitions truly convey what one is. It is easiest to say that a process is a running program; but under CTOS a program may have more than one process. It is more accurate to say that a CTOS process is an independent thread of execution, along with the hardware context: that is, the contents of the processor registers that are necessary to that process. A process has a stack (which contains its history) and a current execution point. A process should not be confused with the code that it is executing: the same code can be executed by several processes at the same time.

The context of a process consists of all the information required by the processor to perform work on behalf of that process. This context includes both hardware and software components.

The hardware context consists of the values to be loaded into the processor registers when the process is scheduled for execution: for example, the CS:IP (the code instruction to be executed) and the data segment (DS) and stack segment (SS), which reference the location of the process's data and the process's stack.

The software context consists of the default response exchange (where the process has its messages sent), the priority at which the process is scheduled for execution, and the interrupt vectors pointing to the software interrupt routines that the process uses.

CTOS processes do not own resources: rather, a process is a resource owned by a higher-level entity (the operating system or the application system of which it is a part).

Process Creation

Any process within CTOS is started up in the same way. CreateProcess, a kernel primitive, creates a new process and schedules it for execution. CreateProcess is called by any of several higher routines that may be used when an application is first loaded, either from an application or from the Executive (the CTOS command line interpreter). An application that is already executing and needs to establish a second, independent process also uses CreateProcess.

For example, when the Executive application program is loaded into memory, the initial Executive process is created. This process is the main routine by which the user interfaces to the system. After the Executive is loaded, it in turn makes a CreateProcess call to start the second Executive process, the time process. This independent thread of execution periodically updates and displays the time on the user's screen. Both processes are part of the same application program, but have different threads of execution for more effective use of system resources.

Each process is known to the operating system by its Process Control Block. The Process Control Block is a system structure that contains information about the process, including its execution state, priority, default response exchange (to be discussed later), user number (a reference to its owner), and context. The context of a process is made up of all the information necessary to resume the execution of the process. This information includes the settings of the microprocessor registers and other hardware registers.

Processes under CTOS have associated priorities ranging from a most-favored priority of 0 to a least-favored one of 255. (One might create a process of priority 255 to, for example, determine how busy the CPU is.) Processes are scheduled to run on the basis of these priorities; hence, CTOS is also priority driven.

Process States

A process can be in one of three execution states: ready, running, or waiting. A process is ready when it is competing for the processor; that is, it could use the processor if it were available. All the ready processes are linked in priority order in a queue called the run queue. The process at the head of the run queue (the one that has the most-favored priority of all the processes in the ready state) is in the running state and is known as the running process. A process is said to be waiting when it needs to receive a message before it can resume processing (waiting for an event); hence, CTOS is also event driven.

Figure 6-1 shows a system that includes six processes. Processes A, B, C, and D are ready. Processes E and F are waiting. Process A has the most favored priority of the ready processes, so it is the running process.

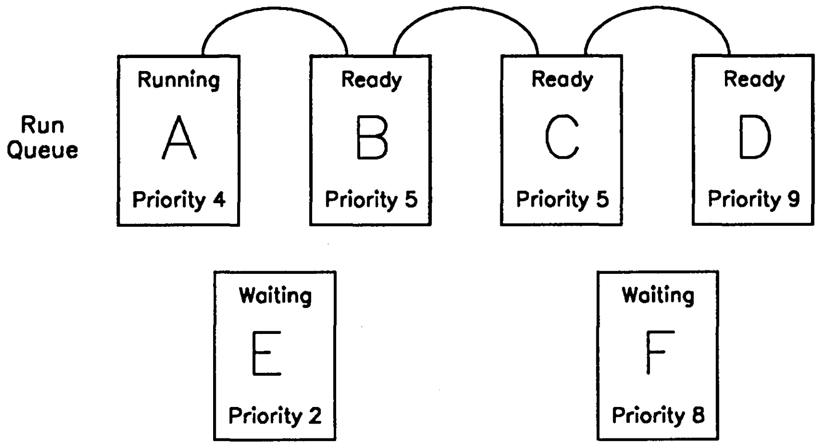


Figure 6-1. A System Including Six Processes

A process can also be suspended. A process can be suspended by a user command or by the system debugger, or when it is swapped out of memory, or pending its termination. A suspended process can be ready or waiting. When a process is both ready and suspended, it does not compete for the processor and does not become the running process, even if it would normally have been at the head of the run queue.

The Scheduling Algorithm

A process moves from the running to the waiting state when it needs to receive a message to be able to continue executing, but that message is not yet available. When the process enters the waiting state, it is taken out of the run queue, and the next process in the queue becomes the running process. In Figure 6-2, Process A is now waiting, and Process B has become the running process.

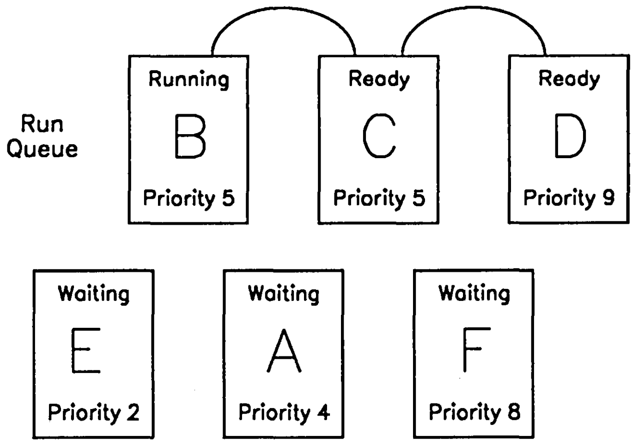


Figure 6-2. Process A Enters the Waiting State

When a waiting process receives a message, it enters the ready state and is inserted in the run queue, behind all processes of the same or more-favored priority and before any process of less-favored priority. In Figure 6-3, Process F has received a message and is inserted into the run queue between Processes C and D.

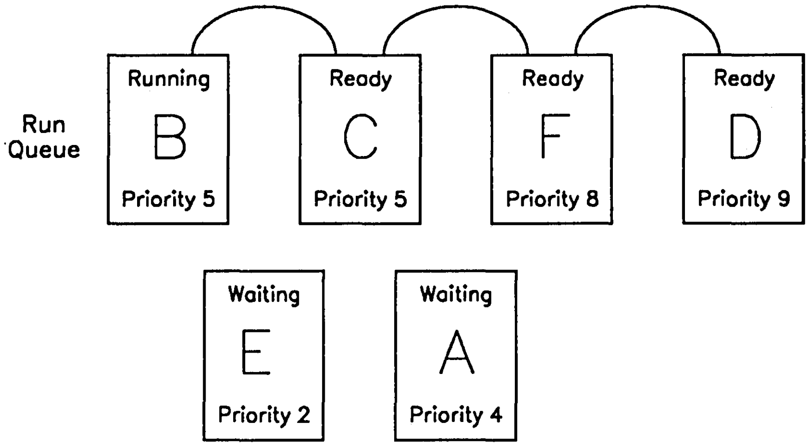


Figure 6-3. Process F Enters the Ready State

When a process receiving a message has a more-favored priority than that of the running process, this receiving process is inserted at the head of the run queue and is made the running process.

As a consequence of the CTOS scheduling algorithm, a process cannot be preempted by another process of the same or lower priority without waiting for a message. This enables the enforcement of mutual exclusion in a single processor environment between processes having the same priority and sharing common variables without having to resort to using semaphores as is done in UNIX or OS/2.

Process priorities range between 0 and 255, but are divided into groups as follows:

0	to	9	Operating system
10	to	64	System services
65	to	254	Application programs
255			Null process

The operating system is given the most favored execution priority to ensure that its work is performed as promptly as possible. System services, being logical extensions to the operating system, have the next most-favored priority level. The null process, the process with the least-favored priority, is executed only when no other process is available to run.

A program can change its default priority with the ChangePriority call, but care must be exercised in assigning priorities. Modifying one's process priority may ensure that the process has control of the processor, but at the expense of preventing even the operating system from functioning.

For application processes, CTOS uses time-slicing to ensure that no single process can prevent others of the same priority from getting the processor. The running process is moved to the end of the list of ready processes of the same priority every 100 ms. This rule applies to processes with priorities in the range 146 to 178 on CTOS workstations.

Time-slicing is available only within the specified range of priorities and not within the range of priorities used by system processes, because the scheduling algorithm mentioned above would be violated. High priority processes performing system level work will never be preempted by a lower priority process.

Interrupt and Trap Handlers

Remember that at the beginning of this chapter we pointed out that there are three different entities that compete for the processor: processes, interrupt handlers, and trap handlers.

Interrupt and trap handlers are software entities that have been declared to the operating system as having to be executed when a given event occurs. Interrupt handlers are triggered asynchronously by hardware events. Trap handlers are initiated by software action (the execution of an INT instruction or a fault, such as division by zero). Trap handlers are also called software interrupt handlers.

The Intel family of microprocessors support vectored interrupts. A vectored interrupt is uniquely identified by an interrupt vector which is put on the hardware data bus in response to an interrupt request by a peripheral device.

Interrupts invoked via the Trap Gate and Interrupt Gate are executed within the current process's environment, including its stack, without an automatic context switch. Interrupts invoked via a Task Gate, though, result in an automatic context switch to the stack of the Interrupt Service Routine task.

An interrupt or trap halts the sequential execution of the currently executing process. The current hardware context is saved, and control is then passed to the interrupt or trap handler. Once the condition causing the interrupt or trap is resolved, the interrupted process's context is restored and its execution is resumed, or a process of more favored priority is executed via a context switch.

Interrupt handlers are usually written as part of a device-handling program. Device handlers perform the hardware I/O to and from an external device. The handler consists of a device handler process that manages the device and initiates I/Os, and a device interrupt handler that is executed when operations are completed or status conditions change at the device.

Even though the two entities are executed asynchronously, they are parts of the same program within CTOS. Communication and syncrhonization are accomplished by using kernel primitives (e.g., PSend) and optionally shared memory for buffer utilization and control information.

The device interrupt handler is executed when the external interrupt occurs. It calls PSend to to start the execution of the device handler process, which has been waiting at its exchange for some work to do. The kernel primitive is the only way to synchronize the interrupt handler and the device handler process. Synchronization is unidirectional only, from the interrupt handler to the device handler process, even though data can flow in either direction with shared memory.

Interrupt handlers are not commonly written by applications programmers. They are primarily of interest to systems and communications programmers and those who need to handle devices. As such, they are beyond our scope here. Most CTOS vendors offer documentation that explains these subjects in detail.

Processes in Other Kinds of Systems

Operating systems such as UNIX and OS/2 also manage multiple processes or threads. UNIX processes can spawn child processes, creating a hierarchy of processes in the system. CTOS processes do not do so.

When a UNIX child process is created, the entire parent process is copied, and both continue to run. The child process must then chain, or the copies will contain duplicate code and data. Memory cannot be shared, since each process has its own user space.

The Sun Microsystems implementation of UNIX includes what are called lightweight processes, which are multiple threads in the same address space. These lightweight processes are similar to the threads of OS/2. CTOS processes have more in common with these lightweight processes or threads than with the classic UNIX processes, in that multiple CTOS processes can exist within a program.

CTOS Programs

So far in this chapter we have looked at processes as distinct entities. We shall return to examine the way in which they communicate via messages. First, however, we shall take a short detour to look at programs, which may contain more than one process, and at memory management. This discussion will equip us with some concepts that we shall need in talking about interprocess communication.

It is impossible to separate CTOS programs from the way they exist in memory. Briefly, memory is divided into logical entities called partitions. A memory partition is not necessarily a contiguous memory area, but it is logically treated as such in these discussions for convenience. Each application has a separate user number, sometimes called a partition handle, identifying the partition in which it resides. We shall return to the details of partitions shortly.

Programs are what an application writer constructs. They are the executable entities that are run by users to do work. An executable program consists of code, data, and one or more processes in memory. The steps in creation of a program are shown in Figure 6-4.

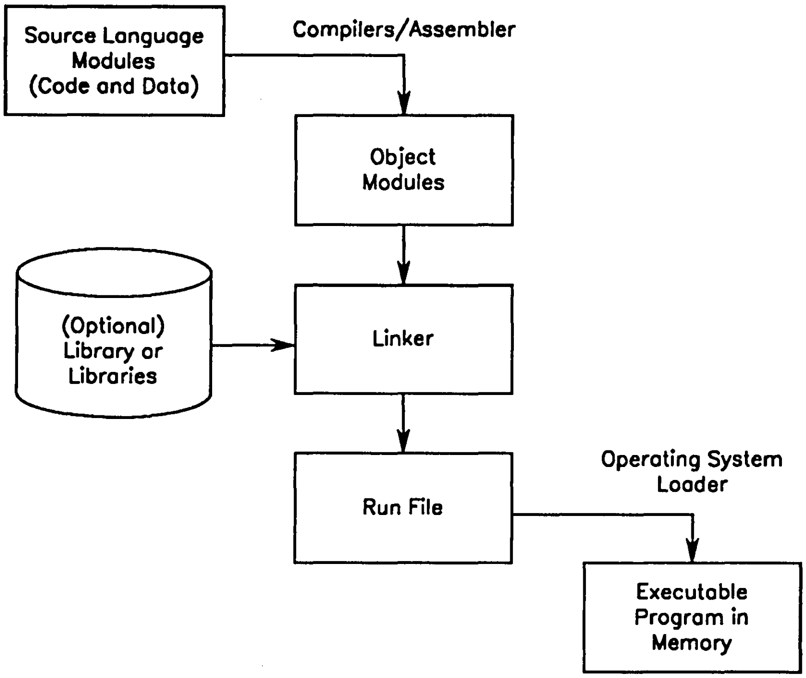

Source Language Modules (Code and Data)
Compilers/Assembler
Object Modules
(Optional) Library or Libraries
Linker
Run File
Operating System Loader
Executable Program in Memory

Figure 6-4. Steps in the Creation of a Program

As in most other systems, the executable file, called a run file, is linked from one or more object modules that were compiled or assembled from source code written in any of several languages.

Commonly, many frequently used object modules are placed into an object module library. Prime examples of this method are the CTOS development libraries, CTOS.lib and CTOSToolKit.lib. These object module procedures are an important adjunct to CTOS and contain the routines that support a major I/O methodology, among other things. (See Chapter 8 for details.)

Once a run file is created, it can be loaded in any of several ways involving related CTOS primitives, such as LoadTask, LoadInteractiveTask, and so on. An application on an end user's system is most commonly loaded, not directly by the user, but by a Chain primitive issued from the Executive command-line interpreter or by a LoadInteractiveTask call from the Context Manager (which is similar in function to the OS/2 Session Manager).

The Chain simply replaces the currently executing application within an application partition with the new application. The Executive, for example, is said to Chain to the application that the user invoked through its command line. A Chain verifies that a given run file can be loaded into the application partition. If there is not enough memory, the Chain fails, and control returns to the caller, along with an error code.

Partitions

Multiprogramming under CTOS is supported with the division of memory into areas called partitions. Before the Intel 80286 microprocessor provided descriptor tables that allowed memory segments to be described in a virtual way, CTOS memory partitions were real contiguous entities. Application programmers were entirely responsible for managing application partition memory and seeing that they did not overwrite each other's territories or that of the operating system. The complexity of managing memory is one reason MS-DOS does not provide multiprocessing capabilities.

Today, a CTOS "partition" is really a logical partition only, and pieces of it may be scattered all over physical memory. CTOS uses the protected mode feature of the 80286 and more advanced microprocessors in the same family to assume responsibility for territorial boundaries. Partitions are still represented in simple drawings as contiguous, but it must be remembered that they really are not.

Memory can be viewed as two separate types: system partitions and application partitions.

System partitions contain the operating system code and extensions to the operating system, the system services. Recall that CTOS itself contains several different processes: the Keyboard Process, the Resource Manager, the Scheduler, the Termination Process, and so on. All the processes that are started during the initialization of the operating system are in a single system partition. Each system service loaded after the initialization of the operating system is contained within its own partition.

Application partitions are created through a partition-managing program (for example, Context Manager) or via a Chain from the operating system after CTOS has completed its initialization sequence. This last method of creating an application partition is the normal method used for the Executive. The organization of application partition memory is shown in Figure 6-5.

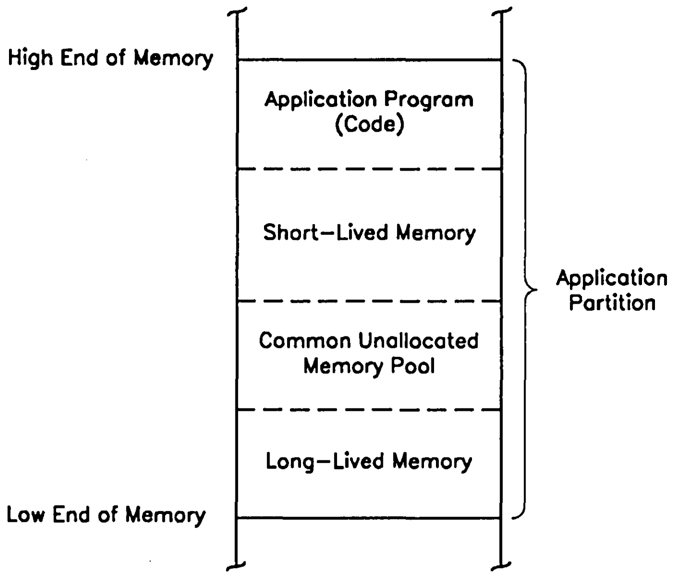


Figure 6-5. Memory Organization of an Application Partition

An application partition can be either fixed or variable. A fixed partition always uses a fixed amount of memory, whereas a variable partition can grow with a program's needs. The operating system Loader determines whether a partition is a fixed or a variable one during the loading of a program. If a partition is to be variable, the application must be sized during the binding (linking) of the program. This information is present in the header of the executable file (on CTOS called a run file). It specifies the maximum and minimum amounts of memory required to load the program. If the minimum is available at load time, the program is loaded. If more memory becomes available, the application can grow to the maximum size specified in the run file header.

Associated with each partition is a number called the user number (historically also called the partition handle). This number is a 16-bit integer that uniquely defines the partition and all the resources associated with the partition. It is worth noting carefully that the user number is owned not by the application in the partition, but by the partition itself. This fact becomes important when dealing with system services and their clients. Resources include file handles, short- and long-lived memory, and exchanges.

Associated with a memory partition are the application code, short-lived memory, long-lived memory, a pool of unallocated memory, and the Local Descriptor Table, or LDT (a table used by the hardware for addressing memory segments on the 80286 and subsequent processors). The application code may or may not be present: if multiple copies of the same program are executing at the same time, only one copy of the program code is present in memory within one partition. Usually, all other partitions in which the same program is loaded are sharing the code of the first program that was loaded.

It is also possible (although not often done) to load more than one program into the same partition.

When a program is initially loaded into memory within an application partition, the code is loaded at what would traditionally have been the high-address end of that memory partition. After the code has been loaded, along with a structure known as the U-Structure (containing all the structures needed by the operating system to manipulate an interactive partition), the remaining memory is then divided into three different sections: short-lived memory at the top of the partition, long-lived at the bottom, and a common pool of unallocated memory in between. (Again, this scheme no longer bears any relation to the physical position of pieces of the partition in memory.)

CTOS makes a distinction between user and system partitions. A system partition is simply an extension of the operating system and is created when a ConvertToSys operation is requested. U-Structures and long-lived memory structures are not associated with a system partition.

Memory

Short-lived memory contains all the code and static data segments of an application program. Additional short-lived memory can be allocated and expanded by the program.

At the bottom of the partition is long-lived memory, which must be allocated by the program if needed.

Short-lived memory grows from the top down; long-lived memory, from the bottom up. The area between these two is the unallocated memory pool.

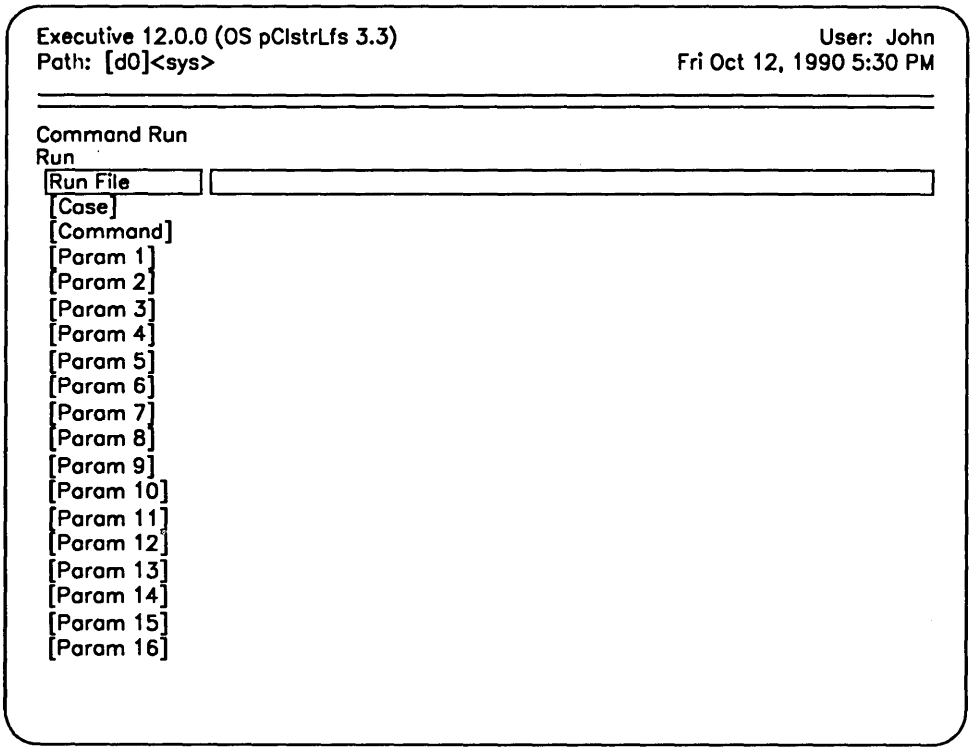
Memory can be deallocated or returned to the common pool of memory, with the caveat that segments must be deallocated in a sequence exactly opposite the order in which they were allocated (preventing fragmentation).

The terms short-lived and long-lived are associated with the contents of the memory areas. Short-lived memory does not survive a Chain from one application program to another within the partition, whereas long-lived memory does. When an application program chains to another program, the new application is loaded into the high area of the partition overwriting the previous short lived memory. Because long-lived memory does survive a Chain, though, parameters or information can be passed from one application to the next via this mechanism.

For example, the Executive command-line interpreter provides a Run command that is often used by developers to execute programs that are under development and do not yet have defined invocation commands. The Executive provides a simple forms interface for the user to fill in, so that no option symbols need be remembered.

The Executive uses long-lived memory to pass parameters to a program executed through the Run command; remember that, long-lived memory survives a chain. Parameters entered by the user within the Run command form are passed to the succeeding program via a data structure called the Variable Length Parameter Block, which resides in long-lived memory.

(The Case and Command fields in this form are not pertinent to the present discussion.)



```
Executive 12.0.0 (OS pClstrLfs 3.3)                     User: John
Path: [d0]<sys>                                Fri Oct 12, 1990 5:30 PM

Command Run
Run
 Run File    [                                                    ]
 [Case]
 [Command]
 [Param 1]
 [Param 2]
 [Param 3]
 [Param 4]
 [Param 5]
 [Param 6]
 [Param 7]
 [Param 8]
 [Param 9]
 [Param 10]
 [Param 11]
 [Param 12]
 [Param 13]
 [Param 14]
 [Param 15]
 [Param 16]
```

Figure 6-6. The Executive Screen and the Run Command Form

The advantage of this differentiation of memory is that it does enable the simple passage of parameters between successive programs. However, there is a price to this mechanism. All memory must be deallocated in exactly the opposite order of allocation. If this order is not followed, memory within an application partition can become checkerboarded, and the application can run out of memory.

Interprocess Communication

Other Environments

UNIX and OS/2, which in large measure seems to have copied UNIX in this regard, both have several mechanisms for communication between processes. These include such things as anonymous or named pipes, sockets, queues, and shared memory. The latter requires the use of semaphores to ensure integrity. In the history of UNIX, each of these mechanisms was invented by different people for different purposes. The result, in both systems, is little consistency.

By contrast, CTOS has one principal means of interprocess communication (IPC): the mechanism of messages and exchanges. Messages are the basis for consistency and standardization of application behavior under CTOS. Clients and system services, all communicating through the same mechanism, can be pulled apart, substituted, and reassembled in different ways without disrupting or recoding the communication methods.

CTOS IPC: Messages and Exchanges

Messages

A CTOS message is the packet of information that can be passed from one process to another, or from an interrupt handler to a process. A message contains 32 bits of data. Often, a message is a pointer to a larger piece of information. When using IPC directly, it is the programmer's responsibility to make sure that the receiver of the message is in the same address space as the sender (and thus can access the data), and that the sender does not destroy the data before it has been received, or process it before it has been generated. Practically, these rules mean that direct IPC is used between processes that are part of the same application.

Protected mode operation on the Intel microprocessors uses indirect addressing through descriptor tables. An application addresses items identified by entries within its local descriptor table (LDT). The LDT contains selectors referencing memory within the application's partition, whereas the global descriptor table (GDT) contains selectors referencing global memory. Direct IPC messages are accessed via the application's LDT. Hence, an address to a data element must be an address within the program's local descriptor table; an address referencing data within another application's LDT would result in a protection fault.

Note that when the request/response mechanism, the next higher level of IPC, is used, the programmer does not have to be concerned about addressability, because the operating system ensures that requests are routed to the right places.

The Exchange

An exchange is the focal point of IPC. Messages are sent to exchanges, not directly to processes. Conversely, a process specifies the exchange at which it expects a message. The exchange is a mailbox in which messages are deposited and from which messages are taken.

When an application is loaded, it automatically is given one exchange: the default response exchange, used in the next higher level of IPC, which we shall discuss later. An application program can find out the identity of its default response exchange by using the QueryDefaultRespExch procedure.

Programs can also dynamically allocate and deallocate other exchanges to be used for direct IPC by using the AllocExch and DeallocExch primitives. An application should not use its default response exchange for communicating with another process.

An exchange has two queues associated with it: a queue of messages and a queue of processes. They are managed as follows:

When a message is sent (using the Send primitive) to an exchange where no process is waiting, the message is appended at the end of the queue of messages. The queue of messages is a nonprioritized FIFO (first-in, first-out) queue.

Alternatively, when a process demands a message (by issuing the Wait primitive) from an exchange where no message is available, that process is put at the end of the queue of processes (regardless of its priority), and it enters the waiting state. This queue is likewise a FIFO queue: the first process in the queue will receive the next message.

A process can alternatively use the Check primitive to examine an exchange to see whether a message is present, but to continue processing if it is not.

When a message is sent to an exchange where at least one process is already waiting, the message is immediately delivered to the first process in the queue. That process is made ready and is inserted into the run queue. If the receiving process has a priority higher than that of the sending (currently running) process, it becomes the running process.

When a process asks (by Wait or by Check) for a message from an exchange where a message is already queued, that process is handed the message and remains in the running state.

Note that if the Wait chronologically precedes the Send, the system behaves differently from the way it would if the Send preceded the Wait. In the first case, the receiving process may lose the processor to another process of the same priority. In the second case, this change would not occur.

The implication of the exchange algorithm is that at any given time at least one of the two queues in the exchange is empty.

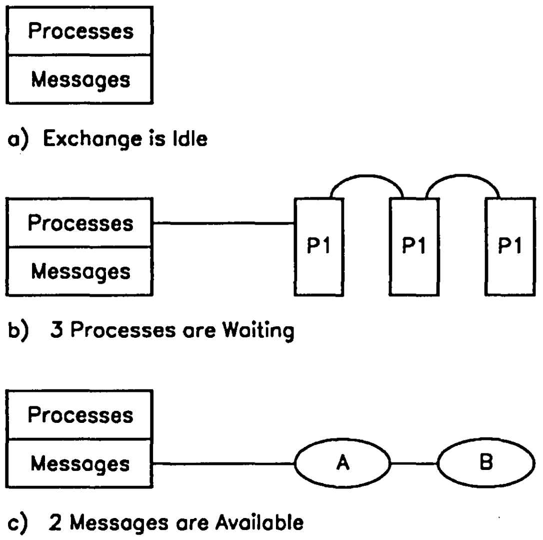


Figure 6-7. Three States of an Exchange

Figure 6-7 shows three states for an exchange. In Figure 6-7(a), we see an idle exchange: no process is waiting and no message is available. Figure 6-7(b) shows three processes waiting in the process queue. P1 will receive the very next message sent to the exchange; P2, the following one; and P3, the third one, regardless of the relative priorities of P1, P2, and P3. In Figure 6-7(c), two messages are available. Message A is the oldest; B is the more recent. The next process to issue a Check or Wait primitive at that exchange will receive message A. The following process will receive message B.

CTOS exchanges are resources. They can be allocated and deallocated at will.

IPC Primitives

At the most basic level, CTOS IPC is carried out through the issuance of three kernel primitives. Send sends a 32-bit message (usually a pointer) to an identified exchange. Wait receives a message from an exchange. If no message is present, the process waits (blocks) until one arrives. Check receives a message from an exchange, but returns immediately with an error code if none is available. Check never causes rescheduling of the processor.

We mentioned in Chapter 1 that the message and exchange system could be used for purposes other than passing data. Indeed, Send and Wait can be used with dummy values, simply for synchronization. They can also be used in resource management where, for example, one process controls the resource and others ask to use it. This usage is a form of semaphore control, and we shall look more closely at it in Chapter 13.

The IPC primitives are used only between processes belonging to the same application (or more formally, running in the same application partition). Messages can be passed between applications in different partitions by another mechanism called the Intercontext Message Service, but this method is used only for special purposes that we shall not address here. The more common way to pass messages between entities that are not in the same partition is the use of requests and responses.

Hiding the Mechanics: The Request/Response Model

In order to communicate using the IPC mechanism, two processes need to have knowledge of the exchange to be used. This requirement can be tolerated when the two processes are part of the same program, but it is very constraining when communication is between an application program and the operating system. This area has been an area of extreme complication in distributed processing. How is the target process named clearly?

To allow a program to ask other programs to perform functions on its behalf, CTOS introduces the notion of requests. A request is a formal way for a process to ask for a service to be performed by another process. A process requests a service by using the Request primitive. This primitive accepts the request block, the self-describing structure containing all the information necessary to pass information between the service and the client.

Upon completion of the service, the other process must formally respond to the request by using the Respond primitive.

For now, we shall call the process that issues the request the client and the process that receives the request the service. Later, we shall extend these names to the application systems that own the processes. However, context should make it clear which one we are talking about.

We shall cover the routing of requests in detail in Chapter 7. Basically, the client process issues a Request primitive to the operating system, passing a pointer to the request block, which contains information about the desired work to be done. The operating system determines what service is serving that type of request and passes the request to that service. Similarly, the response is passed back from service to client via the operating system. Request and Respond are additional primitives that enhance the IPC mechanisms we described previously and that allow formalized messaging between applications not in the same partition.

The Request Block

The vehicle used to carry requests and responses back and forth is called a request block. Figure 6-8 shows the request block in outline. The request block header (which, in turn, is detailed in Figure 6-9) is a fixed-format structure that contains general information applicable to any request block. (During the early development of CTOS, it was this header portion that was added when the request block was redesigned to allow it to be used across the network.)

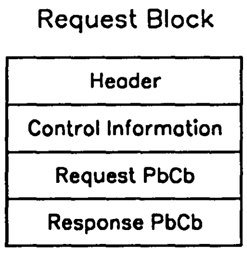


Figure 6-8. General Form of the Request Block

The control information portion of the request block contains parameters transferred from the client to the service. This area is used for short data types, such as characters, integers, or doublewords passed by value; for example, a file handle or a screen coordinate would usually be passed as control information. Pointers should not be included in the control information. Their proper place is in the following portion of the request block, the PbCb pairs.

As shown in Figure 6-8, a PbCb pair is a CTOS type made up of a pointer to an array of bytes (Pb) and a count of bytes (Cb). In other words, a PbCb is an array descriptor. Request PbCb pairs reference data arrays transmitted from the client to the system service. Response PbCb pairs reference data arrays transmitted back from the system service to the client.

Should particular data be passed from client to system service as control information or via request PbCb pairs? The decision is based on the fact that a PbCb has 6 bytes of overhead, whereas control information should be limited to 16 bytes. The entire request block must occupy less than 64 bytes. The way in which the procedural interface for a request is defined also influences this choice.

Response PbCb pairs are the only way (other than the error code) to return data from the system service to a client. A service must not modify the request block (except for the error code returned field), nor the data pointed to by request PbCb pairs; nor should it assume any initial value for the areas pointed to by response PbCb pairs. It is sometimes necessary for a system service to temporarily modify the request block. In such a case, the system service must ensure that any alterations are restored before responding to the request.

The client must not access (read or write) the request block nor any data pointed to by any PbCb pair from the time it issues the Request primitive until it has received back the pointer to the request block through a Wait or Check primitive. Additionally, once the response is received, the data pointed to by response pointers can be invalid if the returned error code is not zero (indicating satisfactory completion).

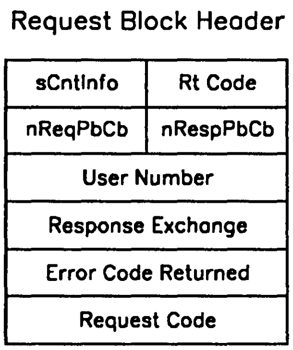


Figure 6-9. Details of the Request Block Header

Figure 6-9 details the structure of the request block header. The sCntInfo field indicates the size (in bytes) of the control information. The routing code is reserved for use by the operating system kernel and should be set to 0 by the client. (Doing so avoids receipt of a no-such-request error code where a remote request is not locally defined.) The nReqPbCb field indicates the number of request PbCb pairs in the request block. Similarly, the nRespPbCb field indicates the number of response PbCb pairs.

The user number denotes the owner of the request block. This field is normally set to 0 by the client and set to the correct value by the kernel. This field is used by a system service to identify the owner of the resources it controls so that it can dispose of them properly in case of the termination of the client. The response exchange shows where the response to the request is to be sent. It is an exchange that must have been previously allocated by the client.

The error code returned is a 16-bit quantity used to convey to the client the success or the reason for the failure of the operation. As a convention, 0 is reserved for successful completion, while any other number indicates a failure. Finally, the request code field indicates what function the issuer of the request block wants to have performed. CTOS uses this field to deliver the request block to the proper service exchange.

When a process submits a request block (using the Request primitive), the operating system makes the request block and the data pointed to by the PbCb pairs available to the system service. After the work is completed, CTOS makes the response data available to the client. These steps can be done because of the structure of the request block and of the knowledge that it gives to the operating system of the pointers and the flow of information. This process is called aliasing. A global descriptor table entry is created for each request and response PbCb pair, allowing the service access to the request data and the client access to the response data. Hiding pointers in the control information field or inside a structure pointed to by a PbCb pair would defeat this mechanism; in fact, an application in which this was done would fail. A protection fault would result because the aliasing would not have been performed, and the required data could not be addressed.

Synchronous and Asynchronous Processing

A process that sends off a request for a service can proceed either synchronously (waiting or blocking as soon as the request has been sent) or asynchronously (continuing execution and checking its response exchange periodically to see whether the service is complete).

Asynchronous processing requires the programmer to build the request block literally: that is, filling out all the required parameters in the request block, and then issuing the Request primitive to pass it to the operating system.

Most application requests, however, do not need to be asynchronous. Where processing is synchronous, the application programmer has a much easier job. The Request primitive is hidden under what is called a request procedural interface. If, for example, the application needs to open a file, it merely makes the OpenFile procedure call, passing the required parameters to the file system, rather than constructing a request block for issuance to the file system.

The application programmer does not even need to know that this call will actually become a request transmitted by CTOS to the file system service. When this call is made, the operating system takes the passed parameters and uses them to build the request block on the client's stack. As part of this process, the operating system retrieves the default response exchange value from the process control block of the issuing program and puts it in the request block header. It then issues the Request on behalf of the client.

When the service issues the response, the operating system reads the request block header to find out what the response exchange is. It extracts the returned error code from the request block and hands it back to the caller/client in register AX.

The caller is never the wiser about this whole process. The existence of the application programming interface (API) made up partially of request procedural interfaces makes the job of application programming very similar to what it is under other systems.

Part of the work of writing a system service is defining the requests and the associated request procedural interfaces that will make up the API for that system service. We shall see more about that in Chapters 12 and 13.

Requests Versus RPC

Much has been written in the last several years of the importance of the Remote Procedure Call (RPC). This concept is actually what system programmers desire when writing an application system: a method for calling a procedure to perform a service irrespective of the location of the service routine. A truly distributed system requires this type of functionality. With an RPC, the calling process waits for the receipt of the message, and when the message is received, continues processing.

The CTOS Request/Response implementation of IPC provides all the functionality of an RPC with one major enhancement. Because there is a formalized manner for describing messages, indirect references to data are possible. The operating system aliases indirect references on behalf of the client and/or service, allowing access to data within different address spaces. Additionally, the operating system, along with the networking software (agents, which we will cover in the next chapter), routes the data referenced by the PbCb pairs so that the client can function irrespective of the location of the service and vice-versa.

A Few Other Mechanisms

Not all of the CTOS API is made up of hidden requests to services. All CTOS procedural interfaces have the same form and are handled in the same way by applications programmers, but some go to other destinations.

Kernel Primitives

CTOS has a limited number of kernel primitives: we have encountered some of them (Send, Wait, and so on) in our discussion of IPC. The remaining ones concern creating processes and manipulating them and their priorities; handling requests and responses; and manipulating interrupt handlers.

System Common

In addition, some system routines that are frequently used and are always used on the local workstation, or that require very high performance, are defined as system common procedures. System common routines are accessed in a manner similar to a UNIX kernel entry (however, they are not actually part of the CTOS kernel): their entry points are accessed directly, unlike those of system services. A system common procedure is executed by the calling process rather than by another one, utilizing a feature of the Intel microprocessor architecture called a Call Gate. Thus, the system common code must reside on the same CPU with the caller. System common routines are not network routable. They are synchronous and must be reentrant. (That is, the system common routine must be able to be suspended, executed by another process, and then later completed within the original scope, transparently to either executing process.) System common routines have no global data. All data must be stack relative to ensure its viability if the routine is suspended.

Since the calling process and the system common are not in the same address space, it is the system common's responsibility to have any pointers made addressable to prevent a protection fault.

CTOS has two varieties of system common routines. The first type is built into the operating system at system generation. The second is the loadable type, which can be loaded at will by an application program. Functionally, the two types are not distinct: they behave identically. The loadable type allows the further customization of an operational environment.

Object Module Procedures

Libraries of object module procedures also exist. Code for these procedures is bound into applications themselves at link time. In general, object module procedures contain functionality that is more specialized than that in system common procedures and would not be used by all programs on a system. They tend to be computational rather than resource-related functions.

CTOS has three standard libraries: Ctos.lib, CtosToolKit.lib, and Enls.lib. Commonly used routines, development aides, and nationalization routines are defined in these libraries.

Additional libraries are provided for many application packages. Programmers can define new libraries by using the Librarian utility. This utility accepts object modules and places them within a user-defined library file.

We shall look at these other mechanisms in more detail in Chapter 8.

7
System Services

When a client issues a request, it does not need to know where the system service is located. The system service may be local (on the same CPU), or it may be at the server workstation of the cluster, or even at a workstation across a CTOS Network. If the system service is not local, the request is transparently routed across the network to the system service.

The previous chapter described many of the architectural concepts underlying the CTOS messaging capabilities. Processes communicate with each other, so processes, memory management, and messaging were covered in great detail.

Recall, though, that we stated that CTOS itself is not a single process; instead there are several components to the operating system, each of which is also a process. Recall also that we stated that operating system functionality could be replaced and/or enhanced with user-written system services.

Now that you understand the underpinnings of CTOS message-based operation, we can begin to look at the way a system service actually goes about its work. In previous chapters, we have talked generally about them. Now we arrive at the point of discussing them in more detail. This chapter describes the interaction between the system service and its clients. It also takes a look at filter processes, a specialized type of system service.

System Service/Client Interaction

Remember that the request block is used to pass messages from one process to another. Generally, in this interaction one process is the client, requesting a service; the other responds by performing the service. The responding service is called the server.

In Chapter 6 we looked at the request block format. Three fields from that request block header play especially important roles in the client/server interaction. These are the user number field and the request code field. The request code identifies the service that is desired. The user number helps to characterize the client. The file handle can be used to uniquely identify a specific series of transactions associated with a specific user number.

Partitions and User Numbers

In Chapter 6 we also talked about the partition. An application system executes inside a memory partition. Note, however, that an application system and a partition are not synonymous. Although there is only one application system per partition at a given time, application systems can succeed each other in the same partition. When this happens, the succeeding application inherits the application partition.

Each partition is identified by a 16-bit user number. It is this user number that appears in the request block header. Whoever reads this field can tell what partition originated the request.

Request Codes

The request code in the request block header is a number that uniquely identifies the service that is desired (not who is going to perform the service). The request code must be unique to ensure that conflicts do not occur. The request code field is used by CTOS to deliver the request block in the proper service exchange.

The Dynamics of Requests and Responses

CTOS maintains an internal table that maps each known request code to the exchange at which it is served. This table is initialized at boot time and can be changed dynamically. When a new system service is installed, it allocates one or more exchanges at which it wants to receive requests for services. Then it informs the operating system that it plans to serve certain requests at certain exchanges.

To do so, the system service makes the ServeRq call once per request code, giving the request code to be served and the system service's exchange. The service then waits at its exchange for requests to come in. (See Figure 7-1.)

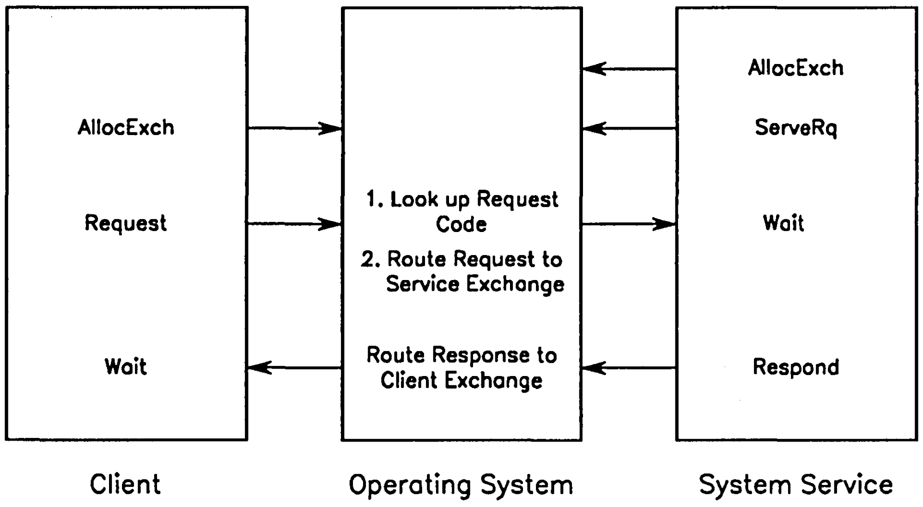


Figure 7-1. The Request-Response Model

When a client makes a request, either directly or through the request procedural interface, the operating system receives a pointer to the request block. The operating system extracts the request code from the request block, looks up in its table the exchange at which that particular request is served, and enqueues the request block at that exchange.

Pointer Aliases

Data pointed to by the PbCb pairs within the request block must be addressable by both the client and the server process. Because the client application and the system service that serves the request are in different partitions, their memory is governed by two separate local descriptor tables. Thus, the system service cannot immediately access the data that the client wants to pass.

This problem is a general one for operating systems on the 80286 and later Intel microprocessors that want to pass information from sender to recipient. Protected-mode operation relies on the local descriptor tables to keep memory for each application distinct. A global descriptor table keeps track of the correlation between the local descriptors and the physical address, but is not generally accessible to applications.

Under CTOS, the solution to the problem that occurs when the client and the service need to pass data to each other is to create alias pointers in the processor's global descriptor table (GDT) for the pointers in the sender's LDT that indicate the location of the data. Once addresses exist in the GDT, they are available to any process in the system, whereas the LDT values are restricted to their owner. Thus, the system service can use the alias pointers in the GDT to access the passed data.

It is interesting that although OS/2, designed to run on the same microprocessors, encounters the same issue in the implementation of dynamic link libraries, it resolves the problem differently. Under OS/2, certain positions in the LDT of each process (the same positions for every process) are reserved so that they can be used to refer to the same shared memory if necessary.

Each of these two approaches has its disadvantage: the CTOS method could conceivably fill up and exhaust the GDT if many processes are running, whereas the OS/2 approach uses up half or so of each LDT for shared memory descriptors.

Reaction of the System Service

Up to this point, the system service has been a process in the waiting state, waiting for a message so that it can become ready to run. When the message (which is the address of the request block) arrives at its exchange, the system service enters the ready state and in fact has a good chance of becoming the running process, since its priority has been set higher than that of the client.

(Since system services can and do become clients of other system services, you may be wondering what happens if the client in fact has a higher priority. The answer is twofold. First, system services are usually waiting for work to do, and while they are in a wait state, any other process at the same priority that is available to run can be scheduled to run. Second, if necessary, processes can change their priorities.)

The system service now does whatever is necessary to serve the request. If the function succeeds, the system service inserts the value 0 into the error code field of the request block. If it fails, the system service inserts a value that represents the cause of that failure. The system service then calls Respond, passing back the pointer to the request block. The system service can now Wait again at its exchange (if it is a synchronous system service; we will say more on this subject later).

The operating system picks out the response exchange value from the request block and thus knows where to enqueue it for the client. (The operating system does not use the user number at this point.) If the call to the service was originally made through the request procedural interface, the operating system extracts the returned error code from the request block, adds it to the AX register, and returns to the client. If the request was made directly by the client using Request, the client receives back the pointer to the request block from the system service and is responsible for extracting information directly.

Connections

Up till now we have a talked about the interaction between a client and server process in terms of only a single transaction. However, CTOS provides the capability for grouping transactions into a series of interactions between a client and the system service that acts as the server.

Where a transaction is part of a series of interactions, the client and system service are said to have a connection; where the transaction is a one-time-only event, the relationship is said to be connectionless. A client may have several connections simultaneously to the same or different system services. These connections are all independent.

In setting up the connection, the client identifies the service it wants. The system service, if the necessary resource is available, allows opening of the connection and hands back to the client in the request block a reference number to be used in further requests within that connection. This reference number is usually called a handle.

A connection is first established, then used over a series of requests to do something (such as read a sequential file), and then finally destroyed. In such a case, the system service must remember not only what file is being read, but also who the client is.

The operating system and network software, as well as the client and system service, use the handle, although they may map it to different values in order to keep it unique. Each connection or handle is associated with a given user. This allow's multiple connections between a client and a system service. If the user number was used rather than the handle, only a single connection would have been possible.

Two types of system events can interfere with a connection: termination and swapping. These require careful handling by both the operating system and the system service that serves a request. When an application is terminated a connection becomes no longer valid. In this case, the server process must scan though all outstanding requests from this client (which it identifies by the user number) and remove them from its queue. The situation is more complicated when the operating system has swapped the client out of its partition. In this case, the system service serving the request must respond, but it must queue up its responses until it receives notification that the client has been swapped back into the partition. Otherwise the response could go to the wrong client and cause a protection fault that would cause a system crash. The operating system and the server process use the user number to identify the partition in which this change has taken place so that the system service can correctly handle such pending requests.

Crossing the Network

Remember our discussion of the need for a formalized mechanism for an RPC? An application system programmer would like to develop an application which is oblivious to the fact that the application is either in a network or on a standalone machine. The CTOS Request/Response model and the Request Procedural Interface supply just this type of mechanism.

When a client issues a request, it does not need to know where the system service is located. The system service may be local (on the same CPU), or it may be at the server workstation of the cluster, or even at a workstation across a CTOS Network. If the system service is not local, the request is transparently routed across the network to the system service.

To understand how this is done, we first need to know a bit about agents.

Agents

An agent is a special type of system service process that intercepts requests destined for other system services. Its function is to participate in routing a request to a system service that is not on the same workstation with the client.

There are two kinds of agents: client agents and server agents. A client agent resides on the same workstation as the original client process that issues the request. The server agent resides on the same workstation with the system service that serves the request. An agent of one kind communicates only with agents of the other kind to transmit requests and responses. Figure 7-2 shows a simple transfer between a client agent and a server agent. A special instance of these two types of agents is when the client is at a local workstation and the server agent is located at the server workstation of that cluster. Here, the client agent is referred to as the cluster workstation agent, and the server agent as the cluster server agent.

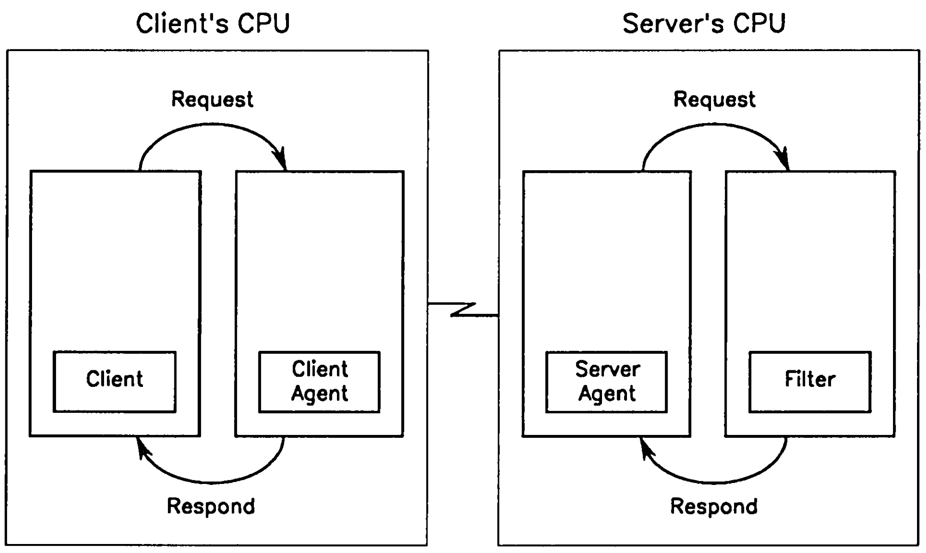


Figure 7-2. Client and Server Agents

In a cluster workstation, if a function is requested and the system service is not available locally, the request will be queued at the exchange of the cluster workstation agent. This process converts the message for transmission across the communications line to the server workstation. The workstation agent process is included in a version of CTOS specialized for cluster workstations.

Once the message is received at the server workstation, the cluster server agent reconstructs the original message and passes the request to the exchange of the system service process. Again, the cluster server agent process is included in a version of CTOS specialized for server workstations.

The format of the request block is what enables this efficient redirection of messages within CTOS. The request block is easily redirected from a process on one machine to another process on a different machine via agents. This is because the request block is self-describing, as previously mentioned, and the agents are able to transfer requests and responses between the cluster workstation and the server workstation without any knowledge of what function is requested or how it is to be performed.

This concept of agents may be extended another level with the CTOS net agent that provides a wide-area network capability in conjunction with CTOS Network software products, such as BNet. The net-agent is one system service, consisting of two processes. One process plays the client role, while the other acts as the server. The operation of these two processes parallels the operation of the cluster communication agents, but allows messaging between two servers in a wide area network. Again, the application systems programmer is unaware of the network topology beneath the application.

There may be more than one pair of client and server agents between a client and its system service. Figure 7-3 shows an example in which an application on a local workstation sends off a request that is to be performed by a system service that is running on the server workstation in a different cluster.

As shown in the lower left corner of the figure, an application makes a request, which is intercepted by the cluster workstation agent because the system service is not installed on the same processor as the client process. The cluster workstation agent is a client agent. It transmits the request over the cluster line to the cluster server agent, which in turn submits the request to the operating system on the server workstation. The cluster server agent then passes the message to the operating system which determines that the system service is not local to it, and the request goes to the net agent on that server workstation. The net agent transmits the request across the network to the network agent on a different server workstation. The net agent then submits the request to the operating system on the second server. This time, the operating system is able to map the request to the actual system service. The response to the request flows back through the system directly via the agents' response exchanges in the same manner. No kernel routing is necessary.

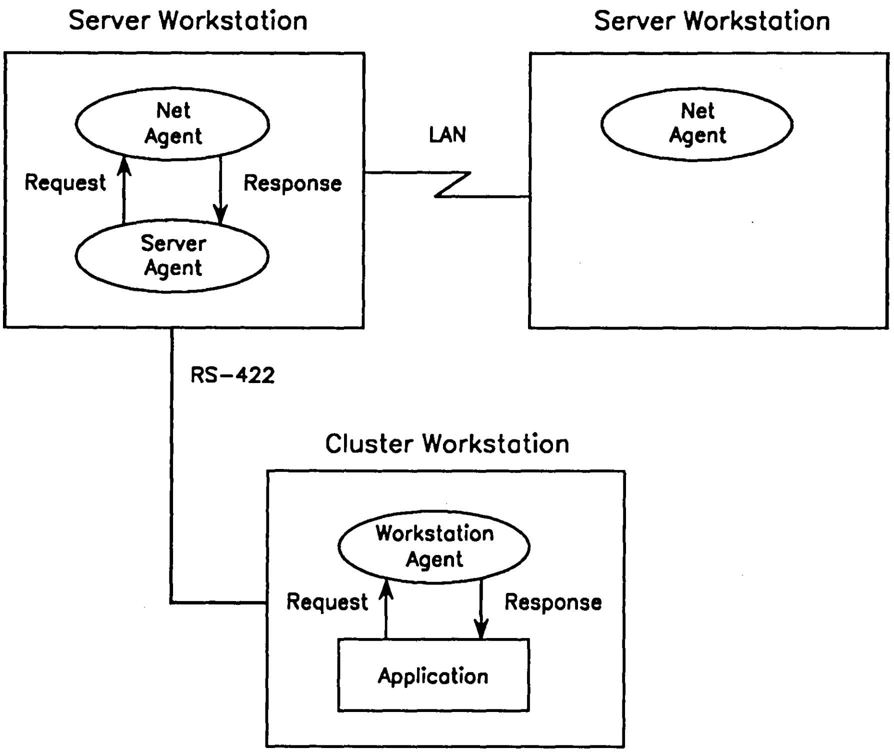


Figure 7-3. Agents Across the Network

Filters

A filter process is a special kind of system service that a programmer may write, which intercepts requests that were destined for another system service.

A filter must be installed after the original system service has been installed. It uses the ServeRq call to indicate to the operating system that it will now serve some or all of the requests sent to the original system service.

Prior to serving the request, though, the application must determine whether the request is currently in use (in case the filter will be deinstalled, the original system service must be reestablished). This is done by using the QueryRequestInfo call.

Upon deinstallation of the filter, the original exchange at which the request is processed will be restored with a ServeRq call and the exchange defined by the request information structure described above.

A filter that responds directly to the client making a request is called a replacement filter. The filter is actually replacing the functionality of the original system service.

If the filter preprocesses the request and then passes it on to the system service, which then responds to the client, it is called a one-way filter. One-way filters are used primarily to enhance statistics gathering (e.g., adding a logging capability for tracking the issuing of certain requests).

Finally, a filter that captures the request, preprocesses and forwards it to the system service, then captures the system service's response, postprocesses it, and then responds to the client is a two-way filter. Two-way filters are used to enhance functionality.

A filter process need not be consistent in the way it handles the various requests of the original system service. It may act as a replacement filter for some requests, use one-way or two-way filtering for others, and not filter some at all. It must, though, serve the system service's termination, abort, and swap requests.

A one-way filter scheme is shown in Figure 7-4. When the filter sends on the request to the system service, it must use the ForwardRequest call, rather than using Request. ForwardRequest does not require a matching Respond, whereas Request does require one. This is because with one-way filtering, we do not modify the contents of the request block. Within the request block is the response exchange where the system service will send the response. The filter process here simply intercepts the message, performs some amount of processing depending on the message (not modifying it, however), and then passes the message to the exchange of the original system service process.

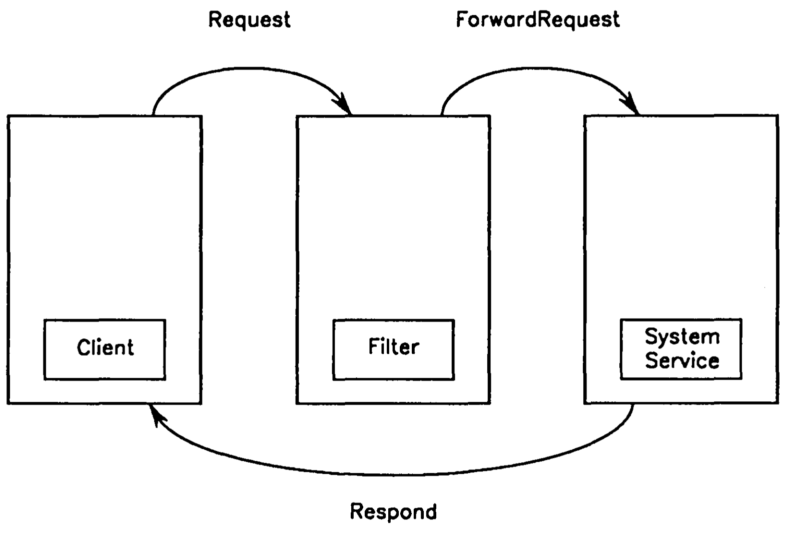


Figure 7-4. A One-Way Filter

However, you might wonder where we retrieve the identity (number) of the original system service's exchange so that we can forward the request to that exchange. Remember our discussion of the request blocks and operating system tables? Associated with each request in the tables is the number of the exchange to which that request would be sent. The only method of filtering requests is to substitute the original service exchange with the new filter system service's exchange via ServeRq. In doing so, though, we must keep track of the original exchange information. This is where we get the proper exchange for forwarding the request.

If there is a need to remove the filter, then the original exchange must be replaced. In this case, the exchange returned with the original QueryRequestInfo call must be used in another ServeRq call. After this call is completed, the Request table once again looks as it did prior to the filtering of the original request.

A one-way filter does not modify the request block, because it will not have a chance to restore the original values. For this reason, one-way filters are used mainly for preprocessing or to collect statistics.

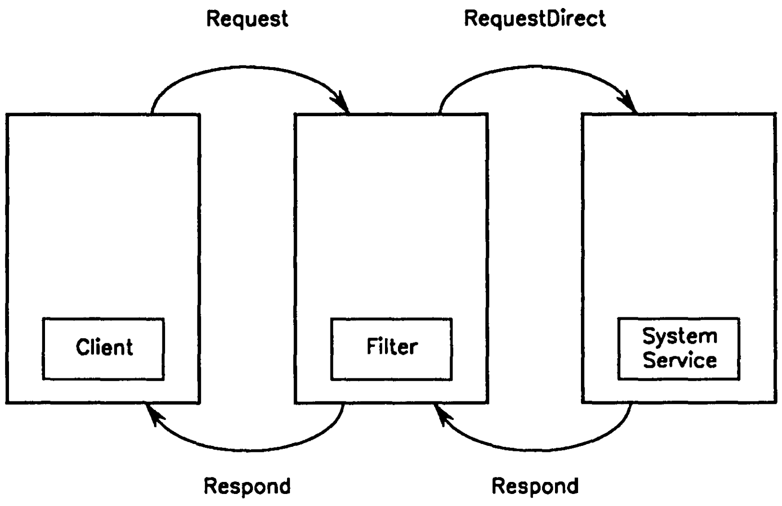


Figure 7-5. A Two-Way Filter

A two-way filter scheme is shown in Figure 7-5. When a two-way filter receives a request, it modifies the response exchange field in the request block to reflect its own exchange. It then passes the request to the system service by using the RequestDirect primitive. When the system service responds, the request block is delivered to the filter, which can then do some postprocessing and restore the request block to its original form before responding to the client. The fact that a two-way filter can modify the request block makes it much more powerful than a one-way filter.

Filters and Extensibility

Now that we see how the filter process works, it is more apparent how truly extensible CTOS is. If a one-way or two-way filter serves some, but not all, of the requests destined for a given system service, it inherits all the functionality of the system service that it does not replace. Thus, if you want to change only one aspect of what an existing system service does, you do not have to rewrite the whole thing: just put a filter in front of it.

The Basic Structure of a System Service

All system services have certain traits in common. They are easiest to see in the simplest kind of system service, which is also the first kind that the CTOS novice should try to write. This type is the single-process, synchronous system service.

In its barest outline, the system service is a loop. As shown in Figure 7-6, it contains a series of calls pertaining to its installation and to its conversion from an application program to being part of the system software. After the conversion, it notifies the operating system of the requests it will serve. It then begins the server loop: as long as it lives, it will Wait at its exchange. When a Wait is satisfied, it will process the received request. Having done so, it responds and then it Waits again.

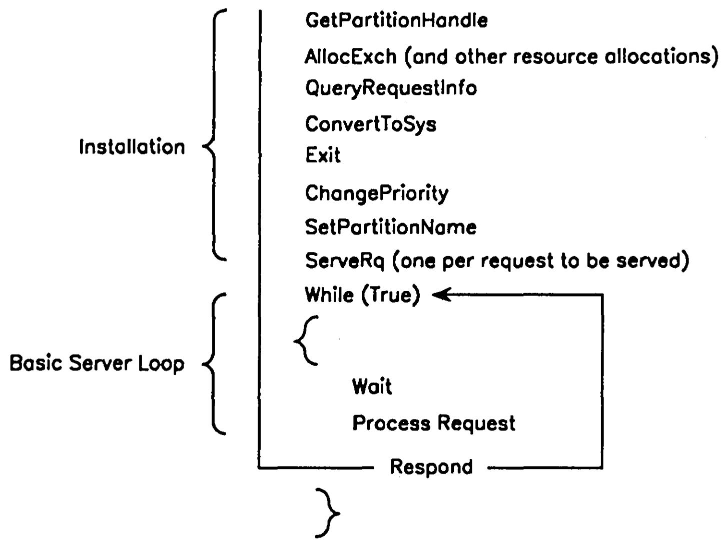


Figure 7-6. Simplest Outline of a System Service

Several calls are involved in the sequence of events in installing a system service, and the actual sequence of the calls is important. However, since there is a defined sequence, installation logic can easily be copied by any system service writer. We shall see the sequence in more detail when we write the Reminder system service in Chapter 12.

A separate deinstallation utility program is also written to aid in removing a system service that is no longer needed. This program simply issues a request informing the service that it should deinstall (clean up its environment, respond to any outstanding requests, and restore any filtered requests to their original state). When the service has completed its cleanup, it responds to the deinstallation utility, which then completes the deinstallation by removing the service from the run-time environment.

A Look Ahead

Of course, even though writing a single-process, synchronous service is not very difficult, writing system services frequently involves much more than we have described above. The need for more sophisticated system services arises rather quickly.

Suppose, for example, that your system service is located on the server workstation of a large cluster and is managing a resource such as a data storage or communications device. Multiple users are sending in requests for services. If the processing of one request takes more than a very small amount of time, it delays processing of all the others. Your users are not going to be very happy if they are stopped in their tracks while the requests sent by their application interfaces are enqueued at your exchange, waiting for their turns. Since the users do not know or care about requests, they only perceive that their systems appear to be hung.

Secondly, if you can handle only one request at a time at one exchange, what happens to the termination requests that the operating system sends you? The operating system cannot finish up a termination until you have responded. Performance on the whole cluster goes down.

So, in fact, there are other kinds of system services that you can write. You can get involved with more than one process running within the system service. You can also write the service so that it does asynchronous processing: it carries on multiple conversations with clients at once. Each of these methods has its advantages and its champions. We shall come back to them in Chapter 13. But for now, let us go on to look at I/O under CTOS.

8

Input/Output Overview

For peripheral devices under CTOS, both device-dependent and device-independent I/O methods exist. In most cases, the device-independent ones are built on top of the device-dependent ones. Each has its advantages. One of the great flexibilities of application programming under CTOS is that the programmer can intermix device-dependent and device-independent I/O methods as suits the need. The burden of understanding the device rests with the application programmer when the application does device-dependent I/O. Conversely, device-independent I/O hides the details of manipulating the device from the application.

Networked system services, as the platform on which distributed applications are built, are the heart of what makes CTOS unique. We have spent a lot of time on them, and we shall come back to them again later. They are not, however, all there is to CTOS. In this section, we shall add another layer to our picture of CTOS structure as we discuss input/output options and how they are related to each other in hierarchies.

Before we look at CTOS I/O, we must first be clear about a few concepts.

Device-Dependence and Device-Independence

Suppose you were writing a letter to ask someone to send you some information. If you were writing to your sister, Sally, you might say,

```
Hi, Sal-

Remember how we talked about Methuselah Youngblood's
out-of-print treatise on aging gracefully?  I'm pretty
sure I saw a copy bound in puce leather sitting on the
third shelf from the bottom in the bookcase in your
study.  Would you mind lending me that?  Please send
it by air.  I need help.

Yours,

Peri
```

If, however, you were writing to a used-book broker to inquire about such literature, you might say,

```
To Whom It May Concern:

Do you have any books about how to age gracefully?  If
you do, I would appreciate your sending me one by
whatever shipping method you consider most expedient.

Sincerely,

Peregrine Brittlebone
```

In the first case, you know Sally pretty well, and you know how her house is laid out, what is in the house, and where various objects are. Your letter is specific in its details, based on your knowledge. Your relationship to Sally is device-dependent: it depends on known qualities of that particular device, Sally and her house.

In the second case, you know nothing about the broker: name, facilities, structures, formats, sources, or methods. You say only what you want to receive as input (a certain kind of information); you know nothing about its form or how to get it. Your communication is device-independent. In fact, it is so generic that you could change the inside address and send exactly the same letter to any of several institutions: bookstores, libraries, even Sally herself (although she might be surprised at your formal tone). You could not, however, send the first letter to anyone other than Sally and expect to have it understood.

By analogy, processes can undertake I/O with peripheral devices in either device-dependent or device-independent ways. A device-dependent I/O communication is specific to the kind of device, and the process must have varying degrees of knowledge about the device and how its contents are arranged in order to send output to it or receive input from it. The burden of understanding the device (and the writing of a lot of sophisticated code) rests with the application programmer when the application does device-dependent I/O. Conversely, device-independent I/O hides the details of manipulating the device from the application. Device-independent I/O communications are so generic that, as with Peregrine's second letter, they can be sent To Whom It May Concern: the same communication to any of several devices will still be understood. This approach requires less of the application programmer. (The necessary sophisticated code didn't go away: it was just written by someone else.)

These approaches have their pros, cons, and trade-offs: we shall get to those as we look in more detail at types of I/O under CTOS.

Hierarchies of CTOS I/O Tools

For peripheral devices under CTOS, both device-dependent and device-independent I/O methods exist. In most cases, the device-independent ones are built on top of the device-dependent ones. Each has its advantages. One of the great flexibilities of application programming under CTOS is that the programmer can intermix device-dependent and device-independent I/O methods as suits the need.

Primitives: Device-Dependent

Each device has ways specific to itself in which it can be called for I/O. These primitive calls require the caller to know, for example, in making a video call, what the screen coordinates are at which the next output should be made. Similarly, the primitive for reading from disk requires the caller to know the location of the data.

Primitives are tailored to the device and the task. They provide the highest performance of any method but require the greatest programming skill and knowledge. Most primitive I/O calls are requests to system services. Some are calls to subroutines that are part of the operating system (but not of the kernel) and are referred to as system common procedures. System common, as it is called casually, is a little off the beaten track of Ctosian theory: it is more reminiscent of UNIX or of other operating systems in which all system calls go to routines that are part of monolithic system software.

System common (which we mentioned at the end of Chapter 6) is mainly composed of those routines for which performance is so critical that the small overhead inherent in request handling would not be tolerable. The largest part of system common consists of primitives for video handling. System common also includes other routines that are so commonly used (hence the name) that to put them in link-time binding libraries would result in excessive duplication of code. (Naturally, semaphore-based techniques must be used to police the action in the system common area: another way in which the Ctosian religion is not strict.)

Primitive calls, whether satisfied by system services or by system common procedures, are the building blocks from which higher-level, more generic I/O tools are composed.

Tools Built on Primitives

Sequential Access Method

The largest group of nonprimitive I/O tools is the group of object module procedures called the Sequential Access Method (SAM). They are library routines that are provided with CTOS but are not part of it; their code is copied into applications at link time. The CTOS object module procedures are contained within a development library, CTOS.lib, and are automatically resolved during the linkage of an application system. Because they are object module procedures, any application that requires them ends up with the object code contained within its executable file. Object module procedures include several different areas of functionality, but the Sequential Access Method accounts for a substantial part of this group of routines.

The Sequential Access Method is more colloquially and easily referred to as "byte streams," because that is precisely what it uses.

The CTOS byte-streams mechanism was the only part of the early CTOS design that truly showed an influence from UNIX. (There was, you may remember, one engineer from AT&T Bell Laboratories on the first CTOS team, and this feature seems to have been his mark.) Under the UNIX byte streams mechanism, every peripheral device is regarded in the same way, that is, as a file, and input from or output to any device is an unformatted stream of bytes, which must be parsed by the recipient. The CTOS idea of byte streams is similar, except that devices are conceptualized in a generic way, and not literally as nodes in the file system, as they are in UNIX. CTOS byte streams are very similar in mechanism to UNIX and OS/2 pipes, although CTOS byte streams do not directly support the transfer of streams of bytes between programs. (Actually, a CTOS reseller is known to have created a system service that uses byte streams to provide a pipe feature.)

Figure 8-1 is a very oversimplified picture of anybody's byte streams in action. On the process side, the shipping department busily loads bytes, one after the other, with nothing between them, onto the outgoing conveyor. This stream of byte-boxes has no built-in meaningful pattern: they just slide out the exit door one after the other. On the device side, the receiving department has a more responsible function. The receiving worker lays some kind of template over the stream of bytes as they flow in. The template adds meaning to the stream of bytes. They can now be arranged on the dolly in patterns, ready for the processing crew to act upon. Similarly, a byte stream can flow in the opposite direction, from device to process. Now it is the device that puts out a generic stream, and the process that must interpret what it receives.

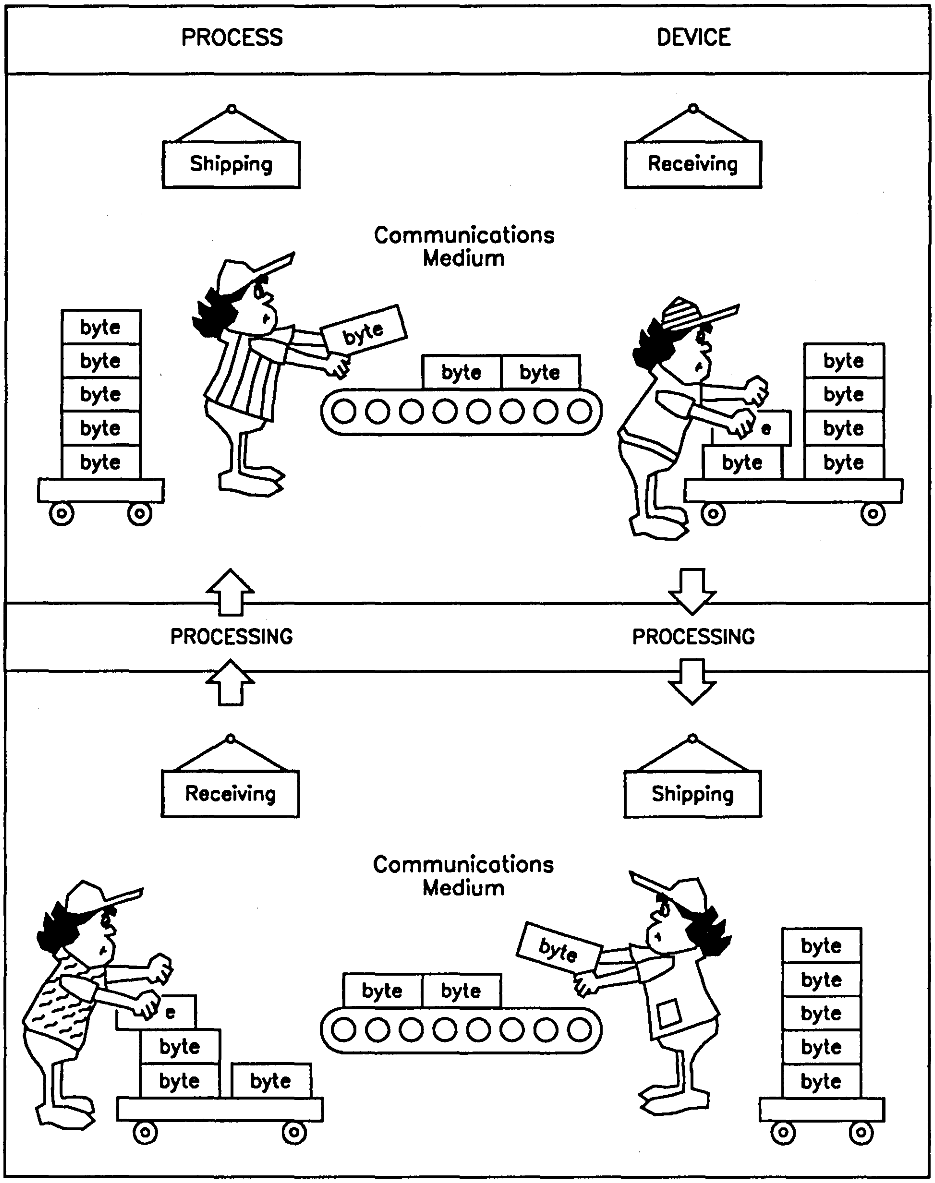


Figure 8-1. A Simplistic Picture of a Byte Stream in Operation

This picture of byte streams illustrates one of the prices of the device-independent process, as well as the process itself. Clearly, some extra work is going on here on both sides. The sender must pack things up in a completely generic way, while the receiver must employ intelligent means of figuring out what it has received. Someone has to figure out who the actual receiver is to be and how to get there. All this doing and undoing means extra code: in CTOS, the Sequential Access Method code that was bound in from libraries at link time. Wouldn't it be more efficient in size and performance to have both sides handle known data patterns directly, in a device-dependent manner? Yes, it would. What do you get for using byte streams?

You get device-independence, with advantages of modularity, reusability, ease of redirection to different devices, and ease of programming. The sender need only know who the receiver is. The format of what is sent is the same for various kinds of receivers. As a result, the sender could contain a generic procedure (perhaps called OuttaHere), in which a sequence of byte stream calls is made that are the same no matter who is the recipient in a given instance. This same OuttaHere procedure could be called (with the name of the recipient) in any number of different places within the sending program to output data to different devices. This generic quality saves a lot of coding on your part in the application. It also means that you do not have to know anything about such matters as blocking factors on the disk or exact coordinates on the video. Your code just works across the hardware platform.

A disadvantage of using byte streams is that there is large-scale duplication of code that results when various applications, all calling for the same kinds of I/O, bind into themselves the same code at link time. Historically, this problem has existed in different systems, which now are starting to handle it in different ways. OS/2 handles this problem by using dynamic link libraries, or DLLs. CTOS soon will also use DLLs here. In dynamic linking, the necessary code is not bound into the application's own executable file at link time. Instead, it is "bound" at run time, as execution jumps to the needed routines, which are separately resident on the system. With the use of DLLs, multiple applications can use the same library code without code duplication. Semaphores are used to control access to critical sections of code.

Having drawn a neat picture of byte streams as a higher layer built upon device-dependent primitives, we now need to fuzz it up a bit for the particular case of CTOS byte streams. CTOS SAM does include this neat, generic layer of byte streams that can be used with any device. It also, however, includes device-dependent extensions to the basic byte stream calls. Thus there exist both device-independent and device-independent routines within SAM, still above the primitive level.

Now that we are into four levels of thinking about generality and specificity in I/O, it seems best to draw a picture. In Figure 8-2, the vertical arrows represent calls from higher layers being transparently expanded into lower-level calls. Ultimately, all I/O calls are satisfied by request-based services or System Common routines.

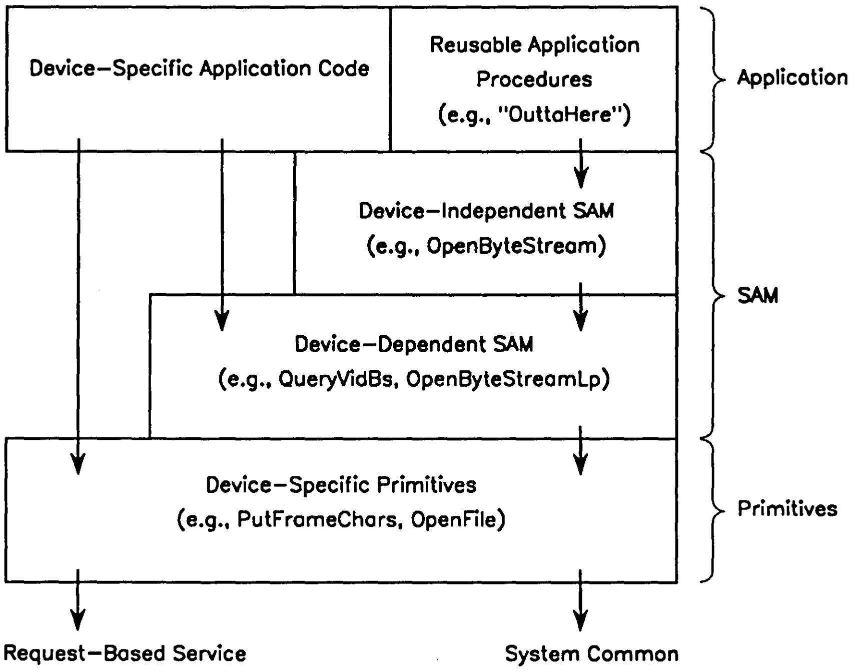

Device–Specific Application Code
Reusable Application Procedures (e.g., "OuttaHere")
Application
Device–Independent SAM (e.g., OpenByteStream)
SAM
Device–Dependent SAM (e.g., QueryVidBs, OpenByteStreamLp)
Device–Specific Primitives (e.g., PutFrameChars, OpenFile)
Primitives
Request–Based Service
System Common

Figure 8-2. The Hierarchy of Device-Independent and Device-Dependent Calls in CTOS I/O

The most often used device-independent SAM procedures are few and rather obvious. They have names such as OpenByteStream, ReadBsRecord, ReadByte, WriteBsRecord, WriteByte, and CloseByteStream. With each call, the address and size of a device name (such as the video device name, [VID], or the keyboard device name, [KBD], or a file specification) are passed, along with the addresses of a work area and buffer to be used by SAM, and password and mode information as needed.

To use byte streams, an application must first open a device or file as a byte stream. It does so by using the OpenByteStream operation, providing the file specification and password for access to the device or file indicating whether read, write, or modify access is desired. (Modify access allows both read and write access.) A buffer is passed in the call to be used exclusively by the SAM operations for I/O. The byte stream is then defined by a byte stream work area (BSWA). This parameter is the address of a 130-byte area also used exclusively by the SAM operations. (However, operation specific information is contained here: for example, the read and write positions in the buffer.) Remember, a template must be supplied for defining the stream of bytes. The BSWA is the template. If the byte stream was opened in Read or Modify mode, then the ReadBsRecord or ReadByte operations can be used.

Both calls take the address of the BSWA that was previously opened with the OpenByteStream operation. The difference between the two, though, is that the first allows a read of a variable amount of data into a buffer with the return of the actual amount of characters read. The second allows a read of only a single character. Why would you use one or the other? Communications processing is dependent on specific characters, where the state may change depending on the incoming character. Here one would use the ReadByte operation. Disk processing, on the other hand, is more efficient when operations are multisectored, so you would use ReadBsRecord here.

If the byte stream was opened in Modify or Write mode, then the WriteBsRecord or WriteByte operations may be used.

Notice that both of these calls also take the address of the BSWA that was previously opened with the OpenByteStream operation. The difference between the two, though, is that the first allows a write of a variable amount of data into a buffer with the return of the amount of characters written. The second allows a write of only a single character.

In succeeding chapters, we shall be more specific about the contents of device-dependent SAM. Suffice it for now to understand the general pattern and to understand that CTOS SAM is configurable The basic package includes asynchronous disk, keyboard, video, parallel printer, null, and spooler byte streams. Also in existence are byte streams for communications, direct serial printing, tape, and the Generic Print System (GPS, further discussed below and in Chapter 11). You can configure the byte streams package to include only what your program actually uses. You can even write your own device-dependent byte stream for any device and add or substitute it into the package. CTOS system documentation covers the method for this customization. In Figure 8-2, this kind of customization would be done within the layer labeled Device-Dependent SAM.

Non-SAM Tools

Not all high-level I/O methods are related to SAM. Two other packages, one designed early on in the history of CTOS, and one just now coming into existence, handle both keyboard and video and facilitate user interface design.

The simpler package, called Forms, first appeared in 1980 with the earliest versions of CTOS. It enables the programmer to quickly and easily put together screen forms for user input and output on character-based video. Although in this day of the bit-mapped graphical, point-and-select user interface, Forms appears to be old-fashioned, it is still useful for character-based applications such as order entry. The Forms package has no relation to byte streams; it is built directly on the Video Display Manager (VDM) and Video Access Method (VAM), which are video primitives resident in System Common.

A newer package, called Extensible Virtual Toolkit, or XVT, is an open standard for creating graphical and character-based windowing user interfaces for character-mapped and bit-mapped systems. XVT allows you to write a single program that can run in several different window environments on different operating systems, for example, MS-DOS, OS/2, and UNIX. XVT is now becoming available as part of the CTOS/Open standard. It simplifies the portation of applications from other environments to CTOS.

XVT handles video, keyboard, and mouse user inputs and outputs. It can adjust to handle a range from the simplest character-based windows through the most sophisticated bit-mapped graphical user interface. XVT is implemented on top of the native toolkit for any given operating system, rather than directly making operating system calls itself. In the CTOS environment, XVT is built on whatever CTOS windowing system service is appropriate for the hardware on which it is running. (We shall discuss XVT further in Chapter 9.)

A CTOS I/O Road Map

To tidy up our thinking before continuing into further chapters on particular areas, let us summarize the types of I/O under CTOS. Within each type, we shall identify the hierarchy of I/O levels, if any. No system would be complete without its exceptions, and we shall touch on those, too.

In the next few chapters we shall look at each area in a little more depth.

Video

The relationships of the various video access tools and methods are shown in Figure 8-3. The underlying primitive layer is composed of the Video Display Manager and Video Access Method, which are resident in System Common or are request based. Video byte streams and the Forms package are built on VDM and VAM. XVT is built on the appropriate windowing package for the hardware on which it runs. CCGI+, the graphics library, also depends on VDM and VAM.

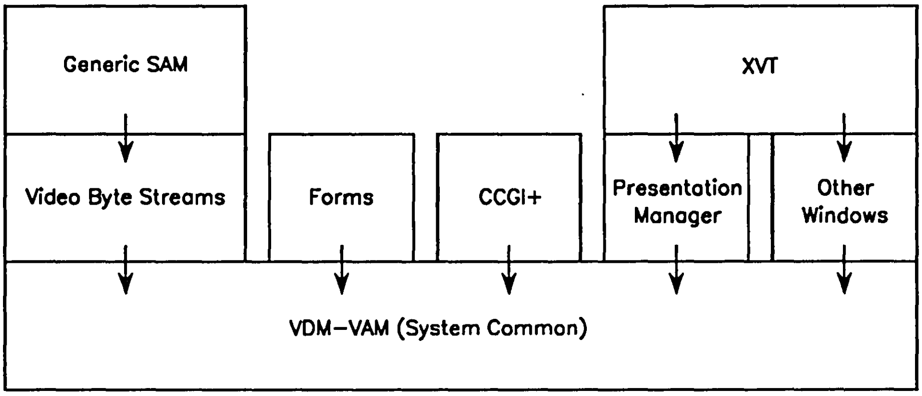


Figure 8-3. Video I/O Tools

Keyboard and Mouse

Keyboard byte streams depend on an underlying keyboard management system service, as shown in Figure 8-4. The Forms package also makes requests to this service, as does XVT.

A separate system service handles the optional mouse. This system service contains a query, ReadInputEvent, that gets both keyboard and mouse events as they occurred in time. (ReadInputEvent is general and expandable to allow for other kinds of events in the future.)

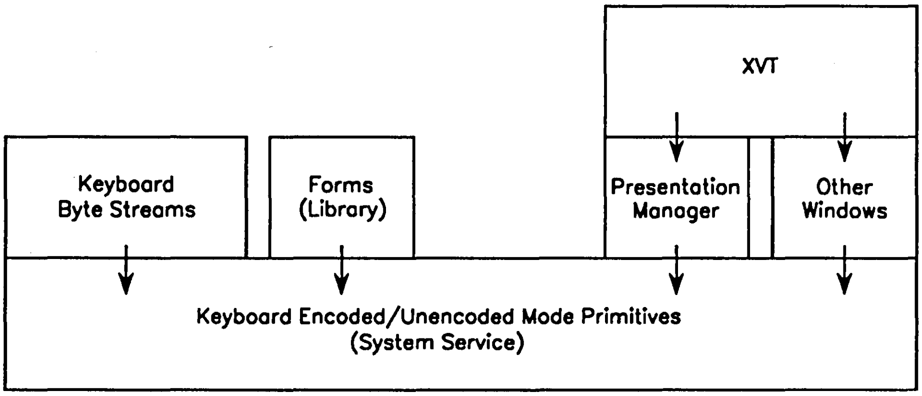


Figure 8-4. Keyboard and Mouse Management

Data Storage

Data storage can involve disk or tape. Disk storage methods (Figure 8-5) are all ultimately built upon the device-dependent file system service. SAM (disk byte streams) depends on it, as well as two less frequently used sets of object module procedures, the Direct and Record Sequential Access Methods (DAM and RSAM). In its most widely used form, the Indexed Sequential Access Method (ISAM) is a system service that in turn makes requests of the file system service. ISAM is a distributed data base tool.

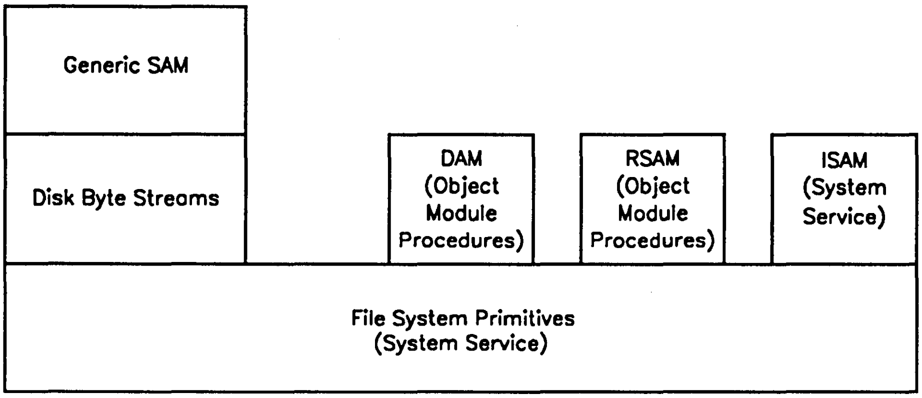

Generic SAM
Disk Byte Streams
DAM (Object Module Procedures)
RSAM (Object Module Procedures)
ISAM (System Service)
File System Primitives (System Service)

Figure 8-5. Disk Storage Methods

Half-inch, quarter-inch (QIC), and digital audio (DAT) tape are supported. The device-dependent tape system service underlies tape byte streams (Figure 8-6).

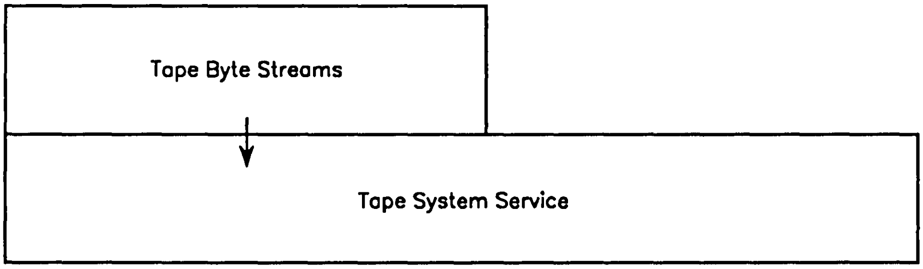


Figure 8-6. Tape Storage Methods

Communications

Generic SAM allows you to treat a communications device in the same way as any other device. This device-independent layer, in turn, calls communications byte streams (SAMC), the communications device-dependent portion of SAM. At the most primitive level are serial port management procedures that reside in System Common for performance reasons. (Communications interrupt service routines can be written to call serial port management procedures directly.) Figure 8-7 shows these relationships.

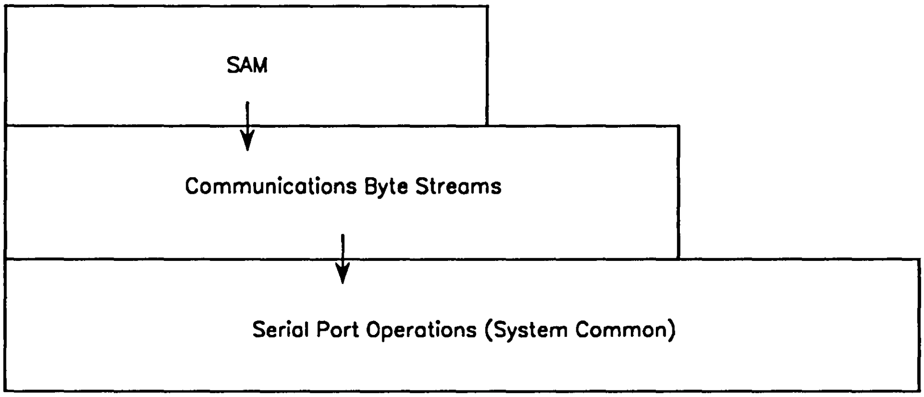

SAM
Communications Byte Streams
Serial Port Operations (System Common)

Figure 8-7. Communications Methods

Printing

The world of printing under CTOS can be confusing to the newcomer at first, because it contains layers of old methods still maintained for backward compatibility, side by side with the more recent Generic Print System (GPS).

The original SAM supports both parallel and serial printers as devices to which byte stream output can be written. When you use this kind of direct printing to a printing device from within an application program, the output goes directly to a locally attached printer, and you cannot use any form of spooling. SAM relies on lower-level communications byte stream routines to carry out the work of direct printing.

Device-independent SAM also recognizes as a device [SPL], the original spooling method designed for CTOS when applications did their own printing. This method is casually referred to as the "old spooler," because, of course, there is a new spooler. Again, communications byte streams underlies this spooler.

The Generic Print System was designed as a more modular and device-independent way of printing in 1986. It allows applications to request printing services without containing printing code themselves, thus cutting out enormous code duplication. GPS methods include GPS byte streams and the Generic Print Access Method (GPAM). The GPS byte streams method is simply the generic SAM we have already discussed, where a printer name known to GPS is used as the device name. GPS byte streams allows only the simplest kind of printing with no sophisticated formatting in the output.

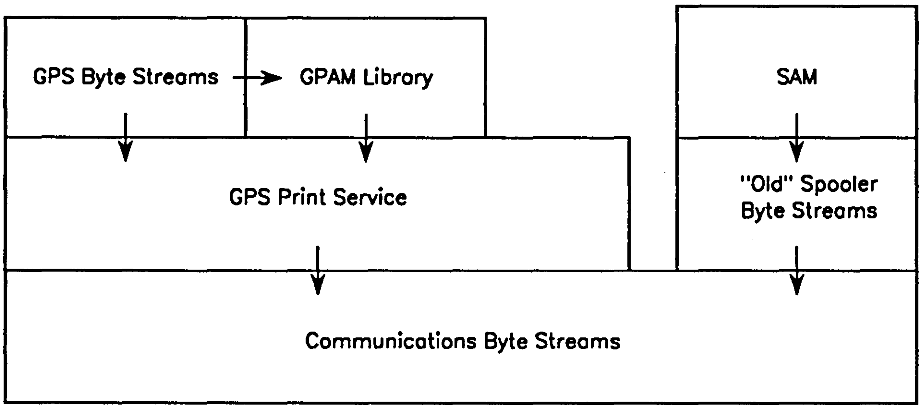


Figure 8-8. Two Printing Systems

GPAM is a separate, SAM-like library to which application programmers can write. GPAM is GPS's own byte stream system. It allows sophisticated formatting. GPAM, in turn, calls GPS byte streams.

GPS byte streams (SAM) makes requests to the underlying Print Services system service in GPS. GPAM also makes some requests directly of the Print Service. It is possible for application programmers to make requests directly to the Print Service also, although this is usually done only for certain specific purposes, and no special formatting can be used via this path. Figure 8-8 shows these relationships.

9 Video and Keyboard Options

With good user-friendly graphical user interfaces becoming standard, it might seem obvious to port one to CTOS and be done with the problem. Typically, however, CTOS developers have not been entirely satisfied with that approach. CTOS has a strong tradition of providing device-independent and backward-compatible solutions that also allow for extensibility in the future. The developers examined the characteristics of many products and decided to combine two of them with what already existed to make a truly comprehensive solution that would open up many new possibilities and provide a solid platform for development through the 90s.

Having taken a brief tour of CTOS I/O tools at various levels, let us focus in on the video and keyboard I/O possibilities and when they might be most profitably used. We will look first at the video options, then keyboard handling, and finally forward to new alternatives offered by a graphical user interface.

Video Frames and Attributes

Before we explain the video options, we need to define some terms used for video.

Frames

A video frame is a rectangular area that you can define on the video display. It can have an optional visible border, and its contents can be scrolled up and down independently of other frames. Normally, CTOS is configured so that you can create up to eight frames, but the operating system can be reconfigured to support up to 256 frames.

The virtue of a frame is that what you write to it is automatically limited to that frame and cannot overwrite the contents of other frames. If you are using very simple video output, you may not need more than one frame. The first (and default) frame is frame 0. The CTOS Executive (the command-line interpreter) sets up its video with three full-width frames. In this case, frame 0 occupies the largest part of the video and is the frame with which the user interacts. Frames 1 and 2 are narrow frames at the top of the screen that display various status information and user notification messages.

A frame can cover any rectangular screen area, and frames can overlap each other. The most commonly used frames are of full screen width, but narrower ones can be defined for such purposes as small forms or pop-up windows.

A frame is created by use of the InitVidFrame procedure, which is part of the Video Display Manager (VDM). The parameters passed with InitVidFrame describe the extent of the frame in terms of screen columns and lines. They also describe the border.

In addition to the InitVidFrame operation, other operations allow you to query frame characteristics and to manipulate frame characters.

We are jumping ahead of ourselves, though. Before a frame can be initialized, there are a number of calls which must first establish the video subsystem for the application system. We shall examine these calls shortly.

Video Attributes

Video attributes can pertain to the entire screen (blank, number of characters per line, and so on) or to individual characters (bold, underlined, and so on). Some attributes (reverse video, half-bright) can pertain to either the full-screen or to characters.

To set video attributes on a full-screen attribute, the SetScreenVidAttr call is used. The passed parameters define the screen attribute and whether the attribute is set or reset.

Additional calls allow the setting and querying of individual characters.

Types of Video Hardware

Both character map and bit map monitors are supported by CTOS workstations. In each case there is a character map, although it is implemented differently on the two types of hardware.

Color monitors are also supported. Programming the color capabilities of these workstations involves calls to standard library functions to manipulate the operating system's color control structures. These operations are separate from the hierarchy of video I/O tools that we shall discuss below.

Levels of Video Access

There are several levels of video access available through CTOS, ranging from the most device-independent to lower-level more device-specific.

Figure 9-1 (which repeats and extends Figure 8-3) shows the video output tools and their interrelationships. Actually, this figure shows two generations of higher-level tools: the Sequential Access Method (SAM), Forms, and CCGI+ on the left, and the new graphical user interface (GUI) on the right.

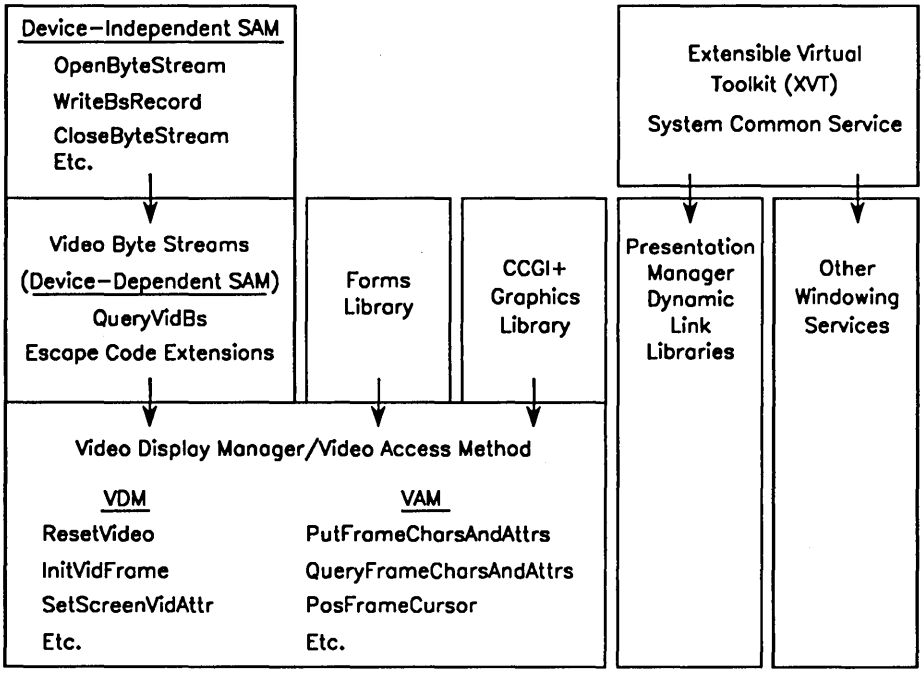


Figure 9-1. Video Tools

At the highest level among the traditional tools on the left is device-independent SAM, or byte streams. These by-now-familiar library routines in turn call the device-dependent SAM layer and the primitive layer, represented by the Video Display Manager (VDM) and the Video Access Method (VAM). The Forms library also makes calls to VDM/VAM.

You can mix the use of these traditional tools within one program: if you are primarily using video byte streams, there is nothing to prevent your resorting to direct VDM/VAM calls where you deem it necessary.

In recent years, graphics programming has often been done through the CCGI+ graphics library, which is compatible with the standard Computer Graphics Interface (CGI). Graphics library routines in turn call VAM routines.

In addition, XVT can call whatever other windowing tool is present, depending on the hardware: a character-based facility, Presentation Manager, and others not yet identified.

Using Video Byte Streams

Device-dependent video byte streams extends generic SAM in three ways. First, certain characters (for example, up arrow, backspace, and tab) sent through the byte stream are interpreted to move the cursor in various ways, blank the frame, and so on. Second, multibyte escape sequences (beginning with 0FFh) can be sent to control a great many things, including screen and character attributes, cursor position, and so on. Third, the one device-dependent video byte stream operation, QueryVidBs, returns information to the caller about the status of the current frame: the current frame number, its size, number of lines, cursor position, attributes, and other characteristics.

To use video byte streams, the programmer must first set up the screen by using VDM operations. In practice, developers usually use video byte streams only when their programs will output a very simple stream of data continuously to a screen inherited from another application, with all the frames predefined. This usually means writing to frame 0 within the Executive screen. Video byte streams are used throughout the Executive itself and related utilities. For example, a listing of files within a directory, or the status messages displayed during the copying of files, are easily and appropriately output via video byte streams. If the display is to involve even slightly sophisticated cursor or scrolling manipulation (back scrolling), among other things, the programmer should immediately move to VAM. VAM also provides better performance, at the cost of writing more complex code.

Using VDM and VAM

The operations of VDM pertain to screen set-up. If a program that uses video byte streams does not inherit a set up screen from a previous program (usually via a Chain from the Executive), it must call VDM to set up its own screen.

Initializing the Video Subsystem

First, the application must determine the level of video capability present on the screen. If the application is going to use graphics, then graphics hardware must be present. The QueryVidHdw call places hardware-specific information in a user-supplied buffer.

Included in the information returned are the basic video capabilities (character versus bit mapped), the size of the screen in terms of columns and lines, and what graphics capabilities are available.

The information returned by QueryVidHdw is necessary so that we can initialize the video subsystem. We do so with the ResetVideo operation. This operation suspends video refresh, resets all screen attributes, and changes the values in the Video Control Block (one of the U-structure entities) to reflect the values passed.

Next we initialize each of the video frames by using the InitVidFrame call that we mentioned when we discussed frames, above. After the video frames have been set, we make a SetScreenVidAttr call to set any desired screen video attributes.

Next, the character map must be initialized for use by the video hardware. We do so with an InitCharMap call, passing the size of the character map, which we got from the ResetVideo call. You can see that the various calls are interdependent.

From the ResetVideo operation until now, the video screen has been blank. Errors received on these operations are not noticeable until the application fails, making it a little bit more difficult to debug an application within this critical section.

To enable the video refresh, we finally issue a SetScreenVidAttr call with the video refresh attribute selected and its flag turned on as the parameters.

Writing to the Video

Once the sequence of VDM events is complete, the complete video subsystem is ready for use by the application. All the video frames have been initialized and can be used with direct VAM or VDM operations, SAM operations, or even graphics or windowing operations.

The VAM operations give the programmer finer control over cursor position, attributes, and scrolling than do the SAM operations that we have already seen. VAM allows you to scroll entire or partial frames up and down. It supports text editing: you can scroll up, for example, the top four lines of a frame and insert a new line of text between the fourth and fifth lines. Character attributes scroll along with the text they affect. You can write to any position in the frame as needed.

VAM also contains primitive graphics operations that are not directly called by applications programmers. Higher level graphics software depends on these primitives.

Keyboard Input Tools

At the primitive level, the keyboard is handled by a request-based system service and an operating system keyboard process. Together these components are called keyboard management. Figure 9-2 shows the keyboard input tools, with keyboard management as the fundamental layer. At the SAM level, keyboard byte streams depend on keyboard management.

Again, the new generation graphical user interface is shown at the right side of the figure. It also relies on keyboard management for primitive procedures.

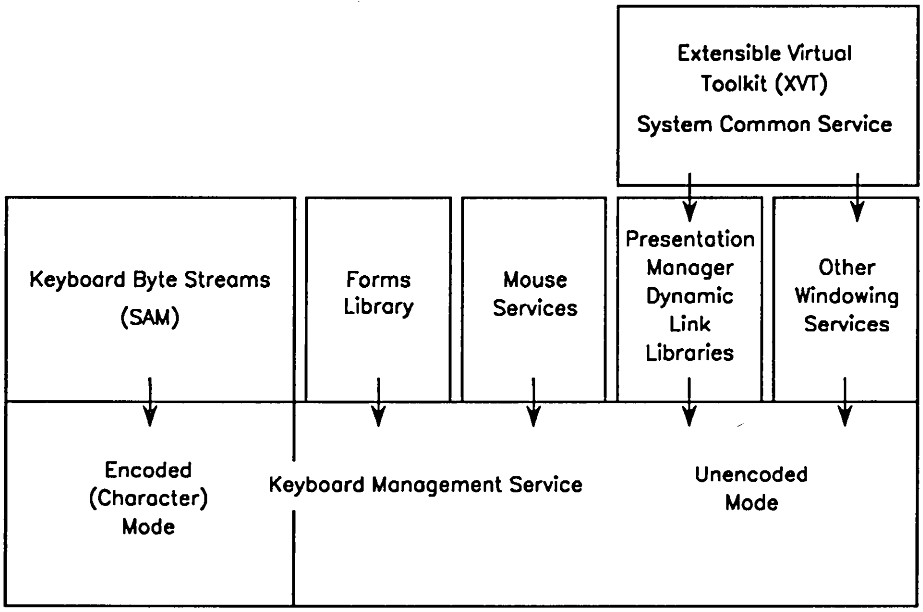

Extensible Virtual Toolkit (XVT)
System Common Service
Keyboard Byte Streams (SAM)
Forms Library
Mouse Services
Presentation Manager Dynamic Link Libraries
Other Windowing Services
Encoded (Character) Mode
Keyboard Management Service
Unencoded Mode

Figure 9-2. Keyboard Tools

Keyboard Byte Streams

Keyboard byte streams provide the easiest way for an application to get information from the keyboard, but they add all the overhead of byte streams to the program They are also too slow for many situations, and so in fact they are seldom used. Keyboard byte streams provide one character at a time to the application.

Keyboard Management

The keyboard system service can provide information to an application in either of two ways. In character mode, the service reports one byte that represents the key typed by the user. (A keyboard mapping table maps keyboard codes to character codes.) In unencoded mode, the service reports both the downstroke and the upstroke of any key pressed, and it reports these events in the order in which they occur, for example:

left shift down
'A' key down
'A' key up
left shift up

This approach allows the application great flexibility in assigning meaning to keyboard interactions but requires more sophisticated and extensive programming than does character mode. Unencoded mode is used, for example, in text-editing applications. Character mode is, however, more generally and easily used.

To set the mode of operation for the keyboard, we would use the SetKbdUnencodedMode call, passing a flag value to determine whether the mode is encoded or unencoded.

We can access keyboard data with the ReadKbd call, passing the address of the next character to be read. This sequence of events shows that an encoded-mode operation requires a single keyboard read, whereas the unencoded-mode operation requires multiple reads.

In most cases the ReadInputEvent call is a better method of getting keyboard data. This call is part of the keyboard process, but cooperates with the Mouse Services that we shall discuss below. ReadInputEvent returns an interleaved stream of keyboard and mouse events as they occurred in time. It eliminates the need for applications to construct polling busy loops to get this information. If no mouse or other pointing device is present, ReadInputEvent still works, returning only keyboard events. The only disadvantage of this call is that it does not work with the system input process (below).

Keyboard byte streams uses keyboard management only in character mode.

System Input Process

The system input process is not really an integral part of the keyboard I/O hierarchy we have been discussing, but it is so closely associated that we shall touch on it here. The system input process allows all keystrokes to be recorded in a file at the same time as they are being reported to the application that requested them. These keystrokes can then be played back from this file to the application, just as if they were being typed at the keyboard again. The most common use of this feature is through the Executive commands Record and Submit. Using these commands, a user can directly create a such file to do, for example, a routine backup, or to generate a monthly report that always requires the same commands. Such files are called "submit files" under CTOS. They are like macro or script files used with other systems. Submit files are also used to automate testing and the building of executable files.

The system input process is flexible enough so that some keystrokes can be designated to come from the real keyboard during the replay of a submit file. A special sequence of characters (an escape sequence) informs the system input process that the following keystrokes will be from the real keyboard rather than from the submit file. The keyboard input then continues until the character specified in the escape sequence is received from the keyboard (for example, the Go or Finish key).

A submit file can also be composed directly in a text editor, rather than recorded from actual commands. However, because the system input process operates only in encoded mode, unencoded applications do not function with it.

Mouse Services

Mouse and other pointing-device input are handled by a system service called Mouse Services. This system service cooperates with the keyboard process to provide input event information to the caller through the ReadInputEvent call. Traditional and forthcoming user interfaces all rely on Mouse Services for primitive mouse support.

Forms Package

Created early in the life of CTOS, the Forms facility is a high-level interface tool based on VDM/VAM that consists of three parts: the Forms Editor, with which you can design forms on the screen and save them into files; a Reporter utility that can display information about a form; and an object module library that displays the form, prompts the user to enter data, and returns the data to the calling program. A "test drive" feature allows you to try out a form as soon as you have designed it, without having to call it from an application program.

The Forms package is entirely character oriented, because when it was designed, there were no graphics workstations. Forms created with this tool have all the features needed for routine, character-oriented data handling.

Forms is still in use. It works well for relatively rapid creation of user interfaces for such purposes as routine order entry on low-cost, character-based systems.

Graphical User Interface

In recent years, as use of small computers in offices proliferated, user interfaces from one system to another have become more standard. Graphics hardware is also now much more common.

The original user interface for CTOS, the Executive, is a command interpreter that works on a fill-in-the-blanks principle and presents the user with a simple form to fill out for each command, rather than requiring that the user remember the sequence of parameters as they must do with a command-line interpreter.

In 1980, before the birth of the Macintosh®, this was an advanced and user friendly interface. In 1984, the CTOS Context Manager took another step forward with an additional interface that allowed the user to interactively start multitasking applications with point-and-select. In the 90s CTOS is moving to the use of a new graphical user interface.

The CTOS GUI Solution

With good, user-friendly GUIs becoming standard, it might seem obvious to port one to CTOS and be done with the problem. Typically, however, CTOS developers have not been entirely satisfied with that approach. CTOS has a strong tradition of providing device-independent and backward-compatible solutions that also allow for extensibility in the future. The developers examined the characteristics of many products and decided to combine two of them with what already existed to make a truly comprehensive solution that would open up many new possibilities and provide a solid platform for development through the 90s.

Presentation Manager

One component of the new GUI is Microsoft's Presentation Manager. A powerful tool for creation of complex, windowed application user interfaces with standard components, Presentation Manager also offers a desktop interface that permits applications to share the screen. However, Presentation Manager has a large and complex API that takes some time for a programmer to learn. Applications written to use Presentation Manager are not easily ported to other windowed systems. Also, Presentation Manager is entirely graphics oriented: it does not support a windowed environment on a character-based monitor. If only Presentation Manager were offered, applications that used it would be limited to run only on the more recently produced graphics hardware.

Extensible Virtual Toolkit (XVT)

The complementary piece of the new GUI is XVT Software Inc.'s Extensible Virtual Toolkit (XVT). As we mentioned earlier, XVT is an open standard for creating graphical user interfaces for character-mapped and bit-mapped systems. It provides the link that will allow applications developed for the CTOS GUI to run with the same user interface on both types of systems.

XVT is also bridge tool. It allows you to write a single program that can run in several different window environments on different operating systems. An application written strictly to the standard XVT interface should be able to run on systems as various as the Apple Macintosh and UNIX Motif systems, with only recompilation and relinking.

XVT has a much simpler API than does Presentation Manager. However, unlike Presentation Manager, it does not provide a user environment or desktop.

XVT is a layer between the application caller and whatever windowing facilities ultimately run the user interface on a given system. If the system has none, XVT supplies its own character-based windowing facility. On CTOS XVT is implemented as a system common service that accepts calls through the standard XVT interface. It has the great advantage that a CTOS application written to XVT is compatible with all CTOS-based systems without recompiling or relinking.

XVT is called extensible because, although you get the maximum portability by writing strictly to the XVT API, you can request information (handles) from it that would allow you to call the underlying windowing service (for example, Presentation Manager) directly.

Figure 9-3 shows the position of XVT in various systems.

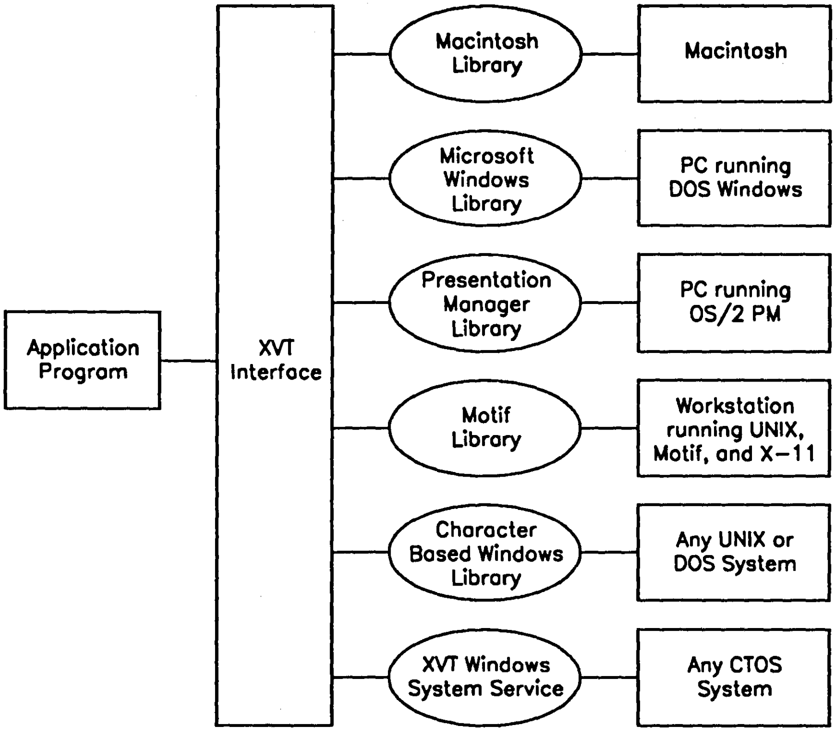

Application Program
XVT Interface
Macintosh Library
Macintosh
Microsoft Windows Library
PC running DOS Windows
Presentation Manager Library
PC running OS/2 PM
Motif Library
Workstation running UNIX, Motif, and X-11
Character Based Windows Library
Any UNIX or DOS System
XVT Windows System Service
Any CTOS System

Figure 9-3. Extensible Virtual Toolkit

Context Manager's Role

On character-based systems where the Presentation Manager desktop is not present, the CTOS Context Manager continues to provide the user interface for context switching. Full-screen standard or XVT windowed applications can be started and switched to from Context Manager.

This solution combines the advantages of each of the components. It is relatively easy for programmers to write to XVT. The resulting programs can be ported readily, and XVT-based applications from other environments can easily enter the CTOS world. Both character-based and graphical environments are handled. Finally, users can do convenient interactive multitasking either through Presentation Manager's Desktop Manager, if that is available, or through Context Manager, if it is not.

Later Additions

In the CTOS tradition, this scheme allows for both backward compatibility with existing hardware and also for future change. Presentation Manager may not be the only windowing facility that developers want to use on CTOS in the future. New facilities can be added to this general scheme as time goes by and as needs change.

Some New Underpinnings

This new scheme could not reasonably be implemented in one great leap. Rather, it has two stages. The first is implementation of XVT for character-based windowing. Later, a ported Presentation Manager for CTOS is to make its appearance.

Porting Presentation Manager to CTOS requires that CTOS itself offer facilities that have never been part of it before. Presentation Manager was initially designed to run with Microsoft's OS/2. Aside from its associated applications and utilities, it is a collection of dynamic link libraries. Presentation Manager relies on semaphores and requires demand-paging for memory management. Thus, semaphores, demand-paging, and dynamic link libraries are being implemented in CTOS itself to support the new GUI.

It is important to note here that CTOS is still and will be a primarily message-based system. Developers should write to that model in order to get the greatest benefits from CTOS. It will certainly be tempting for some people to use the new semaphore facility to continue developing programs in ways to which they are more accustomed. This kind of development probably cannot be prevented, but it would result in mixed-style products that could not readily play in the distributed CTOS world.

As usual, CTOS developers are implementing these new facilities, not as direct copies from another system, but in ways that fit gracefully into the design of CTOS and will provide CTOS with new paths in the future. In Chapter 14, we shall take another look at those ideas.

10

Data Storage and Access

CTOS is, above all, a superlative platform for distributed applications. The file system and data access methods were designed to support that orientation.

Data storage under CTOS is primarily hard disk storage. Floppy disks, tape, and CD-ROM are also supported. Here, we shall concentrate on hard disk file system technology and tools. (Floppy disk access is similar, but stand-alone, floppy-only systems are not part of the CTOS world.)

Before we plunge into structures and algorithms, we should pull back and think about the larger role and goals of the CTOS file system and data access methods. CTOS is, above all, a superlative platform for distributed applications. The file system and data access methods were designed to support that orientation. They do so principally in three areas:

- A simple, trustworthy architecture designed for optimal file access speed and disk reliability. Speed is achieved by the placement of key file structures in the center of the disk to minimize disk access time, hashing techniques, and file-caching in main memory. Reliability is ensured by the duplication of key file structures.

- A distributed file system and data base products that allow the user to distribute a data base over a cluster or network.

- Automatic Volume Recognition (AVR), by which CTOS can recognize and mount a uniquely named volume (disk) on any workstation in a cluster (local network) without any user interaction. This feature implies that if a workstation is removed from the local network, its hard disk can be moved to another workstation and simply used there, without any network reconfiguration.

Disk Storage and the File System

File Specifications

The file system has a fixed hierarchy of four levels: network node, volume (disk), directory, and file. A file's path is syntactically specified as follows:

```
{nodename}[volumename]<directoryname>filename
```

A node is a location in a CTOS Network. Each node may be a standalone workstation or the server of a cluster. A node is specified as a character string with a maximum length of 12 characters.

A volume is the physical media in a hard disk, or it is a floppy disk. Volumes on a cluster must have unique names, but volumes on different network nodes can have the same names. Like the node, a volume name is also a character string with a maximum length of 12 characters. The system recognizes a volume by its name, as distinct from the physical drive in which it rests. This capability is the basis of AVR. (The physical drives also have standard internal names that the operating system recognizes.)

A directory is a group of related files. Like the node and volume names, a directory name is also specified as a character string with a maximum length of 12 characters. In fact, the directory is actually stored as a list of file names with some additional information. When a file is added to the directory, the file name is added to this list; when it is deleted, it is removed from the list. A user cannot simply open a directory, as it is not recognizable to the file system as such. Because of this structure, the CTOS file system is flat; that is, nested (hierarchical) directories are not allowed.

A file is a linear collection of bytes that the system considers to be a unit. The name of the file is specified as a character string with a maximum length of 50 characters, which gives users a great deal of flexibility to assign file names with meaning.

The CTOS file system has been based on the same premises since its conception, although it has been substantially rewritten in the interim. The question of moving from this simple, four-tiered system to a UNIX-like hierarchical file system has been a matter of hot debate over several years, and some work was done toward this end at one point. Many developers have argued that no operating system can consider itself modern without a hierarchical file system; while vendors close to the end-user community point out that they cannot even get their users to create one new directory, much less to understand trees of them. (In fact, there is a movement within the industry to return to a flat type of file system for simplicity's sake).

Disk Structures and Reliability

The CTOS file system is highly reliable. Without the high level of reliability, distributed processing would be impossible. This reliability is achieved through the following capabilities:

- Duplication of volume control structures, ensuring that damage to a single volume control structure will not cause data loss.
- Ordered updating of volume structures, ensuring that the volume will not be corrupted by power failure, hardware malfunction, or software error.

A disk volume is formatted to contain volume-control structures. These structures allow the file system to manage (allocate, deallocate, locate, and avoid duplication of) the space on the disk volume. The control structures are created when the disk is first initialized, and as such, the size of each is static once the volume is initialized. Care must be taken in determining the size of the control structures to prevent a key disk resource from expiring prematurely (e.g., running out of fileheaders).

Figure 10-1 shows the file system structures in memory and on a hard disk volume (within one workstation, for simplicity) that allow the file system to do its work.

The volume home block (VHB) is the anchor of the file system structures. There are not one, but two copies of the VHB on the disk: the initial copy and the working copy. The VHB points to all the other disk file system structures. When a volume is mounted (which for a hard disk occurs at system boot), the VHB is copied into memory. The working copy of the VHB on disk is updated from the copy in memory as files are created and deleted.

The duplication of the VHB on disk is part of the strong reliability scheme in the CTOS file structures. If one copy becomes unreadable because of disk damage, the other is still available.

Among the various files and areas pointed to by the VHB, let us single out a few: the system image, the allocation bit map, the master file directory (MFD), and the file header block (FHB) area.

System Image

The system image is the disk-resident image of the operating system. We mention this file in order to show the importance of the VHB during the initialization of the system. During boot time, the boot ROM must access the disk to load the operating system into memory. The location of the VHB is important because the boot ROM accesses the same location on the disk no matter what type of disk may be present (for example, a 20Mb disk versus a 140Mb one). The initial VHB is located at a predetermined location on track 0. The boot ROM looks at this location to read the initial VHB. The boot ROM can then load the system image into memory based on the address of the system image file specified within the VHB.

Allocation Bit Map

The allocation bit map represents each sector on the disk by a single bit. If a bit is set, that sector is available for allocation. The file system uses this structure in determining where new file extents can be placed. The size of the allocation bit map depends on the size of the disk volume.

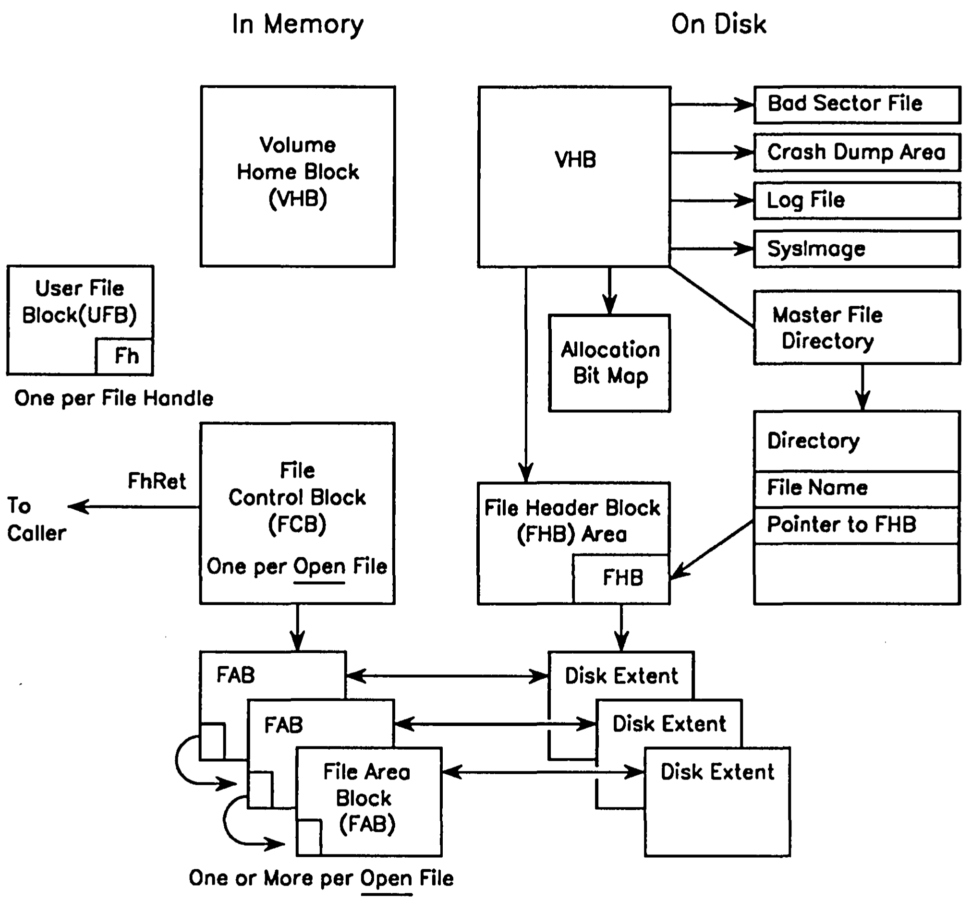


Figure 10-1. Volume Control Structures and System Data Structures

Master File Directory

The Master File Directory (MFD) is essentially a file, <Sys>Mfd.sys, that lists all the directories on a volume. This file is created at volume initialization and is not expandable. The file must, therefore, be originally created to hold the maximum number of directories that will be needed. Each MFD entry points to a disk area containing the directory information for that entry. Additional information associated with each entry includes the password and protection level for the directory and the maximum number of allowable files.

Remember, that the directory is really a list of files. The directory consists of one or more directory sectors. Randomization (hashing) determines the directory sector in which a file entry is entered. Included along with entry name is the file header block index, which points to the specific file header block within the file header file, <Sys>FileHeaders.sys.

The MFD and directories provide for fast, efficient access to the file header block (FHB) for a specific file.

File Header Block

The fileheaders file, <Sys>FileHeaders.sys, contains an FHB for every file on the disk. Within an FHB is all the information associated with an individual file, for example, the size, protection level, password, creation date, and so on.

The FHB, in turn, points to all the disk extents that make up the file. A disk extent is a contiguous group of one or more disk sectors containing disk file data. Thus, a file may be composed of an arbitrary collection of sectors.

In addition to the duplicate copies of the VHB, there are complete sets of duplicate copies of the FHBs (an option specified during volume initialization). These are on different disk sectors. The structures are updated when the primary versions are updated. Again, the probability of loss of information caused by disk damage is minimized by this design.

Frequently accessed structures, including both primary and secondary copies of the FHBs, are located near the physical center of the disk. This placement protects them from edge damage and also minimizes disk arm movement to provide excellent performance.

Structures in Memory

On the memory side of the figure, we see the in-memory copy of the VHB. Also shown is an example of a file control block (FCB) and several file area blocks (FAB), each of which describes the physical structure of the file. When a file is opened, an FCB is created for that file; and a FAB is created for each disk extent of the open file. These structures enable rapid I/O because once the file is open, the file system no longer has to go back to the disk-resident FHB for additional file information.

Figure 10-1 does not show every structure or detail pertaining to the file system, but it gives us enough information to support a cursory description of some file system activities.

File Manipulation

Files are handled by a request-based system service that is simply referred to as the "file system." The file system actually consists of several processes, the key ones being the File System Process and the MassIO Process.

When a program requests that the file system create a file, the file system first verifies that a volume (disk) of the requested name is already on-line by examining the VHB in memory. Following pointers to the MFD, it then verifies that a directory of the requested name is on that volume. Moving on to the directory, it verifies that a file of the requested name does not already exist. Having satisfied these requirements, it allocates an FHB and assigns the requested number of disk sectors by consulting the allocation bit map. Finally, it inserts an entry for the file into the directory.

When a program requests opening a file, a similar path is followed to the directory, where the file system verifies that the file does exist. The file system then allocates one FCB in memory, along with one or more FABs. It then copies information from the FHB to the FCB and each of the FABs. Finally, the file system returns a file handle, which identifies the FCB, to the caller for use in subsequent calls pertaining to that file.

Since a file handle now exists, the caller of the Open request can now issue requests to write or read sectors of the file, in addition to other operations. It is at this point that CTOS provides a unique performance optimization by utilizing a separate process, the MassIO process, to perform the disk read and write operations. Note that in the scenario presented here, the file system process handled the creation of the file, including the allocation of the disk space, and also the opening of the file. In most system environments, read and write operations are requested much more often than the opening and closing of files. Thus, the system should be optimized for reading or writing the files themselves.

This is implemented as follows: when a request to read or write a portion of the file is issued, the file handle is used. These requests are routed to the MassIO process thus bypassing the file system process. That is, requests for simple file I/O do not get queued with requests such as CreateFile or OpenFile that are relatively time consuming. Note that creating a file could cause several disk I/O operations to occur itself, in addition to the verification and search times as explained in the previous paragraphs. Read and write requests are queued only with other read and write requests, and serviced by the MassIO process, thus optimizing the response time for requestors of these operations. This permits a higher level of throughput for file manipulation operations.

When the MassIO process receives user requests, it may break them up into smaller I/O operations due to the physical structure of the disk medium. For instance, a user may request a read of a logical 64 Kbyte segment of data. This logical segment of data may in fact be represented by three different disk extents. This results in MassIO performing three different low level read operations, each corresponding to an individual extent. However, the data is returned in one continguous buffer to the caller, who does not have to be aware of this physical structure.

The CTOS file system is simple but fast, robust, and reliable. It is the best of its type, and it is one of the longest-lasting contributions of those early Ctosians whom we called the pragmatists in Chapter 3.

File System Access Methods

Figure 10-2 shows the two layers of file access methods. Underlying all the higher-level methods is the file system. Calling the file system, in turn, are the more frequently used SAM (sequential access method, or more commonly known as byte streams) and ISAM (indexed sequential access method) and the less often used DAM (direct access method) and RSAM (record sequential access method). As with other devices, you can either use a higher-level method for ease of programming or call the lower-level operations directly for flexibility and performance.

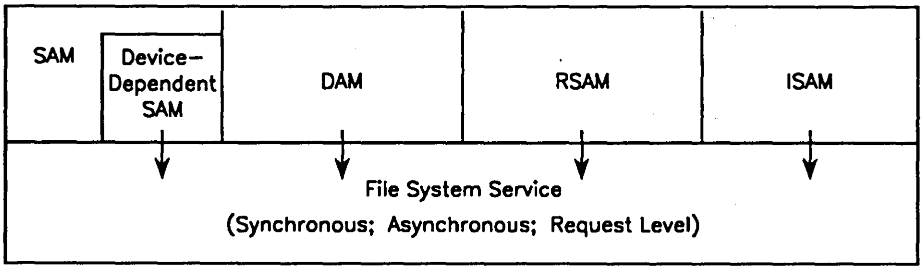


Figure 10-2. File Access Methods

File Management System Service

To perform I/O to a disk file with the file management operations, a program can use the following sequence:

Create the file
Open the file
Write data to the file and subsequently read the data
Close the file

In using the file system directly, you would call the CreateFile operation to create a file. The file name and password of the file to be created are passed as parameters, along with the initial size of the file. The latter is used for reserving disk sectors for the newly created file.

Once the file is created, it must be separately opened via the OpenFile operation. The name and password of the file are passed as parameters, along with the access mode (read, write, modify). This operation returns the file handle by which you must subsequently refer to the open file. All other file access operations require this file handle.

Once you have the file opened, you can write to or read from the file by using either synchronous or asynchronous I/O operations through the procedural interface (or at the most primitive level, you can construct request blocks directly). No matter which approach you use, you must know your file position (by sector) yourself and specify it with each call. At the lowest level of file system operation, that is, the Read and Write calls, the file system does not maintain the current file position; it is up to the application programmer to maintain this information.

Synchronous File Access

The easiest way for the programmer to interact with the file system is to use the procedural interfaces Read and Write to do synchronous file I/O. Synchronous I/O means that control is not passed back to the issuer of the I/O until the requested operation has been completed.

The Read operation transfers an integral number of sectors from disk to memory. The familiar file handle is used to specify which file is to be read. The target buffer's address and size are also specified as parameters, along with the address within the file (which must also be at an integral sector boundary). The operation returns the actual amount of data read.

The Write operation transfers an integral number of sectors from memory to disk. Once again, the file handle is used to specify which file is to be written. The source buffer's address and size are also specified as parameters, along with the address within the file. The operation returns the actual amount of data written.

Both the Read and the Write operations have synonyms: ReadFile and WriteFile. These synonyms are necessary when you are programming in a language where Read and Write are reserved words: for example, in the C or Pascal programming languages.

Asynchronous File Access

It is possible to have your program continue execution after initiating an I/O without waiting for the operation to be completed. If you use the ReadAsync or WriteAsync procedural interfaces, the file system initiates I/O, but your program can continue computation until a later point. The program then issues a CheckReadAsync or CheckWriteAsync call, at which point your program blocks until the I/O is completed. This type of mechanism is useful when you are implementing a double-buffering scheme within an application.

Closing the File

When you have completed the processing of a file, you close it using the operation CloseFile. This routine simply requires the file handle of the file that is to be closed. Note that closing a file does not update the end-of-file pointer. If a file has been extended, the end-of-file pointer must be updated to reflect the current status of the file with the SetFileStatus call. Note, however, that the End-of-File pointer is a logical pointer and does not affect the physical size or contents of the file.

Note that files are handled a little differently under CTOS. A maximum file length is specified when the file is created. You must keep track of the file length and explicitly extend it by using ChangeFileLength if the file grows beyond that maximum. That is, the file is not automatically extended when you write past the end. This optimizes file system resource utilization.

However, if you are using byte streams or one of the higher level access methods, the file is automatically extended as needed.

Sequential Access Method

The byte stream interface is simpler to use than the direct file system calls. When you call OpenByteStream, SAM calls the file system to create the file, if it does not exist, and to open it. ReadBsRecord and WriteBsRecord and their variations do not require you to restrict your I/O to sector-sized blocks. They read or write sequentially at the preexisting file position. To do random file access, you must use SetBsLfa, one of the two device-dependent SAM calls for file access. When you close a file using CloseByteStream, any needed file length changes are handled for you automatically.

Random access using byte streams is not as efficient as it is when you use the file management operations directly, because you do not have as much control over the amount of data being read. If you need randomization techniques, then there are several structured file access methods which provide for randomization.

The Structured File Access Methods

Besides SAM, there are three additional ways to access disk data within CTOS. All three involve data records instead of unstructured bytes. RSAM accesses a file that is a sequence of variable-length records. The other two methods, DAM and ISAM, access a file that is a sequence of fixed-length records. (In fact, DAM and ISAM can access the same data files.) Of the three methods, ISAM is by far the most commonly used.

DAM and RSAM are contained in the standard CTOS libraries of object module procedures. They allow their data files to be accessed by only one user at a time. ISAM, on the other hand, is a separate package consisting of an object module library, whose modules are linked into the calling application, and an ISAM system service, which allows more than one user to access a file at the same time. The ISAM system service, in turn, calls the file management system service to handle its actual file I/O.

Whereas RSAM and DAM use only the data file, an ISAM data set uses two files: the data file, containing the fixed-length records; and an index file. The index file provides a means of rapid access to information contained in the data file records. Because all records within a data file have the same length, disk management is simple and efficient: a record retrieved via the index allows the subsequent sequential retrieval of records.

ISAM allows the user to designate certain fields in each record as keys. For each key field, the index file contains pointers to records that are sorted based on the key field values. Suppose, for example, that all records are of the following format:

First name
Last name
Address

The index could contain pointers to the records sorted alphabetically by the contents of the "Last name" field .

Essentially, an ISAM data set is a DAM file with an extra index file for rapid access to the records, allowing multiple key storage and retrieval. In some cases, the size of the index file can be much greater than that of the data file.

If the ISAM system service is installed at both server and cluster workstations, ISAM users can access data files at both their own and the server workstations. In this way, ISAM supports distributed applications.

Other Data Base Approaches

ISAM is a good basis for distributed CTOS data base applications that will handle a moderate volume of transactions. Applications with greater volume needs can make use of the Oracle data base, which can run across a CTOS network with multiple nodes.

11

Communications and Printing

Beyond the CTOS cluster are software elements that can allow distributed applications to extend themselves over a large area, be it within a building or across the world. All these programs can be related to the Open Systems Interconnection (OSI) standard of the International Standards Organization (ISO).

Communications is a big subject. This one word can be stretched to cover any exchange between intelligent units: everything from getting a computer and a printer to cooperate directly, to running elaborate mail programs on top of standard protocols between unlike computers halfway around the world from each other.

Here we look at communications tools in the CTOS environment, the system software that lets you implement applications at any point along that continuum of complexity. Then we pick up one example of a system that depends on communications tools: the Generic Print System (GPS). Finally, we touch on some wide-area communications products associated with CTOS that also use these tools.

Cluster Communications

CTOS actually has two forms of communications. The local-area network that was built into CTOS from the beginning was originally called the cluster. Although this term is now falling into disuse with changing fashions in marketing terminology, we shall continue to use it here to make clear certain distinctions between cluster communications and the rest of the CTOS communications world.

The cluster originally was said to consist of a master (more recently called the server) and several cluster workstations (which were not slaves because they could function independently). A server workstation and its cluster workstations are connected via one logical multidrop line, and the server cyclically polls the workstations every 1/20 second to see whether they have requests for it. If the server has time at the end of a polling cycle, it repolls the active workstations before starting over.

Cluster communications code is part of the operating system itself. The request/response mechanism that we have so often mentioned works on top of the cluster software. Cluster communication is completely transparent to the application programmer: in Ctosian vernacular, "it just works". This fact is what sets the CTOS cluster apart from other small-computer networks available today.

We have revisited the cluster concept here only so that we can turn around and say that is not what this chapter is about. It is about the building blocks that make communication beyond the cluster lines possible, whether it is communication with a local printer or over X.25 networks around the world.

Hierarchy of Communications Tools

Figure 11-1 shows the three layers of communications tools. Of the three, device-dependent SAM for communications (SAMC) is by far the most important to the programmer.

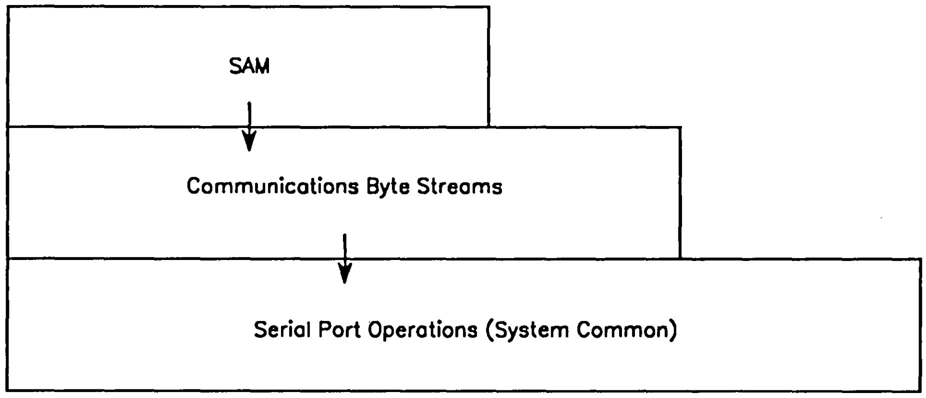

SAM
Communications Byte Streams
Serial Port Operations (System Common)

Figure 11-1. Communication Tools Hierarchy

In this familiar arrangement, serial port operations are the primitives in the fundamental layer. Some of the serial port operations are part of a system service. Others reside in system common for performance and so that they can be called by interrupt service routines (ISRs).

The next layer is SAMC, which is casually called "comm byte streams." It is device dependent; and unlike the situation with some other I/O hierarchies, it is a complete API, not only a set of extensions to SAM. SAMC resides in the standard libraries and is linked into applications. It calls the serial port operations.

Let us recall our earlier discussion on byte streams. Corresponding to the generic OpenByteStream operation is the device-dependent OpenByteStreamC operation. The difference between the two calls is that the first is a generic operation, while the second is specific to the utilization of the RS-232 serial ports. In this call, the device specification can be [COMM] or [PTR]. The first indicates a communications byte stream; the second, a printer byte stream.

At the top of the hierarchy in the figure is generic device-independent SAM, once again. For communications, any SAM routine always calls SAMC. SAM for communications is useful only in those cases in which output might go, for example, to a disk file under some conditions and out over a communications line under others. SAM also does not allow the application to overlap continued execution with a communications call, which SAMC does if the right routines are used. For these reasons, we shall focus the rest of our discussion on the serial port operations and on SAMC.

Serial Port Operations

The serial port operations are written so that the caller, whether it is SAMC or an application, does not incorporate into its own software any specific knowledge of different port addresses, clock frequencies, and so on, that are specific to different machines. Thus, programs that use these operations do not have to be relinked to run on new hardware types. The serial port operations also make raw interrupt handlers compatible with protected-mode CTOS.

The operations include routines that assign the caller to a communications channel or reset the channel for use by someone else. Other routines include setting up the DMA controller to transmit or receive data, reading or writing status values, and manipulating the baud rate.

Communications Interrupt Service Routines

Communications occur through channels, which are external devices; and all external devices interact with the operating system and applications through interrupts. The operating system takes an incoming interrupt and determines from a table (the Interrupt Descriptor Table in protected-mode systems) what interrupt service routine (ISR) should get control in order to take care of the event that the interrupt signals.

Under CTOS in general, an ISR (also called an interrupt handler) is part of a larger device-handler program. The other part of this program is called the device handler process. The two parts of the device handler program split the work of handling the device itself and the client who wants to use the device. Figure 11-2 shows that the device handler process is on the "client end" of this chain and the interrupt handler is on the "device end."

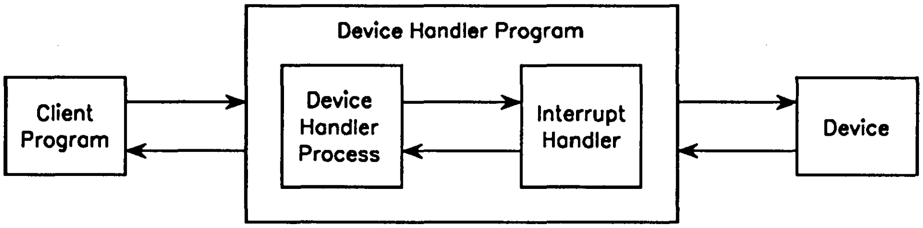


Figure 11-2. Overview of Interrupt Handling

Especially in the case of communications handling, interrupts cannot wait around too long to be taken care of; otherwise data can be lost. Thus the interrupt handler specializes in taking care of things promptly and quickly so as to be ready for the next interrupt, while the device handler process is called into action less frequently to, say, empty a buffer that the interrupt handler is filling, or to handle a request from or a response to a client program.

Most operating systems have external interrupts and methods of handling them. What is interesting about the CTOS method is that the interrupt handler and the device handler process communicate with each other by using interprocess communication (IPC) primitives directly (as well as optional shared memory). They can do so because they are parts of the same program, even though they execute asynchronously, as if they were two processes. The flow of IPC is unidirectional: only the device handler process can perform a Wait, while the interrupt handler can perform the PSend primitive (a variant of the IPC Send). Thus, the device handler process Waits at an exchange either for a request message from a client wanting a service, or for a PSend message from the interrupt handler representing status or data. The device handler process is both a clearinghouse for information related to the device and the agent responsible for determining what the device should do next.

Communications ISRs (interrupt handlers) are built on calls to the serial port operations.

Asynchronous or Synchronous Communications Applications

At the risk of seeming to belabor the obvious, we should clarify some terms, because we are about to use the same word to mean two different things.

Two types of protocols can be used by communications application programs. In synchronous communications, clock signals are synchronized between sender and receiver, and data is transmitted according to fixed time intervals. In asynchronous data transfer, there is no regular or predictable time relationship between sender and receiver.

Different communications ISRs are needed to support synchronous and asynchronous communications under CTOS. The existing communications I/O tools shown in Figure 11-1 support asynchronous communications programs. If you need to write a synchronous communications program to run under CTOS, you must write your own communications ISRs, calling the serial port operations to do so. CTOS systems documentation explains how do write these ISRs.

Communications Byte Streams (SAMC)

SAMC is a set of standard library routines for device-handling. When you link these routines into your application, you make your program into the device-handler program we spoke of earlier. SAMC contains most of the interrupt handlers required. However, to make applications hardware independent, a few interrupt handlers are system common procedures in the operating system. (Because a system common procedure is executed as part of the calling process, this arrangement does not violate the need for processes to be within one program in order to use IPC directly.)

The device-dependent interfaces of SAMC provide a more powerful and flexible set of services than those available at the level of SAM. Although it is more complex to use than SAM, SAMC comprises a complete set of services and can act as a replacement for SAM, provided that only communications and no other device types are being supported. Used in this fashion, SAMC is a general-purpose device driver for asynchronous RS-232 communications. It can form the heart of virtually any communications product except those that use synchronous communications protocols. Both half- and full-duplex communications are supported efficiently, with a variety of line control and data editing options. Among other conveniences, using SAMC frees you from writing interrupt handlers.

SAMC has been optimized for very high performance. It directly uses the task-switching facilities of the recent Intel microprocessors. It has become the basis for CTOS networking beyond the cluster level, as well as for printing services and other applications that require serial communications.

Overlapping Execution

We were at pains to define asynchronous versus synchronous protocols above because we also need to talk about asynchronous programming in another sense. As we saw when we discussed system services in Chapter 7, a program that makes a request for a service can wait (block) until it receives a response before continuing execution. This program behaves in a synchronous manner, which is the default when a request procedural interface is used. If the program makes a request and goes on executing while that request is being processed, later checking to see whether there is a response, this program is behaving asynchronously. To achieve asynchronous behavior (overlapping execution with I/O, for example), you usually must build the request block and issue the request primitive yourself.

Communications byte streams contains a duplicate set of operations for the two purposes. Synchronous behavior (blocking, or nonoverlapping execution) occurs when you use interfaces such as FillBufferC, FlushBufferC, and so forth. For asynchronous behavior, you must use a variant set of interfaces with analogous names: FillBufferAsyncC, FlushBufferAsyncC, and so on. Thus, to achieve asynchronous execution in a communications program using SAMC, you do not build the request block yourself. The asynchronous operations include additional parameter options that allow the caller to specify what SAMC should do it if needs to wait before the operation can be completed. As an example, one option provides using the IPC primitive PSend to send a message to a caller-specified exchange when completion becomes possible.

A SAMC Customer: GPS

As we saw in Chapter 8, both old and more recent printing methods coexist in the CTOS world. The old methods have been preserved for backward compatibility with venerable applications that have not reformed their ways, and there is not much point in our elaborating on them here. What is more interesting is to look at the Generic Print System (GPS), to which most applications that print are now written, and which is a client of SAMC as it communicates with printers.

GPS is a complex product that consists of a collection of system services and libraries, not all running on the same workstation, which encapsulate the various tasks of printing in a modular, device-independent manner and permit applications to request printing without containing any printing code themselves. Further, these applications can output a generic stream of formatting commands that will be interpreted specifically for the destination printer chosen.

Overview

Figure 11-3 shows a simplified diagram of the components of GPS. The central element and traffic cop is the Print Service, which handles routing and spooling of print jobs. The Print Service directs jobs to various device drivers, which are system services themselves, each handling a type of printer. The printers together with their device drivers may be located on the same workstation, on other cluster workstations, or across the network on other nodes from the Print Service. The Print Service "knows" all the printers on its own cluster by name, and it needs a node specification to find a printer on another node. (Once it has found such a remote printer, the Print Service lists and retains this knowledge.)

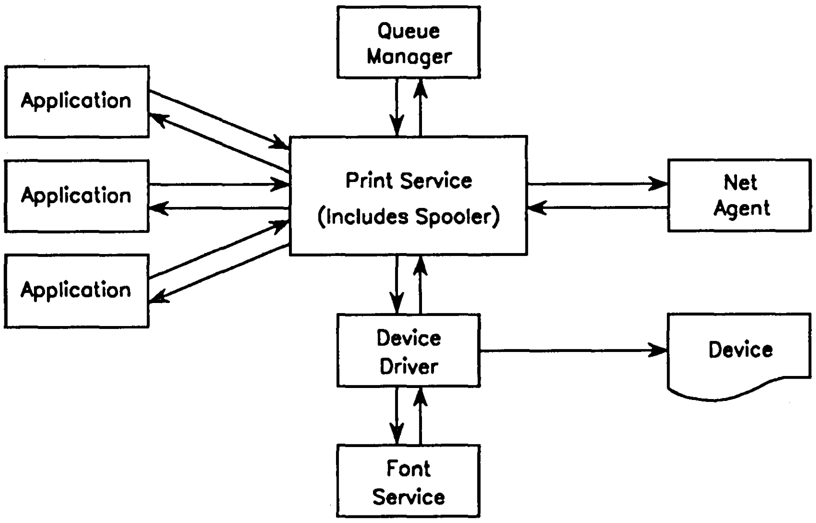


Figure 11-3. Generic Print System

Cooperating in this picture are the Queue Manager, a CTOS system service that is not part of GPS, and the Font Service.

On the application end, there are three choices. An application can use GPS byte streams, or make calls to a byte-stream-like library called the Generic Print Access Method (GPAM), at the lowest level make direct requests via procedural interfaces to the Print Service.

GPS Byte Streams

GPS byte streams is a set of device-dependent SAM routines that are not included in the default configuration of SAM that comes with CTOS, but can be configured in when SAM is built. The GPS byte streams interface is the familiar set of generic operations: OpenByteStream, WriteBsRecord, CloseByteStream, and so on, where the printer name is specified as the target device. GPS byte streams are simple to use and quite adequate for utilitarian applications, but they support only characters, line feeds, and form feeds, and no special formatting.

Generic Print Access Method

GPAM is essentially a page-description language. It is a library that can be called to include formatting and graphics into the outgoing stream of data from an application. GPAM inserts generic formatting commands and in turn calls GPS byte streams for most tasks, although it does make a few direct requests to the Print Service to set parameters.

Calling the Print Service

Applications can make requests of the Print Service through the request procedural interface. An application cannot put formatting into its document by making these requests: in fact, they usually are made for printer control and status information only. Applications can and do call both GPAM and the Print Service, the first for formatting, the second for status information.

The Print Service

In its routing function, the Print Service receives the print request, locates the printer device driver, spools the job if necessary (with the help of the Queue Manager), and finally sends it on to be printed. The Print Service then monitors the printing process.

To route print requests between network nodes, the Print Services on the two nodes interact with the net agents on their nodes. All the mechanics of passing a print request across the network are transparent to the application that submits the print job.

GPS Device Drivers

When a GPS device driver receives the stream of data associated with the print request, it interprets the embedded generic formatting commands to the highest level that it can for the printer it controls and then forwards the stream to the printer. Formatting commands that are too sophisticated are interpreted with the closest approximation possible on that printer, but are not rejected.

Many GPS device drivers are available from various vendors to support different printers including PostScript® printers. In addition, there is a device driver developer's kit. The core of this kit contains control routines and a set of rasterization and vectorization routines to interpret GPAM commands; the developer adds printer-specific output routines.

Wide-Area Communications

Beyond the CTOS cluster are software elements that can allow distributed applications to extend themselves over a large area, be it within a building or across the world. All these programs can be related to the Open Systems Interconnection (OSI) standard of the International Standards Organization (ISO). This well-known, seven-layer OSI standard is shown in Figure 11-4.

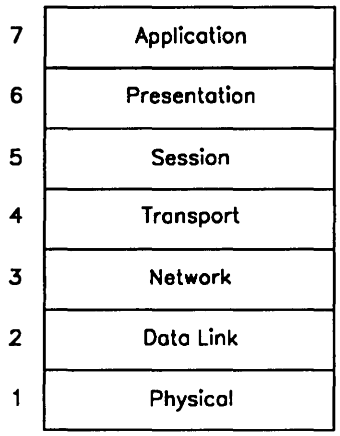


Figure 11-4. The Seven-Layer OSI Model

The CTOS Network: BNet

The most important piece of the communications picture from the point of view of distributed applications is the network that connects clusters together. Interrelated with CTOS itself, the CTOS Network carries the message-based architecture to a wider span, allowing applications access to other nodes through the request-response mechanism. The application need only specify the node name to work across the net.

The various OEM versions of CTOS also have various names for the CTOS Network. We will use BNet here as the example.

BNet is composed of several cooperating system services. A simplified overview of the parts of the network is shown in Figure 11-5.

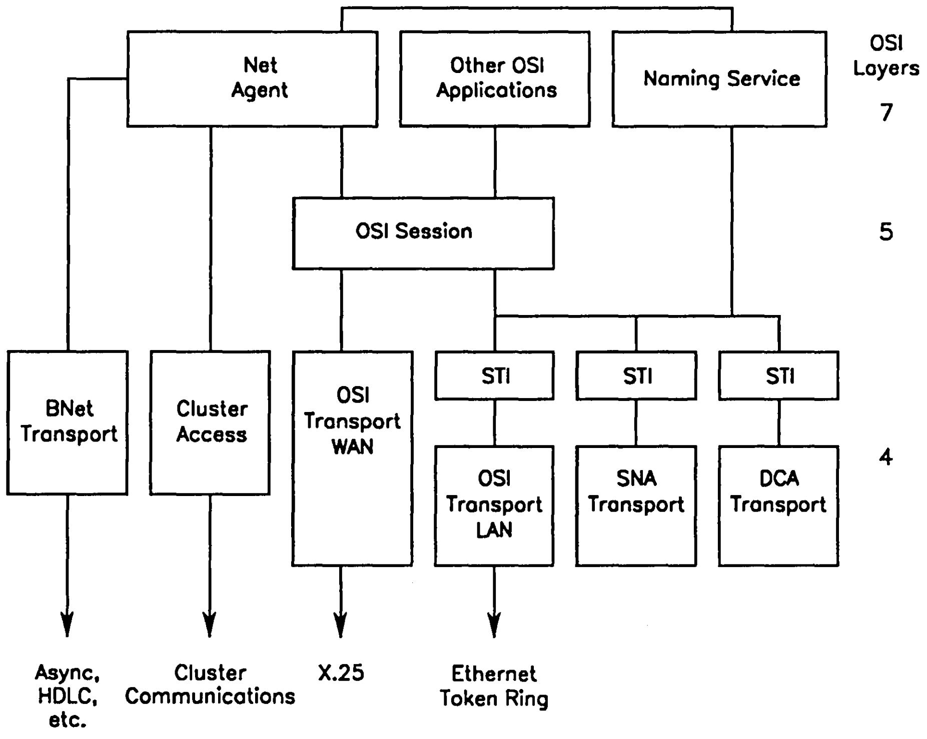


Figure 11-5. BNet Block Diagram

BNet provides bridge processing among heterogeneous networks, thus enabling intertransport communications. BNet architecture is limited only by the underlying transport and system environment. It provides network independence and an open, standard interface to facilitate future expandability. It supports unlimited simultaneous outstanding requests and unbounded simultaneous logical connections.

BNet is a point-to-point routing system. If sending and receiving nodes are not directly connected by the physical medium, other nodes act as intermediaries by relaying the data packets from node to node between sender and receiver.

The top layer in Figure 11-5 corresponds to the application layer (layer 7) in the OSI reference model. At this level are the Net Agent and Net Server, which are two processes contained in one system service. We referred to that one system service in Chapter 7 as the Net Agent. The Net Agent and Net Server processes pass requests and responses to and from client application programs. Other OSI applications, such as X.400 Message Handling Service mail programs, are shown at this level also. They are not part of BNet, but they can utilize BNet to distribute services. Finally, the System Management Services are shown in this top layer. This system service provides administrative functions and a Naming Service. The other application-level components can use the Naming Service to find unidentified network nodes for which they have requests.

There is no need for an OSI presentation layer (layer 6) within BNet because there are no interface incompatibilities to be bridged at this point.

Session control is layer 5 in the OSI model. From this point on, BNet offers more than one option in the construction of a communications stack, partly for historical reasons. One important component at this level is the OSI Session Services system service. It interfaces with both the Net Agent/Net Server and the other OSI applications above it. It also communicates with transport layers below it.

Figure 11-5 also shows the Net Agent/Net Server communicating with two other entities: the CTOS Network Transport and Cluster Access. The CTOS Network Transport provides a pathway for lower-level communication via synchronous and asynchronous media. Cluster Access allows the server workstation or any cluster workstation to communicate with any other on the cluster. Effectively, CTOS Network Cluster Access thus adds an optional layer of peer-to-peer communication on top of the unidirectional cluster network.

The OSI Session Services layer communicates with a range of system services called Session/Transport-LAN Interfaces (STI). These STIs manage various transport backbone types, including SNA and DCA. Extension of this mechanism to other transport entities is possible. The OSI Session system service also can communicate directly (that is, without an intervening STI) with the OSI Transport WAN to interface with X.25 public data networks. SNA and DCA interfaces allow the user to run BNet services over an existing WAN backbone, lowering the cost of CTOS networking.

Beyond

Auxiliary communications software products, including an SNA Network Gateway and 3270 terminal emulator, are available to interface CTOS-based workstations to the Systems Network Architecture (SNA) designed by IBM Corporation. This software allows the workstations to communicate with IBM mainframe computers.

Other communications software products are also available from a variety of vendors to interface to most mainframes and minicomputers.

12

Prototype Until Done: Timekeeper Development

In the CTOS world, unlike some others, one generally does not write application prototypes that are thrown away when the "real" code is written. The more common method is to write a "prototype" that really is the core of the product, successively refining it . . . Our work on Timekeeper involves designing two major components: the local user's interactive application and the Reminder system service.

In the CTOS world, unlike some others, one generally does not write application prototypes that are thrown away when the "real" code is written. The more common method is to write a "prototype" that really is the core of the product, successively refining it until it is in shape to be released (a process called stepwise refinement). This first version includes as many existing pieces as possible: there is no need to recreate the wheel.

Timekeeper's Components

It has been some time since we talked about Timekeeper back in Chapter 5, so let us quickly review the design decisions we have already made about it. Knowing more about CTOS I/O methods, we can now pick the ones that are right for this project.

Timekeeper is to be a workgroup-oriented application that keeps calendars and To-Do lists for group members, sending them reminders of events and deadlines. It will allow users to check each other's calendars and schedule meetings, with Timekeeper doing the work of matching up time slots and finding an available conference room with the right amenities. It will also allow users to send each other electronic messages.

User Interface

We shall design an interactive application to run locally at the user's workstation, to accept inputs and return outputs, and to communicate with the system services. Our best bet is to start right out using the new XVT bridge tool. This approach will give us the greatest compatibility with future CTOS systems.

XVT has a dialog box editor that enables us to prototype the user interface interactively and to test run it separately from the application of which it will be part.

XVT will handle keyboard, mouse, and video for us. Thus, for Timekeeper's interactive application, we do not need to get involved with keyboard or video byte streams or with the I/O methods underlying them: keyboard primitives and VAM/VDM. When we write the Reminder system service, we shall also write a utility to deinstall it. This utility will put up simple screen messages to a screen inherited from the Executive. Thus, video byte streams are a perfect choice for use in the deinstallation utility because they will do the job with minimal effort on our part.

Data Storage

Our data storage (calendar, meeting room, and To-Do list data) must be centrally located so that it is accessible by all users. Of the methods we discussed in Chapter 10, ISAM is clearly the most suitable. It is based on a system service that can handle contention by users for the same data resources, and its performance is excellent for the volume of transactions we expect.

We could also use one of the two existing CTOS-compatible versions of Oracle, but it is not the best choice in this particular case, because ISAM is faster for the kind of work we expect to do. Oracle is optimized for very high transaction volumes, at which level ISAM cannot perform as well.

As a result of this decision, we could use the file system directly for data access, or we could use disk byte streams for access to configuration files. Direct file system calls may be more efficient than the byte stream calls, in addition to being easier to use. However, byte streams provide device independence and the easy redirection of input and output.

Reminder Service

We decided in Chapter 5 to design the Reminder Service ourselves, although Queue Manager would do the job for us. (Remember that we would really like to use tools that are available without resorting to replication of existing material, but that we are taking this route in this case simply for illustration's sake.)

Mail Service

Our Reminder system service will pass user communications from the interactive application to the existing Mail Service for delivery to other users. The Reminder need only call the Mail Service API to do so. Mail notification can be put into an application so that it displays an indication of new mail when mail is received in a mailbox. Here is a prime example of taking advantage of software that is already available. The entire communications backbone is available, allowing a rapid implementation using preexisting tools.

Networking

In most other development environments, we would have to devote considerable time and effort to make our program interface correctly with separate network software–maybe of more than one type. Under CTOS, this part comes for free. If we set up our requests correctly, we do not need to give networking another thought.

To allow for eventual implementation of Timekeeper over wide-area networks, we simply need to route our requests by file specification, including the node-name component.

Printing

We cannot leave I/O methods without discussing everyone's favorite topic, printing. We could just write to GPS byte streams, but although that would be quick, it would not give us any special formatting. If users are going to want to print out their calendars with nice grids of lines or special fonts, we will need to write to GPAM anyway, so we may as well start out by doing so.

Native Language Support

CTOS has quietly acquired a large installed base around the world, much of it in countries where the language is not English. Thus, in the course of CTOS history, it became very important to make it easy for other people to convert CTOS to their native languages without plowing through reams of code, rebuilding the operating system, and risking the creation of new bugs in doing so.

NLS (Native Language Support) is the facility by which both systems and applications developers under CTOS make conversion simple. Using NLS has two aspects. First, certain procedural interfaces that support language conversion must be used in preference to other, older CTOS procedural interfaces. Second, messages that will be displayed on the screen must be segregated into a special disk file for easy editing. This file can be a traditional CTOS message file; or if you are using XVT, it can be the file in which the XVT resources (such as fonts, dialog boxes, icons, and text) for that program are kept.

We bring up the subject of NLS at this point because it is much easier to code an application with the correct NLS calls the first time than to comb back through the code and retrofit it with internationalization. This is especially true with the message file, because hard-coded screen message strings must not be used.

NLS Mechanism

NLS is based on a set of tables that define such language dependent elements as date and time formats, number and currency formats, collating sequence, keycap legends, and so on. These tables are in a system file on disk. When the operating system is booted, these tables are loaded into a special memory area. (Alternatively, if you are going to need to support more than one language at a time, you can link the tables into your program.)

The tables are in an assembly language file that can be edited, assembled, and linked, without change to the operating system. Thus, translators need touch only this external file and need not rebuild CTOS or an NLS-based application.

The other half of the mechanism is a set of standard library object module procedures that internationalized programs must use. These procedural interfaces are easy to recognize by the 'Nls' string embedded in their names: NlsFormatDateTime, NlsNumberAndCurrency, NlsYesNoOrBlank, for example. These procedures refer to the NLS tables to determine what currency symbol to use, what string means "yes" or "no," and so forth.

Message Files and XVT Resource Files

In addition to the NLS facility, you can use the message file or XVT resource file to internationalize your application program. Using this facility, you remove the messages from your applications and place them in the appropriate message file. If you are using the traditional message file, you do not link the message strings with your program, either by hard-coding them or by putting them into a separate module of the program. If you are using the XVT resource file, the compiler associates the correct objects with your program. As a result, your program code remains language independent.

A traditional message file actually exists in two forms: text and binary. You create your messages in text form. (The translator also later translates them in this form.) Then you use a simple command to convert the text file to a binary file so that the messages can be more quickly accessed by your applications. In your program code, you use the message operations (InitMsgFile, GetMsg, PrintMsg, and alternates, which are in the standard CTOS libraries) to display the messages. These routines are built on video byte streams.

In the text file, messages are in a format that can be easily edited and converted to binary by nonprogrammers. This fact saves expense during translation efforts.

As with the NLS procedural interfaces, it is far easier to start off using the message-file technology than to retrofit later.

Interactive Application

Our work on Timekeeper really boils down to designing only two of the major components: the local user's interactive application and the Reminder system service. We are assuming here that you know already how to write applications in general, so we are going to focus on the CTOS tools that you would use to implement the necessary elements of such an application system.

We shall start with the interactive application.

Basic Design

The main duties of the interactive application (IA) are accepting user commands, figuring out what is wanted, making the necessary requests of other services, and displaying video output to the user. Figure 12-1 shows how the IA is related to other pieces of Timekeeper and system software.

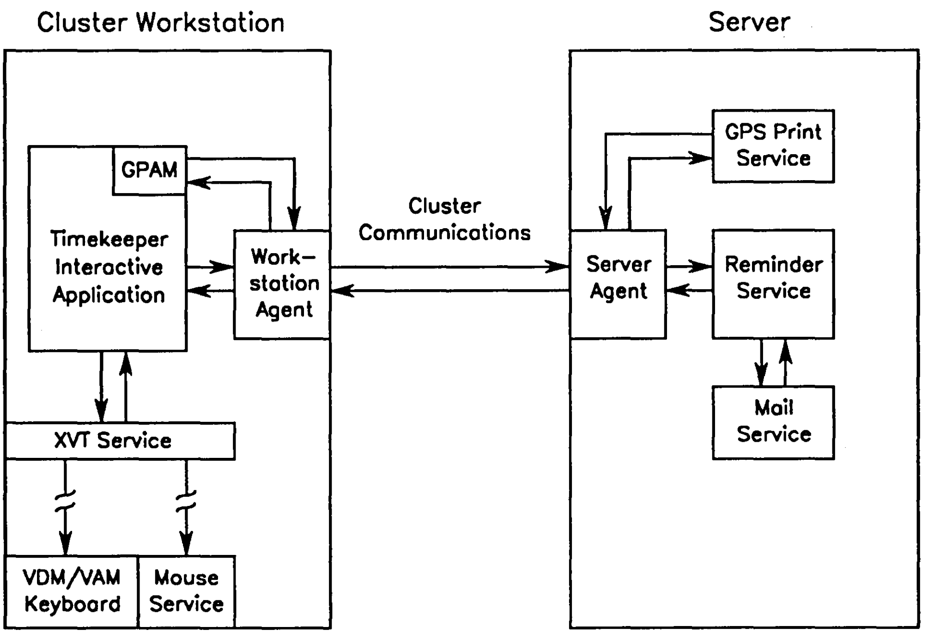


Figure 12-1. Relationship of the Interactive Application to Other Software Entities

One of the most important parts of the IA is the user interface. As we have already mentioned, we shall use XVT's dialog box editor to create the various elements of the user interface. We shall place all screen messages in the resource file so that they can be easily nationalized. XVT itself makes NLS calls where that is necessary, so we do not have to concern ourselves with those aspects of internationalization.

Requests to the Reminder Service

One big thing that the IA will do is make requests of the Reminder system service, which in turn will deal with all the other system services in the Timekeeper application system. Although defining the requests that the IA will use is really part of developing the Reminder system service, we are going to discuss them here, because it is primarily the IA that will use the requests.

Reminder interfaces with the IA via a set of requests that are constructed solely for the use of this application. What types of requests would be necessary here? The most obvious is the abort/termination request, which must be handled by the Reminder service (even though is not issued by the IA, but by the operating system when the application exits). In addition, requests to CreateReminder, ReadReminder, ModifyReminder, and DeleteReminder are obvious. Not so obvious are session-oriented requests. We shall define two of these, OpenReminder and CloseReminder. These session-oriented requests allow multiple IAs (perhaps from different cluster workstations) to utilize the Reminder service.

Before we show the formats of these requests in terms of procedural interfaces, we really need to know the mode of operation of the requests. A Read of a reminder will return a single reminder to the calling application. But how is the application structured? The application may issue reads for all the reminders for a specific day or all the reminders for a specified priority. The number of reminders returned depends on what information is desired. Writes, however, are different. A write will be issued either because of an update or because a new reminder is being sent. Both the reads and the writes are valid only if the session handle returned by the OpenReminder operation is included.

To initiate a session with the Reminder service, an OpenReminder call is issued. All further Reminder operations require that this session handle be passed as part of the parameter strings. The session handle logically establishes a connection between the IA and the Reminder service. Multiple connections are possible with each connection identified by the session handle.

To create a reminder, the IA issues a CreateReminder call. This call accepts the address of the data area containing the reminder information and the address of the memory area where the unique record identifier (URI) is returned. The URI allows the user to modify a given record and store it back in the data base with a ModifyReminder operation; this gives us the ability to go directly to the record by using the underlying data base access routines.

To issue a read to return a reminder, a ReadReminder call is used, where the session handle is passed, along with the address of the data area where the reminder is returned. Additional parameters are the priority, the date structure, and the memory address where the URI will be stored. Why are these parameters necessary? The IA displays reminders based on priority or based on the date. These two data elements are also keys within our ISAM data base. This choice of parameters makes the connections among the IA, the Reminder, and the ISAM data base simpler.

To delete an existing reminder, a DeleteReminder call is issued. The Reminder handle is passed, together with the URI of the record being deleted.

To issue a write to either create a new reminder or update an existing reminder, a WriteReminder call is issued, where the Reminder handle is passed along with the reminder record. Additionally, a URI may be passed (valid if an update to a previous existing record is being performed).

To close a Reminder session, a CloseReminder call is issued. Once this occurs, the connection between the client and the Reminder service is discontinued.

Note that each of the operations mentioned above are very much like high-level I/O operations with defined procedural interfaces allowing easy programmatic interface. The operations could be object-module operations that would translate the user's requested function into low-level file access methods. However, since we are attempting a distributed application under CTOS, we shall use loadable requests to implement the interfaces. Reminder is the system service that will serve these requests, thereby making the service available across the network to any user requiring programmatic access to the Reminder Service.

Relationship With the Mail Service

The IA will not deal directly with the Mail Service, but will communicate with it through the Reminder. We shall discuss the interface between Reminder and the Mail Service later in this chapter. The calls to the Mail Service allow us to utilize preexisting communications routines for the automatic routing of messages.

Printing Through GPAM

The IA will need to send print requests and to provide for special formatting of Reminder documents, such as calendars and meeting notices. It could also provide printer status information to users. For this implementation, let us assume, however, that users would use the Generic Print System's Print Manager utility to check on status.

When the user issues a print request to the IA, it will use a GPAM data stream to send the document for printing, adding commands to describe the sophisticated formatting we need to provide calendar rules and special fonts. GPAM communicates with the Generic Print System much like a byte stream. Calls to GPAM's object-module procedures are used to described page formatting, fonts, and graphics.

The the IA uses a call to GPAMOpen to open the data stream, specifying the document to be printed, special characteristics of the print job, and both the familiar work area buffer and an additional buffer for GPAM procedures.

Within the data stream a series of calls to GPAMs formatting routines describe the characters to be printed and special formatting to be supplied. Graphics routines can also be embedded in the data stream between calls to GPAMBeginGraphics and GPAMEndGraphics.

The Generic Print System's Print Service, located at the server workstation, routes the data stream to the specified printer anywhere in the cluster or local-area network.

What makes the Generic Print System unique is that applications may prepare device-independent print output that is automatically and transparently routed across the network. Special device drivers are supplied for each printer supported for use with CTOS systems. These drivers specifically accept print requests with GPAM formatted files, or simple ASCII files sent through GPS byte streams, and translate the contents into the device-specific information needed to print the file on that printer. The device driver is installed only on the workstation to which the printer is attached.

Thus, Timekeeper formats output in a generic way, and can print on any supported CTOS printer. The code in our application is kept small. The resources required to process the request are distributed, taking up memory and disk space only where they are needed.

Reminder System Service

Our other major effort goes into the Reminder system service. To keep things simple, we shall initially design a single-process, synchronous system service. In the real world, this approach would be adequate as a prototype to enable us to get the product up and running and see how it worked. Once we began to have multiple users actually competing for data, we would need to consider the methods described in Chapter 13 for writing a more sophisticated system service.

Reminder, being a system service, has a specific structure that is mandated by the types of operations required. It is a straightforward implementation with a simple loop (wait until something is received; then perform the requested operation). Before we describe the loop structure, though, let us look at how the service is installed.

Installation

First we perform a call to GetPartitionHandle to see whether the service is already installed in another partition. The Reminder service partition has a unique partition name, and GetPartitionHandle will return the partition number for the partition name requested. We do not want to attempt another installation if Reminder is already running.

If no error results, Reminder is already executing, so we exit with an error message indicating such. An alternative method could be utilized by performing a QueryRequestInfo operation passing one of the Reminder requests as a parameter to determine whether the request is presently being served.

Next, we allocate any permanent resources to be used by the Reminder (for example, exchanges or short-lived memory).

Then we query the status of all the requests to be served by Reminder.
We must serve the CreateReminder, ReadReminder, ModifyReminder, and DeleteReminder requests, as well as OpenReminder and CloseReminder. We query their status with a call to QueryRequestInfo for each request. A status code of 'No such request code' returned by this call indicates that the request has not been served and thus that we can serve it. A status code of 'error OK' returned would indicate that the request is served by some other process and would complicate matters somewhat, forcing our system service to filter the request: but this is another matter. Reminder will exit if a nonzero status code is returned.

Once the requests have been served, we issue a call to ConvertToSys. This operation changes all processes, exchanges, and memory in the partition from application status to system service status. In Ctosian terms, the service at this point becomes an extension of the operating system.

Next, we issue a call to Exit. This operation causes the reload of the exit run file (normally the Executive). The Reminder service is now permanently installed in memory as a system service.

To ensure that the service can complete its required activities in a timely manner, Reminder then calls ChangePriority. The priority specified should be a higher priority than all interactive applications (which normally run at priority 80h).

Next a call to SetPartitionName is issued to identify the partition. Here the partition number (0, indicating the current partition) and the name of the Reminder service are passed as parameters. Remember the earlier step in which we checked to see whether the partition was already installed? The call checked for the same partition name that we used within this SetPartitionName call.

Next, we make a ServeRq call for each of the requests to be served (in our case, CreateReminder, ReadReminder, and so on).

Finally, the system service can go into its wait state, waiting for something to do. When a request is then received by the Reminder, it will process that request.

Deinstallation

Deinstalling a system service is not a trivial task. (Deinstallation is different from the kind of termination we have talked about earlier. Termination refers to the condition when an application program, or system service client, is trying to cease execution.) Deinstallation takes place in three phases. First the deinstallation utility (a separate program from Reminder) sends a predefined message or even a deinstallation request to the system service. An agreed-upon, 4-byte message from the deinstallation utility to the reminder service is all that is necessary unless an eventual filter is required If a request is required, then Reminder must be modified to serve and respond to this request. The deinstallation message informs the service to shut down its operation. Second, when the service receives the deinstallation message, it must perform a sequence of operations. Finally, when the operations are completed, the service responds to the deinstallation message and the deinstallation utility cleans up what is left.

Once the Reminder receives the deinstallation message, it must check for open connections with client applications (in this case, the IA). Reminder refuses to deinstall if there is an open connection.

If Reminder is in a state where it can deinstall, we must restore the request table in the operating system to its pristine state (the state it was in before Reminder was installed) by issuing a ServeRq request for each of the requests with the same information originally received in the request information structure. (Note that we had to save this information before the Reminder initially issued its own ServeRq calls.) We must issue the calls in reverse order to that in which they were originally issued. (The connection-opening request should be "unserved" first.) This approach will ensure that the service receives no new requests while attempting to deinstall.

What do we do if we have any outstanding requests? We respond to all of them with an appropriate error message (all except the deinstallation request). Thus we ensure that no requests are lost when deinstallation is complete.

Next, Reminder closes any connections it has opened as a client (for example, those with the Queue Manager, Mail Service, and ISAM Services).

Now Reminder unlocks its partition by calling SetPartitionLock. This operation allows the service to be removed from the partition. Reminder then responds to the deinstallation message, effectively informing the deinstallation utility that Reminder is ready for removal.

Once all these steps have been completed, the Deinstallation utility then issues an ExitAndRemove, which causes the service and its partition to be removed.

Reminder Loop Structure

Remember that a system service waits for a request and performs some amount of processing based on the request before finally responding to the request. What does this structure look like within Reminder? First, let us ask from whom we expect messages: the user IA, the deinstallation utility, Mail Service, and Queue Manager. We receive responses from the last two in response to request primitives issued from Reminder itself. Why do we issue primitives? Because we do not want to block. (The procedural interface causes a process to wait until the requested service is completed.) We do not want to wait until a mail message is received (we may wait forever) or wait for a queue entry to be removed, so these operations are implemented with the primitive Request rather than via the procedural interface.

Calls to ISAM need not be implemented via primitives, because they will be issued immediately when access to the data base is required. We cannot respond to a ReadReminder, WriteReminder, or DeleteReminder request until the ISAM operation is finished, so we do not care if we block on ISAM. In addition, operations that use the Queue Manager (with updates caused by Writes and Deletes) can be done using the procedural interface for the same reason.

Reminder waits at its exchange, the memory location where the pointer to a request block is returned. Reminder then keys off the request to determine what to do next.

If Reminder receives an OpenReminder request, the service initializes a session for validating incoming requests from the IA. A CloseReminder request basically invalidates the session (although invalidation is dependent on completion of all outstanding Reminder requests for that session).

If Reminder receives a CreateReminder, ReadReminder, WriteReminder, or DeleteReminder request, the system service then issues a series of requests to ISAM to perform the action requested by the user from the IA. CreateReminder, WriteReminder, and DeleteReminder also require an update to the ISAM data base and to the Reminder Queue File, so Reminder then performs these actions before responding to the CreateReminder, WriteReminder, or DeleteReminder request.

Queue Manager responses are received when reminders become due. In our simple system, we notify the user of a due reminder by causing a short repetition of beeps. The user responds to these beeps by running the IA to read the reminder.

Mail Service responses are received when reminders are received from remote sites. These messages must then be stored in the ISAM data base and the Reminder Queue.

Interactions

In addition to interacting with its clients, each copy of the Timekeeper interactive application on the users' workstations, we have seen that the Reminder Service will interact with the ISAM system service, which may be on the same server or across the network. For simplicity, we shall consider it to be on the same server. Reminder also must get system time information from the operating system. Further, Reminder interacts with the Mail Service. Figure 12-2 shows these relationships. For simplicity, we omit the local and server agents and operating system involvement in the transfer of requests and responses.

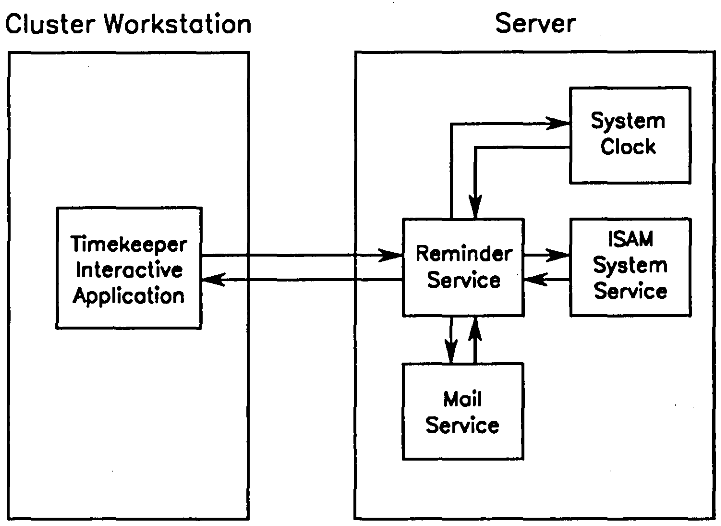


Figure 12-2. Reminder Service as Service and as Client

Interacting With ISAM

The following steps show a very basic method for interfacing to ISAM. This methodology is very simple; a real implementation would be more complex. Remember that ISAM itself is a system service available to multiple clients.

To access ISAM, we issue a VerifyMultiUserISAM call. This call sends a request to ISAM at the node where the application system is running. (In a local area network where clusters are networked together, if ISAM is not local to this node, then the request is simply passed to the node where it is resident.) A nonzero status returned indicates that multiuser ISAM is not available.

If multiuser ISAM is not available, we issue a LoadSingleUserISAM call where we pass the ISAM run file specification and password and the ISAM configuration file specification and password as parameters along with the status block. This call loads ISAM as a task and initializes communications with ISAM. Memory is allocated as short-lived memory from the pool of unallocated memory available to the application system. (Remember that in this case the partition must be large enough for both the application and single-user ISAM.)

We next issue an OpenISAM call to open the ISAM data set. A single call must be issued for each ISAM data set. In our case, since we have a rather simple data base, only one OpenISAM call will be issued. The call requires the data set name as a parameter and returns a handle to be used for all subsequent ISAM operations.

When records are read sequentially by keys, we would use the following procedure: We first issue a SetUpISAM-IterationLimits call to initialize a sequence of read operations for records that have keys for a specified index (for example, all class A To-Do items). We then repetitively issue ReadNextISAMRecord calls to retrieve all the records in key order from the data set. A unique record identifier (URI) is returned for each record. Selection of a displayed record on the To-Do list for deletion or modification requires the URI for that item.

We issue a BeginTransaction call to mark the start of a transaction for the application system. ISAM has a concept of a transaction definition that allows for the definition of transactions that must be completed in totality. This allows the programmer to ensure that a sequence of low-level ISAM operations are completed successfully.

To store a new record in the data set, we call StoreISAMRecord. The indexes are updated to reflect the presence of the new record with the StoreISAMRecord operation.

To modify or delete an existing record, we first use a ReadUniqueISAMRecord call to read the specified record identified by a given key. We then use the key with a following ModifyISAMRecord or DeleteISAMRecord to modify or delete the existing record. Again, all indexes are updated accordingly. With the delete operation, all data in the record is destroyed.

We issue a CommitTransaction call to complete the transaction. Any records that may have been locked are unlocked with the commit.

When we are done we must close the ISAM data set. Issue a CloseISAM call to close and release all the resources associated with the open data set. Once this call is completed, the ISAM handle associated with the previous data set is no longer valid.

As you can see, each of the operations allow a simple interaction between a data base client and the ISAM system service. Since the interfaces are all requests, the ISAM service need not be local to the requesting application (in our case, Reminder). This also shows how a service itself can be a client to another service.

Interacting With the Mail Service

Reminder interfaces to the Mail Service to distribute reminders through the network. Included are the passage of new reminders to be stored within the ISAM data base and the passage of due reminders to the user.

The following steps show only the basics of interfacing to the Mail Service. In a real implementation, you would have to consider other application-specific details.

To send mail messages and attachments from the data base, we use an InitVm call. This call initializes a memory buffer for use by the Message Facilities. We must do so before using any of the other facilities.

To open a mail connection with the Mail Service, we use an EstablishMailConnection or OpenMailConnection call. A mail user name and mail password are parameters for these calls. These parameters are retrieved from an installation parameter (from the VLPB), from a user file (from the MailCenterName and MailUserName entries), or via a hard-coded value. These calls return a parameter, the mail handle, which is used for all subsequent mail operations. In addition, the path ([volume]<directory>) indicating the location of the Mailbox directory is returned. The OpenMailConnection also accepts an additional parameter for specifying this type of mail connection (for example, long-lived, or sending-mail only).

Next, an InitMailMsgBuffer call is issued to initialize a message buffer for creating mail messages.

To create a message, we issue successive PutMsgComponent calls to construct a mail message one field at a time within the previously allocated message buffer (for example, the From and To fields).

Once the message is assembled, we issue a CheckPointMsg call. This call returns the size of the message for subsequent use in the SendMail call.

Next, we make a ReleaseMsg call to release the buffer so that the message can be sent.

Finally, we use a SendMail call to begin the delivery process. This call instructs the Mail Service to send a copy of the message to the designated recipients.

A unique message ID is returned that identifies the message as it passes through the mail system. The message is automatically routed to the mail centers indicated by the names in the To field.

To receive updates in a mail message and/or attachment and incorporate them into a data base, we issue a sequence of calls similar to that we used for sending mail messages. We first use the InitVm call to initialize a memory and then open a mail connection via an EstablishMailConnection or OpenMailConnection call. Next, as before, we make an InitMailMsgBuffer call.

To retrieve the message components, we issue successive calls to GetMsgComponentById to decode the message from its binary format into individual fields and components.

Once the components have been retrieved, we release the buffer by calling ReleaseMsg. Next, we make a call to AcknowledgeMailReceipt to inform the Mail Service that all parts of the message have been properly retrieved. The Mail Service can then delete its copy of the message.

The steps from ReceiveMail to AcknowledgeMailReceipt are repeated until no more mail is available. We can then issue a TerminateMailConnection to the Mail Service to close the mail session.

Again, note that we are interfacing our Reminder Service to another system service. Any process, whether it be an application or a system service can be a client of any other process. All that is required is the issuing of a request.

The Whole Picture

Figure 12-3 shows many of the interactions among the various pieces related to Timekeeper. To simplify matters, we leave out local and server agents. We show only one user instance. We also do not deal with various configuration files and other temporary files that would be present. The point here is to see how a distributed CTOS application really ends up consisting of multiple system services interacting with each other, all through the request/response mechanism independent of the underlying network topology.

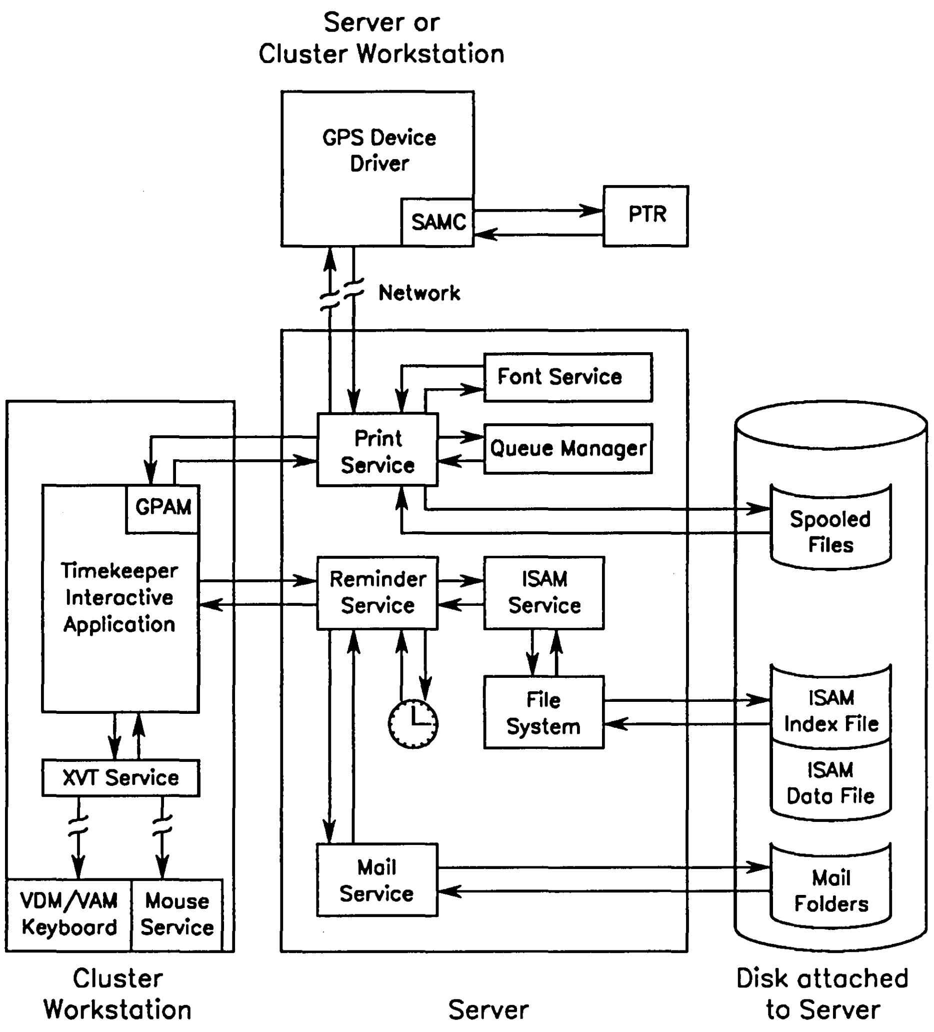


Figure 12-3. Timekeeper and its Services

13
Even More About System Services

CTOS is flexibly featured and permits a variety of approaches. There are as many strongly held opinions about the right way to write a system service as there are talented CTOS developers. Here we shall pick a general path that many agree on: a single-process, asynchronous system service. Later we shall examine a very different and less commonly used method that involves multiple processes.

One of the great virtues of CTOS is that it can be customized and extended. You can do so by writing a system service and either substituting it for an existing one or adding it on as a new one. You can also write a filter process, a system service that intercepts messages headed for another system service and either examines them and passes them on or serves them itself.

Because system services are built on the message-based CTOS interprocess communication, they can be transparently distributed across the local cluster network and almost transparently across wider networks. This factor eliminates a lot of network programming from the process of writing a major piece of distributed software.

CTOS has been successfully extended for special purposes in this way by developers all over the world. One example is the POSIX system service, which allows POSIX-compliant applications to execute under CTOS. Another is the Cluster File Access system service, a filter process written to enable the server workstation to access files at cluster workstations thereby providing basic peer-to-peer communications capabilities for sharing files.

Real-world system services like these are complex and interesting to design. This chapter is about some of these challenges and solutions.

The Multiclient System Service

Except for some hints now and then, we have so far discussed the system service in its simplest form: the single-process, synchronous system service that takes a request from a client, processes it, and responds to the waiting client. Many sophisticated system services, however, face more complex demands than this simple model can handle successfully.

Many system services have multiple clients concurrently sending requests to them across the cluster and perhaps the network. In turn, system services often make requests of other system services or interrupt handlers and must await, receive, identify, and deal with their responses. Every system service also must be able to handle termination and related requests made by the operating system when an application client wants to terminate. These termination requests may arrive at any time and must be handled promptly so as not to cause delays throughout the network.

The Situation

Figure 13-1, which borrows from our Reminder system service example, shows one situation of this kind. Three user-interface clients have sent in various requests to Reminder. Reminder is servicing the first request. The other two are waiting, queued at Reminder's service exchange. While they are waiting, these user interface programs are blocked.

On the "back end," Reminder has made a request to the ISAM Service for data. The ISAM Service in turn has sent a request to the file system. The ISAM Service has also responded to a previous request that Reminder had made, and this response is queued at Reminder's exchange.

In the middle of all this normal business, an application somewhere on the cluster is terminating. The operating system has sent out termination requests on its behalf. One of these termination requests is also queued at Reminder's exchange, and it is fourth in line for attention.

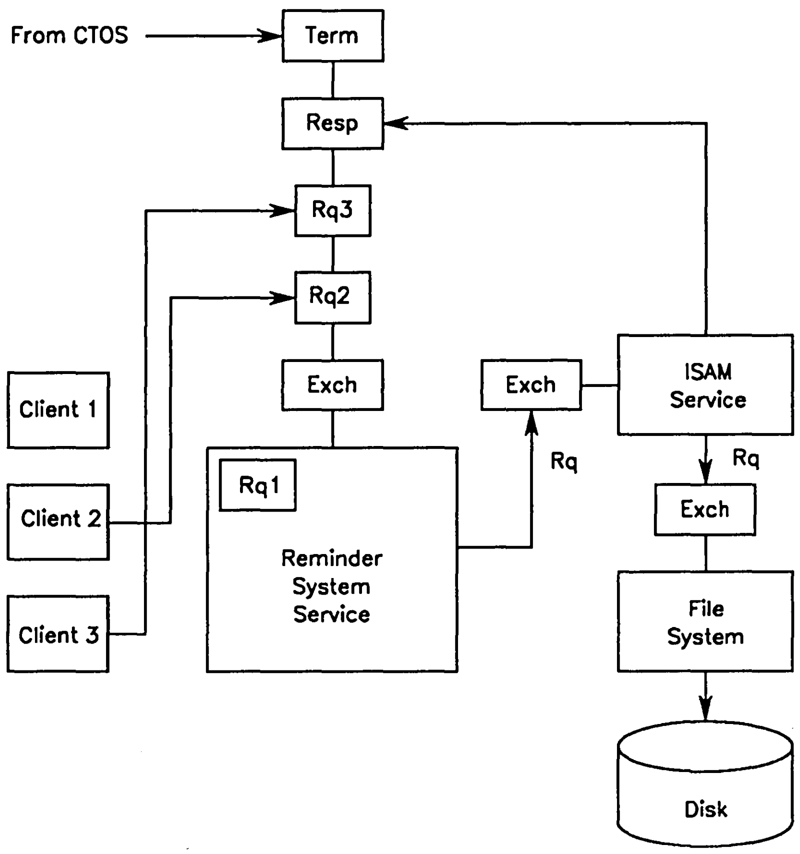


Figure 13-1. Multiple Demands on a System Service

Potential for Delays

In the situation shown here, delays can snowball if the system service cannot somehow interleave all these demands for its attention. If Reminder can service only one request at a time (Rq1 in the figure) and cannot respond to Client 1 until this work is complete, you can see that all other queued requests and responses must wait.

Although this case is not represented in Figure 13-1, suppose that Reminder were to make all its back-end calls to other system services synchronously: that is, Reminder would make the call to ISAM and then would block while waiting on a separate response exchange for ISAM to respond. The queued items at the service exchange must wait also. If ISAM itself could handle only one request at a time, delays would proliferate. (Of course, ISAM does not operate in this way.)

Ending It All

The picture becomes even more interesting when we consider what happens among system services when an application on the network is normally or abnormally terminated or is swapped out to disk by its local operating system. This application may have requests outstanding with various system services. A system service thinks of a client in terms of its user number (its partition) rather than by any unique identifier. Thus, if a system service were to respond to a now nonexistent client, a protection fault would occur because the address of the request block is no longer valid. If an LDT entry does perchance exist for the address of the request block, the response would be sent to the client, however, the client most likely would have a protection fault in trying to address the PbCb pairs within the request block. On such an occasion, the operating system broadcasts to all system services a request asking them to wind up their business with the victim neatly (usually by responding with a special status code). System services then must respond to the operating system so that the operating system can continue the termination, abort, or swap.

Now suppose, the system service is off blocking while its own request is being served elsewhere. If all system services operated this way, it could be a very long time indeed before the operating system got back all the needed responses to a termination/abort/swap request and could proceed. Meanwhile, users sitting at inexplicably hung systems all over the network would be doing user-like things such as continually pressing the GO (transmit) key until the system finally responds.

As one developer puts it, the fastest way to make enemies in the CTOS world is to mess up termination.

An Event-Driven Model

Recall that we have stated that CTOS is an implementation of the event-driven model of processing. Within this model, whenever an event occurs (e.g., the completion of an I/O operation) rescheduling occurs immediately, provided a process is eligible for execution. An event-driven program, may never finish once it is running. After starting, it may receive any number of different kinds of inputs or requests in any order, and it must examine these inputs and choose paths of execution based on their nature. It is a state machine that works on multiple tasks at once.

Clearly, this event-driven or state-machine model is far more appropriate for a sophisticated system service than is the traditional application written to function within a single tasking operating system. There is nothing unique to CTOS here: an operating system must deal with its world this way. Other operating systems such as OS/2, the Macintosh operating system, and Novell® Corporation's NetWare® network operating system all are faced with the same problems and use event-driven solutions. The beauty of CTOS is the clean separation of the applications and the system services by the messaging mechanism. This mechanism is what allows the CTOS event-driven solution to work in a single workstation or transparently across the network.

Blueprint for an Asynchronous System Service

CTOS is flexibly featured and permits a variety of approaches. There are as many strongly held opinions about the right way to write a system service as there are talented CTOS developers. Here we shall pick a general path that many agree on: a single-process, asynchronous system service. Later we shall examine a very different and less commonly used method that involves multiple processes.

This asynchronous system service has one process and one exchange. At this exchange it receives all its inputs. Thus, the single exchange is both its service exchange and its response exchange. After receiving an input at this exchange, the system service decides what the input is and identifies its source (a new request, a response from a back-end request, termination from the operating system). The system service then runs through decision code (for example, a case statement) to choose a path for further action.

All the back-end requests that this system service makes to other system services or to interrupt handlers (if it is managing a device) are asynchronous. That is, the system service makes these requests but does not Wait for the associated responses. Rather, it continues by returning to Wait at its one exchange, thus immediately picking up the next item that is queued there.

Why this insistence on asynchronous requests throughout? A gremlin that immediately pops up where calls among system services are not asynchronous is the deadly embrace (or deadlock). A deadlock is the infinite wait that occurs when system services call each other in a circular manner. Such a circular type of situation is shown in Figure 13-2, where System Service A calls System Service B, and B calls System Service C, but C in turn calls A.

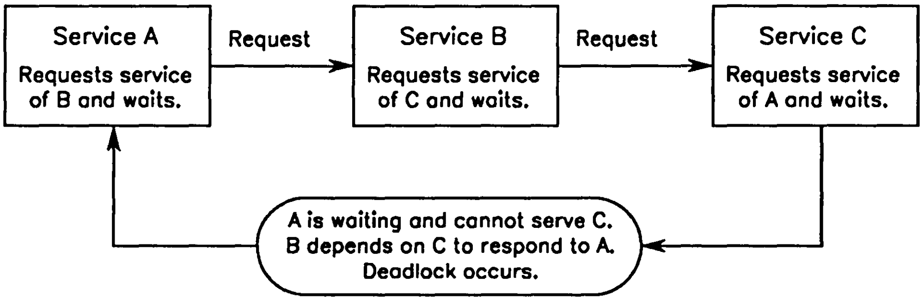

Service A
Requests service of B and waits.
Request
Service B
Requests service of C and waits.
Request
Service C
Requests service of A and waits.
A is waiting and cannot serve C.
B depends on C to respond to A.
Deadlock occurs.

Figure 13-2. Deadlock in a Request Chain

If you are using any synchronous requests, the only way to reduce the possibility of deadlock is to structure the pattern in which system service calls are made to be like the tree structure in Figure 13-3 rather than the circular structure in Figure 13-2. Although it may seem simple to ensure that this is the case, in practice it is not. Deadlock is the most common problem that occurs during development of system services. For example, a potential deadlock may be hidden where a system service in the chain makes a call to a library that in turn makes a call to a system service higher in the chain.

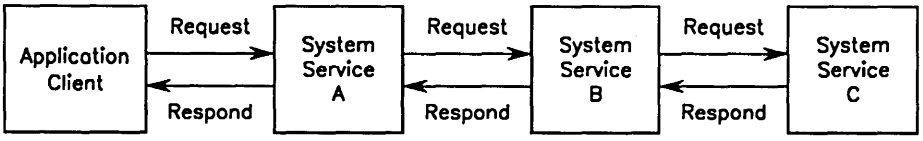


Figure 13-3. Deadlock Avoidance with Synchronous Requests

Asynchronous system services decrease the possibility of deadlock and improve the overall responsiveness of the system. However, system services that use asynchronous processing are more complex to develop. Whether you use asynchronous processing or not, simple data flow diagrams displaying the path of messages through the system will help ensure rapid detection of a deadlock condition.

Client Bookkeeping and Data Structures

Because an asynchronous, single-process system service handles requests from multiple clients at one time, it must do careful bookkeeping as to the status of each request and of its own back-end requests. To do so, it most commonly assigns a data structure for each client request and perhaps one for each back-end request. These data structures must be designed from the beginning in such a way that all states can be represented and that termination status can be shown for that client when necessary.

When the system service picks up a response from a back-end call, it identifies the client for whom the call was made. The system service then follows the trail of status information through its bookkeeping data structures in order to know what to do next for that client. In this way it can pick up the context of any client at any time.

Handling a termination request in this scheme often involves placing it on an internal queue to wait for back-end requests on behalf of that client to return. During this time, only that client must wait. When the response does come in, the system service dispatches a routine to pick up activities on behalf of that client. This routine immediately determines from the client data structure that the client is being terminated. It returns a special status code to the client and then returns to the body of the system service. The system service then responds to the operating system's termination request.

A Basic Set of Requests

As we saw in designing TimeKeeper in Chapter 12, there is a basic set of requests that the developer defines for most system services. These include connection oriented requests such as Open, Read, Write, Close, and QueryStatus types, and the group of termination, abort, and swapping requests. If conventional formats are used in the design of these requests, the application interface to the system service is standard, and the system service is easily networked both within the cluster and within a wide-area network. System documentation from the various suppliers of CTOS usually describes the mechanics of request design.

When to Use a Separate Process

One cannot always maintain a strictly one-process approach. Sometimes it is necessary to add another process. One such case is where intensive CPU activity is required: for example, in a data base search or a long sort. There is a voice processing system service, for example, that does data compression on the voice. This activity is done by a separate process.

In such a case, the main system service process runs at a more favorable priority than does the CPU-intensive one. The main process hands off work to the second process (via a direct IPC Send) and then goes back to Wait at its exchange. The advantage is that the main process can act on any termination or other request that it receives, bumping the second process from the processor. Thus there are no client or termination delays caused by long tasks.

In general, a system service should never work longer than a few milliseconds before going back to its main loop to Wait. On an 80386 microprocessor, this amount of time allows for execution of thousands of machine instructions. In this time, you can do almost anything other than large sorts or moving around quantities of data. Creation of a second process should be reserved for such cases.

The use of the second process often requires semaphore protection of a data structure on which the worker process is operating. Processing of another request from that same client, for example, could cause incorrect alteration of the data structure while the main process has control.

Development Tools

ServerGen

ServerGen is an available tool that is a fill-in-the-blanks template for a subcase of the asynchronous, single-process system service. It has been informally passed around and used by developers in many companies as the basis for successful system service implementations.

ServerGen segregates the standard parts of the system service from the parts that must be written uniquely for that system service. It is a good illustration of the initialization sequence that a system service must go through, as well as its decision-making processes.

ServerGen shows how the system service determines whether what comes into its exchange is a new request, a response coming back from another system service, or a termination-related request. It does so basically by comparing the response exchange named in the request block with its own exchange. If they are the same, then the message is coming back from another system service. If not, it is a new request (or termination).

The service has a simple loop structure once the environment has been established (an exchange has been allocated, the requests have been served, and the server has been converted to a system service). This loop structure in pseudo code is as follows:

```
do forever
   Wait for a message
   Process the message
enddo
```

The routine for a processing the message checks the message to determine what type of work is required of the service. In ServerGen, we have several different types of messages: Timer messages set for performing periodic functions, internal messages sent by the service to itself, new requests from clients, responses from requests filtered to other servers, and system requests (e.g., termination requests). Each request or message can be identified via the request block information. The pseudocode structure for processing a message is as follows:

```
if the Message is a Timer Block then
   Process the Timer Message
elseif the Message is Internal then
   Process an Internal Message
elsedo
   if the Request Block contains my Exchange then
      Restore the Client's Exchange
      Process the Response
   elseif the Request is a System Request then
      Process the System Request
   elseif the Request is a Client Request then
      Process the Request
   endif
enddo
```

The above pseudocode outlines what is done after receiving a message on the ServerGen exchange. Next, ServerGen must respond to a message which has been processed.

A message received from a client which is directly processed by ServerGen will be responded immediately back to the client process.

A message received from a client which is filtered can either come back to the ServerGen (in this case the response exchange in the request block must be replaced with the ServerGen's exchange) or can return directly from the filtered service to the client (in this case the response exchange is not replaced).

A message received from a filtered service must have the original client's response exchange restored prior to responding to the original client.

The following is some pseudocode outlining the cases described above:

```
do case Request Type

   ServerGen Request:
      Respond to Client

   One-Way Filter Request from Client:
      Forward the Request to the Filtered Service

   Two-Way Filter Request from Client:
      Save the Client's Exchange
      Insert ServerGen Exchange in the Request Block
      Issue RequestDirect to the Filtered Service

endcase
```

ServerGen does not handle all the situations that we discussed in the previous chapter. It was built as a template for a two-way filter process, and it assumes that the back-end call is to the default system service whose requests it is intercepting. ServerGen does not handle the more general case of asynchronous back-end requests. It can, however, give you an idea of how to handle them.

Appendix A consists of the main code sequences of ServerGen for those readers who would like to see an actual implementation of the asynchronous, single-process system service.

Asynchronous System Service Library

Taking a very different approach from ServerGen is the Asynchronous System Service Model. It is a method of creating an asynchronous system service based on a library (Async.lib) that handles the difficult parts of asynchronous programming for you. In some CTOS versions, this library is included as part of the CTOS system software.

In using the Asynchronous Model, you create a single-process program that behaves as if it had multiple processes. It provides greater throughput than would a simple single-process system service, but it greatly reduces problems with synchronization and access to shared data that are inherent in multiprocess programs. The design of the Asynchronous Model uses an implied baton or semaphore, although not a real one, that is passed whenever a back-end call is made. In this design, it is impossible for an execution thread to block while holding the "baton."

In this model, the system service handles more than one transaction at a time. Contexts are the key design element of the model. A context is an individual execution thread (but not a process) that has its own stack history (such as local variables). The system service process consists of multiple contexts, all sharing the system service's process stack.

Stack sharing is possible because each context has a unique stack pointer (SP) value. While a given context is being executed, the stack pointer is set to the appropriate value for that context. Before a second context executes, the stack of the first context is saved in a memory structure. Then the stack pointer value is changed to represent the stack for the second context. Because the stack of a context is saved, any context can be resumed where it stopped executing simply by having SP set to its stack pointer value.

In essence, the use of the shared stack is equivalent to having a stack for each context, or in other words, the shared stack emulates a multiprocess service. There is a parallel here with the use of bookkeeping data structures that we discussed earlier, but here the mechanism is hidden from the programmer.

The asynchronous system service of this model basically works as follows:

```
do while true
   Wait for a Request
   if the Request is mine then
      Resume the Context  /* a back-end call */
   else begin  /* a new request coming in */
      Process the Request
      Place the Status code in the Request Block
      Respond to the Client
      end
   endif
enddo
```

With the exception that resuming a context does not mean a process switch, you can see that the general logic is the same as what we have discussed before. Figure 13-4 shows the program flow for this model.

The system service waits for either a request or a response. If a request arrives, the system service can process that request. If the system service needs to send a request to an external agent, it does not wait for the response. Instead, it sends the request using one of the asynchronous request library routines. Within this routine, the context that sent the request is saved, and the system service process returns to the top of its Wait loop to wait for other requests or responses to arrive. If a response from an external agent (another system service) arrives, the context that originally sent the request is resumed. This scheme is analogous to what would occur if several CTOS processes were used instead of the contexts, except that a context does not give up control other than by making a back-end call.

The Asynchronous Model provides a common-code module in the C language that includes the code that must be written for any system service. In addition, you write a main module that serves the requests defined for your service. (In this respect, the Asynchronous Model and the ServerGen program in Appendix A are similar, but in other ways they are quite different.)

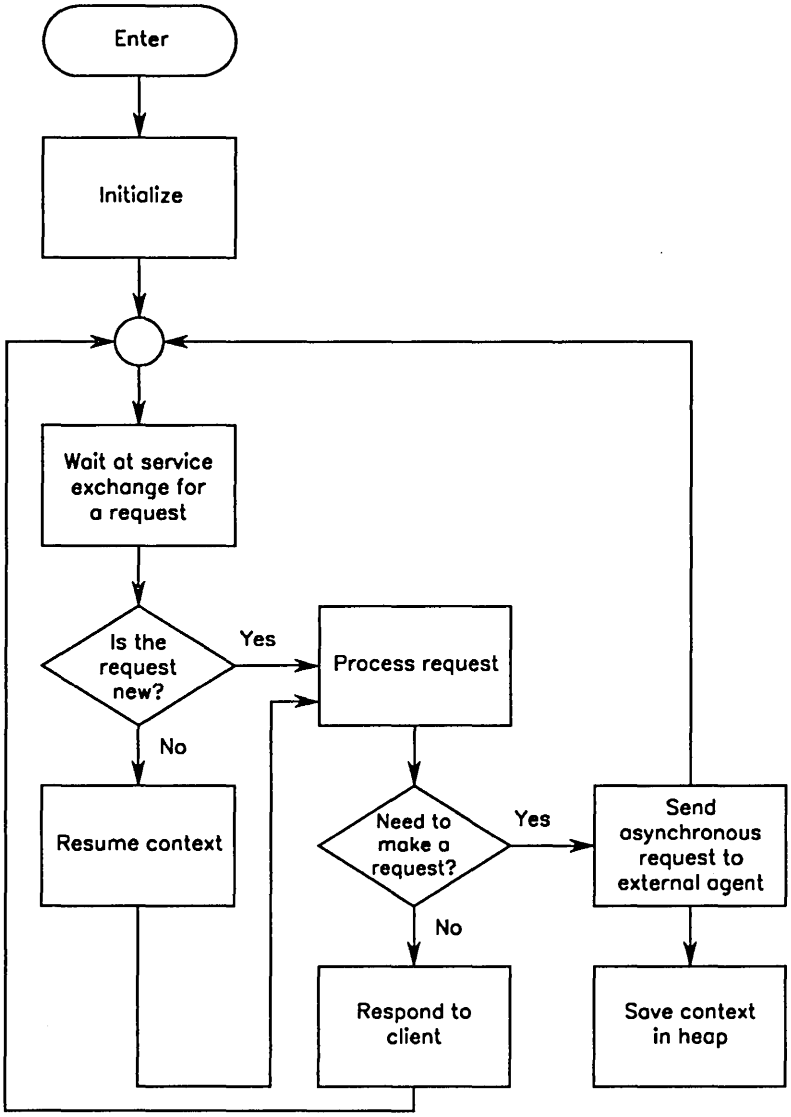


Figure 13-4. Asynchronous System Service Model

The Asynchronous Model has advantages and disadvantages. It was created in 1989 to underlie and simplify a complex multiprocess system service that had some unresolved problems. At this writing, it has not yet been widely used otherwise. Some CTOS theoreticians consider it a brilliant implementation of a tool that hides the complexity of system service writing from the programmer. Others, while not denying that the mechanism is valuable, believe that it should be internal to CTOS rather than existing in a system service. Nevertheless, this method might be the easiest one to use when porting to CTOS a program from another system that expects to use multiple processes.

Testing and Debugging

A final, purely pragmatic word of advice to system service writers from those who have written them is that you cannot expect to successfully develop a complex system service without a thorough understanding of the CTOS Debugger and the Intel microprocessor's registers. At this level of programming, print statements are not going to help you much.

Multiprocess System Services

As we mentioned earlier, there is another approach to designing a multiclient system service. This method, instead of using a single process and interleaving action on requests from different clients, uses multiple processes. Each process is synchronous and can serve any client.

While this approach might seem the most intuitively obvious one to, for example, the UNIX developer, the CTOS environment is quite different. For example, a second process, once begun, cannot be made to rejoin a parent process. CTOS processes also have a certain overhead in system resources. Overusing them can be expensive.

Why Write a Multiprocess System Service?

Despite their pitfalls, multiprocess synchronous system services have the advantage of being easier to conceptualize and visualize, easier to implement initially, and easier to maintain than the single-process, asynchronous system service. If designed correctly, they also have the great virtue of extensibility, as we shall see in a later example.

Why Hesitate to Write One?

Generally, writing a successful multiprocess system service requires such great ability to perceive possible difficulties and such immensely meticulous programming that it is something only a very careful expert should attempt. The creation of multiple processes brings with it multiple opportunities for deadlock. Also, several processes competing for access to the same data structures require the use of semaphore protection. It is possible to create processes that block while holding large semaphores. Finally, there is the process overhead and consumption of system resources that can balloon in an overly ambitious design. An example of this last statement was the initial release of the Generic Print System (GPS); later versions corrected the initial design.

It is especially important in writing a multiprocess system service to keep your initial design simple and to study your modularity carefully. You should plan on adding functionality only after the first design is proved workable.

Message-Based Semaphores

CTOS developers generally dislike the concept of semaphores (a message-based system really does not require them). However, some software designers appreciate the simplicity of use of semaphores. Simple semaphores fall naturally out of CTOS IPC. Suppose a program has three processes: two user processes and a system service process. The system service process owns a buffer. The two user processes are bidding for use of an output port.

At initialization, the program allocates an exchange, and an initializing message (which could be a pointer to the buffer) is sent to the exchange. (Note that we are using IPC directly here, because all these processes are part of the same program.) A user process that wants the buffer does a Wait at the exchange. If the message is present, that user gets the buffer. When this user is through with the buffer, it Sends the message to the exchange again. If the other user does a Wait in the mean time, it is blocked until the message arrives and the buffer is thus available.

This method generalizes neatly to multiple buffers. The messages identifying the buffers can be queued at the exchange, and a process that does a Wait gets the next available buffer. No confusion can occur: as long as one process owns a message, no other process can interfere.

Clearly, this method can be used to control access to entities other than buffers, too. The message used as a baton does not have to be a pointer: it could have any value or be a dummy such as 0.

This kind of semaphore, which you implement yourself based on IPC, should not be confused with traditional semaphore facilities inherent in the operating system, such as those in OS/2.

CTOS Electronic Mail: A Happy Example

Among many excellent office applications that have been developed at different companies to run under CTOS, arguably the best and most widely used is the CTOS electronic mail program, an elegantly simple, richly featured electronic mail system that encircles the world on multiple media in some installations. It is sold under various names including CT-Mail and B-Mail. Originated in 1982 and continuously extended since, the mail program is a very cleanly designed example of the multiprocess approach to solving the problems we have delineated. It also exemplifies some other good CTOS design principles.

Mail was begun with a deceptively simple architecture and highly generalized data formats. Nevertheless, a lot of thought went into creating hooks and leaving latitude for later developments that might or might not be on an existing marketing "wish list." The basic design has three parts: an interactive user application on the local workstations; a central Mail Service; and a Communications Manager Service that had to be written because network software beyond the cluster level did not yet exist in 1982.

A simplified version of the design of the central mail system service, the Mail Service, is shown in Figure 13-5. The figure shows three processes inside the box that represents the Mail Service: one nicknamed the Overlord and two Drones. Actually, there can be up to four Drones at once. Drone processes are identical to each other.

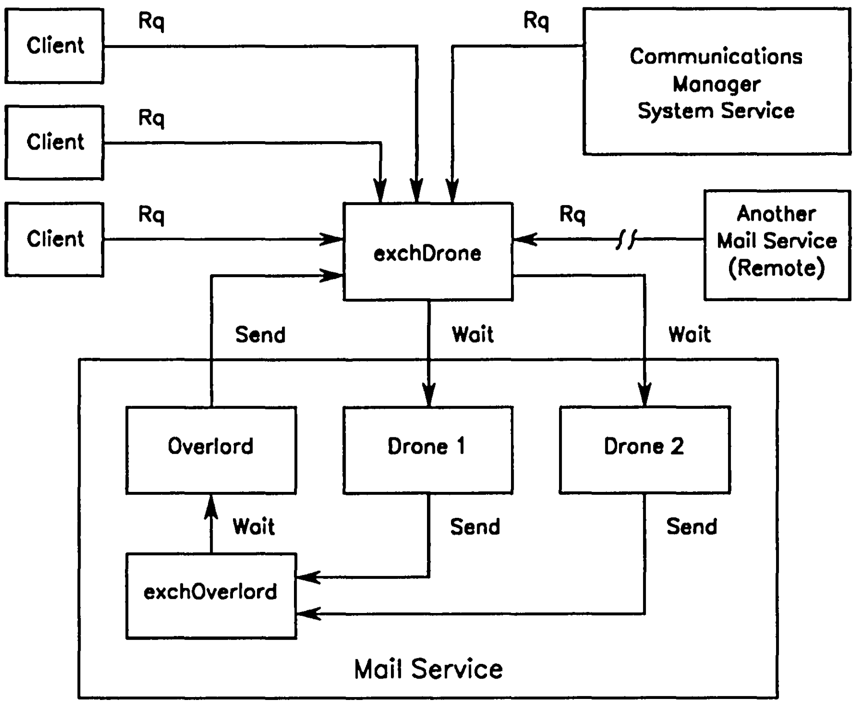


Figure 13-5. Simplified Version of Mail Architecture

The first version of the Mail Service contained only two processes: the Overlord and one Drone. The idea was to get the product working successfully with only these two processes, and then to consider adding more. This first version operated only within the local cluster network. In the next version, communication with other clusters was added, and at this point a second Drone was introduced so that delays would not occur at the Mail Service's input exchange. Development continued in this way, with features (and Drone processes) being added to a working structure.

The Overlord and the Drone have very different duties and do not compete for resources such as data structures or files. The Overlord communicates with the system clock and decides what should be done (for example, when to make certain connections to other Mail Services). Once it has decided what to do, the Overlord Sends a message to the Drones' exchange (which is the only externally visible Mail Service exchange) and gives up control. The Overlord's duties all can be executed quickly, and it runs at a more favorable priority than does the Drone.

Drones Wait at the Drone exchange. Any Drone can process any request that comes in here, whether it is a command from the Overlord or a request from an interactive client application, a remote Mail Service, or a Communications Manager system service. Drones carry out time-consuming duties that involve disk accesses, sending mail, and so forth. They run at a less favorable priority than that of the Overlord. Drones can Send messages to the Overlord using internal exchanges such as the one shown as exchOverlord in the figure.

The Overlord never accesses files. When there was only one Drone, there was no competition for file access. As more Drones appeared, these identical processes did need to access the same structures and files, so message-based semaphores were used to protect them.

The Communications Manager System Service in Mail is actually a creature that is rather rare in the CTOS world: it is an installed system service that does not serve any requests. Instead, it only makes requests of the Mail Service to find out what it should do. Because it does not serve requests, it has no issues with termination. Also, with such a design, it is possible to add as many Communications Managers as you like with no need to coordinate between them, because coordination is all done through one central point. It also allows installation of more than one Communications Manager on one workstation, something that is not possible where a system service serves requests.

The system service that serves no requests is the key to the extensibility of Mail. At various times, through this mechanism, support for gateways to other mail systems and for facsimile transmission from the user's interactive Mail interface have been added. These additions, while some have involved minor changes to Mail, have never required redesign of the Mail Service.

Summary

If you are new to the CTOS world and want to write a system service, the best way to start is with an academic exercise: a single-process, synchronous system service like the one we outlined for TimeKeeper in Chapter 12. Going on from that point, your best approach is to use the ServerGen template if at all possible. If you need to extend your model to make asynchronous back-end calls to other system services or interrupt handlers, you can at least refer to ServerGen for guidance in initialization, identification of inputs, decision-making code, and termination handling.

In general, the experience of many CTOS system service writers has shown that the best approach for a sophisticated system service is the single-process, asynchronous system service. If you wish to consider a duplicate multiprocess design such as that used in Mail, you probably should have reasons comparable to Mail's extensive and varied networking and need for extensibility to do so.

Writing system services is a challenge worth undertaking, not only for the resulting cleanly distributed product, but also because it will further your understanding and appreciation of the CTOS architecture.

Part 3

CTOS and the Future

14

Trends and Paper Napkins

A simple and extensible system like this one allows designers to think up enormous wish lists of enhancements, whether based on their own innovations or the newest industry breakthroughs. Almost all of these things could be done, but only some of them can be. As the idea people continue to sketch, the practical ones remind them that one must consider the needs of the real world and the real tasks for which people want to use these systems.

When the people who started Convergent Technologies sketched their first design on a paper napkin, they were just carrying part of the larger computer culture into their new venture. The excitement of creativity has always fueled people like them and always will.

Those first people hired other people who knew that the processes of creativity were not going to stop with them. They knew that CTOS would live in a changing technological world, and they designed it for that world. They did such a good job of that flexible design that it is hard for us to remember that they were working at a time when 256 Kb was a tremendous amount of memory to put into a workstation, and everyone was hoping that 8-inch disk technology would be reliable early enough to put into their first workstation.

They did such a good job, as a matter of fact, that they have presented their successors with temptation. A simple and extensible system like this one allows designers to think up enormous wish lists of enhancements, whether based on their own innovations or the newest industry breakthroughs.

Almost all such enhancements could be done, but, in a practical world, only some of them will be. As the idea people continue to sketch, the practical ones remind them that one must consider the needs of the real world and the real tasks for which people want to use these systems.

As we said earlier, CTOS development is firmly rooted in several basic principles. The first designers began with modularity, distributed processing, strength, speed, flexibility. Soon compatibility and resiliency also became important. Today, those principles are summed up in the following list:

- Open
- Modular
- Optimized
- Resilient
- Compatible
- Available from Multiple Sources
- Distributed
- Scalable

Future development will continue to be based on these principles. The operating system will remain small and modular, optimized for distributed processing and easy customization. Developers will remain committed to compatibility and to producing a stable platform operating system that can continue to support developers and users alike on into the 90s.

This chapter is intended to discuss possible future developments for CTOS. It touches on the directions and strengths CTOS has had in the past and makes some attempts to foretell the direction for the future. Since, however, the ideas that are sketched on paper napkins are ideas that grow interactively in the stimulating environment that is produced when trends in new technology come together, this chapter can only hint at what will happen in the CTOS environment. The operating system was born out of an effort to take advantage of new technology and has grown out of an effort to stay abreast with and effectively apply each new advance. Paper napkins can easily be crumpled up and thrown away; likewise, the plans outlined here should not be taken too literally, because we know they'll change with time.

Now, let's take a look at where we might go in the future.

Standards and Interoperability: A CTOS Commitment

The CTOS commitments to compatibility and open systems are long-standing. With the introduction of CTOS/Open in 1989, CTOS established itself as an open platform. CTOS/Open makes available an Applications Programming Interface that provides commonality across the various suppliers' versions of CTOS to ensure that applications run optimally from one to the other. It will continue to grow and to provide added functionality as the industry and CTOS continue to grow.

CTOS is also committed to be a platform that embraces other existing and forthcoming standards. It is establishing itself as a base platform onto which the most successful industry standards will be integrated. CTOS developers understand that interoperability and the ability to run specific applications on multiple platforms are of prime importance to users. By embracing standards and the applications that run on them, CTOS will be providing users with an integrated solution platform not matched by any other platform vendor.

In the 90s, CTOS will provide support for several standards. They already include: (PC-Compatible) DOS and Windows™, and will soon include POSIX and Presentation Manager. We shall touch on each of these briefly. Others, of course, will also be included as standards are established.

Several versions of a DOS platform emulation product were already available on CTOS in the mid-80s. With the advent of 80386-based CTOS systems, a software-only solution became available. CTOS now provides a complete environment for supporting off-the-shelf DOS-based programs, including those requiring VGA video. Since the entire PC environment, including the BIOS, is emulated as part of one CTOS context, DOS and CTOS applications run concurrently. The DOS environment supports Windows and will support evolving DOS standards, such as the DOS Protected Mode Interface (DPMI).

CTOS will also provide POSIX compliance, as one of the first non-UNIX platforms to do so. The IEEE developed the POSIX standard partly as a result of a need for a nonproprietary standard for the AT&T UNIX System V platform. It provides a common platform for applications, to ensure that they can be used on multiple platforms with minimal portation work. Although the API is similar to UNIX, it is intended to have a wider application. CTOS will integrate the POSIX platform by implementing a system service that will support the POSIX application programming interface (API). In addition, the shell and utilities part of the POSIX standard will also be provided on CTOS. This marriage of CTOS and POSIX will allow POSIX applications to have access to many of the facilities provided by CTOS, such as built-in networking. This illustrates synergistic value of having CTOS encompass other standards.

Future versions of CTOS will include Presentation Manager as the CTOS standard Graphical User Interface. This will allow PM-based applications to be easily ported to run in the CTOS environment. Future versions of CTOS will also provide support for Dynamic Linked Libraries, which are essential for Presentation Manager-based applications. This basic framework will allow PM applications to work in the CTOS/Open environment.

An additional set of standards which CTOS provides on-going support for is Open Systems Interconnection (OSI). The set of standards under the umbrella of OSI provides a basis for designing software systems which allow communication between dissimilar platforms. The standards define both the interaction between computer systems and the functions necessary for communications. Systems designed to be compatible with the OSI standards, therefore, provide compatibility with products designed to function to the standards.

The model for compatible communications, as specified by the International Standards Organization, is the OSI Reference Model. Here communications is divided into seven different layers, each of which is a group of related functions. OSI protocols define the functions the layers represent. The layering scheme allows a greater flexibility in choosing systems, equipment, and or software, e.g., one system may have all layers implemented in software, an alternate system may have the lower layers implemented in hardware with the remaining in software.

CTOS has current implementations of the lower 5 levels, Physical, Data Link, Network, Transport, and Session, within various communications services. Also, X.400 (OSI Mail) and FTAM (File Transfer Access and Methods) are implemented as extensions of the Presentation Layer. Support for X.500 (Directory Services) will also be made available in the future.

A strong future direction for the CTOS platform will be to continue to respond to the emergence of new standards, adding support for them and for applications designed to them. Looking back, the first CTOS developers did not live in a world of standards. On the contrary, in 1980 all the platform vendors were competing to have the best new and different system. The designers who gave CTOS the message-based system service as a way to support flexibility in the future could not possibly have imagined that one day this facility would actually enable CTOS to support such a wide set of emerging industry standards. Luckily they did design a flexible and modular system. The world continues to change, and CTOS will change with it.

Microprocessor Architecture

In general, CTOS will continue to take advantage of architectural opportunities presented by ongoing developments in the Intel 80*x*86 family of microprocessors. CTOS has been functional on every Intel 80*x*86 microprocessor and will be functional on following microprocessors because of the advantages those systems give to the systems programmer while retaining compatibility with previous versions. Recall that CTOS now functions on the 80186, 80286 and 80386 processors and will soon function on the 80486 microprocessor.

There have always been discussions of putting CTOS on other kinds of hardware–other people's hardware or multiprocessor hardware. These discussions are likely to continue.

Demand Paged Virtual Memory

A program is said to execute in "virtual memory" when it uses a linear address space that is larger than available physical memory. CTOS currently provides only a limited form of virtual memory in that an application may optionally use demand segmentation of code within its own partition. Initially, this mechanism was implemented purely in software and later used the demand segmentation features of the 80286.

Since the introduction of the 80386, true virtual memory based on demand paging has been a possibility for CTOS. Demand paging is a form of virtual memory management in which a program's linear address space is composed of contiguous fixed sized regions called pages. A page is either mapped to physical memory or is marked as "not present". Demand paging requires hardware support for mapping linear to physical addresses and for detecting "not present" pages.

When the program accesses a "not present" page, a page fault occurs. The operating system resolves the fault by mapping the page to physical memory, and if necessary, copying the contents of the page from a disk. In order to resolve a page fault, the operating system may have to take physical memory from another page, and possibly write the contents of the "replaced" page to disk.

Over the years, CTOS developers avoided virtual memory management due to the belief that virtual systems provide poor performance. Today, this objection is no longer valid as current virtual memory policies are known to deliver effective performance.

The main benefit of demand paging in CTOS is more flexible memory management for applications and servers. A paged environment is more flexible because it allows dynamic use of memory, such as allocating/dealloca-ting data structures or loading/unloading code, without the problem of splitting memory into numerous unusable fragments.

Demand paging is particularly useful in the following CTOS environments:

- Workstations, especially servers, that need more system services than will fit in available memory. Currently, CTOS system services must be entirely memory resident.
- Workstations that run more applications than will fit in available memory. Without demand paging, CTOS must swap out entire applications when memory is oversubscribed. With demand paging, each application can continue even though some pages are not present.

- Diskless workstations. Such workstations help to reduce the cost of computing. Virtual memory means that a workstation can be configured with less memory, further reducing the cost.

80386 Flat Model (32 Bit Addressing)

Implementation of CTOS support for direct use of the 32-bit address space, possible for the first time on the 80386 microprocessor, is a matter of debate. CTOS people around the world are clearly aware of the trend in this direction, and there is a growing desire to support 32-bit applications written to run under UNIX. Strong voices are reminding everyone that if 32-bit support is implemented, it should be interoperable with existing 16-bit systems.

CTOS as a Distributed Object System

There is a strong trend in the industry to move toward object-oriented execution environments. Such environments emphasize reusability and interoperability of code modules or objects. They offer the advantages of reduced application size and increasingly standard operation for similar functions.

We repeatedly emphasized throughout this book how very modular and extensible the CTOS architecture is. CTOS is, therefore, well-positioned to adopt some of the features associated with object-based environments. In addition, its distributed heritage is well established. That distributed architecture would provide a solid platform to allow for easy interoperability of distributed system objects.

CTOS is already adopting the look and feel of the object-oriented user interface. The CTOS response to the move in the industry toward an object-oriented approach to programming is yet to be seen.

The Far Term

The modular, extensible architecture that forms the platform for CTOS has proved to be a firm basis for growth in the 80s. In the 90s that same unique strength, combined with the robustness CTOS has gained through maturity and through years of testing in real world situations, makes it an excellent platform on which to continue to build. We don't know exactly where the future will lead, but we do know that CTOS will respond to new trends and capabilities as the computer industry continues to grow.

A

ServerGen: A Sample System Service

. . . because we believe in the practical side of things . . .

This appendix contains two modules written in the C programming language. When compiled and linked along with a few user written routines, these modules comprise a functional system service, which we have called ServerGen. Chapter 13 describes how ServerGen works.

The initial module is called FooServer.C. It contains sample procedures for proving ServerGen's functionality. The other module is Server.C, the main code module in ServerGen.

SerGen is structured with a main routine which first calls an Initialization Routine (where requests are shared and conversion to a system process are accomplished). The service then goes into a loop where the service waits for a message, processes it, and then loops back to the wait.

ServerGen has external stubs which reference user written procedures, examples of which are included in FooServer.C. Note that we have not included these here.

FooServer

```
/* FooServer.c -- Test server using the system service.lib.  Stores a buffer
of open
 *   file specs, then writes them to a file and deinstalls.
 *
 * Log:
 * 10/28/85 JA Created - PLM
 * 2/7/86 TB Converted to C
 */

#define ParamBlkType 1
#define RqType   1
#define SysConfigType 1
#define SysTimeType  1
#define TrbType   1

#define ErrorExitString 1
#define CheckErc  1
#define AllocMemorySL 1
#define SetMsgRet  1
#define SetPartitionName 1
#define CreateFile  1
#define OpenFile  1
#define Write   1
#define CloseFile  1
#define Beep   1

#include "[sys]<h>CTOSLib.h"
#include "[sys]<h>String.h"
#include "ServerGen.h"

/* application-specific definitions */
#define RQOPENFILE     4
#define RQOPENFILELL      97
#define RQCLOSEALLFILES     19
#define RQFILESYSTEMABORT    112
#define RQREOPENFILE    294
#define IBHEXERC    5
#define ERCNOTIMPLEMENTED  7
#define BUFFERSIZE    512
#define BUFFERSIZEL    512L
```

```
/* Structures defined by every service */
unsigned rgServeRq[] = {
 RQOPENFILE, RQOPENFILELL, RQREOPENFILE, RQCLOSEALLFILES, RQFILESYSTEMABORT
 };
unsigned nServeRq = 5;

/* scratch space for deinstall, 1 word per rqCode in rgServeRq */
unsigned rgRqExch[5];

/* System requests (termination, abort, swap) served or filtered */
unsigned rgSystemRq[] = { RQCLOSEALLFILES, RQFILESYSTEMABORT };
unsigned nSystemRq = 2;


/* Application specific declarations */
char *pBuf;
int  ib;
char postScript[] = " erc xxxxh\n";

/* Application local subroutines */
char rgHex[] = "0123456789ABCDEF";

void
ConvertWHex(Word w, char pRet[])
{
 int i;

 for(i = 3; i >= 0; i--) {
  pRet[i] = rgHex[w & 0xF];
  w = w >> 4;
 }
}


/* Standard routines that may be present in all servers. */

void
Initialize()
/* This procedure is called after requests are verified but
 * before requests are actually served.  All OS calls are valid here.
 * Do error checking, memory or exchange allocation, config file reading,
 * extraction of arguments from the command form etc.
 *
 * This example service checks the os type, then allocates and clears a
 * buffer for use during the processing of requests.
 */
{
 if(pConfig->fMultipartition == 0)
  ErrorExitString(ERCNOTIMPLEMENTED, "OS must be MultiPartition", 25);
 CheckErc(AllocMemorySL(BUFFERSIZE, &pBuf));
 *pBuf = 0;       /* null-terminate buffer */

 /* Set message to be printed upon successful installation - */
 CheckErc(SetMsgRet("Installation complete.", 22));
}


/* Requests have been served.  Installation succeeds or fails.
 * From now on all routines must use RequestDirect to issue OS calls listed
 * in rgServeRq above, issuing the rqs to the exchange recorded in rgRqExch.
 * Other requests may still be issued via procedural interface (by name).
 *
 * Some OS calls now no longer work after ConvertToSys.  System services
 * have no video structures so calls to VAM or VDM are illegal
 * (PutFrameChars etc).
 * In SinglePartition os,
 *  memory allocation/deallocation is illegal,
 *  exchange allocation is illegal,
 *  files must be opened using OpenFileLL,
 *  interrupt routines may no longer be set,
 *  bytestreams are not supported.
 *  In MultiPartition os,
 *  memory deallocation will work, then the memory may be reallocated,
 *  but long-lived memory is gone,
 *  exchanges may be freely allocated,
 *  files operate normally,
 *  interrupt routines may be set/reset normally,
 *  bytestreams other than Video are supported.
 */
```

```
/* Optional:
 *
 * This procedure is called once after installation.  Do things that need
 *  not occur unless installation is successful, such as starting the timer.
 */
void
 Start()
 {
 /*  rqTime.counter = rqTime.counterReload; */

  CheckErc(SetPartitionName(0, "FooServer", 9));
 }


/* The following five routines are called as messages are received.
 * Only one routine is ever called at once, i.e. no problems with
 * reentrancy, recursion or semaphores.
 *
 * Each routine returns a Word which may have the following values -
 *  lOk    the routine discharged the request itself; do nothing
 *  lRespond  the request has been processed, respond to the user
 *  lForward  the request should be forwarded to the regular handler
 *  lPass   like lForward, but when it is done call HandleRespond
 *  lOkDeinstall like lOk; also deinstall the service
 *  lRespondDeinstall like lRespond; also deinstall the service
 *  lForwardDeinstall like lForward; also deinstall the service
 *
 * HandleRequest and HandleSystemRequest return different values in each
 * sort of server:
 * A pure system service uses lRespond.  An asynchronous server uses lOk.
 * lForward and lPass are used by one-way and two-way filters, respectively.
 *
 * Timer and HandleMessage always return lOk.
 * HandleRespond returns lRespond or maybe lOk.
 *
 * The Deinstall values are the same as the regular values except after
 * discharging the request, the system service unserves the requests,
 * flushes the exchange and exits.
 */
```

```
/* Optional
 *
 * rqTimer has counted down to 0.  See OS call OpenRTClock.
 * void
 * Timer(char *pRq)
 * {
 *   return(lOk);
 * }
 */


Word
HandleRequest(struct RqType *pRq)
/* An original request has been issued whose request code field matches
 * one of those in the rgServeRq array.  Perform applications-specific
 * operations.  Fields of the request block based on the pointer pRq
 * that may be changed are -
 *  pRq->ercRet  error code to be returned to caller
 *  pRq->pb->  response buffer(s) pointed to by the request block
 * Any request block field may be examined.  See Operating System Manual
 * for a description of request blocks.
 */
{
 /* This example service records the name of every file opened, until its
  *  name buffer is full.  It then deinstalls.  This is a two-way filter.
  */
 if(BUFFERSIZE - ib >= pRq->s0) {
  strncpy(pBuf + ib, (const char *)(pRq->p0), pRq->s0);
  ib += pRq->s0;
  return(lPass);
 } else
  return(lForwardDeinstall);
}
```

```
Word
HandleSystemRequest(struct RqType *pRq)
/* A system request has been issued whose request code field matches
 * one of those in the rgSystemRq array.  All requests with the same userNum
 * as the system request must be processed before the system request is
 * returned (via Respond).  Requests for that userNum remembered by the
 * service must be backed out (ercSwapping), aborted (ercAbort) or
 * completed (erc), and Respond called for each.
 * There are three kinds of system request, depending on the value of the
 * first word of control info in the request block header:
 *
 *                  1st word cntl info  Complete    Abort   Back out
 * Swapping           ercSwapping (37)     ok                  ok
 * Abort              ercAbort   (8200)    ok         ok
 * Termination        other erc            ok         ok
 *
 * All system request processing must be done quickly or the termination
 * process will take a long time, appearing as a delay or hang to the user.
 *
 * If this service is a filter, it must filter system requests of the
 * filtered service, and pass them one- or two-way just as other filtered
 * requests are passed (lForward or lPass).  The default HandleSystemRequest
 * routine (inStdServer.Lib) returns lForward. If the filter service is two-
 * way the user MUST write a HandleSystemRequest routine that returns lPass,
 * and the user HandleRespond must accept system requests, probably just
 * returning lRespond.  This ensures requests passed to the filtered service
 * are flushed before the system request is returned.
 */
{
 /* This example service doesn't store requests, so it need only
  * pass the system request to the filtered service.  Since this
  * is a two-way pass-through filter, it MUST two-way filter the
  * system requests too (lPass instead of lForward).
  */
  return(lPass);
}
```

```
Word
HandleRespond(struct RqType *pRq)
/* A request block has come back via Respond.  It was previously received
 * at HandleRequest, which returned with the code lPass.  The request was
 * passed to the normal receiver of that request, which completed it and
 * Responded back to us.  This routine may now examine the results of the
 * operation, and then must let the request be Responded.
 */
{
 switch(pRq->rqCode) {
case RQOPENFILE:
case RQOPENFILELL:
case RQREOPENFILE:
 /* This example service records the error code returned from any open. */
 if(BUFFERSIZE - ib >= sizeof(postScript)) {
  ConvertWHex(pRq->ercRet, &postScript[IBHEXERC]);
  strncpy(pBuf + ib, postScript, strlen(postScript));
  ib += strlen(postScript);
  return(lRespond);
 } else
  return(lRespondDeinstall);

default: /* System request come back from filtered service.
            No requests stored in this filter,
            filtered service has been flushed,
            lPassed request also flushed.
            Ok to respond.
         */
 return(lRespond);
 }
}


/* Optional
 * Word
 * HandleMessage(Word w)
 *   An interrupt routine has sent a one-word message to the server.
 *   This message may be a data byte or word, or it may indicate some
 *   condition such as "buffer full".
 *   The interrupt routine cannot make requests, so it must "poke" the
 *   server somehow, and the server makes the requests.
 *
 *   The interrupt routine could have set a flag that is checked by the
 *   service, but then the flag would have to be polled, e.g. every 1/10th
 *   second the timer routine could check the flag.  Sending a message
 *   usually is a better solution because the condition is noticed
 *   instantly, also without using any processor time for polling.
 *
 * {
 *   return(lOk);
 * }
 */


void
Cleanup()
/* The service is deinstalling.  The requests have been served back to their
 * original destinations.  All requests pending have been discharged.
 * The service process is about to disappear.  Do any final operations
 * such as writing a log entry or flushing a buffer.
 * It is ok to issue requests by name that this service had been serving.
 */
{
 Word fh, cbWrite;

 /* This example service writes the name buffer to a file. */
 CreateFile("[sys]<sys>OpenNames.dat", 23, NULL, 0, BUFFERSIZEL);
 if(!OpenFile(&fh, "[sys]<sys>OpenNames.dat", 23, NULL, 0, 0x6D6D)) {
  Write(fh, pBuf, BUFFERSIZE, 0L, &cbWrite);
  CloseFile(fh);
 }
 Beep();
}
```

ServerGen

```
pragma Memory_model(Big);
/* Module SERVER MAIN */


#undef Debug
#include "[Sys]<h>SysCom.h"

#define PcbType 1
#define RqType 1
#define TrbType 1
#define SysConfigType 1

#define AllocExch 1
#define DeallocExch 1
#define OpenRTClock 1
#define Respond 1
#define Request 1
#define Wait 1
#define Check 1
#define ForwardRequest 1
#define Send 1
#define RequestDirect 1
#define QueryRequestInfo 1
#define ServeRq 1
#define ConvertToSys 1
#define ChangePriority 1
#define Exit 1
#define ErrorExit 1
#define SetMsgRet 1
#define CheckErc 1
#define Crash 1
#define GetpStructure 1
#define OsVersion 1
#include "[Sys]<h>CTOSLib.h"

static Byte C_1[] = {
 "Installation failed"};

extern void  ProcessMessage(FLAG  fNewRqOk);
extern ErcType  KillProcess(Word  pid);
extern void  ExitAndRemove();

void
CrashIfErcNotOk(ErcType  erc)
{
 if (erc != ercOK)
  Crash(erc);
}
```

```
#define RqErc 1
#include "[Sys]<h>Erc.h"

/* GetpStructure case values */
const lGetpExParDesc = 0;
const lGetpCharMap = 1;
const lGetpVCB = 2;
const lGetpAscb = 3;
const lGetpVLPB = 4;
const lGetpBcb = 5;
const lGetpTypeAhead = 6;
const lGetpRgpVidMemLine = 7;
const lGetpRgLineMap = 8;
const lGetpContextStatus = 9;
const lGetpOPcbRun =0 x24C
const FpType = 10;/* 1st SRP board hardware type */
const StubRqCode = 0x7462;/* 'tb' */
const exchNil = 0;

typedef struct PcbType pcbRun;
typedef Offset oPcbRun;
typedef struct RqType rq;
typedef struct {
 Byte waste[64];
 Byte cbName;
 Byte rgName[30];
} Ascb;
typedef struct SysConfigType config;

config *pConfig;
Pointer pTime;
Ascb *pASCB;
Word wVersion;
ExchType exchServe;
FLAG fConvertToSys = {1};


/* External data are defined in the user-written module. */
extern Word rgServeRq[1];
extern Word nServeRq;
extern Word rgRqExch[1];
extern Word rgSystemRq[1];
extern Word nSystemRq;

/* The following two external symbols may be omitted from user program. */
extern struct TrbType rqTime;
static Pointer prqTime = {
 &rqTime};
```

```
/* Local variables for server main program. */
static rq *pRq;
static struct {
 Pointer pRq;
 ExchType exchResp;
} rgRcb[10];
static oPcbRun *poPcbRun;
static pcbRun *pPcbRun;
static ErcType erc;
static Word wRet;

/* External procedures are defined in the user-written module.
   All routines except HandleRequest may be omitted.
   Omitted routines resolved from library of stubs StdServer.Lib.
*/
extern void  Initialize();
extern void  Start();
extern Word  Timer(Pointer  pRq);
extern Word  HandleRequest(Pointer  pRq);
extern Word  HandleSystemRequest(Pointer  pRq);
extern Word  HandleRespond(Pointer  pRq);
extern Word  HandleMessage(Word  w);
extern void  Cleanup();

Word
Findw(Word rg[], Word w, Word cb)
{
 Word ib;
 for(ib = 0; ib < cb; ib++)
 { if (rg[ib] == w) return ib; }
 return 0xFFFF;
}

void
Init()
{
 Word iRq;
 struct {
  Word exch;
  Word lsc;
 } rqInfo;

 /* Address some interesting structures.
      Their pointers are public so user gets them without effort. */
 erc = GetpStructure(lGetpAscb, 0, &pASCB);
 erc = OsVersion(&wVersion);
 erc = GetpStructure(ATpSysTime, 0, &pTime);
 erc = GetpStructure(ATpConfiguration, 0, &pConfig);
 if (pConfig->fMultipartition == 0) { /* Single Partition only */
  erc = GetpStructure(lGetpOPcbRun, 0, &poPcbRun);
  pPcbRun = (Pointer)BuildPtr(SelectorOf(poPcbRun), *poPcbRun);
 }
 CheckErc(AllocExch(&exchServe));

 /* Use the user-defined structures, if present. */
 if (rqTime.rqCode != StubRqCode) {
  rqTime.exchResp = exchServe;
  CheckErc(OpenRTClock(&rqTime));
 }

 iRq = 0;
 while (iRq < nServeRq) {
  CheckErc(QueryRequestInfo(rgServeRq[iRq], &rqInfo, sizeof(rqInfo)));
  rgRqExch[iRq] = rqInfo.exch;
  iRq = iRq + 1;
 }

 Initialize();

 /* Don't ConvertToSys if flag set */
 if (fConvertToSys) {
  erc = ConvertToSys();
  if (erc != ercOK)
   CheckErc(SetMsgRet(C_1, sizeof(C_1)));
  ErrorExit(erc);
 }

 iRq = 0;
 while (iRq < nServeRq) {
  /* if exch is served on XE530 1.4, first unserve it */
  if ((rgRqExch[iRq] != 0)
   && (pConfig->HardwareType >= FpType)
   && wVersion < 0x0B00)
   CrashIfErcNotOk(ServeRq(rgServeRq[iRq], 0));
  CrashIfErcNotOk(ServeRq(rgServeRq[iRq], exchServe));
  iRq = iRq + 1;
 }

 Start();

 if (rqTime.rqCode != StubRqCode) {
  rqTime.cEvents = 0;
  rqTime.counter = rqTime.counterReload;
 }
}


/* Find a free Rcb and register the rq in it. */
typedef struct RqType rq;
static void
AllocRcb(rq *pRq)
{
 Word iRcb;
 for (iRcb = 0 ; iRcb <= Last(rgRcb) ; iRcb++) {
  if (rgRcb[iRcb].exchResp == exchNil) {
   rgRcb[iRcb].exchResp = pRq->respExch;
   pRq->respExch = exchServe;
   rgRcb[iRcb].pRq = (Pointer)pRq;
   return;
  }
 }
 wRet |= 0x10;/*No rcb available, cause to deinstall*/
}


/* Find the Rcb for the rq and restore the rq. Free the Rcb. */
typedef struct RqType rq;
static void
RestoreRcb(rq *pRq)
{
 Word iRcb;

 for (iRcb = 0 ; iRcb <= Last(rgRcb) ; iRcb++) {
  if (rgRcb[iRcb].pRq == pRq
   && rgRcb[iRcb].exchResp != exchNil) {
   pRq->respExch = rgRcb[iRcb].exchResp;
   rgRcb[iRcb].exchResp = exchNil;
   return;
  }
 }
 Crash(ercInconsistency);
}


/* Check if any Rcb active.  If so, wait for respond to rq. */
static void
FlushRcb()
{
 Word iRcb;

 while (1) {
  for (iRcb = 0 ; iRcb <= Last(rgRcb) ; iRcb++) {
   if (rgRcb[iRcb].exchResp != exchNil)
    goto WaitForAnotherRq;
  }
  return;/*No more rcb in use.*/

WaitForAnotherRq:
  CheckErc(Wait(exchServe, &pRq));
  ProcessMessage(FALSE);
 }
}
```

```
void
DeInstall()
{
 Word iRq;

 if (rqTime.rqCode != StubRqCode)
  rqTime.counter = 0; /*Turn off*/
 for (iRq = 0; iRq < nServeRq; iRq++) {
  CrashIfErcNotOk(ServeRq(rgServeRq[iRq], 0));
  CrashIfErcNotOk(ServeRq(rgServeRq[iRq], rgRqExch[iRq]));
 }

 FlushRcb();

 while (Check(exchServe, &pRq) == ercOK) {
  (void)ProcessMessage(FALSE);
 }

 Cleanup();

 /*VP*/
 if (pConfig->fMultipartition == 3)  ExitAndRemove();
 /*MP*/
 if ((pConfig->fMultipartition) & 1) Exit();
 /*SP with KillProcess*/
 CrashIfErcNotOk(DeallocExch(exchServe));
 if (wVersion >= 0x0900)
  CrashIfErcNotOk(KillProcess(*poPcbRun));
 /*SP, old*/
 CrashIfErcNotOk(ChangePriority(0x0FF));
 erc = Wait(pPcbRun->exchgSync, &pRq); /*forever*/
}


void
ProcessMessage(FLAG  fNewRqOk)
{
 Word wCase;
 Word iRqExch;
 ExchType rqExch;

 if ((pRq == prqTime) && (rqTime.rqCode != StubRqCode))
  if (fNewRqOk)
   wRet = Timer(pRq);
  else
   wRet = 0; /*Ok, don't deinstall*/
```

```
else if (SelectorOf(pRq) == 0)
 if (fNewRqOk)
  wRet = HandleMessage(OffsetOf(pRq));
 else
  wRet = 0; /*Ok, don't deinstall*/

else {
 if (pRq->respExch == exchServe) {
  RestoreRcb(pRq);
  wRet = HandleRespond(pRq);
 } else if (Findw(rgSystemRq, pRq->rqCode, nSystemRq)
    != 0xFFFF)
  wRet = HandleSystemRequest(pRq);
 else if (fNewRqOk)
  wRet = HandleRequest(pRq);
 else {
  /*new rq not ok, deinstalling*/
  pRq->ercRet = ercServiceNotAvail;
  CrashIfErcNotOk(Respond(pRq));
  wRet = 0;
 }

 wCase = wRet & 0x0F;

 if (wCase <= 3)
  switch (wCase) {
  case 0: /*0 - no action*/
   break;
  case 1: /*1 - Respond*/
   CrashIfErcNotOk(Respond(pRq));
   break;
  case 2: /*2 - Forward*/
    {
    if ((iRqExch = Findw(rgServeRq, pRq->rqCode, nServeRq))
      == 0x0FFFF)
     rqExch = 0;
    else
     rqExch = rgRqExch[iRqExch];
    if (rqExch == 0) {
     pRq->ercRet = ercServiceNotAvail;
     CrashIfErcNotOk(Respond(pRq));
    } else
     CrashIfErcNotOk(ForwardRequest(rqExch, pRq));
   }
   break;
  case 3: /*3 - Pass*/
    {
    /*Diddle rq.exchResp to come back to us.*/
    AllocRcb(pRq);
    if ((iRqExch = Findw(rgServeRq, pRq->rqCode, nServeRq))
      == 0x0FFFF)
      rqExch = 0;
     else
      rqExch = rgRqExch[iRqExch];
     if (rqExch == 0) {
      pRq->ercRet = ercServiceNotAvail;
      /* To HandleRespond */
      CrashIfErcNotOk(Send(exchServe, pRq));
     } else
      CrashIfErcNotOk(RequestDirect(rqExch, pRq));
    }
    break;
   }/* CASE */
 }
}


/* Main program */

void
main(Word argc, char *argv[])
{
 Init();
 while (1) {
  CheckErc(Wait(exchServe, &pRq));
  ProcessMessage(TRUE);
  if ((((wRet) >> 4)) & 1) /* 0x10 bit is signal to deinstall */
   DeInstall();
 }
}
```

Glossary

A

abort request

Notifies system services that clients have terminated. Upon notification, system services can release resources, such as open files and locked ISAM records, allocated to the terminating clients. Issuing an abort request ensures that no requests are returned to the program after it has been terminated and replaced in memory by another program. The abort request also informs system services that resources allocated to the program should be freed.

application partition

A partition of user memory in which an application program can execute. A workstation can have any number of application partitions, with an application program executing concurrently in each. *See also* system partition.

application program

Can consist of code, data, and one or more processes executing in an application partition. If the program is executing in a variable partition, the program's code can be located anywhere in memory and can be shared by the same type of program in a different variable partition.

Applications Programming Interface (API)

The collection of operations or interfaces that an application can use to interface to a given software entity.

asynchronous operation

A procedure or protocol that allows for a response within a window of time rather than at an exact time interval.

Automatic Volume Recognition

The method by which CTOS can recognize and mount a uniquely named volume (disk) on any workstation in a cluster. This feature implies that if a workstation is removed from the local network, its hard disk can be moved to another workstation and simply used there, without any network reconfiguration.

B

bootstrap

To start (to boot) the system by reloading the operating system from disk. On other systems, this is often known as initial program load.

byte stream

A character-oriented, readable (input) or writable (output) sequence of 8 bit bytes used by the Sequential Access Method to transfer data to or from a device. An input byte stream can be read until either the program chooses to stop reading or it receives status code 1 ("End of file"). An output byte stream can be written until the program chooses to stop writing.

C

CCGI+

An applications programming interface for CTOS that meets the ANSI CGI standard.

Check

A kernel primitive used by a client to determine if a message is queued at a specified exchange. If one or more messages are queued, the message that was first queued is removed from the queue, and its memory address is returned to the client. If no messages are queued, status code 14 ("No message available") is returned.

client

A process that requests a service provided by a system service. Any process, even a built-in operating system process, can be a client process, since any process can request system services. *See also* system service.

cluster

A local resource-sharing network consisting of a server connected to cluster workstations. One high-speed RS-422 or RS-485 channel is standard on each workstation. In cluster configurations connected to a server workstation, the server and all of the workstations connected to it use this channel for intercluster communications. For large clusters with a shared resource processor server, multiple channels are provided. The operating system executes in each cluster workstation and in the server. *See also* cluster workstation, CTOS Network, server, server workstation, and TeleCluster.

cluster agent

See cluster workstation agent.

Cluster server agent

Reconverts a message from a workstation connected to the cluster line to an interprocess request that is queued at the exchange of the request-based system service on the server that actually performs the intended function. The Server Agent includes the cluster code at the server that polls the cluster workstation for requests. *See also* cluster workstation agent.

cluster workstation

A workstation in a cluster configuration, connected to a server. *See also* cluster and server.

Cluster workstation agent

Converts interprocess requests to interstation messages for transmission to the server. The Cluster Agent service process is included at system build in a system image that is to be used on a cluster workstation. A Cluster Agent is the code that responds to Server Agent polling by sending a request to the server or by informing the server that it has no request to send. It is also sometimes referred to as the workstation agent. *See also* cluster server agent.

ClusterCard

An expansion card for IBM-PC compatible computers that provides cluster communications channels. The card is used to integrate PCs into a CTOS cluster.

ClusterShare

Software that, when used in conjunction with an IBM-PC compatible with a ClusterCard installed, integrates the PC into a CTOS cluster.

Computer Graphics Interface (CGI)

ANSI standard graphics application programming interface.

connection

Where a transaction is part of a series of interactions, the client and system service are said to have a connection; where the transaction is a one-time-only event, the relationship is said to be connectionless. A client may have several connections simultaneously to the same or different system services. These connections are all independent.

context

The collection of all information about a process. The context has both hardware and software components. The hardware context of a process consists of values to be loaded into process or registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack. The software context of a process consists of its default response exchange and the priority at which it is to be scheduled for execution. The Process Control Block is a system data structure that is the root of the combined hardware and software context of a process.

This term is sometimes also used to refer to the process itself.

Context Manager

A partition managing program that the user interacts with to start and switch back and forth between applications.

context switch

Occurs when a process is interrupted and its register contents are saved. When a process is preempted by a process with a higher priority, the operating system saves the hardware context of the preempted process in that Process Control Block. When the preempted process is rescheduled for execution, the operating system restores the content of the registers. The context switch permits the process to resume as though it were never interrupted. *See also* process, process context, and Process Control Block.

CTOS

Distributed, message-based operating system that runs on 80*x*86 microprocessors.

CTOS Network

A network consisting of server workstations connected by communications lines. Each server workstation is a node or junction in the network. A CTOS Network provides access to the system services of interconnected cluster configurations. Cluster workstations in the network can access files on any node. CTOS Network software is sold under various names including BNet and CT-Net.

CTOS/Open Advisory Council

The CTOS/Open Advisory Council was formed as a joint effort of manufacturers, reseller, distributors, software developers, hardware developers, and users to establish and promote the CTOS-based architecture as a standard of distributed network computing. CTOS/Open defines a set of features that are common to the current and future versions of CTOS operating systems.

D

DAM

See Direct Access Method.

deadlock

Also called "deadly embrace", deadlock is the state which results when two or more processes or programs are stopped, each waiting for a response that depends on the other stopped program.

default response exchange

Each process is given a unique default response exchange when it is created. This special exchange is automatically used as the response exchange whenever a client process uses the request procedural interface to a system service. For this reason, the direct use of the default response exchange is not recommended. The use of the default response exchange is limited to requests of a synchronous nature. That is, the client process, after specifying the exchange in a Request, must wait for a response before specifying it again (indirectly or directly) in another Request. *See also* exchange and response exchange.

demand-paging

A form of virtual memory management in which a program's linear address space is series of contiguous fixed sized regions called pages. A page is either mapped to physical memory or is marked as 'not present'. Demand paging requires hardware support for mapping linear to physical addresses and for detecting 'not present' pages.

device-dependent

Describes program interfaces closest to the actual hardware. A device dependent program performs I/O to a limited number of devices. *See also* device-independent.

device driver

A software program that provides the interface between a device such as a printer and other software. The device driver interprets the requests of other programs and provides device specific instructions.

device-independent

Describes program interfaces that are not close to the hardware. A device-independent program can perform I/O to a variety of devices. The Sequential Access Method operations, such as OpenByteStream, ReadByteStream, and CloseByteStream, are device-independent operations. *See also* device-dependent.

Direct Access Method (DAM)

Provides random access to disk file records identified by record number. The record size is specified when the DAM file is created. DAM supports COBOL relative I/O, but can also be called directly from any of the supported languages.

Direct Memory Access (DMA)

Access to memory that does not require processor intervention. A DMA controller in the processor module controls the transfer of data over the X-Bus to the main processor's memory.

directory

A collection of related files on one volume. A directory is protected by a directory password.

disk extent

One or more contiguous disk sectors that compose all or part of a file.

distributed processing

Processing which is spread-out, or distributed, between one or more machines in a network.

DMA

See Direct Memory Access.

dynamically installed system service

A program that was loaded as an application program and converted itself into a system service using the ConvertToSys operation. Once installed, a dynamically installed system service has the same capabilities as a system service that was linked with the System Image during system build. A dynamically installed system service must use CTOS operations (rather than system build parameters) to identify the request codes that it serves, specify its execution priority, establish its interrupt handlers, and so forth.

E

event

An external occurence which causes a response in the process itself.

event-driven priority-ordered scheduling

When processes are scheduled for execution based on their priorities and system events, not on a time limit imposed by the scheduler. *See also* process and event.

exchange

The path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out (FIFO) queues: one of processes waiting for messages and the other of messages for which no process has yet waited. An exchange is referred to by a unique 16 bit integer. *See also* default response exchange and response exchange.

Executive

An interactive application program that accepts commands from the workstation user and requests the operating system to load programs to execute those commands.

exit run file

A user-specified executable file that is loaded and activated when an application program exits. Each application partition has its own exit run file.

Extensible Virtual Toolkit (XVT)

A software library that provides for windowing applications one application programming interface that can be used to provide a graphical user interface or a graphics-like character-mode version of such across multiple operating system platforms and multiple graphical interfaces. For example, an XVT program can use the same calls to provide a Macintosh interface and an X-Windows interface.

F

FAB

See File Area Block.

FCB

See File Control Block.

FHB

See File Header Block.

File Area Block

There is a File Area Block for each disk extent in an open file. The FAB specifies where the sectors are and how many there are in the disk extent. The FAB is pointed to by a File Control Block or another FAB. The FAB is memory-resident. *See also* disk extent.

File Control Block

There is a File Control Block (FCB) for each open file. The FCB contains information about the file such as the device on which it is located, the user count (that is, how many file handles currently refer to this file), and the file mode (modify, peek, or read). The FCB is pointed to by a User Control Block and contains a pointer to a chain of File Area Blocks. The FCB is memory-resident.

file handle

A 16-bit integer that uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

File Header Block

There is a File Header Block (FHB) for each file. The FHB of each file contains information about that file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, and the disk address and size of each of its Disk Extents. The FHB is disk resident and one sector in size.

file specification

A full file specification is a string of characters that specifies the location of a file within a CTOS Network. It includes a node name, volume name, directory name, file name, and can include a password. A file specification can be shortened to leave out the node, volume, and/or directory names if desired. In those cases the operating system assumes that the current path should be used.

file system

A CTOS multiprocess system service that manages file manipulation.

filter process

A system service process that can be included in the System Image at system build or dynamically installed at any time. A filter process is interposed between a client process and a system service process that operate as though they are communicating directly with each other. The Service Exchange table is adjusted to route requests through the desired filter process. Filters can be one or two-way filters.

frame

A separate, rectangular area of the screen. A frame can have any desired width and height (up to those of the entire screen).

full file

Consists of a node name, volume name, directory name, and file name.

G

GDT

See Global Descriptor Table.

Generic Print Access Method (GPAM)

Provides high-level I/O for complex documents that may include text, graphics, or special text attributes. GPAM is an object module library that provides device independent formatting commands used for printing. GPAM is used typically to add rich formatting characteristics to text that is output to a printing device.

Generic Print System (GPS)

The Generic Print System is made up of a set of dynamically installed system services, which work together to handle communication between application programs, the operating system, and the printers and plotters currently installed.

Global Descriptor Table (GDT)

A protected mode structure that contains descriptors for segments, which are shared by all programs. *See also* Local Descriptor Table (LDT).

GPS

See Generic Print System.

I

Indexed Sequential Access Method (ISAM)

Provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files.

Interprocess Communication (IPC)

The method by which individual CTOS processes communicate by sending messages to each other. In this processes exchanges serve as message centers, where processes send messages or where they wait or check for messages. Processes use the request procedural interface to send such messages.

interrupt

External or internal; an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the processor and partly by the operating system.

ISAM

See Indexed Sequential Access Method.

K

kernel

The most primitive and the most powerful component of the operating system. It executes with a higher priority than any process but it is not itself a process. The kernel is responsible for the scheduling of process execution; it also provides IPC primitives.

L

LDT

See Local Descriptor Table.

linker

Links one or more object files into a run file to be loaded into memory.

Local Descriptor Table (LDT)

A protected mode structure in memory that contains descriptors for segments accessable to a run file. The operating system constructs the LDT based on information provided by the Linker.

local file system

Allows a cluster workstation to access files on a local hard disk(s) as well as files on the hard disk(s) at the server. The filter process of the local file system intercepts each file access request and directs it to the local file system or to the server workstation.

long-lived memory

An area of memory in an application partition. It is used for parameters or data passed from an application program to a succeeding application program in the same partition. If a character map other than the one in the system partition is needed, it must be allocated in the long-lived memory area of the application partition. *See also* application partition and system partition.

M

Master File Directory

There is an entry for each directory on the volume in the Master File Directory (MFD), including the Sys directory. The position of an entry within the MFD is determined by randomization (hashing) techniques. The entry contains the directory's name, password, location, and size. The Master File Directory is disk resident.

message

The entity transmitted between processes by the interprocess communication facility. It conveys information and provides synchronization between processes. Although only a single 4-byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message, while the 4-byte data item is conventionally called the address of the message. The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

MFD

See Master File Directory.

multiprogramming

The ability to run more than one program in memory at the same time. Multiprogramming supports the independent invocation and scheduling of multiple processes. In addition, it provides for concurrent I/O and for multiple processor implementations.

multitasking

See Multiprocessing.

multiprocessing

The ability for any program to have more than one process (thread of execution). Multiprocessing also is called multitasking.

N

Native Language Support (NLS)

The CTOS facilities that support translation of software programs by providing special operations that deal with language-specific formats such as date-time formats and that support separate message files, so that message strings and prompts can be easily translated without requiring recompilation and relinking of application programs.

Net Agent

A system service process located at a server workstation that receives requests over the network destined for request-based system services located at remote nodes and forwards these requests to the remote nodes.

Net Server

A process that responds to requests from Net Agent processes. The Net Server receives a request block from the Net Agent, executes the request on behalf of the remote client, and returns the response to the originating Net Agent. *See also* Net Agent.

node

A server workstation in a CTOS Network. Node also refers to the first element (node name) of a full file specification.

O

object module procedure

A procedure supplied as part of an object module file. It is linked with the user-written object modules of an application program and is not supplied as part of the System Image. Most application programs only require a subset of these procedures. When the application program is linked, the desired procedures are linked together in the run file of the application. The Sequential Access Method is an example of object module procedures. *See also* system-common procedure.

operation

An operating system kernel primitive, system service, system-common procedure, or object- module procedure.

P

partition

A logical part of memory, specifically allocated for use by a program such as the operating system or an application system. Processor memory can be divided into several partitions. The partitions can vary in size during use.

partition handle

Another name for a user number. *See* user number.

Partition Management facility

Permits concurrent execution of multiple application programs, each in its own partition. It provides operations for creating, managing, and removing application partitions.

PbCb

A 6-byte entity consisting of the 4-byte memory address of a byte string followed by the 2 byte count of the bytes in that byte string.

PC Emulator

CTOS software that allows MS-DOS to be run in a partition of memory on 80386 workstations in Virtual 8086 mode. The PC Emulator can also be run on 80286 workstations with a PC Emulator coprocessor module attached.

primary partition

When a single application partition exists in memory, this partition is called the primary partition. A primary partition is not under the control of a partition managing program, such as the Context Manager.

primitive

An operation performed by the kernel. *See also* kernel.

priority

Indicates a process's importance relative to other processes and is assigned at process creation. Priorities range from a high of 0 to a low of 254.

priority-ordered scheduling

A scheduling algorithm by which processes are scheduled for execution based on priority.

procedure

A subroutine.

process

An independent thread of execution for a program along with the context (that is, the processor registers) necessary to that thread. One or more processes are created each time a program is scheduled for execution. A process is assigned a priority when it is created so that the operating system can schedule its execution appropriately. *See also* priority.

process context

The collection of all information about a process. The context has both hardware and software components. The hardware context of a process consists of values to be loaded into processor registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack. The software context of a process consists of its default response exchange and the priority at which it is to be scheduled for execution. The Process Control Block is a system data structure that is the root of the combined hardware and software context of a process. *See also* context switch and Process Control Block (PCB).

Process Control Block (PCB)

A system data structure that is the root of the combined hardware and software context of a process. A PCB is the physical representation of a process. *See also* process context.

processor

Consists of the central processing unit (CPU), memory, and associated circuitry.

program

Consists of executable code, data, and one or more processes. The code and data can be unique to the program or shared with other programs. A program is created by translating source programs into object modules and then linking them together. This results in a run file that is stored on disk. When requested by a currently active program, such as the Executive, the operating system reads the run file into the application partition, relocates intersegment references, and schedules it for execution. The new run file can coexist with or replace other run files. *See also* primary task, run file, and secondary task.

protected mode

One of the CTOS operational modes. In protected mode, application programs can use all available free memory above the first megabyte up to the maximum allowed by the processor and the hardware. Features of protected mode are a different type of addressing that uses protected mode structures, such as LDTs and GDTs, to define segments; protection that imposes limits on the memory that programs can access; and virtual 8086 mode, which provides for running multiple operating systems in different memory partitions concurrently. *See also* Global Descriptor Table, Local Descriptor Table, real mode, virtual 8086 mode.

Q

Queue Manager

The Queue Manager is a system service that controls named, priority-ordered, disk-based queues.

R

RAM

Random access memory.

ready state

One of three states in which a process can exist. A process is in the ready state when it could be running, but a higher priority process is currently running. Any number of processes can be in the ready state at a time. *See also* running state and waiting state.

real mode

One of the CTOS operational modes. In real mode, application programs can only access memory within the first megabyte. *See also* protected mode.

Record Sequential Access Method (RSAM)

Provides blocked, spanned, and overlapped input and output. An RSAM file is a sequence of fixed-length or variable-length records. Files can be opened for read, write, or append operations.

reentrant code

Code that can be executed by more than one process at the same time. System-common procedures, for example, must be written in reentrant code

Remote Procedure Call (RPC)

A method for calling a procedure to perform a service irrespective of the location of the service routine. With an RPC, the calling process waits for the receipt of the message and when the message is received continues processing. RPC is implemented on CTOS by the Interprocess Communication facility.

Request

Kernel primitive that directs a request for a system service from a client process to the service exchange of the system service process. Before the primitive is issued, the data required for the system service is arranged in a request block in the client's memory. The easiest way for the client to access the service is to use the request procedural interface, which automatically builds the request block. *See also* request procedural interface.

request block

A block of memory provided by a client that contains a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system. The memory address of the request block is provided by the client during a Request primitive and by the system service during a Respond primitive. A request block is the "element" that the application program (or the operating system) sends to the operating system to request that a particular operation be performed.

request code

A 16 bit value that uniquely identifies a system service. For example, the request code for the Write operation is 36. The request code is used both to route a request to the appropriate system service process and to specify to that process which of the several services it provides is currently being requested.

request procedural interface

A convenient way to access system services, compatible with high-level languages, such as C and Pascal, as well as assembly language. The request procedural interface is a routine within the CTOS operating system that is executed when a program calls a request-based operation. The routine builds a request block message and calls the Request primitive, while the calling program is placed in the waiting state at its default response exchange for the system service to respond. *See also* default response exchange, Request, Respond, and Wait.

Respond

A kernel primitive used to pass a response back to a client process. Respond is typically used in conjunction with Request for communications between applications that are not located in the same partition.

response exchange

The exchange at which the requesting client process waits for the response from a request-based system service. *See also* default response exchange and exchange.

ROM

Read-only memory.

RSAM

See Record Sequential Access Method.

run file

An executable file, created by the Linker, that contains object modules linked together into code and data segments.

running state

One of three states in which a process can exist. A process is in the running state when the processor is actually executing its instructions. Only one process at a time can be in the running state. *See also* ready state and waiting state.

S

SAM

See Sequential Access Method.

SCSI

Small Computer Systems Interface, an American National Standard for the interconnection of computers with peripheral devices such as disk drives, tape drives, and printers.

semaphore

A synchronization primitive to coordinate the activities of two or more processes that are running at the same time and sharing information.

send

A kernel primitive typically used for communication between processes in the same partition (user number). Send accepts any 4-byte field as a parameter. This is usually, but not necessarily, the address of a message.

Sequential Access Method (SAM)

Provides device-independent access to a default set of real devices, such as the screen, printer, files, and keyboard. To transfer data to or from the device, SAM uses a character-oriented sequence of bytes known as a byte stream.

server

A workstation that has a cluster server agent and which acts a server or hub for cluster communications. A server can also be a shared resource processor.

Server Agent

See cluster server agent.

server workstation

A server workstation can serve a cluster configuration. The server workstation provides file management, queue management, and other services to all the cluster workstations. In addition, it supports its own interactive programs. *See also* cluster workstation.

service exchange

An exchange that is assigned to a request-based system service process when the system service is dynamically installed or at system build. The system service process waits for requests for its services at its service exchange.

Shared Resource Processor

A multiprocessor, floor-standing CTOS server. The shared resource processor can be configured to run various programs on each of its loosely coupled processors and can be expanded to contain up to six cabinets, each of which can contain up to 6 processor boards.

short-lived memory

An area of memory in an application partition. When a run file is loaded, the operating system allocates short-lived memory to contain its code and data. (Note that code that is shared by other sized programs in other variable partitions can be located anywhere in memory.) Short-lived memory also can be allocated directly by a client process in its own partition. Common uses of short-lived memory are I/O buffers and the Pascal heap. *See also* application partition.

spooler

A dynamically installed system service that transfers text from disk files to the printer interfaces of the workstation on which the spooler is installed. It can simultaneously control the operation of several printers. A disk-based, priority-ordered queue controlled by the Queue Manager contains the file specifications of the files to be printed and the parameters (such as the number of copies and whether to delete the file after printing) controlling the printing. This allows the spooler to resume printing automatically when reinstalled following an operating system reload.

swap

To copy a partition (user number) into memory or out of memory to a disk file. Swapping is managed by a partition managing program on multipartition operating systems or by the operating system itself on variable partition systems.

system common procedure

A system-common procedure performs a common system function, such as returning the current date and time. The code of the system-common procedure is included in the System Image and is executed in the same context and at the same priority as the invoking process. The Video Access Method, for example, is a system-common procedure. *See also* object module procedure.

system common service

A system service process that contains system-common procedures. *See also* system service.

System Image

Contains a run file (executable file) copy of the operating system ([Sys]<Sys>SysImage.sys).

system memory

An area of memory that is reserved for use by the operating system.

system partition

Contains the operating system or dynamically installed system services. *See also* application partition.

system request

Issued by the operating system to system services to notify the services of clients that are terminating or are being swapped out of memory.

system service

An operating system process that provides services to client processes. System service processes are of two types: request-based system services and system common services. Request-based system service processes serve requests submitted by client processes throughout the network; whereas, system-common services contain system-common procedures that can be used by clients at the local workstation. System service processes can be dynamically installed or linked with the System Image at system build.

T

TeleCluster

A system that supports connection of cluster workstations to a server over twisted-pair (telephone) wiring in a star configuration.

U

user number

A 16-bit integer that uniquely identifies the programs and/or the resources associated with a partition. A user number (historically the same as a partition handle) is not associated with a partition's particular size or physical location in memory, because partitions are not static memory cells into which programs are loaded: a partition is created at the time a program is loaded into memory and is removed when the program is terminated. Each application partition has a different user number. Processes in the same application partition share the same user number. A process obtains its user number with the GetUserNumber operation.

V

VAM

See Video Access Method.

Variable Length Parameter Block (VLPB)

Used by the Executive to communicate parameters to a succeeding application in the partition in which the VLPB is located. The VLPB is created in the long-lived memory of an application partition, and its memory address is stored in the Application System Control Block. *See also* Application System Control Block.

variable partition

Can use up to the maximum amount of memory specified at link time (when the program to be loaded into the partition was sized).

VDM

See Video Display Management.

VHB

See Volume Home Block.

Video Access Method (VAM)

Provides direct access to the characters and attributes of each frame. VAM can put a string of characters anywhere in a frame, specify character attributes for a string of characters, scroll a frame up or down a specified number of lines, position a cursor in a frame, and reset a frame.

Video Display Management (VDM)

Provides direct control over the way that the video appears. With it, an application program can determine the level of video capability, load a new character font into the font RAM, change screen attributes, stop video refresh, calculate the amount of memory needed for the character map based on the desired number of columns and lines and the presence or absence of character attributes, initialize each of the frames, and initialize the character map.

virtual 8086 mode

An operational mode supported by Intel microprocessors beginning with the 80386. In virtual 8086 mode, multiple operating systems, such as MS-DOS, can execute in memory in a multiprogramming environment. A region of memory is allocated and assigned the operating system characteristics of an 8086 microprocessor: the region provides a 1 megabyte address space within which a program can execute. Concurrently, application programs can execute in real mode or in protected mode in other memory regions. All executing programs have virtual machine capability. *See also* multiprogramming, Partition managing program, and virtual machine.

virtual memory

A technique that makes the apparent size of memory in an application partition (from the perspective of the application programmer) greater than its physical size. The primary mechanisms for the implementation of virtual memory are page swapping and segment swapping. (The use of program overlays is not considered virtual memory because it is not transparent to the application programmer.)

VLPB

See Variable Length Parameter Block.

volume

The medium of a disk drive that was formatted and initialized for CTOS with a volume name, a password, and volume control structures such as the Volume Home Block, the File Header Blocks, the Master File directory, and so forth. A floppy disk and the medium sealed inside a hard disk are examples of volumes.

volume control structures

Allow the file management system to manage (allocate, deallocate, locate, avoid duplication of) the space on the volume not already allocated to the volume control structures themselves. A volume contains a number of volume control structures: the Volume Home Block, the File Header Blocks, the Master File directory, and the allocation bit map, among others.

Volume Home Block (VHB)

There is a Volume Home Block for each volume. The VHB is the root structure (that is, the starting point for the tree structure) of information on a disk volume. The VHB contains information about the volume such as its name and the date it was created. The VHB also contains the memory addresses of the Log file, the System Image, the crash dump area, the allocation bit map, the Master File directory, and the File Header Blocks. The VHB is disk resident and one sector in size.

W

Wait

The kernel primitive that a client calls to be placed in the waiting state. *See* waiting state.

waiting state

One of three states in which a process can exist. A process is in the waiting state when it is waiting at an exchange for a message. A process enters the waiting state when it must synchronize with other processes. A process can only enter the waiting state by voluntarily issuing a Wait kernel primitive that specifies an exchange at which no messages are currently queued. The process remains in the waiting state until another process (or interrupt handler) issues a Send (or PSend, Request, or Respond) kernel primitive that specifies the same exchange that was specified by the Wait primitive. Any number of processes can be in the waiting state at a time. *See also* ready state and running state.

Workstation Agent

See cluster workstation agent.

X

X-Bus

The extensible bus used to connect modules on a modular CTOS workstation.

XVT

Sec Extensible Virtual Toolkit.

Index

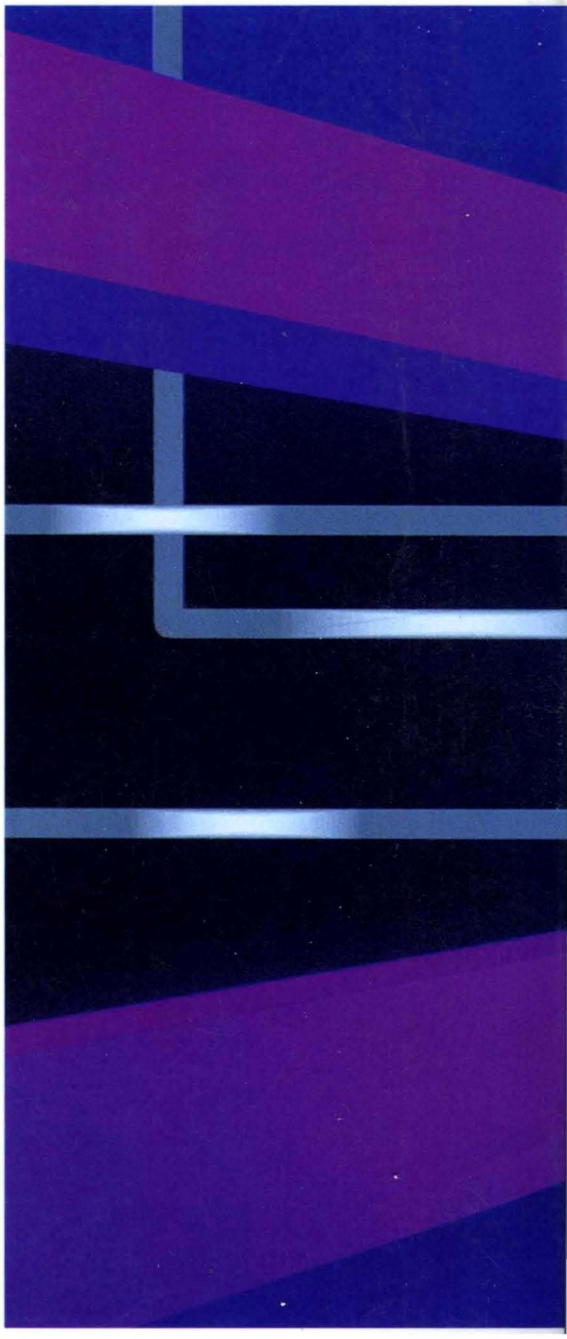
EXPLORING CTOS®

With over 800,000 units installed, the CTOS® operating system has still been called the best-kept secret in the computer world. Since 1981, CTOS has remained the only commercially available, message-based distributed operating system for microprocessor-based computers. Its modular architecture and unique networking capabilities are ideally suited for the needs of today's computer users and their work environment.

Exploring CTOS provides a conceptual overview of the CTOS architecture. Clear and easy-to-use, this book is a must-read selection for those interested in distributed operating systems, and for those interested in developing networked applications.

In this book you'll find out:

- What CTOS is and how it was developed.
- How this message-based operating system implements the client/server model.
- Why a message-based environment is ideal for easy development of networked or distributed applications.
- How CTOS provides built-in, transparent networking for distributed applications.
- How CTOS is modularized through the use of system services.
- How device-independent input/output provides an application with added flexibility.
- How CTOS provides multitasking for distributed applications.



ISBN 0-13-297342-1

90000>

9 780132 973427

PRENTICE HALL
Englewood Cliffs, N.J. 07632