VAXcluster Systems
Introduction to
VAXcluster Application Design

Order Number: AA–JP32B–TE

September 1989

This manual provides VAXcluster-specific information to the application designer and application programmer who are: migrating a VMS application or designing and coding a new application to run on a VAXcluster system.

Revision/Update Information: This is a revised manual.

Operating System and Version: VMS Version 5.2

Optional Software Versions:

Product:	Version:
DECintact	Version 1.0A
VAX ACMS	Version 3.0
VAX DBMS	Version 4.0
VAX DNS	Version 1.1
VAX PA	Version 2.0
VAX PCA	Version 2.0
VAX SPM	Version 3.2
VAX VMS/Rdb	Version 3.0

digital equipment corporation
maynard, massachusetts

First Printing, October 1987
Second Printing, September 1989

The Reader's Comments form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

CI	KDB50	VAX DNS
DECintact	MicroVAX	VAX Rdb/VMS
DECnet-VAX	MSCP	VAX RMS Journaling
DECwindows	PrintServer 40	VAX SPM
DIBOL	ReGIS	VAXstation
DSSI	UDA50	VAX Volume Shadowing
ESE20	VAX ACMS	VMS
Ethernet	VAXcluster	VMS RMS
HSC	VAX DATATRIEVE	
KDA50	VAX DBMS	digital™

HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS

USA*
Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

CANADA
Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL
Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA, Alaska, and Hawaii call 800–DIGITAL.

In Canada call 800-267-6215.

*Any order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminster, Massachusetts 01473.

Contents

PREFACE xi

CHAPTER 1 ADVANTAGES OF A VAXCLUSTER APPLICATION ENVIRONMENT 1–1

1.1 VAXCLUSTER SYSTEM HARDWARE ADVANTAGES 1–1
1.1.1 Hardware Availability 1–2
1.1.2 System and Data Availability 1–4
1.1.3 Mass Storage Device Availability 1–5
1.1.4 Workload Balancing 1–8
1.1.5 Supported VAXcluster Systems 1–9

1.2 VAXCLUSTER SYSTEM SOFTWARE ADVANTAGES 1–11
1.2.1 Connection Manager 1–12
1.2.2 Distributed VMS Lock Manager 1–13
1.2.3 Distributed File System 1–14
1.2.4 Distributed Job Controller 1–14

CHAPTER 2 USING THE VAXCLUSTER SYSTEM FOR APPLICATION DEVELOPMENT 2–1

2.1 TWO TYPES OF VAXCLUSTER APPLICATION ACTIVITY 2–2
2.1.1 I/O-Intensive Application 2–5
2.1.2 CPU-Intensive Application 2–7

2.2 CHARACTERISTICS OF AN APPLICATION SUITABLE FOR A VAXCLUSTER SYSTEM 2–9

2.3 APPLICATIONS THAT MAY NOT BE SUITABLE FOR A VAXCLUSTER SYSTEM 2–12

2.4 IS AN APPLICATION A CANDIDATE FOR DISTRIBUTION BY REPLICATION ACROSS VAXCLUSTER CPUS? 2–13

2.5 IS AN APPLICATION A CANDIDATE FOR DISTRIBUTION BY DECOMPOSITION ACROSS VAXCLUSTER CPUS? 2–16

CHAPTER 3 PROGRAMMING TOOLS FOR VAXCLUSTER APPLICATION DEVELOPMENT 3–1

3.1 VMS LOCK MANAGER 3–2
3.1.1 Lock Management System Services 3–3
3.1.2 Lock Modes 3–6
3.1.3 Locking Levels 3–8
3.1.4 Lock Queues 3–9
3.1.5 Lock Value Block 3–12
3.1.6 Using ASTs and Blocking ASTs for Synchronization of Interprocess Events 3–12

3.2 VMS RECORD MANAGEMENT SERVICES 3–14
3.2.1 VMS RMS and UIC-Based Protection 3–16
3.2.2 VMS RMS and Clusterwide Record Locking 3–16
3.2.3 VMS RMS Buffering and Global Buffering for a VAXcluster Application 3–17
3.2.4 VMS RMS and $QIO System Services 3–18
3.2.5 VMS RMS and XQP Operations 3–18

3.3 VMS BATCH FACILITY 3–19

3.4 CLUSTERWIDE PROCESS SERVICES 3–20
3.4.1 Process Control System Services 3–22
3.4.2 Process Information System Services ($GETJPI and $PROCESS_SCAN) 3–24

3.5 DECNET-VAX 3–25
3.5.1 Transparent DECnet-VAX Task-to-Task Communication 3–28
3.5.2 Nontransparent DECnet-VAX Task-to-Task Communication 3–29

3.6 SINGLE-NODE PROGRAMMING TOOLS NOT AVAILABLE CLUSTERWIDE 3–32

CHAPTER 4 APPLICATION DESIGN MODELS FOR VAXCLUSTER SOFTWARE 4–1

4.1 FILE SHARING MODEL 4–2

4.2 CLIENT-SERVER MODEL 4–6
4.2.1 One-to-One Client-Server Model 4–7
4.2.2 Many-to-One Client-Server Model 4–11

4.3 PARALLELISM MODEL 4–17

CHAPTER 5 DESIGNING DISTRIBUTED APPLICATIONS FOR A VAXCLUSTER SYSTEM 5–1

5.1 DESIGNING AN APPLICATION FOR INCREASED AVAILABILITY 5–1

5.2 DESIGNING AN APPLICATION FOR FASTER COMPLETION OF A TASK 5–3

5.3 DESIGNING AN APPLICATION FOR MAXIMUM THROUGHPUT 5–4

5.4 COMPARISON OF APPLICATION DESIGN MODELS WITH APPLICATION DESIGN GOALS 5–6

5.5 DESIGNING A VAXCLUSTER APPLICATION USING PRODUCTS CLOSELY ASSOCIATED WITH THE VMS OPERATING SYSTEM 5–7
5.5.1 VAX RMS Journaling 5–7
5.5.2 VAX Volume Shadowing 5–9
5.5.3 VMS DECwindows 5–10

5.6 DESIGNING A VAXCLUSTER APPLICATION USING LAYERED PRODUCTS BASED ON THE VMS OPERATING SYSTEM 5–12
5.6.1 DECintact 5–13
5.6.2 VAX ACMS 5–15
5.6.3 VAX DBMS 5–18
5.6.4 VAX Rdb/VMS 5–20
5.6.5 VAX DNS 5–21

CHAPTER 6 PROGRAMMING TECHNIQUES FOR VAXCLUSTER APPLICATIONS 6–1

6.1 REMOTE PROCESS CREATION 6–1
6.1.1 Using Transparent DECnet-VAX Communications 6–2
6.1.2 Using Nontransparent DECnet-VAX Communications 6–3
6.1.3 Using the VMS Batch Facility 6–5

6.2 DATA SHARING 6–16
6.2.1 Using DECnet-VAX Communications 6–17
6.2.2 Using VMS RMS to Control Record Granularity for Multiple Access 6–29
6.2.3 Using Read-Only Global Sections 6–33

6.3 PROCESS SYNCHRONIZATION 6–41
6.3.1 Using Clusterwide Process Services 6–41
6.3.2 Using Lock Management System Services 6–57
6.3.2.1 Using Simple Lock for Exclusive Access to a Shared Resource • 6–58
6.3.2.2 Using Completion ASTs and Blocking ASTs with Lock Management System Services to Synchronize Simultaneous Processes • 6–58
6.3.2.3 Using the Lock Value Block to Pass Information • 6–64
6.3.3 Coordinating Processes Using DECnet-VAX Communications 6–69

6.4 EXCEPTION CONDITIONS 6–74
6.4.1 Recovery from Interprocess Communication Failures 6–75
6.4.2 Recovery from Cluster State Transition Due to Node Failure 6–81

CHAPTER 7 VAXCLUSTER SYSTEM PERFORMANCE CONSIDERATIONS 7–1

7.1 USING VMS UTILITIES TO MONITOR THE CLUSTER AND IDENTIFY BOTTLENECKS 7–2

7.2 POTENTIAL BOTTLENECKS FOR AN I/O-BOUND APPLICATION ENVIRONMENT 7–4

7.3 POTENTIAL BOTTLENECKS FOR A MEMORY-BOUND APPLICATION 7–13

7.4 POTENTIAL BOTTLENECKS FOR A CPU-BOUND APPLICATION ENVIRONMENT 7–14

7.5 LAYERED PRODUCTS AVAILABLE FOR MONITORING CLUSTER PERFORMANCE 7–16
7.5.1 VAX Software Performance Monitor 7–17
7.5.2 VAX Performance and Coverage Analyzer 7–17
7.5.3 VAX Performance Advisor 7–18

CHAPTER 8 SAMPLE APPLICATION FOR A VAXCLUSTER SYSTEM 8–1

8.1 APPLICATION DESIGN 8–4

8.2 APPLICATION IMPLEMENTATION 8–9

GLOSSARY Glossary–1

INDEX

FIGURES

1–1 Conceptual Relationship of the Four Software Components in the VAXcluster System 1–12
2–1 Comparing the Direction of Information Flow for an I/O-Intensive Application Distributed by Replication and a CPU-Intensive Application Distributed by Decomposition 2–4
2–2 I/O-Intensive Application (PROG_A1) Executing Multiple Copies on a VAXcluster System 2–6
2–3 Distributing Work for a CPU-Intensive Application Executing on a VAXcluster System 2–8
2–4 Sample Banking Application on a VAXcluster System 2–17
3–1 Functional Relationship of VAXcluster System Programming Tools 3–2
3–2 VAXcluster Programming Tool: VMS Lock Manager 3–3
3–3 Resource Granularity Locking 3–8
3–4 Lock Queues of the VMS Lock Manager 3–10
3–5 VAXcluster Programming Tool: VMS RMS 3–14

3–6	Application Software Levels Interfacing with the VMS Lock Manager	3–15
3–7	VAXcluster Programming Tool: VMS Batch Facility	3–19
3–8	VAXcluster Programming Tool: Process Control and Process Information System Services	3–21
3–9	VAXcluster Programming Tool: DECnet-VAX	3–25
3–10	Transmitting DECnet-VAX Task-to-Task Data	3–28
3–11	One-to-Many or Many-to-One Nontransparent Communications	3–31
4–1	File Sharing Model Using Distribution by Replication	4–2
4–2	Log In to the Application	4–5
4–3	Search and Display All Available Courses According to a Keyword	4–5
4–4	Select a Course and View a Course Description	4–6
4–5	Register or Withdraw from a Selected Course	4–6
4–6	One-to-One Client-Server Model	4–7
4–7	Using a One-to-One Client-Server Model for Parallel Processing	4–10
4–8	Many-to-One Client-Server Model as a File Server	4–12
4–9	Many-to-One Client-Server Model as a File Server for a Corporate Database	4–17
4–10	Parallelism Model	4–18
4–11	Parallelism Model Using Self-Scheduling and Queueing	4–19
4–12	Car Crash Simulation	4–24
5–1	Running VMS DECwindows in a VAXcluster System	5–12
5–2	The Components of DECintact	5–13
5–3	Running VAX ACMS on Two VAXcluster CPUs	5–16
6–1	Flow Diagram for Programming Example 1	6–7
6–2	Coordinating the Use of Process Control System Services with the Process Information System Services	6–42
6–3	Deadman Lock Scheme	6–59
6–4	Doorbell Lock Scheme	6–61
6–5	Format of Lock Status Block	6–65
6–6	Determining Who is First with a Lock Value Block Scheme	6–68
6–7	A Process Designed for Failover in a VAXcluster System	6–82
7–1	I/O Pathways in a Mixed-Interconnect VAXcluster System	7–5
8–1	Diagram of a Demonstration Application	8–2
8–2	Function of Demonstration Application	8–3
8–3	SERVER.EXE Set-Up Activities	8–5
8–4	USER_IFACE.EXE Message Sending Activities	8–6
8–5	RR_AST Routine Broadcast Protocol Activities	8–8
8–6	Modules for Demonstration Application	8–10

TABLES

1–1	Comparing the CI-Based, Local Area VAXcluster, and Mixed-Interconnect VAXcluster Systems	1–10
2–1	CI Adapter Throughput Capacity (Fastest to Slowest)	2–14
2–2	Ethernet Adapter Throughput Capacity (Fastest to Slowest)	2–14
2–3	Disk Type I/O Rates (Fastest to Slowest)	2–14
2–4	Goals for Distributing an Application Across VAXcluster CPUs	2–18
3–1	Lock Management System Services	3–4
3–2	Lock Modes	3–6
3–3	Lock Mode Compatibility	3–7
3–4	Multiple Lock Requests Creating a Deadlock	3–11
3–5	Supported VMS System Services for Process Control	3–22
3–6	Process Control System Services Status Codes	3–23
3–7	Using Nontransparent DECnet-VAX Communication	3–30
3–8	Transparent and Nontransparent DECnet Communication	3–32
5–1	Comparison of Application Design Models with Application Design Goals	5–6
6–1	Required Modules for Two-Way, Transparent, Task-to-Task Communications	6–17
6–2	Parameters for Lock Management System Services	6–57
7–1	VMS Utilities and Commands for Monitoring a VAXcluster System	7–3
7–2	Summary of VMS Documentation Resources for Identifying Bottlenecks	7–3
7–3	Layered Products for Monitoring a Cluster	7–16

Preface

A VAXcluster system provides you with greater availability of computer processors and data storage than a single VMS system. This manual describes how to design and implement an application program to take advantage of the hardware and software features of a VAXcluster system.

VAXcluster systems include:

- CI-based VAXcluster systems in which CPUs are interconnected by a CI bus
- Local Area VAXcluster (LAVc) systems in which CPUs are interconnected by an Ethernet cable
- Mixed-Interconnect VAXcluster (MIVc) systems in which CPUs are interconnected by a CI bus **and** an Ethernet cable

This manual refers to these VAXcluster types (CI-based, LAVc, and MIVc) as one entity, a VAXcluster system, unless otherwise stated.[1]

Intended Audience

This manual is written for the application designer or application programmer who is designing and implementing a business/commercial application or a scientific/engineering application on a VAXcluster system.

This manual assumes that the application designer and application programmer are familiar with the VMS operating system, have experience in programming a high-level language, and can determine when an application is I/O-intensive or CPU-intensive.

The primary focus of this manual is for programming development in a common-environment VAXcluster system using RMS files.

Purpose of This Document

The purpose of this manual is to provide VAXcluster information to the application designer and application programmer engaged in designing a new application or migrating an existing VMS application to a VAXcluster system. This manual is structured to aid the application designer and application programmer in making application design decisions based on programming tools, application design models, and implementation techniques to optimize the processing capability of their VAXcluster system.

[1] To make this manual easier to read, the word cluster is frequently used to refer to a VAXcluster system.

Goals

This manual presents:

- The VAXcluster concept of system availability based on the features of VAXcluster hardware and software components
- A discussion of an I/O-intensive application and a CPU-intensive application on a VAXcluster system
- A discussion of application types appropriate for use in a VAXcluster system
- VMS programming tools available for the development of VAXcluster software
- Design models for VAXcluster applications with conceptual evaluations of their implementation requirements
- Programming techniques for implementing interprocess communication and synchronization requirements for a VAXcluster application
- Programming techniques for designing an application for greater speed, increased availability, and maximum throughput
- General performance considerations related to I/O-bound, memory-bound, and CPU-bound VAXcluster applications
- A brief product summary of some Digital software products that an application designer may consider for VAXcluster application development (for example, DECintact and VAX ACMS)
- A programming example of an application on a VAXcluster system using some of the models and programming implementation techniques explained in the manual

Non-Goals

This manual does **not** present:

- Programming techniques for a single VAX processor
- Programming techniques for distributing an application across a network
- Parallel programming techniques for a Symmetric Multiprocessor (SMP) VAXcluster node
- Programming techniques for system-level applications (for example, device drivers)
- Guidelines for when to use a VAXcluster system rather than an SMP system to implement a paralleled application
- Guidelines for performing system analysis
- Guidelines for configuring VAXcluster hardware for a specific application
- An evaluation of performance for the application design models based on hard data

- Decision analysis tools for comparing the relative merits of the application design models with the use of one or more VAX products (for example, DECintact and VAX ACMS)
- Information on VAXcluster system management
- Techniques for use of specific programming languages
- VAXcluster hardware and software internals

Associated Documents and Referenced Document

This manual assumes that the application designer and application programmer are familiar with the VMS Documentation Set.[1] When appropriate, this manual provides references to manuals in the VMS Documentation Set. In addition, this manual provides references to the following documentation:

Associated Documents

- *Guidelines for VAXcluster System Configurations*
- *Guide to VAX SPM*
- *HSC User Guide*
- *VAXcluster Software V5.2 Software Product Description 29.78.02*
- *VAX Distributed Name Service Management Guide*
- *VMS Operating System, Version 5.2, Software Product Description 25.01.32*
- *VAX Performance Advisor User's Guide*
- *VAX Performance and Coverage Analyzer User's Guide*
- *VAX RMS Journaling Manual*
- *VAX Systems and Options Catalog*
- *VAX Volume Shadowing Manual*

Referenced Document

Refer to the *VMS VAXcluster Manual* for a summary of the supported VAXcluster configurations, a discussion of quorum, and a brief description of the following VAXcluster software components:

- Connection Manager
- Distributed File System
- Distributed Lock Manager
- Distributed Job Controller
- Mass Storage Control Protocol (MSCP) Server

[1] When referring to the VMS Documentation Set for information pertaining to Section 3.4, Clusterwide Process Services, and Section 6.3.1, Using Clusterwide Process Services, use the *VMS Version 5.2 New Features Manual* if you are not accessing VMS Version 5.2 documentation using the DECwindows Bookreader.

Organization of This Manual

The contents of this manual are as follows:

Chapter 1 discusses the hardware and software advantages that a VAXcluster system provides for application development.

Chapter 2 presents the concepts of an I/O-intensive and a CPU-intensive application in the context of a VAXcluster environment, and describes the characteristics of an application suitable for a VAXcluster system.

Chapter 3 explains the VMS programming tools that are available for developing an application on a VAXcluster system.

Chapter 4 explores three potential application design solutions. After determining if your application will be I/O-intensive or CPU-intensive, you can use the appropriate application design model presented in this chapter.

Chapter 5 presents three application design goals as related to the application design models discussed in Chapter 4. Also, this chapter describes other Digital software products (for example, DECintact and VAX ACMS) that you can use for application design on a VAXcluster system.

Chapter 6 demonstrates some of the available programming techniques that you can use to implement the application design models presented in Chapter 4.

Chapter 7 discusses performance considerations for the three primary system resources (CPU, memory, and I/O system) in your VAXcluster system.

Chapter 8 presents a sample application to demonstrate a distributed application that is designed to use lock management system services for explicit interprocess communications.

Conventions

This manual uses the following conventions:

Convention	Meaning
Bold	Bold typeface indicates emphasis.
Italics	Italicized typeface indicates a manual or a new term located in the Glossary.
`code example`	Smaller typeface indicates programmable code.

1 Advantages of a VAXcluster Application Environment

A VAXcluster system is a loosely-coupled multiprocessor system comprised of standard VAX computers. A VAXcluster system contains separate processors and memories connected by a message-oriented interconnect, running instances of the same copy of the VMS operating system. The VAXcluster system is thus limited to a bounded number of nodes within a local geographical scope. The interconnection or *clustering* of processors and storage controllers in a VAXcluster system, along with the VMS operating system's distributed components, enable you to design and run applications in a common system environment.

Working on a VAXcluster system, you can design and implement an application to take advantage of the VAXcluster system's multiple CPU resources, as well as VAXcluster shared disk and print resources. With the proper use of the available programming techniques, a wide variety of VAXcluster application environments can be supported, intrinsically, with no need to redesign or recompile a VMS-based application that is being migrated from a standalone VMS system.

Chapter 1 presents the hardware and software advantages of the VAXcluster environment as an overview for the reader who is unfamiliar with the VAXcluster concept. If you are already familiar with the VAXcluster concept, it is suggested that you begin reading Chapter 2, Using the VAXcluster System for Application Development.

1.1 VAXcluster System Hardware Advantages

VAXcluster system hardware provides a site with growth potential, extensive data storage potential, and configuration flexibility. The hardware advantages of the VAXcluster system are:

- Hardware Availability

  Using redundant hardware components, a VAXcluster system can be configured to survive single hardware failures. (See Section 1.1.1.)

- System and Data Availability

  The system environment of a VAXcluster system is flexible; the VAXcluster system can be configured as a common-environment or a multiple-environment. (See Section 1.1.2.)

- Mass Storage Device Availability

  The VAXcluster system offers many options for sharing mass storage devices. (See Section 1.1.3.)

- Workload Balancing

  By using the capability of a VAXcluster system to support multiple CPUs, terminal servers connected to the Ethernet, and clusterwide batch and print queues, you can balance the cluster's workload. (See Section 1.1.4.)

1.1.1 Hardware Availability

The two key elements to providing a high level of hardware availability are system *modularity* and *redundancy*. Modularity provides independence of multiple components in the VAXcluster system so that the failure of one component has minimal effect on the system. Redundancy ensures that a particular component has a counterpart so that one component is always available, even if the other is not. VAXcluster systems can provide automatic *recovery* from hardware failures through hardware redundancy provided by:

- Multiple-access paths to disk and disk failover
- Multiple CPUs
- Multiple printers
- Terminals connected with Local Area Transport (LAT) hardware and software
- CI cables and the Star Coupler connections
- VAX Volume Shadowing

Multiple-Access Paths to Disk and Disk Failover

The VMS operating system supports *multiple-access paths* to disks and failover of disks between pairs of HSC subsystems, between local controllers, and between disk servers. Failover occurs when one controller or cable malfunctions and causes one path to break. When the path breaks, the device using that path automatically fails over to the other path. Failover of disk drives between HSC subsystems, between local controllers (for example, UDA50, KDA50, KDB50), or between disk servers helps to provide high data availability. Note that when a disk server fails in configurations that include multiple servers, satellite access to disks fails over to another server.

Multiple CPUs

The processing power of all CPUs is potentially accessible to all users in the cluster. That is, in a common-environment system, the same application can be run on all CPUs. Therefore, if one CPU is removed from the cluster, the application is still available through other CPUs.

In a VAXcluster system, multiple CPUs can function as *disk server* nodes. By configuring a primary and secondary disk server for a VAXcluster satellite node, that satellite can have multiple-access paths to a local or HSC-connected disk. If the primary disk server fails, the disk access request is processed by the secondary disk server.

Also, multiple CPUs on a VAXcluster system increase the availability of batch queues for clusterwide batch requests. A batch job can be executed on any available CPU; if a CPU executing a batch job fails, the job may be restarted (depending on the type of job) on another CPU.

Multiple Printers

A VAXcluster system can have multiple print queues which afford the application clusterwide or local access to printers. Redundant print queues and clusterwide print queue *failover* capabilities provide flexibility and printer availability for the users. If a print job fails due to a printer malfunction, the job can be requeued to the print queue for an operating printer.

Terminals Connected with LAT Hardware and Software

Terminals with LAT connections are connected to a terminal server on the Ethernet and run the Local Area Transport (LAT) protocol. These terminals have access to all VAXcluster CPUs through the Ethernet, unless the LAT software for the terminal server is set up otherwise. If all user terminals are connected to the VAXcluster system through a single terminal server, the terminal server can become a single point of failure for user access to a VAXcluster system.

CI Cables and the Star Coupler Connections

CI cables, the Star Coupler, and the Computer Interconnect Star Coupler Extender (CISCE) are designed with dual paths. If one path fails, the remaining path automatically maintains the message traffic.

VAX Volume Shadowing

VAX Volume Shadowing is an optional product that duplicates data on one or more disks in a *shadow set*. In a shadow set:

- The data on each disk of the shadow set is identical to the data on all other disks in the set. Write operations from any VAXcluster CPU are directed to all disks in the set. When data is updated on one disk, it is automatically updated on all disks. If a disk becomes unavailable, users still have access to the same data from another shadow set member.
- Read requests from any VAXcluster CPU are distributed over the shadow set to improve *performance*.

VAX Volume Shadowing can provide a marked performance increase in certain applications. For additional information on Volume Shadowing, see the *VAX Volume Shadowing Manual*.

1.1.2 System and Data Availability

Based on the application and performance, the VAXcluster system environment can be configured as one of the following:

- Common-environment, for ease of use and ease of management
- Multiple-environment, for specialized needs
- Combination of common-environment and multiple-environment

Common-Environment VAXcluster Configuration

A *common-environment* VAXcluster configuration offers high availability of *resources* for applications, because it lets you run the same application on all VAX CPUs in the VAXcluster system. The main characteristics of a common-environment cluster are:

- A startup command procedure sets up the same system environment for all VAXcluster CPUs with:
  - Identical installed known images
  - Identical logical names defined at startup
  - Same set of shared mass storage devices and queues
- A common-environment VAXcluster configuration uses a single User Authorization File (UAF) to set up the same user environment for all VAXcluster CPUs. The UAF defines which users can log on, and the environment of each user, such as:
  - Default disk
  - Default directory
  - User privileges
  - User Identification Code (UIC)

The UIC determines access to resources and provides the basis for clusterwide protection. Users and applications have the same data access from each CPU. If a CPU fails or is removed, the user can log in on any other CPU in the cluster and continue working.

Multiple-Environment VAXcluster Configuration

A *multiple-environment* VAXcluster configuration is defined by the system manager. The main characteristics of a multiple-environment cluster are:

- Each CPU in a VAXcluster system can be configured with its own known images, logical names, and devices that are different from other VAXcluster CPUs. Therefore, certain software, devices, and files on one CPU may not be available to other CPUs in the cluster.
- Each CPU in the cluster can have a different UAF.
- An application using multiple CPU resources may require a more complicated design.

However, a multiple-environment configuration does not take advantage of the unique accessibility features of a VAXcluster system. A failure on one CPU affects all users on that CPU because they may be unable to access their data from another CPU.

Combination of Common-Environment and Multiple-Environment VAXcluster Configuration

Most multiple-environment clusters are actually a combination of common-environment and multiple-environment characteristics that include either:

- All CPUs sharing some subset of the resources
- Some CPUs sharing all resources, while other CPUs have their own nonshared resources

For example, you can have a three-CPU VAXcluster system in which two CPUs are sharing the same user environment, queues, and access to mass storage devices, and the third CPU is restricted to the personnel department accessing payroll files. For more information on common-environment and multiple-environment operating system configurations, refer to the *VMS VAXcluster Manual*.

1.1.3 Mass Storage Device Availability

The VAXcluster system offers many options for configuring mass storage devices. Typically, a VAXcluster environment includes some of the following configurations:

- Disks available clusterwide through an HSC controller
- Disks available clusterwide from a CPU
- Disks dual-ported between HSC controllers
- Disks dual-ported between CPUs
- DSSI-connected disks in dual-host LAVc configurations
- Disks that are part of a shadow set
- System disks
- Tape configurations

Disks Available Clusterwide Through an HSC Controller

This disk configuration offers the capability for clusterwide shared disk resources (except in a LAVc; see Table 1–1). Two important advantages for sharing disks are:

- Users and applications on multiple CPUs can access the same data.
- Users and applications have access to the same files from any CPU in the cluster. Therefore, if one CPU fails, the disk data is still available from another CPU.

Any disk in a cluster **can** be accessible clusterwide.

Disks Available Clusterwide from a CPU

Disks connected locally to a VAX CPU require additional system manager or user action to be made available clusterwide. (Refer to the *VMS VAXcluster Manual* for more information about using the MSCP server to make local disks available clusterwide.)

Disks Dual-Ported Between HSC Controllers

In a VAXcluster system (except LAVc; see Table 1–1), a dual-ported disk can be physically connected between two HSC nodes. Thus, a dual-ported disk has multiple-access paths and it can be accessed clusterwide in a coordinated way through either HSC controller. When a disk is dual-ported and one of the HSC controllers to which it is connected fails, the remaining HSC controller still provides access to the disk. For more information about dual-ported disks, refer to the *VMS VAXcluster Manual* and the *VAXcluster Software V5.2 Software Product Description 29.78.02*.

Disks Dual-Ported Between CPUs

In a VAXcluster system, a dual-ported disk can be physically connected between two CPU nodes. Thus, a dual-ported disk has multiple-access paths and it can be accessed clusterwide. When a disk is dual-ported and one of the local controllers to which it is connected fails, the remaining disk controller still provides access to the disk. This is done transparently by software when the active controller fails.

DSSI-Connected Disks in Dual-Host LAVc Configurations

In dual-host configurations of the MicroVAX 3300/3400 or 3800/3900, you can connect Integrated Storage Element (ISE) disks (RF series disks) on a Digitial Small Storage Interconnect (DSSI) bus between pairs of MicroVAX 3300/3400 or 3800/3900 CPUs set up as boot/disk servers. The disks are simultaneously accessible to both servers and can be served to all satellites by either server. If one of the servers fails, the remaining server and satellites can continue to access the disks. For more information on dual-host configurations, refer to the *VAX Systems and Options Catalog* and the appropriate installation and operations guides.

Disks that are Part of a Shadow Set

The shadow set created with the VAX Volume Shadowing product is a single virtual unit designed to protect against hardware failure. When the shadow set members are dual-ported between two HSC controllers and one HSC controller fails, the first CPU in the cluster to access the shadow set reconstructs the shadow set for the entire cluster, and switches the set to the alternate HSC controller. For more information about shadow sets, refer to the *VAX Volume Shadowing Manual*.

System Disks

When using a common system disk for a VAXcluster system, the following characteristics apply:

- All CPUs share a common directory.
- All operating system and layered product files are stored in a common root directory, and therefore need to be installed only once to be available to all VAXcluster CPUs.
- Each CPU has a unique root directory where node-specific information is stored, such as the DECnet-VAX node name.
- In a CI-based VAXcluster system, a common system disk must be connected to an HSC. In a MIVc system, the system disk does not need to be connected to an HSC and the LAVc rules for system disks apply. In a LAVc system, a common system can be a locally connected disk on a LAVc *boot node*. And, in a dual-host configuration for a LAVc system, any disk on the DSSI-connected ISE can be a common system disk.

When you design your application, it is important to remember that if a single, common system disk fails, all CPUs booting from that disk also fail.

Therefore, there are some advantages to using multiple, common system disks instead of a single, common system disk:

- For systems that do not have a single, common system disk that is volume shadowed, if a system disk fails, multiple common system disks enable the unaffected CPUs in the VAXcluster system to keep running.
- There is less likelihood of an I/O bottleneck when a VAXcluster system has multiple, common system disks.

It is also possible to have a local system disk for a VAXcluster CPU. However, for each local system disk, the system manager must install the known images, logical names, and queue resources to initialize the system environment for the booting CPU. Also, it is possible to create a system environment for a VAXcluster system using a combination of: local system disks; a single, common system disk; and multiple, common system disks. For more information about setting up a system disk, refer to the *VMS VAXcluster Manual*.

Tape Configurations

As with disks, you can connect tape drives to:

- Individual CPUs
- An HSC controller
- A pair of HSC controllers

If a tape drive is connected to an individual CPU, it is available to that CPU only. If a tape drive is connected to an HSC, it is available to any CI-connected CPU in the VAXcluster system through that HSC controller.

VMS supports dual-ported tape drives between two HSC controllers. If one HSC controller fails, the tape drive mounted through that HSC controller automatically fails over to the remaining HSC controller. The tape is then available to the application through the remaining HSC controller.

1.1.4 Workload Balancing

The flexibility of a VAXcluster system offers possibilities for *workload balancing*. The VAXcluster system provides three methods to balance the workload:

- CPU power
- Distributed print and batch queues
- Terminal configurations

CPU Power

If the VAXcluster application environment is compute-bound, the workload throughput can be improved by adding an additional CPU to the VAXcluster system. The additional cluster member increases overall VAXcluster throughput by providing more computing power at times of high load. In addition, during nonpeak periods, an additional CPU helps to ensure high availability in the event that a VAXcluster CPU fails.

Distributed Print and Batch Queues

The VAXcluster system enables the system manager to define clusterwide generic batch and print queues. Any batch or print job that is queued to the generic batch or generic print queue is automatically executed on the batch or print queue of the VAXcluster system with the most available capacity.

Terminal Configurations

Terminals in a VAXcluster system can be connected to:

- Individual CPUs
- A terminal switch
- A *terminal server* connected to the Ethernet to which the VAXcluster system is also connected

If you connect a terminal to an individual CPU, the terminal can access the cluster only when that CPU is available.

If you connect one or more terminals to a terminal switch or terminal server:

- Terminal use is independent of the availability of any one CPU.
- Each terminal has access to all CPUs in the cluster.

With terminals connected to a terminal server, the Ethernet cable connects the terminal server with CPUs and information processing products in the cluster. Terminal I/O performance is essentially the same as a terminal directly connected to a CPU. However, terminal servers provide a variety of unique benefits not provided by locally-connected terminals or terminal switches. These benefits include:

- Multiple terminal access to the same or different CPUs that are connected to the Ethernet and are running the LAT protocol.
- Job load balancing at login time to the VAXcluster CPU with the greatest available computing capacity. This can occur as long as a cluster service group name, known to the LAT protocol, has been assigned. Refer to the *VMS VAXcluster Manual* for more information on using the cluster service group name.
- Failover access to another node if the current node fails. The terminal server does not provide automatic failover when a CPU fails. However, if the CPU to which you are connected fails, the terminal server lets you switch your session to another CPU. Assuming a common-environment cluster, you can then log in to that CPU and reestablish the same application.
- Terminal servers offload some of the I/O overhead from the VAXcluster CPUs, permitting the CPUs to devote more time to the application.
- Terminal locking to prevent unauthorized use of a terminal, thus providing increased data security.

1.1.5 Supported VAXcluster Systems

There are three types of VAXcluster configurations:

- Computer-Interconnect VAXcluster system (CI-based)
- Local Area VAXcluster system (LAVc)
- Mixed-Interconnect VAXcluster system (MIVc)

Table 1–1 compares the major hardware components of the CI-based VAXcluster system, Local Area VAXcluster system, and Mixed-Interconnect VAXcluster system.

Table 1–1 Comparing the CI-Based, Local Area VAXcluster, and Mixed-Interconnect VAXcluster Systems

Characteristic	CI-Based	LAVc	MIVc
Communication bus	Dual-path CI bus	Single-path Ethernet	Both CI bus and Ethernet
Supported CPUs[1]	CPUs with a CI-port	CPUs with an Ethernet-port	All CPUs require an Ethernet-port, and CPUs in CI-portion require a CI-port
HSC controllers	Yes	No	Yes, on the CI-portion
Multiple-access paths to disk	Dual porting of certain disks between HSCs, or between CPUs, with automatic failover. Also, dual-host, RF-series access.	Dual porting of certain disks between CPUs with automatic failover. Also, dual-host, RF-series access.	Dual porting of certain disks between HSCs with automatic failover on the CI-portion and dual porting of certain disks between CPUs with automatic failover. Also, dual-host, RF-series access.
VAX Volume Shadowing	Yes	No	Yes, on the CI-portion[3]
Single common system disk[2]	Supported	Supported	Supported
Multiple common system disks[2]	Supported	Supported	Supported
Local system disk[2]	Supported	Supported	Supported
Disks available clusterwide	Yes	Yes	Yes

[1]Refer to the *VAXcluster Software V5.2 Software Product Description 29.78.02* and the *Guidelines for VAXcluster System Configurations* for a list of the supported CPUs for each type of VAXcluster system.

[2]Refer to the *VMS VAXcluster Manual* for a discussion on how to configure a system disk for each type of VAXcluster system.

[3]Shadow sets can also be served to *satellite* nodes by a CI-based CPU acting as a disk server.

1.2 VAXcluster System Software Advantages

While most software components available to a VAXcluster system are the same as those available to a programmer using a single VAX CPU, some VMS software components have enhanced features for the VAXcluster system environment. These software components are:

- Connection Manager

  The *connection manager* manages membership of VAXcluster CPUs in the cluster. (See Section 1.2.1.)

- Distributed VMS Lock Manager

  The *distributed VMS lock manager* synchronizes the use of clusterwide shared resources. (See Section 1.2.2.)

- Distributed File System

  The *distributed file system* coordinates file operations throughout the cluster. (See Section 1.2.3.)

- Distributed Job Controller

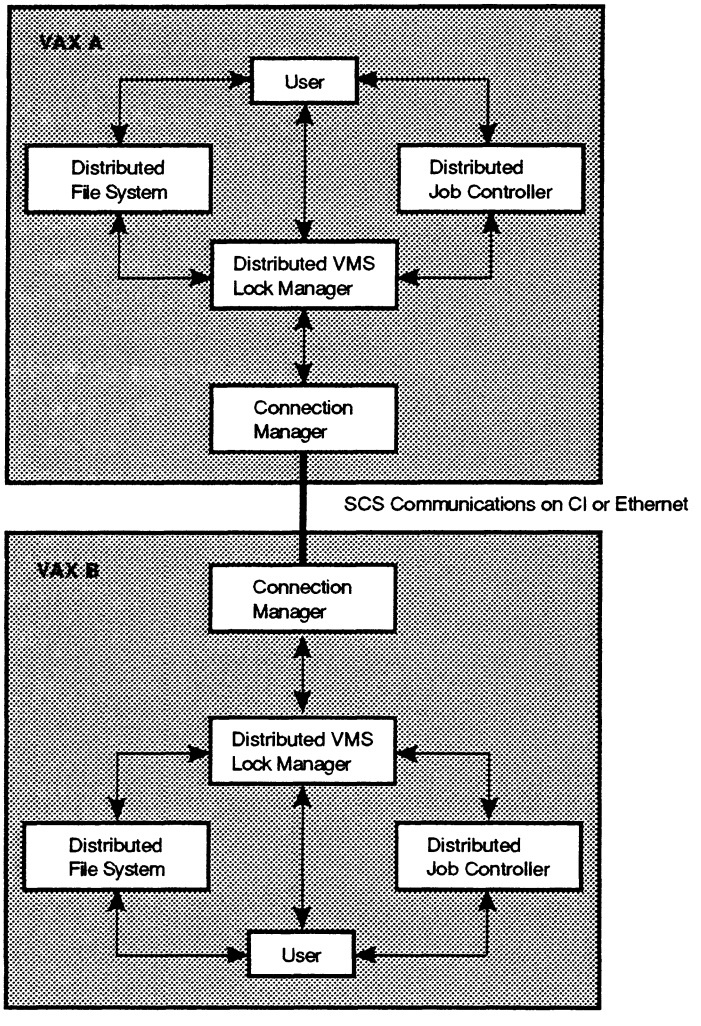
  The *distributed job controller* manages clusterwide batch and print queues. (See Section 1.2.4.)

These software components, resident on each node as part of VMS, intercommunicate to maintain a consistent view of the *clusterwide lock database* and the *cluster membership*. These four software components, resident on every VAXcluster CPU, are highly integrated in a VAXcluster system and act as a distributed operating system.

Figure 1–1 represents the central importance of the distributed VMS lock manager for the synchronization of the VAXcluster file system. With the distributed VMS lock manager, the VAXcluster file system uses the concept of clusterwide access for individual processes, and each *process* can uniquely own a mass storage resource. In a VAXcluster system, these four software components communicate to:

- Dynamically update the locking database for changes in clusterwide, lock ownership and cluster membership
- Adjust the cluster membership for a VAXcluster member node joining or leaving the cluster
- Provide coordinated access to the distributed file system
- Distribute print and batch job requests to the most available clusterwide queue, and restart batch and print jobs in the event of a VAXcluster CPU failure

Figure 1–1 Conceptual Relationship of the Four Software Components in the VAXcluster System



1.2.1 Connection Manager

The connection manager:

- Has no user programming interface
- Runs on each active CPU in the cluster

- Determines cluster membership and coordinates *cluster state transitions*

  The connection manager creates a cluster when the first active CPU is booted, and manages cluster reconfigurations as additional CPUs join the cluster or existing CPUs leave the cluster. As a cluster configuration changes, a cluster state transition occurs.

- Prevents cluster partitioning

  A partitioned cluster exists when two or more cluster members, each unaware of the others, share the same data resources. *Partitioning* can cause disk file corruption since there must be coordination between all VAXcluster CPUs sharing the resource.

  The connection manager prevents partitioning by using System Communication Services (SCS) and a quorum-based algorithm. Proper use of the *quorum* scheme requires participation by the cluster system manager. For more information on the implementation of the quorum scheme and its use by the system manager, refer to the *VMS VAXcluster Manual*.

1.2.2 Distributed VMS Lock Manager

In a VAXcluster environment, the VMS operating system provides the same data integrity as it does on a single VAX CPU because the VMS lock manager is distributed between all cluster members. The distributed VMS lock manager:

- Is accessed through the same programming interface in a cluster environment as on a standalone VMS system
- Is used by several VMS software components to coordinate the sharing of VAXcluster resources, such as data, devices, print and batch queues. Access to shared resources is transparent to the user.

  In a VAXcluster system, the distributed VMS lock manager runs on each VAXcluster CPU to enable applications to coordinate the sharing of resources. The distributed VMS lock manager is used by:

  — The distributed file system

  — VMS Record Management Services (VMS RMS)

  — The distributed job controller

  — User-written VAXcluster applications

- Exists on each member of the cluster
- Stores *resource names* and information in a name space called resource blocks
- Stores lock information in the clusterwide lock database, and communicates this information to all cluster members

See Section 3.1, VMS Lock Manager, for more information.

1.2.3 Distributed File System

The distributed file system has enhanced capabilities in a VAXcluster system. It allows:

- All available disk volumes to appear to any process as if they are local to every VAXcluster CPU
- All cluster users to have the same access to disk volumes and disk files clusterwide

To access disks directly connected to VAX CPUs, the distributed file system uses software called Mass Storage Control Protocol (MSCP) Server. The Mass Storage Control Protocol (MSCP) Server implements the MSCP protocol, which is used to communicate with a controller for Digital Standard Architecture (DSA) disks. For more information on using the MSCP Server, refer to the *VMS VAXcluster Manual*.

The functions of the distributed file system include:

- Creating, deleting, extending, and truncating files
- Maintaining file directories
- Mapping virtual to logical I/O
- Arbitrating runtime access to files
- Enforcing file protection
- Enforcing and maintaining disk usage quotas

Because these activities can affect shared data, they must be coordinated. This coordination is achieved by distributing locking information throughout the cluster. The distributed file system uses the distributed VMS lock manager's clusterwide lock database to arbitrate access to shared file resources. For more information on the programmer's use of VMS RMS to interface with the distributed file system, see Section 3.2, VMS Record Management Services. For more information on non-RMS use of the distributed file system, see the *VMS I/O User's Reference Manual*.

1.2.4 Distributed Job Controller

The distributed job controller lets users submit batch and print jobs to generic queues. It then distributes the print or batch processing workload across cluster nodes, using the distributed VMS lock manager to signal each local job controller on each VAXcluster CPU to examine its own queues for print or batch jobs that need processing. The distributed job controller lets a print or batch job execute on any clusterwide queue from any cluster node. Refer to Section 3.3 for information on how you can use the VMS Batch Facility to interface with the distributed job controller.

2 Using the VAXcluster System for Application Development

The application environment of a VAXcluster system offers you the same VMS environment as a single VMS system. Consequently, in most cases, you will not need to redesign your application in order to make your application "work" on a VAXcluster system. When you run an application written for the VMS operating system on a VAXcluster system, either one of two possible situations will occur:[1]

- Application works clusterwide
- Application works on only one node in the cluster

The rationale for some applications working clusterwide and others only working on a single node is based on the technology of a VAXcluster system. When the VAXcluster system was designed, some parts of the VMS operating system were distributed to all VAXcluster members (for example, the distributed VMS lock manager, distributed file system, and distributed job controller), and some parts of the VMS operating system were not distributed (for example, management of CPU memory). The distributed parts of the VAXcluster VMS operating system support clusterwide processes. However, applications that depend on an undistributed part (CPU memory management) for interprocess communications cannot be distributed across nodes in a VAXcluster system.

There are several programming mechanisms that enable VMS processes to communicate with each other. Some of these mechanisms do not work between processes executing on different VAXcluster CPUs. The following programming tools work on a single CPU, but are **not** supported clusterwide:

- Permanent and temporary mailboxes
- Common event flags
- Logical names
- $CREPRC[2]
- Writable global sections

There are several ways you can modify your programs to achieve clusterwide comparable services. For information and suggestions on modifying single-node software to run in a VAXcluster environment, see Section 3.6, Single-Node Programming Tools Not Available Clusterwide.

---

[1] All known occurrences of an application **not** being able to run in a VAXcluster system are attributed to programming errors. A common programming error that will cause an application to **not** run in a VAXcluster environment is if an application does not know what to make of the "node$" prefixed to certain device specifications. For example, an application might depend on having VMS return disk names of the "DUA47:" format rather than "FRED$DUA47:".

[2] A specific group of Process Control and Process Information system services **are** supported clusterwide in VMS Version 5.2. See Section 3.4 for a discussion of the Clusterwide Process Services.

All three types of VAXcluster configurations (CI-based, LAVc, and MIVc) offer the capability for incremental growth, transparent data sharing, efficient resource sharing, and centralized system management. Using the combined advantages of VAXcluster system hardware and software components, you can modify a VMS-based application to optimize the processing capabilities of your installation's VAXcluster system. By modifying an application that can only run on one VAXcluster CPU at a time to a clusterwide application, you can realize several benefits:

- Increased application availability

  When multiple CPUs in a VAXcluster system provide the same software environment for an application, the application has increased availability for the users because the application can be run from any node.

- Higher throughput

  With multiple copies of the same program executing on multiple CPUs, throughput can be increased because multiple CPUs provide more CPU cycles for concurrent executions of a program.

- Faster completion of a single application

  If a single application is divisible into multiple units, each unit can execute separately in parallel; consequently, an application can execute faster by using multiple VAXcluster nodes.

- Improved utilization of VAXcluster CPU resources

  By using the VAXcluster features of the distributed batch facility, the distributed VMS lock manager, the distributed file system, the distributed job controller, and Clusterwide Process Services (see Section 3.4), the application designer can use the VAXcluster system as a single programming environment.

2.1 Two Types of VAXcluster Application Activity

VAXcluster system application activity can generally be divided into two categories:

- I/O-intensive activity
- CPU-intensive activity

The two categories map directly into the most abundant hardware resources on a VAXcluster system: data storage and multiple CPUs. The concepts of *I/O-intensive* or *CPU-intensive* applications represent a starting point for VAXcluster application design. To help in application design, the application designer and programmer should consider the application's I/O and CPU requirements.

Applications can often be categorized as being either I/O-intensive or CPU-intensive, when the application activity falls primarily in one category. Other applications have some parts that are I/O-intensive and others that are CPU-intensive.

With an I/O-intensive application, the greatest activity is disk I/Os generated by file access requests or updates to display devices.[3] The VAXcluster system enables multiple users from multiple VAXcluster CPUs, all running the same application, to transparently use the VAXcluster hardware and software components for extensive and flexible resource sharing.

With a CPU-intensive application, the application primarily uses CPU compute cycles. The programmer can execute a CPU-intensive application on one VAXcluster CPU, or take advantage of the unique hardware and software components of a VAXcluster system and distribute CPU-intensive application *tasks* to multiple VAXcluster CPUs. In this way, the application can be designed to take advantage of the VAXcluster system's multiple CPU resources by executing multiple instruction streams concurrently.

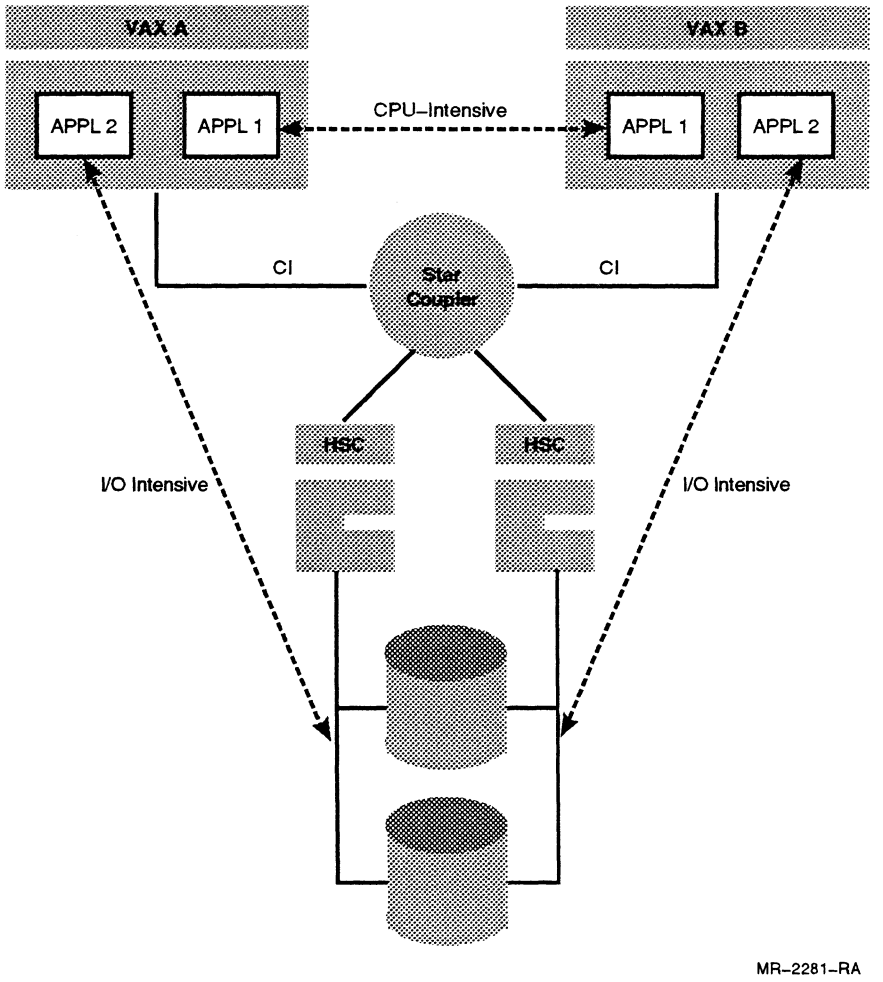
These two types of work or activity for an I/O-intensive or CPU-intensive application are represented in Figure 2–1. The vertical arrows show the direction of information flow in an I/O-intensive application, and the horizontal arrows show the direction of information flow in a CPU-intensive application.

Notes on Figure 2–1

The processing activities along the vertical lines are the disk I/Os requested by APPL2, an **I/O-intensive application** distributed by *replication* (see Section 2.1.1). The processing activities across the horizontal line are the *interprocess communications* required for distributing APPL1, a **CPU-intensive application**, by *decomposition*. (See Section 2.1.2.)

[3] In this manual, application I/O-intensiveness refers primarily to disk I/Os, not terminal I/Os. For information on application design considerations for terminal I/Os, refer to the *VMS Device Support Manual* and the *Guide to VMS Performance Management*.

Figure 2–1 Comparing the Direction of Information Flow for an I/O-Intensive Application Distributed by Replication and a CPU-Intensive Application Distributed by Decomposition



2.1.1 I/O-Intensive Application

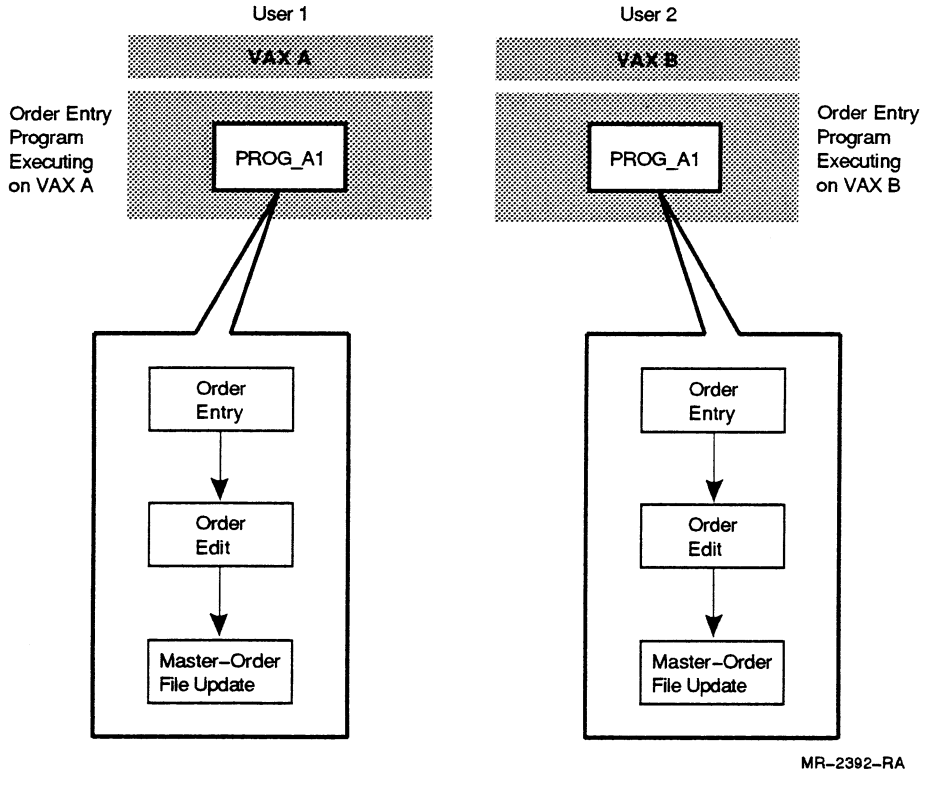
An I/O-intensive VAXcluster application is characterized by one or more users accessing clusterwide data resources. With multiple users running the same application, clusterwide file sharing is arbitrated by the VMS Record Management Services (VMS RMS) and the distributed VMS lock manager. Typically, the programmer designs the application using read and write operations to perform disk operations accessing the file resources. The application designer must work within the site-specific hardware configuration to optimize performance.

Figure 2–2 represents distributing an I/O-intensive application (PROG_A1) by replication on a VAXcluster system. PROG_A1 is a generic order entry application that performs the following file-related functions: order entry, order edit, and master-order file update. When PROG_A1 is executed by multiple users on different VAXcluster CPUs, PROG_A1 executing on VAXA is simultaneously requesting the same files and possibly the same records as the second copy of PROG_A1 executing on VAXB. As VAXA and VAXB simultaneously execute copies of PROG_A1, the distributed file system and distributed VMS lock manager work together to arbitrate *concurrent access* requests for the same data element.

By specifying file and record access modes from a high-level language, the programmer automatically uses VMS RMS to interface with VAXcluster software components. Transparent to the application PROG_A1, VMS RMS uses the distributed VMS lock manager and the distributed file system to synchronize multiple access requests for the same data elements. (See Section 3.2 and Section 6.2.2 for more information on using VMS RMS.)

I/O-intensive applications distributed by replication on a VAXcluster system enable users scattered across the VAXcluster system to have synchronized read or write access to the same disk files down to the record level. This functionality is **exactly** the same as on a standalone VMS system. However, in a VAXcluster system, VMS RMS and the distributed file system use the clusterwide lock database maintained by the distributed VMS lock manager as an arbitrator for processes concurrently accessing the same data.

Figure 2–2 I/O-Intensive Application (PROG_A1) Executing Multiple Copies on a VAXcluster System



When distributing an I/O-intensive application by replication on a VAXcluster system, the programmer realizes the following advantages:

- The VAXcluster system supports file sharing by multiple processes on multiple VAXcluster CPUs; consequently, having a greater number of CPUs running an application increases an application's availability.

Note: Assuming the VAXcluster hardware configuration remains unchanged, the critical factor is the total number of users executing an application. As the total number of users increases, there is the potential for an I/O bottleneck if all of the users are seeking access to the same devices.

- Executing multiple copies of an application on multiple CPUs takes advantage of the VAXcluster system's capability to synchronize clusterwide file access by using the distributed file system and the distributed VMS lock manager.

Distributing I/O activity can have the following disadvantages:

- Distributing I/O-intensive applications can cause increased traffic over the VAXcluster interconnects and slower I/O activity. For example, shifting I/O requests from a disk-serving node in a LAVc to the satellites increases Ethernet traffic and, potentially, could shift an I/O bottleneck from a disk drive to the Ethernet. (For more information on potential I/O bottlenecks in a VAXcluster system, see Section 7.2, Potential Bottlenecks for an I/O-Bound Application Environment.)
- Distributing I/O activity targeted to a single file requires that the file be shared and the access coordinated; this can slow down each I/O when compared to performing the I/O from a single process.

See Section 2.4 for further information on how to distribute an application by replication. In Chapter 4, the File Sharing and Client-Server Models discuss considerations for implementing an I/O-intensive application, and Section 7.2 discusses performance considerations for I/O-intensive applications.

Also, refer to the *Guide to VMS File Applications* for a discussion of I/O and locking performance considerations when programming for "VAXcluster Shared Access."

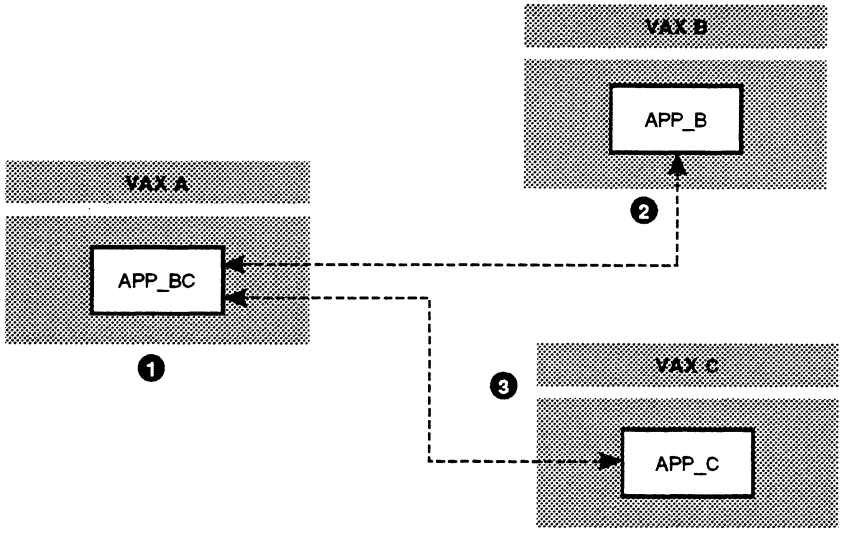
2.1.2 CPU-Intensive Application

A CPU-intensive application can be executed using a single CPU in a VAXcluster system. However, dividing the work of an application into one or more discrete functional units for execution on different VAXcluster nodes can effectively decrease execution time and increase throughput. A CPU-intensive application can usually be divided into functional units and distributed to multiple CPU resources for *synchronous* or *asynchronous* execution.

For optimal performance of CPU-intensive application distributed by decomposition, the application designer should take into account the site-specific hardware configuration.

Figure 2–3 represents a CPU-intensive application for producing an insurance amortization table based upon a statistical analysis of many client-supplied variables. The user executes APP_BC on VAXA to input all the required information to perform a statistical analysis of the client's insurability. Depending on the type of insurance requested, as specified by a rate entry key, different statistical computations must be performed. The work or statistical analysis of this application is distributed to either APP_B running on VAXB, or APP_C running on VAXC.

Figure 2–3 Distributing Work for a CPU-Intensive Application Executing on a VAXcluster System



❶ Rate Entry Program for Insurance Rates executing on VAXA as APP_BC.

❷ When Rate Entry key is 1 to 1000, input is verified by APP_B, and if successful, APP_B performs statistical analysis to produce an amortization table.

❸ When Rate Entry key is greater than 1000, input is verified by APP_C, and if successful, APP_C performs statistical analysis to produce an amortization table.

The distribution of CPU-intensive work illustrated in Figure 2–3 shows how the input program APP_BC, executing on VAXA, can become more responsive for multiple users on VAXA. The screen presentation and entry functions performed by APP_BC have been separated from the CPU-intensive functions of the application. By distributing the CPU-intensive functions to APP_B and APP_C on different VAX CPUs, APP_BC becomes more responsive to the users on VAXA, because this part of the application is only managing screen presentation for the user's input. Once the user's input is captured, the input can be sent to either APP_B on VAXB or APP_C on VAXC for statistical analysis and production of the client's amortization table. While APP_B or APP_C are executing, APP_BC can begin to process the next input item for a new client.

In general, you can realize the following advantages by distributing an application's work to multiple VAXcluster CPUs:

- The user's *response time* is improved because an application's CPU-intensive processing activities can be performed on another VAXcluster CPU as a dedicated CPU.
- The functions of an application can be distributed to the VAXcluster CPU best suited for the processing activity. Typically, the criterion for this choice is the power of a CPU.
- The application's throughput is increased because by distributing CPU-intensive work to multiple VAXcluster CPUs, more CPU cycles per unit time are applied to the CPU-intensive task.

Note: If a VAXcluster CPU fails, there is no automatic failover for any programs on the failed CPU. If *high availability* is a requirement of the application tasks distributed on multiple VAXcluster CPUs, the programmer must include code for failover in the event of a CPU failure. See Section 6.4.2, Recovery from Cluster State Transition Due to Node Failure, for more information.

Distributing CPU-intensive work may have the following disadvantages:

- Implementing a VAXcluster application distributed by decomposition may increase an application's complexity and the effort required to maintain that application.
- Distributing units of an application's work may increase the total CPU cycles required to complete that application's work.

See Section 2.5 for more information on how to design an application for distribution by decomposition. Chapters 3 and 5 of this manual present discussions of application design models and programming implementation techniques for distributing work to VAXcluster CPU resources. See Section 7.4 for considerations related to performance of CPU-intensive tasks.

2.2 Characteristics of an Application Suitable for a VAXcluster System

As a general rule, applications that run on a single VAX CPU can also run on a single CPU in a VAXcluster system without modification. I/O-intensive applications that run poorly on a single VAX CPU will also tend to run poorly on a VAXcluster system. (For a specific application, there may be an improvement in I/O performance by using VAX Volume Shadowing.) However, a CPU-intensive application that runs poorly on a single VAX CPU can, with modification, sometimes run better on a VAXcluster system.

VAXcluster software enables the application designer to plan for work distribution using the multiple CPU resources or concurrent access to the shared disk data of the VAXcluster system. The programmer can implement an application design based upon using either distribution of multiple program copies or multiple instruction streams.

- I/O-intensive applications are often designed with multiple copies of the application that run on multiple VAXcluster nodes, as shown in Figure 2–2. If a single user I/O-intensive application is executed on a VAXcluster system, the single user application may require modification to support multiple users. However, an I/O-intensive application designed for multiple users and not involving explicit interprocess communication can usually run on multiple CPUs in the VAXcluster system **without modification**. See Section 4.1, File Sharing Model, for further information.
- CPU-intensive applications are often designed as multiple instruction streams, with execution for each instruction stream distributed to multiple VAXcluster CPUs. A CPU-intensive application that distributes work to multiple CPUs is a complex application design which requires explicit interprocess communications to synchronize the work. A distributed CPU-intensive application requires **some modification** to take advantage of the communication and synchronization mechanisms available on a VAXcluster system. Refer to Section 4.2 and Section 4.3 for further information.

Applications that can benefit from running on a VAXcluster system generally have the following requirements:

- High availability

  Some applications are critical, requiring a high percentage of system availability. The VAXcluster system lets applications access data from multiple nodes and can be designed to provide failover access to data if a CPU or HSC system in the cluster fails. For more information on programming for *exception conditions*, see Section 6.4.

- Ease of system growth

  As the needs of your application grow, you may need to add devices such as disks, tapes, and CPUs. Flexibility of your VAXcluster system enables the system to grow to accommodate these devices.

- High performance

  Some applications use significant amounts of CPU time. When designing these applications, the CPU-intensive sections of code can be divided into separate tasks. When implementing the application, the CPU-intensive tasks can be distributed to multiple CPUs to improve an application's execution time and increase the VAXcluster system's overall throughput.

- A shared file system

  The application requires access to shared data. VMS software components, using the distributed file system and the distributed VMS lock manager, provide the capability for clusterwide arbitration for shared file access. See Section 4.1, File Sharing Model, for more information on application design for a shared file system.

The VAXcluster system provides you with the following resources for clusterwide programming development: a common system environment for accessing the run-time libraries, the VMS linker, and shareable *images*. Typical examples of application environments that may be developed to run on a VAXcluster system are:

- Transaction processing
- Batch processing
- Time sharing

Transaction Processing

When a *transaction processing* application is implemented on a VAXcluster system, the front-end processing activities can take advantage of a VAXcluster hardware redundancy for increased application availability, while back-end processing activities take advantage of the VAXcluster system's multiple CPUs to increase an application's throughput. For more information on Digital's Online Transaction Processing (OLTP) products, see Section 5.6.1, DECintact, and Section 5.6.2, VAX ACMS.

Batch Processing

A typical batch processing application is non-interactive, long running, and is submitted and controlled from a Digital Command Language (DCL) job stream. The advantage of executing a batch application on a VAXcluster system is the availability of multiple batch queues. Consequently, there can be parallel execution for independent tasks within the batch application and, in the event of a CPU failure on an executing batch queue, you can use DCL to provide an automatic failover to an operating batch queue on another VAXcluster CPU. For more information on using DCL to restart a batch job, refer to Section 3.3.

Time Sharing

A *timesharing* application takes advantage of the cluster's multiple CPUs because each user is independent of other users; this allows a high degree of concurrency. Communication and synchronization between the independent user processes are achieved through VMS and the distributed file system's arbitration for shared disks and files. A timesharing application can also take advantage of a VAXcluster system's hardware availability. If one CPU in a VAXcluster system fails, users from that CPU can generally continue working on another VAXcluster CPU. However, failover from a failed CPU to an alternate VAXcluster CPU is not automatic; each timesharing user must log in again to continue working.

2.3 Applications That May Not Be Suitable for a VAXcluster System

Applications that may **not** behave as expected in a cluster environment are those that require:

- Nonstop processing

  Such an application is one that requires realtime access to nonreproducible data. It cannot tolerate pauses such as those that occur when a system joins or leaves the cluster.

  In some cases you may be able to effectively use a cluster to process data collected in real time by using either of the following:

  — A separate, fully redundant system connected to a VAXcluster system using the DECnet network

  Using DECnet, data can be transferred from the dedicated computer system to the VAXcluster system, where data processing and database management take place. This type of configuration protects the application from cluster interrupts, but not from the failure of the dedicated node or its own resources.

  An example of this type of configuration is a large VAXcluster system in Tucson, Arizona that is used to store and process astronomical data. The astronomical observatory requires realtime data collection from telescopes, which must not be interrupted even for the short period of time required by cluster state transitions. This computing problem is solved by dedicating a special-purpose computer to collecting the data from the telescopes. The data is stored on media and transferred to the VAXcluster system on a regular basis. Scientists access the astronomical data using applications running on the VAXcluster system. The VAXcluster system also runs software for processing the astronomical data, formulating and testing results, and for administrative purposes. Thus, the high fault-tolerant requirements of this application are served by a dedicated computer system. The remaining requirements of the computing facility are met by a VAXcluster system, providing all the benefits of the cluster design to the system users.

  — Devices with adequate data buffering

  For example, if you are collecting data from a satellite orbiting the earth, and the data downline loaded to a VAXcluster system for processing, you can ensure nonstop processing by collecting the telemetry data in a device with adequate data buffering.

- Complete fault tolerance

  A VAXcluster system provides high availability of disk and CPU resources, but does not provide 100 percent fault tolerance. Also, there is no automatic failover of processes from one node to another.

  However, in the event of a VAXcluster CPU failure, you can write an application to be restarted on another CPU. See Section 6.4.2, Recovery from Cluster State Transition Due to Node Failure, for more information.

- Low-overhead communications between processes on separate VAX CPUs

  CPUs in a VAXcluster system do not share common memory. Therefore, they require more overhead to communicate and synchronize activity than CPUs in a tightly coupled system.

2.4 Is an Application a Candidate for Distribution by Replication Across VAXcluster CPUs?

Typically, an I/O-intensive application is a good candidate for replication in a VAXcluster system. There are six steps to consider when determining whether to design a distributed application for replication on multiple CPU resources:

1. Determine the system environment.
2. Evaluate CPU resources.
3. Evaluate I/O resources.
4. Analyze the user's demand for the application.
5. Establish the goals of the application.
6. Arrive at a decision.

Step 1 — Determine the System Environment

An application is easiest to replicate in a common-environment VAXcluster system because in such an environment there are no restrictions on its availability to users on any VAXcluster CPU. However, in a large VAXcluster system containing both multiple and common-environments, you can replicate an application on only the common-environment segment.

Step 2 — Evaluate CPU Resources

To avoid degradation in performance, you should evaluate the existing load capacity of the VAXcluster CPUs where you may want to execute the image. You can then choose the VAXcluster CPUs that will execute the replicated image most effectively.

Step 3 — Evaluate I/O Resources

The load on the I/O system is an important consideration and must be evaluated. The VAXcluster I/O system contains: CI adapters, Ethernet adapters, and disk drives. To evaluate the load on the I/O system, consider the relative I/O throughput capacity for each of these types of hardware components in your I/O system.

The relative I/O throughput capacity for these components is presented in Table 2–1, Table 2–2, and Table 2–3. For individual performance values for these Digital hardware components, refer to the latest supplement of the *VAX Systems and Options Catalog* and the *Guidelines for VAXcluster System Configurations*.

Table 2–1 CI Adapter Throughput Capacity (Fastest to Slowest)

Type of CI Adapter
CIBCA-B
CIBCA-A
CIBCI
CI780
CI750

Table 2–2 Ethernet Adapter Throughput Capacity (Fastest to Slowest)

Type of Ethernet Adapter
DEBNA, DEBNI
DEQNA, DELQA
DELUA
DEUNA

Table 2–3 Disk Type I/O Rates (Fastest to Slowest)

Type of Disk
ESE20
RA90, RA82, RA70, RF30, RF71, RZ22, RZ23
RA81
RA60, RD52, RD53, RD54
RD31, RD32

It is important to determine if replication will overload a component of the I/O system that is already close to being a bottleneck. For example, if you have replicated an application and multiple images are requesting I/Os through the same Ethernet adapter, you may encounter a bottleneck at that Ethernet adapter. Also, disk access speed is not the same for all VAXcluster CPUs; VAXcluster CPUs accessing a disk over the Ethernet take longer than VAXcluster CPUs that do not need to send data over the Ethernet to access a disk.

For more information on performance considerations for an *I/O-bound* application, see Section 7.2.

Step 4 — Analyze the User's Demand for the Application

Once an application is replicated, will the user's demand for that image be fairly equal for each instance of the application that is being executed on the VAXcluster system? For example, if there are 10 users for an application, are each of the users running the image about the same amount of time? Or, are one or two of the users running the application a disproportionate amount of time? If multiple copies of this application are replicated on multiple VAXcluster CPUs, then the number of users assigned to each available VAXcluster CPU should be balanced according to the amount of work the users perform on each CPU.

When the demand for the application is relatively constant among all of the users, it is important that users follow a method for choosing from which VAXcluster CPU to execute the image. The users may use a cluster alias with the LAT software to assign an available CPU for logging on to run a copy of the image. Or, each user of the application may be instructed to only run the image from a specific VAXcluster CPU.

Step 5 — Establish the Goals of the Application

There are two objectives for replicating an application on a VAXcluster system:

- Increasing application throughput
- Increasing application availability

When the number of users remain constant, you can reduce user response time and increase the application's throughput (assuming there are not I/O system bottlenecks and the workload is balanced equitably).

Also, when the user population for an application remains constant, running multiple copies of an image on multiple VAXcluster CPUs enables you to increase the application's availability because if a VAXcluster CPU fails, the application remains available to users on other VAXcluster CPUs.

Step 6 — Arrive at a Decision

Based on your evaluation of Steps 1 to 5, consider replicating an application on multiple VAXcluster CPUs if:

- There is a homogeneous (common-environment) segment of the VAXcluster system.
- CPUs in the VAXcluster system are not at maximum load capacity.
- The I/O system in the VAXcluster system is not at maximum load capacity.
- There are multiple demands for the application, and the demand for the image is fairly even among users.
- The user demand can be fairly distributed to the VAXcluster CPUs where the image is replicated.
- For a constant user population, a replicated application can decrease the user response time and increase system throughput for the application.

For more information on how to design an application for replication, see Chapter 3, Programming Tools for VAXcluster Application Development, and Chapter 4, Application Design Models for VAXcluster Software.

2.5 Is an Application a Candidate for Distribution by Decomposition Across VAXcluster CPUs?

Typically, a CPU-intensive application is a good candidate for distribution by decomposition in a VAXcluster system. However, there are five steps you should consider when determining whether your application should be designed to distribute units of work to multiple CPU resources:

1. Determine suitability.
2. Analyze application.
3. Establish goals.
4. Determine divisibility.
5. Arrive at a decision.

Step 1 — Determine Suitability

Make sure the application type does **not** fall into one of the categories described in Section 2.3, Applications That May Not Be Suitable for a VAXcluster System.

Step 2 — Analyze Application

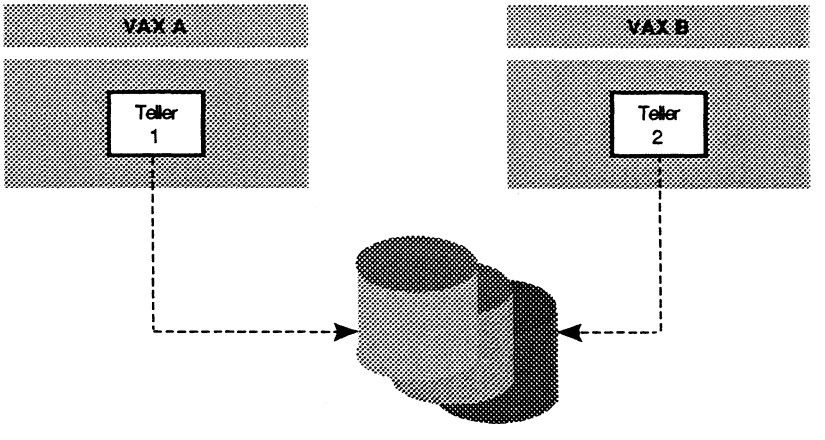
Can the application designer state the functional requirements of the application design for the application programmer? Usually, this can only occur after the system designer has made a thorough analysis of the application constraints in the context of the user's environment. In a business environment, the analysis of the user's needs is very application dependent.

Sample Application Analysis

For example, consider an analysis of the business problem for the following banking application:

a. Last National Bank wants to design an application to process teller transactions and update all the associated files in their customer database. Figure 2–4 represents what the teller application must do. In the course of servicing bank customers, each teller will use the programmed interface to process the customers banking transaction and the application will update all the associated database records.

Figure 2–4 Sample Banking Application on a VAXcluster System



Transaction = Recalculate Customer Balance for Master Account and Update Files X, Y, Z

Teller Processing Steps:

1. Input Transaction
2. Computer Must Successfully Update Files X,Y,Z for Each Transaction
3. Next Transaction

MR–2394–RA

b. There are several possible ways of designing an application to accomplish the required processing steps:

— Write all daily transactions to a file for batch processing at the end of the business day.

— Write all daily transactions to a file for batch update processing every *delta* minutes.

— Perform an online update using callable programs (shareable images) to update files x,y,z.

— Distribute the client (teller) input to one or more servers performing the update function for files x,y,z and allow the teller to begin processing the next transaction.

c. To decide which functional approach is preferable, the application designer must determine the constraints of the business environment for which this application is being specifically designed. In the context of the teller application for Last National Bank, the application designer may want to consider:

   — How important is it for the data to be current?

   — How many users will access the data?

   — How many transactions will occur within a time period?

   — What are the tasks for a transaction?

   — What is the relationship between the tasks?

   — What delays are acceptable?

   — What are the CPU resources?

Step 3 — Establish Goals

After establishing the environmental constraints, the application designer should establish application goals relating to speed, availability, throughput, and error tolerance. The established goals will help to determine the approach the designer should use. See Table 2–4 to help make this determination.

Table 2–4 Goals for Distributing an Application Across VAXcluster CPUs

Application Goal	Approach	Refer To
Increased Availability	Divide an application into *front-end* and *back-end* tasks to increase the availability of the front-end application to the users.	Section 4.2, Section 5.1
Speed	Use multiple CPU resources to obtain greater aggregate performance for a single application.	Section 4.2, Section 4.3, Section 5.2
Increased Throughput	Distribute the work over multiple CPU resources to load balance and increase the throughput.	Section 4.1, Section 4.2, Section 4.3, Section 5.3
Error Tolerant Processing	Distribute the work over multiple CPUs to approximate error tolerant processing by providing a failover mechanism for VAXcluster node failures.	Section 4.2, Section 4.3, Chapter 8

Step 4 — Determine Divisibility

First, establish if an application can be divided into discrete functional units:

- Can the application be divided into functional units?
- Does each functional unit accomplish a single task?

Proceed **only** when you can answer the two preceding questions with "yes." Then determine the following attributes and interrelationship of the application's functional units:

— Are there data dependencies between functional units?

When functional units require simultaneous access to the same data, there is a *data dependency*. If the simultaneous access is for updating, the data dependent functional units cannot execute in parallel.

— Can the functional units execute in parallel?

If there are no data dependencies between functional units, the functional units can be designed as multiple instruction streams executing in parallel.

— Is disk I/O a significant bottleneck?

If the functional units require disk access, and the disk I/O has been identified as a significant bottleneck, it is important that disk I/Os be minimal. Relative to a CPU compute cycle, a disk I/O is very time-consuming. For more information on identifying disk I/O bottlenecks, see Section 7.1, Using VMS Utilities to Monitor the Cluster and Identify Bottlenecks.

— How much communication is required between the functional units?

To synchronize the execution of functional units, interprocess communications are required to define the interrelationship of the functional units. When the interrelationships of the functional components are complex, the communication overhead for distributing the application increases.

— Are the distributed functional units performing an optimal amount of work?

Each functional unit must perform a sufficient amount of work to offset any communication overhead incurred by the functional unit.

— What is the ratio of distributed work to the amount of synchronization required?

As long as the amount of work performed by the functional unit is significantly larger than the communication overhead required to synchronize the functional unit, the application is a good candidate for distributed application design.

Step 5 — Arrive at a Decision

After a thorough analysis of an application using Steps 1 to 4, consider an application designed to distribute by decomposition if:

- The gain in application performance justifies the additional effort and expense of implementing a distributed application.
- The application goal is to increase speed (decreased wall-clock time) or to increase system throughput by running a distributed application at the lowest priority available on the system at a time when the cluster is normally idle.
- The application can be divided into functional units.
- Each functional unit can be designed for one task.
- There is a low data dependency between functional units; that is, the tasks are relatively independent of each other.
- There are minimal disk I/Os required for each functional unit because many disk I/Os will tend to diminish the performance gained by distributing the application.
- Each functional unit is used frequently.
- The amount of work performed by each functional unit is large compared to the overhead associated with interprocess communications.

For more information on how to design an application for distribution by decomposition, refer to Chapter 3, Programming Tools for VAXcluster Application Development, and Chapter 4, Application Design Models for VAXcluster Software.

3 Programming Tools for VAXcluster Application Development

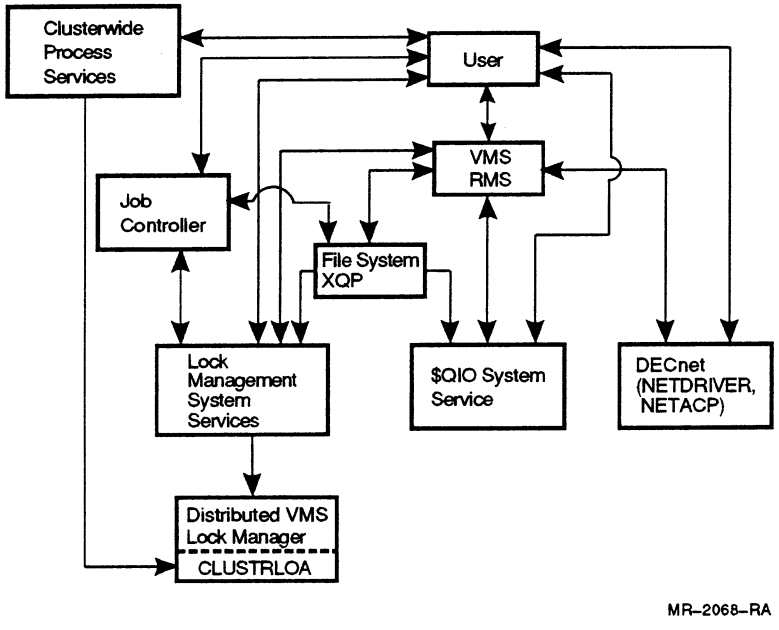
When developing applications for use in a VAXcluster environment, you can use several programming tools supported by VMS software and VMS layered products. VMS supports the following programming tools:[1]

- The VMS Lock Manager is distributed to all VAXcluster CPUs to coordinate clusterwide resource access. (See Section 3.1.)
- VMS Record Management Services (VMS RMS) interfaces with the distributed VMS lock manager and the distributed file system. (See Section 3.2.)
- The VMS Batch Facility interfaces with the distributed job controller. (See Section 3.3.)
- Clusterwide Process Services act on any process executing on the VAXcluster system. (See Section 3.4.)
- DECnet-VAX software creates communication links between VAXcluster CPUs. (See Section 3.5.)

Figure 3–1 shows the relationships between these VAXcluster programming tools.

[1] See Section 5.6 for more information on programming with VMS layered products.

Figure 3–1 Functional Relationship of VAXcluster System Programming Tools

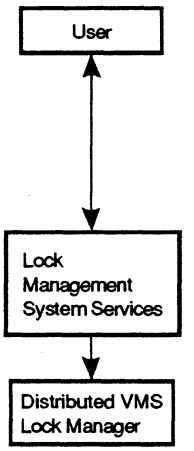


3.1 VMS Lock Manager

The VMS lock manager, illustrated in Figure 3–2, is the basic *process synchronization* mechanism for the VAXcluster system. The lock manager enables clusterwide synchronization by granting locks, when appropriate, to requesting processes. By using lock management system services, you can request, release, or query locks on a resource.

Each resource in a VAXcluster system is represented by a unique *resource name*. When a process' lock request is granted for a lock on a resource name, this resource name is placed in the distributed lock manager's *clusterwide lock database*. The VMS lock manager does not actually allocate or control the resource, and the resource name need not represent an actual physical resource. The VMS lock manager mediates the access of a resource by determining the compatibility of the requested locks with those existing on the resource name in the clusterwide lock database. If the *lock modes* are compatible, the lock is granted, and the requesting process can share the actual resource. To gain access to any resource, all processes executing on any VAXcluster CPU use lock management system services, directly or indirectly, to interface with the VMS lock manager.

Figure 3–2 VAXcluster Programming Tool: VMS Lock Manager



All resource names are known clusterwide; therefore, to determine the location of information for all existing locks granted for a resource name, the lock manager must reference the locking information for that resource name. The VMS lock manager accomplishes this by using a directory look-up to establish which VAXcluster CPU is the *resource manager* for the resource name. Typically, the lock manager assigns a VAXcluster CPU as resource manager for the first request for a lock on a resource name. Subsequently, the second process requesting a lock for the same resource name is pointed to the VAXcluster CPU that is the resource manager for that resource name. The resource manager CPU for the resource name maintains the clusterwide lock database for all locks that have been granted for that resource name.

Note: You should not assume that you can predict where a lock manager resource tree will be mastered. If you do determine where a tree is mastered (perhaps by using $GETLKI), mastership may move to another node at any time.

3.1.1 Lock Management System Services

Lock management system services can be used directly or indirectly from an application. You can use the lock management system services directly by calling a lock management system service routine from your application.

The lock management system services are used indirectly when you utilize components of VMS which themselves use the lock manager to synchronize their activities, such as the following:

- Using VMS RMS to access files, either by using the input/output services provided by a high-level programming language, or by explicitly calling VMS RMS system service routines
- Using the distributed file system to create and delete files, or to otherwise modify the structure of mass storage volumes
- Using the ALLOCATE command (or the $ALLOC system service) to allocate a device (allocation is implemented by using a lock representing the device)

The VMS lock management system services listed in Table 3–1 interface with the VMS lock manager.

Table 3–1 Lock Management System Services

Service Name	Meaning	Function
$ENQ[1]	Enqueue lock request	Requests a lock on a resource or converts a resource lock mode.[2]
$ENQW[1]	Enqueue lock request and wait	Requests a lock or lock mode conversion on a resource, and waits for the lock to be granted or converted.
$GETLKI[1]	Get lock information	Requests information about the lock database. $GETLKI does not wait for the information to be returned.
$GETLKIW[1]	Get lock information and wait	Requests information about the lock database. $GETLKIW waits for the information to be returned.
$DEQ[1]	Release lock request (dequeue)	Performs one of the following functions: • Unlocks a process' granted locks • Gives up ownership of the resource • Releases the sublock of a resource • Cancels a queued lock request that has not been granted

[1]Refer to the *VMS System Services Reference Manual* for more information.

[2]For more details about lock modes, see Section 3.1.2 and Section 3.1.3.

When using lock management system services, you must specify a lock mode as part of the system service argument. The VMS lock manager uses the lock mode argument for a lock request to determine the compatibility or incompatibility of clusterwide locking requests for the same resource name. The VMS lock manager uses the matrix illustrated in Table 3–3 to determine lock mode compatibility. Incompatible lock requests are queued and granted when the conflicting lock is released or converted. In addition, 16 bytes of programmer-supplied information can be associated with each lock granted to a resource name.

To request a lock on a resource, a process must use the lock management system services, supplying the following information:

- The resource name as defined by the requesting process

  The VMS lock manager uses the resource name to look for other locks granted for that name. (Resource names are implicitly qualified by the UIC group number of the process from which they are requested.)

- The lock mode

  The lock mode indicates whether the process wants to share the resource with other processes, and if so, how.

- The address of the lock status block

  The *lock status block* for the resource name indicates the queue in which the lock is placed:

  - GRANTED — The lock request has been granted.
  - WAITING — The lock request is waiting to be granted.
  - CONVERSION — The lock request has been granted at one mode and is waiting to be granted a higher lock mode.

  When the lock has been granted, the lock status block contains the lock identification and the status of the lock. VMS uses the lock identification to refer to a lock request after it is queued.

The lock management system services can be used to perform the following functions:

- Coordinate clusterwide access to shared resources

  Typically, a programmer uses a high-level language as an interface to VMS RMS, and VMS RMS calls the lock management system services to interface with the VMS lock manager and the file system. (See Section 3.1.2, Section 3.1.3, and Section 3.1.4.)

- Store and pass information between all cluster processes accessing the same resource name

  All locks granted by the lock manager have a data structure, the *lock value block*, that **can** contain 16 bytes of programmer-supplied information. (See Section 3.1.5.)

- Synchronize interprocess events across the entire cluster

  By specifying an *asynchronous system trap* (AST) or a *blocking asynchronous system trap* (blocking AST) with the lock request, the programmer can synchronize interprocess events. (See Section 3.1.6.)

3.1.2 Lock Modes

A process can access a resource by locking it in different modes. Lock modes allow *cooperating processes* to:

- Share access with other processes
- Prevent access by other processes
- Coordinate access with respect to time

Table 3–2 provides a summary of the lock modes. The table presents the lock modes in an ascending order from the less restrictive to the most restrictive mode. For more detailed information, see the *VMS System Services Reference Manual*.

Table 3–2 Lock Modes

Lock Mode	Mode Name	Lock Description
NULL MODE (NL)	LCK$K_NLMODE	This mode sets a place holder in the name space and indicates future interest in the resource. Later, the lock can be converted to a higher lock mode more quickly than if the lock were initiated in that mode.
CONCURRENT READ (CR)	LCK$K_CRMODE	This mode allows one process to read data from a resource in an unprotected manner while other processes can modify the data. This mode is typically used when sharing processes are only reading data, or when additional locking is being performed by other processes at a finer granularity using sublocks.
CONCURRENT WRITE (CW)	LCK$K_CWMODE	This mode allows one process to write data in an unprotected manner while other processes can simultaneously write data to the same resource. This mode is typically used when additional locking is being performed at a finer granularity using sublocks.
PROTECTED READ (PR)	LCK$K_PRMODE	This mode allows processes to share read access to a resource, but not to write to the resource at the same time.

Table 3–2 (Cont.) Lock Modes

Lock Mode	Mode Name	Lock Description
PROTECTED WRITE (PW)	LCK$K_PWMODE	This mode allows a process write access to the resource, and allows other processes to read from but not write to the resource at the same time. The other processes must have concurrent read access.
EXCLUSIVE (EX)	LCK$K_EXMODE	This mode allows write access to the resource and prevents other processes from reading from or writing to the resource. This lock prevents any other cooperating process from taking ownership until you release the resource.

If a process places a highly restrictive lock on a resource name, that process can prevent other processes from accessing the resource name and can slow the application's performance. For example, if a process places an EXCLUSIVE (EX) lock on a resource name, it blocks all other cooperating processes from gaining access to that resource name.

Note: Even though the VMS lock manager grants an EX lock to a specific resource name, the VMS lock manager does not prevent uncoordinated access to the underlying resource by other processes. The VMS lock manager is like a traffic light. A red traffic light does not physically stop your car. You stop your car at a red traffic light because that is the convention. Similarly, you do not access a resource, represented by a resource name, without obtaining the proper lock because that is the convention. In either situation, ignoring the conventions can cause data corruption.

Table 3–3 shows the compatibility between all lock modes. For example, as the table shows, a PROTECTED READ lock mode can be used with a NULL, CONCURRENT READ, or another PROTECTED READ. However, it is not compatible with a CONCURRENT WRITE, PROTECTED WRITE, or EXCLUSIVE lock mode.

Table 3–3 Lock Mode Compatibility

Mode	NL	CR	CW	PR	PW	EX
NL	Yes	Yes	Yes	Yes	Yes	Yes
CR	Yes	Yes	Yes	Yes	Yes	No
CW	Yes	Yes	Yes	No	No	No
PR	Yes	Yes	No	Yes	No	No
PW	Yes	Yes	No	No	No	No
EX	Yes	No	No	No	No	No

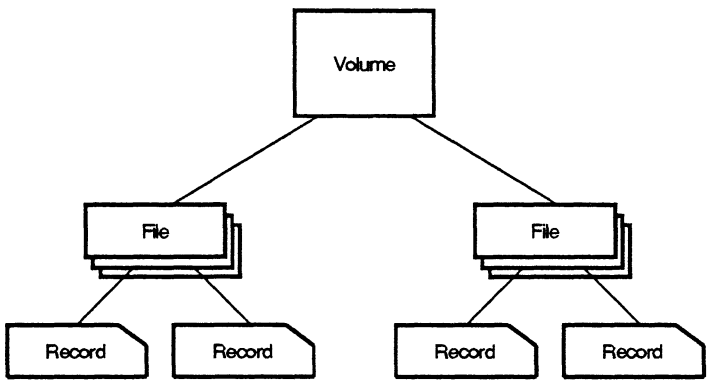
3.1.3 Locking Levels

In conjunction with lock modes, a process can share data resources with other processes by locking data at different levels. Many applications find it useful to describe their components in a hierarchical fashion. For example, a database can be considered as a hierarchy of:

- The entire database
- Files within the database
- Records within the files

The name space in the clusterwide lock database for resource names is tree structured to allow locks at different levels. The ability to have different lock levels on a resource name is called *resource granularity*. While the use of resource granularity is optional, the concept of granularity lets you have greater control over locking when accessing a resource, and is useful among cooperating processes that require complex lock structures. An example of this tree structure is shown in Figure 3–3.

Figure 3–3 Resource Granularity Locking



The degree of granularity is defined by the level of the resource being locked. For example:

- Coarse granularity locks an entire volume of data.
- Fine granularity locks files within the volume.
- Finer granularity locks records within files.

When processes have locks on a database at different levels, the lock at the highest level is the *parent lock*. Other locks at lower levels are *sublocks*. When using the lock management system services directly, you choose the scheme for the granularity. For example:

- Process A has an EXCLUSIVE (EX) lock on a parent, which represents a file.
- Process A also has a CONCURRENT READ (CR) lock on a sublock, which represents a specific record in the file.
- Another process (process B) wants a CONCURRENT WRITE (CW) lock on the sublock for the same record in the file.

The lock manager may or may not grant access to the sublock depending on the scheme used by process B. If process B first requests a CR on the parent (file), its lock request will not be granted, since CR is not compatible with EX. Process B cannot get a lock on the sublock (specific record in the file), since it does not hold a lock on the parent.

If instead, process B first requests a NULL (NL) lock on the parent and then the CW lock on the specific record in the file, both locks will be granted. (The NL lock on the parent is only granted if there are no other lock requests for the parent lock in the waiting or conversion queues.)

Note: When using a parent lock with sublocks, do not expect an EX lock on a parent to automatically exclude other processes from accessing any of the parent's sublocks. The use of a parent lock with sublocks requires cooperating processes; any corruption of this convention can be disastrous.

Resource management of sublocks occurs on the same node as the parent lock. To get a sublock, a process must first be granted a lock on the parent. As long as a lock is held on the parent, the resource manager for all related sublocks is known. Therefore, the VMS lock manager does not need to perform a directory look-up to determine which VAXcluster CPU is the resource manager for each sublock. In a VAXcluster system, this reduces the amount of time required to acquire a sublock.

3.1.4 Lock Queues

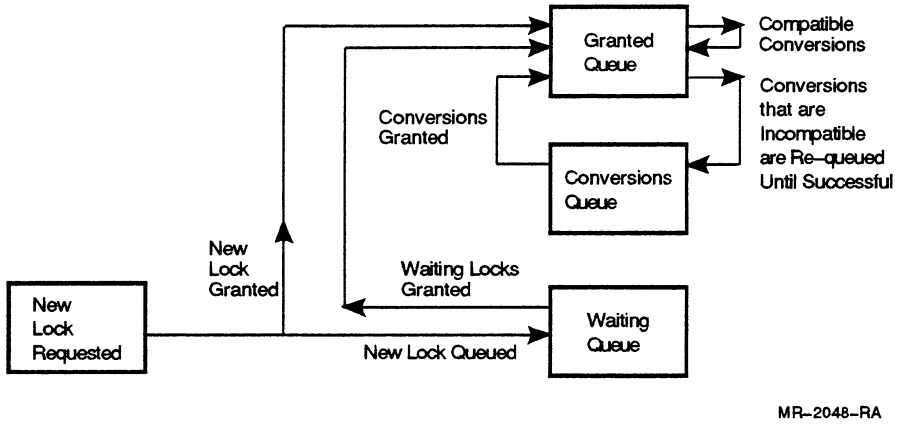
When a new lock request is made, the VMS lock manager determines whether the resource name is known or not known. If the resource name is **not known** in the clusterwide lock database, the lock manager creates the resource name data structures, grants the lock request, and updates the clusterwide lock database.

If the resource name is **known**, then the lock manager references the clusterwide lock database by a directory lookup. Before the lock request is granted, the lock manager determines if other lock requests for the same resource name are in either the clusterwide waiting queue or the conversion queue (see Figure 3–4). If no other lock requests for the same resource name are in the waiting or conversion queues, the locking request is:

- Granted, if compatible with existing granted locks
- Placed in the waiting queue, if incompatible with existing locks

However, if other lock requests for the same resource name are in either the waiting or conversion queues, then the locking request is placed at the end of the waiting queue.

Figure 3–4 Lock Queues of the VMS Lock Manager



Preventing Deadlocks

Deadlocks occur because of errors in designing locking schemes. For example, a *deadlock* can occur when the holder of the lock that you are waiting for cannot proceed because it is waiting for a lock that is currently granted to your program. In this case, A is waiting for B, B is waiting for C, and C is waiting for A. The distributed VMS lock manager detects these deadlock situations. Simply waiting for a locking request to be granted does not constitute a deadlock.

If you use VMS RMS to access files and the records therein, then VMS RMS automatically uses the VMS lock manager to handle deadlock situations among processes. However, if you are explicitly calling lock management system services, then, by properly coding your locking scheme, you can avoid deadlocks.

Deadlocks can occur in at least the following situations:

- When multiple lock requests are waiting for each other in a circular fashion, as illustrated in Table 3–4.

Table 3–4 Multiple Lock Requests Creating a Deadlock

Step	Action of Process 1	Action of Process 2	Compatible with Existing Lock?
1	Takes an EXCLUSIVE lock on Resource A	Takes a PROTECTED WRITE lock on Resource B	Yes
2	Requests a NULL lock on Resource B		Yes
3		Requests a PROTECTED WRITE lock on Resource A	No
4	Requests a lock conversion on Resource B from NULL to EXCLUSIVE		No

Note on Table 3–4

Result: In Table 3–4, Process 2 cannot get its PW lock on Resource A until Process 1 gets its EX lock on Resource B and gives up Resource A. However, Process 1 cannot get its EX lock on Resource B until Process 2 gets its lock on Resource A and gives up Resource B. This causes a deadlock.

These conflicting requests can cause processes to wait indefinitely for resources that may never be made available.

When the lock manager detects that a deadlock exists, it arbitrarily chooses a *victim* process. If the victim has requested a new lock, the request is denied and the deadlock is broken. If the victim has requested a lock conversion, the request is denied and the process retains its original lock.

In either case, the SS$_DEADLOCK status is returned in the victim's lock status block and any locks that the victim already holds are not revoked. You might consider including code that instructs the application to **do** something if the status returned in the lock status block is SS$_DEADLOCK.

- When you want to convert a lock from one mode to another and **forget** to set the LCK$M_CVT flag. For example:

  a. Process 1 has a CONCURRENT READ lock on Resource A.

  b. Process 1 wants to convert the lock to EXCLUSIVE.

If you do not set the LCK$M_CVT flag first, the lock manager perceives the lock conversion request as a new lock request. Because EXCLUSIVE mode is not compatible with CONCURRENT READ mode, the lock manager does not grant the lock. The program cannot continue until the CONCURRENT READ lock is released. However, the CONCURRENT READ lock cannot be released unless the program continues.

3.1.5 Lock Value Block

When the VMS lock manager grants a requested lock, a lock value block is associated with the lock status block; however, programmer use of the lock value block is **optional** and requires care.

You can use an $ENQ or $ENQW request to retrieve the 16 bytes of information contained in the lock value block and, when releasing or converting a lock, you can specify a new value for the lock value block. Consequently, as the ownership of a resource name changes, you can use the information in the lock value block as a source of information about the most recent state of the lock. By dynamically updating the contents of the lock value block for each use of a specific lock, you can potentially use the 16-byte lock value block as a clusterwide memory area.

Note: Due to the volatility of the lock value block during VAXcluster state transitions, the programmer is advised to observe special precautions when using the lock value block to store information.

A VAXcluster state transition occurs each time a VAXcluster CPU leaves or joins the cluster. In the event a CPU leaves the cluster, all the locks managed by the exiting node are distributed to the remaining VAXcluster resource managers. In this type of state transition, the lock value block is marked invalid when:

- Any process holding a PW or EX lock is abnormally terminated.
- The existing locks on a resource name are NL or CR.

For more information on using the lock value block, see *Introduction to VMS System Services* and Section 6.3.2.3 in this manual.

3.1.6 Using ASTs and Blocking ASTs for Synchronization of Interprocess Events

When you use the $ENQ or $ENQW system service, you can specify the delivery of an asynchronous system trap (AST) or blocking asynchronous system trap (blocking AST) with the locking request. When the AST or blocking AST is delivered by the VMS lock manager in response to an event external to the program, a prearranged address points to an AST or blocking AST service routine that then executes.

Based upon the events triggering the delivery of an AST or blocking AST, the programmer can code the associated AST or blocking AST service routines to respond to:

- The granting of a requested lock for the specified resource name
- A request for an incompatible lock for the specified resource name

ASTs

When a locking request on a resource name specifies an AST, and the locking request is granted, then the AST is delivered, interrupting execution of the requesting process, and executing the AST service routine.

Note: The granting of a lock is not the only mechanism of the lock manager to trigger the delivery of a completion AST. If a lock request is aborted because of a deadlock or if a $DEQ is executed for the lock before it is granted, the request is considered complete. If an AST has been specified, it is delivered. Therefore, always check the status in the lock status block every time the lock manager notifies you of completion.

Blocking ASTs

When a resource name has acquired a lock and a blocking AST has been specified for that granted lock, the blocking AST routine is invoked when any new locking request for the same resource name is incompatible with the lock currently granted for that resource name. (See Table 3–3 for a representation of Lock Mode Compatibility.) The process requesting the new lock for the resource name communicates to the process holding the incompatible lock that the lock holder is in the way.

Like the execution of an AST service routine, execution of a blocking AST service routine interrupts the process that has specified the blocking AST. When using blocking ASTs, remember the following:

- A program does not automatically know that it is blocking another locking request. It must declare a blocking AST with a lock request for a resource name to ask for notification.
- A requesting program has no guarantee that the current holder will release the incompatible lock.

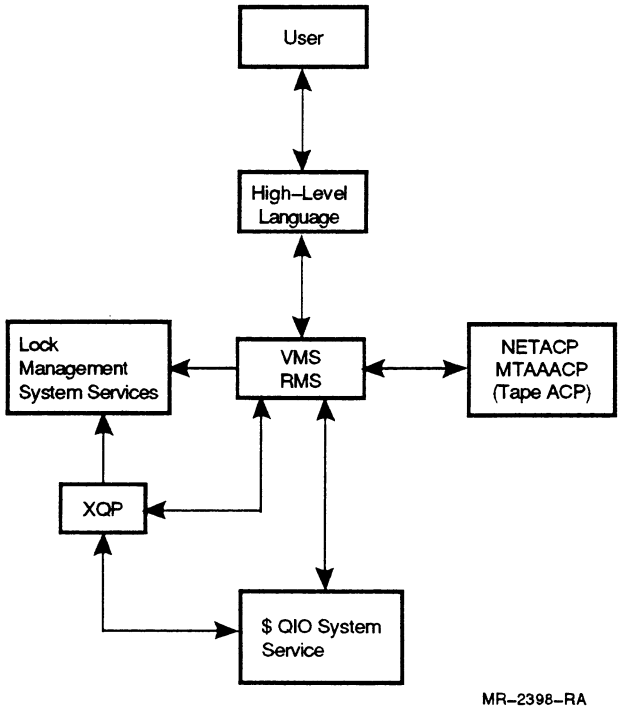
For more information on the use of ASTs and blocking ASTs with lock management system services, see Section 6.3.2.2, Using Completion ASTs and Blocking ASTs with Lock Management System Services to Synchronize Simultaneous Processes. Also refer to the *Introduction to VMS System Services* and the *VMS System Services Reference Manual* for more information on using ASTs and blocking ASTs with the VMS lock manager.

3.2 VMS Record Management Services

VMS Record Management Services (VMS RMS), illustrated in Figure 3–5, is the data management subsystem of the VMS operating system that is resident on each CPU in a VAXcluster system. VMS RMS provides a variety of disk file organizations, record formats, and record access modes from which you can select the VMS RMS features best suited to an application. In addition, you can use RMS utilities to manage and maintain file organization. For example, you can use the CONVERT utility to change from one RMS file organization to another.

Refer to the *Guide to VMS File Applications* for a complete discussion of the VMS RMS features for file organization and management.

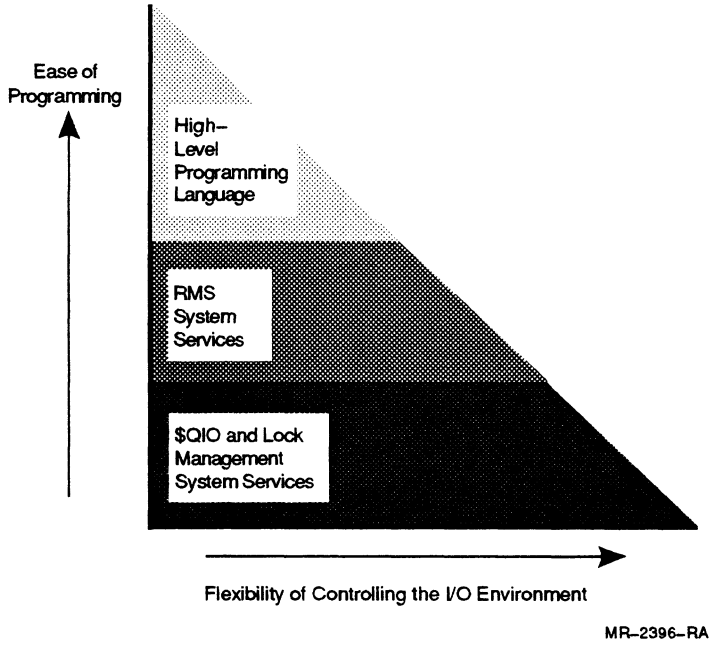
Figure 3–5 VAXcluster Programming Tool: VMS RMS



Typically, VMS RMS is invoked from a high-level programming language through language-specific processing options. In a VAXcluster system, any application executing from any CPU on the VAXcluster system uses the procedure-based code of a high-level programming language to call VMS RMS system services. For example, the OPEN statement from COBOL calls VMS RMS system services from the local VAXcluster CPU, and RMS uses the distributed VMS lock manager and the distributed file system to coordinate access. Sometimes the programmer may decide to call RMS system services directly rather than using high-level language statements for file operations. Refer to the *VMS Record Management Services Manual* for more information on the options for the VMS RMS system service.

Figure 3–6 represents the inverse relationship between the different software levels that an application can use to interface with the VMS lock manager and an application's control over the I/O subsystem. That is, as your application uses these different software levels to interface with the VMS lock manager, your application can become more difficult to program. However, when an application explicitly calls the $QIO and lock management system services, that application has the greatest control of the I/O environment.

Figure 3–6 Application Software Levels Interfacing with the VMS Lock Manager



High-level languages may support only a subset of VMS RMS features. If you intend to use VMS RMS from a high-level language, refer to your language manual to determine the VMS RMS capabilities available to you. Also, like $QIO and lock management system services, full VMS RMS capabilities are callable from any high-level language.

VMS RMS ensures safe and efficient clusterwide file sharing by performing the following functions:

- Interfaces with the distributed file system and displays messages to the requesting process for restricted file access, according to the VMS UIC-based file protection of the requesting resource.
- Uses the clusterwide lock database and the distributed VMS lock manager to provide automatic file and record locking to control data integrity of common resources.

- Provides local and global buffering to enable buffer sharing by single or multiple processes accessing a file. RMS global buffering can be used in an application executing on a single CPU or a VAXcluster system to minimize I/O operations.
- Interfaces with the $QIO system services to coordinate the functions required to process virtual I/O requests for mounted volumes.

3.2.1 VMS RMS and UIC-Based Protection

After VMS RMS requests a lock from the lock management services to access a file, VMS RMS uses the distributed file system to determine the user's right to access a file based on UIC protection. If the file has an associated Access Control List (ACL) identifier, then the ACL specification may modify the user's right to access as determined by UIC-based protection. For more information on file and device protection using UIC-based protection and ACLs, refer to the *Guide to VMS Files and Devices* and the *Guide to VMS System Security*.

3.2.2 VMS RMS and Clusterwide Record Locking

Often an application executing on a VAXcluster system has many users requesting concurrent access to the same file. VMS RMS provides record-locking capability for all sequential, relative, or indexed file organizations. VMS RMS uses automatic record locking for accessing specific records in a shared file. Automatic record locking allows a number of options for coordinating record access to RMS files. For more information on the available options for VMS RMS record locking, refer to the *Guide to VMS File Applications*. When implementing record access from a high-level language, you must use the language's keywords to specify the type of sharing that you want. The implementation of VMS RMS automatic record locking options may vary between high-level languages; therefore, refer to your high-level language manual for a discussion of its use of VMS RMS record locking.

VMS RMS system services interface with the distributed VMS lock manager to ensure that no other user can access the same record until the first user is finished with the record. VMS RMS implements individual record locking by using the distributed lock manager's capability to structure resource names as a *resource tree*. See Section 3.1.3, Locking Levels, for more information on the VMS lock manager's support of resource granularity.

VMS RMS implements record locking by having each stream accessing the same file share a common parent or root lock for the file, and each stream accessing a different record then holds a sublock that does not allow multiple access. Using this technique, VMS RMS can lock records in shared files to synchronize access to individual records by different streams, thus ensuring the consistency of the data.

However, although RMS files can be shared clusterwide down to the record level, VMS RMS is largely unaware of whether the programs calling a RMS service are executing in a VAXcluster environment; VMS RMS always refers to the locking database of a given CPU. In the VAXcluster system, VMS RMS uses the clusterwide lock database, maintained by the distributed VMS lock manager, to determine the resource's accessibility and compatibility for each locking request. When executing a file sharing application on a VAXcluster system, the programmer should be aware that there can be a difference in VMS RMS performance on a single VMS system compared to that on a VAXcluster system. If the locking request submitted by VMS RMS to the distributed VMS lock manager is not resource managed on the local VAXcluster CPU, then VMS RMS will wait slightly longer for the distributed VMS lock manager to process a locking request for a resource name that is resource managed on a remote VAXcluster CPU. This minor delay is caused by inter-CPU communications of the distributed VMS lock manager. See Section 3.1, VMS Lock Manager, for more information on *resource management*, and refer to the *Guide to VMS File Applications* for a discussion of locking considerations when programming for VAXcluster shared access.

3.2.3 VMS RMS Buffering and Global Buffering for a VAXcluster Application

VMS RMS provides you with several RMS options for specifying the size and number of buffers. Two types of buffer caches are available using VMS RMS: local and global. Local buffers reside within process memory space and are not shared among processes even if multiple processes are accessing the same file and reading the same records. Global buffers, which are designed for applications that access the same file and may even access the same records, are shared among processes (but are charged to each process' working set). VMS RMS global buffers are supported on a single CPU and across the VAXcluster system. For more information on using VMS RMS local and global buffering on a single CPU, refer to the *Guide to VMS File Applications*.

VMS RMS provides RMS global buffer caching for a VAXcluster application. Even though RMS global buffering is supported on a VAXcluster system, the programmer must remember that each VAXcluster CPU has a unique, independent global section, which is not shared by any other VAXcluster CPU. Basically, when RMS global buffering is implemented for a shared file of a cluster application executing on multiple VAXcluster CPUs, the VMS RMS resident on each VAXcluster CPU brings its own copy of the file as a global buffer from disk. When a block of data from the global buffer is updated by a process on one VAXcluster CPU, and a second process executing on another VAXcluster CPU requests the same block of data, the two processes are synchronized by RMS global buffering. RMS global buffering automatically writes the current contents of the global buffer to disk for the first process before allowing the second process to read the updated global buffer from disk.

For the case where two different processes on two different VAXcluster CPUs want to update the global buffer file, using RMS global buffering could possibly degrade performance if the two processes are highly contentious (for example, two processes on different nodes that are contending for the same global buffer). However, where two processes executing on the same VAXcluster CPU are updating a globally buffered file, there is likely to be a performance advantage because both processes have access to the same global buffer mapped to main memory. Refer to the *Guide to VMS File Applications* for more information on the use of global buffering for shared file access on a VAXcluster system. Also, in this manual, see Section 7.2, Potential Bottlenecks for an I/O-Bound Application Environment, for further information on using RMS for global buffering to increase an application's I/O performance.

3.2.4 VMS RMS and $QIO System Services

For each process on each VAXcluster CPU attempting to access a file or record resource on disk volumes, VMS RMS calls $QIO system services to perform the following functions:

- Open and create files
- Close and delete files
- Read and write files
- Modify file characteristics

A number of disk and tape I/O functions are too complex for the $QIO system service to handle. In the case of disk I/O functions, $QIO calls the appropriate XQP procedure to perform certain functions on its behalf. The XQP procedures perform file system management functions such as:

— Allocating disk space to a file

— Creating and modifying directory entries

In the case of tape I/O functions, $QIO calls the appropriate tape ancillary control process (MTAAACP) to map logical to physical I/O. See the *VMS I/O User's Reference Manual* for more information on the $QIO system services.

3.2.5 VMS RMS and XQP Operations

XQP operations ensure that all file system functions, except for simultaneous attempts to access the same resource, can proceed in parallel with file system requests issued from other processes, whether the processes are executing on the same processor or another CPU in the VAXcluster system. See the *VMS I/O User's Reference Manual* for more information on XQP operations.

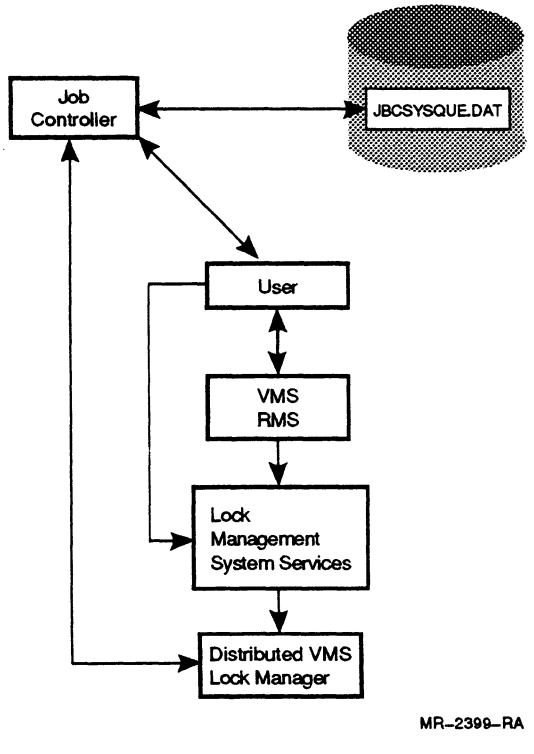
3.3 VMS Batch Facility

The VMS batch facility, illustrated in Figure 3–7, provides you with a clusterwide mechanism for batch process creation and resource allocation for clusterwide batch and print queues. In a VAXcluster system (assuming the VMS defaults for system logical names), you can submit batch jobs on the clusterwide queue SYS$BATCH, and the distributed job controller routes each batch job to the batch queue with the most available capacity for execution. The VMS batch facility provides the programmer with:

- Some degree of automatic workload balancing when the batch job is executed
- A log file as audit trail for each batch execution
- Automatic restart capability

In addition, the system manager can define local execution queues on specific nodes for batch or print services that are available only to that VAXcluster CPU. Refer to the *VMS VAXcluster Manual* for more information on setting up and managing local or generic queues in a VAXcluster system.

Figure 3–7 VAXcluster Programming Tool: VMS Batch Facility



The management of batch jobs is the responsibility of a VMS process called the job controller. While there is a job controller process executing on each VAXcluster CPU, in a VAXcluster system, all of the job controller

processes communicate with each other to coordinate the clusterwide batch queues. When all the distributed job processes are working together, their combined activity can be referred to as the distributed job controller.

All the job controller processes communicate with each other using a shared queue file and the distributed VMS lock manager. When a batch job is submitted, the job information is written to a clusterwide queue file, JBCSYSQUE.DAT. The job controller process on the VAXcluster CPU where the batch job was submitted uses JBCSYSQUE.DAT to determine the number of jobs executing on each clusterwide batch queue. Then the job controller process determines which batch queue is least busy, and uses the distributed VMS lock manager to notify the appropriate job controller process on the appropriate VAXcluster CPU that there is a batch job to execute. Upon notification, the specified job controller process reads JBCSYSQUE.DAT to determine the batch request's location, and begins execution. When the least busy batch queue is on the same VAXcluster CPU as the batch request, the job controller process does not use the distributed VMS lock manager, but handles the batch request directly.

The implementation of clusterwide print queues is similar to clusterwide batch queues. Users can queue a request to a local print queue (for a printer not necessarily attached to their own node) or let the distributed job controller choose an available print queue from those in the cluster. In addition, when using a clusterwide batch or print queue, any files to be accessed by a batch or print job must be accessible from the CPU on which the batch or print job executes.

Both batch and print queues can be declared restartable using the /RESTART DCL parameter. If a VAXcluster CPU fails, restartable jobs are either requeued to complete on another node in the cluster or executed when the failed node reboots (for jobs that must execute on a specific node). Also, you can request the restart of a batch or print job at a specific point of execution by using a DCL command procedure to SET RESTART_VALUE. For more information on using RESTART_VALUE, see Section 6.1.3 and the *VMS DCL Dictionary*.

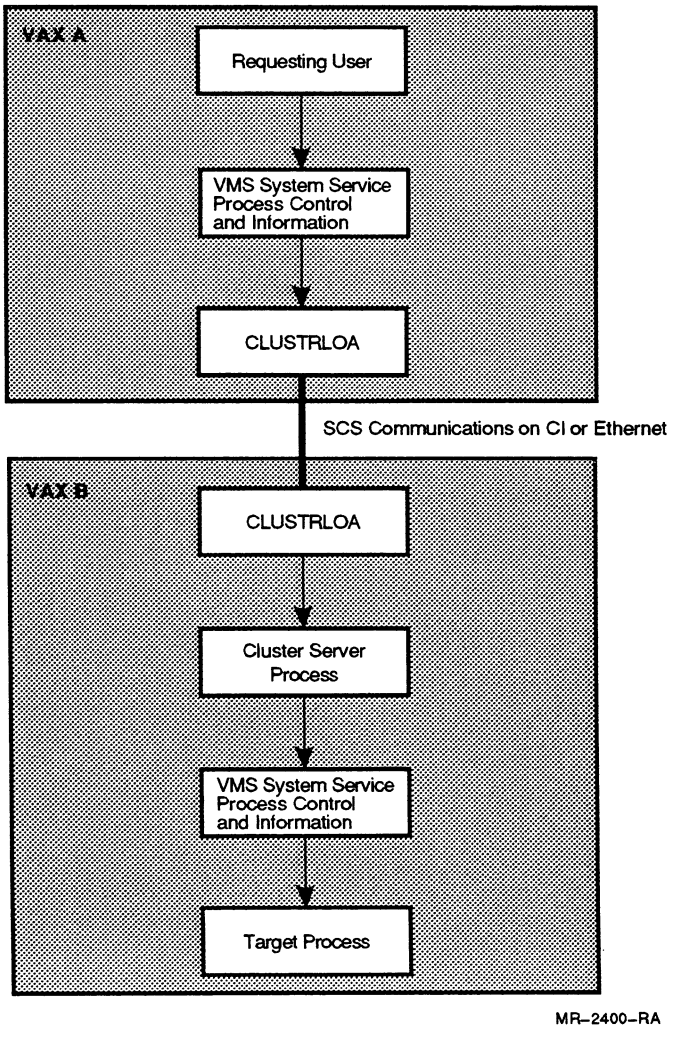
3.4 Clusterwide Process Services

Process control and process information system services, illustrated in Figure 3–8, include several programming tools that support the concept that any process executing on a VAXcluster system is an object that can be seen and manipulated clusterwide. These system services include:

- Selected VMS *process control* system services that let you perform process control services clusterwide
- Selected VMS *process information* system services that let you gather information about one process or group of processes across the entire VAXcluster system, including:
  - An extension of the $GETJPI system service to work clusterwide
  - A $PROCESS_SCAN system service for use with $GETJPI for wildcard scans throughout the cluster

These services are described in the following two sections of this manual.

Figure 3–8 VAXcluster Programming Tool: Process Control and Process Information System Services



3.4.1 Process Control System Services

You can use process control system services to examine and modify processes clusterwide. Except for $CREPRC, used to create a process, all other VMS system services for process control on a single CPU are supported clusterwide. By using the process control system services, a process on one VAXcluster CPU can direct the execution of a process on another VAXcluster CPU. VMS system services for process control across VAXcluster CPU boundaries include the system services presented in Table 3–5.

Table 3–5 Supported VMS System Services for Process Control

VMS System Service[1]	Function	DCL Interface	DCL Command[2]
$CANWAK	Cancel wakeup	Yes	CANCEL
$DELPRC	Delete process	Yes	DELETE
$FORCEX	Force image exit	No	
$RESUME	Resume process	Yes	SET PROCESS/RESUME
$SCHDWK	Schedule wake for process	No	
$SETPRI	Set priority	Yes	SET PROCESS/PRIORITY
$SUSPND	Suspend process	Yes	SET PROCESS/SUSPEND
$WAKE	Wake process	No	

[1]If you are using DECwindows Bookreader, refer to the *Introduction to VMS System Services* and the *VMS System Services Reference Manual* for a discussion of these VMS system services; otherwise refer to the *VMS Version 5.2 New Features Manual.*

[2]These DCL commands are extended clusterwide in VMS Version 5.2. If you are using DECwindows Bookreader, refer to the *VMS DCL Dictionary* for more information; otherwise refer to the *VMS Version 5.2 New Features Manual.*

When using process control system services across VAXcluster CPU boundaries, the UIC-based GROUP/WORLD privileges are processed exactly as they would be on a single VMS system.[1] You cannot do anything to a process on another VAXcluster CPU which you could not do to a process on your local VAXcluster CPU.

The process identification (PID) is a unique identifier of processes across the cluster. To reference a process on any VAXcluster CPU, specify its PID as the **pidadr** argument.

Any process executing on a VAXcluster system can use process control system services by calling the appropriate system service. All system services that control processes use **pidadr** as the first argument and **prcnam** as the second argument.

[1] As in other instances related to system management and setup, the system manager must ensure that there are no duplicate or overlapping UICs in a multiple-environment VAXcluster system or unpredictable results may occur.

The process name for the second argument **prcnam** has also been extended to reflect clusterwide accessibility. To access information about a remote process, the node name must be used as a prefix to the process name. For example, to reference the process "PROCESS_99" on node "ATHENS", use the name "ATHENS::PROCESS_99".

This change to process naming has the following implications:

- Process name strings can be up to 23 characters long with the following requirements:
  - — 15 characters for the process name
  - — 6 characters for the node name
  - — 2 characters for the colons (::) that follow the nodename
- A process name can be local or remote. Therefore, if you specify "ATHENS::SMITH", the system will check for a process named "ATHENS::SMITH" on the local node before checking node "ATHENS::" for a process named "SMITH".

Table 3–6 presents the status codes returned from the process control system services.

Table 3–6 Process Control System Services Status Codes

Status	Explanation
SS$_INCOMPAT	The remote node is running a version of VMS prior to Version 5.2, and is unable to handle the request.
SS$_NOSUCHNODE	The specified node is not currently a member of the VAXcluster system.
SS$_REMRSRC	The remote node has insufficient resources to respond to the request (bring this error to the attention of your system manager).
SS$_UNREACHABLE	The remote node is a member of the VAXcluster system, but is not accepting requests (this is normal for a brief period early in the system boot process).

For more information on process control system services, see the *Introduction to VMS System Services* and the *VMS System Services Reference Manual* or the *VMS Version 5.2 New Features Manual*. Also, for more information on programming with process control system services, see Section 6.3.1, Using Clusterwide Process Services.

3.4.2 Process Information System Services ($GETJPI and $PROCESS_SCAN)

You can either use the process information system service $GETJPI independently or with the $PROCESS_SCAN system service to obtain clusterwide process information.

$GETJPI System Service

You can use the Get Job/Process Information ($GETJPI) system service to examine information for a single process on a VAXcluster system or to perform a wildcard search on a local VAXcluster CPU. Specify the process to be examined by its process name or process identification number (PID). If a process name or a PID is not specified, $GETJPI returns data on the calling process. When using $GETJPI to examine a remote process across VAXcluster CPU boundaries, remember:

- You must specify a specific PID or process name.
- All remote $GETJPI operations are asynchronous, and must be properly synchronized.

Note: Many applications that are not correctly synchronized may seem to work on a single CPU because many $GETJPI operations are actually synchronous — but these applications will fail when examining processes on remote VAXcluster CPUs. For more information on how to synchronize $GETJPI operations, see "Synchronizing Service Completion" and "Recommended Method for Testing Asynchronous Completion" in the *Introduction to VMS System Services*.

Refer to the *VMS System Services Reference Manual* or the *VMS Version 5.2 New Features Manual* for a detailed description of the $GETJPI system service.

$PROCESS_SCAN System Service

The $PROCESS_SCAN system service is used **with** $GETJPI to perform wildcard searches on the local node or across the cluster. $PROCESS_SCAN and $GETJPI must be used together to obtain process information from a local or clusterwide search. When used with $GETJPI, the $PROCESS_SCAN system service lets you specify local or clusterwide wildcard scans to locate processes which match single or multiple selection criteria. Instead of wildcarding across all processes in a VAXcluster system to locate a few processes, $PROCESS_SCAN can locate the specified processes by using selection criteria filters.

In addition, when $PROCESS_SCAN and $GETJPI are used to extract a large amount of information from remote VAXcluster CPUs, the PSCAN$_GETJPI_BUFFER_SIZE item code can be used with $PROCESS_SCAN to specify a buffer size for the execution of $GETJPI on a remote VAXcluster CPU.

For more information on the use of $PROCESS_SCAN with $GETJPI, refer to the *VMS System Services Reference Manual* or the *VMS Version 5.2 New Features Manual*. Also, in this manual, see Section 6.3.1 for more information on programming with process information system services.

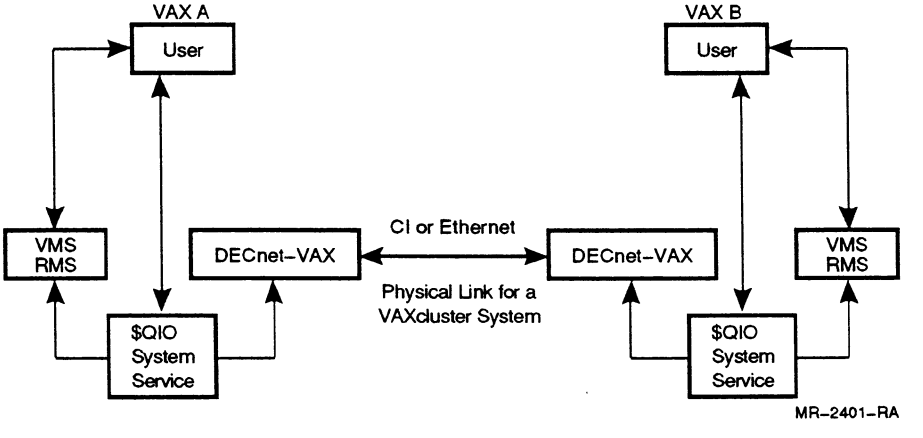
3.5 DECnet-VAX

DECnet-VAX software is the implementation of DECnet protocol that enables a VMS operating system to function as a network node. A DECnet-VAX node can communicate with other DECnet-VAX nodes in the VAXcluster system, or with any other operating system in the network that supports the DECnet network. As illustrated in Figure 3–9, DECnet-VAX nodes can communicate directly with each other without having to go through a central node.

DECnet-VAX is required in any VAXcluster system. (Refer to the *VMS VAXcluster Manual*.) With DECnet-VAX:

- You can perform task-to-task communications either between VAXcluster CPUs or to any networked CPU.
- You have access from every VAXcluster CPU to disks that are located on another system in the network.

Figure 3–9 VAXcluster Programming Tool: DECnet-VAX



DECnet Task-to-Task Operations

DECnet-VAX uses data communications hardware and software to establish a *logical link* between a sending (source) program and a receiving (target) program. For a logical link to exist between the source and target programs, the process executing the source program must be able to create a process on the CPU where the target program is executing.

There are three methods for controlling the access of a source process attempting to create a process on the target CPU:

- The source process is given explicit access to a user account in SYSUAF.DAT of the target CPU. A process is created on the remote node when the source process supplies the appropriate username and password.
- The source process is given access to an enabled proxy account in NETUAF.DAT of the target CPU. A process is created on the remote node if the source process is defined in the remote node's proxy database.
- If neither of the first two conditions exist, the source process can have default access to an enabled DECnet account in the SYSUAF.DAT of the target CPU.

When the source process creates a process on the target program's CPU using one of these three methods, the method of process creation defines the type of process created according to the UIC, default directory, and privileges in the associated SYSUAF.DAT or NETUAF.DAT. If the source process has logged in to the target CPU using the third method, Digital provides the following defaults for the DECnet account:

UIC — [376,376]
Default directory — SYS$SYSDEVICE:[DECNET]
Privilege — TMPMBX, NETMBX

Even though a user can create a process on another DECnet node using the default DECnet account, for that process to run a .EXE on the target CPU, the remote process must have file access privilege to the executable image.

After the logical link has been established, the application programmer specifies a series of system service calls in both the source and target programs to direct the operation of the logical link. See Section 3.5.1 and Section 3.5.2 for more information.

In addition to using DECnet-VAX to connect a user program to another user program, you can also connect a user program to a DECnet network object that has been defined on the target CPU using the Network Control Program (NCP) object database. For example, a process executing a FORTRAN program on a source node could request a logical link to the remote DECnet node PLUTO:

```
OPEN (UNIT=1, FILE='PLUTO::"TASK=XYZ"', TYPE='NEW' )
```

In this case, when the FORTRAN source program executes the OPEN statement, the DECnet software on the target CPU searches the NCP object database for a task that has been defined as XYZ by the system manager using the NCP Utility. If the task has not been defined in the NCP database for the target CPU, the DECnet software searches the process' default directory. If, as in the preceding example, the default DECnet account is enabled on node PLUTO, then the default DECnet directory SYS$SYSDEVICE:[DECNET] is searched for XYZ.EXE.

In addition to using the NCP Utility for network task definitions, NCP also contains Digital-supplied network objects. For example, when the VMS Mail and VMS Phone Utility are used for communication between network nodes, these network objects are referenced through the NCP object database. See the *VMS Networking Manual* for more information on using the NCP object database.

Whether you are using DECnet-VAX to connect to a user program or to a network object, the DECnet software that manages each end of the logical link guarantees the following:

- All transmitted data is delivered to the destination node in proper sequential order.
- All data received by DECnet software on the destination node is given to the target program in proper sequence.

To guarantee proper sequencing, DECnet software numbers the segments it transmits over a link. The receiving DECnet software, using the transmit numbers for identification, acknowledges the delivery of the segments. If a segment is not acknowledged within a certain period of time, the sending DECnet software retransmits it.

Remote File and Record Access

Using DECnet-VAX, a program executing on a VAXcluster CPU can access a file on any other network node providing that it has file access privileges. However, as in DECnet task-to-task communication, when the user requests access to a remote file, the user must create a process on the remote network node. To access a remote file, the remote process must have file access privileges to the remote file. DECnet remote file access capability can be used with the following:

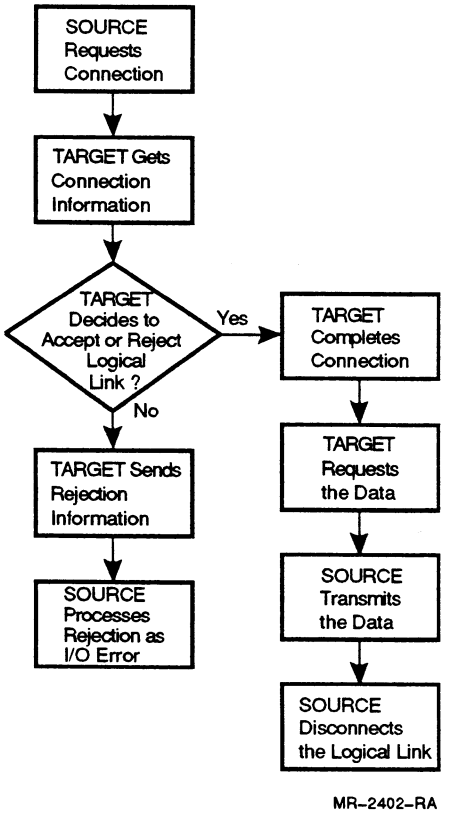
- VMS DCL commands for remote file operations (copy, delete, submit, append)
- A user-written program to perform record-level operations on a remote file

When you access a remote file using either of these remote file access methods, a process at the remote node performs the file access on your behalf. As in task-to-task communication, the two processes must establish a logical link before they can begin to execute file operations. The access control methods for remote file access are exactly the same as those described for DECnet task-to-task operations. For more information about DECnet-VAX remote file access, see the *Guide to DECnet-VAX Networking*.

3.5.1 Transparent DECnet-VAX Task-to-Task Communication

In a VAXcluster system, transparent DECnet-VAX techniques enable two programs executing on different VAXcluster CPUs to perform task-to-task communication — that is, exchange data over a logical link. With DECnet-VAX, software performing task-to-task communication is similar to performing input/output (I/O). The logical link represented in Figure 3–10 between the two programs (SOURCE and TARGET) is an I/O channel over which both programs can send and receive data.

Figure 3–10 Transmitting DECnet-VAX Task-to-Task Data



Task-to-task communications are considered transparent because DECnet-VAX software can be invoked by using standard I/O statements from some high-level languages, for example, FORTRAN READ and WRITE commands. The DECnet-VAX task-to-task capability translates each DECnet-specific high-level language command into the same set of DECnet software messages regardless of the language in which they are programmed. Therefore, when designing task-to-task operations, you can ignore the programming language of the executable image on the remote network node. You must, however, consider what high-level languages are installed on your local VAXcluster CPU since not all high-level languages support DECnet-VAX communications. Refer to the *VMS Networking Manual* for a list of the programming languages you can use for DECnet-VAX transparent task-to-task communications and for use of VMS DCL for this purpose.

The programmer can also invoke transparent DECnet by using the following system service calls from MACRO or a higher-level language:

- $ASSIGN — request a logical link connection
- $QIO — send (write) or receive (read) a message
- $DASSGN — terminate a logical link

Refer to the *VMS Networking Manual* for a discussion of the system service calls for transparent DECnet-VAX operations.

3.5.2 Nontransparent DECnet-VAX Task-to-Task Communication

Using nontransparent DECnet-VAX task-to-task communications enables a source task to manage multiple connection requests created over a single physical line by defining an I/O channel for each logical link request. Inbound requests are queued to a network *mailbox*, and a logical link is assigned and deassigned. The DECnet-VAX programming mechanism for coordinating multiple link requests from target tasks to a source task is to associate a different AST with $QIO requests for each target.

Table 3–7 shows the high-level *protocol* used between two processes running nontransparent DECnet task-to-task communication. The processes, called SOURCE and TARGET, are running on different DECnet-VAX nodes in a VAXcluster system. SOURCE is designed to send data to TARGET.

Table 3–7 Using Nontransparent DECnet-VAX Communication

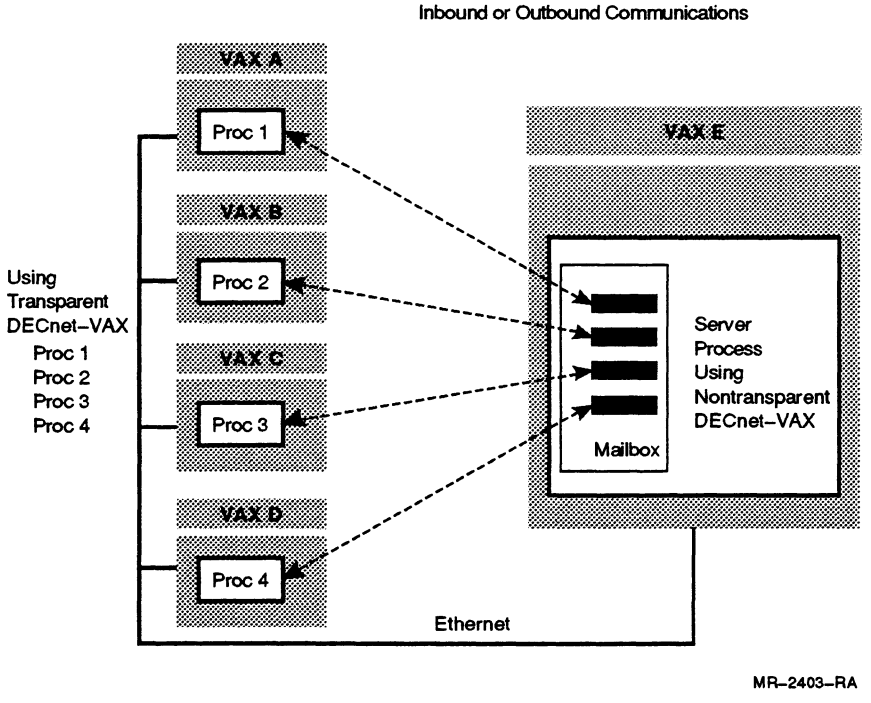
Step	SOURCE Action (Local Node)	TARGET Action (Remote Node)
1	Assigns an I/O channel.	Creates a network mailbox.
2	Establishes a mailbox.	Establishes a network channel associated with the network mailbox.
3	Sends a connect request to the mailbox on the remote node where TARGET is running.	
4		Reads the network mailbox for connect requests. Starts an existing command file on the remote node, which either accepts or rejects the connect request.
5		Sends a message to SOURCE indicating that the connection has been accepted or rejected.
6	Reads the message and determines its action depending on whether TARGET accepted or rejected the connection.	
7	If the connection is established, may request information from TARGET, or may send data to TARGET immediately.	Responds to requests for information and accepts data as SOURCE sends it.
8		When all data has been accepted, sends a disconnect message to SOURCE.
9	Receives the disconnect message and deassigns its I/O channel.	
10		Deassigns its I/O channel.

Using nontransparent DECnet-VAX, you can:

- Communicate with network nodes outside the VAXcluster system, just as you can with transparent DECnet-VAX
- Establish multiple inbound or outbound logical links

  Typically, a nontransparent DECnet-VAX program communicates to or receives communications from many independent programs that are executing on different network nodes. Figure 3–11 illustrates how independent programs can send messages to and receive messages from the nontransparent program using transparent DECnet-VAX communications.

Figure 3–11 One-to-Many or Many-to-One Nontransparent Communications



- Use network system messages to determine the status of each network channel for multiple logical links

  Monitoring DECnet-VAX system messages helps facilitate error recovery. See Section 6.4 for more information on using DECnet-VAX system messages for error recovery.

Nontransparent DECnet uses the following VMS system service calls:

- $ASSIGN — assign an I/O channel
- $QIO — request and accept a logical link connect, and send (write) or receive (read) a message
- $CREMBX — create a mailbox
- $CANCEL — cancel I/O on a channel
- $DASSGN — deassign an I/O channel

Refer to the *VMS Networking Manual* for more information on using VMS system services for nontransparent DECnet-VAX operations.

Comparison of Transparent and Nontransparent DECnet Communication

Table 3–8 summarizes the differences between transparent and nontransparent communication.

Table 3–8 Transparent and Nontransparent DECnet Communication

Transparent DECnet Communication	Nontransparent DECnet Communication
Easy to program	Harder to program
Less flexible	More flexible
To complete link request, requires process creation and image activation on the remote node	To complete multiple link requests, requires multiple process creations and image activations on remote nodes
Process on remote node can only handle a link from a single process on local node	Process on remote node can receive link requests from multiple processes by creating a mailbox and declaring an attention AST
	Can communicate with multiple processes on other nodes

3.6 Single-Node Programming Tools Not Available Clusterwide

The following VMS programming tools work on a single VMS system, but are not supported clusterwide:

- Permanent and temporary mailboxes
- Common event flags
- Logical names
- $CREPRC
- Writeable global sections

Note: While a writeable global section is supported clusterwide, it is not recommended because of the programming complexity required to handle clusterwide, concurrent update operations.

Permanent and Temporary Mailboxes

You cannot use permanent and temporary mailboxes to communicate between processes on different VAXcluster CPUs. Either you can use DECnet-VAX communications with a mailbox or you can use a resource lock with a lock value block (with a capacity of 16 bytes) to store information passed between processes.

For more information on using DECnet-VAX mailboxes for clusterwide task-to-task communications, see Section 3.5.2 and Section 6.1.2.

Common Event Flags

You cannot use common event flags as a signalling mechanism for processes on different VAXcluster CPUs. However, if a program has been designed to wait for a common event flag, and you want to implement the program as part of a clusterwide application, you can use the lock management system services as a clusterwide signalling mechanism. You can modify the program to perform a completion or blocking AST routine for a lock request as a signalling mechanism for event notification between processes. For more information on using lock management system services for clusterwide process synchronization, see Section 3.1.6 and Section 6.3.2.

Note: The use of process-private event flags for synchronizing operations within a single process (for example, in conjunction with $QIOs) is allowable and necessary in a VAXcluster application since no synchronization with other processes is involved.

Logical Names

As long as your VAXcluster is configured as a common-environment, the logical names will be the same on all of the VAXcluster CPUs at system start-up. However, some applications are designed to communicate by defining logical names in one process and translating those names in another process. If you change a logical name on one CPU, the logical name tables will not be updated on the other CPUs.[1] If you need to dynamically change logical names on more than your local node, you may be able to substitute a resource lock if the value of the logical name is not more than 16 bytes. If 16 bytes of a lock value block is not sufficient, you can use a disk file that is accessible clusterwide, or you can create a process on each of the VAXcluster CPUs that you want to change a logical name. For more information on methods for creating a remote process, see Section 6.1.

$CREPRC

While some process control system services are supported clusterwide,[2] $CREPRC is **not** supported clusterwide. In order to create a remote process on a different VAXcluster CPU from a local process, you can use the following programming techniques:

- VMS Batch facility
- DECnet-VAX communications

---

[1] You can use the SYSMAN Utility to update a logical name on all existing cluster nodes, but any new node joining the cluster will not contain the logical name changes. For more information on the use of the SYSMAN Utility, refer to the *VMS SYSMAN Utility Manual*.

[2] A specific group of Process Control and Process Information system services **are** supported clusterwide in VMS Version 5.2. See Section 3.4.1 for a discussion on how to program using the Clusterwide Process Services.

In addition, you can also start a process on a remote VAXcluster CPU at system start-up and place it into *hibernation*. This process then can function as a server that is awakened when there is a link request for service. You can either use DECnet-VAX communications to establish the logical link and request service or you can use the clusterwide process control system services. For more information on programming techniques for remote process creation, see Section 6.1.

Writeable Global Sections

Converting the writable global sections in an existing program that is accessed by multiple users on the same VAXcluster CPU to a clusterwide application is difficult. In order to accomplish this, you need to carefully consider the interprocess communication requirements so that each process that accesses the writable global section is mapping the most current version of the global section. For more information on the use of writable global sections for shared access on a VAXcluster system, see the *Guide to VMS File Applications*.

4 Application Design Models for VAXcluster Software

This chapter introduces three application design models to consider when you design VAXcluster software:

- File Sharing Model
- Client-Server Model
- Parallelism Model

These models use both distributing by replication and by decomposition for designing an application for a VAXcluster system.

The model descriptions include:

- Advantages and disadvantages of using each model
- Implementation requirements for the model with respect to:
  - — Remote Process Creation

    A local process may need to create a process on a remote VAXcluster CPU.
  - — Data Sharing

    Processes in a VAXcluster environment can share data from a common data resource and processes can exchange messages using DECnet-VAX communications.
  - — Process Synchronization

    Using a programming mechanism for interprocess communication in a VAXcluster system, processes can synchronize their activities based on exclusion or barriers.

    With process synchronization based on *exclusion*, process activity is regulated by granting only one process at a time access to a resource. With *barriers*, process activity is regulated by defining a "time" in the processing cycle when all processes must be in the same state. A barrier can be a point of synchronization at either the beginning or end of an application's parallel streams.
  - — Exception Conditions

    An application that is distributed by decomposition must be able to recover if there is an interprocess communication error or hardware failure for a task being performed on a remote VAXcluster CPU.

  For more information on the programming techniques for these implementation requirements, refer to Chapter 6.
- An example of an application using the model under discussion

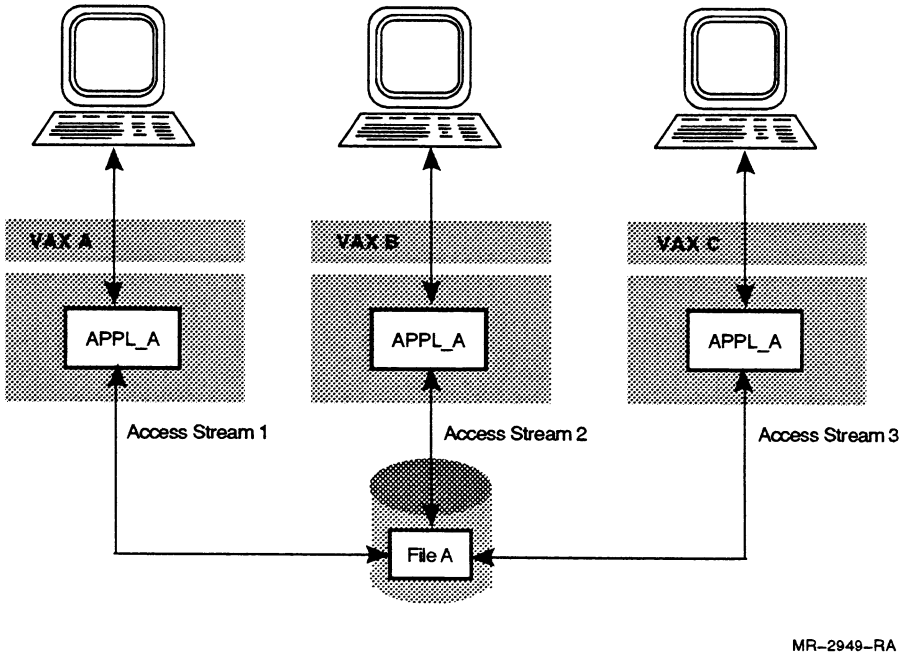
4.1 File Sharing Model

Using the File Sharing Model, multiple users can obtain coordinated access to shared file elements. An application using this type of model has the following characteristics:

- Identical code is run on multiple VAXcluster CPUs.
- Multiple VAXcluster CPUs can potentially share file access.
- In a VAXcluster system, VMS RMS uses the distributed lock manager to arbitrate concurrent file access.

Figure 4–1 illustrates the implementation of a File Sharing Model where executable images of an application are distributed across multiple VAXcluster CPUs. For more information on how to determine if your application is a candidate for distribution by replication, see Section 2.4.

Figure 4–1 File Sharing Model Using Distribution by Replication



Advantages of Using the File Sharing Model

Using the File Sharing Model in a VAXcluster system offers the following advantages:

- When the total number of users remains constant, distribution by replication offers:
  - — Increased application availability
  - — Increased application throughput
- The File Sharing Model enables you to design applications with either the simultaneous running of multiple copies of a given program, or the simultaneous running of different programs, each performing different functions but using the same data files. (There is no dependency between the programs other than the data in the files.)
- File Sharing Model implementation is made relatively simple because the interprocess communication required for synchronizing clusterwide file access is implicitly handled for you through the VMS RMS interface. You can use a high-level language to interface with VMS RMS, and VMS RMS uses the distributed VMS lock manager to coordinate file access for clusterwide users.
- Based upon a File Sharing Model, a multi-user application can be migrated to a VAXcluster environment with no modifications to the application.

Disadvantages of Using the File Sharing Model

Using the File Sharing Model in a VAXcluster system has the following disadvantages:

- An increased process demand for data elements on the same disk can create a disk I/O bottleneck that will tend to degrade an application's performance.
- Many users in contention for the same data element can degrade an application's performance.
- Locking requests that are mastered by the distributed VMS lock manager on a remote VAXcluster CPU will take slightly longer to process than those mastered on the local VAXcluster CPU.
- If the File Sharing Model is implemented for network users, the application's throughput for a network user will tend to degrade because of the overhead associated with DECnet-VAX communications. To access the image, a network user can either use DECnet-VAX communications to SET HOST to a VAXcluster CPU or execute the image remotely and perform file activities across a network channel. In both cases, the network user can encounter a bottleneck on the communications link.

Implementation Requirements for the File Sharing Model

The File Sharing Model has the following implementation requirements:

- Remote Process Creation

  The File Sharing Model does not require *remote process creation* as an implementation technique; the replicated image is executed on each VAXcluster CPU selected for distribution.

- Data Sharing

  The File Sharing Model requires that multiple processes have clusterwide access to shared data. Use VMS RMS to coordinate multiple users that are concurrently accessing records for READ or WRITE operations. Depending upon your application, you can use disk file Read-Only global sections for data sharing if you are not modifying information in a global section. In general, this type of application design does not use DECnet-VAX communications for message passing between processes executing the image. For more information on performance considerations for a potentially I/O-bound application environment, see Section 7.2.

- Process Synchronization

  VMS RMS implicitly provides process synchronization by allowing you to restrict access to a record to one process at a time. Barriers are not commonly needed to synchronize processes; explicit interprocess communications can create additional overhead that could degrade an application's I/O performance.

- Exception Conditions

  If a VAXcluster CPU running a distributed image fails, the user can recover by logging onto a functioning VAXcluster CPU and re-executing the application image. To facilitate complete recovery and help you re-establish the transaction context when your process executing the image failed, use VAX RMS Journaling. For more information on VAX RMS Journaling, see Section 5.5.1.

Example of an Application Using the File Sharing Model

With the File Sharing Model, you can use your VAXcluster system to provide the users of an electronic course registration application with maximum availability. If a VAXcluster CPU fails, the course registration application remains available on the functioning VAXcluster CPUs.

For example, the components of Technical University's course registration application enable the user to do the following:

1 Log in to the application

2 Search and display all available courses according to a keyword

3 Select a course and view a course description

4 Register or withdraw from a selected course

To use Technical University's course registration system, follow the steps illustrated in Figure 4–2, Figure 4–3, Figure 4–4, and Figure 4–5.

Figure 4–2 Log In to the Application

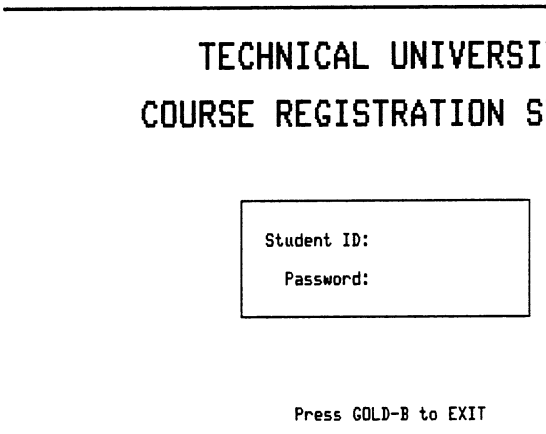


Figure 4–3 Search and Display All Available Courses According to a Keyword

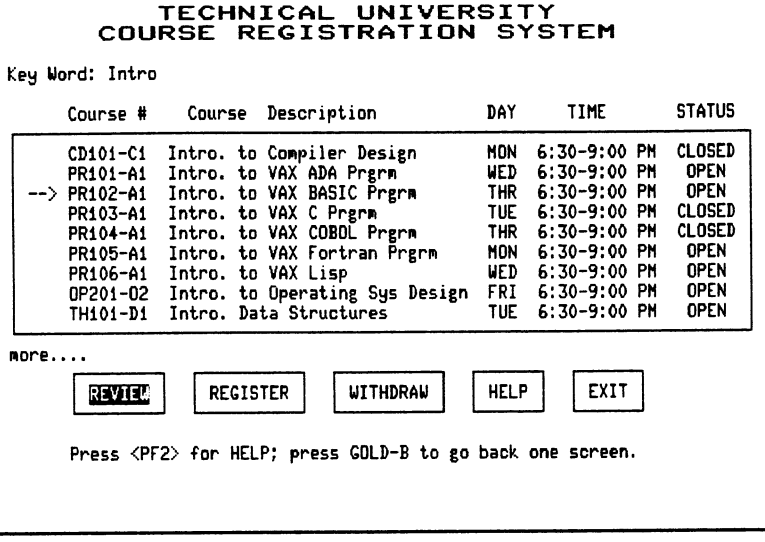


Figure 4–4 Select a Course and View a Course Description

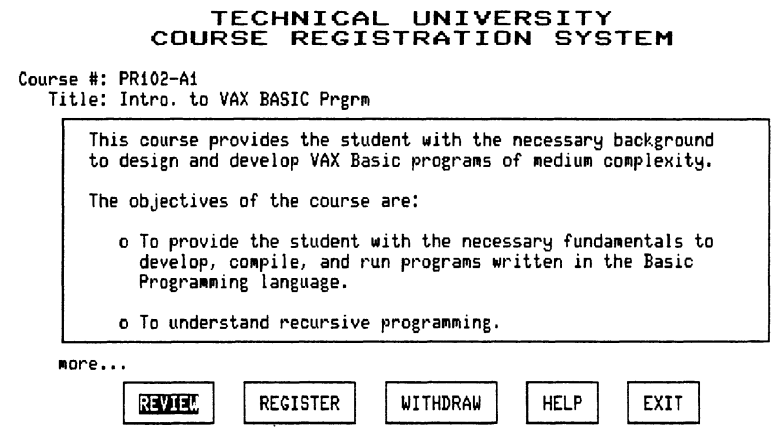
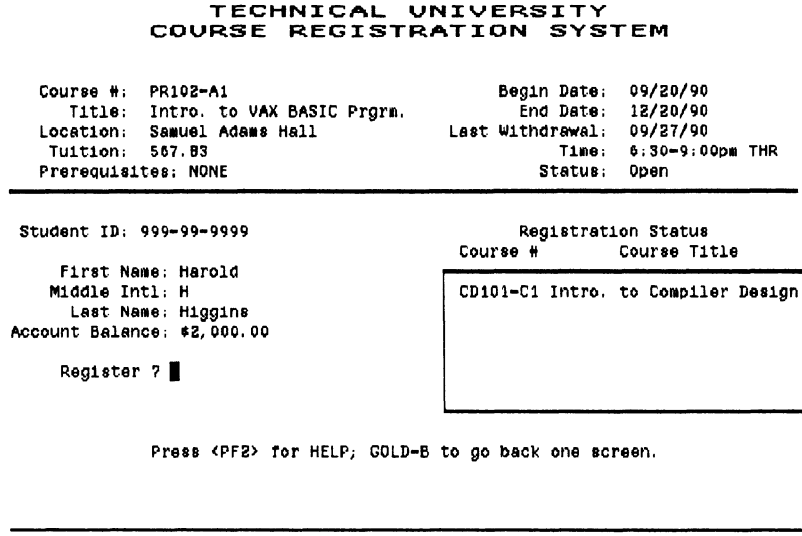


Figure 4–5 Register or Withdraw from a Selected Course



4.2 Client-Server Model

The Client-Server Model presents a method for decomposing an application into its interrelated functional units; one functional unit is the client process and the other is the server process. (For more information on how to decompose an application, see Section 2.5.) The client-server relationship is based on a client process requesting work to be done on its own behalf, and a server process performing the work requested by a client. Typically, a server process is designed to more effectively implement a specific feature of an application. While the server process

is processing the request of the client, the client process can wait or perform other application activities. Whether the client process waits for the server process to complete its request or works in parallel depends on your specific application. In general, there are two variations of the Client-Server Model:

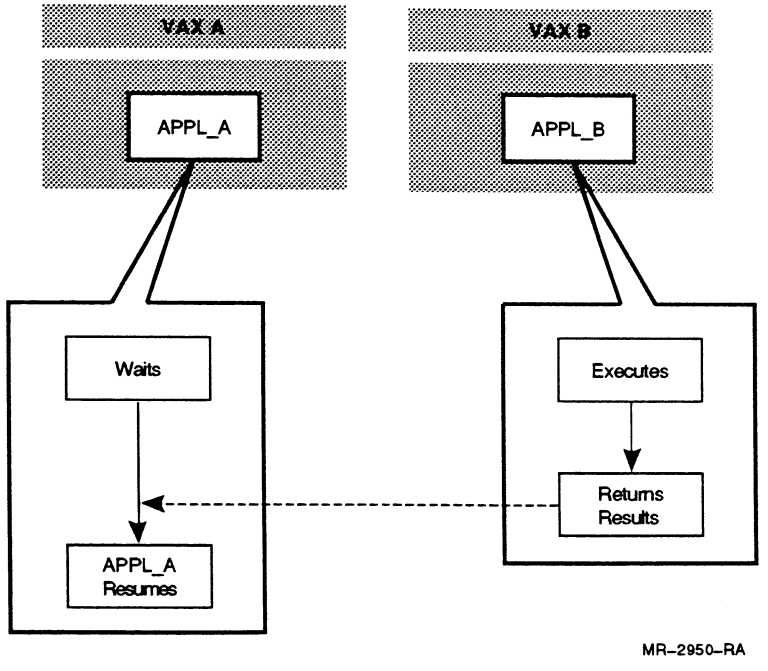
- One-to-One Client-Server Model
- Many-to-One Client-Server Model

These two models are based on two different client-server relationships: a one-to-one relationship or a many-to-one relationship. These two variations of the Client-Server Model are discussed in the following sections of this manual.

4.2.1 One-to-One Client-Server Model

The simplest version of the One-to-One Client-Server Model is illustrated in Figure 4–6. A client process (APPL_A on VAXA) creates a remote server process and waits while the remote server process (APPL_B on VAXB) performs some work. When the remote server process notifies the client process that the remote work is completed, the client process resumes work.

Figure 4–6 One-to-One Client-Server Model



Advantages of Using the One-to-One Client-Server Model

Using the One-to-One Client-Server Model offers the following advantages:

- A server process can provide a special function to help increase an application's throughput.
- When you design an application as a One-to-One Client-Server Model, it lets you decompose an application's tasks to take advantage of a hardware resource.
- When you implement a VAXcluster-based server process for a network client, you can improve the network client's performance by having a server process perform disk I/Os locally.

Disadvantages of Using the One-to-One Client-Server Model

Using the One-to-One Client-Server Model has the following disadvantages:

- It may require more system resources if each server only provides the service to one client at a time.
- If the One-to-One Client-Server Model is implemented, based on synchronous communications, the client process may lose some processing time while waiting for the server process to complete the work request.

Implementation Requirements for the One-to-One Client Server Model

The One-to-One Client-Server Model has the following implementation requirements:

- Remote Process Creation

  The server process can be created on an appropriate cluster node at application startup or on an as-needed basis. If you are sure the server will be needed, it probably makes sense to start it at application startup and use interprocess communications to tell the server when there is work. If you are not likely to need the server, it can be created on an as-needed basis. Since process creation is expensive, it makes sense to keep a server process alive until you are sure that it will no longer be needed.

  DECnet-VAX transparent communication is an effective technique for remote server process creation in the One-to-One Client-Server Model, because only one logical link is required for communications between the client and server processes. However, if there is an elaborate communication pattern between the client and server processes, nontransparent DECnet-VAX communications may provide more flexibility. In addition, when a client-server relationship is contained within your VAXcluster system, the client process can use process control system services to issue $WAKE and $HIBER system service calls to awake and hibernate the server process as needed.

  The VMS batch facility is also an appropriate technique for remote process creation, especially if the application is written in DCL.

- Data Sharing

  If the client and server processes are concurrently accessing the same file, use VMS RMS to coordinate file access for the client and server. Depending upon the application, data sharing can be implemented by using disk-file Read-Only global sections. If there are ongoing information exchanges between the client and server processes, you can use DECnet-VAX communications for message passing. For application performance, you must decide whether to send numerous DECnet-VAX information exchanges for small units of data or to use large data exchanges.

- Process Synchronization

  You can construct a communication protocol for information exchanges between the client and the remote server using DECnet-VAX communications or lock management system services. You can establish a communications path between the client and server processes by using either DECnet-VAX communications for message passing or lock management system services for event notification. Also, you may want to use a combination of DECnet-VAX communications and resource locks with blocking ASTs to communicate between the two processes when information is ready to be sent or received.

  With DECnet-VAX communications, the client and remote server processes can be designed to alternate execution, as in a dialogue. In this case, the client or server process waits and then wakes up when an AST is delivered by the "working" processes' $QIO WRITE with the results. Even though there is a dialogue between the client and server, the client process remains in control of the client-server relationship by making requests to the server.

  When using lock management system services, the client and remote server processes use lock management system services for event notification to synchronize their activities. If the client process takes out an EX lock on a resource name with a blocking AST, when the remote server process completes the client's work request, the remote server requests an incompatible lock for that resource name. This triggers the blocking AST and the client process executes an AST routine to get the results.

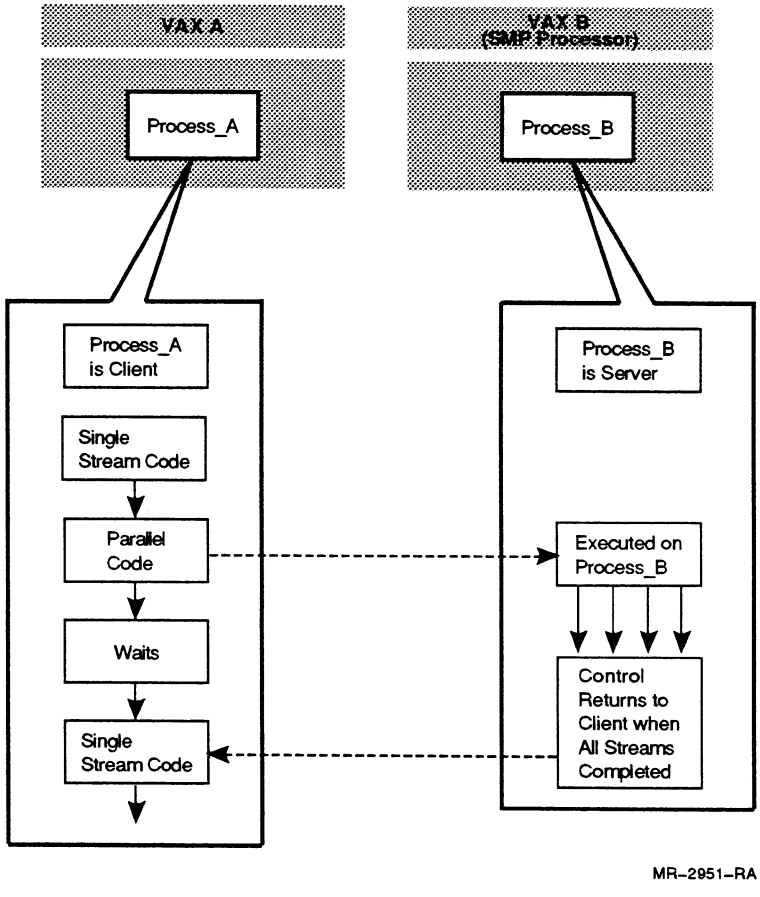
- Exception Conditions

  The client process must detect DECnet-VAX error messages and perform the appropriate recovery. If the client and server are in a dialogue with alternating execution, the client and server should have a time limit for their wait periods. Consequently, if either process terminates unexpectedly or the node where it is running fails, the remaining process can recover. In some cases, a deadman lock scheme (see Section 6.3.2.2) may be used to terminate both the client and server process when either process fails. It is also possible to implement the One-to-One Client-Server with a backup server process to take over the client's requests in the event that the primary remote server fails.

Example of an Application Using the One-to-One Client-Server Model

The One-to-One Client-Server Model illustrated in Figure 4–7 demonstrates a client-server relationship for using a compute resource in a VAXcluster environment. Process_A is a FORTRAN application with a section of code for parallel execution, and Process_B is a remote server for a SMP processor. Instead of designing Process_A to execute the parallel section by spawning subprocesses on VAXA, Process_B creates a process on VAXB to execute a program containing Process_A's parallel section of FORTRAN. VAXB provides a hardware resource (a SMP processor) that can execute the FORTRAN parallel section more rapidly than executing the parallel streams as subprocesses on VAXA. All of the parallel streams executed by Process_B on VAXB are synchronized to form a barrier; at the completion of all the streams, Process_B returns control to Process_A.

Figure 4–7 Using a One-to-One Client-Server Model for Parallel Processing



4.2.2 Many-to-One Client-Server Model

The Many-to-One Client-Server Model uses a client-server relationship to manage various requests to a server for special services from multiple clients. Typically, the server process is installed as a resident process on a VAXcluster CPU and multiple client processes in a VAXcluster environment tend to come and go, only using the remote server as an interface to a resource. A remote server can provide access to a resource by functioning as one of the following:

- Compute server
- Specialized hardware server
- Database or file server

Compute Server

A server designed as a compute resource can provide a special compute function for several clients that are all performing the same computations. Depending on the amount of work to be done, it may be more efficient to execute a compute-intensive program segment for an application on one of the fastest VAXcluster CPUs than to execute the compute-intensive section of an application on another VAXcluster CPU. Using a remote server process as a computer server can allow you to more efficiently use the processors in your VAXcluster system.

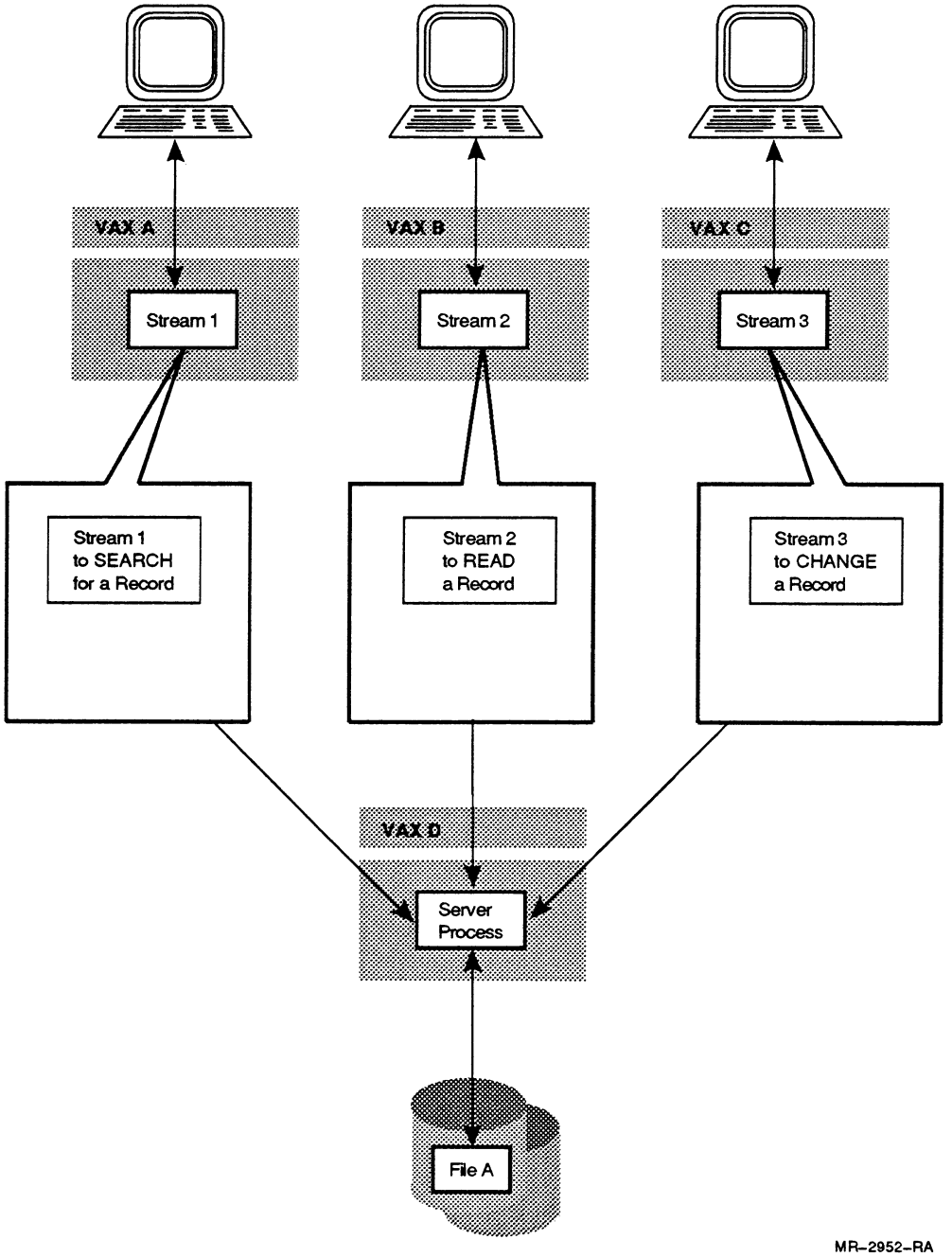
Specialized Hardware Server

There are many types of specialized hardware servers: for example, a hardware server could be a print server, a terminal server, or a network server. For any type of hardware server, there are many clients requesting the use of that specific hardware resource. The hardware server schedules the client requests for use of that resource. For instance, the PrintServer 40 Supporting Host Software is a layered product that enables a suitably configured VMS system within a DECnet Ethernet network to provide support functions for the PrintServer 40.

Database or File Server

Figure 4–8 illustrates the implementation of a remote server process for multiple clients requesting access to the same file or database. Each client process communicates a file operation request to the server, and the remote server process accesses the file or database on behalf of the client.

Figure 4–8 Many-to-One Client-Server Model as a File Server



A file or database server is probably the most common implementation of the Many-to-One Client-Server Model in the VAXcluster application environment. Rather than having an application access a file directly, the application requests file access from the remote server process.

There are three reasons for accessing a file through a remote file server process:

1 Restrict file access to protect a file's integrity

To restrict file access, you can design a remote server process to serve only one client at a time. Once a client process' request is granted by the server process, all other clients must wait until the server completes the first request. In this way, the server process synchronizes access to a file or database. The server is the owner of this file or database because all clients that request access to the file or database contents must make their request through the remote server. This method of implementing a remote server is used to:

— Enforce additional security by only letting the server process access the resource

— Simplify network access to a shared resource

— Optimize access to a resource by keeping all users of that resource on a single node in a single process

2 Avoid the locking overhead of simultaneous updates to the same file from several different applications

To avoid a large locking overhead associated with numerous processes using VMS RMS for simultaneous updates to the same file, you can design a remote server process to manage multiple clients requesting access to the same file. By maintaining a communication path to each client, the server is a common collecting point for all of the clients requesting access to the server's database. However, because all file activities must pass through the server, the server can help you avoid programming for simultaneous updates from several different applications. You can build the server software to control the order (from the application standpoint) for processing file updates that originate from different applications. Also, you can construct the server to let clients perform different file functions. For example, a server process could provide every client with the following functions:

— Read a database record

— Change a database record

— Search for a database record

3 Make more efficient use of system resources

Using one remote file server to service network access requests from multiple clients is an efficient use of system resources. It is often much less expensive, in terms of system resources, to use a Many-to-One Client-Server Model and design a server process that can service multiple clients. If each of the clients must use separate communications paths to access a remote file resource, you must create a separate process for each client request on the target CPU. In addition, if a process accesses the remote file to perform a search operation for a particular string, all the data bytes in the file must travel over the network to the local process where the comparison against the search string is made. However, when you perform the

search operation using a server, a client sends a protocol message saying "SEARCH FOR STRING X" and the server would take it from there. No data would be transmitted over the network, as the search takes place entirely within the server. Consequently, only the result of the search needs to be transmitted from the server to the client.

To assess if your application can be designed to use any of these variations (compute server, specialized hardware server, database or file server) of the Many-to-One Client-Server Model, you should determine if your application is a candidate for distribution by decomposition. For more information on how to determine if your application is a candidate for decomposition, see Section 2.5.

Advantages of Using the Many-to-One Client-Server Model

Using the Many-to-One Client-Server Model offers the following advantages:

- Since process creation and image activation are expensive in terms of CPU cycles and elapsed time, it can improve performance for many clients to share one server process.
- The end users of an application implemented with a server cannot see any difference in the way their requests are handled between a per-process or server application; the difference in implementation is completely transparent.
- When a VAXcluster CPU is dedicated to a server process, there is a performance advantage if a large database is being accessed for a small subset of information.
- Use of the Many-to-One Client-Server Model can increase throughput for a compute-bound application. In general, the application designer can create the remote server process on the most powerful CPU node and use the remote process to execute critical regions of code that are compute-intensive. This implementation performs most efficiently when the data from the client's request can be accessed by the server through a DECnet $QIO. For more information on performance considerations for a potentially *CPU-bound* application environment see Section 7.4.

Disadvantages of Using the Many-to-One Client-Server Model

Using the Many-to-One Client-Server Model has the following disadvantages:

- Since the Many-to-One Client-Server Model does require more communication paths and more complicated communication, it will generally take longer to program than the One-to-One Client-Server Model.
- When you use the Many-to-One Client-Server Model for file serving activities, you must design interprocess communications between each client process and the remote server. Compared to using VMS RMS for each process to separately access a file, constructing a server process requires implementing explicit interprocess communication.

- To access large amounts of data from the database, it is more efficient to directly access the data from each process executing the application.
- After a client sends a request to the server and the remote server process is executing, the client usually waits for the server; thus, the client process does not utilize the computing capability of its CPU.
- If this model is implemented without a backup server, then the failure of the server will impact all of the clients.

Implementation Requirements for the Many-to-One Client-Server Model

The Many-to-One Client-Server Model has the following implementation requirements:

- Remote Process Creation

  As with the One-to-One Client-Server Model, in the Many-to-One Client-Server Model the server can be created on an appropriate cluster node at application startup or on an as-needed basis. If you are sure the server will be needed, it probably makes sense to start it at application startup and use a programming mechanism for interprocess communications to tell the server when there is work. If you are not likely to need the server, then consider creating it on an as-needed basis. Since process creation is expensive, keep a server process alive until it will no longer be needed.

  You can use DECnet-VAX communications or the Batch facility to create the remote server process. If the remote server's task is time-sensitive and you cannot wait for the Batch facility to schedule the task, then DECnet-VAX communication is the most effective technique for image activation on a remote VAXcluster CPU. Where the remote server's task is not time-sensitive, using the batch facility for remote process creation is preferable. In addition, when a client-server relationship is contained within your VAXcluster system, any client process can use process control system services to issue $WAKE and $HIBER system service calls to awake and hibernate the server process as needed.

- Data Sharing

  VMS RMS record locking must be coordinated so that any file or record locks held by any client process are released or converted to a compatible locking mode if the remote server process is to access the same file or record data. Depending on the application, you can also implement data sharing by using disk file Read-Only global sections. Also, when multiple clients are concurrently accessing a remote server process, a nontransparent DECnet-VAX logical link between the remote server and each of the clients can provide communication paths to exchange data.

- Process Synchronization

  You can use DECnet-VAX communications or lock management system services to establish your application's interprocess communication needs between clients and a remote server process.

  When using DECnet-VAX communications, each client process sends a request to the remote server and then waits for an AST to be delivered. After the remote server completes a client's work request, the remote server process executes a $QIO WRITE to that client process. The remote server process' $QIO WRITE may send the results back, or the remote server may put the results in a file and the $QIO WRITE will tell the client process where to find the results.

  Using lock management system services, each client process sends a work request to the remote server and takes out an EX lock with a blocking AST on a unique resource name. When the remote server completes the client's work request, the remote server requests an incompatible lock for the resource name that corresponds to the client process that made the work request. The appropriate client process then executes its blocking AST routine to get the results from the server.

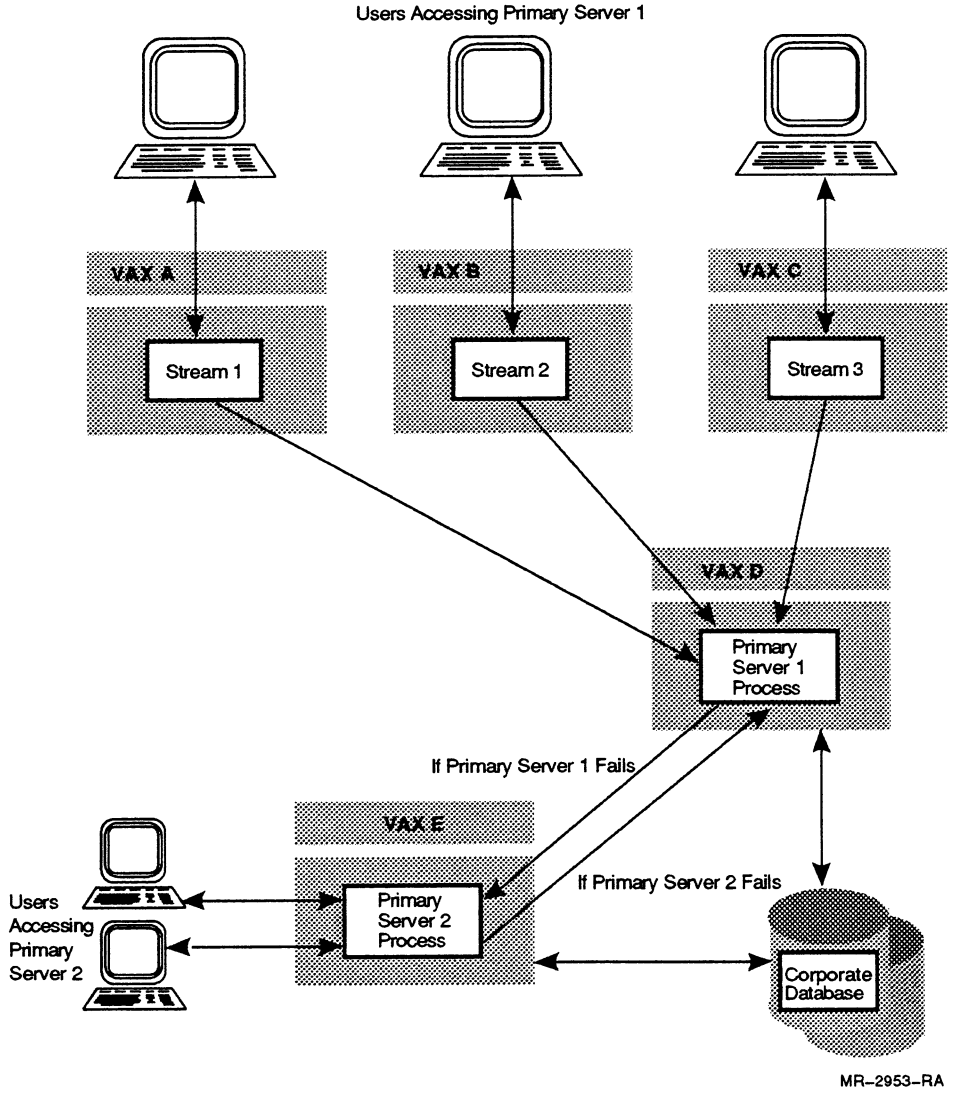
- Exception Conditions

  When each client process communicates to the remote server process, there should be a time limit for the connect wait period so that, if the remote server process dies or the server's VAXcluster CPU fails, a client process can recover. On the other side, there should be a recovery procedure so that the remote server process can detect and react to a VAXcluster CPU failure for a client process' node. Typically, the Many-to-One Client-Server Model is implemented with a backup server process that takes over all of the clients' requests when the primary server process fails.

Example of an Application Using the Many-to-One Client-Server Model

An example of the Many-to-One Client-Server Model (see Figure 4–9) is an electronic corporate phone directory implemented for clusterwide and network access to a database. The primary server process allows up to 15 clients to access the corporate database at one time. When a client requests access to the database to modify a record, the record is locked until the client completes the update. When the 16th client process requests access to the primary server, the primary server automatically forwards the 16th client's request to a backup server process working in tandem with the primary server. When the backup server becomes overloaded, requests are forwarded to the primary server which also acts as a backup server. Using this model, a series of database servers provide this application with high availability and high throughput.

Figure 4–9 Many-to-One Client-Server Model as a File Server for a Corporate Database



4.3 Parallelism Model

When you use the Parallelism Model, you can design an application to use multiple VAXcluster CPUs to simultaneously execute multiple instruction streams.[1] For more information on how to determine if your application is a candidate for distribution by decomposition, see Section 2.5.

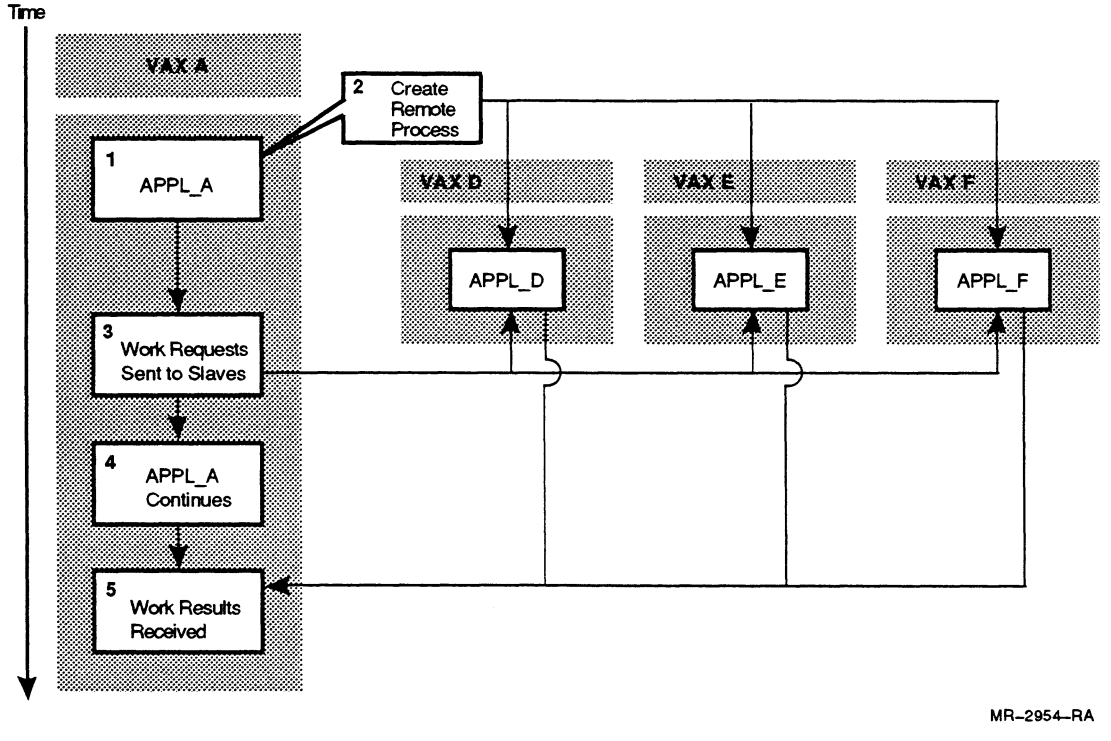
[1] The Parallelism Model is a model for application design structure and can be implemented on multiprocessor systems with or without shared memory. This manual describes the implementation of the Parallelism Model on a VAXcluster system, a multiprocessor system **without** shared memory.

Depending on the priority of your application, you can use the Parallelism Model to implement an application in either of two ways:

- Use dedicated VAXcluster CPUs to decrease your application's wall-clock time.
- Schedule the distributed application for the hours of the day that your VAXcluster system is at a minimum load. In this way, your parallel application can use idle VAXcluster CPU cycles on multiple VAXcluster CPUs.

Figure 4–10 illustrates a single master process that is directing work assigned to three remote slave processes. In this application, a single master (APPL_A) sends work requests to each of the slave processes (APPL_D, APPL_E, and APPL_F). The master and slaves all work in parallel. When necessary, the master process asynchronously manages a communications path to the appropriate slave process.

Figure 4–10 Parallelism Model



There are two common variations for designing a Parallelism Model:

- Self-scheduling
- Queueing

Self-Scheduling

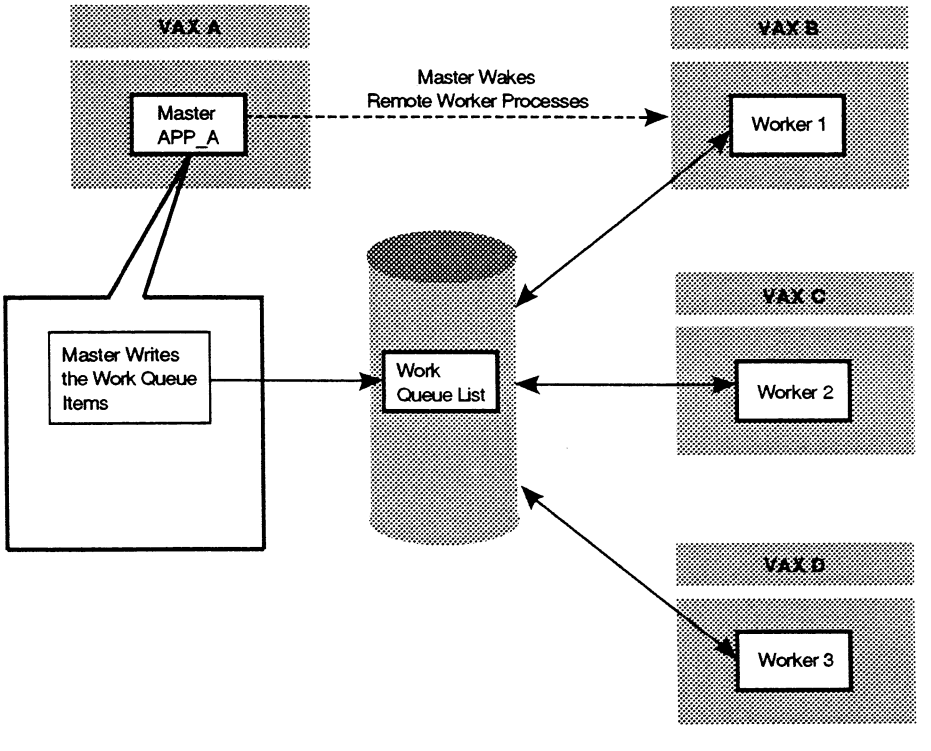
The first variation is called self-scheduling. You can design the master process to build a queue of work items and then create an appropriate number of remote slave processes to perform the task defined by each work item. The remote slave processes are assigned work by executing the task defined by the work item at the top of the master's work list. The remote slave processes continue to perform until there are no more tasks on the master's work list.

Queueing

The second variation is called queueing. Queueing is similar to the implementation of self-scheduling except that the master process builds a queue of work items, and when the remote slave processes execute, any additional items of work created by the remote slaves during execution are placed in the queue of work items. The remote slaves place the additional work items at the end of the queue and, when a work item is completed, the remote servers execute the next work item at the top of the queue.

Figure 4–11 demonstrates an implementation of the Parallelism Model that uses self-scheduling and queueing.

Figure 4–11 Parallelism Model Using Self-Scheduling and Queueing



In this application, the master queues work requests, and the workers self-schedule the execution of work requests from the work queue built by the master. The work queue built by the master resides in a disk file.

The work queue data structure has a counter to keep track of how many workers are accessing the work queue. Also, each work item on the work queue has two flags: one flag to indicate the item is being worked on, and one flag to indicate the item is completed.

The processing steps of this application are as follows:

1 Master process (APP_A) writes work requests to a queue.

2 APP_A uses PROCESS_SCAN to find available (hibernating) workers in the VAXcluster environment.

3 APP_A uses a $WAKE to wake the available workers.

4 When each worker is awakened, they look at a predetermined work queue.

5 As each worker accesses the work queue, they sign into a worker counter and look for the first work item without a flag.

6 When a worker locates an unflagged work item on the work queue, the worker signs out the work item by setting a flag for the appropriate work item and begins working.

7 When a worker completes a work item, the worker returns to the work queue and sets a second flag for the appropriate item on the work queue to indicate that this work item has been completed.

8 If a worker has produced any additional work items while processing its work item, the additional work is placed at the bottom of the work queue. Then, the worker signs out the next unflagged work item.

9 When the end of the work queue list is reached and there are no more work items, each worker finishes by removing itself from the list of active workers and putting itself back into hibernation.

10 After all of the workers have finished, the master process starts to prepare another work queue.

(See Programming Example 1 in Section 6.3.1 that demonstrates another possible implementation of self-scheduling and queueing.)

Advantages of Using the Parallelism Model

Using the Parallelism Model offers the following advantages:

- You can decrease a distributed application's wall-clock time by executing parallel tasks on multiple VAXcluster CPUs.
- You can use multiple remote slave processes as identical workers to perform the same functions in parallel, or use each remote slave as a specialized worker to perform a specific function.
- Where the amount of time needed to complete a task varies from work item to work item, you can implement a Parallelism Model with queueing and self scheduling.

Disadvantages of Using the Parallelism Model

Using the Parallelism Model has the following disadvantages:

- Because of increased interprocess communication, the application's performance may approach a limit where overhead becomes unacceptable.
- A design using interprocess communications between master and remote slave processes should include a plan for recovery from an exception condition.
- Using Parallelism with queueing, self-scheduling, and many workers can cause a bottleneck if all of the workers are awakened simultaneously. Depending on the way that work items are assigned, all the workers may attempt to concurrently access the work queue. This can cause contention as all workers attempt to access the same resource, and the work queue can become a bottleneck.
- It is more difficult to design, implement, and maintain a parallel application.

Implementation Requirements for the Parallelism Model

The Parallelism Model has the following implementation requirements:

- Remote Process Creation

  DECnet nontransparent communication lets you create a communication path which connects a master process to multiple logical links by assigning multiple I/O channels. The batch facility can also be useful for performing work from a remote slave process that is non-interactive and compatible with the batch mode. In addition, if all master-slave relationships for your Parallelism Model are contained within your VAXcluster system, a master process can use process control system services to issue $WAKE and $HIBER system service calls to awake and hibernate a slave process as needed.

- Data Sharing

  Depending upon the design of your application's master process, you must determine the appropriate means of message passing between master and slave processes. While a master process may use DECnet-VAX $QIO WRITES to transmit data to remote slave processes, this can cause timing problems. Typically, the master sends work to the slaves and remote slave processes do the work. A timing problem can arise when the master is waiting, synchronously, for the results from one of the specialized workers but another worker completes its work first. If the application permits the master to use only one of a number of remote slave process' results at a specific point in the master's instruction stream, consider using the event flag option with the $QIO WRITE from the remote slave processes. Then the master process can use the event flag to determine, asynchronously, what procedure to execute.

If the master or slave processes are concurrently accessing the same data, use VMS RMS record locking to coordinate access to the file.

You can also use a common VMS RMS file to pass messages between the master and slaves.

You can also map data common to the master and slave process on each VAXcluster CPU when you use a disk file Read-Only global section.

- Process Synchronization

  You can use DECnet-VAX communications or lock management system services to implement interprocess communication requirements.

  When you use nontransparent DECnet-VAX communications, the master process assigns work to each remote slave process over a different communications link and waits for a completion signal from any remote slave process. When a remote slave completes a work request, the slave performs a $QIO WRITE on a specific communications channel, and an AST is delivered to the master process. The master process can respond either synchronously or asynchronously to the remote slave process' $QIO WRITE. The remote slave may send the results back with a $QIO WRITE, or may put the results in a file with the $QIO WRITE telling the master process where to find the results. If the remote slave sends the results back, the master performs the appropriate procedure as determined by the evaluation of an event flag delivered with the AST for the remote slave's $QIO WRITE.

  To use lock management system services, each remote slave process is assigned a work item and the master process takes out an EX lock with a blocking AST, using a different resource name for each work item. When a remote slave completes its work item, the slave process requests an incompatible lock for the appropriate resource name. Then the master process executes the corresponding blocking AST routine to get the results. In this way, the master process uses the master's blocking AST routine to evaluate the completed work item's resource name and determine on which channel to issue a $QIO READ to get the results.

  When the Parallelism Model is implemented using queueing and self-scheduling, exclusion and barriers are used to affect process synchronization. Each worker must have exclusive access to the work queue to schedule work items, and all the workers must sign out of the work queue before control of the application returns to the master process.

- Exception Conditions

  When you implement *parallelism*, you must be sure to include a means to recover from exception conditions because the Parallelism Model requires more interprocess communication.

The master process must be able to detect DECnet-VAX error messages that are delivered with $QIO READs or WRITEs from the remote slave processes, and the master's event-flag-driven procedures must contain some recovery logic for when DECnet-VAX error conditions are detected. There must also be a recovery procedure in each remote slave to detect DECnet-VAX communications errors from a master's $QIO READ or WRITE to the remote slave's channel.

Example of an Application Using the Parallelism Model

A car manufacturer develops an application to test the structural integrity of the passenger compartment for one of their new cars. After a car is designed as an electronic model using computer aided design (CAD) software, the manufacturer uses an application to simulate a car crash.

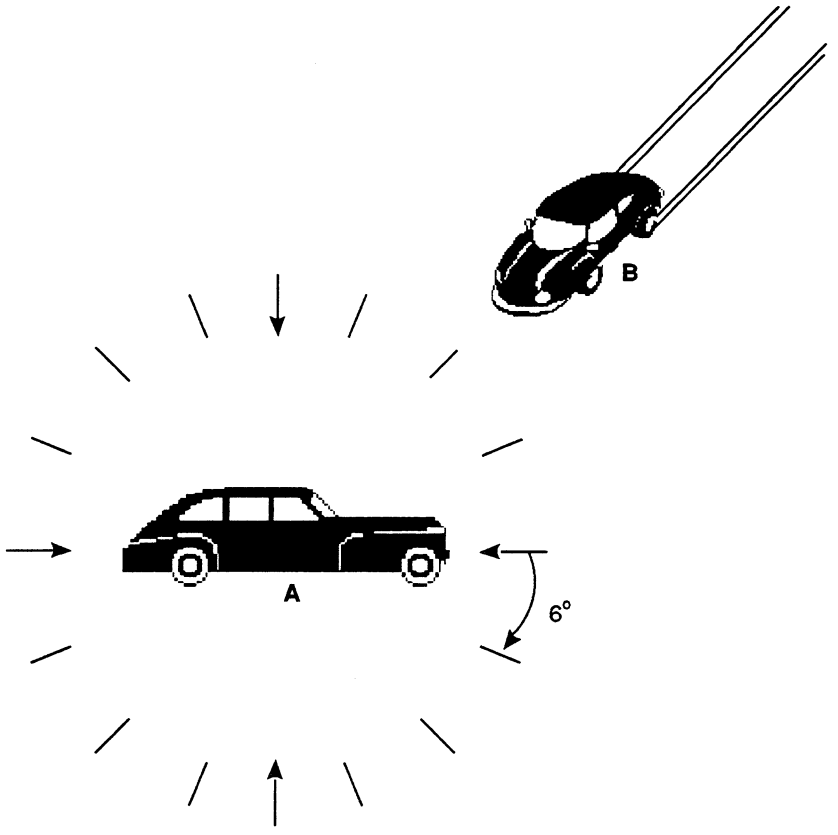
In the example shown in Figure 4–12, Car A and Car B are used to simulate a crash for different rates of speed and different angles of impact for Car B. In the test, 60 tests are run for the crash simulation. (Each test simulates crashes by varying the velocity of Car B from 1 to 70 mph in increments of 1 mph and the angle of impact at 6 degree intervals.) All 60 tests could be executed in parallel, but a master process is used to assign each test to a remote slave process executing on multiple VAXcluster CPUs. Each time the master process assigns a test to a remote slave process, the master process uses a lock value block to pass a message containing the angle of impact for the test that the slave is to execute. The master process maintains a table to track the progress of the simulation and if any of the slave processes fail, the master process can reassign the work item to another slave process.

The Parallelism Model is used to implement this car crash simulation application because:

- Each code segment is performed hundreds of times.
- Multiple tasks for a specific set of iterations can run in parallel.
- Each test component for this application can be run on any VAXcluster CPU available to this application.

By using multiple VAXcluster CPUs to run this car crash simulation, the car manufacturer can rapidly detect potential structural problems at the design stage and the wall-clock time for this application is reduced. Consequently, a car manufacturer can perform repeated car crash simulations until the car's structural design meets their requirements.

Figure 4–12 Car Crash Simulation



5 Designing Distributed Applications for a VAXcluster System

When you design applications for a VAXcluster system, you can distribute an application on multiple VAXcluster CPUs to achieve the following goals:

- Increased availability
- Faster completion
- Maximized throughput

These goals are discussed in Sections 5.1, 5.2, and 5.3. Section 5.4 contains a comparison of the application design models and application design goals. Section 5.5 and Section 5.6 provide an overview of some Digital products that you can use when designing an application for your VAXcluster system.

5.1 Designing an Application for Increased Availability

Designing an application for increased availability helps to ensure that you are nearly **always** able to run your application. You can achieve this availability by running any component of your application on any VAXcluster CPU.[1] This chapter presents three application design models that you can use to design an application for increased availability:

- File Sharing Model
- One-to-One Client-Server Model
- Many-to-One Client-Server Model

When implementing any one of these models, do not make a component of the application dependent upon a specific VAXcluster CPU. If a component is moved from its initial location to a different VAXcluster CPU, other application components must be able to continue to find it. You can use the VAX Distributed Name Service (DNS) to ensure that your application can find its various components anywhere in the cluster. For more information on DNS, see Section 5.6.5.

Using the File Sharing Model

The File Sharing Model (see Section 4.1) provides increased availability to users by distributing an application by replication on multiple VAXcluster CPUs. To use a File Sharing Model to increase application availability only requires a common-environment on your VAXcluster system. (For more information on configuring a common-environment for a VAXcluster system, see Section 1.1.2.)

[1] Assuming that your VAXcluster hardware is properly configured. For more information on configuring your VAXcluster hardware for high availability, see the *VMS VAXcluster Manual*.

To achieve an acceptable level of application performance when implementing the File Sharing Model, ensure that:

- The VAXcluster CPUs are not at maximum load capacity.[2]
- The I/O system in the VAXcluster system is not at maximum load capacity.
- There are multiple demands for the image.
- There is an even distribution of users on the VAXcluster CPUs where the image is replicated.
- The total number of application users does not increase.

For more information on how to determine if your application is a candidate for distribution by replication, see Section 2.4.

Using the One-to-One Client-Server Model

When using the One-to-One Client-Server Model to increase application availability, replicate the one-to-one client-server application on different VAXcluster CPUs. (If you use the same CPU for all instances of the application, this CPU may become a single point of failure.) In addition, you can also design an application based on the One-to-One Client-Server Model for increased availability by providing failover mechanisms for a backup client and backup server.

For further information on how to determine if your application is a candidate for distribution by decomposition using the One-to-One Client-Server Model, see Section 4.2.1.

Using the Many-to-One Client-Server Model

When your application executes on VAXcluster CPUs and non-clustered network nodes, you can use the Many-to-One Client-Server Model (see Section 4.2.2) to design for increased availability. You can design a file server to manage multiple client I/O requests from inside and outside your VAXcluster system. A file server can increase the availability of a VAXcluster file resource to client processes outside your VAXcluster system. It is more efficient for the server to access the database locally and perform the requested task, than to transmit the requested file contents to the client and have the client perform the file manipulation.

In addition to increasing the network availability of a VAXcluster resource, you can also increase the availability of the functions of a file server by designing a failover mechanism for a backup server. When using the Many-to-One Client-Server Model to design a file server to increase application availability, you should also:

- Restrict file access so that only the server process can access your file to protect file integrity
- Coordinate client requests from any other applications that are attempting to update a file simultaneously

[2] You may consider overconfiguring CPU resources for an application based on what CPU capacity might be required for the application to function in a degraded condition.

For further information on how to determine if your application is a candidate for distribution by decomposition using the Many-to-One Client-Server Model, see Section 4.2.2.

5.2 Designing an Application for Faster Completion of a Task

In Section 4.3, the Parallelism Model is presented as a means of distributing modules of an application for parallel execution. By decomposing your application into tasks that can be executed in parallel, you can complete the execution of your application more rapidly (decrease wall-clock time) because your application can simultaneously use multiple CPU resources.

Using the Parallelism Model

Designing an application for fast completion using Parallelism requires that:

- The application be decomposed into tasks.
- There is minimal contention between the tasks for shared resources.
- The tasks are relatively independent of each other so they can be executed in parallel.
- The amount of work performed by each task is large compared to the overhead associated with interprocess communications.
- There is enough potential gain in application performance to justify the additional effort and expense of implementing a distributed application.
- For faster execution time (decreased wall-clock time), tasks execute on dedicated hardware resources.

The performance of many paralleled applications is limited by the amount of time spent performing non-paralleled operations. Typically, non-paralleled operations have the following characteristics:

- Initialization activities
- Processing activities for which an algorithm for decomposition cannot be constructed
- I/O processing with high contention for a common resource

In general, an application that spends 50% of the time performing non-paralleled operations could achieve a maximum of 1.6 times performance improvement if the paralleled part was decomposed into four tasks that could execute in parallel on four VAXcluster CPUs.

The mathematical equation that defines this improvement in a paralleled application is Amdahl's Law. It states that the maximum performance gain from parallelism is calculated by dividing the percentage of the application that can be run in parallel by the number of processors, and adding the percent that cannot be run in parallel. In the preceding example, this would be: (50% of paralleled time divided by 4) + 50% non-paralleled time). The best that Amdahl's Law says this application can achieve from parallelism is running in 62.5% of its original runtime or 1.6 times faster.

For further information on how to determine if your application is a candidate for distribution by decomposition using the Parallelism Model, see Section 2.5 and Section 4.3.

5.3 Designing an Application for Maximum Throughput

You can use all the models described in Chapter 5 to maximize throughput, the total work done. All these models let you maximize application throughput because multiple VAXcluster CPUs are used in the implementation. When designing an application for maximum throughput, consider which method of distribution is the most appropriate use of multiple CPUs:

- Distribution by replication
- Distribution by decomposition

Whether you distribute by replication or decomposition, you can design an application to maximize throughput for either:

- A VAXcluster system running multiple applications

  Each instance of an application competes with other application images and all other processes on a specific VAXcluster CPU.

- A VAXcluster system running a single application

  When a VAXcluster system is dedicated to a single application, you should design for maximum throughput to fully utilize your VAXcluster CPU resources.

A VAXcluster System Running Multiple Applications

If you have multiple small applications contending for the same resources, when distributing by replication, you can maximize throughput by:

- Designing all applications to run on any CPU, with each image running entirely on a single CPU; this avoids the need for communication between components
- Minimizing contention for shared resources
- Balancing the workload throughout the cluster

If you have multiple applications contending for the same resources, when distributing by decomposition, you can maximize throughput by:

- Scheduling the jobs at times of low-activity to use idle CPU cycles
- Minimizing contention for read access to shared resources by using disk file Read-Only global sections

A VAXcluster System Running a Single Application

If your cluster is dedicated to a single application and you are distributing by replication, you can maximize throughput by:

- Running a copy of the same application on each CPU in the cluster
- Balancing the workload evenly
- Designing your application to use RMS global buffers, depending on the characteristics of file access requests (For more information on how to use RMS global buffers, refer to Section 3.2.3 in this manual and the *Guide to VMS File Applications*.)

  For example, for a transaction processing application, you can:

  — Use terminals connected to a terminal server on the Ethernet to distribute users.

  — Use a shared VMS RMS, DBMS, or Rdb/VMS database.

  — Distribute the file or database components over several disks to avoid an I/O bottleneck.

  — Open all the files at the start of the application, rather than opening and closing each file for every access request. Use this strategy when users need to access several different files at different parts of the application.

  — Dedicate different VAXcluster CPUs to run terminal handling procedures and the transaction processing code.

If your cluster is dedicated to a single application and you are distributing by decomposition, you can maximize throughput by:

- Decomposing application tasks into the largest possible work items
- Designing tasks that minimize contention for shared resources
- Reducing the need for interprocess communication
- Balancing the cluster work load evenly across all VAXcluster CPUs
- Designing a master process to monitor the progress of each of its workers and reallocate task assignment if there is an exceptional delay for the completion of an assigned unit of work

5.4 Comparison of Application Design Models with Application Design Goals

Table 5–1 relates and compares the application design models described in Chapter 4 with the application design goals discussed in this chapter. In addition, Table 5–1 also compares the models with respect to developmental complexity and network-wide access.

Table 5–1 Comparison of Application Design Models with Application Design Goals

Design Model	Increased Availability	Fast Completion of Task	Maximize Through-put	Developmental Complexity	Network-Wide Access
File Sharing	High	Low	Yes[1]	Low	Sometimes[2]
One-to-One Client-Server	Medium[3]	Medium[4]	Yes	Medium[5]	Yes
Many-to-One Client-Server	Medium[5]	Medium[4]	Yes	Medium[5]	Yes
Parallelism	Low[6]	High	Yes[7]	High[8]	Sometimes[9,10]

[1]Assuming that your application is not I/O bound and there is minimum contention for the same data elements.

[2]Network access is possible when using Digital's distributed file service (DFS). For more information on DFS, see the associated documentation for *VAX Distributed File Service*.

[3]You can increase availability by replicating the one-to-one client-server application on different VAXcluster CPUs. However, if one of the replications fails, that instance of the application only remains available if there is a backup server.

[4]When the server is a dedicated hardware resource, the application can execute faster.

[5]To ensure availability, implement this model with a backup server. This requires explicit interprocess communications.

[6]Using programming techniques for VAXcluster failover, you can decrease the effect of a VAXcluster CPU failure. For more information on VAXcluster failover mechanisms, see Section 6.4 and Chapter 8.

[7]You can maximize throughput by either running the parallel application on dedicated VAXcluster CPUs, or you can run the parallel application during non-peak hours to maximize otherwise idle CPU cycles.

[8]Providing process synchronization for distributed application tasks requires explicit interprocess communications.

[9]If explicit interprocess communications are implemented using DECnet-VAX communications, then network-wide access is possible.

[10]If explicit interprocess communications are implemented using lock management system services, then the application is distributable only within the VAXcluster environment.

5.5 Designing a VAXcluster Application Using Products Closely Associated with the VMS Operating System

In your application design, you can use the following products that are closely associated with the VMS operating system:[3]

- VAX RMS Journaling
- VAX Volume Shadowing
- VMS DECwindows

Sections 5.5.1, 5.5.2, and 5.5.3 briefly describe these products and their features when used on a VAXcluster system, and provide references to the appropriate documentation.

5.5.1 VAX RMS Journaling

VAX RMS Journaling provides data integrity and data consistency to data-intensive applications that do not require the features of a full database management system, such as VAX DBMS or VAX Rdb/VMS. VAX RMS Journaling is supported to disk only; VAX RMS Journaling does not journal to tape. VAX RMS Journaling provides three different types of *journaling*:

- *After-image journaling*

  Provides the ability to redo a series of modifications to a data file. This type of journaling lets a file be recovered if it is lost or corrupted, for example, if the file is inadvertently deleted or the data is lost due to a disk head crash. Such a recovery restores the contents of the file from the point of the latest BACKUP copy of that file. To restore data with after-image recovery, use the RMS Recovery Utility with the DCL command RECOVER/RMS_FILE/FORWARD, and specify the backup copy of the data file as the parameter to the command.

- *Before-image journaling*

  Provides the ability to undo a series of modifications to a data file. This type of journaling lets a file be returned to a previous known state. This is useful if a file is updated with bad or erroneous data from, for example, a noisy data communications line or operator error. To restore data with before-image recovery, use the RMS Recovery Utility with the DCL command RECOVER/RMS_FILE/BACKWARD, and specify the data file as the parameter to the command.

- *Recovery unit journaling*

  Provides transaction integrity. A "transaction" is a series of VMS RMS record operations on one or more files that are viewed as an atomic operation; that is, either all or none of the operations are completed.

[3] VAX RMS Journaling and VAX Volume Shadowing are system integrated products that require a separate license and are installed separately from VMS; VMS DECwindows is installed separately from VMS Version 5.1 or higher, but does not require a separate license.

Journaling is applied on a file-by-file basis. A file can be marked for after-image, before-image, or recovery unit journaling, or any combination of these. Within a given application, you can use any combination of journaling types.

VAX RMS Journaling stores the information necessary for data recovery in files called "journal files." Multiple files can journal to the same journal file.

VAXcluster Features

Once after-image or before-image journaling has been turned on for a given file, every WRITE access to the file from any process within a VAXcluster system will be journaled. If an attempt to open a file for write access is made from a VAXcluster node with VAX RMS Journaling not installed, the open will fail.

When recovery unit journaling is used with a VAXcluster application that is executing on multiple VAXcluster CPUs, recovery unit journaling can segment the application into groups of file modification operations which logically belong together. Each such group is called a recovery unit. The database being maintained by the application is considered in an invalid state if some, but not all, of the modifying operations within a recovery unit have been completed. If an application fails to complete a recovery unit, the database is rolled back to its state prior to the beginning of that recovery unit. This provides two types of clusterwide protection:

- Protects records modified from one recovery unit from access by other recovery units until the whole set of modifications is complete
- Protects a database from an incomplete set of updates because of hardware or software failure

In both conditions, recovery from a recovery-unit failure is automatic. Recovery-unit recovery is performed in the context of any process that is executing the image because the application is programmed to include calls to the $START_RU and $END_RU system services. (In addition, a $SET FILE/RU_JOURNAL must be issued to all data files to be protected by the recovery units.) These calls declare the beginning and end points of the recovery units. In the case of a VAXcluster CPU failure or process deletion, the recovery-unit rollback may be initiated by a process on a different VAXcluster CPU. This occurs only if another process exists that is currently accessing the same recovery unit journaled data files. If a single VMS system or all VAXcluster CPUs accessing the databases fail, recovery-unit recovery is initiated by the next process to open the recovery unit journaled data files.

VAX RMS Journaling is a product that complements VAX ACMS (see Section 5.6.2) by increasing the efficiency of VAX ACMS in a VAXcluster system. VAX ACMS enables the programmer to set up a server that controls all data file updates and I/O operations. In a VAXcluster system, rather than having each process act as a recovery unit, the server performs all update transactions. Instead of having a recovery unit journal file for each process, there is a single recovery unit journal file for the server. This prevents the recovery unit journal disk from becoming a bottleneck.

You can use VAX RMS Journaling with VAX Volume Shadowing and VAX ACMS to provide data integrity, data availability, and processing efficiency benefits. For more information on VAX RMS Journaling, see the *VAX RMS Journaling Manual*.

5.5.2 VAX Volume Shadowing

VAX Volume Shadowing is a feature available on VMS systems using HSC controllers with RA-series disks. This product enhances data availability by duplicating all data written to disk onto two or three compatible disk volumes. (Compatible disks are those that have the same physical geometry.) You can shadow any system or data disks.[4]

A set of two or three disks on which the data is duplicated is called a shadow set. Shadow sets can be constituents of a bound volume set. A disk which is part of a shadow set is called a "member." Every member of a shadow set is identical to every other member of the shadow set. In the event of failure of any shadow set member, disk I/O continues with the remaining member or members of the shadow set. Since it does not provide for recovery from accidental file deletion, volume shadowing is not a substitute for regular BACKUP operations.

The minimum number of disks in a shadow set is one. The maximum number of disks in a shadow set is three. The ability to have a shadow set with only a single member allows a 2-member shadow set to continue operation after the failure of one member. VAX Volume Shadowing also allows normal operation of a single-member shadow set and the addition of a second member during a volume backup procedure.

VAXcluster Features

The shadow set appears to act as a single disk, that is, the user need not take any actions in the application code to have data propagated to all members of a shadow set. VAX Volume Shadowing replicates the data on all shadow-set members.

The syntax and semantics of reading and writing data to and from a shadow set is identical to the syntax and semantics used for non-shadowed I/O operations. Thus, all commands and program language features which address data on non-shadowed disks can be used to address data on shadowed disks.

Enabling and disabling shadowing for a particular disk requires, logically, dismounting and remounting the disk. Disks can be added to, or removed from, the shadow set by a system manager with minimal effect to either the user or any application.

Shadow-set members can be dual-ported between two HSC controllers; that is, each disk may be connected to each of the two controllers. (See Section 1.1.1 in this manual for a discussion of automatic shadow set failover when a shadow-set is dual-ported between two HSC controllers.)

[4] If a disk does not conform to the FILES-11 ODS2 standard, see the *VAX Volume Shadowing Manual* for information on how to include a "foreign" disk in a shadow set.

Note: In a VAXcluster system, you cannot shadow the quorum disk. The maximum number of disks per HSC (or per HSC pair in the case of dual-ported disks) that may be shadowed is 16. The maximum number of shadow sets per HSC (or per HSC pair in the case of dual-ported disks) is 8.

Also, you can use VAX Volume Shadowing with VAX RMS after-image journaling (see Section 5.5.1); however, it does not completely replace after-image journaling. After-image journaling provides the following ways to recover data which VAX Volume Shadowing does not address:

- Mistaken deletion of a file by a system user or operator
- Corruption of file system pointers
- VMS RMS file corruption from a software error or incomplete bucket writes to an indexed file

VAX Volume Shadowing can provide increased data availability for your application; in addition, it can help to maximize throughput for your application. For more information on VAX Volume Shadowing, see the *VAX Volume Shadowing Manual*.

5.5.3 VMS DECwindows

VMS DECwindows is an advanced windowing system supported by VMS for Version 5.1 or higher. VMS DECwindows provides a windowing environment with the following features:

- Graphics-oriented interaction with the VMS operating system

  DECwindows handles all user interactions with the system. These interactions include initializing a session, managing your environment and resources, starting applications, and managing windows. You can create windows, move windows, resize windows, and shrink windows to icons.

- Consistent user interfaces

  DECwindows user interfaces consist of graphic objects that look and function the same, regardless of the application you are using. This makes it easy to learn about and use new applications.

- A library of desktop applications

  DECwindows provides a variety of applications, including: Bookreader, Calculator, Calendar, Cardfiler, Clock, DDIF Document Viewer, EVE Text Editor, FileView, Mail, Notepad Text Editor, Paint Graphics Editor, Postscript Document Viewer, and DECterm.

- Powerful libraries of routines that simplify the development of graphics-oriented applications

  These routines perform functions such as drawing and window management. Applications using these routines run on all supported hardware without modifications.

- Libraries for creating and accessing documents that contain text, graphics, and scanned images
- Network-transparent application interface

  The DECwindows architecture lets you run an application on a remote node, while displaying and keying in data on a local workstation. The network functions that make this possible are transparent; the application appears to run locally.

  Network transparency allows workstations to access the power and resources of other systems on the network.
- Extensible architecture

  The DECwindows architecture has been designed to accommodate new technology, such as 3-dimensional graphics, as it becomes available.

The DECwindows programming environment makes it easy to develop applications with simple, consistent, graphics-oriented interfaces. The core of the graphics programming environment consists of the Xlib and XUI Toolkit programming libraries. Xlib consists of low-level routines for performing basic graphic and windowing functions. The XUI Toolkit consists of high-level routines for creating and managing user-interface objects like menus, scroll bars, and buttons. Applications written in any programming language can call routines from both libraries.

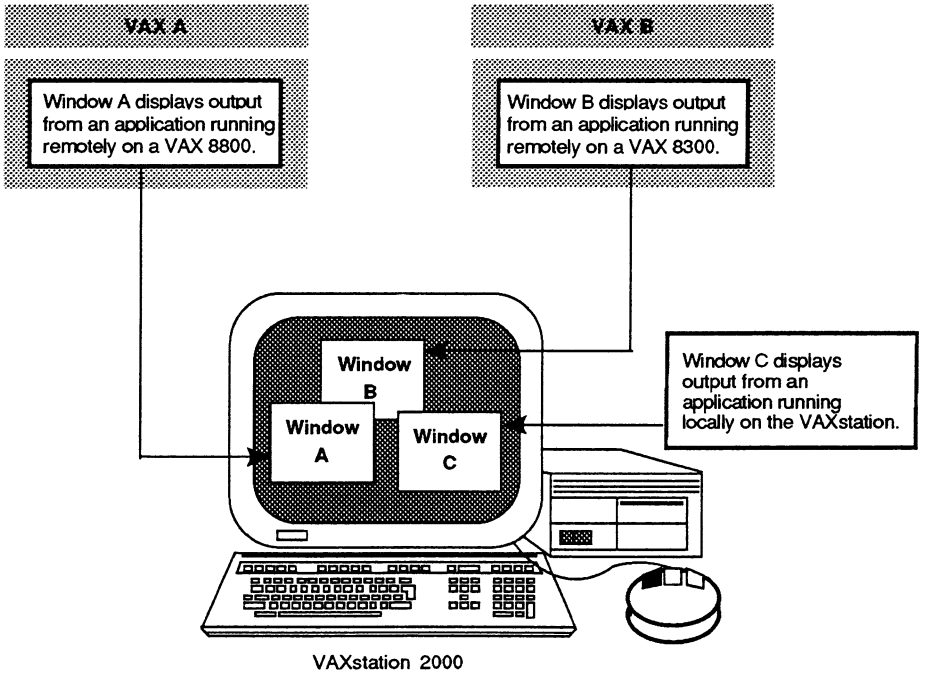
XUI Toolkit routines save you time because they simplify the task of creating a user interface. For example, you can create a menu with one call to an XUI Toolkit routine. Creating the same menu using Xlib routines requires many more calls and program lines. Using XUI Toolkit routines also ensures that an application interface conforms to the XUI style. Applications that conform to the XUI style are easy to learn and use.

The XUI Toolkit includes additional tools that further simplify the process: the User Interface Language (UIL) Compiler and the XUI Resource Manager (DRM) routines. In essence, UIL and DRM let you create the entire interface with one library call. UIL and DRM also let you separate form and function in an application. You can store the user interface in a file that is independent of the application, and modify the file without recompiling the rest of the application.

VAXcluster Features

Using VMS DECwindows support of applications based on the Client-Server Model (see Section 4.2), you can design the server to support asynchronous input from the user to the application and asynchronous output from the application to a display. Figure 5–1 illustrates the DECwindows platform for simultaneous sessions that are using different VAXcluster CPUs as compute servers. Consequently, you can design your application to use the most powerful VAXcluster CPUs as compute servers to provide faster task completion. For more information on VMS DECwindows, refer to the VMS DECwindows documentation set.

Figure 5–1 Running VMS DECwindows in a VAXcluster System



5.6 Designing a VAXcluster Application Using Layered Products Based on the VMS Operating System

In your application design, you can use the following layered products that are based on the VMS operating system:

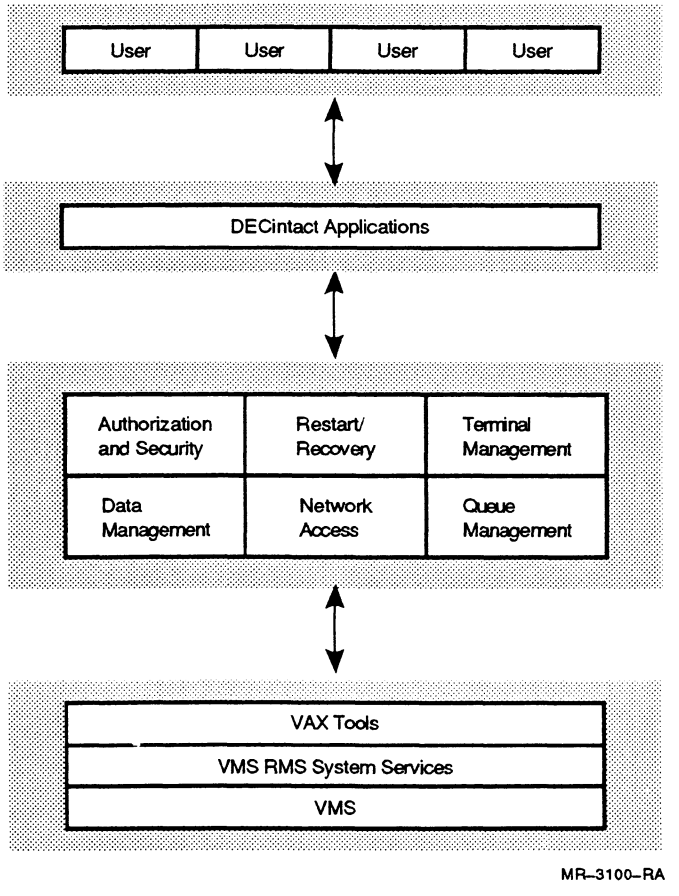
- DECintact
- VAX ACMS
- VAX DBMS
- VAX Rdb/VMS
- VAX DNS

Sections 5.6.1 through 5.6.5 briefly describe these products and their features when used on a VAXcluster system, and provide references to the appropriate documentation for more product information.

5.6.1 DECintact

DECintact (Integrated Application Control System) provides a foundation for building simple and complex transaction processing applications on one or more VAX systems under the VMS operating system. DECintact runs as a collection of services and processes under the VMS operating system supporting either single-threaded (per-process) or multi-threaded (server) application program design methodologies. Multiple DECintact applications can execute at the same time, sharing physical memory on the same VAXcluster CPU, and be completely independent. Figure 5–2 illustrates the major components of DECintact.

Figure 5–2 The Components of DECintact



- Authorization and Security

  DECintact provides an extensive security and menu system for establishing security profiles of local and remote users.

- Restart and Recovery

  Each record operation is considered as an individual unit for the purposes of restart or recovery. Two separate recovery strategies, roll forward and roll backward, are available. They may be combined to ensure maximum reliability. Roll backward recoveries are supported on line with no interruption or loss of service.

- Terminal Management

  DECintact supports VT100-, VT200- and VT300- (in VT200 emulation mode) series terminals as though they were intelligent block-mode terminals. Programmers create screen forms interactively through the Terminal Forms Editor (TFE).

- Data Management

  DECintact uses VMS RMS for file and record access. The DECintact Data Management component enhances the use of relative files by means of implicit file openings, file sharing, and logically deleted records. DECintact also supplies its own *hash file* access support which provides a high performance method of inserting and retrieving records.

- Network Access

  DECintact supports explicit and implicit remote access at the menu item level and uses DECnet-VAX facilities for intersystem communication. Network access is supported at two levels: peer-to-peer and front-end-to-host. Peer-to-peer access allows users with sufficient entitlement to invoke applications remotely on another participating DECintact system. Front-end-to-host access provides a transparent method of off-loading forms management (including both built-in and user-written validation) and menu level security from a host onto a VAX front-end system. A VAX front-end system also offers automatic host failure rollover in the event that a CPU within a host VAXcluster system fails.

- Queue Management

  DECintact provides a comprehensive set of routines that support the creation of disk and memory-based queues. By using recoverable, distributed queueing, transactions can be designed to progress through various states by passing data or functionality forward into a queue. Queues can lead anywhere; for example, to a database or, on a networked system, to another CPU. Because queues are reliable, if a transaction has made it to a certain queue, the transaction was successful in updating the database or file structure.

VAXcluster Features

All DECintact file accesses, as well as all disk-based queue operations, utilize the distributed VMS lock manager to arbitrate clusterwide system resources. DECintact support of restart and recovery strategies can be used to simplify the implementation of a high availability application on a VAXcluster system. DECintact allows one or more VAXcluster nodes to attach to a given terminal. One of the nodes actually succeeds in its allocation of the terminal while the others are placed in a queue. In the event that the initial node fails, one of the surviving VAXcluster CPUs is given access to the terminal, and the user can restart the application on the node where their terminal has been reconnected. At this point, the user can re-enter data for the transaction.

In addition, if a VAXcluster system is running a common DECintact application, the DECintact application can perform an automatic rollback of incomplete transactions when a VAXcluster CPU fails. Automatic rollback provides data integrity by backing out partially completed transactions to return the databases of a failed node to a consistent state. Once the DECintact application backs out the incomplete transactions, the surviving CPUs in a VAXcluster system can continue to process transactions without fearing that the database is in an inconsistent state. By virtue of terminal failover, the user then can re-enter, on a running CPU, any incomplete transactions that had started on a failed node.

DECintact lets you design an Online Transaction Processing (OLTP) application for high availability and to ensure data integrity. With DECintact, you can set up an OLTP environment that recovers from application and VAXcluster CPU failures. For more information on designing an application using DECintact, see the DECintact documentation set.

5.6.2 VAX ACMS

VAX ACMS (Application Control and Management System) is a TP (Transaction Processing) monitor providing three different environments:

- Run-time environment for executing VAX ACMS-developed applications
- Development environment in which to write VAX ACMS applications
- Application control and management environment for overseeing the run-time environment

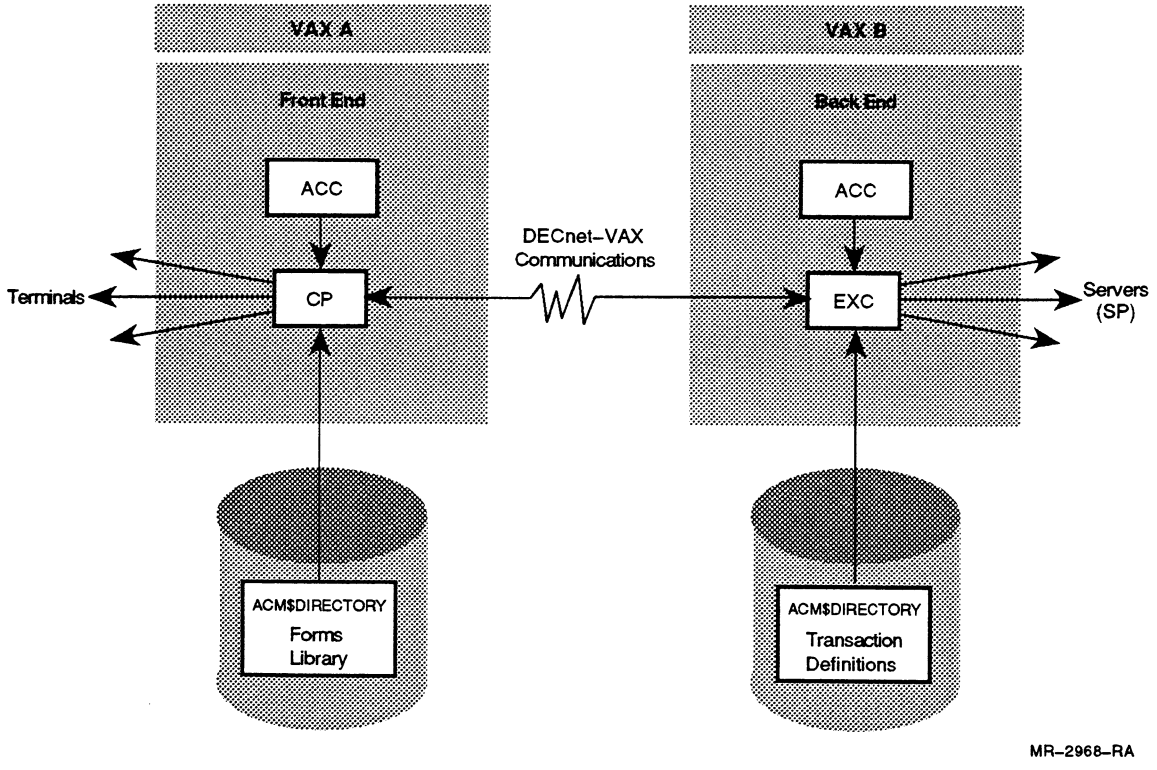
TP applications are typically applications with many users doing predefined tasks against shared data. VAX ACMS has been used in such transaction processing applications as order processing and inventory control, Materials Requirement Planning (MRP) and shop floor control, financial services applications, as well as, customer and administrative systems across a wide variety of industries.

VAX ACMS was designed to address the following specific needs of TP applications:

- Make more efficient use of system resources with specialized servers
- Provide flexibility in how users access data
- Improve application availability, particularly on a VAXcluster or configurations with separate front-end processors
- Provide an integrated application development environment with an underlying data dictionary (VAX CDD) sharing data among VAX TDMS forms, VAX Rdb/VMS databases, and VAX DBMS databases
- Provide a modular development style and a utility to build applications, task groups, and menus
- Provide utilities to manage and control these complex applications

Figure 5–3 illustrates how the VAX ACMS runtime system can include a front-end requestor for terminal I/O and a back-end server controller for database processing.

Figure 5–3 Running VAX ACMS on Two VAXcluster CPUs



The VAX ACMS runtime system processes include:

- ACMS Central Controller (ACC)

  The ACMS central controller (ACC) runs on each VAX processor where VAX ACMS is installed and in use. Its primary purpose is to start up and monitor the front- and back-end processes that perform application work.

- Command Process (CP)

  The command process (CP) is a multi-threaded process that handles terminal I/O for many users at the same time. ACMS users log in to the VMS operating system through the CP and therefore do not require their own processes. The CP is the front end of VAX ACMS and can run either on the same processor as the back end or on a separate processor.

- Execution Controller (EXC)

  The execution controller (EXC) is a multi-threaded process that communicates with the CP to handle computation and database I/O for ACMS users. The EXC is the back end of VAX ACMS and controls servers for many CPs. The EXC can run either on the same VAX processor as the CP or on a separate processor.

- Server Processes (SP)

  VAX ACMS servers are serially reusable under EXC control and handle database access for many users simultaneously, which reduces database contention and record-locking overhead. Because ACMS servers are single-threaded, code problems are isolated to a server and do not affect the rest of the system.

For a major transaction processing (TP) application, VAX ACMS has considerable advantages over standard programming methods under VMS because fewer VMS processes are required to support a given number of users. The VAX ACMS approach has the following advantages:

- A single process services many terminal users, thus conserving computer resources.
- Server processes can be started (performing initialization activities like opening files or databases only once) and then left dormant, available to service requests from any of the terminal users.
- A database can be accessed by a limited number of server processes, thus avoiding the need for complex and high-overhead locking.
- Server processes can be started and stopped cleanly, thereby effectively controlling the corresponding processing steps. As a result, it is possible to shut down part of an application without affecting the users.
- VAX ACMS supports online application modification; a modified version of an application can be brought up without affecting the users.

VAXcluster Features

You can use VAX ACMS on a VAXcluster system to design a distributed application. VAX ACMS applications can improve throughput in terminal I/O-intensive applications by off-loading the terminal control and forms processing to a separate VAXcluster CPU. Also, VAX ACMS lets a user on one VAXcluster CPU select a task for execution and that task can be designed to execute on another VAXcluster CPU. In addition, you can change your application configuration without reprogramming by using logical names or by changing an ACMS definition.

Applications developed using VAX ACMS can be used in:

- A VAXcluster environment with terminal users distributed across the VAXcluster system
- An off-loaded environment, where terminal users and resulting forms processing are off-loaded to a front-end processor from the back-end processor within a local-area or wide-area network (The front-end processor could be a single node or a VAXcluster CPU.)

VAX ACMS can increase application availability by allowing applications to fail over and *fail back* in a VAXcluster environment. If the VAXcluster CPU where users are working fails, they can continue on an alternate VAXcluster CPU without having to log into VAX ACMS again. VAX ACMS uses VMS search lists to specify primary and alternate VAXcluster CPUs for each application. When the original VAXcluster CPU becomes available again, the original system can again be specified as the primary one without stopping the VAX ACMS application.

VAX ACMS lets you design an application for increased availability to eliminate any single point of hardware failure that will result in application downtime. Also, VAX ACMS works equally well in any VAXcluster configuration, allowing you to pick the VAXcluster configuration most appropriate for your application requirements, whether those requirements are for increased capacity, increased throughput, higher availability, or increased database performance. For more information on VAX ACMS, see the VAX ACMS documentation set.

5.6.3 VAX DBMS

VAX DBMS (Database Management System) is a multi-user, general-purpose CODASYL-compliant database management system that runs under the VMS operating system. VAX DBMS is used to access and administer databases ranging in complexity from simple hierarchies to complex networks with multi-level relationships.

VAX DBMS supports full concurrent access in a multi-user environment without compromising the integrity and security of the user's databases. Some of the features offered by DBMS are:

- Full concurrent access capabilities (storage, retrieval, update, and deletion) in a multi-user environment
- Record locking and journaling
- Automatic transaction rollback

- Multiple database support (one or more databases per process)
- Integrated with VAX Common Data Dictionary/Plus (CDD/Plus) facility
- Schema, Subschema, Storage Schema, and Security Schema Data Definition Languages (DDLs)
- FORTRAN Data Manipulation Language (FDML)
- COBOL DML statements supported by the VAX COBOL compiler
- Generic DML preprocessor for BASIC, BLISS, C, DIBOL, PASCAL, PL/I, and VAX Ada
- Callable interpretive interface for any VAX language that adheres to the VAX calling standard
- VAX DATATRIEVE, an optional interface to the VAX DBMS database, provides a nonprocedural query and report generating facility
- DECnet-VAX database access to provide full remote read and write access to non-redundant distributed databases
- Standard VMS file security (SYSTEM, OWNER, GROUP, WORLD) to protect database storage areas (An application program can only access records defined in its SUBSCHEMA.)

VAXcluster Features

In a VAXcluster environment, VAX DBMS allows concurrent, multiple-node database access. VAX DBMS automatically recovers the database if a processor in the VAXcluster fails and provides optional after-image journaling to further protect the integrity of the VAXcluster database.

VAX DBMS uses the distributed VMS lock manager to synchronize clusterwide updates to the database root file, to initiate the automatic recovery process when a node fails, and to coordinate concurrent updates to the same database from processes running on different nodes.

In a properly configured VAXcluster environment, VAX DBMS lets you design an application for increased availability. Each VAXcluster CPU that accesses the database has a process, called the DBMS monitor, that is started at system startup time. The monitor's major responsibility is detecting failures and initiating the necessary recovery processes on behalf of failed, incomplete transactions.

If a VAXcluster CPU fails, the automatic recovery procedure of the DBMS monitor lets database access continue uninterrupted. VAX DBMS transparently recovers a failed CPU's outstanding database activity on one of the remaining CPUs. Also, when a process on a VAXcluster CPU begins to access an existing database, the database monitor processes on each VAXcluster CPU establish communication with each other using the distributed VMS lock manager. By providing these mechanisms for restart and recovery, VAX DBMS ensures complete data consistency and integrity for all the database files residing on clusterwide disks. For more information on VAX DBMS, see the VAX DBMS documentation set.

5.6.4 VAX Rdb/VMS

VAX Rdb/VMS (Relational Database Management System) is a full-function relational database management system designed for the VMS operating system. It is intended for general purpose, multi-user, centralized, or distributed applications.

VAX Rdb/VMS supports a complete set of utilities and precompilers that enable users to maintain and manipulate databases. VAX Rdb/VMS includes a VAX SQL component, Digital's implementation of the Structured Query Language, an ANSI standard interface to relational database products.

VAX Rdb/VMS provides the following features for concurrency:

- Full concurrent access (storage, retrieval, update, and deletion) in a multi-user environment
- Concurrent access to the same database by multiple applications
- Optional read-only (snapshot) mode for increased concurrency in large retrieval and report writing applications. When a read-only transaction is started, the operations do not lock out other users.

VAX Rdb/VMS provides the following features for security:

- VMS file protection for database files
- A set of Access Control Lists (ACLs), associated with entities in the database, providing rights to perform database operations

VAXcluster Features

Application programs developed to access VAX Rdb/VMS databases and which run under VMS on a given node on a VAXcluster system or in a DECnet network can:

- Access Digital Standard Relational Interface (DSRI) databases on the same node
- Access DSRI databases on other nodes in the network

VAXclusters offer higher availability to VAX Rdb/VMS databases in a properly configured VAXcluster environment. Using VAX Rdb/VMS in a VAXcluster environment allows concurrent, multiple-processor database access. VAX Rdb/VMS automatically recovers the database if a processor in the VAXcluster fails, and provides optional after-image journaling to further protect the integrity of the VAXcluster database.

VAX Rdb/VMS uses the distributed VMS lock manager to synchronize clusterwide updates to the database root file, to initiate the automatic recovery process when a node fails, and to coordinate concurrent updates to the same database from processes running on different nodes.

If a VAXcluster CPU fails, the automatic recovery procedure in a VAXcluster for VAX Rdb/VMS allows database access to continue uninterrupted. When a process on a VAXcluster CPU begins to access an existing database, the database monitor processes on each VAXcluster CPU establish communication with each other using the distributed VMS lock manager.

In addition, the Data Base Administrator (DBA) has complete control over the placement of database files, called storage areas, on multiple disks. A DBA can use the VAX Rdb/VMS database structuring capabilities to significantly reduce I/O bottlenecks by mapping storage areas to specific disks at the physical design stage.

For more information on VAX Rdb/VMS, refer to the VAX Rdb/VMS documentation set.

5.6.5 VAX DNS

VAX Distributed Name Service (DNS) provides selected Digital products with a DECnet-wide name-to-attribute mapping service. DNS presents and maintains a consistent, network-wide set of names for network resources, also known as objects. Objects can be files, disks, CPUs, queues, mailboxes, and so forth. The object names are constructed without including any location information in them, thus permitting users to reference these network resources independent of their physical location. Because VAX DNS names are location independent, users and applications do not need to know on which CPU an object resides. DNS translates an object's name to a set of attributes; its network address is one of these attributes.

VAX DNS is based on a client-server design, in which two cooperating components of software, the client and the server, work together to make the service function. Install VAX DNS on a DNS server CPU. (A DNS server is any CPU in your network on which DNS object names are stored, and the client is any application needing to access the DNS namespace.) You can have as many DNS servers in your network as you want. Applications that are designed to work with VAX DNS use DNS names to name their objects. For example, DNS is a prerequisite for the following applications:

- Remote System Manager (RSM) V2.0
- VAX Distributed File Service (DFS)

DFS uses DNS names to name its file access points, and RSM uses DNS names to name its client nodes. Over time, additional Digital applications will use DNS to name their objects.

VAXcluster Features

If an application executing on your VAXcluster system is also part of a network application, VAX DNS can provide the following features:

- A network-wide name-to-attribute mapping service which allows selected Digital applications to create, read, modify, and delete names in the namespace.
- Ability to store and manage a large number of names using a hierarchical structure.
- Overall availability and performance of the name service can be enhanced by installing VAX DNS on multiple nodes. VAX DNS automatically maintains consistency of the DNS namespace across all VAX DNS nodes.
- Access control to each name in the namespace. This set of access control rights consists of read, write, delete, test, and control.
- Management control program to control DNS operation and display statistical and error information.
- Network event logging using the standard DECnet-VAX event logging facility. The DECnet-VAX NCP utility can be used to enable and disable DNS events.

For more information on VAX DNS, refer to the *VAX Distributed Name Service Management Guide*.

6 Programming Techniques for VAXcluster Applications

Chapter 6 presents programming techniques to implement the following application requirements:

- Remote Process Creation

  In a VAXcluster system, a process executing on a local VAXcluster CPU can create a process on a remote VAXcluster CPU. (See Section 6.1.)

- Data Sharing

  Processes executing on different VAXcluster CPUs can achieve coordinated access for shared data. (See Section 6.2.)

- Process Synchronization

  Interprocess communications between processes on different VAXcluster CPUs can be designed to implement a VAXcluster application. (See Section 6.3.)

- Exception Conditions

  Processes can be programmed to recover from a VAXcluster CPU failure or an interprocess communication failure. (See Section 6.4.)

This chapter also provides programming examples that demonstrate how to use these programming techniques to implement a VAXcluster application. When implementing the application design models presented in Chapter 4, Application Design Models for VAXcluster Software, apply the programming technique that is most appropriate for your specific application design. Generally, you will be able to use more than one programming technique to implement your application. However, when deciding which programming technique to use, consider your application's performance and availability goals, and which programming technique your site considers the easiest to use and maintain.

6.1 Remote Process Creation

Remote process creation enables you to design an application to execute single or multiple instruction streams on different VAXcluster CPUs. Remote process creation is an application requirement that is particularly important when implementing the Parallelism Model discussed in Chapter 4. The programming techniques for remote process creation in a VAXcluster system are:

- Transparent DECnet-VAX communications
- Nontransparent DECnet-VAX communications
- VMS batch facility

These techniques are described in the following sections of this manual.

6.1.1 Using Transparent DECnet-VAX Communications

Use transparent DECnet-VAX communications to implement a one-to-one task-to-task communication (see Section 3.5 and Section 4.2.1). Using a high-level programming language that supports DECnet-VAX communications, you can create a remote process and establish transparent task-to-task communications between a local and a remote process. In addition, you can use transparent DECnet-VAX task-to-task communications to create multiple logical links by specifying different unit numbers for each link request. Whether you use single or multiple links, the task-to-task operations for transparent DECnet-VAX communications are performed synchronously for each logical link.

Transparent DECnet–VAX — Programming Example

The following two FORTRAN programs (TASK1.FOR and TASK2.FOR) demonstrate the use of transparent DECnet-VAX communications for remote process creation. The sequence of events for these transparent DECnet-VAX communications is:

1. TASK1.EXE executes on a local node and initiates a logical link request, supplying explicit access control information, to execute TARGET.COM on a remote node.
2. The remote process created on the remote node runs TARGET.COM on the remote node.
3. TARGET.COM runs TASK2.EXE to complete the logical link requested by TASK1.EXE.
4. TASK1.EXE and TASK2.EXE exchange messages.
5. TASK2.EXE and TASK1.EXE disconnect the logical link.

- **TASK1.FOR**

```
c  This program illustrates task-to-task communication by
c  establishing a logical link to run TARGET.COM on a
c  remote node. Prior to running this program
c  the logical name REMOTE must be defined:
c
c $ASSIGN node"""username password""":::"""TASK=TARGET.COM""" -
c         REMOTE
c
c  Note: The logical name REMOTE should only be defined interactively
c        because hardcoding a username and password in a command
c        procedure can create a potential security problem on your
c        system.  In addition, specifying username and password is
c        only done here for the purpose of demonstrating an example.
c        In a production system, other access methods (for example,
c        access through a proxy account) should be used to ensure
c        system security.
c
c
c       Establish network link.
        OPEN (UNIT=1, FILE='REMOTE', STATUS='NEW')
        DO J = 1,10
c
c               Send a message to other task.
                WRITE (1,*) J
```

```
                TYPE *, 'Sent message ', J
c
c               Get message from other task.
                READ  (1,*) I
                TYPE *, 'Received value ', I
        ENDDO
        CLOSE (UNIT=1)
        END
```

- **TARGET.COM**

```
$!  This command procedure is invoked by TASK1.EXE to
$!  establish task-to-task communication between
$!  itself and TASK2.EXE.
$!
$ RUN DISK$WORK:[PROG.FOR]TASK2.EXE
$ PURGE TARGET.LOG
```

- **TASK2.FOR**

```
c  This program illustrates task-to-task
c  communication. It is run by TASK1.EXE by
c  executing the command procedure TARGET.COM
c  on the remote node.
c
c       Connect to network link.
        OPEN (UNIT=1, FILE='SYS$NET', STATUS='OLD')
        DO I = 1,10
c
c               Read message from parent task.
                READ(1,*) J
                J = I * J
c
c               Send message to other task.
                WRITE(1,*) J
                TYPE *, 'J = ', J
        ENDDO
        CLOSE (UNIT=1)
        END
```

6.1.2 Using Nontransparent DECnet-VAX Communications

When programming an application designed to implement a many-to-one or a one-to-many task-to-task communications (see Section 4.2, Client-Server Model, and Section 4.3, Parallelism Model), use nontransparent DECnet-VAX communications to establish multiple logical links for asynchronous operations. In addition, you can use nontransparent task-to-task communications with a mailbox. A mailbox is a virtual device used to establish a queue for a specified I/O channel (unit number). Once a mailbox is created using a VMS system service and associated with a logical link, the mailbox can store all incoming messages. Incoming messages can be either solicited (requested) or unsolicited (unexpected). Solicited messages are always normal data messages. Unsolicited messages either originate from a remote task or are issued by Network Services Program (NSP) as a notification of an exceptional event. For more information on using VMS system services to create a mailbox and NSP message types, see the *VMS Networking Manual*.

Nontransparent DECnet–VAX — Programming Example

In this FORTRAN example, the local process (LOCAL_1.EXE) establishes a network connection with a mailbox to the remote process (REMOTE_1.EXE) to write a message to a disk file.

- **LOCAL_1.FOR**

```
      PROGRAM MATMUL_TEST
c
c  This version of the test program demonstrates the creation of the
c  remote process.  The remote process will only write a message to a
c  disk file and then exit to later prove that it was created.
c
c
c  This call to the Assign with Mailbox run-time library routine will
c  create the channel to the remote process.  By creating this channel,
c  the remote process will be created.
c
c  Most applications would later communicate with this remote process
c  using the QIO system services.  To facilitate this, the channel
c  number that defines the communications path is stored in the
c  variable ICHANNEL.
c
      ISTAT = LIB$ASN_WTH_MBX (
     1                         'NODE1::"0=REMOTE_1"',
     1                         450,
     1                         450,
     1                         ICHANNEL,
     1                         IGNORE)

c
c  If transparent DECnet was going to be used, the communications link
c  could be created by the following FORTRAN statement:
c
c       open (unit=1,file='node1::"0=remote_1"',type='new')

c
c  Exit out.
c
      CALL EXIT (ISTAT)
      END
```

- **REMOTE_1.FOR**

```
      PROGRAM MATMUL_REMOTE_TEST

c  This portion is the remote portion of the first example.  This
c  program will be run on the remote system, write out a "Hello"
c  message to a disk file (to prove it ran), and exit.
c
c
c  Include the VMS system service definitions.
c
      INCLUDE '($SSDEF)'
```

```
c
c  The first thing that is normally done is to establish the
c  communications link back to the calling program.  This example will
c  also use the Assign with Mailbox run-time library routine.  It is
c  important to note that this program does not need to know the name of
c  the calling program. In DECnet/VMS, the logical name SYS$NET contains
c  all the information needed to tell the system which remote link to
c  connect to.
c
        ISTAT = LIB$ASN_WTH_MBX ('sys$net',
        1                        450,
        1                        450,
        1                        ICHANNEL,
        1                        IGNORE )

        IF (ISTAT .NE. SS$_REMOTE) CALL EXIT (ISTAT)

c
c  If transparent DECnet was going to be used, this connection to
c  the calling program could be accomplished by the following
c  FORTRAN statement:
c
c       open (unit=1,file='SYS$NET',type='old')
c

c
c  Next, open a disk file and write a message to indicate that this
c  process was created, and then exit.
c
        OPEN (UNIT=1,FILE='TEST_OUTPUT.DAT',TYPE='NEW')
        WRITE (1,10)
10      FORMAT ('  THE REMOTE PROCESS WAS SUCCESSFULLY CREATED.')

        CALL EXIT
        END
```

6.1.3 Using the VMS Batch Facility

Use the VMS batch facility to create one or more remote processes for batch execution. You can either submit the batch job to a clusterwide batch queue, allowing the distributed job controller to assign your job to a batch execution queue, or your system manager can establish a local queue for batch processing on a dedicated VAXcluster CPU. In addition, when you implement an application design that requires remote process creation, the batch facility provides you with the following features:

- Ability to execute batch jobs in parallel using the distributed job controller to load balance clusterwide batch queues
- Capability for an automatic failover and restart for any batch queues stopped by a hardware failure
- Capability for a DCL command procedure to implement *checkpointing* to prevent duplication of completed batch work during a batch restart

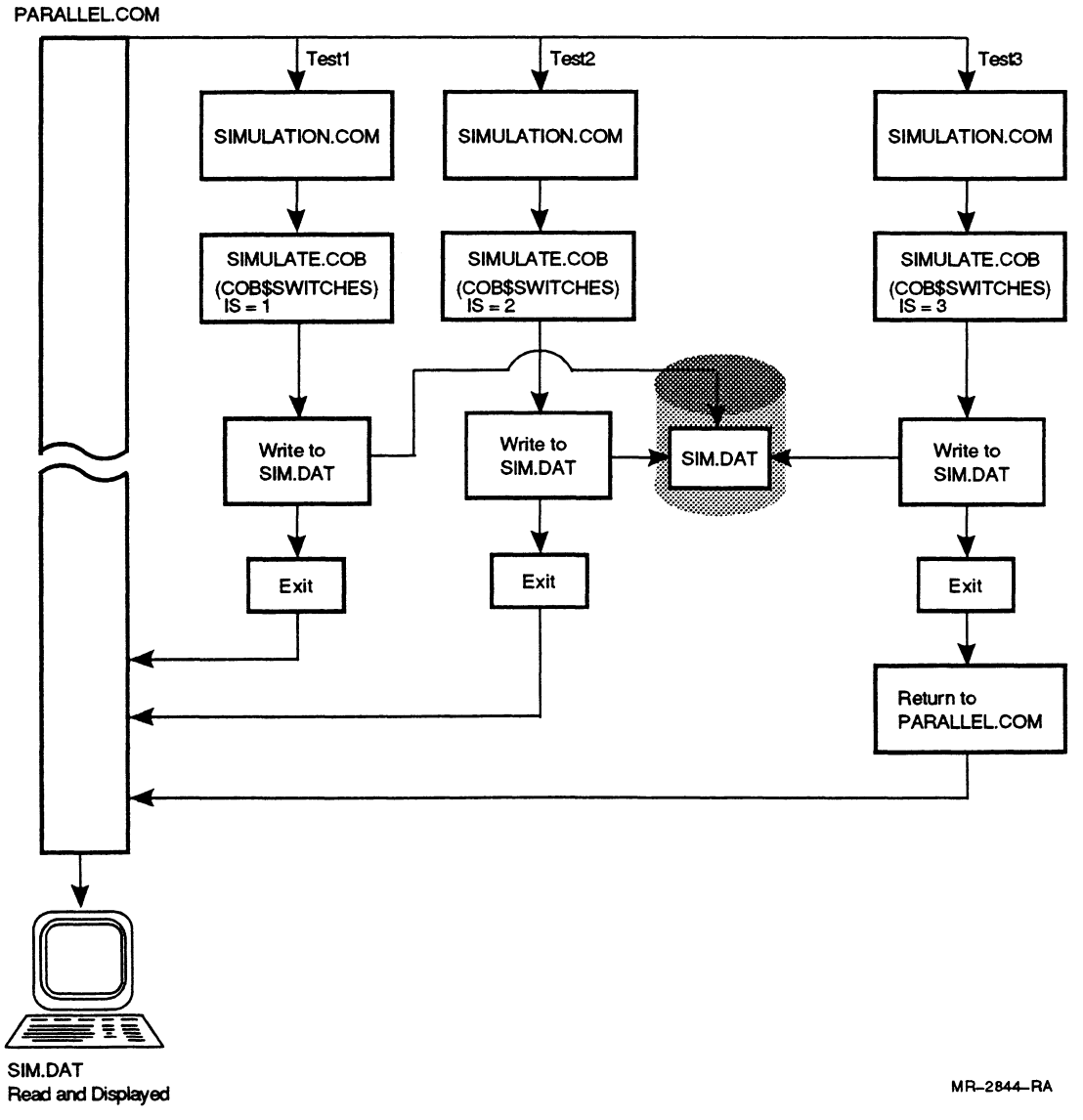
Executing Batch Jobs in Parallel — Programming Example 1

By submitting batch jobs to the clusterwide batch queue, you can use multiple batch execution queues to run batch jobs in parallel. In this example, the DCL command procedure PARALLEL.COM demonstrates the use of a clusterwide batch queue to speed up the processing of a CPU-intensive simulation program (SIMULATE.COB).

Imagine that you are a businessman who wants to buy widgets by the case and you want to determine the appropriate number of cases to buy to maximize profits. Your supplier is willing to sell you widgets in quantities of 100, 200, and 300 cases. Based on a history of widget demand, you develop a frequency distribution and encapsulate that distribution in the COBOL program SIMULATE.COB. SIMULATE.COB determines the average profit for demand quantities of 100, 200, and 300 cases of widgets, if widgets could be sold over a period of 100,000 days.

PARALLEL.COM, see Figure 6–1, submits three batch jobs (TEST1, TEST2, and TEST3) to the clusterwide batch queue (SYS$BATCH). Using a value passed by PARALLEL.COM in the P1 parameter, each batch job executes SIMULATION.COM. For each of the three executions of SIMULATION.COM, the P1 parameter passed from PARALLEL.COM is evaluated and assigned a COB$SWITCHES value. SIMULATION.COM executes SIMULATE.EXE; depending on the COB$SWITCHES value, SIMULATE.EXE executes the appropriate module and writes the result to the output file (SIM.DAT). PARALLEL.COM reads SIM.DAT, checks for three records to ensure that jobs TEST1, TEST2, and TEST3 have all completed, and displays the answer to the user.

Figure 6–1 Flow Diagram for Programming Example 1



- **PARALLEL.COM**

```
$!      NOTE: An empty sequential file SIM.DAT must be created prior
$!            to execution of this procedure.
$!
$ HOUSEKEEPING:
$       SHOW TIME
$       ON ERROR THEN EXIT
$       SET NOON
$       IF F$MODE() .NES. "INTERACTIVE" THEN EXIT
$       SIM_RECORD_COUNT = 0
$       SET ON
$       GOSUB DEFINE
$       GOSUB SUBMIT
$       GOSUB DISPLAY_RESULTS
$       DEASSIGN COMDIR
$       DEASSIGN DEMAND$SIMULATION
$       SHOW TIME
$       EXIT
$!
$!      Define logicals.
$!
$ DEFINE:
$       DEFINE DEMAND$SIMULATION DISK$USER:[EXAMPLES.DATA]SIM.DAT
$       DEFINE COMDIR DISK$USER:[EXAMPLES.COM]
$       RETURN
$!*********************************************************************
$! Parallel processing.
$!
$!      Submit batch jobs in parallel and get job entry number using
$!      the new $ ENTRY feature of VMS Version 5.2.
$!
$ SUBMIT:
$ SUBMIT /NOPRINT /QUEUE=SYS$BATCH /NAME=TEST1 /KEEP /NOID /PARAM = 100 -
        COMDIR:SIMULATION.COM
$ JOB_1 = $ENTRY
$ SUBMIT /NOPRINT /QUEUE=SYS$BATCH /NAME=TEST2 /KEEP /NOID /PARAM = 200 -
        COMDIR:SIMULATION.COM
$ JOB_2 = $ENTRY
$ SUBMIT /NOPRINT /QUEUE=SYS$BATCH /NAME=TEST3 /KEEP /NOID /PARAM = 300 -
        COMDIR:SIMULATION.COM
$ JOB_3 = $ENTRY
$!
$ SYNCHRONIZE:
$       SET NOON        !sync will give error if other jobs done
$!
$!      The next three command lines which use $SYNCHRONIZE illustrate
$!      the way to synchronize on job completion on systems running VMS
$!      versions prior to VMS 5.2. This method continues to work on V5.2
$!      systems but is slower than the method that comes after the three
$!      commented lines.
$!
$!      SYNCHRONIZE /QUEUE=SYS$BATCH TEST1
$!      SYNCHRONIZE /QUEUE=SYS$BATCH TEST2
$!      SYNCHRONIZE /QUEUE=SYS$BATCH TEST3
$!
$       SYNCHRONIZE /ENTRY='JOB_1
$       SYNCHRONIZE /ENTRY='JOB_2
$       SYNCHRONIZE /ENTRY='JOB_3
$       SET ON
$       RETURN
$!*********************************************************************
$! Straight line processing.
$!
$!      Open the simulation file SIM.DAT and read contents.
$!
$ DISPLAY_RESULTS:
$       OPEN/ERROR=NO_FILE SIMULATION DEMAND$SIMULATION
$       WRITE SYS$OUTPUT ""
$       GOSUB READ_SIM_FILE
```

```
$       CLOSE SIMULATION
$       RETURN
$!
$!      Error to indicate that the SIM.DAT simulation output file
$!      was not found.
$!
$ NO_FILE:
$       WRITE SYS$OUTPUT "SIM.DAT SIMULATION FILE DOES NOT EXIST."
$       RETURN
$!
$!      Read simulation file and display information.
$!
$ READ_SIM_FILE:
$    READ/ERROR=RETURN/END_OF_FILE=CHK_COMPLETE_STATUS SIMULATION -
     SIM_RECORD_1
$       DEMAND_QTY = F$EXTRACT(0,3,SIM_RECORD_1)
$       TOTAL_PROFIT = F$CVSI(24,31,SIM_RECORD_1)
$       WRITE SYS$OUTPUT "     IF YOU BUY ",  DEMAND_QTY, " CASES, ", -
                         "YOUR PROFIT WILL BE: ", TOTAL_PROFIT
$       SIM_RECORD_COUNT = SIM_RECORD_COUNT + 1
$       GOTO READ_SIM_FILE
$!
$!      Make sure all three portions of the simulation have completed
$!      and returned their data to SIM.DAT.
$!
$ CHK_COMPLETE_STATUS:
$       IF SIM_RECORD_COUNT .LT. 3
$          THEN TYPE SYS$INPUT

                ONE OR MORE OF THE THREE SIMULATION COMMAND PROCEDURES
                HAS FAILED. CHECK THE FILES TEST1.LOG, TEST2.LOG, AND
                TEST3.LOG FOR CAUSE OF PROBLEM.

$       ENDIF
$       RETURN
```

- **SIMULATION.COM**

```
$!  The command procedure SIMULATION.COM is submitted to the
$!  SYS$BATCH queue three times. Depending on the setting of the
$!  logical COB$SWITCHES, the simulation program generates a
$!  simulation for 100, 200, and 300 cases. COB$SWITCHES is set in
$!  this command procedure. SIMULATION.COM determines what switch
$!  setting to use based on the submit parameter values of 100,
$!  200, or 300. Parameter P1 contains the appropriate demand
$!  quantity for the simulation.
$!
$ DETERMINE_DEMAND:
$       DEFINE DEMAND$SIMULATION DISK$USER:[EXAMPLES.DATA]SIM.DAT
$       IF P1 .EQ. 100 THEN $DEFINE COB$SWITCHES "1"
$       IF P1 .EQ. 200 THEN $DEFINE COB$SWITCHES "2"
$       IF P1 .EQ. 300 THEN $DEFINE COB$SWITCHES "3"
$       SWITCHES = F$TRNLNM("COB$SWITCHES")
$       IF SWITCHES .EQS. "" THEN GOTO EXIT
$!
$ DEFINE_AND_RUN:
$       ON ERROR THEN GOTO EXIT
$       RUN DISK$USER:[EXAMPLES.EXE]SIMULATE.EXE
$!
$!
$ EXIT:
$       DEASSIGN DEMAND$SIMULATION
$       DEASSIGN COB$SWITCHES
$       EXIT
```

- **SIMULATE.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     SIMULATE INITIAL.
AUTHOR.         DIGITAL.
DATE-WRITTEN.   01-10-89.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*****************************************************************
* COB$SWITCHES is a COBOL specific logical name that is defined  *
* in the SPECIAL NAMES paragraph and is defined external to the  *
* image in SIMULATION.COM                                        *
*****************************************************************
SPECIAL-NAMES.
        SWITCH 1 ON IS DEMAND-100
        SWITCH 2 ON IS DEMAND-200
        SWITCH 3 ON IS DEMAND-300.
SOURCE-COMPUTER.  VAX-11.
OBJECT-COMPUTER.  VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT SIM-FILE ASSIGN "DEMAND$SIMULATION"
                ORGANIZATION IS SEQUENTIAL
                FILE STATUS IS PROGRAM-IO-STATUS.

I-O-CONTROL.
        APPLY LOCK-HOLDING ON SIM-FILE.

DATA DIVISION.
FILE SECTION.

FD      SIM-FILE.

01      SIM-RECORD.
        02 SIM-NUMBER-OF-CASES          PIC 9(03).
        02 SIM-AVERAGE-PROFIT           PIC S9(09) COMP.


*****************************************************************
WORKING-STORAGE SECTION.

01      WS-DAYS                 PIC S9(07) COMP         VALUE 100000.
01      WS-SEED                 PIC 9(09)  COMP.
01      WS-TIME                 PIC 9(08).
01      WS-CALL-STATUS          PIC S9(09) COMP.
01      WS-PROFIT               PIC S9(09) COMP.
01      WS-TOTAL-PROFIT         PIC S9(09) COMP.
01      WS-DEMAND-FOR-CASES     PIC S9(09) COMP.
01      WS-RANDOM-NUMBER        COMP-1.
01      WS-RAND-1               PIC 9V99.
01      WS-SCREEN-LINE          PIC 9(02).

01 WS-PRODUCTION-ARRAY.
   02 WS-PRODUCTION-OF-CASES OCCURS 3 TIMES.
      03 WS-NUMBER-OF-CASES     PIC 9(03)  COMP.
      03 WS-AVERAGE-PROFIT      PIC S9(09) COMP.

01      WS-SUB1                 PIC 9(09) COMP.
01      WS-SUB2                 PIC 9(09) COMP.
01      WS-SUB3                 PIC 9(09) COMP.
01      WS-ERROR-OCCURRED       PIC X(01)       VALUE SPACES.
01      WS-ERROR-TEXT           PIC X(40)       VALUE SPACES.

* Standard COBOL error codes and variables.

01 ERROR-VARIABLES.
        02 PROGRAM-IO-STATUS             PIC X(02) VALUE SPACES.
```

```
01 COBOL-IO-STATUS-CODES.
*
* For the purposes of this example, only a subset of all the
* possible COBOL error codes are used. Actual production programs
* should include all error codes and the appropriate logic to trap
* and record them.
*
        02 STS_SUCCESS                    PIC X(02) VALUE "00".
        02 STS_DUPLCT_ALTRT_KEY_WRTTN     PIC X(02) VALUE "02".
        02 STS_END_OF_FILE_10             PIC X(02) VALUE "10".
        02 STS_END_OF_FILE_46             PIC X(02) VALUE "46".
        02 STS_INVLD_KEY_21               PIC X(02) VALUE "21".
        02 STS_INVLD_KEY_22               PIC X(02) VALUE "22".
        02 STS_INVLD_KEY_23               PIC X(02) VALUE "23".
        02 STS_INVLD_KEY_24               PIC X(02) VALUE "24".
        02 STS_RCRD_LOCKED_AVLBL          PIC X(02) VALUE "90".
        02 STS_RCRD_LOCKED_NOT_AVLBL      PIC X(02) VALUE "92".

PROCEDURE DIVISION.
DECLARATIVES.

SIM-FILE-ERROR SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON SIM-FILE.

END DECLARATIVES.
*****************************************************************
MAIN SECTION.

000-MAIN-PROCESS.

        ACCEPT WS-TIME          FROM TIME.
        MOVE WS-TIME            TO WS-SEED.
        IF DEMAND-100
           THEN MOVE 100                TO WS-NUMBER-OF-CASES(1)
           ELSE IF DEMAND-200
                   THEN MOVE 200        TO WS-NUMBER-OF-CASES(1)
                   ELSE MOVE 300        TO WS-NUMBER-OF-CASES(1)
                END-IF
        END-IF.
        PERFORM 100-ACCEPT-DAYS THRU 100-ACCEPT-DAYS-EXIT.
        STOP RUN.

000-MAIN-PROCESS-EXIT.  EXIT.

100-ACCEPT-DAYS.
* The EVALUATE WS-RANDOM-NUMBER block determines the appropriate
* demand for cases based on the number generated by the MTH$RANDOM
* system service call. The random number range (that is, .00
* thru .19) is based on the historical frequency of demand for 100,
* 200 or 300 cases. For example, history has shown that for 20% of
* the time, 100 cases were required to satisfy demand. Profit is
* calculated on 5 dollars per case sold. Appropriate penalties are
* incurred if random demand is greater or less than demand set by
* COB$SWITCHES.
```

```
           MOVE 0          TO WS-AVERAGE-PROFIT(WS-SUB1)
                              WS-TOTAL-PROFIT
                              WS-SUB2.
           MOVE 1          TO WS-SUB1.
           PERFORM VARYING WS-SUB2 FROM 1 BY 1
                  UNTIL WS-SUB2 > WS-DAYS
                  PERFORM 300-GET-RANDOM-NO THRU 300-GET-RANDOM-NO-EXIT
                  EVALUATE WS-RANDOM-NUMBER
                        WHEN .00 THRU .19 MOVE 100 TO WS-DEMAND-FOR-CASES
                        WHEN .20 THRU .39 MOVE 200 TO WS-DEMAND-FOR-CASES
                        WHEN .40 THRU .99 MOVE 300 TO WS-DEMAND-FOR-CASES
                  END-EVALUATE
                 IF WS-DEMAND-FOR-CASES = WS-NUMBER-OF-CASES(WS-SUB1)
                      THEN COMPUTE WS-PROFIT ROUNDED =
                                        5 * WS-NUMBER-OF-CASES(WS-SUB1)
                 ELSE IF WS-DEMAND-FOR-CASES > WS-NUMBER-OF-CASES(WS-SUB1)
                             THEN COMPUTE WS-PROFIT ROUNDED =
                                    (5 * (WS-NUMBER-OF-CASES(WS-SUB1))) +
                                    (-3 * (WS-DEMAND-FOR-CASES -
                                     WS-NUMBER-OF-CASES(WS-SUB1)))
                             ELSE COMPUTE WS-PROFIT ROUNDED =
                                    (5 * WS-DEMAND-FOR-CASES) +
                                    (-10 * (WS-NUMBER-OF-CASES(WS-SUB1) -
                                     WS-DEMAND-FOR-CASES))
                             END-IF
                    END-IF
                    COMPUTE WS-TOTAL-PROFIT ROUNDED =
                                    WS-TOTAL-PROFIT + WS-PROFIT
              END-PERFORM.
       COMPUTE WS-AVERAGE-PROFIT(WS-SUB1) ROUNDED = WS-TOTAL-PROFIT/WS-DAYS.
       PERFORM 400-PRINT-OUT-RESULTS THRU 400-PRINT-OUT-RESULTS-EXIT.

    100-ACCEPT-DAYS-EXIT.   EXIT.

    300-GET-RANDOM-NO.
           CALL "MTH$RANDOM" USING
                   BY REFERENCE WS-SEED
                   GIVING WS-RANDOM-NUMBER.

           COMPUTE WS-RAND-1 ROUNDED = WS-RANDOM-NUMBER * 1.
           MOVE WS-RAND-1          TO WS-RANDOM-NUMBER.

    300-GET-RANDOM-NO-EXIT. EXIT.

    400-PRINT-OUT-RESULTS.
           OPEN EXTEND     SIM-FILE        ALLOWING ALL.
           PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.

           INITIALIZE SIM-RECORD.
           MOVE WS-NUMBER-OF-CASES(1)      TO SIM-NUMBER-OF-CASES.
           MOVE WS-AVERAGE-PROFIT(WS-SUB1) TO SIM-AVERAGE-PROFIT.
           IF WS-ERROR-OCCURRED = SPACES
              THEN MOVE STS_RCRD_LOCKED_NOT_AVLBL  TO PROGRAM-IO-STATUS
                   PERFORM UNTIL PROGRAM-IO-STATUS NOT =
                                        STS_RCRD_LOCKED_NOT_AVLBL
                           WRITE SIM-RECORD ALLOWING NO OTHERS
                           PERFORM CHECK-ERROR-STATUS
                                   THRU CHECK-ERROR-STATUS-EXIT
                   END-PERFORM
           END-IF.

           CLOSE SIM-FILE.

    400-PRINT-OUT-RESULTS-EXIT.     EXIT.

    CHECK-ERROR-STATUS.
```

```
        IF PROGRAM-IO-STATUS NOT = STS_SUCCESS
         AND PROGRAM-IO-STATUS NOT = STS_RCRD_LOCKED_AVLBL
         AND PROGRAM-IO-STATUS NOT = STS_DUPLCT_ALTRT_KEY_WRTTN
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_21
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_22
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_23
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_24
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_10
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_46 THEN
           MOVE "Y"                TO WS-ERROR-OCCURRED
        END-IF.

CHECK-ERROR-STATUS-EXIT.        EXIT.

END PROGRAM SIMULATE.
```

Using Checkpointing with a Batch Restart — Programming Example 2

SYSTEMBUILD.COM demonstrates the use of checkpointing with a restartable batch job to build a software product. This software product is built in three stages. The first stage is executed by submitting a compile command for all of the source code residing on PROGLIST.DAT to the clusterwide batch queue (SYS$BATCH) by executing COMPILE1.COM, COMPILE2.COM, and COMPILE3.COM.

After the compile stage has completed, the object library stage and link stage are executed by SYSTEMBUILD.COM. If during any of the stages of SYSTEMBUILD.COM there is a hardware failure for the batch execution queue running SYSTEMBUILD.COM, SYSTEMBUILD.COM can recover because SYSTEMBUILD defines a RESTART_VALUE to restart SYSTEMBUILD.COM at the stage where the failure occurred. (When a restart is executed, the entire stage is re-done.) SYSTEMBUILD demonstrates checkpointing by using a dynamic value for the RESTART_VALUE.

Note: To implement checkpointing for the execution of the COMPILE.COM, BUILD.COM, and LINK.COM command procedures submitted from SYSTEMBUILD.COM, a RESTART_VALUE must be defined in each of these command procedures.

The following example is **SYSTEMBUILD.COM**.

- **SYSTEMBUILD.COM**

```
$       DEFINE COMDIR   DISK$USER:[SYNCH]
$       DEFINE OUTDIR   DISK$USER:[SYNCH.DATA]
$       DEFINE WORKDIR  DISK$USER:[SYNCH.LOG]
$!
$! If restarting, go to section of code last executing.
$ SET VERIFY
$ IF $RESTART THEN GOTO 'RESTART_VALUE'
$!
$! Hold the code section over a restart.
$ SET RESTART_VALUE="COMPILE"
$!
$ COMPILE:
$       SET PROC/NAME = "COMPILE"
$!
$!
$!*****************************************************
$! This is the COMPILE section of code that fires off  *
$! batch jobs to compile the programs in PROGLIST.DAT. *
$!*****************************************************
$!
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=COMPILE1 COMDIR:COMPILE1.COM
$!
$! COMPILE1.COM creates the file COMPILEDONE_1.DAT
$! as a message flag to indicate the successful completion
$! of COMPILE1.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=COMPILE2 COMDIR:COMPILE2.COM
$!
$! COMPILE2.COM creates the file COMPILEDONE_2.DAT
$! as a message flag to indicate the successful completion
$! of COMPILE2.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=COMPILE3 COMDIR:COMPILE3.COM
$!
$! COMPILE3.COM creates the file COMPILEDONE_3.DAT
$! as a message flag to indicate the successful completion
$! of COMPILE3.COM.
$!
$!
$       SET PROCESS/NAME = "COMPILE1"
$ CALL CHECKDONE COMDIR:COMPILEDONE_1.DAT 20
$       SET PROCESS/NAME = "COMPILE2"
$ CALL CHECKDONE COMDIR:COMPILEDONE_2.DAT 20
$       SET PROCESS/NAME = "COMPILE3"
$ CALL CHECKDONE COMDIR:COMPILEDONE_3.DAT 20
$!
$!
$!
$!
$ SET RESTART_VALUE="OBJECT_LIBRARY"
$! Hold the code section over a restart.
$!
$ OBJECT_LIBRARY:
$       SET PROCESS/NAME = "OBJECT"
$!
$!
$!*****************************************************
$! This is the OBJECT LIBRARY section of code that     *
$! fires off batch jobs to build the object libraries.*
$!*****************************************************
$!
```

```
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:OBJECTLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=BUILDLIB1 COMDIR:BUILD1.COM
$!
$! BUILD1.COM creates the file BUILDDONE_1.DAT
$! as a message flag to indicate the successful completion
$! of BUILD1.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:OBJECTLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=BUILDLIB2 COMDIR:BUILD2.COM
$!
$! BUILD2.COM creates the file BUILDDONE_2.DAT
$! as a message flag to indicate the successful completion
$! of BUILD2.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:OBJECTLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=BUILDLIB3 COMDIR:BUILD3.COM
$!
$! BUILD3.COM creates the file BUILDDONE_3.DAT
$! as a message flag to indicate the successful completion
$! of BUILD3.COM.
$!
$!
$       SET PROCESS/NAME = "OBJECT1"
$ CALL CHECKDONE COMDIR:BUILDDONE_1.DAT 20
$       SET PROCESS/NAME = "OBJECT2"
$ CALL CHECKDONE COMDIR:BUILDDONE_2.DAT 20
$       SET PROCESS/NAME = "OBJECT3"
$ CALL CHECKDONE COMDIR:BUILDDONE_3.DAT 20
$!
$!
$!
$!
$ SET RESTART_VALUE="LINK"
$!
$ LINK:
$  SET PROCESS/NAME = "LINK"
$! Hold the code section over a restart
$!
$!
$!*****************************************************
$! This is the LINK section of code that fires off    *
$! batch jobs to build the executable images.         *
$!*****************************************************
$!
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=LINK1 COMDIR:LINK1.COM
$!
$! LINK1.COM creates the file LINKDONE_1.DAT
$! as a message flag to indicate the successful completion
$! of LINK1.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=LINK2 COMDIR:LINK2.COM
$!
$! LINK2.COM creates the file LINKDONE_2.DAT
$! as a message flag to indicate the successful completion
$! of LINK2.COM.
$!
$ SUBMIT/NOPRINT/LOG=OUTDIR: /PARAMETER=(WORKDIR:PROGLIST.DAT) -
  /QUEUE=SYS$BATCH  /NAME=LINK3 COMDIR:LINK3.COM
$!
$! LINK3.COM creates the file LINKDONE_3.DAT
$! as a message flag to indicate the successful completion
$! of LINK3.COM.
```

```
$!
$       SET PROCESS/NAME = "LINK1"
$ CALL CHECKDONE COMDIR:LINKDONE_1.DAT 20
$       SET PROCESS/NAME = "LINK2"
$ CALL CHECKDONE COMDIR:LINKDONE_2.DAT 20
$       SET PROCESS/NAME = "LINK3"
$ CALL CHECKDONE COMDIR:LINKDONE_3.DAT 20
$!
$ EXIT
$!
$!
$!*************** CHECKDONE subroutine ********************
$!   Called with filename in P1 and retry count in P2.    *
$!*********************************************************
$!
$ CHECKDONE: SUBROUTINE
$ COUNT=0
$ RETRY:
$ OPEN/ERROR=WAITFOR XX 'P1'
$ CLOSE XX
$ DELETE 'P1';*
$ EXIT
$ WAITFOR:
$ COUNT=COUNT + 1
$ IF COUNT .GE. P2 THEN STOP
$ WAIT 0 00:00:30.00
$ GOTO RETRY
$ ENDSUBROUTINE
$!
$!
$ EXIT
```

6.2 Data Sharing

When you design and implement a data sharing application on a VAXcluster system, you can use the following programming techniques:

- DECnet-VAX communications

  Transmitting data between two processes using DECnet-VAX communications can reduce disk I/O requests.

- Using VMS RMS to control record granularity for multiple access

  A high-level programming language uses VMS RMS to provide file synchronization for processes using the same records.

- Read-Only global sections

  Read-Only global sections can provide access to read-only data without requiring disk I/Os to access the data.

- Manual record locking using $QIO and the lock management system services

  By using $QIO and the $ENQ—$DEQ system services, the programmer can control the locking granularity to synchronize file or record access. (For more information on how to use the lock management system services to control resource granularity, refer to Section 3.1.3 and the *Introduction to VMS System Services*.)

All of these techniques, except the use of $QIO and lock management system services for file synchronization, are described in the following sections of this manual.

Note: You can use $QIO and the lock management system services to write a "home-grown" record management service. By specifying a locking mode associated with an $ENQ request for a file, you can control the granularity of data elements at the file or record level. However, you cannot mix $ENQ-$DEQ file locking with VMS RMS file locking in the same application.

6.2.1 Using DECnet-VAX Communications

You can use either transparent or nontransparent DECnet-VAX communications to send and receive data between local and remote processes. Transparent DECnet-VAX communications can only be used for exchanging data synchronously; nontransparent DECnet-VAX communications can be used for a synchronous or an asynchronous exchange of data. In both cases, the local or remote processes can send the data as a whole chunk or part-by-part. Typically, if a large chunk of data is generated by a local process and written to a remote process, the local process will sequentially read the data chunk into an array for data manipulation by the remote process. When data is exchanged part-by-part, the local and remote processes can execute an interactive "conversation."

Data Sharing with Transparent DECnet–VAX Communications Programming Example 1

The following two COBOL programs (HOST.COB and TARGET.COB) demonstrate transparent DECnet-VAX communications for data sharing. The objective of this application is to facilitate a lookup of employee data from any system in a DECnet-VAX computer network by using two-way, transparent, task-to-task communications. The software components of this application are summarized in Table 6–1.

Table 6–1 Required Modules for Two-Way, Transparent, Task-to-Task Communications

HOST VAXcluster CPU	TARGET VAXcluster CPU
HOST.COM	TARGET2.COM
HOST.EXE	TARGET.EXE
	EMPLOYEE.DAT

The sequence of events for these COBOL transparent DECnet-VAX communications is:

1 HOST.COM runs on the local node and initiates a logical link connection request to execute TARGET2.COM on the remote node.

2 The remote process runs TARGET2.COM on the remote node, and TARGET2.COM executes TARGET.EXE to complete the logical link connection.

3 HOST.COM executes HOST.EXE.

4 HOST.EXE sends an employee number over the logical link to the program TARGET.

5 TARGET.EXE reads the employee number and uses the number to perform a keyed read on the EMPLOYEE.DAT file. TARGET either sends the requested employee data back to HOST or sends an appropriate message indicating the status of the employee lookup.

6 TARGET.EXE disconnects the network link.

7 HOST.EXE disconnects the network link.

8 HOST.COM disconnects the network link.

Note: When performing network operations using the COBOL language, you must establish a logical link for read/write in DCL before you run the programs. The programs (HOST.COB and TARGET.COB) issue two open statements on the established link, one for input, the other for output. You must establish a logical link using a DCL command procedure before you run the COBOL programs, because the COBOL language does not provide a file open mode for a sequential file organization that supports both read and write operations.

- **HOST.COM**

```
$!      Note:
$!
$! This procedure assumes that the logical REMOTE has been
$! established prior to running this command procedure. The
$! logical is assigned as follows:
$!
$! DEFINE REMOTE  node""""username password""""::""""task=target2.com""""
$!
$!  Note: The logical name REMOTE should only be defined interactively
$!        because hardcoding a username and password in a command
$!        procedure can create a potential security problem on your
$!        system. In addition, specifying username and password is
$!        only done here for the purpose of demonstrating an example.
$!        In a production system, other access methods (for example,
$!        access through a proxy account) should be used to ensure
$!        system security.
$!
```

```
$!
$!
$ ESTABLISH_LINK:
$    OPEN/READ/WRITE/ERROR=NO_REMOTE LOGICAL_LINK REMOTE
$    DEFINE/USER SYS$INPUT SYS$COMMAND
$    RUN HOST.EXE
$    CLOSE LOGICAL_LINK
$    EXIT
$!
$ NO_REMOTE:
$    TEST_LOG :== 'F$TRNLNM("REMOTE")
$    IF TEST_LOG .EQS. ""
$      THEN WRITE SYS$OUTPUT "LOGICAL REMOTE IS NOT ASSIGNED."
$      ELSE WRITE SYS$OUTPUT "LOGICAL LINK TO REMOTE NODE HAS FAILED."
$    ENDIF
$    EXIT
```

- **HOST.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     HOST.
AUTHOR.         DIGITAL.
DATE-WRITTEN.   01-25-89.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*********************     d e s c r i p t i o n       ***************
*                                                                     *
*   This program is part of an application used to demonstrate        *
*   transparent task-to-task communication. HOST.COB is run from the  *
*   command procedure HOST.COM on the local node.                     *
*                                                                     *
***********************************************************************
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT  HOST-LINK-INPUT         ASSIGN "LOGICAL_LINK"
                ORGANIZATION IS SEQUENTIAL
                FILE STATUS IS PROGRAM-IO-STATUS.

        SELECT  HOST-LINK-OUTPUT        ASSIGN "LOGICAL_LINK"
                ORGANIZATION IS SEQUENTIAL
                FILE STATUS IS PROGRAM-IO-STATUS.

DATA DIVISION.
FILE SECTION.

FD      HOST-LINK-OUTPUT.

01      NETWORK-OUTPUT-RECORD.
        02 FILLER                              PIC X(01) VALUE SPACES.
        02 NETOUT-EMPLOYEE-NUMBER              PIC 9(04) VALUE SPACES.
        02 FILLER                              PIC X(40) VALUE SPACES.

FD      HOST-LINK-INPUT.

01      NETWORK-INPUT-RECORD.
        02 NETIN-STATUS                        PIC X(01).
        02 NETIN-EMPLOYEE-NUMBER               PIC 9(04).
        02 NETIN-EMPLOYEE-DATA.
           05 NETIN-EMPLOYEE-FIRST-NAME        PIC X(15).
           05 NETIN-EMPLOYEE-LAST-NAME         PIC X(25).
        02 REDEFINES NETIN-EMPLOYEE-DATA.
           05 NETIN-MESSAGE                    PIC X(40).
```

```
****************************************************************
WORKING-STORAGE SECTION.

01 WS-ERROR-OCCURRED          PIC X(01)                  VALUE SPACES.
01 WS-END-OF-FILE             PIC X(01)                  VALUE SPACES.
01 WS-ERROR-TEXT              PIC X(40)                  VALUE SPACES.

01 WS-RECORD-LENGTH           PIC 9(03)                  VALUE ZEROS.

01 WS-EMPLOYEE-NUMBER         PIC 9(04)                  VALUE ZEROS.

****************************************************************

* Standard COBOL error codes and variables.

01 ERROR-VARIABLES.
        02 PROGRAM-IO-STATUS             PIC X(02) VALUE SPACES.

01 COBOL_IO_STATUS_CODES.
* For the purposes of this example, only a subset of all the
* possible COBOL error codes are used. Actual production programs
* should include all error codes and the appropriate logic to trap
* and record them.
*
        02 STS_SUCCESS                   PIC X(02) VALUE "00".
        02 STS_DUPLCT_ALTRT_KEY_WRTTN    PIC X(02) VALUE "02".
        02 STS_END_OF_FILE_10            PIC X(02) VALUE "10".
        02 STS_END_OF_FILE_46            PIC X(02) VALUE "46".
        02 STS_INVLD_KEY_21              PIC X(02) VALUE "21".
        02 STS_INVLD_KEY_22              PIC X(02) VALUE "22".
        02 STS_INVLD_KEY_23              PIC X(02) VALUE "23".
        02 STS_INVLD_KEY_24              PIC X(02) VALUE "24".
        02 STS_RCRD_LOCKED_AVLBL         PIC X(02) VALUE "90".
        02 STS_RCRD_LOCKED_NOT_AVLBL     PIC X(02) VALUE "92".

PROCEDURE DIVISION.
****************************************************************
DECLARATIVES.

HOST-LINK-INPUT-ERROR   SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON HOST-LINK-INPUT.

HOST-LINK-OUTPUT-ERROR  SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON HOST-LINK-OUTPUT.

END DECLARATIVES.
****************************************************************
MAIN-PROCESS SECTION.

BEGIN-PROCESSING.
*
*       Establish network link.
*
        OPEN OUTPUT HOST-LINK-OUTPUT ALLOWING ALL.
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.
        IF WS-ERROR-OCCURRED = SPACES
         THEN OPEN INPUT HOST-LINK-INPUT ALLOWING ALL
           PERFORM CHECK-ERROR-STATUS    THRU CHECK-ERROR-STATUS-EXIT
        END-IF.
        DISPLAY "" AT LINE 1 AT COLUMN 1 ERASE TO END OF SCREEN.
        MOVE 1                          TO WS-EMPLOYEE-NUMBER.
        PERFORM GET-EMPLOYEE-RECORDS THRU GET-EMPLOYEE-RECORDS-EXIT
                UNTIL WS-EMPLOYEE-NUMBER = 0    OR
                WS-ERROR-OCCURRED = "Y".
        PERFORM CLOSE-FILES             THRU CLOSE-FILES-EXIT.
        DISPLAY "" AT LINE 1 AT COLUMN 1 ERASE TO END OF SCREEN.
        STOP RUN.

BEGIN-PROCESSING-EXIT.  EXIT.
```

```
GET-EMPLOYEE-RECORDS.

     MOVE ZEROS              TO WS-EMPLOYEE-NUMBER.
     DISPLAY "" AT LINE NUMBER 5 AT COLUMN NUMBER 1 ERASE TO END OF LINE
     DISPLAY "" AT LINE NUMBER 6 AT COLUMN NUMBER 1 ERASE TO END OF LINE
     DISPLAY "EMPLOYEE NUMBER. <CR> OR 0 TO EXIT"
                AT LINE NUMBER 3
                AT COLUMN NUMBER 1
                ERASE TO END OF LINE.
          ACCEPT WS-EMPLOYEE-NUMBER
                FROM LINE NUMBER 3
                FROM COLUMN NUMBER 36
                BOLD
                WITH CONVERSION
        END-ACCEPT.

        MOVE WS-EMPLOYEE-NUMBER         TO NETOUT-EMPLOYEE-NUMBER
        WRITE NETWORK-OUTPUT-RECORD
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT
        IF WS-ERROR-OCCURRED NOT = "Y"  AND
           WS-EMPLOYEE-NUMBER > 0
           THEN READ HOST-LINK-INPUT RECORD
                     AT END CONTINUE
                END-READ
                PERFORM CHECK-ERROR-STATUS THRU CHECK-ERROR-STATUS-EXIT
                IF WS-ERROR-OCCURRED NOT = "Y"
                   THEN PERFORM DISPLAY-EMPLOYEE-RECORD
                                THRU DISPLAY-EMPLOYEE-RECORD-EXIT
                END-IF
        END-IF.

GET-EMPLOYEE-RECORDS-EXIT.      EXIT.

DISPLAY-EMPLOYEE-RECORD.
        EVALUATE NETIN-STATUS
           WHEN "E"     DISPLAY  NETIN-MESSAGE
                                 AT LINE NUMBER 10
                                 AT COLUMN NUMBER 1
                                 ERASE TO END OF SCREEN
           WHEN "W"     PERFORM  DISPLAY-TITLE-HEADERS
                                 THRU DISPLAY-TITLE-HEADERS-EXIT
                        DISPLAY  NETIN-EMPLOYEE-NUMBER
                                 AT LINE NUMBER 14
                                 AT COLUMN NUMBER 21
                                 ERASE TO END OF LINE
                        DISPLAY  NETIN-MESSAGE
                                 AT LINE NUMBER 14
                                 AT COLUMN NUMBER 30
           WHEN OTHER   PERFORM  DISPLAY-TITLE-HEADERS
                                 THRU DISPLAY-TITLE-HEADERS-EXIT
                        DISPLAY  NETIN-EMPLOYEE-NUMBER
                                 AT LINE NUMBER 14
                                 AT COLUMN NUMBER 21
                                 ERASE TO END OF LINE
                        DISPLAY  NETIN-EMPLOYEE-FIRST-NAME
                                 AT LINE NUMBER 14
                                 AT COLUMN NUMBER 30
                        DISPLAY  NETIN-EMPLOYEE-LAST-NAME
                                 AT LINE NUMBER 14
                                 AT COLUMN NUMBER 49
         END-EVALUATE.

 DISPLAY-EMPLOYEE-RECORD-EXIT.   EXIT.
```

```
DISPLAY-TITLE-HEADERS.
        DISPLAY ""
                AT LINE NUMBER 10
                AT COLUMN NUMBER 1
                ERASE TO END OF SCREEN.
        DISPLAY "EMPLOYEE: "
                 AT LINE NUMBER 10
                 AT COLUMN NUMBER 10
                 ERASE TO END OF LINE.
        DISPLAY  "NUMBER"
                 AT LINE NUMBER 12
                 AT COLUMN NUMBER 20
                 ERASE TO END OF LINE.
        DISPLAY  "FIRST NAME"
                 AT LINE NUMBER 12
                 AT COLUMN NUMBER 33.
        DISPLAY  "LAST NAME"
                 AT LINE NUMBER 12
                 AT COLUMN NUMBER 57.

DISPLAY-TITLE-HEADERS-EXIT.     EXIT.

CHECK-ERROR-STATUS.
*
*       Process I-O status returned in program-io-status variable.
*
        IF PROGRAM-IO-STATUS NOT = STS_SUCCESS
         AND PROGRAM-IO-STATUS NOT = STS_RCRD_LOCKED_AVLBL
         AND PROGRAM-IO-STATUS NOT = STS_DUPLCT_ALTRT_KEY_WRTTN
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_21
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_22
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_23
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_24
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_10
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_46 THEN
           MOVE "Y"                 TO WS-ERROR-OCCURRED
        END-IF.

CHECK-ERROR-STATUS-EXIT.        EXIT.

*
*       Disconnect the network link.
*
CLOSE-FILES.
        CLOSE HOST-LINK-INPUT.
        CLOSE HOST-LINK-OUTPUT.

CLOSE-FILES-EXIT.       EXIT.
```

- **TARGET2.COM**

```
$!  This DCL command procedure is invoked by HOST.EXE to
$!  establish task-to-task communication between
$!  itself and TARGET.EXE.  TARGET2.COM is executed
$!  on the remote node and must be in the default
$!  directory of the username specified in the DEFINE
$!  command described in HOST.COM.
$!
$ INITIATE_TARGET:
$       OPEN/READ/WRITE REMOTE_LINK SYS$NET
$       RUN TARGET.EXE
$       CLOSE REMOTE_LINK
$       EXIT
$!
```

- **TARGET.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     TARGET.
AUTHOR.         DIGITAL.
DATE-WRITTEN.   01-25-89.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*********************     d e s c r i p t i o n       ***************
*                                                                     *
*  TARGET is part of the two-way transparent task-to-task operation   *
*  that receives an employee number from HOST, performs a file lookup *
*  and returns employee information back to HOST for display.         *
*                                                                     *
***********************************************************************
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT TARGET-LINK-INPUT  ASSIGN TO "REMOTE_LINK"
                ORGANIZATION IS SEQUENTIAL
                FILE STATUS IS PROGRAM-IO-STATUS.

        SELECT TARGET-LINK-OUTPUT ASSIGN TO "REMOTE_LINK"
                ORGANIZATION IS SEQUENTIAL
                FILE STATUS IS PROGRAM-IO-STATUS.

        SELECT EMPLOYEE-FILE ASSIGN "EMPLOYEE.DAT"
               ORGANIZATION IS INDEXED
               FILE STATUS IS PROGRAM-IO-STATUS.

DATA DIVISION.
FILE SECTION.
FD      TARGET-LINK-INPUT.

01      NETWORK-INPUT-RECORD.
        02 FILLER                                PIC X(01).
        02 NETIN-EMPLOYEE-NUMBER                 PIC 9(04).
        02 FILLER                                PIC X(40).


FD      TARGET-LINK-OUTPUT.

01      NETWORK-OUTPUT-RECORD.
        02 NETOUT-STATUS                         PIC X(01).
        02 NETOUT-EMPLOYEE-NUMBER                PIC 9(04).
        02 NETOUT-EMPLOYEE-DATA.
           05 NETOUT-EMPLOYEE-FIRST-NAME         PIC X(15).
           05 NETOUT-EMPLOYEE-LAST-NAME          PIC X(25).
        02 REDEFINES NETOUT-EMPLOYEE-DATA.
           05 NETOUT-MESSAGE                     PIC X(40).

FD      EMPLOYEE-FILE
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS EMPLOYEE-NUMBER.

01      EMPLOYEE_RECORD.
        02 EMPLOYEE_NUMBER                       PIC 9(04).
        02 EMPLOYEE_FIRST_NAME                   PIC X(15).
        02 EMPLOYEE_LAST_NAME                    PIC X(25).
```

```
*****************************************************************

WORKING-STORAGE SECTION.

01 WS-ERROR-OCCURRED                    PIC X(01) VALUE SPACES.
01 WS-ERROR-TEXT                        PIC X(40) VALUE SPACES.

01 WS-ERROR-MSG-LINE.
        02 FILLER                PIC X(19) VALUE "COBOL ERROR NUMBER ".
        02 WS-PROGRAM-IO-STATUS PIC X(02) VALUE SPACES.
        02 FILLER                PIC X(14) VALUE " HAS OCCURRED.".

*****************************************************************

* Standard COBOL error codes and variables.

01 ERROR-VARIABLES.
        02 PROGRAM-IO-STATUS             PIC X(02) VALUE SPACES.

01 COBOL_IO_STATUS_CODES.
* For the purposes of this example, only a subset of all the
* possible COBOL error codes are used. Actual production programs
* should include all error codes and the appropriate logic to trap
* and record them.
*
        02 STS_SUCCESS                   PIC X(02) VALUE "00".
        02 STS_DUPLCT_ALTRT_KEY_WRTTN    PIC X(02) VALUE "02".
        02 STS_END_OF_FILE_10            PIC X(02) VALUE "10".
        02 STS_END_OF_FILE_46            PIC X(02) VALUE "46".
        02 STS_INVLD_KEY_21              PIC X(02) VALUE "21".
        02 STS_INVLD_KEY_22              PIC X(02) VALUE "22".
        02 STS_INVLD_KEY_23              PIC X(02) VALUE "23".
        02 STS_INVLD_KEY_24              PIC X(02) VALUE "24".
        02 STS_RCRD_LOCKED_AVLBL         PIC X(02) VALUE "90".
        02 STS_RCRD_LOCKED_NOT_AVLBL     PIC X(02) VALUE "92".

PROCEDURE DIVISION.
*****************************************************************
DECLARATIVES.

TARGET-LINK-INPUT-ERROR SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON TARGET-LINK-INPUT.

TARGET-LINK-OUTPUT-ERROR        SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON TARGET-LINK-OUTPUT.

EMPLOYEE-FILE-ERROR             SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON EMPLOYEE-FILE.

END DECLARATIVES.
*****************************************************************
MAIN-PROCESS SECTION.

BEGIN-PROCESSING.
*
*       Connect to network link and open the employee file.
*
     OPEN INPUT TARGET-LINK-INPUT ALLOWING ALL.
     PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.
      IF WS-ERROR-OCCURRED = SPACES
        THEN OPEN INPUT EMPLOYEE-FILE
          PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT
            IF WS-ERROR-OCCURRED = SPACES
               THEN OPEN OUTPUT TARGET-LINK-OUTPUT ALLOWING ALL
                 PERFORM CHECK-ERROR-STATUS THRU CHECK-ERROR-STATUS-EXIT
            END-IF
        END-IF.
```

```
        IF WS-ERROR-OCCURRED = SPACES
           THEN PERFORM UNTIL WS-ERROR-OCCURRED = "Y"   OR
                              NETIN-EMPLOYEE-NUMBER = 0
                        PERFORM PROCESS-NETWORK-LINK
                                THRU PROCESS-NETWORK-LINK-EXIT
                        PERFORM CHECK-ERROR-STATUS
                                THRU CHECK-ERROR-STATUS-EXIT
                        IF WS-ERROR-OCCURRED = SPACES
                           THEN PERFORM SEND-RESPONSE-TO-HOST
                                        THRU SEND-RESPONSE-TO-HOST-EXIT
                        END-IF
                END-PERFORM
        END-IF.
        IF WS-ERROR-OCCURRED = "Y"
           THEN MOVE "E"                TO NETOUT-STATUS
                MOVE PROGRAM-IO-STATUS  TO WS-PROGRAM-IO-STATUS
                MOVE WS-ERROR-MSG-LINE  TO NETOUT-MESSAGE
                PERFORM SEND-RESPONSE-TO-HOST
                        THRU SEND-RESPONSE-TO-HOST-EXIT
        END-IF.
        PERFORM CLOSE-FILES             THRU CLOSE-FILES-EXIT.
        STOP RUN.

BEGIN-PROCESSING-EXIT.  EXIT.

*****************************************************************
*                                                               *
*   1.) Read the customer identifier passed from the source program   *
*       over the network line.                                  *
*                                                               *
*   2.) Use the customer identifier to perform a keyed read on the    *
*       employee file.                                          *
*                                                               *
*****************************************************************

PROCESS-NETWORK-LINK.
        READ TARGET-LINK-INPUT
                AT END CONTINUE
        END-READ.
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT

        IF WS-ERROR-OCCURRED = SPACES
           THEN MOVE NETIN-EMPLOYEE-NUMBER      TO EMPLOYEE-NUMBER
                READ EMPLOYEE-FILE RECORD
                     KEY IS EMPLOYEE-NUMBER
                     INVALID KEY INITIALIZE NETOUT-MESSAGE
                             MOVE "W"                TO NETOUT-STATUS
                             MOVE EMPLOYEE-NUMBER
                                     TO NETOUT-EMPLOYEE-NUMBER
                             MOVE "EMPLOYEE WITH THIS # DOES NOT EXIST."
                                     TO NETOUT-MESSAGE
                END-READ
                PERFORM CHECK-ERROR-STATUS  THRU CHECK-ERROR-STATUS-EXIT
                IF PROGRAM-IO-STATUS = STS-SUCCESS
                 THEN INITIALIZE NETWORK-OUTPUT-RECORD
                   MOVE EMPLOYEE-NUMBER     TO NETOUT-EMPLOYEE-NUMBER
                   MOVE EMPLOYEE-LAST-NAME  TO NETOUT-EMPLOYEE-LAST-NAME
                   MOVE EMPLOYEE-FIRST-NAME TO NETOUT-EMPLOYEE-FIRST-NAME
                END-IF
        END-IF.

PROCESS-NETWORK-LINK-EXIT.      EXIT.

SEND-RESPONSE-TO-HOST.

        WRITE NETWORK-OUTPUT-RECORD.
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.

SEND-RESPONSE-TO-HOST-EXIT.     EXIT.
```

```
CHECK-ERROR-STATUS.
*
*       Process I-O status returned in program-io-status variable.
*
        IF PROGRAM-IO-STATUS NOT = STS_SUCCESS
         AND PROGRAM-IO-STATUS NOT = STS_RCRD_LOCKED_AVLBL
         AND PROGRAM-IO-STATUS NOT = STS_DUPLCT_ALTRT_KEY_WRTTN
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_21
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_22
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_23
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_24
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_10
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_46 THEN
           MOVE "Y"                TO WS-ERROR-OCCURRED
        END-IF.

CHECK-ERROR-STATUS-EXIT.        EXIT.

*
*       Disconnect the network link.
*
CLOSE-FILES.
        CLOSE TARGET-LINK-INPUT.
        CLOSE TARGET-LINK-OUTPUT.
        CLOSE EMPLOYEE-FILE.

CLOSE-FILES-EXIT.       EXIT.
```

Data Sharing with Nontransparent DECnet–VAX Communications — Programming Example 2

In this FORTRAN example, nontransparent DECnet-VAX communications are used to send the contents of two input arrays from the local process (LOCAL_2.EXE) to the remote process (REMOTE_2.EXE). The remote process then performs matrix multiplication using the input arrays, and returns the results to the local process.

- **LOCAL_2.FOR**

```
        PROGRAM MATMUL_TEST
c
c  In this example program, the contents of the two input arrays are
c  sent to the remote process.  The remote process then performs the
c  matrix multiplication and returns the results.
c
        EXTERNAL IO$_READVBLK, IO$_WRITEVBLK
        INTEGER SYS$QIO, SYS$QIOW, READ_FUNCTION, WRITE_FUNCTION
        INCLUDE '($SSDEF)'
        PARAMETER ISIZE=100
        INTEGER IN_ARRAY1(ISIZE,ISIZE), IN_ARRAY2(ISIZE,ISIZE),
        1       RESULTS(ISIZE,ISIZE)
        INTEGER*2 IOSB(4)

        READ_FUNCTION = %LOC(IO$_READVBLK)
        WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)
c
c  Put sample data into the input arrays for the matrix multiplication.
c
        DO I = 1, ISIZE
           DO J = 1, ISIZE
              IN_ARRAY1(I,J) = J-(ISIZE/2)
              IN_ARRAY2(I,J) = J-(ISIZE/2)
           ENDDO
        ENDDO
```

```
c
c  Open the channel to the remote process.
c
      ISTAT = LIB$ASN_WTH_MBX (
     1                         'NODE1::"0=REMOTE_2"',
     1                         450,
     1                         450,
     1                         ICHANNEL,
     1                         IGNORE)

      IF (ISTAT .NE. SS$_REMOTE) CALL EXIT (ISTAT)

c
c  Send all the data to the remote process.
c
c  Note that to make this example more realistic, the size of the
c  arrays (250 by 250) has been selected to be too large to fit
c  into a single QIO call. The arrays are therefore sent one row
c  at a time.
c
      DO J = 1,ISIZE
            ISTAT = SYS$QIO (,%VAL(ICHANNEL),%VAL(WRITE_FUNCTION)
     1        ,IOSB,,,IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
            ISTAT = SYS$QIO (,%VAL(ICHANNEL),%VAL(WRITE_FUNCTION)
     1        ,IOSB,,,IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
      ENDDO
c
c  Finally, wait for the results to be sent back.  They are
c  sent back row by row.
c

c
c  Set up to receive a message back from this node.
c
     DO J = 1,ISIZE
        ISTAT = SYS$QIOW (,%VAL(ICHANNEL),%VAL(READ_FUNCTION),IOSB,,,
     1       RESULTS(1,J),%VAL(ISIZE*4),,,,)
     ENDDO

c
c  This completes the work performed on the remote portion.  Deassign
c  the I/O channel and exit the program.
c
      CALL SYS$DASSGN (%VAL(ICHANNEL))

      CALL EXIT
      END
```

- **REMOTE_2.FOR**

```
      PROGRAM MATMUL_REMOTE_TEST
c
c  This is the remote portion of the example that will perform a
c  matrix multiply on a remote node.
c
      EXTERNAL IO$_READVBLK, IO$_WRITEVBLK
      PARAMETER ISIZE=100
      INTEGER IN_ARRAY1(ISIZE,ISIZE), IN_ARRAY2(ISIZE,ISIZE),
     1       RESULTS(ISIZE,ISIZE), READ_FUNCTION, WRITE_FUNCTION
      INTEGER*2 IOSB(4)

c
c  Establish the communication link back to the calling
c  program.
```

```
c
      ISTAT = LIB$ASN_WTH_MBX ('sys$net',
      1                        450,
      1                        450,
      1                        ICHANNEL,
      1                        IGNORE )

      READ_FUNCTION = %LOC(IO$_READVBLK)
      WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)

c
c  Start by getting all the data.  Since the calling program sends
c  the data one row at a time, this program must receive the data in
c  the same array-location order.
c
     DO J = 1,ISIZE
        ISTAT = SYS$QIOW (,%VAL(ICHANNEL),%VAL(READ_FUNCTION),IOSB,,,
     1       IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
        IF (ISTAT .NE. SS$_NORMAL) CALL EXIT (ISTAT)
        ISTAT = SYS$QIOW (,%VAL(ICHANNEL),%VAL(READ_FUNCTION),IOSB,,,
     1       IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
        IF (ISTAT .NE. SS$_NORMAL) CALL EXIT (ISTAT)
     ENDDO

c
c  All the data is here.  Perform the matrix multiplication on
c  each row.
c
c  Note that as a performance optimization, each row is sent back
c  as it is calculated.  Thus, the time spent performing the QIOs
c  overlaps with the calculations of the next rows.
c
c  loop for every row in the results array
      DO J = 1,ISIZE

c  and for every column in the results array
         DO I = 1,ISIZE

c  This is the inner loop of the matrix multiply.
            RESULTS(I,J) = 0
            DO K = 1,ISIZE
               RESULTS(I,J) = RESULTS(I,J) + (IN_ARRAY1(I,K) *
     1             IN_ARRAY2(K,J))
            ENDDO

         ENDDO

c
c  Now that an entire row is completed, send it back.
c
         ISTAT = SYS$QIOW (,%VAL(ICHANNEL),%VAL(WRITE_FUNCTION)
     1       ,IOSB,,,RESULTS(1,J),%VAL(ISIZE*4),,,,)

      ENDDO

c
c  All done.
c
c  Exiting from the program breaks the network link.  To
c  avoid breaking this link before the local program received all the
c  data, this next read operation keeps this program active until
c  the local program terminates the link from its end.
c
      ISTAT = SYS$QIOW (,%VAL(ICHANNEL),%VAL(READ_FUNCTION),IOSB,,,
      1       IN_ARRAY1(1,1),%VAL(ISIZE*4),,,,)

      CALL SYS$DASSGN (%VAL(ICHANNEL))

      CALL EXIT
      END
```

6.2.2 Using VMS RMS to Control Record Granularity for Multiple Access

High-level languages can be programmed to control resource granularity which is the mix of file and record access paths. Some high-level languages can be programmed to specify the degree of file sharing at the record level. Consult the user manual of your high-level language's to determine the capabilities of your language.

Using VMS RMS to Control Record Granularity — Programming Example

The following COBOL program (EMPLOYEE.COB) demonstrates the use of VMS RMS automatic record locking. In this program, multiple users are allowed concurrent read access to the employee file (EMPLOYEE.DAT) **except** in the case where a record of the employee file is locked exclusively by another user for updating.

- **EMPLOYEE.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     EMPLOYEE.
AUTHOR.         DIGITAL.
DATE-WRITTEN.   01-04-89.

********************      d e s c r i p t i o n       *****************
*  This cobol program demonstrates the use of automatic record locking  *
*  in an environment where multiple users are allowed access to the     *
*  employee file (Concurrent Read). No other users are allowed access   *
*  to a given employee record when that record is locked exclusively by *
*  another user.                                                        *
*                                                                       *
*  This program assumes the EMPLOYEE.DAT file exists and contains       *
*  several records for modification.                                    *
************************************************************************

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.  VAX-11.
OBJECT-COMPUTER.  VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT EMPLOYEE-FILE ASSIGN "EMPLOYEE.DAT"
               ORGANIZATION IS INDEXED
               FILE STATUS IS PROGRAM-IO-STATUS.

*****************************************************************

DATA DIVISION.
FILE SECTION.

FD      EMPLOYEE-FILE
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS EMPLOYEE-NUMBER.

01      EMPLOYEE_RECORD.
        02 EMPLOYEE_NUMBER                    PIC 9(04).
        02 EMPLOYEE_FIRST_NAME                PIC X(15).
        02 EMPLOYEE_LAST_NAME                 PIC X(25).
```

```
*****************************************************************
WORKING-STORAGE SECTION.

01 WS-ERROR-OCCURRED          PIC X(01)                  VALUE SPACES.
01 WS-END-OF-FILE             PIC X(01)                  VALUE SPACES.
01 WS-ERROR-TEXT              PIC X(40)                  VALUE SPACES.

01 WS-SAVE-EMPLOYEE-RECORD    PIC X(44)                  VALUE SPACES.

01 WS-LAST-NAME-MSG-LINE.
        02 FILLER                                  PIC X(23)
                VALUE "CHANGE LAST NAME FROM: ".
        02 WS-EMPL-LAST-NAME                      PIC X(25).

01 WS-FIRST-NAME-MSG-LINE.
        02 FILLER                                  PIC X(24)
                VALUE "CHANGE FIRST NAME FROM: ".
        02 WS-EMPL-FIRST-NAME                     PIC X(15).

*****************************************************************

* Standard COBOL error codes, and variables.

01 ERROR-VARIABLES.
        02 PROGRAM-IO-STATUS             PIC X(02) VALUE SPACES.

01 COBOL_IO_STATUS_CODES.
* For the purposes of this example, only a subset of all the
* possible COBOL error codes are used. Actual production programs
* must include all error codes and the appropriate logic to trap
* and record them.
*
        02 STS_SUCCESS                   PIC X(02) VALUE "00".
        02 STS_DUPLCT_ALTRT_KEY_WRTTN    PIC X(02) VALUE "02".
        02 STS_END_OF_FILE_10            PIC X(02) VALUE "10".
        02 STS_END_OF_FILE_46            PIC X(02) VALUE "46".
        02 STS_INVLD_KEY_21              PIC X(02) VALUE "21".
        02 STS_INVLD_KEY_22              PIC X(02) VALUE "22".
        02 STS_INVLD_KEY_23              PIC X(02) VALUE "23".
        02 STS_INVLD_KEY_24              PIC X(02) VALUE "24".
        02 STS_RCRD_LOCKED_AVLBL         PIC X(02) VALUE "90".
        02 STS_RCRD_LOCKED_NOT_AVLBL     PIC X(02) VALUE "92".

PROCEDURE DIVISION.
*****************************************************************
DECLARATIVES.

EMPLOYEE-FILE-ERROR     SECTION.

        USE AFTER STANDARD ERROR PROCEDURE ON EMPLOYEE-FILE.

END DECLARATIVES.
*****************************************************************

MAIN-PROCESS SECTION.

BEGIN-PROCESSING-EMPLOYEES.
*
*       Open the employee file for concurrent access.
*
        OPEN I-O EMPLOYEE-FILE ALLOWING ALL.
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.
        MOVE 1                          TO EMPLOYEE-NUMBER.
        DISPLAY "" LINE 1 COLUMN 1 ERASE TO END OF SCREEN.
        PERFORM GET-EMPLOYEE-RECORDS THRU GET-EMPLOYEE-RECORDS-EXIT
                UNTIL EMPLOYEE-NUMBER = 0       OR
                      WS-ERROR-OCCURRED = "Y".
        STOP RUN.

BEGIN-PROCESSING-EMPLOYEES-EXIT.        EXIT.
```

```
GET-EMPLOYEE-RECORDS.

     MOVE ZEROS              TO EMPLOYEE-NUMBER.
     DISPLAY "" AT LINE NUMBER 5 AT COLUMN NUMBER 1 ERASE TO END OF LINE
     DISPLAY "" AT LINE NUMBER 6 AT COLUMN NUMBER 1 ERASE TO END OF LINE
     DISPLAY "EMPLOYEE NUMBER. <CR> OR 0 TO EXIT"
                AT LINE NUMBER 3
                AT COLUMN NUMBER 1
                ERASE TO END OF LINE.
       ACCEPT EMPLOYEE-NUMBER
                FROM LINE NUMBER 3
                FROM COLUMN NUMBER 36
                BOLD
                WITH CONVERSION
       END-ACCEPT.

      IF EMPLOYEE-NUMBER > 0
       THEN READ EMPLOYEE-FILE RECORD
         KEY IS EMPLOYEE-NUMBER
         INVALID KEY DISPLAY "EMPLOYEE WITH THAT NUMBER DOES NOT EXIST.
                             " TRY AGAIN"
                             AT LINE NUMBER 26
                             AT COLUMN NUMBER 3
                             WITH BELL
                             ERASE TO END OF LINE
            END-READ
          IF PROGRAM-IO-STATUS = STS_RCRD_LOCKED_NOT_AVLBL
           THEN DISPLAY "EMPLOYEE WITH THAT NUMBER IS BEING ACCESSED BY
                        " ANOTHER USER. TRY LATER."
                        AT LINE NUMBER 26
                        AT COLUMN NUMBER 3
                        WITH BELL
                        ERASE TO END OF LINE
           ELSE PERFORM CHECK-ERROR-STATUS THRU CHECK-ERROR-STATUS-EXIT
            END-IF

            IF PROGRAM-IO-STATUS = STS_SUCCESS AND
              WS-ERROR-OCCURRED = SPACES
              THEN MOVE EMPLOYEE-RECORD    TO WS-SAVE-EMPLOYEE-RECORD
                      DISPLAY "" AT LINE NUMBER 26
                                 AT COLUMN NUMBER 3
                                 ERASE TO END OF LINE
                     PERFORM MODIFY-EMPLOYEE-DATA
                             THRU MODIFY-EMPLOYEE-DATA-EXIT
              END-IF
       END-IF.

GET-EMPLOYEE-RECORDS-EXIT.      EXIT.

MODIFY-EMPLOYEE-DATA.

       MOVE EMPLOYEE-FIRST-NAME                TO WS-EMPL-FIRST-NAME.
       DISPLAY WS-FIRST-NAME-MSG-LINE
               AT LINE NUMBER  5
               AT COLUMN NUMBER 3
               ERASE TO END OF LINE.
       ACCEPT  WS-EMPL-FIRST-NAME
               FROM LINE NUMBER 5
               FROM COLUMN NUMBER 50
       END-ACCEPT.
       IF WS-EMPL-FIRST-NAME NOT = SPACES
          THEN MOVE WS-EMPL-FIRST-NAME         TO EMPLOYEE-FIRST-NAME.

       MOVE EMPLOYEE-LAST-NAME                 TO WS-EMPL-LAST-NAME.
       DISPLAY WS-LAST-NAME-MSG-LINE
               AT LINE NUMBER  6
               AT COLUMN NUMBER 3
               ERASE TO END OF LINE.
       ACCEPT  WS-EMPL-LAST-NAME
               FROM LINE NUMBER 6
               FROM COLUMN NUMBER 50
       END-ACCEPT.
```

```
        IF WS-EMPL-LAST-NAME NOT = SPACES
           THEN MOVE WS-EMPL-LAST-NAME          TO EMPLOYEE-LAST-NAME.

        IF WS-SAVE-EMPLOYEE-RECORD NOT = EMPLOYEE-RECORD
           THEN REWRITE EMPLOYEE-RECORD
                INVALID KEY DISPLAY "RECORD NOT REWRITTEN. CALL 888-8888
                                    " FOR ASSISTANCE"
                                    AT LINE NUMBER 26
                                    AT COLUMN NUMBER 3
                                    ERASE TO END OF LINE
                END-REWRITE
           ELSE UNLOCK EMPLOYEE-FILE RECORD
        END-IF.
        PERFORM CHECK-ERROR-STATUS      THRU CHECK-ERROR-STATUS-EXIT.

MODIFY-EMPLOYEE-DATA-EXIT.      EXIT.

CHECK-ERROR-STATUS.

        IF PROGRAM-IO-STATUS NOT = STS_SUCCESS
         AND PROGRAM-IO-STATUS NOT = STS_RCRD_LOCKED_AVLBL
         AND PROGRAM-IO-STATUS NOT = STS_DUPLCT_ALTRT_KEY_WRTTN
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_21
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_22
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_23
         AND PROGRAM-IO-STATUS NOT = STS_INVLD_KEY_24
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_10
         AND PROGRAM-IO-STATUS NOT = STS_END_OF_FILE_46 THEN
             MOVE "Y"               TO WS-ERROR-OCCURRED
        END-IF.

CHECK-ERROR-STATUS-EXIT.        EXIT.

CLOSE-FILE.
        CLOSE  EMPLOYEE-FILE.

CLOSE-FILE-EXIT.        EXIT.

END PROGRAM EMPLOYEE.
```

6.2.3 Using Read-Only Global Sections

A disk file Read-Only global section allows multiple processes to read common data elements by using the VMS system service, $CRMPSC, to map a file to an address space in memory. In general, you can use a disk file Read-Only global section for processing I/O without explicitly accessing a disk.

Disk file Read-Only global sections are disk files, or portions of disk files that are mapped into the memory of a **single** VAXcluster CPU. Therefore, only processes on the same VAXcluster CPU can map a disk file global section. When a process calls the $CRMPSC system service to map a disk file Read-Only global section, the service only maps to memory on the VAXcluster CPU on which the process is executing. To map the same disk file Read-Only global section to different processes on different VAXcluster CPUs, you must map the same global section into memory on each VAXcluster CPU.

In order to create a **global** section, you must use the SEC$M_GBL flag for $CRMPSC; otherwise, the default setting for $CRMPSC opens the disk file as a **private** section. Also, when using $CRMPSC with the SEC$M_GBL flag, your program should not attempt to modify the contents of the global section as the results are unpredictable. If your program only needs read access to the file data, using the $CRMPSC with SEC$M_GBL flag will provide a slight performance improvement since the contents of the Read-Only global section do not have to be written back to the disk file when the Read-Only global section is closed.

If the data in the global section is to be modified, the $CRMPSC call should include the SEC$M_GBL flag and the SEC$M_WRT flag to open the section file for both reading or writing. When this type of global section is closed, all of the section data in memory is written back to the file when the global section is deleted. Generally, when implementing a VAXcluster application, it is not advantageous to use Read-Write global sections for concurrent access streams because it is difficult to synchronize global section updates.

If the disk file Read-Only global section is commonly accessed by many programs in the VAXcluster application environment, then the disk file global section can be installed as a known image. The system manager can install a disk file Read-Only global section as a known image on a specific CPU or on all VAXcluster CPUs (depending on whether the VAXcluster system is a common-environment or multiple-environment configuration). By installing a disk file global section as a known image, you can reduce the time required to locate the global section on the disk. When your system manager installs a disk file Read-Only global section as a known image, a pointer is created on the system disk for the location of that file on the disk.[1] For more information on using the $CRMPSC system service to create or map a disk file Read-Only global section, see the *Introduction to VMS System Services* and the *VMS System Services Reference Manual.*

[1] For details on how to create and identify shareable images and how to link them with private object modules, see the *VMS Linker Utility Manual.* For information about how to install shareable images and make them available for sharing as global sections, see the *Guide to Maintaining a VMS System.*

Disk File Read–Only Global Section — Programming Example

In the following example, INIT_GLOBAL.BAS maps a global section as writeable and puts calendar data into the section, and RO_GLOBAL.BAS maps the already existing global section as read-only and reads calendar data from the global section. (A more realistic example would be federal, state, and local tax tables because these would be read frequently but only occasionally updated.)

The macro routines (SECWOPEN.MAR and SECROPEN.MAR) are called by the BASIC file open routines to set the proper RMS attributes and return the internal channel number to the main program. The channel number is needed by the $CRMPSC system service. SECWOPEN.MAR is for the routine that is used by INIT_GLOBAL.BAS to write the section, and SECROPEN.MAR is used by RO_GLOBAL.BAS to read the section.

Use INIT_GLOBAL_BUILD.COM and RO_GLOBAL_BUILD.COM to compile and link the read-only global section example. This example is designed for INIT_GLOBAL.EXE to be run first and then RO_GLOBAL.EXE. **INIT_GLOBAL.EXE will not function properly if RO_GLOBAL.EXE is active.** To coordinate access between these two programs, you can use lock management system services. This is left as an exercise for the reader.

- **INIT_GLOBAL_BUILD.COM**

```
$       !INIT_GLOBAL_BUILD.COM
$
$       BASIC/LIST      INIT_GLOBAL     !Complile main program
$       MACRO/LIST      SECWOPEN        !Assemble user open routine
$                                       !Now link them...
$       LINK/MAP        INIT_GLOBAL,-   ! The main program
                        SECWOPEN,-      ! The user open routine
                        SYS$INPUT/OPT   ! The options file follows this
!This is the OPTIONS file
!  Option to cause the psect/common for the global section to be
!  page aligned.
PSECT_ATTR=CUM_DAYS,PIC,OVR,REL,GBL,SHR,NOEXE,RD,NOWRT,PAGE
$       EXIT    !INIT_GLOBAL_BUILD
```

- **INIT_GLOBAL.BAS**

```
PROGRAM INIT_GLOBAL     ! Initialize a section for the read-only
                        ! global section example.
                        ! Since this program is very similar to
                        ! the one that read the section, only
                        ! those places where they differ have
                        ! been commented.
                        !
%IDENT "V1.00"
%TITLE "INIT GLOBAL"
%SBTTL "Declare and init data"

OPTION  TYPE = EXPLICIT,                        &
        SIZE = INTEGER LONG,                    &
        CONSTANT TYPE = INTEGER

DECLARE LONG    I,                              &
                INADDR(1 TO 2),                 &
                RETADDR(1 TO 2),                &
                FLAGS,                          &
                RETURN_STATUS,                  &
        STRING  FYLE,                           &
                SECTION_NAME                    &

EXTERNAL LONG   FUNCTION                        &
                        SYS$CRMPSC

EXTERNAL LONG   CONSTANT                                    &
                   !Flags (bits) to control                 &
                   ! section creation.                      &
                   SEC$M_DZRO,      !  Demand zero flag       &
                   SEC$M_WRT,       !  Writable section flag  &
                   SEC$M_GBL,                      &
                   SS$_NORMAL

COMMON  (SECWCOM)                               &
                WORD    SEC_CHAN

COMMON  (CUM_DAYS)                              &
                LONG    CUMULATIVE_DAYS(1 TO 12)

FYLE = "DATE_INFO"
SECTION_NAME = "RO_GSECT"
%PAGE
%SBTTL "Open file and create/map to section."
!OPEN file with user open for writeable section
OPEN FYLE FOR OUTPUT AS FILE #2,        &
        ORGANIZATION SEQUENTIAL FIXED,          &
        RECORDSIZE 512,                         &
        RECORDTYPE NONE,                        &
        USEROPEN SECWOPEN,      !User open routine for a  &
                                ! writable section      &
        FILESIZE 1,                             &
        DEFAULTNAME ".DAT"
```

```
        !Create and Map Section
        FLAGS = (SEC$M_GBL    !Global section. Default is private     &
                 OR           !boolean OR of flag bits               &
                 SEC$M_DZRO !Start with zeroed pages, send to section &
                              ! file after they are modified         &
                 OR           !Boolean OR of flag bits               &
                 SEC$M_WRT)   !Writable section. Default is read-only

        INADDR(1) = LOC(CUMULATIVE_DAYS(1))
        INADDR(2) = LOC(CUMULATIVE_DAYS(12))

        RETURN_STATUS = SYS$CRMPSC(                  !  Arg name     &
                                INADDR(1),           !INADR          &
                                RETADDR(1),          !RETADR         &
                                ,                    !ACMODE         &
                                FLAGS BY VALUE,      !FLAGS          &
                                SECTION_NAME,        !GSDNAM         &
                                ,                    !IDENT          &
                                ,                    !RELPAG         &
                                SEC_CHAN BY VALUE,   !CHAN           &
                                1 BY VALUE,          !PAGCNT         &
                                ,                    !VBN            &
                                ,                    !PROT           &
                                ,)                   !PFC

        IF (RETURN_STATUS AND 1) = 0 THEN
                PRINT "Create section failure."
                PRINT "Status =";RETURN_STATUS
                GOTO ABNORMAL_EXIT
        ELSE
                PRINT "Section created."
        END IF
        %PAGE
        %SBTTL "Do something with the section."
        !Initialize the array and, therefore, the section
        !   (Leap years are ignored)
        CUMULATIVE_DAYS(1) = 0                  !No days before Jan
        CUMULATIVE_DAYS(2) = CUMULATIVE_DAYS(1) + 31 !Days in Jan
        CUMULATIVE_DAYS(3) = CUMULATIVE_DAYS(2) + 28 !Days in Feb
        CUMULATIVE_DAYS(4) = CUMULATIVE_DAYS(3) + 31 !Days in Mar
        CUMULATIVE_DAYS(5) = CUMULATIVE_DAYS(4) + 30 !Days in Apr
        CUMULATIVE_DAYS(6) = CUMULATIVE_DAYS(5) + 31 !Days in May
        CUMULATIVE_DAYS(7) = CUMULATIVE_DAYS(6) + 30 !Days in Jun
        CUMULATIVE_DAYS(8) = CUMULATIVE_DAYS(7) + 31 !Days in Jul
        CUMULATIVE_DAYS(9) = CUMULATIVE_DAYS(8) + 31 !Days in Aug
        CUMULATIVE_DAYS(10)= CUMULATIVE_DAYS(9) + 30 !Days in Sep
        CUMULATIVE_DAYS(11)= CUMULATIVE_DAYS(10)+ 31 !Days in Oct
        CUMULATIVE_DAYS(12)= CUMULATIVE_DAYS(11)+ 30 !Days in Nov

        PRINT CUMULATIVE_DAYS(I) FOR I = 1 TO 12     !Sanity check
        !Exits
NORMAL_EXIT:
ABNORMAL_EXIT:

        CLOSE 2
END PROGRAM
```

- **SECWOPEN.MAR**

```
        .TITLE SECWOPEN
        ;  This is a routine called while a file is being opened
        ;  for use as the backing file for a writable global
        ;  section. This routine is similar to the read-only
        ;  useropen routine and is documented only where they
        ;  differ.
        ;
        .IDENT /V1.00/
;
        .PSECT SECWCOM  PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG
                                ;This PSECT name must match
                                ; the COMMON name in BASIC program
SEC_CHAN:       .BLKW   1
;
;
        .PSECT CODE     EXE,NOWRT,LONG
        .ENTRY  SECWOPEN, ^M<R2>
                                ;Entry point must match
                                ; USEROPEN clause in BASIC program

        MOVL    4(AP) , R2

        INSV    #1, #FAB$V_CIF,-        ;Create if not there
                #1, FAB$L_FOP(R2)

        INSV    #1, #FAB$V_CTG,-
                #1, FAB$L_FOP(R2)

        INSV    #1, #FAB$V_UFO,-
                #1, FAB$L_FOP(R2)

        INSV    #1, #FAB$V_UPI,-
                #1, FAB$B_SHR(R2)


        $CREATE FAB = (R2)              ;Call RMS to create the file

        BLBC    R0, 10$

        MOVW    FAB$L_STV(R2), SEC_CHAN

        MOVL    #SS$_NORMAL, R0

10$:    RET
        .END                            ;SECWOPEN user open routine.
```

- **RO_GLOBAL_BUILD.COM**

```
$       !RO_GLOBAL_BUILD.COM
$
$       BASIC/LIST      RO_GLOBAL       !Complile main program
$       MACRO/LIST      SECROPEN        !Assemble user open routine
$                                       !Now link them...
$       LINK/MAP        RO_GLOBAL,-     ! The main program
                        SECROPEN, -     ! The user open routine
                        SYS$INPUT/OPT   ! The options file follows this
!This is the OPTIONS file.
!  Option to cause the psect/common for the global section
!  to be page aligned.
PSECT_ATTR=CUM_DAYS,PIC,OVR,REL,GBL,SHR,NOEXE,RD,NOWRT,PAGE
$       EXIT    !RO_GLOBAL_BUILD
```

- **RO_GLOBAL.BAS**

```
PROGRAM RO_GLOBAL       !Read-only global section example
%IDENT "V1.00"
%TITLE "R/O GLOBAL"
%SBTTL "Declare and init data"

OPTION  TYPE = EXPLICIT,                          &
        SIZE = INTEGER LONG,                      &
        CONSTANT TYPE = INTEGER

DECLARE LONG    DAY,            !Day of month from user      &
                MONTH,          !Month from user             &
                I,              !Loop index                  &
                INADDR(1 TO 2), !Desired section boundaries  &
                RETADDR(1 TO 2),!Actual section boundaries   &
                FLAGS,          !Flags for section creation  &
                RETURN_STATUS, !System Service return status  &
        STRING  FYLE,           !Filename to map section to  &
                SECTION_NAME    !Name for section            &

EXTERNAL LONG   FUNCTION SYS$CRMPSC     !Create and Map Section
                                        ! System Service

EXTERNAL LONG   CONSTANT                          &
                      SEC$M_GBL,    !Global section flag  &
                      SS$_NORMAL    !System Service success value

!Common area to communicate with USEROPEN routine.
! The name of the common area must match the PSECT name in
! the USEROPEN routine.
!
COMMON  (SECRCOM)                                 &
                WORD    SEC_CHAN

!Map the date array into a common area.
! This will get mapped to a private section.
!
COMMON  (CUM_DAYS)                                &
                LONG    CUMULATIVE_DAYS(1 TO 12)
                        !Days in the year through preceding
                        ! month.

FYLE = "DATE_INFO"
SECTION_NAME = "RO_GSECT"
%PAGE
%SBTTL "Open file and create/map to section."
!OPEN the section file specifying a user open routine.
!
OPEN FYLE FOR INPUT AS FILE #2, &
        ORGANIZATION SEQUENTIAL FIXED,            &
        RECORDSIZE 512,                           &
        RECORDTYPE NONE,                          &
        USEROPEN SECROPEN,    !This name must match the       &
                              ! .ENTRY name in the USEROPEN   &
                              ! routine.                      &
        DEFAULTNAME ".DAT"
```

```
        !Create and map the section.
        !
        FLAGS = (SEC$M_GBL)      !Make it a global section. (Private
                                 ! section is the default.)

        INADDR(1) = LOC(CUMULATIVE_DAYS(1))    !Longword containing the
                                               ! desired starting
                                               ! address of the section.
        INADDR(2) = LOC(CUMULATIVE_DAYS(12))   !Longword containing the
                                               ! desired ending address
                                               ! of the section.

        RETURN_STATUS = SYS$CRMPSC(                    ! Arg Name      &
                                INADDR(1)              !inadr          &
                                RETADDR(1),            !retadr         &
                                ,                      !acmode         &
                                FLAGS BY VALUE,        !flags          &
                                SECTION_NAME,          !gsdnam         &
                                ,                      !ident          &
                                ,                      !relpag         &
                                SEC_CHAN BY VALUE,     !chan           &
                                1 BY VALUE,            !pagcnt         &
                                ,                      !vbn            &
                                ,                      !prot           &
                                ,)                     !pfc

        IF (RETURN_STATUS AND 1) = 0 THEN  !Always check return status.
                PRINT "Create section failure."
                PRINT "Status =";RETURN_STATUS
                GOTO ABNORMAL_EXIT
        ELSE
                PRINT "Section created."        !Sanity check
        END IF

        !In case we should need to debug this, remove the comment
        ! character from the next line.
        !PRINT CUMULATIVE_DAYS(I) FOR I = 1 TO 12

        %PAGE
        %SBTTL "Do something with the section."
        !Read and use the information from the section.
        !
READ_LOOP:
        !  Get information from the user.  The user is assumed to be
        !  a perfect typist, familiar with the Gregorian calendar and
        !  rationale.
        !
        PRINT
        INPUT "Month (number) in question"; MONTH
        GOTO NORMAL_EXIT IF MONTH = 0   !Escape route

        INPUT "Day in question";DAY

        PRINT
        PRINT USING "  <0>#/<0>#  is day number ###.",             &
                MONTH,DAY,CUMULATIVE_DAYS(MONTH)+DAY
                ! This will print the answer in the form
                !  "mm/dd is day number nnn."
        PRINT "-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-"

        GOTO READ_LOOP

        !Exits
NORMAL_EXIT:
ABNORMAL_EXIT:
        CLOSE 2
END PROGRAM
```

- **SECROPEN.MAR**

```
        .TITLE SECROPEN
        ;  This is a routine called while a file is being opened
        ;  for use a the backing file for a read-only global
        ;  section. It allows us to modify the way RMS will handle
        ;  the file.  It also allows access to the internal
        ;  channel number that would normally not be available
        ;  to a high level language.
        ;
        .IDENT /V1.00/
;
        .PSECT SECRCOM  PIC,OVR,REL,GBL,SHR,NOEXE,RD,WRT,LONG
                                        ;This PSECT name must match
                                        ;COMMON name in BASIC program.
SEC_CHAN:       .BLKW   1
;
;
        .PSECT CODE     EXE,NOWRT,LONG

        .ENTRY  SECROPEN, ^M<R2>
                                ;Entry point name must match name
                                ; in USEROPEN clause in BASIC program.

        MOVL    4(AP) , R2              ;Fetch FAB address

        INSV    #1, #FAB$V_CTG,-        ;Contiguous
                #1, FAB$L_FOP(R2)

        INSV    #1, #FAB$V_UFO,-        ;User File Open
                #1, FAB$L_FOP(R2)

        INSV    #1, #FAB$V_UPI,-        ;User Provided Interlocking
                #1, FAB$B_SHR(R2)

        $OPEN   FAB = (R2)          ;Call RMS to open the file.
                                    ; R0 will contain completion status.

        BLBC    R0, 10$                 ;If error, return with it.

        MOVW    FAB$L_STV(R2), SEC_CHAN ;Store channel number assigned
                                        ; by RMS where mainline code
                                        ; can find it.

        MOVL    #SS$_NORMAL, R0         ;Return success

10$:    RET

        .END                            ;SECROPEN user open routine.
```

6.3 Process Synchronization

When designing and implementing VAXcluster software, process synchronization is critical for applications that have been decomposed to execute discrete functional units of work on multiple VAXcluster CPUs. When coordinating two or more processes to accomplish one function, the programmer must use the programming technique of process synchronization. Most interprocess communication requirements for process synchronization can be implemented by using the following VMS programming tools:

- Clusterwide process services
- Lock management system services
- DECnet-VAX communications

These techniques are described in the following sections of this manual.[2]

6.3.1 Using Clusterwide Process Services

You can use the following process control system service calls to synchronize process execution across VAXcluster node boundaries:

- $CANWAK — cancel wakeup
- $DELPRC — delete process
- $FORCEX — force image exit
- $RESUME — resume process
- $SCHDWK — schedule wake for process
- $SETPRI — set priority
- $SUSPND — suspend process
- $WAKE — wake process

In addition to these process control system service calls, some of these system services are supported from DCL commands. For more information on using the process control system services, see the *VMS System Services Reference Manual*, the *VMS DCL Dictionary*, and the *VMS Version 5.2 New Features Manual*.

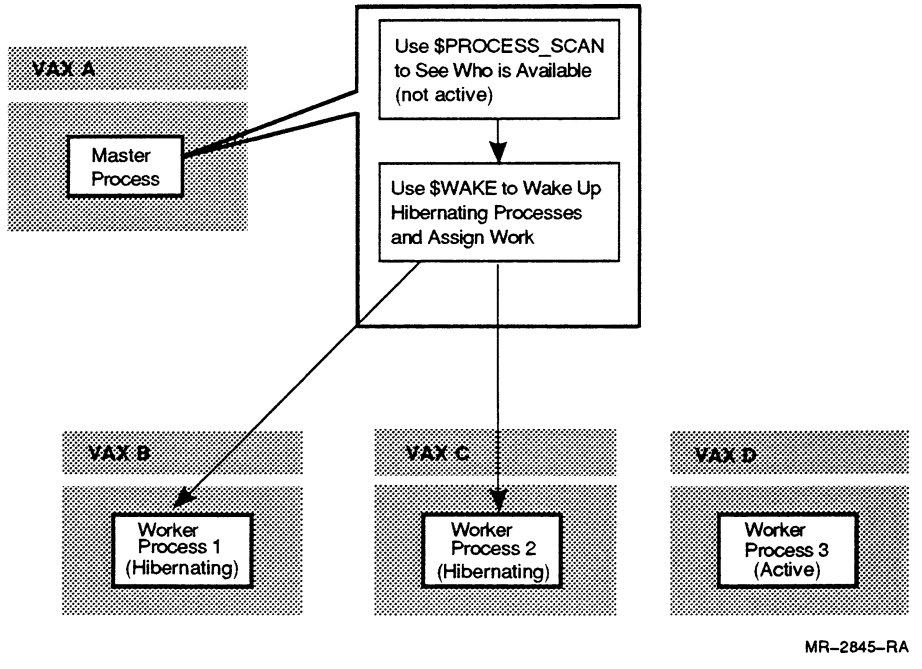
Also, you can obtain clusterwide process information by using the process information system services: $GETJPI or $PROCESS_SCAN with $GETJPI. From DCL, you can use the F$GETJPI or F$CONTEXT lexical functions. For more information on using the process information system services, see the *VMS System Services Reference Manual*, the *VMS DCL Dictionary*, and the *VMS Version 5.2 New Features Manual*.

[2] Lock management system services for process synchronization are slightly faster than DECnet techniques, but the implementation of complex synchronizations using the VMS lock manager may be more difficult to maintain. Conversely, if an application is being designed to run on a VAXcluster system **and** a network, use DECnet for process synchronizations because lock management system services and clusterwide process services are cluster-specific.

Note: When you use either of the process control or process information system services across VAXcluster nodes, the UIC-based GROUP/WORLD privileges are processed exactly as they would be on a single node. You are not allowed to do anything to a process on another node unless you could do it on a local node.

You can design an application that coordinates the use of the process control system services with the process information system services. In Figure 6–2, a Master Process is using $PROCESS_SCAN with $GETJPI to determine the process state of a selected group of Worker Processes. After the eligible (non-active) processes are identified, the Master Process can assign work and then use the appropriate process control system service to wake the Worker Processes.

Figure 6–2 Coordinating the Use of Process Control System Services with the Process Information System Services



Process Control System Services — Programming Example 1

In this example, MASTER.COB is run on one VAXcluster CPU and SLAVE.COB on another. Either MASTER.EXE or SLAVE.EXE can be run first. If MASTER.EXE is run first, work requests to add two numbers are stored in a file, and when SLAVE.EXE runs, the requests from the file are processed. If SLAVE.EXE runs first, there will be no requests in the work file, and SLAVE.EXE will hibernate until MASTER.EXE places a work request in the work file and issues a wake-up (SYS$WAKE) to SLAVE.EXE.

- **MASTER.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    APPL_MASTER.
*
*   This is the 'master' program of this example application. In this
*   program we collect two numbers to be added by our slave.  We put
*   the numbers in a file and wake up the slave.
*
*   With VMS V5.2 the services used here ($HIBER and $WAKE) have been
*   extended to allow the master and slave to be on different nodes in
*   the cluster.  We use the $PROCESS_SCAN service to find the slave.
*   If the slave is not working now we can still collect numbers to be
*   added later.
*-
*
*  Written in VAX COBOL V4.2-41, under VMS V5.2
*
*  To link this program you must include the symbol definitions from
*  the $PSCANDEF macro.  Put the following lines in a small MACRO
*  program, and link the resulting object file with this program.
*
*      $pscandef GLOBAL
*      .end
*-

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

* The work-file is declared as optional so that it can be created by
* the open statement if it does not already exist.

    SELECT OPTIONAL WORK-FILE  ASSIGN       TO "APPL_WORK_FILE"
                               ORGANIZATION IS INDEXED
                               ACCESS       IS DYNAMIC
                               RECORD KEY   IS WORK-TIMESTAMP
                               FILE STATUS  IS FILE-STATUS.

DATA DIVISION.
FILE SECTION.

FD WORK-FILE.
01  WORK_RECORD.
    02  WORK-TIMESTAMP    PIC  S9(18) COMP.
    02  WORK-SLAVE-NAME   PIC  X(15).
    02  WORK-FIRST-NO     PIC  9(09).
    02  WORK-SECOND-NO    PIC  9(09).
    02  WORK-SUM-TOTAL    PIC  9(09).

WORKING-STORAGE SECTION.

01 RETURN-STATUS          PIC S9(09) COMP  VALUE ZERO.
   88 SS$_NORMAL                           VALUE EXTERNAL SS$_NORMAL.
   88 SS$_NOMOREPROC                       VALUE EXTERNAL SS$_NOMOREPROC.

01 FILE-STATUS            PIC  X(02)       VALUE SPACES.

01 IS-WORK-FILE-OPEN      PIC  9(01)       VALUE 0.
   88 WORK-FILE-NOT-OPEN                   VALUE 0.
   88 WORK-FILE-IS-OPEN                    VALUE 1.

01 WS-SLAVE-NAME          PIC  X(15)       VALUE "APPL_SLAVE     ".
01 WS-ENTERED-NAME        PIC  X(15)       VALUE SPACE.

01 WS-NODE-WILDCARD       PIC  X(01)       VALUE "*".
01 WS-PID-CONTEXT         PIC S9(09) COMP  VALUE ZERO.
```

```
01 WS-PSCAN-ITMLST.
   02 BUF-LEN        PIC  9(04) COMP  VALUE 1.
   02 ITM-CODE       PIC  9(04) COMP  VALUE EXTERNAL  PSCAN$_NODENAME.
   02 PTR            USAGE IS POINTER VALUE REFERENCE WS-NODE-WILDCARD.
   02 ITM-FLAGS      PIC  9(09) COMP  VALUE EXTERNAL  PSCAN$M_WILDCARD.

   02 BUF-LEN        PIC  9(04) COMP  VALUE 10.
   02 ITM-CODE       PIC  9(04) COMP  VALUE EXTERNAL  PSCAN$_PRCNAM.
   02 PTR            USAGE IS POINTER VALUE REFERENCE WS-SLAVE-NAME.
   02 ITM-FLAGS      PIC  9(09) COMP  VALUE EXTERNAL  PSCAN$M_CASE_BLIND.

   02 TERMINATOR          PIC S9(04) COMP  VALUE ZERO.

01 WS-JPI-EVENT-FLAG      PIC S9(09) COMP  VALUE ZERO.
01 WS-JPI-IOSB            PIC S9(18) COMP  VALUE ZERO.
01 WS-SLAVE-PID           PIC S9(09) COMP  VALUE ZERO.

01 WS-JPI-ITMLST.
   02 BUF-LEN             PIC S9(04) COMP  VALUE 4.
   02 ITM-CODE            PIC S9(04) COMP  VALUE EXTERNAL  JPI$_PID.
   02 PTR                 USAGE IS POINTER VALUE REFERENCE WS-SLAVE-PID.
   02 ITM-LEN             PIC S9(09) COMP  VALUE ZERO.

   02 TERMINATOR          PIC S9(04) COMP  VALUE ZERO.

01 WHAT-TO-DO             PIC 9(01)        VALUE 0.
   88 THE-OPERATOR-SAYS-QUIT               VALUE 1.
   88 THE-COWS-ARE-HOME                    VALUE 1.

PROCEDURE DIVISION.
000-CONTROL.
*+
*  Before we start gathering work we ask for the name of the slave
*  and try to find if the slave is available for work.
*-
        PERFORM 500-TRY-TO-FIND-SLAVE THRU 500-END.
*+
*  This is the main control loop. Here we start the process of
*  collecting numbers.
*-
        PERFORM 050-LOOP THRU 050-END  UNTIL THE-OPERATOR-SAYS-QUIT.
*+
*  The user is finished, so close the file if it was opened and exit.
*-
        IF WORK-FILE-IS-OPEN THEN CLOSE WORK-FILE.
        STOP RUN.

000-END.

050-LOOP.
*+
*  Here's the main logic of the program.  We get two numbers from the
*  user, then we get the system time for a unique key, write the
*  record in the data file, and then tell the slave to get busy adding
*  the numbers.
*-
        PERFORM    100-GET-NUMBERS   THRU 100-END.

        IF THE-OPERATOR-SAYS-QUIT
           THEN NEXT SENTENCE
           ELSE PERFORM    200-GET-TIME-NAME THRU 200-END
                PERFORM    300-WRITE-RECORD  THRU 300-END
                PERFORM    400-WAKE-SLAVE    THRU 400-END.

050-END.
```

```
100-GET-NUMBERS.
*+
*  Here we prepare the work. This involves getting two numbers that
*  are to be added together from the user.  If the user enters zero
*  for both numbers, then the program will exit.
*-
        INITIALIZE WORK_RECORD.

*+
*  Display a brief header for the user to read....
*-
   DISPLAY  " ".
   DISPLAY  "  This is the master program that will ask you for two  ".
   DISPLAY  "  numbers to be added together by the slave.     ".
   DISPLAY  "  (To exit enter 0 for both numbers.)                   ".
   DISPLAY  " ".

*+
*  Get the first number.
*-
        DISPLAY  " ".
        DISPLAY  " Enter the first number : " WITH NO ADVANCING.
        ACCEPT WORK-FIRST-NO WITH CONVERSION.

*+
*  Get the second number.
*-
        DISPLAY  " ".
        DISPLAY  " Enter the second number : " WITH NO ADVANCING.
        ACCEPT WORK-SECOND-NO WITH CONVERSION.

*+
*  If the user enters zero for both numbers, then we stop asking
*  questions and exit from the program.
*-
        IF (WORK-FIRST-NO) = 0 AND (WORK-SECOND-NO = 0)
           THEN SET THE-OPERATOR-SAYS-QUIT TO TRUE.

100-END.

200-GET-TIME-NAME.
*+
*  If the user signaled that we should write a shutdown record, then
*  move a large number in to the timestamp field, other wise use the
*  system time. This ensures for at least a few more years that the
*  shutdown record will be the last record in the file.
*-
        CALL "SYS$GETTIM"    USING  BY REFERENCE WORK-TIMESTAMP
                             GIVING RETURN-STATUS
        IF NOT SS$_NORMAL
           THEN DISPLAY "..Error getting system time...exiting.."
                CALL "LIB$STOP" USING BY VALUE RETURN-STATUS.
*+
*   Insert the name of the slave to do the work into the record.
*-
        MOVE WS-SLAVE-NAME TO WORK-SLAVE-NAME.

200-END.

300-WRITE-RECORD.
*+
*  Here we go to write the collected data into the file. If the file
*  has not been opened, then open it and note that it is open.
*-

        IF WORK-FILE-NOT-OPEN
           THEN OPEN I-O WORK-FILE ALLOWING ALL
                SET WORK-FILE-IS-OPEN TO TRUE.
```

```
*+
*  Since we are using the clock time for a key, we should seldom get
*  an invalid condition here, but if we do, send a nice message and
*  exit.
*-
        WRITE WORK-RECORD
              INVALID DISPLAY "Invalid key (Status=" FILE-STATUS
                              ") on write of WORK-RECORD ...exiting"
                      CALL "LIB$STOP" USING BY VALUE RETURN-STATUS.
300-END.

400-WAKE-SLAVE.
*+
*  Here we call the $WAKE service to let the slave know that there
*  is work to be done. We can use process name because we know that the
*  slave is in the same UIC group as the calling program.
*-
        CALL "SYS$WAKE"   USING  BY REFERENCE WS-SLAVE-PID,
                                 OMITTED,
                          GIVING RETURN-STATUS.

*+
*  Here we check to see if the slave has woken properly. If not, then
*  we display a message.
*-
    IF NOT SS$_NORMAL
      THEN DISPLAY "Slave must have died....work will not be performed "
           DISPLAY " until another slave is started.".

400-END.

500-TRY-TO-FIND-SLAVE.
*+
*  Here we ask the user the name of the slave to do the work.  The
*  default offerred is the same as the default name for the slave. If
*  the user enters nothing, then we use the default value.
*-

        DISPLAY "Master process starting...."
        DISPLAY "Enter name of slave [APPL_SLAVE] : " WITH NO ADVANCING

        ACCEPT WS-ENTERED-NAME PROTECTED WITH CONVERSION

        IF WS-ENTERED-NAME NOT = SPACES
           THEN MOVE WS-ENTERED-NAME TO WS-SLAVE-NAME.

*+
*  Now we search for the slave process using the $PROCESS_SCAN service
*  to establish a context for the following $GETJPI call. The node is
*  wildcarded to search for the slave on any node in the cluster.
*-
        CALL "SYS$PROCESS_SCAN" USING  BY REFERENCE WS-PID-CONTEXT
                                       BY REFERENCE WS-PSCAN-ITMLST
                                GIVING RETURN-STATUS.

       IF NOT SS$_NORMAL
          THEN DISPLAY "..Error setting procscan context....exiting.."
               CALL "LIB$STOP" USING BY VALUE RETURN-STATUS.
```

```
*+
*  Here we issue a $GETJPI call to return the PID of the slave which
*  will be unique across the cluster.  We will later use the PID in
*  the calls to $WAKE when we put the slave to work.
*-
        CALL "SYS$GETJPIW"      USING  BY VALUE     WS-JPI-EVENT-FLAG
                                       BY REFERENCE WS-PID-CONTEXT
                                       OMITTED
                                       BY REFERENCE WS-JPI-ITMLST
                                       BY REFERENCE WS-JPI-IOSB
                                       OMITTED
                                       OMITTED
                                GIVING RETURN-STATUS.

     IF SS$_NOMOREPROC
     THEN DISPLAY "..Slave process was not found...." WS-SLAVE-NAME
          DISPLAY "  work will be performed when slave is started.... "
*+
*  The GETJPI completes asynchronously so we MUST use the $SYNCH
*  service to ensure proper completion of the JPI call.
*-
        CALL "SYS$SYNCH"        USING  BY VALUE     WS-JPI-EVENT-FLAG
                                       BY REFERENCE WS-JPI-IOSB
                                GIVING RETURN-STATUS.

       IF NOT SS$_NORMAL
          THEN DISPLAY "..Error getting proc information ....exiting.."
               CALL "LIB$STOP" USING BY VALUE RETURN-STATUS.

500-END.
```

- **SLAVE.COB**

```
IDENTIFICATION DIVISION.
PROGRAM-ID.    APPL_SLAVE.
*+
*  This is the slave program that performs the work of addition in
*  our conceptual application.  Once the master has collected the two
*  numbers to be added and written them into a file, the slave will
*  read the file and add the numbers together and display the result
*  on the screen.
*
*  The program performs the work and then HIBERnates again, waiting
*  for another opportunity to serve its master.  When the slave starts
*  work we must check to see if there is any work outstanding, and the
*  slave must not stop working until all the work is done.
*-
*
*  Written in VAX COBOL V4.2-41, under VMS V5.2
*
*-

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

* The work-file is listed as optional so the open will create
* an empty file if one does not already exist.

    SELECT OPTIONAL WORK-FILE  ASSIGN       TO "APPL_WORK_FILE"
                               ORGANIZATION IS INDEXED
                               ACCESS       IS DYNAMIC
                               RECORD KEY   IS WORK-TIMESTAMP.
```

```
DATA DIVISION.
FILE SECTION.

FD WORK-FILE.
01  WORK_RECORD.
    02  WORK-TIMESTAMP     PIC  S9(18) COMP.
    02  WORK-SLAVE-NAME    PIC  X(15).
    02  WORK-FIRST-NO      PIC  9(09).
    02  WORK-SECOND-NO     PIC  9(09).
    02  WORK-SUM-TOTAL     PIC  9(09).

WORKING-STORAGE SECTION.
01 RETURN-STATUS           PIC S9(09) COMP    VALUE ZERO.
   88 SS$_NORMAL                              VALUE EXTERNAL SS$_NORMAL.

01 WORK-FILE-POINTER       PIC  9(02)         VALUE ZERO.
   88 NOT-END-OF-FILE                         VALUE 0.
   88 END-OF-FILE                             VALUE 2.

01 IS-WORK-FILE-EMPTY      PIC 9(01)          VALUE 0.
   88 WORK-FILE-NOT-EMPTY                     VALUE 0.
   88 WORK-FILE-IS-EMPTY                      VALUE 1.

01 WS-PROCESS-NAME         PIC X(15)          VALUE "APPL_SLAVE     ".
01 WS-ENTERED-NAME         PIC X(15)          VALUE SPACES.
01 WS-LEN-CTR              PIC 9(09)  COMP    VALUE ZERO.

01 WHERE-ARE-THE-COWS      PIC 9(02)          VALUE ZERO.
   88 THE-COWS-COME-HOME                      VALUE 1.
   88 THE-COWS-ARE-HOME                       VALUE 1.

PROCEDURE DIVISION.
000-CONTROL.

*+
*  Set the name of the slave to whatever the user enters as the
*  default.
*-
        PERFORM 300-SET-PROCESS-NAME THRU 300-END.

*+
*   Open the data file.
*-
        OPEN I-O WORK-FILE ALLOWING ALL
*+
*   Here we start the main processing loop, basically an infinite loop.
*   For the purposes of this example, the slave must be interrupted by
*   control-Y.  A proper application should have some way of sending a
*   message to the slave to cause a normal shutdown.
*-
    PERFORM 100-WORK-SLEEP-LOOP  THRU 100-END UNTIL THE-COWS-COME-HOME.
    STOP RUN.

000-END.

100-WORK-SLEEP-LOOP.
*+
*   This is the main processing loop for the program. Here we start the
*   file pointer at the beginning and loop thru reading the file and
*   adding records, and when all the work is done we take a break.
*-
```

```
*+
*  Set file pointer to first record in file.  We assume there is
*  something in the file and set the pointer. If the operation fails,
*  the file must be empty.
*-
        SET WORK-FILE-NOT-EMPTY TO TRUE.
        SET NOT-END-OF-FILE     TO TRUE.
        INITIALIZE WORK-RECORD.

        START WORK-FILE KEY NOT LESS THAN WORK-TIMESTAMP
            INVALID KEY DISPLAY "File is empty....waiting for work..."
                        SET work-FILE-IS-EMPTY TO TRUE.
*+
*  If the file is not empty, then we will loop through it to see if
*  there is any work to be done.
*-
        IF work-FILE-NOT-EMPTY
           THEN SET BEGINNING-OF-FILE TO TRUE
                PERFORM 200-READ-LOOP THRU 200-END UNTIL END-OF-FILE.

*+
*  All the work has been done, so we can take a break. This section of
*  the program will place the slave process in a hibernate state waiting
*  to be awakened by the master.  We can only take a short break and must
*  be ready when the master has more work for us....we go to sleep using
*  the $HIBER service.
*-
        DISPLAY "Sleeping.........".
        CALL "SYS$HIBER"  GIVING RETURN-STATUS.

100-END.

200-READ-LOOP.
*+
*  This is the read loop that processes the data file and adds the
*  fields in the records displaying the work. We then delete the record
*  from the file.  The addition should probably have some size/error
*  control in order to handle overflow conditions better.
*-

*+
*  Read the next record. If no more records, we can take a break.
*-
        READ WORK-FILE NEXT RECORD
             AT END SET END-OF-FILE TO TRUE
                    GO TO 200-END.
*+
*  Just set this....we are no longer at the beginning or end of
*  the file.
*-
        SET MIDDLE-OF-FILE TO TRUE.
*+
*  Check record to see if this is our work and add them if it is...
*-
        IF (WS-PROCESS-NAME  = WORK-SLAVE-NAME)
         THEN ADD WORK-FIRST-NO TO WORK-SECOND-NO GIVING WORK-SUM-TOTAL

              DISPLAY "Adding........ " WORK-FIRST-NO  WITH CONVERSION
                      " + "             WORK-SECOND-NO WITH CONVERSION
                      " = "             WORK-SUM-TOTAL WITH CONVERSION

              DELETE WORK-FILE
                     INVALID KEY DISPLAY "Invalid on delete...".

200-END.
```

```
300-SET-PROCESS-NAME.
*+
*  Here we set our process name to whatever the user enters, or the
*  default if nothing is entered.  The name must be unique so if the
*  task fails, we exit the program.
*-
      DISPLAY "Slave processing starting.....".
      DISPLAY "Enter name for slave [APPL_SLAVE] : " WITH NO ADVANCING

        ACCEPT WS-ENTERED-NAME PROTECTED WITH CONVERSION
        IF WS-ENTERED-NAME NOT = SPACES
           THEN MOVE WS-ENTERED-NAME TO WS-PROCESS-NAME.

        INSPECT WS-PROCESS-NAME TALLYING WS-LEN-CTR
                FOR CHARACTERS BEFORE INITIAL SPACE.
*+
*   Here we set our process name.  This must be unique or we must exit.
*-
  CALL "SYS$SETPRN" USING   BY DESCRIPTOR WS-PROCESS-NAME(1:WS-LEN-CTR)
                    GIVING  RETURN-STATUS.

      IF NOT SS$_NORMAL
         THEN DISPLAY "Already a slave by this name....exiting."
              CALL "LIB$STOP" USING BY VALUE RETURN-STATUS.

300-END.

END PROGRAM APPL_SLAVE.
```

Process Information System Services — Programming Example 2

- **PSCAN.BAS**

```
on error goto error_routine
! Program name  -- pscan.bas
! Author        -- Digital
! Date          -- 11-May-89
! BASIC Version -- 3.3
! VMS Version   -- 5.2 or higher

! Program Description

! This program demonstrates the use of the $PROCESS_SCAN and
! $GETJPI system service calls to obtain information about
! processes on all nodes in a VAXcluster system. Process
! information is obtained and displayed on the terminal.

! Compile/link Instructions

! In order to obtain the process scan definitions, create the
! following PSCANDEF.MAR macro program:
!
!             $pscandef GLOBAL
!             .END
!
!     $MACRO PSCANDEF.MAR
!     $BASIC PSCAN.BAS
!     $LINK  PSCAN, PSCANDEF
!

! Modification Log

! Date        Name            Modification
! ----        ----            ------------
! 11-May-89   Digital         Initial Release
```

```
! External References

     external long function sys$getjpiw(long by value,            &
                                        long BY REF,,             &
                                        item_list by REF,         &
                                        iosb_record BY REF,,),    &
                            sys$process_scan(long BY REF,         &
                                             pscan_descriptors by REF), &
                            ots$cvt_l_tz(Long BY REF,             &
                                         string BY DESC,          &
                                         long BY VALUE,),         &
                            sys$synch(long BY VALUE,              &
                                      iosb_record BY REF),        &
                             lib$stop(long by value,,,,,)


      external long constant ss$_normal,      &
                             ss$_nomoreproc,  &
                             ss$_nopriv,      &
                             jpi$_nodename,   &
                             jpi$_username,   &
                             jpi$_prcnam,     &
                             jpi$_pid,        &
                             jpi$_imagname,   &
                             jpi$_nodename,   &
                             pscan$_nodename, &
                             pscan$_getjpi_buffer_size, &
                             pscan$m_wildcard

! special datatypes

record item_list
   group item (5)
      variant
          case
              word buffer_length
              word item_code
              long buffer_address
              long length_address
          case
              long list_terminator
      end variant
   end group item
end record

  declare item_list           getjpi_item_list

record pscan_descriptors
   group pscan_item (2)
      variant
          case
              word buffer_length
              word item_code
              long buffaddr_itemval
              long item_spec_flags
          case
              long list_terminator
      end variant
   end group pscan_item
end record
  declare pscan_descriptors pscan_descriptor_list
```

```
record print_record_record
      variant
        case
              string nodename   =  6
              string pid        =  8
              string username   = 12
              string procname   = 15
              string imagename  = 30
        case
              string all_data   = 71%
      end variant
end record
  declare print_record_record   print_record

record iosb_record
              word iosb_field(1 to 4)
end record
  declare iosb_record           iosb

! variable declaration section

  declare word                  getjpi_username_length, &
                                getjpi_prcname_length,  &
                                getjpi_pid_length,      &
                                getjpi_imagename_length,&
                                getjpi_nodename_length, &
                                pscan_nodename_length

  declare long                  pid_context,            &
                                event_flag,             &
                                getjpi_status,          &
                                synch_status,           &
                                pscan_status,           &
                                ots_status,             &
                                getjpi_pid,             &
                                imagname_location,      &
                                nodename_location,      &
                                print_array_subscript,  &
                                array_subscript,        &
                                process_to_print,       &
                                process_count

  map (getjpi_user) string      node_wildcard    =  1%, &
                                getjpi_username  = 12%, &
                                getjpi_prcname   = 15%, &
                                getjpi_imagename = 80%, &
                                getjpi_nodename  =  6%, &
                                ots_character_string = 8%

  dim integer                   nodename_ascii_array(6%)

! constant declaration section

  declare long   constant       pscan_getjpi_buffer_size = 800%

  declare word   constant       getjpi_username_buff  = 12%,    &
                                getjpi_prcname_buff   = 15%,    &
                                getjpi_pid_buff       =  4%,    &
                                getjpi_imagename_buff = 64%,    &
                                getjpi_nodename_buff  =  6%,    &
                                ots_number_of_digits  =  8%
```

```
declare string constant title_heading = "VAXcluster Process Scanner", &
                    top_of_screen = "",                &
                    header_one = "    NODE",                          &
                    header_two = "    NAME      PID        USERNAME"  &
  + "      PROCESS NAME            IMAGE NAME          ",             &
                  print_line = "  'LLLLL   'LLLLLLL   'LLLLLLLLLL" &
  + "  'LLLLLLLLLLLLL 'LLLLLLLLLLLLLLLLLLLLLLLLL",                    &
                dashed_line_80 = "------------------------------"    &
  + "----------------------------------------------",                 &
                    center_80  = "'CCCCCCCCCCCCCCCCCCCCCCCCCCCCC"     &
  + "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC"

 !*********************************************************************
 ! program control section
 !
   gosub build_item_lists
   gosub call_process_scan
   gosub call_getjpi
   gosub display_process_info
   goto exit_program

 ! This routine builds the item lists for both $getjpi and
 ! $process_scan. The process information returned by $getjpi includes:
 ! username, process name, process identification (PID), image name,
 ! and node name.
 !
 ! Control information passed to $process_scan includes a wildcard
 ! directive and a process information buffer. The $getjpi call
 ! utilizes this buffer to store information from processes obtained
 ! from a remote node. Process information is packed into the buffer
 ! on the remote node and transmitted back in a single message to the
 ! calling node. For this example, information from approximately ten
 ! processes can be transmitted at once. Use of buffering can
 ! significantly improve the performance of wildcard scans.

build_item_lists:

 getjpi_item_list::item(0)::buffer_length   = getjpi_username_buff
 getjpi_item_list::item(0)::item_code       = jpi$_username
 getjpi_item_list::item(0)::buffer_address  = loc(getjpi_username)
 getjpi_item_list::item(0)::length_address  =    &
                                  loc(getjpi_username_length)
 getjpi_item_list::item(1)::buffer_length   = getjpi_prcname_buff
 getjpi_item_list::item(1)::item_code       = jpi$_prcnam
 getjpi_item_list::item(1)::buffer_address  = loc(getjpi_prcname)
 getjpi_item_list::item(1)::length_address  =    &
                                  loc(getjpi_prcname_length)
 getjpi_item_list::item(2)::buffer_length   = getjpi_pid_buff
 getjpi_item_list::item(2)::item_code       = jpi$_pid
 getjpi_item_list::item(2)::buffer_address  = loc(getjpi_pid)
 getjpi_item_list::item(2)::length_address  = loc(getjpi_pid_length)
 getjpi_item_list::item(3)::buffer_length   = getjpi_imagename_buff
 getjpi_item_list::item(3)::item_code       = jpi$_imagname
 getjpi_item_list::item(3)::buffer_address  = loc(getjpi_imagename)
 getjpi_item_list::item(3)::length_address  =    &
                                  loc(getjpi_imagename_length)
 getjpi_item_list::item(4)::buffer_length   = getjpi_nodename_buff
 getjpi_item_list::item(4)::item_code       = jpi$_nodename
 getjpi_item_list::item(4)::buffer_address  = loc(getjpi_nodename)
 getjpi_item_list::item(4)::length_address  =    &
                                  loc(getjpi_nodename_length)
 getjpi_item_list::item(5)::list_terminator = 0%
 !
```

```
!      assign pscan items
!
   node_wildcard = "*"
 pscan_descriptor_list::pscan_item(0)::buffer_length  =   &
                                  len(node_wildcard)
 pscan_descriptor_list::pscan_item(0)::item_code      =   &
                                  pscan$_nodename
 pscan_descriptor_list::pscan_item(0)::buffaddr_itemval = &
                                  loc(node_wildcard)
 pscan_descriptor_list::pscan_item(0)::item_spec_flags = &
                                  pscan$m_wildcard
 pscan_descriptor_list::pscan_item(1)::buffer_length   = 0%
 pscan_descriptor_list::pscan_item(1)::item_code       =   &
                                     pscan$_getjpi_buffer_size
 pscan_descriptor_list::pscan_item(1)::buffaddr_itemval =   &
                                     pscan_getjpi_buffer_size
 pscan_descriptor_list::pscan_item(1)::item_spec_flags  = 0%
 pscan_descriptor_list::pscan_item(2)::list_terminator  = 0%

return ! end build_item_lists

 ! Call the $process_scan function to create a wildcard context for
 ! use by $getjpi.

call_process_scan:

    pscan_status = sys$process_scan(pid_context by ref,          &
                                    pscan_descriptor_list by ref)

    if (pscan_status and 1%) = 0%
       then
         call lib$stop(pscan_status by value,,,,,)
    end if

return          ! exit call_process_scan

 !
 ! The $getjpi call utilizes the wildcard context established by
 ! $process_scan to obtain information about processes across a
 ! VAXcluster system. The context describes the selective search for
 ! $getjpi. Once process information is obtained, it is stored in the
 ! print_line_array for further processing.
 !
 !                     *****  Important  *****
 !
 ! Since all remote $getjpi calls are asynchronous, a call to $synch
 ! is used to wait for the completion of $getjpi. The status of the
 ! call is returned in getjpi_status. The status of the remote
 ! operation is returned in the I/O status block (iosb). Alternatively,
 ! an ast could have been used to notify this program that $getjpi has
 ! completed. Regardless of the method, all $getjpi calls MUST be
 ! synchronized.
 !

call_getjpi:

    print_array_subscript = 50%
    dim string print_line_array(print_array_subscript)
    getjpi_status = sys$getjpiw(event_flag  BY VALUE,          &
                                pid_context BY REF,,           &
                                getjpi_item_list BY REF,       &
                                iosb BY REF,,)
```

```
    synch_status = sys$synch(event_flag BY VALUE, iosb BY REF)
    if (synch_status and 1%) = 0%
       then
         call lib$stop(synch_status by value,,,,,)
       else
          if (iosb::iosb_field(1) <> ss$_nomoreproc) and         &
             (iosb::iosb_field(1) <> ss$_normal)     and         &
             (iosb::iosb_field(1) <> ss$_nopriv)
             then
               call lib$stop(synch_status by value,,,,,)
          end if
    end if

    while iosb::iosb_field(1) <> ss$_nomoreproc

       if iosb::iosb_field(1) <> ss$_nopriv
          then
               imagname_location = 1
               until imagname_location = 0
                   imagname_location  = pos(getjpi_imagename,"]",0%)
                   getjpi_imagename   = right$(getjpi_imagename,    &
                                              imagname_location + 1)
               next

               ots_status = ots$cvt_l_tz(getjpi_pid by ref,         &
                                         ots_character_string by desc,  &
                                         ots_number_of_digits by value,)

               change getjpi_nodename to nodename_ascii_array
               for array_subscript = 0 to 6
                       if nodename_ascii_array(array_subscript) = 0
                          then
                            nodename_ascii_array(array_subscript) = 32
                       end if
               next array_subscript
               change nodename_ascii_array to getjpi_nodename

               print_record::nodename  = getjpi_nodename
               print_record::pid       = ots_character_string
               print_record::username  = getjpi_username
               print_record::procname  = edit$(getjpi_prcname,4%)
               print_record::imagename = edit$(getjpi_imagename,4%)
               process_count = process_count + 1
              if process_count > print_array_subscript
                then
                    gosub redimension_print_line_array
               else
               print_line_array(process_count) = print_record::all_data
              end if
       end if
getjpi_status = sys$getjpiw( event_flag BY VALUE,pid_context BY REF,, &
                             getjpi_item_list BY REF,                 &
                              iosb by REF,,)

       synch_status = sys$synch(event_flag BY VALUE, iosb BY REF)
       if (synch_status and 1%) = 0%
          then
             call lib$stop(synch_status by value,,,,,)
          else
             if (iosb::iosb_field(1) <> ss$_nomoreproc) and &
                (iosb::iosb_field(1) <> ss$_normal)     and &
                (iosb::iosb_field(1) <> ss$_nopriv)
                then
                   call lib$stop(synch_status by value,,,,,)
                end if
    end if

    next

return ! end of call_getjpi

!  Display information collected during the process search on the
!  terminal.
```

```
display_process_info:

    print top_of_screen
    print using center_80, title_heading
    print ""
    print header_one
    print header_two
    print dashed_line_80
    print ""

    for process_to_print = 1 to process_count
       print_record::all_data = print_line_array(process_to_print)
       print using print_line, print_record::nodename,          &
                               print_record::pid,               &
                               print_record::username,          &
                               print_record::procname,          &
                               print_record::imagename

    next process_to_print

return ! end display_process_info

! This routine dynamically allocates array storage as required
! in intervals of 50 elements.

redimension_print_line_array:

       dim string save_print_line_array(print_array_subscript)
       for array_subscript = 1 to print_array_subscript
              save_print_line_array(array_subscript) =          &
              print_line_array(array_subscript)
       next array_subscript

       print_array_subscript = print_array_subscript + 1
       dim string print_line_array(print_array_subscript)
       for array_subscript = 1 to (process_count - 1%)
              print_line_array(array_subscript) =               &
              save_print_line_array(array_subscript)
       next array_subscript
       print_line_array(process_count) = print_record::all_data

return !

error_routine:

  resume exit_program

exit_program:

 end
```

- **PSCAN.BAS TERMINAL DISPLAY**

```
                        VAXcluster Process Scanner

 NODE
 NAME      PID        USERNAME      PROCESS NAME       IMAGE NAME
 ------------------------------------------------------------------------

 VAXA    20205C2F       SMITH         SMITH_1
 VAXB    27E00079       SMITH         SMITH_2          PSCAN_2.EXE;24
```

6.3.2 Using Lock Management System Services

Lock management system services offer several programming techniques. The flexibility of lock management system services is based upon your selection of the parameters for each instance of a lock management system service call. Lock management system services have the following parameters:

- Event flag — efn
- Lock mode — lkmode
- Lock status block — lksb
- Modify lock action — flags
- Resource name — resnam
- Parent lock ID — parid
- Address of AST routine — astadr
- Parameter passed to AST routine — astprm
- Address of blocking AST routine — blkast
- Access mode — acmode

Table 6–2 presents the required and optional [ ] parameters for each of the lock management system services.

Table 6–2 Parameters for Lock Management System Services

Function	System Service	Parameters
Queue a new lock or lock conversion on a resource	$ENQ[1]	[efn], lkmode, lksb, [flags], [resnam], [parid], [astadr], [astprm], [blkast], [acmode], nullarg
Queue a lock request and wait	$ENQW[1]	[efn], lkmode, lksb, [flags], [resnam], [parid], [astadr], [astprm], [blkast], [acmode], nullarg
Release locks and cancel lock requests	$DEQ[1]	[lkid], [valblk], [acmode], [flags]
Get information about the lock database	$GETLKI[1]	[efn], lkidadr, itmlst, [iosb], [astadr], [astprm], nullarg
Queue an information request and wait	$GETLKIW[1]	[efn], lkidadr, itmlst, [iosb], [astadr], [astprm], nullarg

[1]For more information on the arguments used with the lock management system services parameters, see the *VMS System Services Reference Manual.*

By selecting the appropriate parameter values for each lock management system service call, you can implement process synchronization techniques using the following fundamental capabilities of lock management system services:

- Simple lock
- AST and blocking AST
- Lock value block

6.3.2.1 Using Simple Lock for Exclusive Access to a Shared Resource

If Process_A executes a lock request and the lock mode (lkmode) parameter specifies an EX lock mode for a resource name, by convention, when the lock is granted, Process_A has exclusive access to the resource. You can use an exclusive lock to synchronize clusterwide access to a critical resource. For example, you can use an EX lock on a resource name, that represents a shared global section, to coordinate access to the global section so only one process at a time can modify the global section.

Using Simple Lock Conversion for Event Notification

Lock conversion is a variation of using an EX lock mode for exclusive access. By converting the lock on a resource name to another lock mode, you can establish a new lock condition that may, indirectly, act as a signal to other processes. For example, Process_A on Node A is holding an EX lock on a resource name for updating, and upon completion, Process_A converts the EX lock mode to a CR lock mode. Then, all other processes in the VAXcluster system with locking requests for that resource in the waiting queue are serviced by the distributed VMS lock manager on a first-in / first-out basis.

6.3.2.2 Using Completion ASTs and Blocking ASTs with Lock Management System Services to Synchronize Simultaneous Processes

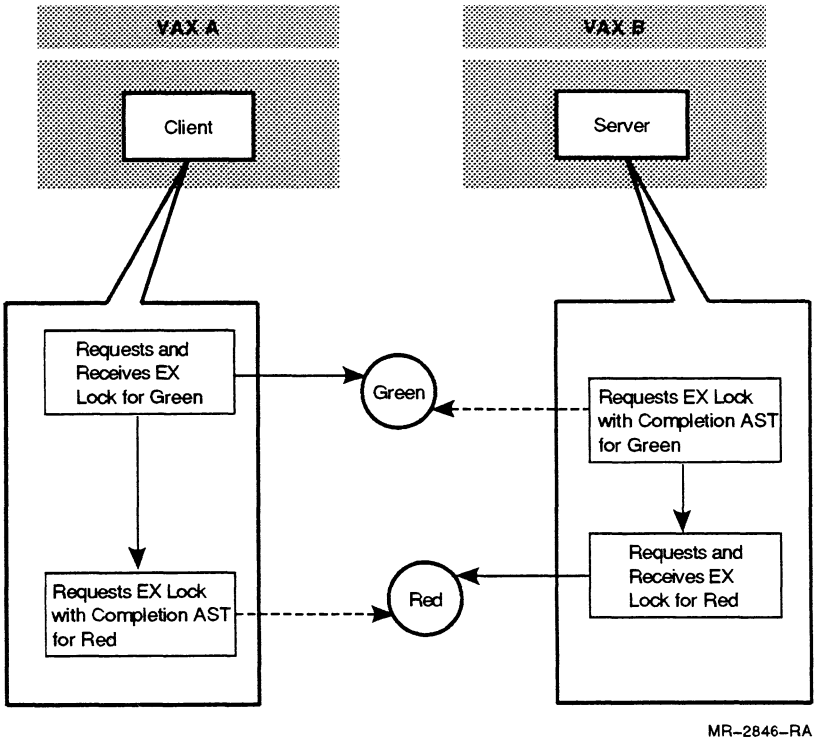
Using completion ASTs and blocking ASTs with lock management system services allows you to send direct signals for clusterwide, interprocess communication. By specifying the required parameters for an AST or blocking AST, you can implement the following types of interprocess communications:

- Deadman lock
- Doorbell lock

Deadman Lock

There are two cooperating processes (call them SERVER and CLIENT) that communicate with each other and are dependent upon each other; if one process fails, the other must also terminate. For example, if SERVER is the only process providing access to a database, and SERVER fails, then CLIENT needs an immediate notification. By implementing a lock strategy called a deadman lock, each process requests a lock with an AST on the lock held by the other process. When either process terminates, the lock requested by the other process is granted, triggering the AST, and the existing process executes an AST routine to notify the operator console and terminate the remaining process. Figure 6–3 illustrates a deadman lock scheme.

Figure 6–3 Deadman Lock Scheme



Doorbell Lock

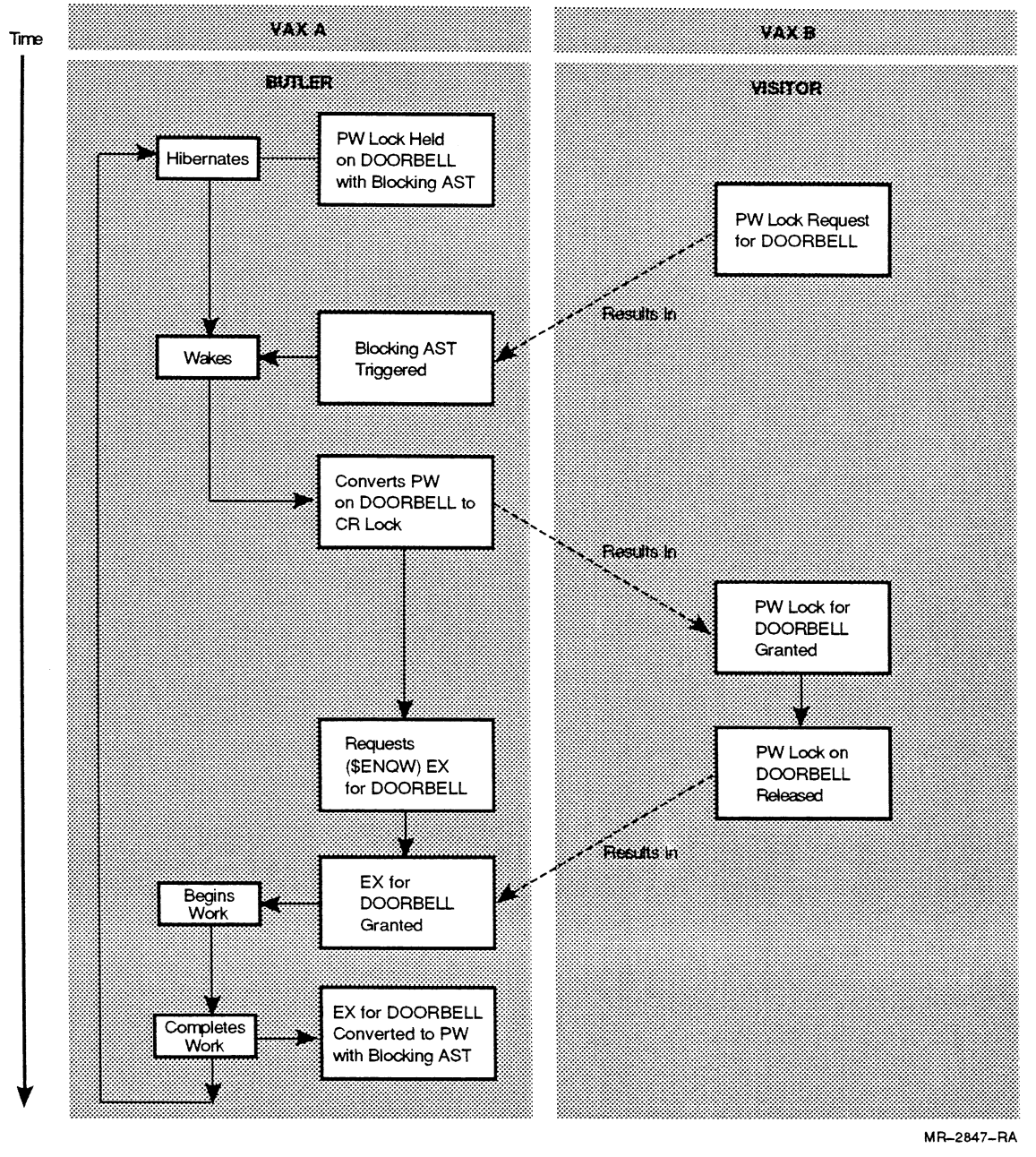
A doorbell lock strategy allows one process (VISITOR) to notify another process (BUTLER) that there is work to be done. In order to be notified, the BUTLER holds a lock on the resource name DOORBELL. To implement the doorbell scheme, the BUTLER process acquires a PW lock mode on DOORBELL and specifies a blocking AST. At this point, the BUTLER process can hibernate and wait for notification of work to be done.

When the VISITOR process has work for the BUTLER process, the VISITOR requests a PW lock mode on DOORBELL. This triggers the BUTLER's blocking AST. The BUTLER process then converts the PW mode lock on DOORBELL to CR mode. This allows the VISITOR's PW lock request to be granted. When the VISITOR acquires the PW lock, the VISITOR knows that the BUTLER "heard" the DOORBELL and can release the PW lock.

When the BUTLER process converts the PW lock to a CR lock for DOORBELL, the BUTLER process also requests (using an $ENQW) an EX lock for the resource DOORBELL. When the VISITOR process releases the PW lock, the BUTLER process acquires the EX lock and begins performing the appropriate "work" activities. When the BUTLER process completes the "work," the EX lock for DOORBELL is converted to a PW lock with a blocking AST, and the BUTLER process goes back into hibernation until the doorbell rings again. The doorbell lock scheme is represented in Figure 6–4.

Note: The doorbell lock scheme illustrated in Figure 6–4 is a simplified example. When implementing a doorbell lock scheme, you must be aware that timing windows may exist. In this example, the BUTLER process does not get AST notification of additional VISITOR requests until the BUTLER process has completed "work" from the first VISITOR request for work. In this doorbell scheme, the VISITOR process must operate on the assumption that a delay in acquisition of the resource DOORBELL means the BUTLER must be busy.

Figure 6–4 Doorbell Lock Scheme



Doorbell Scheme Using a Blocking AST — Programming Example

The following is an example FORTRAN application that implements the doorbell scheme described in Figure 6–4.

- **BUTLER.FOR**

```
C       This program uses the blocking AST feature of
C       the VMS lock manager to implement a doorbell lock.
C
        IMPLICIT        INTEGER*4   (A-Z)
        INCLUDE         '($LCKDEF)'
        CHARACTER*8     SHARE_ITEM      /'DOORBELL'/
        CHARACTER*16    VAL_BLOCK
        DIMENSION       STATBLK(2)
        COMMON /LOCK_STAT_BLK/ STATBLK,VAL_BLOCK
        COMMON          DONE,WAS_SLEEPING,ACTIVE_REQUESTS
        VOLATILE        DONE,WAS_SLEEPING,ACTIVE_REQUESTS
        EXTERNAL        BLOCKING
C
        WAS_SLEEPING = 0
        DONE = 0
C       Place a PW lock on DOORBELL
        TYPE *, 'Placing a PW lock on DOORBELL...'
        STAT= SYS$ENQW(%VAL(1), %VAL( LCK$K_PWMODE),
        1               STATBLK,,
        2               SHARE_ITEM,,,,
        3               BLOCKING,,)
        IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
        IF (.NOT. STATBLK(1)) CALL LIB$STOP(%VAL(STATBLK(1)))
C
C       This is the section where the work is done
C
        DO WHILE (.NOT. DONE)
C
           TYPE *, 'About to sleep waiting for work'
           STAT = SYS$HIBER()
           IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
           TYPE *, 'Returned from AST about to convert'
           FLAG= LCK$M_CONVERT
           STAT= SYS$ENQW(%VAL(1), %VAL( LCK$K_PWMODE),
        1               STATBLK,%VAL(FLAG),
        2               SHARE_ITEM,,,,
        3               BLOCKING,,)
           IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
           IF (.NOT. STATBLK(1)) CALL LIB$STOP(%VAL(STATBLK(1)))
           TYPE *, 'Lock converted from NL to PW mode'
C
C
        END DO
        TYPE *, 'Finished.'
        END
C
C
        SUBROUTINE BLOCKING
C
C       This is used as a blocking AST routine.  It
C       may be entered from hibernation or when the
C       the main converts to PW with blocking.
```

```
C
C
      IMPLICIT        INTEGER (A-Z)
      INCLUDE         '($LCKDEF)'
      CHARACTER*8     SHARE_ITEM      /'DOORBELL'/
      CHARACTER*16    VAL_BLOCK
      DIMENSION       LCK_STATBLK(2)
      COMMON /LOCK_STAT_BLK/ LCK_STATBLK,LCK_VAL_BLOCK
      VOLATILE        LCK_STATBLK,LCK_VAL_BLOCK
      COMMON          DONE,WAS_SLEEPING,ACTIVE_REQUESTS
      VOLATILE        DONE,WAS_SLEEPING,ACTIVE_REQUESTS
C
      TYPE *, 'MY lock is blocking another lock.'
      STAT=SYS$WAKE(,)
      IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
C
C     Convert PW lock to CR so VISITOR gets the lock.
C
      TYPE *, 'Converting to concurrent read'
      FLAG= LCK$M_CONVERT
      SEVERITY= SYS$ENQW(%VAL(3), %VAL( LCK$K_CRMODE),
     1                   LCK_STATBLK,
     2                   %VAL( FLAG),
     3                   SHARE_ITEM,,,,,)
      IF (.NOT. SEVERITY) CALL LIB$STOP( %VAL( SEVERITY))
      IF (.NOT. LCK_STATBLK(1)) CALL LIB$STOP(%VAL(LCK_STATBLK(1)))
C
C     Try to get the lock back. When I do the VISITOR has
C     written the work request.
C
      SEVERITY= SYS$ENQW(%VAL(2), %VAL( LCK$K_EXMODE),
     1                   LCK_STATBLK,
     2                   %VAL( FLAG),
     3                   SHARE_ITEM,,,,,)
      IF (.NOT. SEVERITY) CALL LIB$STOP( %VAL( SEVERITY))
      IF (.NOT. LCK_STATBLK(1)) CALL LIB$STOP(%VAL(LCK_STATBLK(1)))
C
C     Here the program would get the work request and then perform
C     the requested work.
C
       DO I=1,50
            TYPE *, 'Doing work'
       END DO
       TYPE *, 'Finished work'

      RETURN
      END
```

- **VISITOR.FOR**

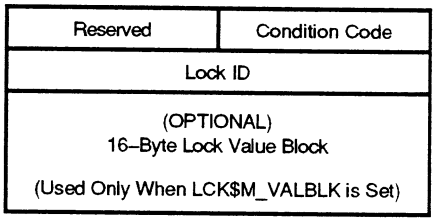
```
C       This program queues rings to the BUTLER's "doorbell,"
C       places a work request in a known location, and then
C       gives up the lock.
C
        IMPLICIT        INTEGER*4   (A-Z)
        INCLUDE         '($LCKDEF)'
        CHARACTER*8     SHARE_ITEM      /'DOORBELL'/
        CHARACTER*16    VAL_BLOCK
        DIMENSION       STATBLK(2)
        COMMON /LOCK_STAT_BLK/  STATBLK, VAL_BLOCK
C
C       Place a PW lock request on DOORBELL
        TYPE *, 'Placing a PW lock on DOORBELL...'
        STAT= SYS$ENQW(%VAL(1), %VAL( LCK$K_PWMODE),
        1               STATBLK,,
        2               SHARE_ITEM,,,,,,)
        IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
        IF (.NOT. STATBLK(1)) CALL LIB$STOP (%VAL(STATBLK(1)))
C
C       In an actual application this program would place a work
C       request in a file or possibly in the lock value block.
C
        DO I = 1,30
         TYPE *, 'Placing work request in known location'
        END DO
C
C       Give up the PW lock on DOORBELL so that the BUTLER knows
c       the work request has been written.
C
        STAT= SYS$DEQ(%VAL(STATBLK(2)),,,)
        IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
        TYPE *, 'Finished.'
        END
```

6.3.2.3 Using the Lock Value Block to Pass Information

The lock value block illustrated in Figure 6–5 is a 16-byte area in the lock status block for data storage. The use of the lock value block is optional. By specifying an appropriate value for the address of the lock status block (lksb), allocating memory for the lock value block, and setting the LCK$M_VALBLK flag for the lksb, you can read and possibly modify the contents of the lock value block.

- You can read the lock value block when:
  - — A new lock is granted.
  - — The existing lock is converted to a higher lock mode.
- You can modify and store the lock value block when:
  - — The lock is dequeued from an EX or PW lock mode.
  - — The lock is converted from EX or PW lock mode to the same or lower mode.

Figure 6–5 Format of Lock Status Block



MR–2848–RA

Passing Information with the Lock Value Block — Programming Example

In this programming example, first WRITELOCK.EXE requests a PW lock with a blocking AST for the resource name DATABASE. Then, READLOCK.EXE executes and requests a PR lock mode for the resource DATABASE. The blocking AST routine of WRITELOCK.EXE executes, and WRITELOCK.EXE converts the PW lock mode to a NL lock mode and writes 16 bytes of information to the lock value block for the resource name DATABASE. After READLOCK.EXE acquires the PR lock mode for DATABASE, READLOCK.EXE reads the lock value block written by WRITELOCK.EXE.

- **WRITELOCK.FOR**

```
c  WRITELOCK.FOR is executed first.  This program
c  uses the blocking AST feature of the VMS lock manager
c  for notification that another process (READLOCK.FOR)
c  has placed a locking request for the resource DATABASE.
c  WRITELOCK.FOR writes 16 bytes of information to the lock value
c  block of DATABASE and converts the PW lock to a NL lock.
c
        IMPLICIT        INTEGER*4   (A-Z)
        INCLUDE         '($LCKDEF)'
        CHARACTER*8     SHARE_ITEM      /'DATABASE'/
        CHARACTER*16    VAL_BLOCK
        DIMENSION       STATBLK(2)
        COMMON /LOCK_STAT_BLK/  STATBLK, VAL_BLOCK
        COMMON          HAVELOCK
        VOLATILE        HAVELOCK
        EXTERNAL        BLOCKING
c
        DONE = 0
c  Place a PW lock on DATABASE.
        TYPE *, 'Placing a PW lock on DATABASE...'
 30     STAT= SYS$ENQW(%VAL(1), %VAL( LCK$K_PWMODE),
        1               STATBLK,,
        2               SHARE_ITEM,,,,
        3               BLOCKING,,)
        IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
        IF (.NOT. STATBLK(1)) CALL LIB$STOP(%VAL(STATBLK(1)))
c
 40     HAVELOCK = 1
        DO 70 WHILE ((HAVELOCK .EQ. 1)
        1           .AND. (DONE .EQ. 0))
```

```
c  Operations which require write access
c  to the resource could be performed here.
c  Set DONE to 1 when finished.
 70     END DO
c
c  Convert PW lock to NL and write to the lock value block.
       VAL_BLOCK='1234567812345678'
       FLAG= LCK$M_CONVERT .OR. LCK$M_VALBLK
       SEVERITY= SYS$ENQW(%VAL(2), %VAL( LCK$K_NLMODE),
       1                  STATBLK,
       2                  %VAL( FLAG),
       3                  SHARE_ITEM,,,,,,)
       IF (.NOT. SEVERITY) CALL LIB$STOP( %VAL( SEVERITY))
       IF (.NOT. STATBLK(1)) CALL LIB$STOP(%VAL(STATBLK(1)))
       TYPE *, 'Lock converted from PW to NL mode'
c
c  Here we could test if done = 1.  If not,
c  we should call $ENQW to convert lock mode to PW
c  and go back to statement 40.
c
       TYPE *, 'Finished.'
       END
c
c
       SUBROUTINE BLOCKING
c
c  This is used as a blocking AST routine.  It
c  tells the main program (by means of a flag) to
c  release its lock, which is blocking another lock.
c
       IMPLICIT        INTEGER (A-Z)
       COMMON          HAVELOCK
       VOLATILE        HAVELOCK
c
       HAVELOCK = 0
       TYPE *, 'Our lock is blocking another lock.'
       RETURN
       END
```

- **READLOCK.FOR**

```
c  This program is run after WRITELOCK.FOR executes.
c  READLOCK.FOR enqueues a null lock on the
c  resource DATABASE, which is later converted
c  to a protected read (PR) lock which triggers the blocking
c  AST held by WRITELOCK.FOR on the resource DATABASE.
c  When READLOCK acquires the PR lock, the lock value block
c  of DATABASE is read.
c
       IMPLICIT        INTEGER*4   (A-Z)
       INCLUDE         '($LCKDEF)'
       INCLUDE         '($SSDEF)'
       CHARACTER*8     SHARE_ITEM      /'DATABASE'/
       CHARACTER*16    VAL_BLOCK
       DIMENSION       STATBLK(2)
       COMMON /LOCK_STAT_BLK/  STATBLK, VAL_BLOCK
c
c Place a null lock on DATABASE.
c
       TYPE *, 'Placing a NL lock on the resource...'
       STAT= SYS$ENQW(%VAL(1), %VAL( LCK$K_NLMODE),
       1               STATBLK,,
       2               SHARE_ITEM,,,,,,)
       IF (.NOT. STAT) CALL LIB$STOP( %VAL( STAT))
       IF (.NOT. STATBLK(1)) CALL LIB$STOP (%VAL(STATBLK(1)))
c
c  Operations not requiring access to the
c  resource could be performed here.
c
c      Convert null lock to PR.
```

```
c
        FLAG = LCK$M_CONVERT .OR. LCK$M_VALBLK
        TYPE *, 'Converting lock from NL to PR mode...'
        SEVERITY= SYS$ENQW(%VAL(2), %VAL( LCK$K_PRMODE),
        1                  STATBLK,
        2                  %VAL( FLAG),
        3                  SHARE_ITEM,,,,,,)
        IF (.NOT. SEVERITY) CALL LIB$STOP( %VAL( SEVERITY))
c
        IF (STATBLK(1) .EQ. SS$_VALNOTVALID) THEN
c
c             Perform appropriate actions based on knowledge that a
c             code with the capability to write to the value block
c             has terminated unexpectedly.
c
              TYPE *, 'Value block is invalid'
        ELSE IF (.NOT. STATBLK(1)) THEN
              CALL LIB$STOP (%VAL(STATBLK(1)))
        ELSE
c
c             Operations requiring access to the resource
c             could be performed here.
c
              TYPE *, 'Value block contains ',VAL_BLOCK
        END IF
        TYPE *, 'Finished.'
        END
```

For more information on using the LCK$M_VALBLK flag to allocate memory for the lock value block, see the *Introduction to VMS System Services* and the *VMS System Services Reference Manual.*

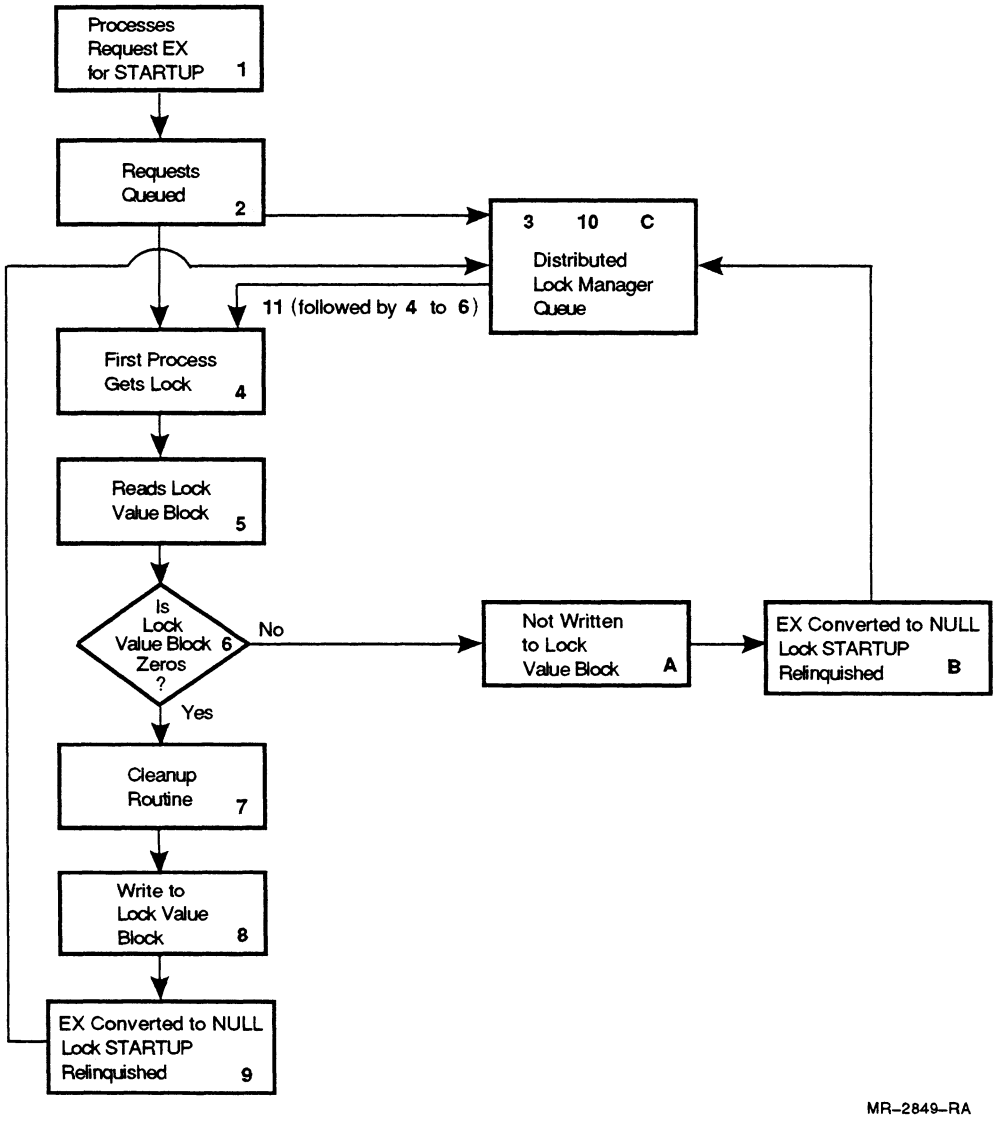
Using the Lock Value Block to Determine Who is First

When multiple processes of an application are attempting to access a common resource, it can be important for any process accessing the shared resource to determine if it is the first to access the resource. For example, if multiple VAXcluster nodes are executing a SERVER process accessing a shared database and the VAXcluster system enters a state transition, it may be important for a SERVER process re-accessing the database to perform cleanup activities. Once the VAXcluster system is reestablished, the first SERVER process started in the VAXcluster system must delete invalid data left in the database from before the system failure. Only the first SERVER process must perform a cleanup routine to identify any invalid data in the database; other SERVERs must not because they might delete valid data from SERVERs that have already started.

By using the lock value block (see Figure 6–6), any SERVER process accessing the database can identify if it is first. This is accomplished by having each SERVER request an EX mode lock for resource name STARTUP. Only the first SERVER process to request the EX lock on STARTUP is granted the lock; all other SERVER processes are placed on the waiting queue. The first SERVER to be granted the EX lock reads the lock value block for the resource name STARTUP. If the lock value block is zeros, then the SERVER process is the first process to access the database. The **first** SERVER process continues to hold the EX lock on STARTUP and performs a cleanup routine. (If the lock value block is non-zero, the process converts the EX lock to a NL and performs regular work.)

After completing the cleanup routine, the first SERVER process writes "DON'T DELETE" into the lock value block. Then, the first SERVER process converts the EX lock on STARTUP to a NL lock. Consequently, after the first SERVER process converts the EX lock to a NL lock, the SERVER process at the top of the waiting queue acquires the EX lock for resource name STARTUP. This SERVER process then reads the lock value block and determines if the contents are zeros. When the SERVER process evaluates the lock value block as non-zero, the SERVER process converts the EX lock to a NL and performs regular work.

Figure 6–6 Determining Who is First with a Lock Value Block Scheme



Notes on Figure 6–6

In this example, it is not necessary for the SERVER to check for an invalid lock value block when reading the lock value block. The first SERVER process acquires access to the resource STARTUP and begins clean-up work. If the first SERVER should encounter a CPU failure, a new message will not be written to the lock value block. Thus, the next SERVER process to access STARTUP will read the lock value block as zeros and begin the clean-up work that was never completed by the first SERVER.

6.3.3 Coordinating Processes Using DECnet-VAX Communications

DECnet-VAX communications can be used to coordinate processes synchronously or asynchronously. In transparent DECnet-VAX communications, you can use the $QIO system service or high-level language READs and WRITEs to coordinate processes synchronously. That is, first one process writes and the second reads, then the second writes and the first reads. However, when you use nontransparent DECnet-VAX communications which supports asynchronous communications, you have greater flexibility for implementing complex interprocess communications.

Coordinating Processes Using Nontransparent DECnet–VAX Communications — Programming Example

The following nontransparent DECnet-VAX example demonstrates a local server process (LOCAL_3.EXE) responding asynchronously to requests from multiple remote processes (REMOTE_3.EXE). LOCAL_3.EXE assigns segments of work, as chunks of an array, and each REMOTE_3.EXE performs matrix multiplications on the assigned input. When each REMOTE_3.EXE has completed the assigned work, they wait, asynchronously, for the LOCAL_3.EXE to assign a new chunk of the array as input.

- **PART3.INC**

```
PARAMETER ICHAN_MAX = 10
PARAMETER IMAX_BUFFER = 450
PARAMETER ISIZE = 100

INTEGER*2 ICHANNEL(ICHAN_MAX)
BYTE ICOL_STAT(ISIZE)
INTEGER IN_ARRAY1(ISIZE,ISIZE), IN_ARRAY2(ISIZE,ISIZE),
1       RESULTS(ISIZE,ISIZE), IREC_BUF(ISIZE,IMAX_BUFFER),
1       ICOL_ASSIGNED(ICHAN_MAX), READ_FUNCTION, WRITE_FUNCTION
REAL*8 IOSB(ICHAN_MAX)

INTEGER SYS$QIO, SYS$QIOW, SYS$ASSIGN, SYS$DASSGN

EXTERNAL IO$_READVBLK, IO$_WRITEVBLK, IO$M_NOW, IO$M_NORSWAIT

COMMON /DIST_DATA/ READ_FUNCTION, WRITE_FUNCTION, ICHANNEL,
1       IOSB, IO_RECEIVED_ADDR, IN_ARRAY1, IN_ARRAY2, RESULTS,
1       IANSWER, ICOL_ASSIGNED
```

- **LOCAL_3.FOR**

```
      PROGRAM MATMUL_TEST
c
c  In this version of the sample program, the single client - single
c  server model is replaced with a single master - multiple slaves
c  model.  In this example, two remote nodes, both executing the
c  REMOTE_3.FOR code, cooperate on performing this matrix multiply.
c
c  Get the array definitions for this example.
c
      INCLUDE 'PART3.INC'

      EXTERNAL IO_RECEIVED

      READ_FUNCTION = %LOC(IO$_READVBLK)
      WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)
      IO_RECEIVED_ADDR = %LOC(IO_RECEIVED)

c
c  Set up the initial parameters for the matrix multiply.
c  The matrix is divided among the multiple remote processes by
c  each slave sending a message to this program asking for the next
c  piece of work that is available.  The work is assigned in full
c  columns. For example, the first remote process that requests work
c  is directed to perform the calculations for the first column of the
c  output array.  This does add some additional work as compared to
c  just splitting the work "50/50" between the two nodes at the start,
c  but it properly handles the case where the remote systems are not
c  identical -- either in hardware or in system loading.  Using this
c  technique, whichever system can do more work will do more work
c  instead of having the fastest node waiting for the slowest to
c  complete its "half".
c
c  To keep track of the next piece of work, the "INEXT" variable
c  defines the next column to be allocated.  "ITOTAL_COLS_DONE"
c  monitors how many have been completed (that is, the calculations
c  done and the data returned) and "IOTHER_NODES" denotes how many
c  remote nodes are going to be used.
c
      INEXT = 1
      ITOTAL_COLS_DONE = 0
      IOTHER_NODES = 2

c
c  First step, open the remote channels.
c
      ISTAT = LIB$ASN_WTH_MBX (
     1                         'NODE1::"0=REMOTE_3"',
     1                         450,
     1                         450,
     1                         ICHANNEL(1),
     1                         IGNORE)

      ISTAT = LIB$ASN_WTH_MBX (
     1                         'NODE2::"0=REMOTE_3"',
     1                         450,
     1                         450,
     1                         ICHANNEL(2),
     1                         IGNORE)
```

```
c
c  Start by sending all the data to the remote nodes.
c
      DO I = 1,IOTHER_NODES
         DO J = 1,ISIZE
            ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),%VAL(WRITE_FUNCTION)
     1        ,IOSB(ICHAN),,,IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
            ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),%VAL(WRITE_FUNCTION)
     1        ,IOSB(ICHAN),,,IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
         ENDDO
c
c  Since we are now dealing with multiple remote processes,
c  asynchronous QIOs must be set up to handle an I/O being sent
c  from any remote slave. When an I/O arrives, the IO_RECEIVED
c  routine will be called.
c
          ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),
     1    %VAL(READ_FUNCTION),IOSB(I),%VAL(IO_RECEIVED_ADDR),%VAL(I),
     1    IREC_BUF(1,I),%VAL(ISIZE*4),,,,)

      ENDDO

c
c  This call to SYS$HIBER suspends this main routine until all
c  remote calculations are completed.  Once all data has been
c  received, a SYS$WAKE is executed; this reawakens this process.
c
      CALL SYS$HIBER

      CALL EXIT
      END

c-------------------------------------------------------------------------

      SUBROUTINE IO_RECEIVED (ICHAN_BY_VALUE)
c
c  Execute this routine when an incoming packet is received.  If the
c  length of this packet = 4, the packet is a request for the next
c  piece of work to be done. If the length is greater, the remote
c  process is returning the column that it has calculated.
c
      INCLUDE 'PART3.INC'

c
c  First, get a local copy of the parameter coming into this routine
c  "by value" instead of the usual "by reference."
c
      ICHAN = %LOC(ICHAN_BY_VALUE)

c
c  Determine the length of the incoming packet from the I/O Status
c  Block.
c
      CALL CNVT_IOSB_STATUS (IOSB(ICHAN), ISTATUS, IBUF_LEN)
```

```
c
c  If the length was four, this is a request for more work.
c
      IF (IBUF_LEN .EQ. 4) THEN
c
c  Yes, it was a request.  Get the next column number that is
c  available.
c
         IF (INEXT .LE. ISIZE) THEN
            ICOL_ASSIGNED(ICHAN) = INEXT
            INEXT = INEXT + 1
         ELSE
c
c  No more columns left, return -1 to tell the remote process to exit.
c
            ICOL_ASSIGNED(ICHAN) = -1
         ENDIF

c
c  Send the column number back to the remote process.
c
         ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),%VAL(WRITE_FUNCTION)
     1       ,IOSB(ICHAN),,,ICOL_ASSIGNED(ICHAN),%VAL(4),,,,)
      ELSE
c
c  This is not a request for work but is a column of data being
c  returned.
c
         ITOTAL_COLS_DONE = ITOTAL_COLS_DONE + 1
c
c  Move the column from the receive buffer into the real place for it.
C
         DO I = 1,ISIZE
            RESULTS(I,ICOL_ASSIGNED(ICHAN)) = IREC_BUF(I,ICHAN)
         ENDDO

      ENDIF
c
c  Decide if everything is done and a SYS$WAKE should be executed
c  or not.
c
      IF (ITOTAL_COLS_DONE .GE. ISIZE) THEN
c
c  Yes, everything is done.  Signal the remote process to exit and do
c  a SYS$WAKE on our own process.
c
         ICOL_ASSIGNED(ICHAN) = -1
         ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),%VAL(WRITE_FUNCTION)
     1       ,IOSB(ICHAN),,,ICOL_ASSIGNED(ICHAN),%VAL(4),,,,)
         CALL SYS$WAKE (,)
      ELSE

c
c  Nope, not done yet.  Reissue the QIO for the next incoming packet
c  from this remote system.
c
         ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),
     1      %VAL(READ_FUNCTION),IOSB(ICHAN),%VAL(IO_RECEIVED_ADDR),
     1      %VAL(ICHAN),IREC_BUF(1,ICHAN),%VAL(ISIZE*4),,,,)
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE CNVT_IOSB_STATUS (IOSB_QUAD, ISTATUS, IBUF_LEN)
c
c  Routine to pull the iosb status field out of the quadword (defined
c  as double precision for array convenience) and put it into the
c  integer field.
c
      INTEGER*2 IOSB_QUAD(4)
      INTEGER ISTATUS, IBUF_LEN

      ISTATUS = IOSB_QUAD(1)
      IBUF_LEN = IOSB_QUAD(2)
      RETURN
      END
```

- **REMOTE_3.FOR**

```
      PROGRAM MATMUL_REMOTE_TEST
c
c  This is the remote portion of this example program.  Most of
c  this program is identical to the previous, but there is the
c  addition of the arbitration to determine which of these remote
c  systems will execute which column.
c
c  Note that since each remote process does not calculate the entire
c  output array and calculates only one column at a time, the
c  answers will be put into a ISEND_BUF array that is large enough
c  for just a single column of data.
c
      INCLUDE 'PART3.INC'
      INCLUDE '($SSDEF)'

      INTEGER ISEND_BUF(ISIZE), MY_COL

c
c  Open the channel back to the local program.
c
      ISTAT = LIB$ASN_WTH_MBX ('sys$net',
     1                         IMAX_BUFFER,
     1                         IMAX_BUFFER,
     1                         ICHANNEL(1),
     1                         IGNORE )

      READ_FUNCTION = %LOC(IO$_READVBLK)
      WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)


c
c  Start by getting all the data.
c
      write (6,123)
123   format (' reading the data')
      DO J = 1,ISIZE
         ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),
     1       %VAL(READ_FUNCTION),IOSB(1),,,
     1       IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
         IF (ISTAT .NE. 1) CALL EXIT (ISTAT)

         ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),
     1       %VAL(READ_FUNCTION),IOSB(1),,,
     1       IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
         IF (ISTAT .NE. 1) CALL EXIT (ISTAT)
      ENDDO
```

```
c
c  OK, all the data is here.  Ask the master process what column
c  is available for processing.  To do this, send a packet of length
c  of four to it.  (Note that the signal is the packet length, not
c  the contents of the variable being sent.)
c
100     write (6,151)
151     format (' requesting a column')
        ISTAT = SYS$QIO (,%VAL(ICHANNEL(1)),%VAL(WRITE_FUNCTION)
        1       ,IOSB(1),,,MY_COL,%VAL(4),,,,)

        write (6,162)
162     format (' reading the column')
        ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),%VAL(READ_FUNCTION)
        1       ,IOSB(1),,,MY_COL,%VAL(4),,,,)

        IF (ISTAT .NE. 1) CALL EXIT (ISTAT)

c
c  No more rows?  Exit.
c
        IF (MY_COL .EQ. -1) CALL EXIT

c
c  Got the column number.  Do this calculation and then return the
c  data.
c
    DO I = 1,ISIZE
      ISEND_BUF(I) = 0
      DO K = 1,ISIZE
        ISEND_BUF(I) = ISEND_BUF(I) + (IN_ARRAY1(I,K) * IN_ARRAY2(K,I))
      ENDDO
    ENDDO
c
c  Send this column back to the local process.
c
        write (6,11)
11      format (' sending the data back')
        ISTAT = SYS$QIO (,%VAL(ICHANNEL(1)),%VAL(WRITE_FUNCTION)
        1       ,IOSB(1),,,ISEND_BUF,%VAL(ISIZE*4),,,,)
        IF (ISTAT .NE. 1) CALL EXIT (ISTAT)

c
c  Go back for more.
c
        GO TO 100
        END
```

6.4 Exception Conditions

Exception conditions must be explicitly defined when considering an application designed for distribution by decomposition. In some designs for decomposed distributed applications, there may be more single points of failure than in non-distributed applications because a distributed application uses multiple VAXcluster CPUs. In addition to normal exception conditions, when designing a distributed application, you must plan for recovery from the following exception conditions:

- Failure of an interprocess communication
- Failure of a VAXcluster CPU

Programming techniques for recovering from these exception conditions are described in the following sections of this manual.

6.4.1 Recovery from Interprocess Communication Failures

Basically, interprocess communication failures can occur for three reasons:

- The process you are communicating with aborts.
- There is a failure in the DECnet-VAX transmission link.
- The VAXcluster CPU on which the process you are communicating with has a hardware failure.

In the first and second cases, it may be possible to recover by retransmitting. However, recovery in the third case requires implementing a failover capability as part of your application design.

Using Nontransparent DECnet–VAX Communications to Recover from Interprocess Communication Failures — Programming Example

In this nontransparent DECnet-VAX programming example, the master process (LOCAL_4.EXE) is designed to transmit all of the elements of an input array to multiple remote slave processes. Each remote slave process (REMOTE_4.EXE) performs a computation for a part of the array, returns the result, and requests another part of the array for computation from the master. To recover from a remote slave process failure, the master process tracks what parts of the array have been allocated to a remote slave process for computation and what parts of the array have been computed. If there is a communications failure of any kind, the part of the array allocated to the failed remote process is declared "unallocated," and is re-allocated to another remote slave process.

- **PART4.INC**

```
PARAMETER ICHAN_MAX = 10
PARAMETER IMAX_BUFFER = 450
PARAMETER ISIZE = 100

INTEGER*2 ICHANNEL(ICHAN_MAX)
BYTE ICOL_STAT(ISIZE)
INTEGER IN_ARRAY1(ISIZE,ISIZE), IN_ARRAY2(ISIZE,ISIZE),
1       RESULTS(ISIZE,ISIZE), IREC_BUF(ISIZE,IMAX_BUFFER),
1       ICOL_ASSIGNED(ICHAN_MAX), READ_FUNCTION, WRITE_FUNCTION
REAL*8 IOSB(ICHAN_MAX)

INTEGER SYS$QIO, SYS$QIOW, SYS$ASSIGN, SYS$DASSGN

EXTERNAL IO$_READVBLK, IO$_WRITEVBLK, IO$M_NOW, IO$M_NORSWAIT

COMMON /DIST_DATA/ READ_FUNCTION, WRITE_FUNCTION, ICHANNEL,
1        IOSB, IO_RECEIVED_ADDR, IN_ARRAY1, IN_ARRAY2, RESULTS,
1        IANSWER, ICOL_STAT, ICOL_ASSIGNED
```

- **LOCAL_4.FOR**

```
      PROGRAM MATMUL_TEST
c
c  This example shows one type of error handling that can be done in a
c  single client as master - multiple remote processes as slaves model.
c  Rather than having a single variable containing the "next column" to
c  be allocated, a "column status" array is used to keep track of what
c  columns have been allocated and which are completed.  Each element
c  in the column status array (ICOL_STAT) will have one of the
c  following values:
c
c       0 = column not yet assigned to any process
c      -1 = column assigned, but answers not yet back
c       1 = column assigned, answers back  (column totally done)
c
c  The way the columns are allocated is that a search is made
c  of the column status array to find the first unallocated column and
c  then that column number is returned.  (For clarity of code, the
c  search always starts at the base of the array.  For performance,
c  there should be a variable to control where the searching begins.)
c
c  If there is a communications error of any kind, the column
c  currently allocated to that remote process will be declared
c  "unallocated".
c
c  Get the array definitions for this example.
c
      INCLUDE 'PART4.INC'
      INCLUDE '($SSDEF)'

      EXTERNAL IO_RECEIVED

      IO_RECEIVED_ADDR = %LOC(IO_RECEIVED)
      WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)
      READ_FUNCTION = %LOC(IO$_READVBLK)

      ITOTAL_COLS_DONE = 0
      IOTHER_NODES = 2

c
c  Initialize the column status array and the array storing which
c  column is currently allocated to which process.
c

      DO I = 1,ISIZE
         ICOL_STAT(I) = 0
      ENDDO
      DO I = 1,IOTHER_NODES
         ICOL_ASSIGNED(I) = -1
      ENDDO


c
c  First step, open the remote channels.
c
      ISTAT = LIB$ASN_WTH_MBX (
      1                       'NODE1::"0=REMOTE_4"',
      1                       450,
      1                       450,
      1                       ICHANNEL(1),
      1                       IGNORE)

      ISTAT = LIB$ASN_WTH_MBX (
      1                       'NODE2::"0=REMOTE_4"',
      1                       450,
      1                       450,
      1                       ICHANNEL(2),
      1                       IGNORE)
```

```
c
c  Initialize the arrays to test values.
c
        DO I = 1,ISIZE
           DO J = 1,ISIZE
              IN_ARRAY1(I,J) = I - (ISIZE/2)
              IN_ARRAY2(I,J) = J - (ISIZE/2)
           ENDDO
        ENDDO

c
c  Start by sending all the data to the remote nodes.
c
        DO I = IOTHER_NODES,1,-1
           DO J = 1,ISIZE
              ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),%VAL(WRITE_FUNCTION)
     1          ,IOSB(I),,,IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
              ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),%VAL(WRITE_FUNCTION)
     1          ,IOSB(I),,,IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
           ENDDO
c
c  Since we are now dealing with multiple remote slaves, asynchronous
c  QIOs must be set up to handle an I/O being sent from any remote
c  slave to the master.  When an I/O arrives, the IO_RECEIVED routine
c  will be called.
c
          ISTAT = SYS$QIO (,%VAL(ICHANNEL(I)),
     1     %VAL(READ_FUNCTION),IOSB(I),%VAL(IO_RECEIVED_ADDR),%VAL(I),
     1     IREC_BUF(1,I),%VAL(ISIZE*4),,,,)

        ENDDO

c
c  This call to SYS$HIBER suspends this main routine until all
c  remote calculations are completed.  Once all the data has been
c  received, a SYS$WAKE is executed which reawakens this process.
c
        CALL SYS$HIBER

c
c  Sum up all of the locations in the answer to confirm that this test
c  worked properly.
c
        ITOTAL = 0
        DO I = 1,ISIZE
           DO J = 1,ISIZE
              ITOTAL = ITOTAL + RESULTS(I,J)
           ENDDO
        ENDDO

        IF (ITOTAL .EQ. 833500000) THEN
           WRITE (6,10)
10         FORMAT (' The results from this test are correct.')
        ELSE
           WRITE (6,20) ITOTAL
20         FORMAT (' The results are not correct from this test.',/,
     1          '   Anticipated answer: 835500000',/,
     1          '   Obtained answer: ',i)
        ENDIF

        CALL EXIT
        END

c-------------------------------------------------------------------------

        SUBROUTINE IO_RECEIVED (ICHAN_BY_VALUE)
c
c  Execute this routine when an incoming packet is received.  If the
c  length of this packet = 4, the packet is a request for the next
c  piece of work to be done. If the length is greater, the remote
c  process is returning the column that it has calculated.
c
```

```
      INCLUDE 'PART4.INC'
      INCLUDE '($SSDEF)'

c
c  First, get a local copy of the channel parameter that was passed
c  to this routine by value instead of by reference.
c
      ICHAN = %LOC(ICHAN_BY_VALUE)

c
c  Determine the length of the incoming packet from the I/O Status
c  Block.
c
      CALL CNVT_IOSB_STATUS (IOSB(ICHAN), ISTATUS, IBUF_LEN)

c
c  Check the status of the I/O that just completed.
c
      IF (ISTATUS .NE. SS$_NORMAL) THEN
c
c  Something went wrong.  Since there can be two kinds of packets
c  coming back (data or a request for the next column), check to see
c  which packet type it is by determining if there is a column
c  currently assigned to the process.  If there is, deassign it.
c  In either case, deassign the I/O channel (which stops the remote
c  process if it has not already stopped for some reason) and return
c  without setting up another QIO.
c
         IF (ICOL_ASSIGNED(ICHAN) .GT. 0) THEN
            ICOL_STAT(ICOL_ASSIGNED(ICHAN)) = 0
         ENDIF

         CALL SYS$DASSGN (%VAL(ICHAN))
         RETURN

      ENDIF

c
c  If the length was four, this is a request for more work.
c
      IF (IBUF_LEN .EQ. 4) THEN
c
c  Yes, it was a request.  Get the next column number that is
c  available. Search through the column status array for the
c  first entry containing a zero.
c

c
c  Note, the value of icol_assigned(ichan) equals -1 coming into this
c  section, so if no free columns are found, an "exit out" message is
c  sent back to the remote process.
c
c  As an aside, if this loop determines that there are no free columns
c  available, we could do a performance optimization based on the fact
c  that the calculations of each column are not destructive, that is:
c  multiple remote processes could all be calculating row 5 at once
c  without interfering with each other.  If we had 5 remote processes
c  and 4 of them are executing columns and 1 just realized that there
c  were no more columns to execute, we could "doubly assign" one of
c  the columns to the "idle" remote process and take the data of
c  whichever one completes first.  This will be left as an "exercise
c  for the reader" to implement.
c  (Hint: the remote process has to use asynchronous QIOs to permit
c  an I/O message from the local program saying, "forget what you are
c  doing since someone else already completed it.")  One caution, this
c  technique can get better performance as measured in wall-clock time,
c  but adds to the overhead as measured in terms of raw CPU cycles used
c  since multiple nodes are calculating the same answers.  Use this
c  technique sparingly, preferably on standalone systems.
```

```
c
     DO I = 1,ISIZE
        IF (ICOL_STAT(I) .EQ. 0 .AND. ICOL_ASSIGNED(ICHAN) .LT. 0) THEN
              ICOL_STAT(I) = -1
              ICOL_ASSIGNED(ICHAN) = I
        ENDIF
     ENDDO

c
c  And send the column number back to the remote process.
c
            ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),%VAL(WRITE_FUNCTION)
     1         ,IOSB(ICHAN),,,ICOL_ASSIGNED(ICHAN),%VAL(4),,,,)
           ELSE
c
c  This is not a request for work but rather is a column of data being
c  returned.
c
            ITOTAL_COLS_DONE = ITOTAL_COLS_DONE + 1
c
c  Move the column from the receive buffer into the real place for it.
            DO I = 1,ISIZE
               RESULTS(I,ICOL_ASSIGNED(ICHAN)) = IREC_BUF(I,ICHAN)
            ENDDO

c
c  And now adjust the column status array to reflect that it is done.
c  In addition, change the status of the icol_assigned array to reflect
c  that this remote process now has no columns assigned to it.
c
            ICOL_STAT(ICOL_ASSIGNED(ICHAN)) = 1
            ICOL_ASSIGNED(ICHAN) = -1

           ENDIF
c
c  Decide if everything is done and a SYS$WAKE should be executed
c  or not.
c
           IF (ITOTAL_COLS_DONE .GE. ISIZE) THEN
c
c  Yes, everything is done.  Signal the remote process to exit and do
c  a SYS$WAKE on our own process.
c
            ICOL_ASSIGNED(ICHAN) = -1
            ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),%VAL(WRITE_FUNCTION)
           1         ,IOSB(ICHAN),,,ICOL_ASSIGNED(ICHAN),%VAL(4),,,,)
            CALL SYS$WAKE (,)
           ELSE

c
c  Nope, not done yet.  Reissue the QIO for the next incoming packet
c  from this remote system.
c
            ISTAT = SYS$QIO (,%VAL(ICHANNEL(ICHAN)),
           1         %VAL(READ_FUNCTION),IOSB(ICHAN),%VAL(IO_RECEIVED_ADDR),
           1         %VAL(ICHAN),IREC_BUF(1,ICHAN),%VAL(ISIZE*4),,,,)
           ENDIF

           RETURN
           END


           SUBROUTINE CNVT_IOSB_STATUS (IOSB_QUAD, ISTATUS, IBUF_LEN)
c
c  Routine to pull the iosb status field out of the quadword (defined
c  as double precision for array convenience) and put it into the
c  integer field.
```

```
c
        INTEGER*2 IOSB_QUAD(4)
        INTEGER ISTATUS, IBUF_LEN

        ISTATUS = IOSB_QUAD(1)
        IBUF_LEN = IOSB_QUAD(2)
        RETURN
        END
```

- **REMOTE_4.FOR**

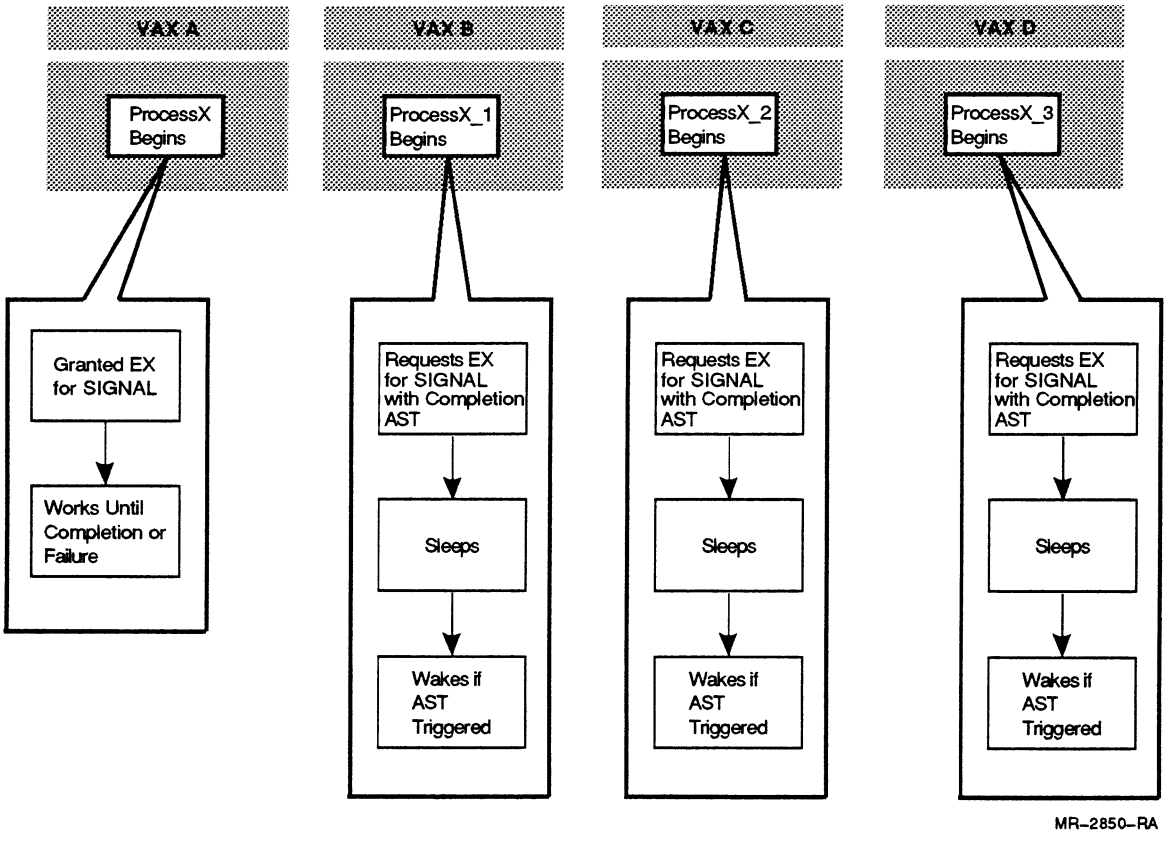
```
        PROGRAM MATMUL_REMOTE_TEST
c
c  This is the remote portion of this example program.  Most of this
c  program is identical to the previous, but adds the arbitration to
c  determine which of these remote systems will execute which column.
c
c  Note that since each remote process does not calculate the entire
c  output array and that only one column at a time is calculated, the
c  answers are put into a ISEND_BUF array that is only large enough
c  for a single column of data.
c
        INCLUDE 'PART4.INC'
        INCLUDE '($SSDEF)'

        INTEGER ISEND_BUF(ISIZE), MY_COL

c
c  Open the channel back to the local program.
c
        ISTAT = LIB$ASN_WTH_MBX ('SYS$NET',
        1                        IMAX_BUFFER,
        1                        IMAX_BUFFER,
        1                        ICHANNEL(1),
        1                        IGNORE )

        READ_FUNCTION = %LOC(IO$_READVBLK)
        WRITE_FUNCTION = %LOC(IO$_WRITEVBLK)


c
c  Start by getting all the data.
c
        DO J = 1,ISIZE
           ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),
        1       %VAL(READ_FUNCTION),IOSB(1),,,
        1       IN_ARRAY1(1,J),%VAL(ISIZE*4),,,,)
           IF (ISTAT .NE. 1) CALL EXIT (ISTAT)

           ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),
        1       %VAL(READ_FUNCTION),IOSB(1),,,
        1       IN_ARRAY2(1,J),%VAL(ISIZE*4),,,,)
           IF (ISTAT .NE. 1) CALL EXIT (ISTAT)
        ENDDO

c
c  Ok, all the data is here.  Ask the master process what column
c  is available for processing.  To do this, send a packet of length
c  of four to it.  (Note that the signal is the packet length, not
c  the contents of the variable being sent.)
c
100     ISTAT = SYS$QIO (,%VAL(ICHANNEL(1)),%VAL(WRITE_FUNCTION)
        1       ,IOSB(1),,,MY_COL,%VAL(4),,,,)

        ISTAT = SYS$QIOW (,%VAL(ICHANNEL(1)),%VAL(READ_FUNCTION)
        1       ,IOSB(1),,,MY_COL,%VAL(4),,,,)

        IF (ISTAT .NE. 1) CALL EXIT (ISTAT)
```

```
c
c  No more rows?  Exit.
c
        IF (MY_COL .EQ. -1) CALL EXIT

c
c  Got the column number.  Do this calculation and then return
c  the data.
c
   DO I = 1,ISIZE
     ISEND_BUF(I) = 0
     DO K = 1,ISIZE
        ISEND_BUF(I) = ISEND_BUF(I) + (IN_ARRAY1(I,K) * IN_ARRAY2(K,I))
     ENDDO
   ENDDO
c
c  Send this column back to the local process.
c
        ISTAT = SYS$QIO (,%VAL(ICHANNEL(1)),%VAL(WRITE_FUNCTION)
        1       ,IOSB(1),,,ISEND_BUF,%VAL(ISIZE*4),,,,)
        IF (ISTAT .NE. 1) CALL EXIT (ISTAT)

c
c  Go back for more.
c
        GO TO 100
        END
```

6.4.2 Recovery from Cluster State Transition Due to Node Failure

If the execution of a process is critical to an application, the process can be designed to have failover capability. Figure 6–7 demonstrates a method for ensuring a failover capability of ProcessX. If a hardware failure occurs on VAX A, causing ProcessX to release the EX lock for SIGNAL, then a backup process (ProcessX_1, ProcessX_2, or ProcessX_3) begins executing on a different VAXcluster CPU. The backup process at the top of the waiting queue acquires the EX lock for SIGNAL, triggering the AST. When the AST routine executes, the backup process is awakened to execute a recovery routine and then continue the work started by ProcessX. For a description of a BASIC application that has been designed to use this failover method, see Chapter 8.

Figure 6–7 A Process Designed for Failover in a VAXcluster System



7 VAXcluster System Performance Considerations

The multiple system components of a VAXcluster system are comprised of three primary system resources: CPU, memory, and I/O system. Depending on which of the system resources is most heavily used, computer applications are classified as: CPU-intensive, memory-intensive, and I/O-intensive. To ensure your VAXcluster system is properly balanced for the work that your application is performing, a system manager should understand the following:

- The capacity of the primary system resources in your VAXcluster system
- The demand placed upon these resources by the application workload

Typically, a system manager builds a historical perspective for system performance by monitoring performance over some representative period of time. Based upon an understanding of the "normal" performance of your VAXcluster system, you can work with the system manager to assess the nature of your application's performance attributes.

If your application places more demand on a resource than that resource can handle, a processing *bottleneck* occurs. One of the three system resources becomes the system-limiting or binding resource; that is, one of the three resources is being used to capacity before the other two. A bottleneck condition causes decreased throughput and increased response time.

It is essential to your application that you correctly identify the system-limiting resource. Ignoring any of the three primary resources in your investigation increases the danger that you will employ an inappropriate remedy for your application's performance problem. Once you have identified the system-limiting resource, you can consider three approaches:

- Increase the capacity of the limiting resource.

  This is the most obvious solution but may also be the most costly. Therefore, employ it only after the other two options have been investigated.

- Reduce the demand on the limiting resource.

  This may be very difficult to achieve if the limiting resource is being used to its capacity doing useful work. However, if the resource is ineffectively used, then this may be the easiest and the most cost-effective solution.

- Off-load the demand from the limiting resource onto one of the less used resources.

  This technique is possible because of the relationships that exist between the three primary resources; they are not autonomous. For example, excess memory capacity is often used to reduce the demand on an overworked I/O subsystem by increasing the size of each I/O, thereby reducing the total number of I/Os. The CPU benefits as well, because it needs to handle fewer I/Os.

The following sections of this chapter discuss:

- VMS utilities for identifying bottlenecks
- Three major types of bottlenecks that limit the performance of your application:
  - — I/O bottlenecks
  - — Memory bottlenecks
  - — CPU bottlenecks
- VMS layered products for analyzing VAXcluster application and system performance

7.1 Using VMS Utilities to Monitor the Cluster and Identify Bottlenecks

Monitor cluster performance carefully to ensure that the data flow is not hampered by:

- Inefficient resource usage
- An overloaded CPU

Table 7–1 lists several VMS utilities and commands available for monitoring a cluster. For more information on further documentation on these VMS utilities and commands, refer to Table 7–2.

In addition to these VMS utilities and commands, refer to Section 7.5 for more information on tools for identifying bottlenecks on your VAXcluster system.

Table 7–1 VMS Utilities and Commands for Monitoring a VAXcluster System

Utility	Function
VMS MONITOR Utility:	
MONITOR CLUSTER	Displays clusterwide information
MONITOR DLOCK	Displays distributed lock management statistics
MONITOR DISK	Displays I/O request rate or depth of I/O queue
MONITOR MSCP	Displays MSCP Server statistics
MONITOR RMS	Displays I/O activity for individual files
MONITOR PAGE	Displays hard and soft page fault rates
MONITOR PROCESSES/TOPFAULT	Displays which processes are generating most of the page faults
SHOW Commands:	
SHOW CLUSTER	Provides the view of the cluster as seen from a single node
SHOW DEVICE/SERVED	Shows information about MSCP-served disks on the local system
SHOW DEVICE/FULL	Shows the status of a device, and is useful for determining the configuration of disks
SHOW DEVICE/FILES	Lists files opened only from the local node
HSC SETSHO Utility:	
Using SET and SHOW with commands	Allows you to issue commands to display HSC characteristics

Table 7–2 Summary of VMS Documentation Resources for Identifying Bottlenecks

Resource	For More Information About...
Guide to VMS Performance Management	• Calculating disk response time • Managing system resources — Understanding the CPU resource — Understanding the memory resource — Understanding the disk I/O resource • Diagnosing resource limitations
VMS Show Cluster Utility Manual	Complete documentation on the SHOW CLUSTER utility
VMS DCL Dictionary	Description of the SHOW DEVICE command
HSC User Guide	Complete documentation on HSC utility programs
VMS Monitor Utility Manual	Complete documentation on the MONITOR utility

7.2 Potential Bottlenecks for an I/O-Bound Application Environment

While I/O requests may be for disk, network, or terminal I/Os, disk I/O bottlenecks are the focus of this section. I/O-bound applications are limited in both throughput and responsiveness by disk I/O operations. For example, a VAXcluster hardware configuration can only perform a certain number of disk I/Os per second, but an application is attempting to perform more than that number of disk I/O operations for every second of computation. Consequently, the application ends up waiting for disk I/Os to complete before proceeding with further computations. The I/O pathway (see Figure 7–1) of a VAXcluster system has several possible I/O bottlenecks:

- CPU's QIO processing
- Processor's I/O adapter (Ethernet or CI adapter)
- Medium between the I/O adapter and the controller (the Ethernet or the CI)
- Controller (HSC or MSCP serving VAX)
- Disk drive

CPU's QIO Processing

For a given I/O operation, the CPU's QIO processing that initiates the QIO request is on the order of hundreds of microseconds; at the most, the CPUs QIO processing is usually less than a millisecond.[1] While the CPUs QIO processing is a part of an I/O operation, the length of time required to complete this operation does not usually present an I/O bottleneck.

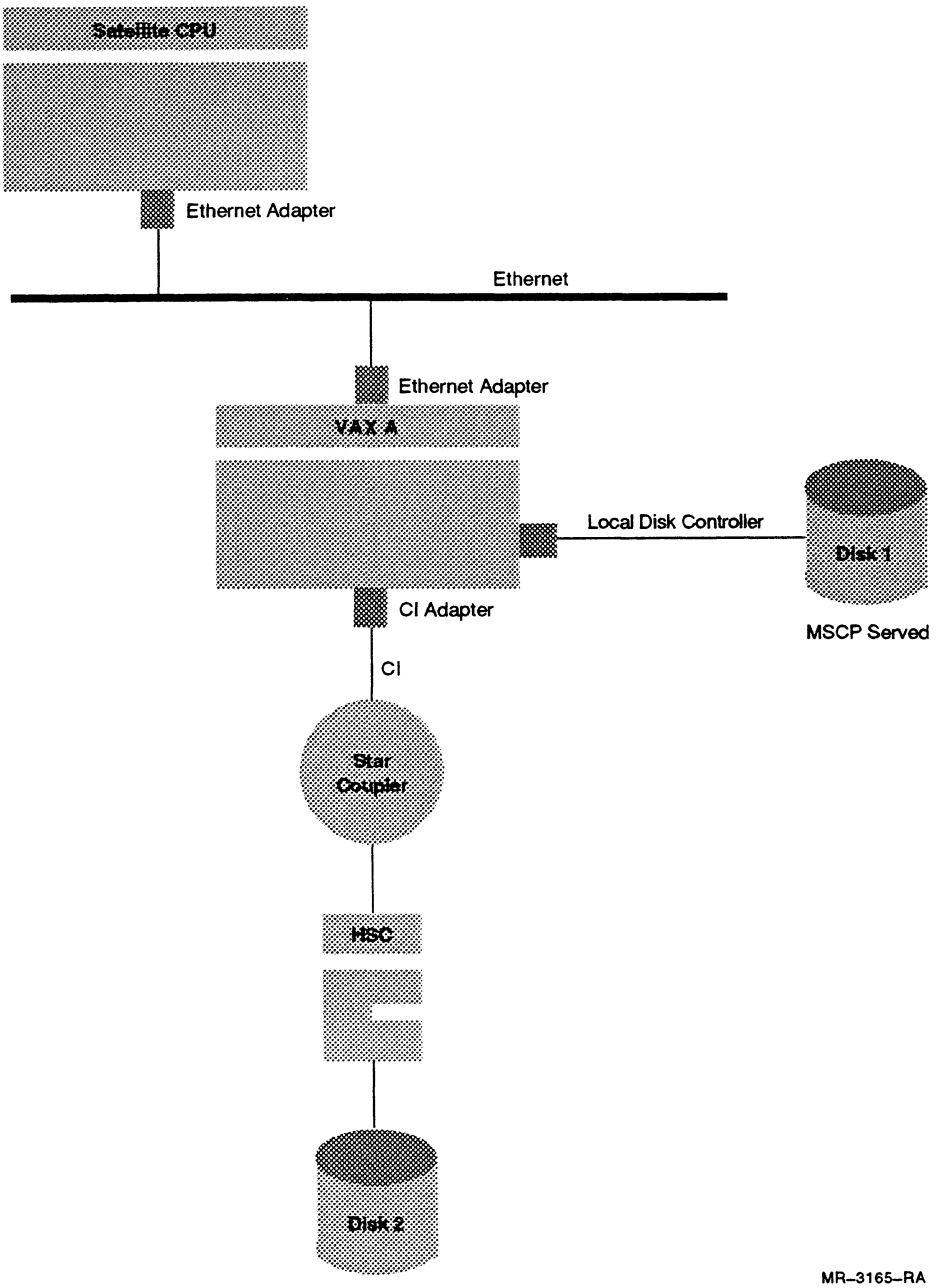
Processor's I/O Adapter

The respective attributes of the various types of Ethernet or CI adapters can impact your I/O throughput. For instance, when considering the Ethernet as the processor's I/O port with a medium-speed Ethernet adapter (DELUA), the transmission rate is about 0.7 Mbits per second. (For a disk block, this is less than 5.7 milliseconds.) When considering the CI as the processor's I/O port, with a slow CI adapter (CI780), the transmission rate is about 1.8 Mbytes per second. (For a disk block, this is less than 0.29 milliseconds.) For more performance information on Ethernet and CI adapters, see the *Guidelines for VAXcluster System Configurations* and the *VAX Systems and Options Catalog*.

The CI adapter is almost never a bottleneck in a slow VAX processor, but the likelihood of it being a bottleneck increases with the speed of the VAX CPU. For example, while larger CPUs are suitable for CPU-intensive applications, care should be used when considering these machines for applications with a significant disk I/O workload.

[1] QIO processing time is CPU dependent and can be greater than 1 millisecond on a VAX 11/780 or MicroVAX II.

Figure 7–1 I/O Pathways in a Mixed-Interconnect VAXcluster System



Medium Between the I/O Adapter and the Controller

The Ethernet and CI interconnects are clearly faster than the slowest Ethernet adapter or CI adapter; the Ethernet interconnect supports a transmission rate of 10 Mbits per second and the CI interconnect supports 70 Mbits (8.75 Mbytes) per second.

In theory, a large number of powerful VAX CPUs or workloads with high I/O demands could overload the CI adapter. However, the CI interconnect itself is very unlikely to be the primary I/O bottleneck.

The Ethernet interconnect may become a potential bottleneck because of its lower transmission rate and error checking protocols. The total aggregate of bandwidth of the Ethernet interconnect depends on the load generated by all nodes on all segments. For a very large and busy Ethernet, total useable bandwidth per segment is about 7 Mbits per second. Thus, you should consider the capacity of the Ethernet when configuring a cluster with many Ethernet-connected nodes or when the Ethernet is used to support a larger number of terminals or printers.

Controller

As with the CI interconnect, the HSC controller rarely becomes overloaded. An HSC *channel* could become overloaded if a large number of heavily used disks are connected to the same channel because each channel performs only one data transfer at a time.[2] In this case, you can use multiple channels and balance the I/O load between the channels to improve performance. When disks are dual-ported and the disk load is split between HSC channels, it is unlikely that the HSC controller will fall behind the demand.

HSC throughput depends on the block-size of the transfer. Small transfers are limited by the maximum I/O rate of each HSC:

- The HSC50 rate is about 550 I/Os per second.[3]
- The HSC40 rate is about 800 I/Os per second.[4]
- The HSC70 rate is about 1150 I/Os per second.[3]

[2] An HSC channel is also called a k.sdi module or a requestor (from the HSC side).

[3] With HSC Version 3.7 software.

[4] With HSC Version 3.9 software.

Large transfers are limited by the maximum transfer rate of the HSC's interface to the CI. For the HSC40, HSC50 and HSC70, this rate is about 3.8 Mbytes per second.

When most or all of the activity of an HSC controller originates from one CI-based VAX processor, the CI adapter of the VAX is more likely to be the limiting factor than the HSC controller. When most or all activity on a HSC is directed at one or a few disks, the disks themselves are more likely to be the bottleneck. The HSC controller is likely to be the bottleneck only when there is a heavy disk I/O demand from several VAXcluster CPUs and most of the disks attached to the HSC controller are very busy. In large VAXcluster systems, where several large VAX systems place heavy demands on most of the disks attached to an HSC controller, the HSC70 provides better performance than the HSC40 or HSC50.

In the case of the VAX MSCP Server using the local disk controller to enable clusterwide disk access to the local disk, typically a MSCP-served disk has over 90 percent of the performance of a local disk, provided sufficient memory is dedicated on the MSCP-serving CPU. However, if buffer allocations for the MSCP Server are not large enough, a larger percent of the disk access requests will encounter delays due to MSCP-server latency.

Disk Drive

When an I/O request does require action by a disk, this action consists of two steps:

1 Access

Access time is composed of *seek time*, the time it takes for the disk to first move its heads to the proper cylinder, and *rotational delay time*, the time it takes for the desired sectors to rotate under the heads. Typically for a 4 block transfer on a RA series disk, seek time takes about 25 milliseconds, and rotational delay time is about 8 milliseconds.

2 Transfer

The disk reads or writes data as the sectors move under the heads, and transfers the data from the disk to the controller or from the controller to the disk. For a 4 block transfer on a RA series disk using an HSC50 controller, there is about a 2 millisecond transfer time for data being transferred from the controller to the disk or from the disk to the controller.

While both of these steps take time, clearly the largest delay is the seek time required. The access and transfer time may vary depending on the type of CPU, disk drive, and controller. In addition, the following characteristics of the I/O requests also affect the access and transfer time:

— I/O size

A large I/O size contributes to the amount of time spent actually transferring data to and from disks.

— Seek distance

If I/O requests are randomly scattered over the whole disk, access times will be longer than if consecutive requests touch neighboring sectors.

— I/O rate

A large number of I/O requests typically means a large number of head movements. This contributes to the amount of time waiting for access to sectors on the disks.

In a VAXcluster system, disk drives are a common I/O bottleneck. The following are four common situations in which disk drives can be I/O bottlenecks.

- Multiple CPUs are making I/O requests to the same disk.

  Many systems and application have one or more files that are used intensively by many jobs on several VAXcluster CPUs. Files are often located on disks primarily on the basis of available space and the logical organization of directory trees. This can result in an I/O demand that is concentrated on one or few disks. These "hot" disks can become I/O bottlenecks.

  For small I/O transfers on RA series disks, the maximum average rate for acceptable performance is between 15 and 20 I/Os per second. On systems doing large I/O transfers, significant delays can occur at a lower I/O rate. Refer to the *Guide to VMS Performance Management* for information about how to calculate disk response time.

- Applications are requesting data that can cause a large number of seek operations, depending on the location of data on the disk.

  As files are created and deleted on a disk, the free space that started as one or two contiguous pieces becomes fragmented into numerous smaller areas scattered over the disk. Over time, the free space consists of holes left by the deletion of files. Some of these holes are large and some are small. When files are created or extended, the file system uses larger pieces of free space in order to keep the files as contiguous as possible. However, when a disk is nearly full, there are not enough large pieces of free space, and the files tend to be made up of many small pieces scattered over the disk. When reading or writing one of these fragmented files, the disk must make more head movements than it would if the file were one contiguous piece. For this reason, disks that are nearly full tend to become fragmented, and fragmented disks can be I/O bottlenecks.

- CPUs are added or upgraded to be faster causing more I/O requests per unit of time.

  When additional CPUs are accessing the same disk for reads and writes, disk allocation becomes less contiguous more quickly. If disk fragmentation becomes too great, performance suffers because extra seek operations are required to complete an I/O.

- An overloaded common system disk is used.

  In some large VAXcluster configurations with a large number of fast processors, a common system disk may be the cause of an I/O bottleneck if the aggregate amount of I/O requests exceeds the capability of that drive and its controller.

Summary of Performance Considerations for I/O in a VAXcluster System

The only items from the I/O pathway illustrated by Figure 7–1 that depend on the local processor are the CPU's QIO processing, the processor's I/O port, and the buffer size for the MSCP-controller. These aspects of the I/O pathway are rarely the point of an I/O bottleneck. The disk drive is most often the point of an I/O bottleneck.

In general, a VAXcluster system has a greater potential for overloading the disks than a single VMS system for the following reasons:

- Multiple VAXcluster CPUs can make simultaneous I/O requests to the same disk.
- Increasing the number of application users places more demand for concurrent I/O requests to the same disk.
- Faster CPUs can submit more I/O requests to an I/O queue for a disk.

Thus, running multiple copies of an I/O-bound application on different VAXcluster CPUs may not be beneficial if the disk drive is the cause of the I/O bottleneck. However, in the case where an I/O port is flooded by I/Os to many different disks, then running multiple copies of an I/O-bound application on different VAXcluster CPUs may be beneficial.

Typically, most I/O-bound applications are bottlenecked on the disk drive, not the processor's I/O port. There are VMS utilities for diagnosing the more common cases where the disk drive is the bottleneck; but, when an application is limited by the port, there is not a VMS utility to diagnose this problem.[5]

[5] However, the I/O load on the I/O port can be deduced by obtaining the I/O load on a disk and comparing this with the known I/O capacity of the disk.

In a LAVc system, there are two sources of I/O bottlenecks:

- System disk

  Nodes on a Local Area VAXcluster system often boot from the same system disk. Such a disk can become an I/O bottleneck in large clusters.

- Multiple satellite nodes

  As you add more satellite nodes to a Local Area VAXcluster system, the chances of an I/O bottleneck may increase because of a potential Ethernet overload.

For more information on how to use the VMS utilities discussed in Section 7.1 to identify and diagnose an I/O bottleneck, refer to the *Guide to VMS Performance Management*.

Solving an I/O Bottleneck

After you identify that you have an I/O bottleneck, you can take appropriate action to solve the bottleneck. For an I/O bottleneck, you might consider the following:

- Pre-allocate disk space when you create a file.

  Allocate enough disk space to store the file in one contiguous section of the disk. You should also consider allocating additional space in anticipation of file growth to reduce the number of required extensions. For more information on initial file allocation and extending a file, see the *Guide to VMS File Applications*.

- Regularly backup and restore disks to alleviate fragmented disks.

  By using the Backup utility to perform an image backup, all files on an output disk can be stored contiguously. Contiguous storage of files eliminates disk fragmentation and creates contiguous free blocks of disk space. For more information on how to use the Backup utility to alleviate fragmented disks, refer to the *Guide to Maintaining a VMS System*.

- Tune VMS RMS indexed files.

  The process of designing your files to achieve better processing performance is called *file tuning*. You can use the Analyze/RMS_File utility to examine the internal structure of a VMS RMS file. Analyze/RMS_File can check the structure of a file for errors, generate a statistical report on the structure and use of the file, or generate a File Definition Language (FDL) from a data file. If you want to generate a reformatted and reorganized output file, you can use the VMS Convert utility and specify a FDL file specification (obtained from the Analyze/RMS_File utility) on the Convert utility command line. For more information on how to use the Analyze/RMS_File utility and Convert utility to optimize and redesign file characteristics, refer to the *VMS Analyze/RMS_File Utility Manual*, *VMS Convert and Convert/Reclaim Utility Manual*, and the *Guide to VMS File Applications*.

- Utilize VMS RMS global buffering.

  In cases when several processes on different VAXcluster CPUs access a file for mostly shared reads and there are a limited number of update requests, consider implementing VMS RMS global buffering. When a process requests a record that is located in a VMS RMS global buffer, the record can be transferred directly from the global buffer to the program, eliminating an I/O read operation. Note that if the previous accessor modified the record, VMS RMS writes the buffer to disk before returning the record to the new accessor.

  For more information on VMS RMS global buffering, refer to Section 3.2.3 in this manual and the *Guide to VMS File Applications*.

- Disable highwater marking unless you need it for disk read security for a high volume of files.

  Highwater marking sometimes increases the number of disk I/Os. This is particularly true in fragmented volumes supporting applications that do a lot of file extensions and creations. If you need disk read security for only a few files, use SET FILE/ERASE_ON_DELETE. This provides much of the same level of security as highwater marking, but has less impact on performance.[6]

- Move busy files to other disks. For example:
  - — Move all nonsystem files from a system disk to other disks.
  - — Place page and swap files on a less used locally-connected disk, if available. This reduces the I/O load on the cluster common disks.
  - — Move frequently accessed system files to other disks, and use logical names or other pointers to access them. These files include user authorization files, a mail database, a job controller database, ERRFMT log files, DECnet-VAX accounts, and MONITOR log files.

- Devote more memory to I/O using:
  - — Larger or more VMS RMS buffers
  - — Disk file Read-Only global sections (see Section 6.2.3)
  - — Increased MSCP Server buffers after using MONITOR MSCP to determine the necessity or AUTOGEN's feedback mechanism will automatically set the appropriate values for MSCP buffers

[6] With VMS Version 5.0 and higher, if you specify the sequential only option (SQO) for access to a sequential file, you get optimizations for sequential access that includes highwater marking.

- Balance the I/O workload across multiple disks:
  - — In a CI-based cluster, connect multiple disks to HSCs and balance the I/O load between HSC channels. An HSC optimizes I/O when it has a queue of requests by:
    a. Performing I/O operations in the most efficient order, not necessarily the order in which they are requested
    b. Breaking an I/O request into fragments, if necessary, for efficiency

    However, instead of relying on HSC optimizations in such a situation, you can substantially improve performance by adding a disk and diverting some of the I/O load to that disk.
  - — Use multiple HSC disk channels.

    In a shadow set, you should use multiple disk channels to avoid a single point of failure as well as improve performance. When you use multiple channels, you can perform multiple data transfers in parallel.

    (Although an HSC channel can accept up to four disks, performance may tend to degrade if more than two busy disks are on the same HSC channel.)
  - — Use volume sets for automatic load balancing when the write activity is high.

    In a volume set, each new file is placed on the volume with the most free space. For more information on how to use volume sets for load balancing, see the *Guide to Maintaining a VMS System*.
  - — Use volume shadowing if the read activity is high.
  - — Use search lists to combine directories on different disks into one logical directory. (For more information on how to use search lists, see the *VMS Install Utility Manual*.)
  - — Connect a local disk to the CPU that will access it the most.
- Whenever possible, run batch jobs with high I/O activity at non-peak hours to lessen the impact on users during the peak hours.

For more information on identifying program and configuration I/O bottlenecks, refer to Section 7.5 in this chapter and the *Guide to VMS Performance Management*.

7.3 Potential Bottlenecks for a Memory-Bound Application

Memory is typically the least expensive and the most versatile of the three major resources. Off-loading demand from the other resources at the expense of memory is a common trade-off inherent in the design of all virtual memory systems. At the same time, the overcommitment of memory manifests itself both as a CPU bottleneck (excessive interrupt stack time for hard paging and swapping) and an I/O bottleneck (contention for the disk containing the page and swap files). The end result of a memory bottleneck is that a CPU has unacceptable application response times. For more information on using AUTOGEN and SYSGEN operations to set the values of system parameters (such as the page and swap files), refer to the *Guide to Setting Up a VMS System*.

Hardware or software upgrades to CPUs in your VAXcluster system can also have an impact on CPU memory requirements. Consider the following precautions when you make changes to CPUs in your VAXcluster environment:

- Upgrading to a faster CPU

  Assuming that you are upgrading because the CPU is a bottleneck, be careful when you assess the memory requirements, since a faster CPU will reference more pages in a given unit of time.

- Upgrading a single CPU system to a VAXcluster system

  Static memory requirements will typically be from 1 to 2 Mbytes of per CPU memory over and above single CPU usage.

- Software upgrades or additions

  Assess the impact of the memory requirements for layered products that you may want to add to your VAXcluster system.

For more information on the hardware and software memory requirements, refer to the current *VAX Systems and Options Catalog* and the appropriate *VMS Software Product Description*.

Solving a Memory Bottleneck

The key to good performance of the memory subsystem is to maintain properly sized working sets for the resident processes on each VAXcluster CPU. As a rule, the total of all process working set quotas, for each VAXcluster CPU, should be within the amount of free memory available on that system. For more information on working set adjustment, refer to the *Guide to VMS Performance Management*.

If you increase the number or size of the I/O buffers at the application level by adjusting the VMS RMS buffer parameters, make sure that you adjust the working sets for these processes accordingly or you run the risk of trading file I/O for page faults.

Install images with the appropriate attributes. When an image is accessed concurrently by more than one process on a routine basis, the image should be installed with the Install utility using the /SHARED qualifier so that all processes on a VAXcluster CPU use the same physical copy of the image. In addition, you can conserve on memory allocated for an installed image with a directory pointer to the disk file location for the image by installing the image as /OPEN/HEADER_RESIDENT/SHARED. For more information on how to use the Install utility, refer to the *VMS File Definition Language Facility Manual.*

You may also want to consider adding memory to reduce the amount of I/O required in applications, thus improving overall performance. In order to improve application performance, dynamic memory could possibly be added into the VAXcluster storage hierarchy at two levels:

- Adding main memory at the host level

  Main memory provides the fastest access to data; consequently accessing data from memory rather than a magnetic storage device can improve system throughput and response time by reducing an application's hard page fault rate. However, because main memory is volatile, use of this resource is not a viable alternative in applications where data is permanent and must outlive the program.

- Adding memory at the device level

  A solid state disk, for example the Electronic Storage Element-20 (ESE20), is a device level option for adding memory and enhancing I/O performance. The ESE20 operates at speeds of five to ten times that of magnetic storage devices and is characterized by access times in a range of 1 to 10 milliseconds. Consequently, an ESE20 can be used to decrease the disk I/O access time and significantly speed up a disk-based application. In addition, the ESE20 is a full MSCP protocol disk device that be accessed through an HSC, supports VAX Volume Shadowing, and is transparent to the application. For more information on Digital solid state disks, see the *VAX Systems and Options Catalog*.

For more information on identifying program and configuration memory bottlenecks, refer to Section 7.5.

7.4 Potential Bottlenecks for a CPU-Bound Application Environment

CPU performance is vital to the performance of the cluster as a whole because the CPU provides instruction execution service to user processes. Therefore, it is important to keep the CPU doing useful work; not scheduling over-committed memory or handling a large number of I/O requests.

A CPU can become overloaded when:

- Multiple applications continually need the CPU for computations.
- Multiple applications need to move a large amount of data within memory.

- You are trying to run a very large application on a CPU that is not powerful enough to run the application efficiently.
- Excessive CPU cycles are required to manage memory.

A consideration for a CPU-bound environment is the amount of locking in an application and the economy of the locks used. If there are extensive interprocess communications for an application using DECnet-VAX communication, then alternatives using locking techniques should be explored because locks have a lower overhead than DECnet-VAX communication.

When using locks, there are two strategies for reducing the VMS overhead associated with the internal data structures for the locks:

1 If a resource is accessed and released numerous times during a program's execution, then the lock should not be $ENQ—$DEQ for each access. Rather, the lock should be initially $ENQed and then converted to the NULL locking mode and reconverted for each subsequent access request. This strategy is useful for conserving locking data structures.

2 When possible, construct a locking scheme to use resource sublocks to reduce lock manager data structures and improve locking performance.

For more information on isolating a CPU limitation in your VAXcluster system, refer to the *Guide to VMS Performance Management*.

Solving a CPU Bottleneck

For a CPU bottleneck, consider the following:

- Use a more powerful CPU.
- Add one or more CPUs to the cluster.

  If you are considering adding CPU power to your VAXcluster system, either by upgrading or by adding a new CPU, also consider the effect that the additonal CPU power will have on the other primary resources. A change in the balance of resources will occur and may result in a shift in the bottleneck. For example, given that the CPU is the resource limiting the performance of your system, removing this binding condition allows more I/O requests to be generated per unit of time. This may result in an I/O bottleneck at a disk.

- Design an application to run on any CPU in the cluster so the application can be offloaded from an overloaded CPU. (For more information on designing an application for high availability, refer to Section 5.1.)
- If possible, break a large application into discrete components, and run each component on a different CPU. (Refer to Section 2.5 and Section 4.3 for more information on how to distribute by decomposition.)

- Minimize process creations.

  Only create a process if there is a significant amount of work to do. You can use server processes (see Section 4.2) as a means of minimizing process creations for remote requests.

- Minimize image activations.

  Only activate an image if you intend to do a reasonable amount of work; once you activate an image, stay there. For every image activation, there is an image rundown which can be a CPU-intensive operation.

- Use the VMS batch facility.

  The job controller balances the job load across CPUs. If a generic batch queue is associated with more than one execution queue, the batch load is proportional to the job limit of each queue.

- Use terminals connected to a terminal server on the Ethernet.

  Terminal servers balance the user load across CPUs. Each CPU computes a service rating based on the type of CPU and the percentage of idle time over a recent interval. The terminal server then connects each user to the CPU with the highest rating service.

  Terminal servers also allow manual load balancing. Users can log into whichever node they choose; usually the nodes with the best response time.

For more information on identifying program and configuration CPU bottlenecks, refer to Section 7.5.

7.5 Layered Products Available for Monitoring Cluster Performance

Table 7–3 lists three layered products available for monitoring cluster performance. The features of these products are briefly described in the text following Table 7–3.

Table 7–3 Layered Products for Monitoring a Cluster

Product	Function
VAX Software Performance Monitor (SPM)	Collects and reports clusterwide performance data in more detail than VAX MONITOR does.
VAX Performance and Coverage Analyzer (PCA)	Gathers data on a running program, and produces performance histograms and tables.
VAX Performance Advisor (VAX PA)	Identifies possible CPU and cluster problems, and provides recommendations for enhancing performance.

7.5.1 VAX Software Performance Monitor

VAX Software Performance Monitor (VAX SPM), Version 3.2, is a software performance management facility for VAX and VAXcluster systems. It collects, displays, reports, and graphs performance information useful in system tuning, trend analysis, and workload forecasting. This information includes resource utilization and load balance data for a single VAXcluster CPU or a VAXcluster system. VAX SPM software is designed for use by system managers and system programmers.

VAX SPM provides a flexible facility for collecting and archiving performance data. Data may be collected by using a variety of user-specified parameters. The user can start and stop data collection for all CPUs in a VAXcluster system from a single terminal, and can archive all the performance data in a single file.

VAX SPM video displays dynamically show a variety of statistics using bar charts, Kiviat plots, and numerical data. These can display data for a single node, or for summary information for all nodes or accessible disks (up to eight concurrently) in a VAXcluster system. The latter is particularly useful for balancing the workload across several nodes or disks.

VAX SPM tabular reports and graphs contain the level of detail necessary to quantify system resource utilization (CPU, memory, and I/O) and to identify system performance bottlenecks. Proper analysis of these reports may reveal under-utilized resources that can be traded against a bottleneck. Graphs and reports of data collected over long periods of time show long-term trends in resource utilization. These are helpful in planning future hardware acquisitions.

For more information on VAX SPM, refer to the *Guide to VAX SPM*.

7.5.2 VAX Performance and Coverage Analyzer

The VAX Performance and Coverage Analyzer (VAX PCA) is a tool to help VMS users analyze the execution behavior of their applications programs. VAX PCA has two functions:

- To pinpoint execution bottlenecks and other performance problems in application programs
- To provide test coverage analysis by measuring what parts of a user program are executed or not executed by a given set of test data

The VAX PCA is an aid in tuning the performance and testing of applications programs. It is not a tool for the analysis of operating system performance or for use as an aid in hardware resource planning.

The VAX PCA consists of two parts:

- Collector

  The Collector gathers performance and test coverage data on a running user program.

- Analyzer

  The Analyzer processes and displays the collected data.

Both the Collector and the Analyzer are fully symbolic and use the Debug Symbol Table (DST) information in the user program to access the symbolic names of program locations. Applications written in any of the VMS languages which produce DST information can be analyzed by using the VAX PCA.

For more information on VAX PCA, refer to the *VAX Performance and Coverage Analyzer User's Guide.*

7.5.3 VAX Performance Advisor

The VAX Performance Advisor (VAX PA) is a VMS-layered product that reduces the time and effort required to manage and monitor VMS system performance, as well as plan for future resource requirements. It can be used with both standalone VAX systems and VAXcluster systems.

VAX PA gathers VMS system data and, through the application of expert system technology, analyzes the data, identifies specific conditions causing performance degradation, and presents detailed evidence to support its conclusions. Further, VAX PA provides recommendations for attaining improved system performance.

In addition to its expert system analysis, the VAX Performance Advisor assists in capacity planning exercises by providing data archival and graphing capabilities for long term trend analysis, and performance modeling to determine future system performance given changes in workload or configuration.

The components of VAX PA are:

- Performance Knowledge Base and Rule Compiler

  The VAX Performance Advisor contains a knowledge base of rules and thresholds which it uses to analyze VAX/VMS system data. VAX PA rules fall into five categories: Memory, CPU, I/O, Cluster, and Miscellaneous. The VAX PA knowledge base may be modified and expanded at the user's discretion.

- Analysis, Reporting, and Graphing

  VAX PA aids the system manager in monitoring system activities and in making performance evaluations by quickly identifying performance problems. Through its analysis, VAX PA will also identify potential bottlenecks and the specific device on which the bottleneck will occur. VAX PA reports are generated at the request of the user and include: the Analysis Report, the Performance Evaluation Report, VAX PA Graphs, and the Raw Data Dump Report.

  — Analysis Report

    The VAX PA Analysis Report contains conclusions drawn from the VAX PA analysis as well as recommendations for improving system performance. In addition to identifying performance related problems and recommendations, the Analysis Report contains the conditions under which the identified problems occurred, along with supporting evidence to substantiate VAX PA conclusions.

  — Performance Evaluation Report

    The Performance Evaluation Report provides system statistics which can assist the system manager in gauging the impact of changes made to the system. It is particularly useful in monitoring system performance after implementation of a VAX PA recommendation.

    This report provides summaries of disk and tape activity, CPU and memory utilization, as well as detailed statistics on workload data, interactive, batch, and network processes.

  — Graphs

    VAX PA provides a facility to graphically represent the data which has been collected in the VAX PA database. VAX PA graphing provides the system manager with "pictures" of the system's performance metrics, and is a valuable source of information from which a better understanding of resource utilization and overall workload characteristics can be obtained. A wide range of predefined graphs plus the ability to define custom graphs can be generated for a ReGIS or ANSI output device.

  — Dump Report

    This report provides data from the VAX PA database in user readable format. The user may optionally choose to dump the full database record, or only a selected portion of the database record.

- Workload Characterization, Performance Modeling, "What If" Analysis, and Prediction Reporting

  VAX PA allows the system manager to define the system's total workload in terms of manageable units which VAX PA will then report against. Workloads and groups of workloads (workload families) are used in model generation and can be reported against in VAX PA's Performance Evaluation Reports or Graphics.

The VAX PA modeling component is used to predict performance of standalone or VAXcluster systems, and to determine system performance levels for various workloads and configurations. The performance statistics are provided in either summary or detailed reports and include:

— Resource utilization

— Response time

— Throughput per transaction class as well as aggregate

— Performance information for each CPU, HSC, disk, channel, adapter, and CI or Ethernet bus for both current and projected workloads or configurations

— System level parameters

When using the performance modeling component of VAX PA, a baseline model is first generated from the current data in the VAX PA database and can be validated against the data in the Performance Evaluation Report. After validating the baseline model, the user can change parameters in the model input file through interactive prompting or by directly editing the model input.

This process can be repeated as often as necessary until the user is satisfied with the performance of a given configuration and workload. This is often referred to as "What If" analysis, as it allows the user to answer questions such as "What are the performance implications to my system if I increase workload x by 20%?", or "What if I add another CPU to my VAXcluster system?"

VAX PA provides a prediction reporting capability which automatically determines system performance based on an incremental workload of 25, 50, 75, and 100 percent, or until a system component becomes saturated. This report also indicates the smallest workload at which a component will saturate, defined by 90% utilization. Prediction Reporting identifies bottlenecks before they occur, thereby allowing the system manager to be proactive in eliminating bottlenecks and providing a consistent level of system performance to the user base.

- Data Archiving and Data Extraction

  VAX PA provides data archiving capabilities so that the VMS performance data collected by VAX PA on a daily basis can be used in long-term performance studies.

  VAX PA provides data extraction procedures which may be called explicitly from user written programs. This facility allows system and application programmers to call VAX PA library procedures for extraction of daily VAX PA data. VAX PA provides record definitions for the following languages: Ada, Basic, Bliss, C, Datatrieve, Fortran, Lisp, Macro, Pascal, and PL/I.

- Data Collection and Storage

  VAX PA records VMS system data for subsequent processing by VAX PA. The data collector runs as a detached process and is activated at system startup. VAX PA supports data collection and reporting for a maximum of 1024 concurrent processes.

  In a VAXcluster system, VAX PA uses the distributed VMS lock manager to synchronize the data collection on all nodes. All data records will then contain a time stamp that is consistent across the entire VAXcluster system. The database files have a default retention period of seven days unless otherwise specified by the user.

For more information on VAX PA, refer to the *VAX Performance Advisor User's Guide*.

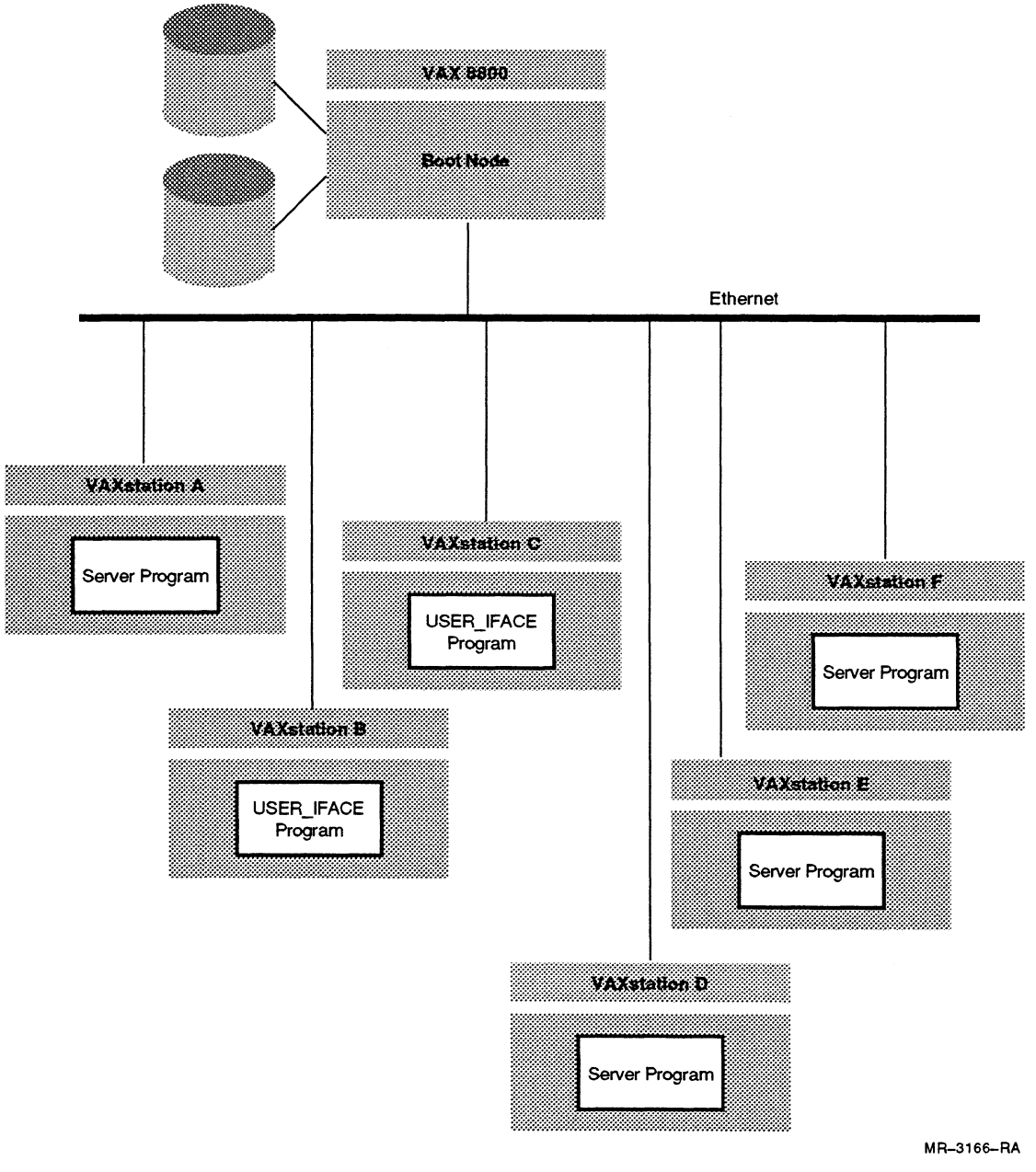
8 Sample Application for a VAXcluster System

The following demonstration application is written in BASIC and consists of two components:

- A user interface program (USER_IFACE.BAS) that prompts a user to send a message to a server process on a particular node
- A server program (SERVER.BAS) running on multiple nodes

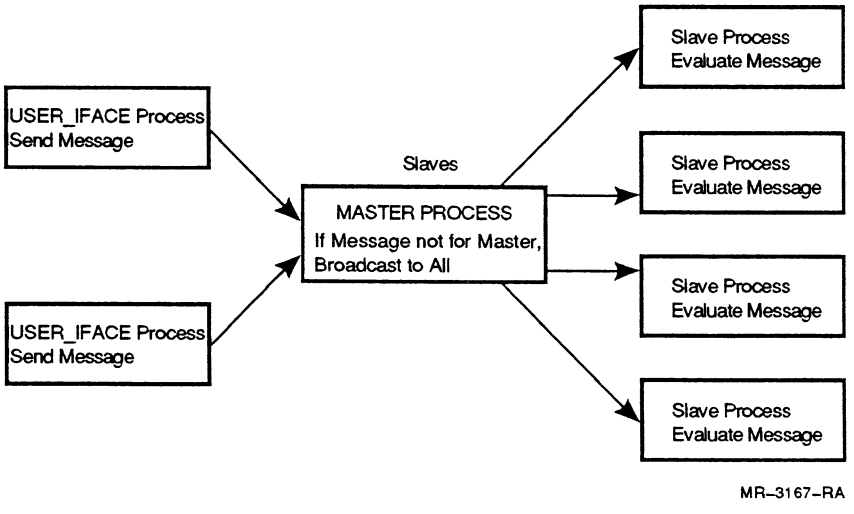
Figure 8–1 shows a diagram of the application components on a Local Area VAXcluster system.

Figure 8–1 Diagram of a Demonstration Application



When the application is executed, multiple processes running USER_IFACE.EXE can send messages to any one of the processes running SERVER.EXE in the VAXcluster environment. One of the server processes has been granted an EX mode lock for resource MAST; this server is called the "master." The other server processes have queued an EX mode lock for the MAST resource; these server processes are called the "slaves." All processes running USER_IFACE.EXE from any VAXcluster CPU always "talk" to the server process which is currently functioning as the "master." The "master" determines if the message sent by a user process is for itself; if not, the message is broadcast to all slave processes. Upon receipt of the message broadcast from the master, each slave process evaluates the message to see if it is for them. Figure 8–2 illustrates the communication pathways for the demonstration application.

Figure 8–2 Function of Demonstration Application



The demonstration application illustrates:

- A distributed application

  The user interface runs on one or more nodes; the server program runs on one or more other nodes.

- High availability

  This application demonstrates a "master" server that is designed for high availability using redundant back-up servers. As long as there is at least one process that is executing SERVER.EXE, there will be a "master" process to coordinate communications.

- Use of lock management system services with completion and blocking AST routines

  This demonstration application has several examples of using completion and blocking AST routines to implement Deadman and Doorbell locking schemes.

- Communication between nodes

  The two components communicate through the lock value block, and therefore the messages must be 16 or fewer bytes.

- Reliability

  Using the lock management system services with the lock value block, a communication protocol is designed so that a message is never lost or overwritten.

- Failover

  If the master process fails, another server gets the lock on the resource and becomes the new master.

8.1 Application Design

There are two phases for this application:

1 Run SERVER.EXE to set-up the application environment. (Figure 8–3 demonstrates the set-up activities by SERVER.EXE.)

2 Run USER_IFACE.EXE to communicate to a process in the application environment.

  There are two phases for a communication from a user interface process:

  a. Send the message from any user interface process to the "master." (Figure 8–4 demonstrates these communication activities.)

  b. If the message is not for the "master," the "master" sends the message to all slave processes in the application environment. (Figure 8–5 demonstrates these communication activities.)

Note on Figure 8–3

The SERVER.EXE set-up activities perform the following steps:

1 The first process to execute SERVER.EXE acquires the EX lock for resource MAST. All other processes in this application have queued an EX lock with a completion AST (MAST_AST routine) for resource MAST. The first process is the "master" and the remaining processes are the slaves. (The EX requests for resource MAST in the lock manager's waiting queue is the order of succession for failover to the "master" role.)

2 The "master" executes the completion AST routine (MAST_AST routine) when the EX lock for the MAST resource is granted. The "master" assumes the role of being the "special" process that the user interface will talk to (see Figure 8–2) by acquiring the EX lock with a blocking AST for resource UI.

3 All processes (except the master) that run SERVER.EXE acquire a CR lock with a Blocking AST (RR_AST routine) for resource RR.

Note: In this demonstration application, after SERVER.EXE completes all of the set-up activities, the program SERVER.EXE goes to sleep and all of the AST and Blocking AST activities are performed while each SERVER.EXE process remains asleep. In a "real world" application, SERVER.EXE could include code for work for each server process to perform when awakened from an AST routine. For example, when a server (either the master or a slave server) is sent a message from the user interface process, the appropriate server could be awakened from AST level and perform code instructing the server to read a disk file for further instructions for work to be performed.

Figure 8–3 SERVER.EXE Set-Up Activities

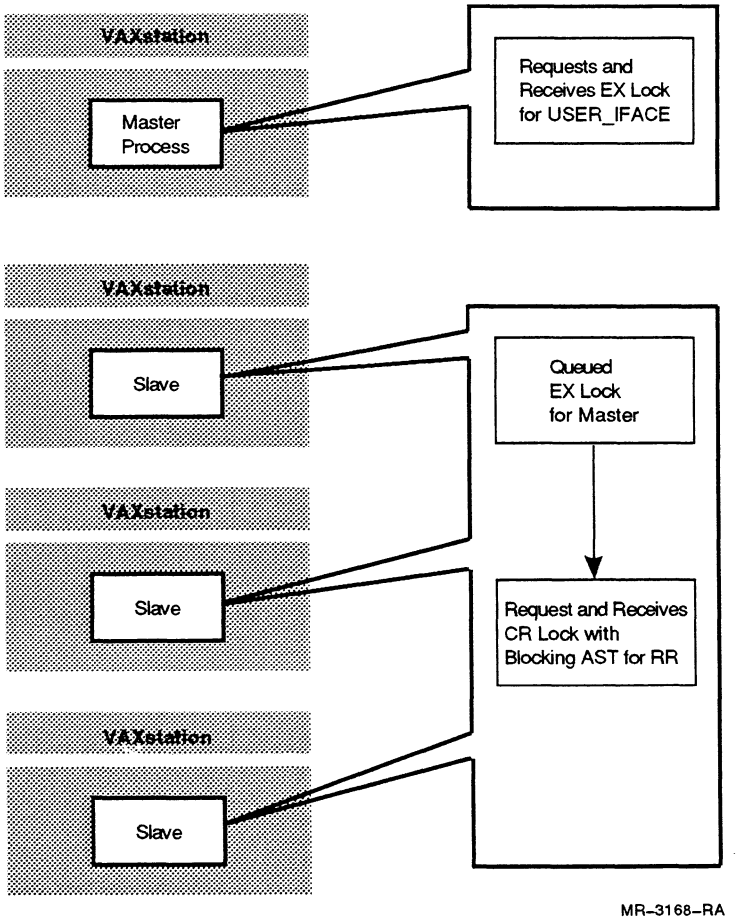
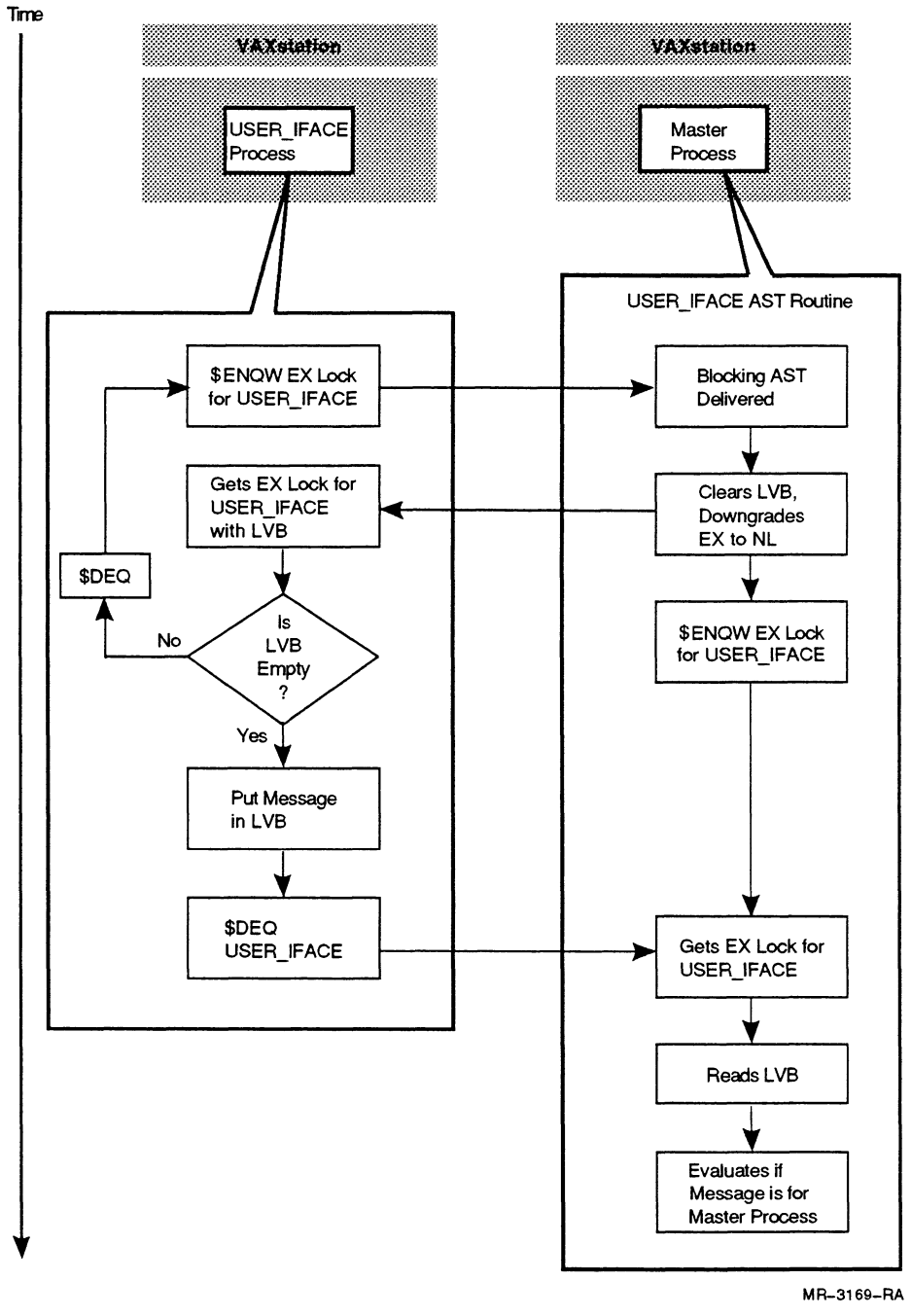


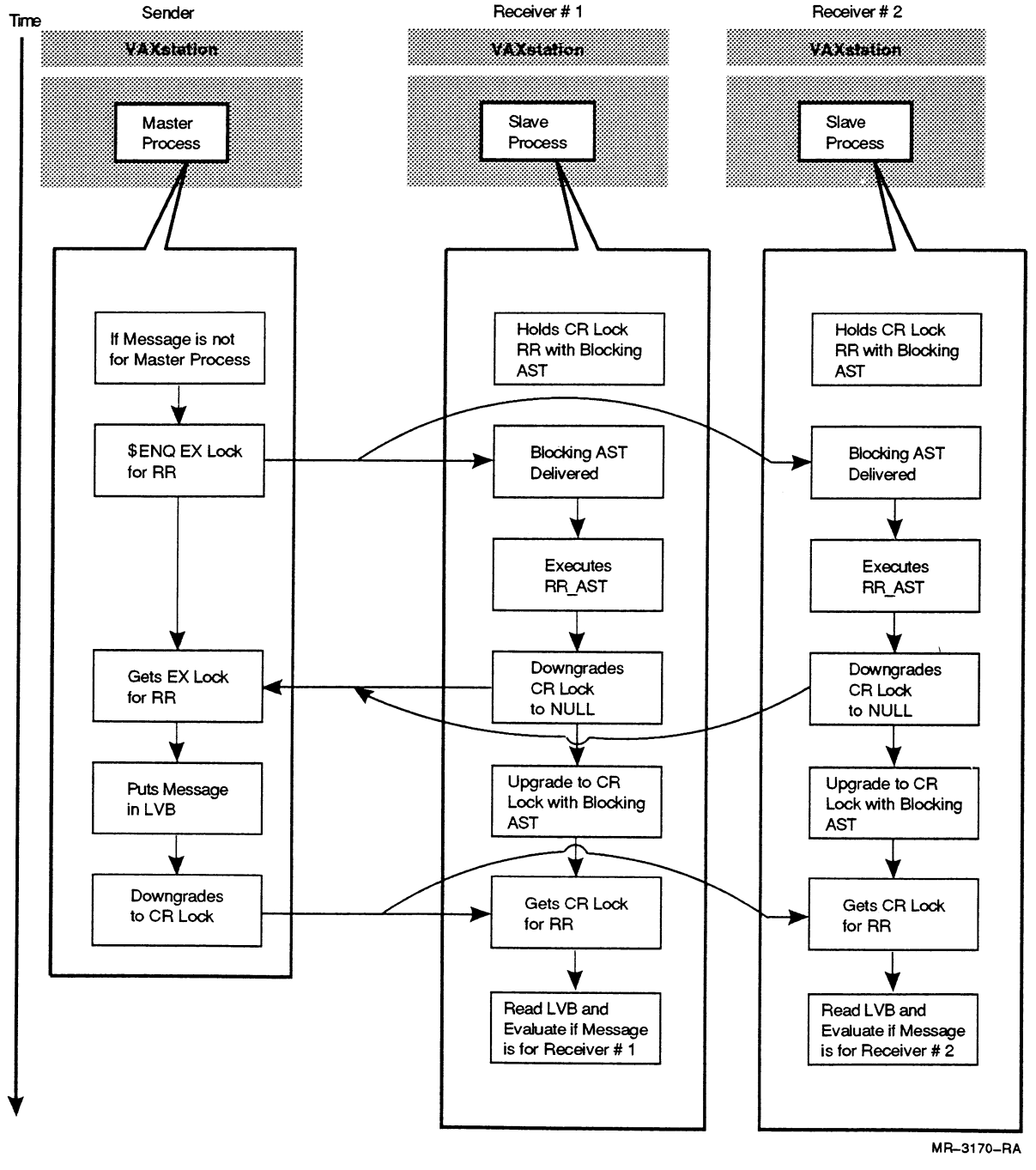
Figure 8–4 USER_IFACE.EXE Message Sending Activities



Note on Figure 8–4

The design of this demonstration application of having all user interface processes talk to one server (the "master") may be the best implementation if most of the messages sent by the user interface processes are for the "master." However, if a large number of messages are for other server processes, you may consider modifying this application so that those user interface processes can broadcast directly to all servers. In addition, this demonstration application also assumes that the master process will only send messages to the other servers and will not need to receive messages from the other servers.

Figure 8–5 RR_AST Routine Broadcast Protocol Activities



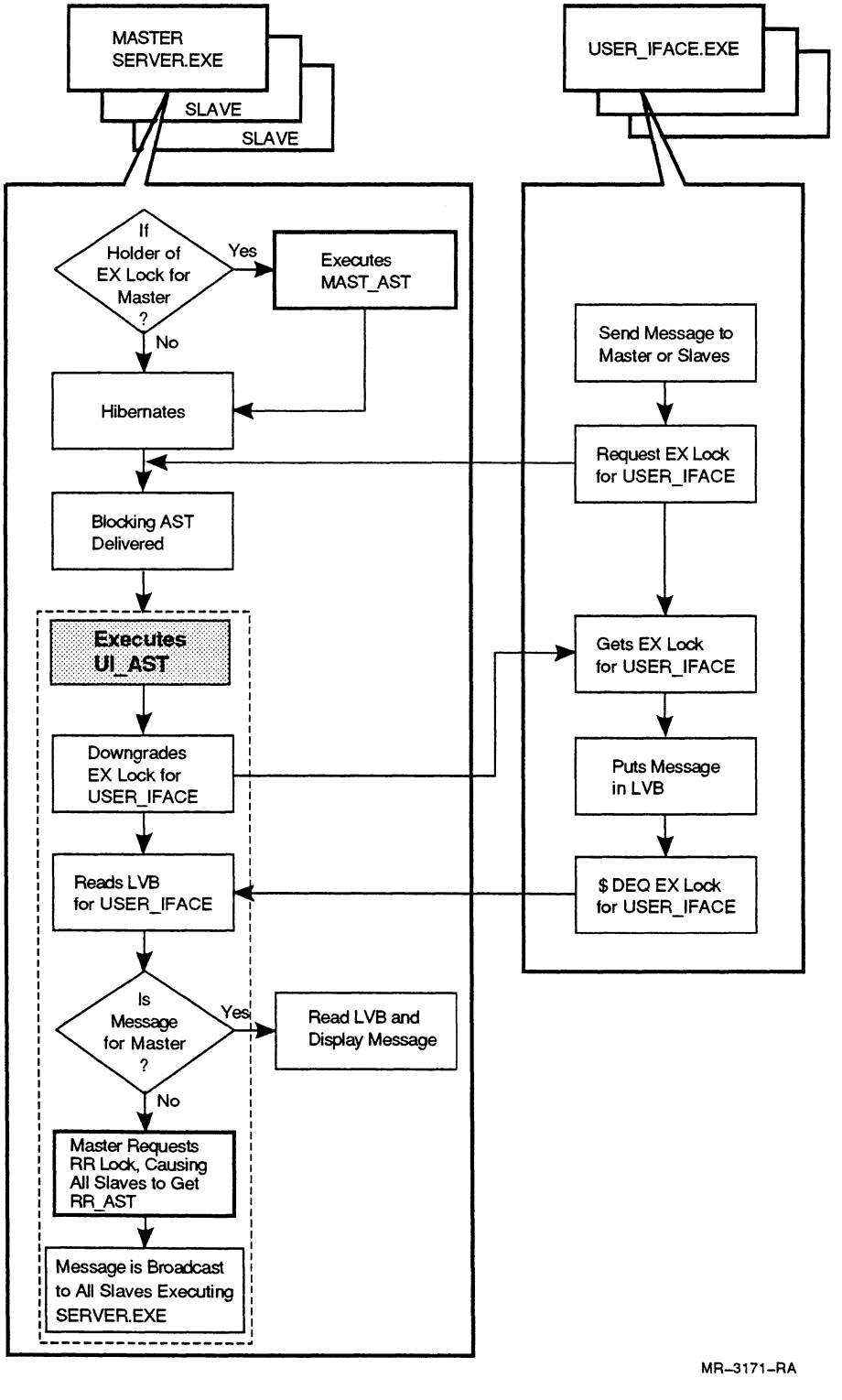
Note on Figure 8–5

All Receivers must be able to downgrade CR for RR to NL. If for some reason, a server process cannot downgrade, then the communication will hang. (There is no problem with the communication if a server process leaves the application due to a hardware failure. Only in the case where a server process is hung on a node, but the node remains in the cluster.)

8.2 Application Implementation

A flowchart of the modules of this application is presented in Figure 8–6. The complete BASIC code for all modules follows. In addition, there are compile and link instruction in BUILD_LOCK_EXAMPLE.COM.

Figure 8–6 Modules for Demonstration Application



BUILD_LOCK_EXAMPLE.COM

```
$       !BUILD_LOCK_EXAMPLE.COM - Command procedure to build the example
$       !
$       !
$       CREATE LCKDEF.MAR

        ;$ENQ/$DEQ System Service parameter definitions

        $LCKDEF GLOBAL  ;Make them global definitions.


        .END            ;That's all folks.
$             !Assemble the definitions.
$       MACRO   LCKDEF
$             !Compile the subroutines and programs.
$       BASIC/LIST      MAST_AST
$       BASIC/LIST      RR_AST
$       BASIC/LIST      UI_AST
$       BASIC/LIST      SERVER
$       BASIC/LIST      USER_IFACE
$             !Link the programs.
$       LINK    SERVER, MAST_AST, RR_AST, UI_AST, LCKDEF
$       LINK    USER_IFACE, LCKDEF
$             !All done.
$       EXIT    !from BUILD_LOCK_EXAMPLE.COM
```

LCKDEF.MAR

```
        ;Lock services definition
        $LCKDEF GLOBAL
        .END
```

SERVER.BAS

```
 1      PROGRAM SERVER
        %TITLE "Server program"
        %IDENT "V1.00"
10      option type=explicit

        external long function sys$enqw,sys$hiber,sys$enq,sys$trnlog,&
                lib$get_ef,lib$free_ef
        external long constant rr_ast,mast_ast  ! AST routines

        external long constant          &
        lck$m_system,   &
        lck$m_valblk,   &
        lck$m_convert,  &
        lck$k_nlmode,   &
        lck$k_crmode,   &
        lck$k_exmode,   &
        ss$_normal

        declare long ret_status,     ! for system service status returns &
                mast_flags,rr_flags, ! flags modifiers for locks &
                pos_colon,           ! position of ":" in sys$node string &
                eflag1,eflag2        ! event flags

        declare string node     ! Our node, obtained from logical SYS$NODE

        ! Maps for mast and rr lock blocks.
        map (mast_lock$block) long mast_lblk(6%)
        map (rr_lock$block)   long rr_lblk(6%)

        ! These two are in common so they can be shared with AST routines.
        common (scommon) long master,string our_node=6%

        master=0%               ! If 1 (set by MAST AST) means we are
                                ! the master.
```

```
! Get an event flag to use for system services.
ret_status=lib$get_ef(eflag1)
if ret_status<>1% then &
        print "error getting an event flag: ";ret_status
        goto 300
end if

! Find our own nodename by translating logical SYS$NODE.
node=space$(12%)        ! Lots of room for _ and ::
ret_status=sys$trnlog("SYS$NODE",,node,,,)

if ret_status<>1% then &
        print "error getting our own node name: ";ret_status
        goto 300
end if

! Remove any leading underscore and trailing colons.
node=right$(node,2%) if left$(node,1%)="_"
pos_colon=instr(1%,node,":")
node=left$(node,pos_colon-1%) if pos_colon<>0%
our_node=node
print "our node is ";our_node

mast_flags=lck$m_valblk ! lck$m_system not specified
                        ! Assumes user interface and server run
                        ! under the same UIC.
! print "Enqueueing request for the MAST lock..."
ret_status=sys$enq(     eflag1 by value,        ! efn           &
                        lck$k_exmode by value,  ! lkmode        &
                        mast_lblk() by ref,     ! status_block  &
                        MAST_FLAGS by value,    ! flags         &
                        "MAST",                 ! resource name &
                        ,                       ! parent lockid &
                        mast_ast by value,      ! ast addr      &
                        ,                       ! ast parm      &
                        ,                       ! blocking ast  &
                        ,       )               ! access mode, null

if not ret_status and 1% then &
        print "error from MAST $enq : ";ret_status
        goto 300
end if

! If and when we get the MAST lock, the MAST_AST routine will set
! MASTER to 1%,  otherwise, it will remain 0%.
! The MAST_AST will also obtain the user-interface lock for us.

! In any case, we enqueue a request for the "RR" lock in
! CR-mode, specifying RR_AST as a blocking AST.

! Print "ENQing round-robin lock request..."
rr_flags=lck$m_valblk !or lck$m_system
ret_status=sys$enqw(    eflag2 by value,        ! efn           &
                        lck$k_crmode by value,  ! lkmode        &
                        rr_lblk() by ref,       ! status_block  &
                        RR_FLAGS by value,      ! flags         &
                        "RR",                   ! resource name &
                        ,                       ! parent lockid &
                        ,                       ! ast addr      &
                        ,                       ! ast parm      &
                        rr_ast by value,        ! blocking ast  &
                        ,       )               ! access mode, null

if not ret_status and 1% then   ! Unexpected error getting lock.
  print "Server: error queueing CR-mode RR lock : ";ret_status
     goto 300
end if

100     if master=1% then print "MASTER "; else print "slave ";

200     print "hibernating, waiting for ASTs..."
        ret_status=sys$hiber()
```

```
300     print "An error occured. Server exiting..."
        ret_status=lib$free_ef(eflag1)
        ret_status=lib$free_ef(eflag2)
32767   end
```

MAST_AST Routine

```
1       %TITLE "MAST AST routine"
        %IDENT "V1.00"
        SUB MAST_AST ( LONG OUR_PARAM, R0, R1, PC, PSL)
        OPTION TYPE = EXPLICIT

        external long function sys$enqw,lib$get_ef,lib$free_ef,sys$exit
        external long constant mast_ast,ui_ast ! AST routines

        external long constant                  &
        lck$m_system,   &
        lck$m_valblk,   &
        lck$k_exmode

        declare long ret_status, ! For system service status returns &
                ui_flags,        ! flags modifiers for locks &
                eflag3           ! event flag to use with system services

        ! These two are in common so AST routines can get at them.
        common (scommon) long master, string our_node=6%

        ! Maps for MAST and UI lock blocks.
        map (mast_lock$block) long mast_lblk(6%)
        map (ui_lock$block) long ui_lblk(6%)

20      print "We got the MAST AST. We are now functioning as MASTER !"
        master=1%

        ui_flags= lck$m_valblk ! Did not specify lck$m_system.
                               ! Assumes server and user-interface have
                               ! same UIC.

        print "MAST AST: Requesting EX-mode lock on User_interface"

        ! Get an event flags to use for system service call.
        ret_status=lib$get_ef(eflag3)

        ret_status=sys$enqw(    eflag3 by value,        ! efn           &
                                lck$k_exmode by value,  ! lkmode        &
                                ui_lblk() by ref,       ! status_block  &
                                UI_FLAGS by value,      ! flags         &
                                "UI",                   ! resource name &
                                ,                       ! parent lockid &
                                ,                       ! ast addr      &
                                ,                       ! ast parm      &
                                ui_ast by value,        ! blocking ast  &
                                ,       )               ! access mode, null

        if not ret_status and 1% then   ! Unexpected error getting lock.
                print "MAST AST: error queueing UI lock: ";ret_status
                ! Kill this server process.
                ret_status=sys$exit(ret_status by value)
        else
                print "MAST AST: Got User Interface lock"
                ret_status=lib$free_ef(eflag3) ! Done using event flag.
        end if

32767   END SUB
```

USER_IFACE.BAS

```
1       PROGRAM USER_IFACE
        %TITLE "User Interface"
        %IDENT "V1.00"
        !Prototype Server-wide user interface
        !
        !Program prompts for a node and a message of up to 10
        !characters. It $ENQWs an EX-mode lock on resource "UI" which is
        !held by the master server. When it gets the lock, it puts the
        !message in the value block, causing the server (which tries to
        !keep ownership of that resource in EX-mode, but which has a
        !blocking AST that gives the lock up momentarily if someone else
        !requests it) to get the UI_AST, which reads the message.
        !If no server is running, the message simply vanishes into the
        !ether.

10      option type=explicit

        external long function sys$enqw,sys$deq

        external long constant          &
        lck$m_system,                   &
        lck$m_valblk,                   &
        lck$m_convert,                  &
        lck$k_nlmode,                   &
        lck$k_exmode

        declare long ret_status,        ! system service status &
                flags
        declare string msg_string

        map (lock$block) word cond,word fill,long lock_id, string msg=16%
        map (lock$block) long lblk(6%)

        on error goto 10000
        flags=lck$m_valblk ! No lck$m_system, assumes server has same UIC.

        ! Prompt for message to send by means of the lock value block
15      input "node:msg ";msg_string

20      ! print "Requesting UI lock..."
        ret_status=sys$enqw     ( ,                     ! efn           &
                                lck$k_exmode by value,  ! lkmode        &
                                lblk() by ref,          ! status_block  &
                                FLAGS by value,         ! flags         &
                                "UI",                   ! resource name &
                                ,                       ! parent id     &
                                ,                       ! ast addr      &
                                ,                       ! ast parm      &
                                ,                       ! blocking ast  &
                                ,       )               ! access mode,null
        if not ret_status and 1% then
            print "error from ENQW : ";ret_status;"   good-bye."
              goto 32767
        end if

        ! print "Got UI lock."
        ! print "Status    :";cond
        ! print "Lock-id   :";lock_id
        ! print "Old-Msg   :'";msg;"'" if edit$(msg,132%)<>""

        ! If lock value block has a message, it was released by another
        ! user-interface process rather than by a server. In that case,
        ! try again.
```

```
        If edit$(msg,6%)<>"" then ! characters other than spaces, nulls, etc. &
                print "Lock block already has message '"+msg+"'"
                print "Letting lock go and trying again..."
                ret_status=sys$deq(lock_id by value ,   &
                                   msg by ref,,)
                if not ret_status and 1% then
                 print "Error from $DEQ : ";ret_status;" good-bye."
                      goto 32767
                end if
                sleep 1%        ! Wait 1 second to allow a server to get it.
                goto 20 ! Try again.
        else
                lset msg=left$(msg_string,16%)  ! Leftmost 16 chars
                ! print "Putting '";msg;"' in block and dequeueing..."
                ret_status=sys$deq(lock_id by value ,   &
                                   msg by ref,,)
                if not ret_status and 1% then &
                        print " * Error from $DEQ : ";ret_status;" good-bye."
                        goto 32767
                end if
        end if

        goto 15 ! Loop prompting for messages from the user forever.

10000   resume 32767 if err=11% ! (end of file, ctrl-z)
        print "error ";err;" at line ";erl;" ";ert$(err)
        resume 32767    ! Exit if any BASIC error.

32767   end
```

UI_AST Routine

```
1       %TITLE "UI AST routine"
        %IDENT "V1.00"
        SUB UI_AST ( LONG OUR_PARAM, R0, R1, PC, PSL)
        ! This Blocking AST gets invoked on the server which is functioning
        ! as MASTER when a user interface tries to gain access to the
        ! user-interface lock in exclusive mode. (This server holds
        ! the UI-lock in EX-mode.)
        ! It down-converts the EX-mode UI lock to NL-mode and then
        ! ENQWs a request to re-convert the lock to EX-mode.
        ! When that second request is satisfied, it obtains the value in
        ! the UI lock's value block.
        !
        ! In this particular application, we expect that data to be of the
        ! form NODE:MSG. If NODE matches our own node, we simply print the
        ! msg on the terminal.  In a real implementation (such as we envision)
        ! it COULD do something more useful, for example a QIO to write the
        ! message to a mailbox which is already open on the channel passed
        ! in the AST parameter. This would allow users anywhere on the
        ! cluster to send messages to that mailbox, as long as messages do
        ! not exceed the maximum length.
        !
        ! If the node does NOT match ours, we (the master) have to send the
        ! message to all other servers.  We $ENQW an upgrade to EX-mode
        ! of the "RR" lock that we are holding (in CR mode). This is blocked
        ! by the CR locks that all the other servers are holding, and thus
        ! causes them all to get the RR AST (which lowers their lock to NL
        ! and $ENQW's an upgrade to CR specifying the RR blocking AST again
        ! When the $ENQW completes, the other servers read the value block.)
        !
        ! The value block for this lock contains the destination
        ! node, the message, and a flag byte that the destination node can
        ! set to indicate receipt of message.

        OPTION TYPE = EXPLICIT
        external long function sys$enqw,lib$get_ef,lib$free_ef,sys$exit
        external long constant ui_ast,rr_ast
```

```
        external long constant          &
        lck$m_system,   &
        lck$m_valblk,   &
        lck$m_convert,  &
        lck$k_nlmode,   &
        lck$k_crmode,   &
        lck$k_exmode

        declare long ret_status,        ! system service status &
                ui_flags,pos_colon,rr_flags, eflag4

        declare string msg_string,node_part,msg_part

        map (ui_lock$block) word ui_cond,word ui_fill,long ui_lock_id, &
                string ui_msg=16%
        map (ui_lock$block) long ui_lblk(6%)

        map (rr_lock$block) word rr_cond,word rr_fill,long rr_lock_id, &
                string rr_msg=16%
        map (rr_lock$block) long rr_lblk(6%)
        map (rr_lock$block) string rr_misc=8%, &
                string rr_node=6%,string rr_msg_part=10%

        common(scommon) long master, string our_node=6%

        ret_status=lib$get_ef(eflag4)   ! Get an event flag to use.

20      ! print "UI AST: Down-grading lock on UI to NL-mode..."
        ! Clear out the value in the lock value block.
        ui_msg=""

        ui_flags=lck$m_valblk OR lck$m_convert !OR lck$m_system
        ret_status=sys$enqw     ( eflag4 by value,      ! efn           &
                                  lck$k_nlmode by value,  ! lkmode        &
                                  ui_lblk() by ref,       ! status_block  &
                                  UI_FLAGS by value,      &
                                  "UI",                   ! resource name &
                                  ,                       ! parent id     &
                                  ,                       ! ast addr      &
                                  ,                       ! ast parm      &
                                  ,                       ! blocking ast  &
                                  ,       )               ! access mode,null
        if not ret_status and 1% then
                print "UI AST: UI downgrade error from ENQW : ";ret_status
                goto 32767
        end if
        ! print "UI AST: Lock converted to NL-mode."

        ! print "UI AST: Requesting ($ENQW) UI lock upgrade to EX-mode"
        ret_status=sys$enqw     ( eflag4 by value,      ! efn           &
                                  lck$k_exmode by value,  ! lkmode        &
                                  ui_lblk() by ref,       ! status_block  &
                                  UI_FLAGS by value,      &
                                  "UI",                   ! resource name &
                                  ,                       ! parent id     &
                                  ,                       ! ast addr      &
                                  ,                       ! ast parm      &
                                  ui_ast by value,        ! blocking ast  &
                                  ,       )               ! access mode,null
        if not ret_status and 1% then
                print "UI AST : UI upgrade error from ENQW : ";ret_status;
                goto 32767
        end if

        !print "UI AST: UI Lock upgraded to EX-mode."
        !print "Status    :";ui_cond
        ! print "Lock-id   :";ui_lock_id
        print 'UI AST: ';
        print "(MASTER)"; if master=1%
        print 'WE GOT "';trm$(ui_msg);'"' ! print user-interface msg
        goto 32767 if edit$(ui_msg,6%)=""
        pos_colon=instr(1%,ui_msg,":")
        node_part=EDIT$(left$(ui_msg,pos_colon-1%),32%) ! upcase
        msg_part=right$(ui_msg,pos_colon+1%)
```

```
! Note: In any real application, we should also check the status
! in the lock block and make sure it is not SS$_VALNOTVALID. That
! might occur if there was a cluster transition while a requestor
! was holding the lock in EX-mode. We'd still get the lock, but
! couldn't trust the value in the block. (It would probably be
! a duplicate msg.)

if node_part=trm$(our_node) then &
        print "UI_AST: msg '";trm$(msg_part);"' is for us ("; &
          node_part;")"
        ! Master's own message processing goes here.
        goto 32767
end if

goto 32767 if msg_part=""       ! Don't broadcast a null msg.

! Message is not for us.
! Request upgrade of RR lock to EX-mode.
! Wait for slaves to release.

!print "UI_AST: Waiting for RR lock in EX-mode..."
rr_flags=lck$m_convert !OR lck$m_system
ret_status=sys$enqw     ( eflag4 by value,      ! efn           &
                        lck$k_exmode by value,  ! lkmode        &
                        rr_lblk() by ref,       ! status_block  &
                        RR_FLAGS by value,      &
                        "RR",                   ! resource name &
                        ,                       ! parent id     &
                        ,                       ! ast addr      &
                        ,                       ! ast parm      &
                        ,                       ! blocking ast  &
                        ,       )               ! access mode,null
if not ret_status and 1% then
        print "UI AST: RR upgrade error ENQW : ";ret_status;
        goto 32767
end if
!print "UI AST: Master's CR-mode RR Lock up-graded to EX-mode."

! Put Node and msg-part into RR lock value block.
lset rr_node=node_part
lset rr_msg_part=msg_part

! Downgrade round-robin lock to CR mode so slaves can get it.
!print "UI_AST: waiting for downgrade to CR of RR lock..."
rr_flags=lck$m_convert OR lck$m_valblk !or lck$m_system
ret_status=sys$enqw( eflag4 by value,! efn      &
                lck$k_crmode by value,  ! lkmode        &
                rr_lblk() by ref,       ! status_block  &
                RR_FLAGS by value,      ! flags         &
                "RR",                   ! resource name &
                ,                       ! parent lockid &
                ,                       ! ast addr      &
                ,                       ! ast parm      &
                ,                       ! blocking ast  &
                ,       )               ! access mode, null

if not ret_status and 1% then   ! unexpected error getting lock
        print "UI_AST: RR $enqw DOWNGRADE error: ";ret_status
        print "killing this server"
        ret_status=sys$exit(ret_status by value)
end if

!print "UI_AST: RR lock downgraded to CR-mode"

32767   ret_status=lib$free_ef(eflag4) ! Done using event flag.
        END SUB
```

RR_AST Routine

```
1       SUB RR_AST ( long OUR_PARAM, R0, R1, PC, PSL)
        ! This blocking AST gets delivered when a slave is blocking the
        ! master's request for the RR lock in EX mode.
        ! If the server is MASTER, it merely logs the AST (can happen
        ! once after mastership switches).

        ! If the server getting this AST is a slave, it downgrades the
        ! lock it holds on RR to NL-mode and waits for the lock again in
        ! CR mode. When it gets the lock, it reads the value block to see
        ! if the message in the block is for it.

        ! There is no back-communications path from slaves back to
        ! the master.
        !
        OPTION TYPE = EXPLICIT
        external long function sys$enqw,lib$get_ef,lib$free_ef,sys$exit,&
                lib$wait
        external long constant rr_ast

        declare long constant           &
        !lck$m_system=X'00000010'L,     &
        lck$m_valblk=X'00000001'L,      &
        lck$m_convert=X'00000002'L,     &
        lck$k_nlmode=X'00000000'L,      &
        lck$k_crmode=X'00000001'L,      &
        lck$k_exmode=X'0000005'L

        declare long foo_status,        ! system service status &
                rr_flags,eflag5

        map (rr_lock$block) word rr_cond,word rr_fill,long rr_lock_id, &
                string rr_msg=16%
        map (rr_lock$block) long rr_lblk(6%)
        map (rr_lock$block) string rr_misc=8%, &
                string rr_node=6%,string rr_msg_part=10%

        common (scommon) long master, string our_node=6%

        foo_status=lib$get_ef(eflag5) ! Get an event flag for system services.

20      !print "RR AST: Blocking mode AST delivered "
         print "RR_AST: Status    :";rr_cond if rr_cond<>1%
        !print "Lock-id   :";rr_lock_id
        !print 'RR_AST ';
        !print "(MASTER) "; if master=1%
        !print "(slave) "; if master<>1%

        if master=1% then       &
                print "RR_AST: ? We are MASTER - exiting AST"
                goto 32767
        end if

        ! We are slave. Convert RR lock to NL mode so master can get it
        ! in EX mode.

        !print "RR_AST: downgrading CR-mode RR lock to NL-mode"
        rr_flags= lck$m_convert or lck$m_valblk
        foo_status=sys$enqw(eflag5 by value,    ! efn           &
                        lck$k_nlmode by value,  ! lkmode        &
                        rr_lblk() by ref,       ! status_block  &
                        RR_FLAGS by value,      ! flags         &
                        "RR",                   ! resource name &
                        ,                       ! parent lockid &
                        ,                       ! ast addr      &
                        ,                       ! ast parm      &
                        ,                       ! blocking ast  &
                        ,       )               ! access mode, null

        if not foo_status and 1% then   ! Unexpected error getting lock.
                print "RR_AST: fatal error downgrading RR lock: ";foo_status
                foo_status=sys$exit(foo_status by value) ! Kill ourself.
        end if
```

```
! All slaves have to be in NL: mode at once for the master
! to be able to send the message.
! We could delay calling $getlki to make sure that there is no process
! waiting for the RR-lock in EX-mode, but instead we take the
! lazy way out. A 1 second delay should be enough time for all the
! slave servers to get the AST and downgrade their RR-lock to NL:

foo_status=lib$wait(1.0)  ! Delay 1 second.

! Request upgrade to CR mode. Read value block when request completes.
! Don't specify a blocking ast.
! print "RR_AST: Slave upgrading NL-mode RR lock to CR, no
! blocking ast"
!
foo_status=sys$enqw(eflag5 by value,     ! efn           &
                lck$k_crmode by value,   ! lkmode        &
                rr_lblk() by ref,        ! status_block  &
                RR_FLAGS by value,       ! flags         &
                "RR",                    ! resource name &
                ,                        ! parent lockid &
                ,                        ! ast addr      &
                ,                        ! ast parm      &
                ,                        ! blocking ast  &
                ,       )                ! access mode, null


!print "RR_AST: RR lock is now in CR-mode"
if not foo_status and 1% then   ! Unexpected error getting lock.
        print "RR_AST: fatal RR $enqw upgrade error: ";foo_status
        foo_status=sys$exit(foo_status by value) ! Kill ourself.
end if

! Upgrade from CR to CR, this time specify blk AST and no VBLK.
rr_flags= lck$m_convert
foo_status=sys$enqw(eflag5 by value,     ! efn           &
                lck$k_crmode by value,   ! lkmode        &
                rr_lblk() by ref,        ! status_block  &
                RR_FLAGS by value,       ! flags         &
                "RR",                    ! resource name &
                ,                        ! parent lockid &
                ,                        ! ast addr      &
                ,                        ! ast parm      &
                rr_ast by value,         ! blocking ast  &
                ,       )                ! access mode, null

If not foo_status and 1% then   &
        print "RR AST: CR CR upgrade err ";foo_status
end if

! Parse value block from when we got CR lock earlier.
! Format of our value block is:  rr_node   (6 bytes, fixed length)
!                                rr_msg_part (10 bytes)
!print "destination_node: ";trm$(rr_node); &
!  "' msg: '";trm$(rr_msg_part);"'"

if trm$(rr_node)=trm$(our_node) then &
        print "   Msg '";trm$(rr_msg_part); &
        "' is for us!"
else
        print "   Msg '";trm$(rr_msg_part); &
        "' is for node ";trm$(rr_node); ", not for us"
end if

32767   foo_status=lib$free_ef(eflag5) ! Done using event flag.
        END SUB
```

Glossary

For additional definitions of commonly used terms in documentation of the VMS operating system, see the *VMS Glossary*.

after-image journaling: A feature of VAX RMS journaling that allows you to reconstruct a data file up to the last transaction that was successfully completed.

application: A set of procedures that performs a task or function.

AST: See *asynchronous system trap*.

asynchronous event: An asynchronous event does not necessarily complete the requested operation before allowing the requesting program to continue execution.

asynchronous system trap (AST): A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously, with respect to the user process, of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when the event occurs. When the AST routine exits, the system resumes execution of the process at the point where it was interrupted.

See also *blocking AST* and *completion AST*.

automatic record locking: VMS RMS capability for different locking options for record access.

automatic restart: The VMS batch facility has an automatic restart capability when a batch or print job is submitted with a /RESTART parameter.

availability: The proportion of time that service is available from a VAXcluster system to perform user applications.

back-end: The data and computational processes in a transaction processing system in which terminal and menu functions are handled by separate processes.

barrier synchronization: A synchronization method that establishes some barrier or point that all concurrent tasks must reach before continuing their work.

before-image journaling: A feature of VAX RMS journaling that allows you to undo a series of modifications to a data file to return the file to a previous known state.

blocking AST: An AST requested using the lock management system services and is delivered to the process holding a lock on a resource when the lock mode is preventing another process from accessing that resource.

boot node: A VAX CPU in a Local Area VAXcluster system or Mixed-Interconnect VAXcluster system responsible for booting and providing system disk service to one or more satellites.

bottleneck: A condition of degraded performance caused by overcommitting a resource, such as: the VAXcluster I/O subsystem, memory of a CPU, or a VAXcluster CPU.

channel: An HSC interface between disk drives or tape formatters and its buffer memory.

checkpointing: A method of using the RESTART_VALUE command in a DCL command procedure to restart the appropriate segment of the DCL command procedure that was executing when a batch execution queue fails.

CI-based VAXcluster system: A VAXcluster system in which all nodes are connected to a CI bus.

client: Any process requesting service from a server.

clustering: The formation of a VAXcluster system from independent VAX processors.

cluster membership: At the formation of a VAXcluster system, cluster membership is composed of all the participating VAXcluster CPUs voting to achieve quorum. Anytime there is a cluster membership change, the current value for votes is compared to quorum.

cluster state transition: The change in cluster membership when a node joins or leaves the cluster.

clusterwide lock database: The clusterwide database for all granted locks maintained by the distributed VMS lock manager. The distributed VMS lock manager arbitrates the access of a resource name by determining the compatibility of the requested locks with those existing for that resource name on the clusterwide lock database.

common-environment: A VAXcluster operating environment that provides the same resources, devices, logical names, software, and access to every user on every node.

common system disk: A VMS disk that supports the booting of two or more processors.

completion AST: An AST requested using the lock management system services and is delivered to the process that has requested a lock when that process is granted access to the resource.

computer interconnect (CI): A high-speed bus, with dual data paths that connect all nodes in a CI-based VAXcluster system. The bus bandwidth is 70 megabytes per second per path.

concurrent access: The simultaneous use of a file or database by more than one user.

configuration: The arrangement of interconnected nodes and their peripheral devices in a VAXcluster system.

connection manager: A VMS software component that determines and maintains cluster membership, synchronizes cluster transitions, and prevents partitioning.

Conversion queue: Clusterwide queue maintained by the distributed VMS lock manager for lock conversion requests of locking requests that have already been granted for one mode.

cooperating processes: Processes that are following site-specific conventions when using the lock management system services.

CPU-bound: Pertaining to slow system response caused by the number of computations in a CPU-intensive application.

CPU-intensive: An application that uses a large number of CPU compute cycles.

data dependence: A situation in which information produced by one part of a program is necessary before another part can produce accurate results.

data disk: A disk that is strictly used for application (non-system) data. No system roots exist on it; no VAX CPU boots from it.

data sharing: A programming technique that allows multiple users or processes to read and modify a common file or database.

deadlock: A state in which a process is waiting for a particular lock that can never be granted. Deadlock can occur whenever processes compete for resources or whenever processes wait for each other to complete certain actions.

DECnet node name: A unique name in the DECnet network associated with each node in the VAXcluster system.

DECnet task-to-task communication: The DECnet service that allows one process to communicate with another.

See also *task-to-task communication*.

DECnet-VAX: The VMS-specific use of DECnet software that enables a VAX CPU to act as a network node.

decomposition: The action of modifying a single-stream program into a parallel program by creating parallel sections that can be executed concurrently.

disk server: A VAX CPU that provides disk service to other VAXcluster nodes.

distributed file system: The VMS software component that allows all VAX processes to share mass storage disks connected to a VAX CPU or an HSC. The disks function as though they were local to each VAX CPU. The distributed file system coordinates access to files using the distributed VMS lock manager.

distributed job controller: The VMS software component that controls the use of clusterwide print and batch queues.

distributed VMS lock manager: The VMS software component that synchronizes access to shared resources on a clusterwide basis. The distributed VMS lock manager also detects and resolves deadlocks.

dual-ported: A dual-ported disk can be physically connected between two HSC nodes or two local CPU controllers. Thus, a dual-ported disk has multiple-access paths and it can be accessed clusterwide in a coordinated way through either HSC or local CPU controller. When a disk is dual-ported and one of the HSCs or local CPU controllers to which it is connected fails, the remaining HSC or CPU controller automatically provides access to the disk.

Ethernet: The coaxial communications cable that connects nodes and information processing products and allows them to exchange data. The Ethernet connects: nodes within a cluster, and nodes in a cluster with nodes in a Local Area Network (LAN).

exception condition: An event, detected by hardware or software, that causes a change in the flow of instruction execution.

exclusion: A synchronization method that allows only one process at a time to access some critical resource.

explicit locking: The use of calls to lock management system services from a high-level programming language.

fail back: After a hardware failover has initiated a failover, an application may fail back when the hardware failure is corrected.

failover: The automatic or manual action of switching to an alternate path or component after the failure of a path or component. For example, if the access path to a dual-ported disk fails, the alternate path is automatically available to all nodes accessing the disk. Automatic failover is transparent to the user; manual failover must be initiated by operator intervention.

fault tolerance: The ability of the system to guard against failures that could lower productivity or cause a corruption or loss of data.

file tuning: The process of designing your files to achieve better processing performance.

front-end: The processes controlling terminal and menu functions in a transaction processing system in which data manipulation and computation are handled by separate processes.

global section: I/O buffer into which a block of disk data is read. Two or more processes on the same node can read from or write to this data block. In a VAXcluster system, processes on one node cannot access the global buffers of another node. Each node must read a data block from disk into its own buffer. If a node writes to its buffer, it must store the buffer back on disk with a lock value block indicating that the data is changed.

Granted queue: Clusterwide queue maintained by the distributed VMS lock manager for granted lock requests.

hash file: A high-performance file organization which uses an algorithm to manage a table of keys to access records.

hibernation: A state in which a process is inactive, but known to the system with all of its current status. A hibernating process becomes active again when a wake request is issued. It can schedule a wake request before hibernating, or another process can issue its wake request. A hibernating process can also become active long enough to service any AST it may receive while it is hibernating.

Hierarchical Storage Controller (HSC): An intelligent mass storage subsystem, a non-CPU node on the CI, that provides shared access to Digital Storage Architecture (DSA) disks and tapes.

high availability: An advantage found in a VAXcluster system where, if one of the CPUs in the VAXcluster system fails, the other VAXcluster CPUs are minimally affected.

image: Procedures and data that have been bound together by the linker. There are three types of VMS images: executable, shareable, and system.

implicit locking: The use of a high-level language, interfacing with VMS RMS, for record locking operations.

interprocess communication: The passing of information between two or more processes.

I/O-bound: The condition in which an I/O-intensive application is waiting for an I/O device before the application can continue to execute.

I/O channel: The logical connection between a user program and a file or device.

I/O-intensive: An application that uses a large number of I/O operations.

journaling: The process of recording information about operations on a database or file onto a recoverable resource. The type of information recorded depends on the type of journal being created.

See also *after-image journaling*, *before-image journaling*, and *recovery-unit journaling*.

layered product: A Digital software product, such as VAX ACMS or DECintact, that is layered on the VMS operating system.

Local Area VAXcluster system (LAVc): A VAXcluster system in which all nodes are connected using the Ethernet. Disks in a LAVc system are connected to adapters on LAVc CPUs.

local CPU: The VAXcluster CPU where a process is logged in.

local node: A network or cluster node at which a terminal is logged in or a user is physically located.

local system disk: A disk used as a system disk by a VAX CPU connected directly to it by a local adapter. If dual-ported, a local disk cannot be a local system disk.

lock, locking: An association between a process and a resource name maintained by the distributed VMS lock manager. A lock is normally used to synchronize access by multiple processes to shared objects.

lock manager: See *distributed VMS lock manager*.

lock mode: A value associated with a request to the distributed lock manager indicating the type of lock requested and its compatibility with other locks. For example, multiple users can request CONCURRENT READ locks on the same resource at the same time.

lock status block: A block that contains lock information about the status of a process, such as the lock mode, and the address for the resource.

lock value block: An optional data block that is created when a resource lock is requested. It is also used to communicate information among processes sharing a resource.

logical link: A communication path between processes running on two different nodes. Contrast with physical link. A logical link carries a stream of traffic between two user-level processes. Each logical link is a temporary data path that exists until one of the two processes terminates the connection.

mailbox: A software data structure that is used for interprocess communication. One process writes data to the mailbox; another process reads the data from the mailbox.

manual record locking: By explicitly calling $QIO and lock management system services, the user can control the locking granularity to synchronize file or record access.

Mass Storage Control Protocol (MSCP): A standardized protocol for communication between hosts and mass storage controllers. VMS uses MSCP to pass disk I/O requests to disk controllers.

master process: A process that controls and monitors the activity of one or more slave processes.

Mixed-Interconnect VAXcluster system (MIVc): A VAXcluster system in which some nodes are connected with the Ethernet and other nodes are connected with both the Ethernet and the CI.

modularity: Modularity is a concept used when designing a cluster configuration to ensure that there is an independence of multiple components in the VAXcluster system so that the failure of one component has minimal effect on the system.

modular programming: A method of breaking down a program consisting of many operations or large processing requirements into separate modules to enable multiple CPUs to share the work load and to provide software maintainability.

MSCP server: The software that allows disks to be available to other nodes in the cluster by software emulation of an HSC.

multiple-access paths: A VAXcluster system can provide multiple-access paths to ensure an application's ability to access a disk.

multiple-environment: A VAXcluster operating environment that allows a group of nodes to share one set of resources, while another group shares a different set.

node: A single VAX CPU (or SMP system) or HSC in the VAXcluster system or an individual computer system in a network that can communicate with other computer systems in the network.

parallelism: A method of computing that occurs when a section of an application is divided into multiple tasks, and those tasks are executed simultaneously on multiple CPUs.

parent lock: A lock held on a resource at a coarse granularity, such as an entire file, that can be divided into component parts, so that sublocks can be made at a finer granularity.

partitioning: The division of a cluster into two or more groups. Each group is unaware that the other exists, and each is trying to access the same resource in an uncoordinated manner. This can cause corruption of disk file structures because a cluster only coordinates access to resources among cluster members.

performance: The ability of an application to run efficiently without heavy demand on system resources.

process: The basic entity scheduled by the VMS operating system that provides the context in which an image executes. A process consists of an address space and both a hardware and software context.

process control: The mechanism that allows one process to control another process by starting it, guiding its operations, or terminating it.

process information: The mechanism that allows one process to obtain process information about processes on a local or remote nodes.

process synchronization: The method of preventing two or more processes from interfering with each other when reading or modifying shared data or a set of constraints that affects or controls the ordering of events in a decomposed application.

protocol: The conventions or rules for the format and timing of messages sent and received.

quorum: An algorithm that ensures that enough nodes are present to form a cluster. Maintaining quorum avoids cluster partitioning.

quorum disk: A disk that acts as a virtual node in a cluster. For more information on a quorum disk, see the *VMS VAXcluster Manual*.

Record Management Services (VMS RMS): A set of operating system procedures that is called by programs to process files and records within files. RMS allows processes to share data at the record level. VAX RMS is an integral part of the VMS operating system; its procedures run in executive mode.

recovery: The process of restoring data to a known condition after a system or program failure.

recovery-unit journaling: A feature of VAX RMS journaling that allows you to perform a transaction rollback for all file or database recovery units that have not successfully completed.

redundancy: The characteristic of having two or more identical features to increase dependability and availability.

remote CPU: Any VAXcluster CPU other than the one at which the user is located.

remote node: Any node other than the one at which the user is located.

remote process creation: The method of a process on a local CPU creating a process on a remote CPU in the VAXcluster environment or a remote node in the network.

replication: The action of running multiple copies of a single-stream program on multiple CPUs in a VAXcluster system.

resource: A physical unit, such as a file, device, or memory, required by a process to complete its activity.

resource granularity: The level of resource locking being performed by a process. The granularity can be coarse (locking an entire database) or fine (locking records within a database). Granularity provides complex interlocking mechanisms and greater control over interactions among processes.

resource manager: The VAXcluster CPU that controls the granting of lock requests on a given resource tree for which it maintains information about all granted and waiting lock requests.

resource name: The name that a process uses to identify a resource when requesting a lock, for a specified lock mode, on that resource.

resource tree: The hierarchical (tree-like) structure of resource names used by the VMS lock manager.

response time: The time it takes a system to answer or react to a query from a terminal.

rotational delay time: The time that it takes for the desired sectors to rotate under the heads.

satellite node: A workstation or similar MicroVAX CPU, running VMS, connected to the Ethernet and booted by a boot server.

seek time: The time it takes for a disk to move its heads to the proper cylinder.

server process: A process that performs specific functions or activities for one or more clients.

shadow set: Two or more disks containing identical copies of data. The set is treated as one disk by the CPU that defined the set. When the CPU writes data, the data is written to both disks in the set. When the CPU reads data, the data is read from either disk, depending on the position of the head.

See also *volume shadowing*.

sharable lock: A resource lock that allows multiple processes to lock the same resource at the same time.

shared disk: A disk that is mounted on a cluster-accessible device by one or more nodes in the cluster.

slave process: Any process that is being controlled or monitored by a master process.

Star Coupler: The common connection point for all processor and HSC nodes connected to the CI bus.

state transition: See *cluster state transition.*

sublock: A lock held on a resource of a finer granularity, such as records or data items, than that of its parent lock.

See also *parent lock* and *resource granularity.*

synchronous events: A synchronous event executes a requested operation simultaneously as the requesting program continues execution.

system disk: A disk on which the VMS operating system is located.

task: A component of work that is defined and scheduled within an application.

task-to-task communication: The method used to communicate among processes in the DECnet networking environment.

terminal server: A communications device that connects terminals, modems, or printers to an Ethernet network.

throughput: The amount of work completed per unit of time. In a cluster environment, it is possible to get more work done in a certain amount of time than is possible on a single CPU.

time sharing: A method of allocating computer time in which each process gets use of the CPU in turn.

transaction processing: A technique for organizing multi-user, high volume, on-line applications that provides control over user access and updates of files or databases.

VAXcluster system: An integrated organization of VAX systems that use VMS software and communicate over a high-speed communications path (the CI bus or Ethernet). A VAXcluster system has all the capabilities of a single-node VAX system, plus the ability to share CPU resources, queues, and disk and tape storage. Like a single VAX system, the VAXcluster system provides a single security and management environment. Member nodes can either share the same operating system environment or serve specialized needs.

See also *CI-based, Local Area,* and *Mixed-Interconnect VAXcluster systems.*

victim process: In a deadlock situation, the distributed VMS lock manager eliminates contention for a resource by returning an error status to a chosen process (a "victim") when that process makes a new lock or lock conversion request.

volume shadowing: A procedure by which two or more identical copies of data are written to multiple disks defined as a set. This provides updated back-up copies of current data at all times.

See also *shadow set*.

Waiting queue: Clusterwide queue maintained by the distributed VMS lock manager for lock requests that are waiting to be granted.

workload balancing: A technique used to evenly distribute multiple users and resources between VAX CPUs for maximum productivity.

Index

A

Application design • 4–1 to 4–24
  See Client-Server Model
  See File Sharing Model
  See Parallelism Model
  comparison of models • 5–6
  using layered products based on VMS • 5–12 to 5–22
  using products associated with VMS • 5–7 to 5–12
Applications
  availability • 1–3, 2–10
  demonstration application • 8–1 to 8–19
  designing for faster completion • 5–3 to 5–4
  designing for increased availability • 5–1 to 5–3
  designing for maximum throughput • 5–4 to 5–5
  performance • 2–10, 3–7
  software levels • 3–15
  suitable for a cluster • 2–9 to 2–11
  unsuitable for a cluster • 2–12 to 2–13
ASTs
  use with lock management system services • 3–13, 6–58 to 6–60
Availability
  applications • 1–3, 2–10
  disks • 1–5 to 1–7
  hardware • 1–2
  resources • 3–6

B

Batch queues • 1–14
Blocking ASTs
  code example • 6–62 to 6–64
  use with lock management system services • 3–13, 6–58 to 6–60
Bottlenecks • 7–2
  See CPU bottlenecks
  See I/O bottlenecks
  See memory bottlenecks
  monitoring • 7–2 to 7–3, 7–16 to 7–21

C

CI adapter
  I/O bottleneck • 7–4
  I/O capacity • 2–14
CI bus • 1–3
  bottlenecks • 7–6
Client-Server Model • 4–6 to 4–17
  advantages of using many-to-one client-server • 4–14
  advantages of using one-to-one client-server • 4–8
  disadvantages of using many-to-one client-server • 4–14 to 4–15
  disadvantages of using one-to-one client-server • 4–8
  example of using many-to-one client-server • 4–16 to 4–17
  example of using one-to-one client-server • 4–10
  illustration of many-to-one client-server • 4–11
  illustration of one-to-one client-server • 4–7
  implementation requirements of many-to-one client-server • 4–15 to 4–16
  implementation requirements of one-to-one client-server • 4–8 to 4–9
  many-to-one client-server • 4–11 to 4–17
  one-to-one client-server • 4–7 to 4–10
  using for increased availability • 5–2 to 5–3
  using to maximize throughput • 5–4 to 5–5
Clusterwide lock database • 3–2 to 3–3
Common-environment configuration • 1–4
Common system disk
  characteristics • 1–7
  disadvantages • 1–7
Communications
  internode, example • 8–4
Computer Interconnect
  See CI bus
  See DSSI bus
  See Ethernet
Compute server • 4–11
  See also Client-Server Model
Configurations
  See common-environment configuration
  See multiple-environment configuration
  determining for disks • 7–3
Connection manager • 1–12 to 1–13

CONVERSION queue • 3–5
CPU bottlenecks • 7–14 to 7–16
  possible solutions • 7–15 to 7–16
CPU-intensive application • 2–7 to 2–9
  compared to I/O-intensive application • 2–3
  illustration of comparison • 2–5
CPUs
  failover access • 1–9

D

$DEQ system service
  See lock management system services
Data sharing • 6–16 to 6–40
  using DECnet-VAX communications • 6–17 to 6–40
  using Read-Only global sections • 6–33 to 6–40
  using VMS Record Management Services • 6–29 to 6–32
Deadlocks • 3–10 to 3–12
  detecting • 3–10 to 3–12
  example • 3–10 to 3–12
  preventing • 3–10 to 3–12
DECintact • 5–13 to 5–15
  features for application design • 5–13 to 5–14
  using in a VAXcluster system • 5–15
DEC-net VAX
  See also nontransparent DECnet-VAX communications
  See also transparent DECnet-VAX communications
  functions • 3–25 to 3–27
  illustration • 3–25
  remote file and record access • 3–27
  task-to-task operations • 3–25 to 3–27
  transparent versus nontransparent • 3–32
Designing applications
  See applications
Disks
  See also common system disk
  See also individual system disk
  availability • 1–2, 1–5 to 1–7
  connection to CPUs • 1–6
  connection to HSCs • 1–5, 1–6
  determining configuration of • 7–3
  displaying information of MSCP-served • 7–3
  DSSI-connected • 1–6
  dual-ported • 1–6
  failover • 1–2
Disks (Cont.)
  I/O rate of type • 2–14
  multiple-access paths • 1–2, 1–3
  system • 1–7
  VAX Volume Shadowing • 1–6
Distributed applications
  conceptual example • 2–17
  considerations for decomposition • 2–16
  considerations for designing • 2–16, 2–19
  considerations for replication • 2–13
  demonstration application designed as • 8–3 to 8–9
  goals • 2–18
Distributed file system
  capabilities • 1–14
Distributed job controller
  See VMS batch facility
Distributed lock manager
  See VMS lock manager
Distributing
  workloads • 1–8
DSSI bus
  connected to ISE • 1–6

E

$ENQ system service
  See lock management system services
Ethernet
  connected to a terminal server • 1–9
  I/O bottleneck • 7–6
Ethernet adapter
  I/O bottleneck • 7–4
  I/O capacity • 2–14
Exception conditions • 6–74 to 6–82
  using DECnet-VAX communications • 6–75 to 6–81
  using lock management system services • 6–81 to 6–82

F

Failover
  See also exception conditions
  access to CPUs • 1–9, 6–81
  of HSC subsystem • 1–6
  of terminal server • 1–9
  process • 8–4

File server • 4–11 to 4–14
See also Client-Server Model
File Sharing Model • 4–2 to 4–6
advantages of using • 4–3
disadvantages of using • 4–3
example of using • 4–4 to 4–6
illustration • 4–2
implementation requirements • 4–4
using for increased availability • 5–1 to 5–2
using to maximize throughput • 5–4 to 5–5

G

$GETJPI system service
See process information system services
$GETLKI system service
See lock management system services
GRANTED queue • 3–5
Granularity • 3–8
coarse • 3–8
example • 3–9
fine • 3–8
illustration • 3–8

H

Hardware redundancy • 1–2
Computer Interconnect (CI) bus • 1–3
disks • 1–2, 1–3, 1–6
HSCs • 1–2
Star Coupler • 1–3
terminals • 1–3
VAX Volume Shadowing • 1–3
Hierarchical Storage Controller
See HSC
High-level programming language
use of lock management system services • 3–3
use of transparent DECnet-VAX communications • 3–28
use of VMS Record Management Services • 3–14
Highwater marking • 7–11
HSC
automatic recovery • 1–6
disks • 1–6
displaying characteristics • 7–3
dual-ported disks • 1–6
I/O bottlenecks • 7–6 to 7–7
HSC SETSHO command • 7–3

I

I/O bottlenecks • 7–4 to 7–12
See also HSC
controller • 7–6 to 7–7
disk drive • 7–7 to 7–9
I/O adapter • 7–4
illustration of I/O pathway • 7–5
medium between I/O adapter and controller • 7–6
performance considerations • 7–9 to 7–10
possible solutions • 7–10 to 7–12
QIO processing • 7–4
I/O-intensive application
advantages • 2–6
compared to CPU-intensive application • 2–3
illustration of comparison • 2–5
Individual system disk • 1–7
Internode communication, example • 8–4

L

LAT
terminals • 1–3
terminal server • 1–3
Local Area Transport
See LAT
Local Area VAXcluster
application example • 8–1
dual-host configuration • 1–6
Lock management system services
functions • 3–5
lock value block • 3–12, 6–64 to 6–65, 6–67 to 6–69
table of parameters • 6–57
table of services • 3–4
use by an application • 3–3, 6–58, 6–81
used in a demonstration application • 8–2 to 8–19
use for process synchronization • 6–57 to 6–69
using ASTs and blocking ASTs • 3–12 to 3–13, 6–58 to 6–60
Lock manager
See VMS lock manager
Lock modes • 3–5, 3–6
exclusive • 3–7
setting up • 3–6

Lock modes (Cont.)
table for compatibilities • 3–7
table of • 3–6
Locks
convention for use • 3–7, 3–9
getting information about • 3–4
illustration of queues • 3–10
levels of • 3–8 to 3–9
parent • 3–9
releasing • 3–4
requesting • 3–9
requesting conversion • 3–4
restrictive • 3–7
sharing • 3–6
sublock • 3–9
waiting for grant • 3–4
Lock status block • 3–5
checking for deadlock • 3–11
CONVERSION lock status • 3–5
GRANTED lock status • 3–5
requesting a lock value block • 6–64
WAITING lock status • 3–5
Lock value block
code example • 6–65 to 6–67
use by an application • 3–12, 6–64 to 6–65, 6–67, 8–6 to 8–9, 8–9

M

Mass Storage Control Protocol
See MSCP
Memory bottlenecks • 7–13 to 7–14
possible solutions • 7–13 to 7–14
Monitoring
cluster performance • 7–2 to 7–3, 7–16 to 7–21
MONITOR utility • 7–3
MSCP
server • 1–6
Server • 1–14
server bottlenecks • 7–7
Multiple-environment configuration • 1–4

N

Nontransparent DECnet-VAX communications • 3–29 to 3–31
code example demonstrating data sharing • 6–26 to 6–28
Nontransparent DECnet-VAX communications (Cont.)
code example demonstrating exception conditions • 6–75 to 6–81
code example demonstrating process synchronization • 6–69 to 6–74
code example demonstrating remote process creation • 6–4 to 6–5
versus transparent DECnet-VAX communications • 3–32

P

$PROCESS_SCAN
See process information system services
Parallelism Model • 4–17 to 4–24
advantages of using • 4–20
concept of queueing • 4–19
concept of self-scheduling • 4–19
disadvantages of using • 4–21
example of using • 4–23 to 4–24
illustration of • 4–18
implementation requirements • 4–21 to 4–23
using for faster completion • 5–3 to 5–4
using to maximize throughput • 5–4 to 5–5
Parent lock • 3–9
Performance
applications • 2–10, 3–7
monitoring • 7–2 to 7–3, 7–16 to 7–21
Process
failover • 6–81, 8–4
Process control system services
code example demonstrating a query for process information • 6–50 to 6–56
code example demonstrating process synchronization • 6–42 to 6–50
function • 3–22
illustration • 3–20
status codes • 3–23
support of DCL • 3–22
use by an application • 3–22
Process information system services
function • 3–24
illustration • 3–20
support of DCL • 6–41
use of $GETJPI system service • 3–24, 6–50
use of $PROCESS_SCAN system service • 3–24, 6–50
Process synchronization • 6–41 to 6–74
using DECnet-VAX communications • 6–69 to 6–74

Process synchronization (Cont.)
using lock management system services • 6–57 to 6–69
using process control system services • 6–41
using process information system services • 6–41 to 6–42
Programming techniques • 6–1
See data sharing
See exception conditions
See process synchronization
See remote process creation
Programming tools—VMS
See VMS programming tools

Q

Queues
CONVERSION • 3–5
GRANTED • 3–5
WAITING • 3–5

R

Read-Only global section
See also data sharing
code example demonstrating data sharing • 6–34 to 6–40
function • 6–33
Remote process creation • 6–1 to 6–16
using nontransparent DECnet-VAX communications • 6–3 to 6–5
using the VMS batch facility • 6–5 to 6–16
using transparent DECnet-VAX communications • 6–2 to 6–3
Resource name
locating on clusterwide lock database • 3–9 to 3–10
lock status block • 3–5
use by VMS lock manager • 3–2, 3–5
Resources
accessing • 3–6
availability • 3–6
granularity • 3–8
holding name space • 3–6
preventing access • 3–7
synchronizing access • 3–4

S

SHOW CLUSTER command • 7–3
SHOW commands • 7–3
SHOW DEVICE/FILES command • 7–3
SHOW DEVICE/FULL command • 7–3
SHOW DEVICE/SERVED command • 7–3
Single-node VMS programming tools
common event flags • 3–33
$CREPRC system service • 3–33
logical names • 3–33
not supported clusterwide • 2–1
permanent and temporary mailboxes • 3–32
work arounds • 3–32 to 3–34
writeable global sections • 3–34
Specialized hardware server • 4–11
See also Client-Server Model
Star Coupler • 1–3
Sublock • 3–9
System disks • 1–7
common • 1–7
I/O bottlenecks • 7–9, 7–10
individual • 1–7

T

Tape drive
configurations • 1–7
connections to CPUs • 1–7
connections to HSC subsystems • 1–7
Terminals
configurations • 1–8
connected to terminal server • 1–3
Terminal server
benefits • 1–9
failover • 1–9
LAT • 1–3
Transparent DECnet-VAX communications • 3–28 to 3–29
code example demonstrating data sharing • 6–17 to 6–26
code example demonstrating remote process creation • 6–2 to 6–3
versus nontransparent DECnet-VAX communications • 3–32

U

UAF • 1–4
UIC • 1–4
  with VMS lock manager • 3–5
  with VMS Record Management Services • 3–16
User Authorization File
  See UAF
User Identification Code
  See UIC

V

VAX ACMS • 5–15 to 5–18
  features for application design • 5–15 to 5–17
  using in a VAXcluster system • 5–18
VAXcluster system
  advantages for application development • 2–1, 2–2
  CI-based • 1–9
  comparison of supported types • 1–9
  CPU-intensive application • 2–7 to 2–9
  definition • 1–1
  displaying configuration information • 7–3
  hardware advantages • 1–1 to 1–9
  I/O-intensive applications • 2–5 to 2–7
  illustration of software components • 1–11
  Local Area • 1–9
  Mixed-Interconnect • 1–9
  monitoring performance • 7–2 to 7–3, 7–16 to 7–21
  partitioning • 1–13
  performance considerations • 7–1 to 7–21
  state transitions • 1–13
  VMS software advantages • 1–11 to 1–14
VAX DBMS • 5–18 to 5–19
  features for application design • 5–18 to 5–19
  using in a VAXcluster system • 5–19
VAX DNS • 5–21 to 5–22
  features for application design • 5–21
  using in a VAXcluster system • 5–22
VAX PA • 7–18 to 7–21
VAX PCA • 7–17 to 7–18
VAX Performance Advisor
  See VAX PA
VAX Performance and Coverage Analyzer
  See VAX PCA
VAX Rdb/VMS • 5–20 to 5–21
  features for application design • 5–20
VAX Rdb/VMS (Cont.)
  using in a VAXcluster system • 5–20 to 5–21
VAX RMS Journaling • 5–7 to 5–9
  features for application design • 5–7 to 5–8
  using in a VAXcluster system • 5–8 to 5–9
VAX Software Performance Monitor
  See VAX SPM
VAX SPM • 7–17
VAX Volume Shadowing • 1–6, 5–9 to 5–10
  disks dual-ported between HSC subsystem • 1–6
  features for application design • 5–9
  hardware redundancy • 1–3
  using in a VAXcluster system • 5–9 to 5–10
Victim process • 3–11
VMS batch facility
  code example demonstrating checkpointing • 6–13 to 6–16
  code example demonstrating remote process creation • 6–6 to 6–16
  distributed job controller • 1–14
  functions • 3–19
  illustration • 3–19
  restart capability • 3–20
  use of checkpointing • 6–13
  use of distributed job controller • 3–20
VMS DECwindows • 5–10 to 5–12
  features for application design • 5–10 to 5–11
  using in a VAXcluster system • 5–11 to 5–12
VMS lock manager • 3–2 to 3–13
  detecting deadlocks • 3–11
  displaying statistics • 7–3
  illustration • 3–2
  interface with VMS Record Management Services • 3–16
  lock management system services • 3–3 to 3–5
  lock queues • 3–9 to 3–10
  overview • 1–13
  resource manager • 3–3
  resource name • 3–2, 3–5
  use of clusterwide lock database • 3–2, 3–9 to 3–10
VMS programmimng tools
  not supported in a VAXcluster system • 2–1
VMS programming tools • 3–1 to 3–32
  DECnet-VAX • 3–25 to 3–32
  process control system services • 3–20 to 3–23
  process information system services • 3–24 to 3–25
  VMS batch facility • 3–19 to 3–20
  VMS lock manager • 3–2 to 3–13
  VMS Record Management Services • 3–14 to 3–18

VMS Record Management Services
code example demonstrating data sharing • 6–29 to 6–32
functions • 3–15 to 3–16
global buffering • 3–17
illustration • 3–14
interface with the VMS lock manager • 3–16
local buffering • 3–17
performance considerations with global buffering • 3–18
use of $QIO system services • 3–18
use of automatic record locking • 3–16
use of XQP operations • 3–18
use with high-level programming languages • 3–14
Volume shadowing
See VAX Volume Shadowing

W

WAITING queue • 3–5
Workloads
distributing • 1–8, 1–14

VAXcluster Systems
Introduction to
VAXcluster Application Design
AA–JP32B–TE

READER'S COMMENTS

Your comments and suggestions help us to improve the quality of our publications.

For which tasks did you use this manual? (Circle your responses.)

(a) Installation (c) Maintenance (e) Training
(b) Operation/use (d) Programming (f) Other (Please specify.) _____

Did the manual meet your needs? Yes ☐ No ☐ Why? _____

Please rate the manual in the following categories. (Circle your responses.)

	Excellent	Good	Fair	Poor	Unacceptable
Accuracy (product works as described)	5	4	3	2	1
Clarity (easy to understand)	5	4	3	2	1
Completeness (enough information)	5	4	3	2	1
Organization (structure of subject matter)	5	4	3	2	1
Table of Contents, Index (ability to find topic)	5	4	3	2	1
Illustrations, examples (useful)	5	4	3	2	1
Overall ease of use	5	4	3	2	1
Page Layout (easy to find information)	5	4	3	2	1
Print Quality (easy to read)	5	4	3	2	1

What things did you like *most* about this manual? _____

What things did you like *least* about this manual? _____

Please list and describe any errors you found in the manual.

Page	Description/Location of Error
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions for improving this manual: _____

Name _____ Job Title _____
Street _____ Company _____
City _____ Department _____
State/Country _____ Telephone Number _____
Postal (ZIP) Code _____ Date _____

Fold Here and Tape

Affix
Stamp
Here

DIGITAL EQUIPMENT CORPORATION
CORPORATE USER PUBLICATIONS
200 FOREST STREET MRO1-3/L12
MARLBOROUGH, MA 01752-9101

Fold Here