EY-2280E-SG-0001

VAX/VMS INTERNALS II

Student Workbook

Prepared by Educational Services
of
Digital Equipment Corporation

Second Edition, October 1986

Printed in U.S.A.

The manuscript for this book was created using DIGITAL Standard Runoff. Book production was done by Educational Services Development and Publishing in Nashua, NH.

CONTENTS

SG STUDENT GUIDE

INTRODUCTION SG-3
GOALS. SG-4
NON-GOALS. SG-5
PREREQUISITES. SG-5
RESOURCES. SG-5
COURSE MAP SG-6
COURSE OUTLINE SG-7

1 SYSTEM PROCESSES

INTRODUCTION 1-3
OBJECTIVES 1-3
RESOURCES. 1-4
Reading. 1-4
Source Modules 1-4
TOPICS 1-5
OVERVIEW OF SYSTEM PROCESSES 1-7
THE JOB CONTROLLER 1-10
Job Controller Functions 1-11
SYMBIONTS. 1-13
Output Symbionts 1-14
Symbiont Services. .1-17
User-Supplied Symbionts. 1-18
THE ERROR LOGGER 1-19
Error Logging. 1-20
Waking the Error Logger. 1-21
OPERATOR COMMUNICATION .1-23
SUMMARY. 1-25

2 FORMING, ACTIVATING AND TERMINATING IMAGES

INTRODUCTION 2-3
OBJECTIVES 2-3
RESOURCES. 2-4
Reading. 2-4
Source Modules 2-4
TOPICS 2-5
FORMING AN IMAGE 2-7
Program Sections 2-7
Format of an Image File. 2-8
Image Section Descriptor Formats 2-9
Format of the Image Header 2-10
IMAGE ACTIVATION AND START-UP. 2-11
Mapping an image to Virtual Address Space. 2-12
Bringing Pages of Image into Physical Memory 2-13

Translating Virtual to Physical Addresses. 2-14
Locating Image Pages on Disk 2-15
Summary of Image Formation and Activation. 2-16
Image Start-Up 2-17
Installing Files 2-18
The Known File Entry 2-19
IMAGE EXIT AND RUNDOWN 2-20
Exit System Service. 2-20
Termination Handlers 2-21
DCL Operation. 2-22
SUMMARY. 2-23
APPENDIX - LINKER CLUSTERS 2-24

3 PAGING

INTRODUCTION 3-3
OBJECTIVES 3-3
RESOURCES. 3-4
Reading. 3-4
Additional Suggested Reading 3-4
Source Modules 3-4
TOPICS 3-5
BASIC VIRTUAL ADDRESSING 3-7
Virtual Address Space. 3-8
Associating Virtual and Physical Addresses 3-9
S0 Virtual Address Translation 3-10
Hardware Checks. 3-11
Process Headers in S0 Space. 3-12
Page Tables. 3-13
Page Table Mapping .3-14
Referencing a P0 Virtual Address 3-15
OVERVIEW OF PAGE FAULT HANDLING. 3-16
Working Set List 3-17
Image Section Descriptor Formats 3-18
Process Section Table (PST). 3-19
Process Section Table Entry. 3-20
HOW PTEs, PSTEs ARE FILLED IN. 3-21
Process Header 3-22
Sample Program 3-24
Template for Process Paging Example. 3-25
MORE ON PAGING 3-26
Different Forms of Page Table Entry. 3-27
The Paging File. 3-28
PFN DATABASE 3-29
Using a Process CRF Page 3-30
Initial Status of Process CRF Page 3-31
Page Fault on Process CRF Page (Step 1). 3-32

Page Fault on Process CRF Page (Step 2). 3-33
Page Fault on Process CRF Page (Step 3). 3-34
Removing Process CRF Page from Working Set 3-35
Process CRF Page Moved from MPL to FPL 3-36
Removing Process CRF Page from Memory. 3-37
DATA STRUCTURES USED BY THE PAGER. 3-38
GLOBAL PAGING DATA STRUCTURES. 3-39
Global Page Table. 3-39
Relationship Among Global Section Data Structures. 3-40
Using a Global Read/Write Page 3-41
Initial Status of Global Read/Write Section Page 3-42
Adding Global Read/Write Section Page to Working Set 3-43
Initial Status of PTE of Second Process. 3-44
Adding Global Read/Write Section Page
to Second Working Set 3-45
Removing Global Read/Write Section Page from Working Set .3-46
Removing Global Read/Write Section Page from Memory. . . .3-47
SUMMARY OF THE PAGER 3-48
APPENDIX - SUPPLEMENTARY INFORMATION 3-51
PROCESS VIRTUAL ADDRESS TRANSLATION. 3-51
PHYSICAL ADDRESS SPACE 3-52
IMAGE ACTIVATOR AND PROCESS HEADER 3-53
Image Activator. 3-53
PAGE READ CLUSTERING 3-54
Why Cluster Pages. 3-54
How a Cluster is Made. 3-54
Maximum Cluster Size Determination 3-55
Changing/Controlling Cluster Size. 3-56

4 SWAPPING

INTRODUCTION 4-3
OBJECTIVES 4-4
RESOURCES. 4-5
Reading. 4-5
Additional Suggested Reading 4-5
Source Modules 4-5
TOPICS 4-7
COMPARISON OF PAGING AND SWAPPING. 4-9
Similarities 4-9
OVERVIEW OF THE SWAPPER, THE SYSTEM-WIDE MEMORY MANAGER. . . .4-10
Swapper Main Loop. 4-11
MAINTAINING THE FREE PAGE COUNT. 4-12
How Modified Page Writer Gathers Pages 4-13
Modified Page Write Clustering 4-14
Trimming and Swapping Working Sets 4-15
Expanding and Shrinking Working Sets 4-18
WAKING THE SYSTEM-WIDE MEMORY MANAGER. 4-19

OUTSWAPPING A PROCESS. 4-20
Outswap Rules. 4-21
Locating Disk Files for Swap 4-22
How Swapper's P0 Page Table is Used to Speed Swap I/O. . .4-23
Swapper's Pseudo Page Tables 4-24
Partial Outswaps and the Process Header. 4-25
INSWAPPING A PROCESS 4-26
INSWAP RULES 4-27
SUMMARY. 4-29
APPENDIX - SWAPPER - MAIN LOOP 4-31

5 I/O CONCEPTS AND FLOW

INTRODUCTION 5-3
OBJECTIVES 5-3
RESOURCES. 5-4
Reading. 5-4
Source Modules 5-4
TOPICS 5-5
OVERVIEW OF I/O COMPONENTS AND FLOW. 5-7
COMPONENTS OF THE I/O SYSTEM 5-9
THE I/O DATABASE 5-14
METHODS OF DATA TRANSFER 5-16
SUMMARY. 5-16

6 RMS IMPLEMENTATION AND STRUCTURE

INTRODUCTION 6-3
OBJECTIVES 6-3
RESOURCES. 6-4
Reading. 6-4
Source Modules 6-4
TOPICS 6-5
USER-SPECIFIED DATA STRUCTURES 6-7
RMS INTERNAL DATA STRUCTURES 6-9
Process I/O Segment. 6-10
The Overall Control Information Areas. 6-11
Impure Data Areas. .6-12
Process Impure Data Area (PIO$GW_PIOIMPA). 6-12
Image Impure Data Area (PIO$GW_IIOIMPA). 6-12
File-Oriented and Record-Oriented Data Structures. 6-13
RMS PROCESSING 6-17
SUMMARY. 6-21
APPENDIX - RMS FUNCTIONS AND MODULES 6-23

7 VMS IN A MULTIPROCESSING ENVIRONMENT

INTRODUCTION 7-3
OBJECTIVES 7-3
RESOURCES. 7-3
    Reading. 7-3
    Source Modules 7-3
TOPICS 7-5
MULTIPROCESSING ENVIRONMENTS 7-7
NETWORKS 7-9
THE VAX-11/782 7-10
    Definitions. 7-11
    VAXclusters. 7-15
    VAXcluster Benefits. 7-15
SUMMARY. 7-17
GLOSSARY 7-19
APPENDIX - THE VAX-11/782. 7-21
    Definitions. 7-21
    Initialization 7-23
    Hooks into VMS 7-24
    SCB Changes. 7-24
    Secondary Processor States 7-25
    Exceptions for CPU2. 7-26
    MA780. 7-26
    Faults 7-27
    Restrictions 7-27

8 VMS IN A VAXcluster ENVIRONMENT

INTRODUCTION 8-3
OBJECTIVES 8-3
RESOURCES. 8-4
    Reading. 8-4
    Source Modules 8-4
TOPICS 8-5
OVERVIEW OF VAXcluster FEATURES. 8-6
    System Processes in a VAXcluster 8-10
    Cache Server Process 8-11
    Cluster Server Process 8-11
    Configure Process. .8-11
    The Connection Manager 8-13
    Distributed Lock Manager 8-14
    Distributed File System. 8-15
    Record Management Services (RMS) 8-15
    Class Driver 8-16
    SCS (Systems Communications Services). 8-16
    Port Drivers 8-16
    Distributed Batch and Print Services 8-17
    I/O in a VAXcluster Environment. 8-18

JOINING A VAXcluster 8-24
LEAVING A VAXcluster 8-25
ADDITIONAL CONSIDERATIONS IN A VAXcluster ENVIRONMENT. 8-27
SUMMARY. 8-29
APPENDIX - VAXcluster SYSGEN PARAMETERS. 8-31

EXERCISES. EX-3

TESTS. TP-3

FIGURES

1 Communication with the Job Controller 1-10
2 Job Controller Code Flow. 1-12
3 Output Symbiont Flow Diagram. 1-14
4 User Symbiont Flow Diagram. 1-16
5 Overview of Error Logging 1-19
6 Code Flow for Error Logger Wake 1-21
7 Code Flow for Error Logger. 1-22
8 Overview of OPCOM 1-23

1 Creating an Image File. 2-8
2 Image Section Descriptor Formats. 2-9
3 The Image Header. 2-10
4 Mapping to Virtual Address Space. 2-12
5 Bringing Pages into Physical Memory 2-13
6 Translating Virtual to Physical Addresses 2-14
7 Locating Image Pages on Disk. 2-15
8 Summary of Image Formation and Activation 2-16
9 Transfer Address Array Formats. 2-17
10 The Known File Database 2-18
11 KFE with a Resident Header. 2-19
12 Exit System Service .2-20
13 Termination Handlers. 2-21
14 DCL Operation 2-22
15 Linker Clusters 2-26

1 Physical and Virtual Memory 3-7
2 Virtual Address Space 3-8
3 Associating Virtual and Physical Addresses. 3-9

4 S0 Virtual Address Translation. 3-10
5 Hardware Checks if Access Allowed 3-11
6 PHDs and S0 Page Table in S0 Space. 3-12
7 Page Table Mapping. .3-14
8 Referencing a P0 Virtual Address. 3-15
9 Resolving Page Faults 3-16
10 Working Set List. .3-17
11 Image Section Descriptor Formats. 3-18
12 Process Section Table 3-19
13 Process Section Table Entry 3-20
14 How PTEs, PSTEs are Filled In 3-21
15 The Process Header. .3-22
16 Overview of Page Fault Handling 3-23
17 Template for Process Paging Example 3-25
18 Free and Modified Page Lists. 3-26
19 Different Forms of Page Table Entry 3-27
20 Page File Control Block 3-28
21 PFN Database. 3-29
22 Initial Status of Process CRF Page. 3-30
23 Page Fault on Process CRF Page (Step 1) 3-31
24 Page Fault on Process CRF Page (Step 2) 3-32
25 Page Fault on Process CRF Page (Step 1) 3-33
26 Removing Process CRF Page from Working Set. 3-34
27 Moving Pages from MPL to FPL. 3-35
28 Removing Process CRF Page from FPL. 3-36
29 Data Structures Used by the Pager 3-37
30 Process PTEs Map to Global PTEs 3-38
31 Relationship Among Global Section Data Structures 3-40
32 Initial Status of Global Read/Write Section Page. 3-42
33 Adding Global Read/Write Section Page to Working Set. . .3-43
34 Initial Status of PTE of Second Process Mapping
the Same Global Section. 3-44
35 Adding Global Read/Write Section Page to
Second Working Set .3-45
36 Removing Global Read/Write Section Page
from Working Set 3-46
37 Removing Global Read/Write Section Page from Memory . . .3-47
38 Summary of the Pager. 3-48
39 Process Virtual Address Translation 3-51
40 Physical Address Space. 3-52
41 Image File and Process Header 3-53

1 Swapper Main Loop 4-11
2 How Modified Page Writer Gathers Pages. 4-13
3 Expanding and Shrinking Working Sets. 4-18
4 Locating Disk Files for Swap. 4-22
5 How Swapper's P0 Table is Used to Speed Swap I/O. 4-23
6 Swapper's Pseudo Page Tables. 4-24
7 Overview of Swapper Functions 4-29

1 Input/Output Flow (Brief) 5-7
2 Input/Output (Full) 5-8
3 RMS Interfaces. 5-9
4 $QIO and FDT Routines 5-10
5 XQPs. 5-11
6 ACPs. 5-12
7 Components of Device Drivers. 5-13
8 Summary Layout of I/O Database. 5-15
9 Buffered I/O. 5-16
10 Direct I/O. 5-16

1 Virtual Address Space 6-8
2 Process I/O Segment in Pl Space 6-9
3 IFAB and IRAB Tables. 6-15
4 IFAB and Associated Blocks. 6-16
5 RMS Interfaces. 6-17
6 RMS Dispatching 6-18
7 RMS Components in a GET Operation 6-19

1 Relationship Between Different
Multiprocessing Configurations 7-7
2 Relationship Between Different
Multiprocessing Configurations 7-8
3 Sample VAX-11/782 Configuration 7-12
4 Secondary Processor States. 7-13
5 MP.EXE in Nonpaged Pool 7-14
6 VAXcluster Hardware Configuration 7-16
7 Sample VAX-11/782 Configuration 7-22
8 Secondary Processor States. 7-25
9 MP.EXE Loaded into Nonpaged Pool. 7-28

1 Relationships Between Different
Multiprocessor Configurations. 8-7
2 Sample VAXcluster Hardware Configuration. 8-8
3 VAXcluster Software Components. 8-12
4 Cluster I/O Database. 8-18
5 VAXcluster Hardware/Software Block Diagram. 8-19
6 Flow of Standard I/O Operations 8-20
7 Cluster I/O Available on Version 4.0 8-22
8 Data Flow for an MSCP Request 8-23
9 VAXcluster System Start-Up Flow 8-24

TABLES

1 VMS System Processes. 1-8
2 Processes Created by STARTUP.COM. 1-9

1 PSECT Attributes. 2-7

2 How Termination Handlers are Established for Each Access Mode 2-21
3 Summary of Image Formation, Activation, and Termination. .2-23
4 SYSGEN Parameters Related to Image Formation, Activation, and Termination 2-23

1 Where Memory Management Information is Stored 3-38
2 SYSGEN Parameters Related to Paging 3-49
3 Cluster Sizes and Where They are Stored 3-55

1 Differences Between Paging and Swapping 4-9
2 Order of Search for Trim and Swap Candidates. 4-16
3 Description of Special Swapper Flags. 4-17
4 Selected Events that Cause the Swapper or Modified Page Writer to be Awakened. 4-19
5 Rules for Scan of Working Set List on Outswap 4-21
6 Rules for Rebuilding the Working Set List and the Process Page Tables at Inswap. 4-27
7 SYSGEN Parameters Relevant to the Swapper 4-30

1 The I/O Database. .5-14

1 RMS Calling MACROS and the Resulting Code 6-20
2 RMS Functions and Primary Module Names. 6-23

1 Different Multiprocessing Implementations 7-8
2 Different Multiprocessing Implementations 7-17
3 System Locations and the Resulting MP Locations 7-29

1 Different Multiprocessing Implementations 8-7
2 System Processes Specific to a VAXcluster 8-10
3 VAXcluster Processes Created by STARTUP.COM 8-10
4 Selected VAXcluster SYSGEN Parameters 8-29

EXAMPLES

1 SHOW SYSTEM Output. 1-7
2 Listing Buffer Declarations for Error Logger. 1-20
3 OPCOM Main Code 1-24

1 MACRO Program with Four PSECTs. 3-24

1 Swapper - Main Loop 4-31

1 SHOW SYSTEM Output for a VAXcluster 8-9
2 Booting a VAXcluster System 8-26
3 Leaving a VAXcluster. 8-26

Student Guide

INTRODUCTION

The VAX/VMS Operating System Internals course is intended for the student who requires an extensive understanding of the components, structures, and mechanisms contained in the VAX/VMS operating system. It is also an aid for the student who will go on to examine and analyze VAX/VMS source code.

This course provides a discussion of the interrelationships among the logic or code, the system data structures, and the communication/synchronization techniques used in major sections of the operating system.

Technical background for selected system management and application programmer topics is also provided. Examples of this information include:

- The implications of altering selected system parameter values
- The implications of granting privileges, quotas, and priorities
- How selected system services perform requested actions.

Information is provided to assist in subsequent system-related activities such as:

- Writing privileged utilities or programs that access protected data structures
- Using system tools (for example, the system map, the system dump analyzer, and the MONITOR program) to examine a running system or a system crash.

This course concentrates on the software components included in (and the data structures defined by) the linked system image. Associated system processes, utilities, and other programs are discussed in much less detail.

GOALS

- Describe the contents, use, and interrelationship of selected VAX/VMS components (job controller, ancillary control processes, symbionts), data structures (SCB, PCB, JIB, PHD, P1 space), and mechanisms (synchronization techniques, change mode dispatching, exceptions and interrupts).

- Describe and differentiate system context and process context.

- Discuss programming considerations and system management alternatives in such problems as:

  - Assigning priorities in a multiprocess application
  - Controlling paging and swapping behavior for a process or an entire system
  - Writing and installing a site-specific system service

- Use system-supplied debugging tools and utilities (for example, SDA, XDELTA) to examine crash dumps and to observe a running system.

- Describe the data structures and software components involved when a process is created or deleted, an image is activated and rundown, and the operating system is initialized.

- Describe how the following interrupt service routines are implemented:

  - AST delivery
  - Scheduling
  - Hardware clock
  - Software timers

- Briefly describe the components of the I/O system, including system services, RMS, device drivers and XQPs.

- Briefly describe how RMS processes I/O requests, including the user-specified and internal data structures involved.

- Describe certain additional VMS mechanisms used on a VAX system in a cluster (for example, synchronization and communication mechanisms).

NON-GOALS

- Writing device drivers (see the VAX/VMS Device Driver course)
- Writing ancillary control processes, ACPs (see the VAX/VMS Device Driver course)
- Comprehensive understanding of RMS internals
- DECnet internals (see the DECnet courses)
- Layered product internals
- Command language interpreter internals
- System management of a VAXcluster

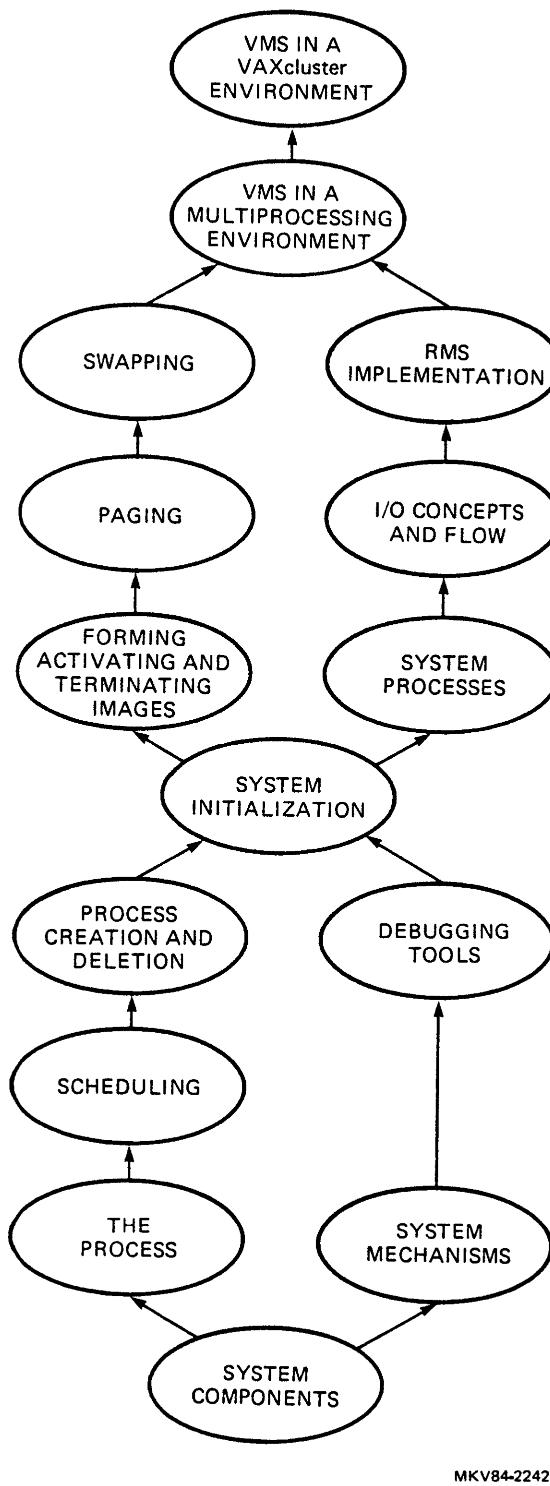
PREREQUISITES

- Ability to program in at least one VAX native language. This may be obtained through language programming experience and completion of an appropriate language programming course (for example, Assembly Language Programming in VAX-11 MACRO). In addition, completion of the Introduction to VAX-11 Concepts course is recommended.
- Ability to read and comprehend programs written in VAX-11 MACRO is required. In addition, ability to program in VAX-11 MACRO or BLISS is recommended.
- Completion of one of the Utilizing VMS Features courses.

RESOURCES

1. <u>VAX/VMS Internals and Data Structures</u>
2. <u>VAX/VMS System Dump Analyzer Reference Manual</u>
3. <u>VMS Internals I and II Source Listings</u>

COURSE MAP


VMS IN A VAXcluster ENVIRONMENT
VMS IN A MULTIPROCESSING ENVIRONMENT
SWAPPING
RMS IMPLEMENTATION
PAGING
I/O CONCEPTS AND FLOW
FORMING ACTIVATING AND TERMINATING IMAGES
SYSTEM PROCESSES
SYSTEM INITIALIZATION
PROCESS CREATION AND DELETION
DEBUGGING TOOLS
SCHEDULING
THE PROCESS
SYSTEM MECHANISMS
SYSTEM COMPONENTS
MKV84-2242

COURSE OUTLINE

I. System Components

A. How VMS Implements the Functions of an Operating System

B. How and When Operating System Code is Invoked

C. Interrupts and Priority Levels

D. Location of Code and Data in Virtual Address Space

E. Examples of Flows for:

1. Hardware clock interrupt
2. System event completion
3. Page fault
4. RMS request for I/O
5. $QIO request for I/O

F. Examples of System Processes

1. Operator Communication (OPCOM)
2. Error logger (ERRFMT)
3. Job controller (JOB_CONTROL)
4. Symbionts (SYMBIONT_n)

G. Software Components of DECnet-VAX

II. The Process

A. Process vs. System Context

B. Process Data Structures Overview

1. Software context information
2. Hardware context information

C. Virtual Address Space Overview

1. S0 space (operating system code and data)
2. P0 space (user image code and data)
3. P1 space (command language interpreter, process data)

D. SYSGEN Parameters Related to Process Characteristics

III. System Mechanisms

A. Hardware Register and Instruction Set Support

B. Synchronizing System Events

1. Hardware Interrupts
2. Software Interrupts
   Example: Fork Processing
3. Requesting Interrupts
4. Changing IPL
5. The Timer Queue and System Clocks

C. Process Synchronization Mechanisms

1. Mutual Exclusion Semaphores (MUTEXes)
2. Asynchronous System Traps (ASTs)
3. VAX/VMS Lock Manager

D. Exceptions and Condition Handling

E. Executing Protected Code

1. Change Mode Dispatching
2. System Service Dispatching

F. Miscellaneous Mechanisms

1. System and Process Dynamic Memory (Pool)

G. SYSGEN Parameters Controlling System Resources

IV. Debugging Tools

A. VAX/VMS Debugging Tools

B. The System Dump Analyzer (SDA)

1. Uses
2. Requirements
3. Commands

C. The System Map File

D. Crash Dumps and Bugchecks

1. How bugchecks are generated
2. Sample stacks after bugchecks
3. Sample crash dump analysis

E. The DELTA and XDELTA Debuggers

V. Scheduling

A. Process States

1. What they are (current, computable, wait)
2. How they are defined
3. How they are related

B. How Process States are Implemented in Data Structures

1. Queues
2. Process data structures

C. The Scheduler (SCHED.MAR)

D. Boosting Software Priority of Normal Processes

E. Operating System Code that Implements Process State Changes

1. Context switch (SCHED.MAR)
2. Result of system event (RSE.MAR)

F. Steps at Quantum End

1. Automatic working set adjustment

G. Software Priority Levels of System Processes

VI. Process Creation and Deletion

A. Process Creation

1. Roles of operating system programs
2. Creation of process data structures

B. Types of Processes

C. Initiating Jobs

1. Interactive
2. Batch

D. Process Deletion

E. SYSGEN Parameters Relating to Process Creation and Deletion

VII. System Initialization and Shutdown

A. System Initialization Sequence

B. Function of initialization programs

C. How memory is structured and loaded

D. Start-up command procedures

E. How hardware differences between CPUs affect initialization

F. Shutdown procedures and their functions

G. Auto-restart sequence

H. Power-fail recovery

VIII. System Processes

A. For selected VAX/VMS processes:

1. Job controller
2. Symbionts
3. Error Logger
4. OPCOM

We will be describing their:

1. Primary Functions
2. Implementation
3. Methods of communication with other VMS components
4. Basic internal structure (on a module basis)

IX. Forming, Activating and Terminating Images

A. Forming an Image

1. PSECTs in source/object modules
2. Format and use of the image header

B. Image Activation and Start-Up

1. Mapping virtual address space
2. Overview of related data structures
3. Image start-up (SYS$IMGSTA)
4. Installing Known Files

C. Image Exit and Rundown

1. $EXIT system service
2. Termination Handlers
3. DCL Sequence

D. SYSGEN parameters relating to image formation, activation and termination

X. Paging

A. Basic Virtual Addressing

1. Virtual and physical memory
2. Page table mapping

B. Overview of Page Fault Handling

1. Resolving page faults
2. Data structures in the process header

C. More on Paging

1. Free and modified page lists
2. The paging file
3. Cataloging pageable memory (the PFN database)

D. Global Paging Data Structures

E. Summary of the Pager

XI. Swapping

A. Comparison of Paging and Swapping

B. Overview of the Swapper, the System-Wide Memory Manager

C. Maintaining the Free Page Count

1. Write Modified Pages
2. Shrink Working Sets
3. Outswap Processes

D. Waking the System-Wide Memory Manager

E. Outswapping a Process

1. Swap files
2. Scatter/Gather
3. Partial Outswaps

F. Inswapping a Process

XII. I/O Concepts and Flow

A. Overview of I/O components and flow

B. Components of I/O system

1. RMS
2. I/O system services
3. XQPs, ACPs
4. Device drivers

C. The I/O database

1. Driver tables
2. IRPs
3. Control blocks

D. Methods of data transfer

XIII. RMS Implementation and Structure

A. User-specified data structures (FABs, RABs, and so on)

B. RMS Internal Data Structures

1. Process I/O Control Page (for example, default values, I/O segment area)
2. File-Oriented and Record-Oriented Data Structures (IFAB, IRAB, BufDescBlk, I/O Buffer)

C. RMS Processing

1. RMS Dispatching
2. RMS routines and data structures
3. Examples of flows of some common operations

XIV. VMS in a Multiprocessing Environment

A. Loosely coupled processors

B. Tightly coupled processors (11/782)

1. MP.EXE structures
2. Scheduling differences
3. Startup /shutdown

C. Clustered processors

XV. VMS in a VAXcluster Environment

A. Cluster synchronization and communication mechanisms

1. Distributed lock manager
2. Distributed job controller
3. Interprocessor communication

B. System initialization and shutdown differences

1. VMB, INIT and SYSINIT differences
2. Joining a cluster
3. Leaving a cluster

C. SYSGEN parameters relevant to the VAXcluster environment

D. Relevant system operations

System Processes

INTRODUCTION

VMS consists of many pieces all working together to perform specific functions. Some parts of VMS work in user process context (such as System Services) or in system context (such as Scheduling). There are still other duties that must be performed in process context but are not 'called' by the user. These parts run in the context of their own process. They are known as "System Processes."

We will be examining several of these processes in this module, including:

- Job Controller (JOB_CONTROL)
- Print Symbiont (SYMBIONT_n)
- Error Format (ERRFMT)
- Operator Communications (OPCOM)

OBJECTIVES

1. To describe, for selected VAX/VMS processes, their
   - Functions, primary and otherwise
   - Implementation
   - Methods of communication with other VMS components
2. To describe, for certain VAX/VMS processes, their internal structure (on a module basis)

RESOURCES

Reading

- VAX/VMS Internals and Data Structures, chapters on Error Handling plus Interactive and Batch Jobs.

Source Modules

Facility Name	Module Name
JOBCTL	CONTROL SCHEDULER UNSOLICIT
ERRFMT	ERRFMT
SYS	ERRORLOG
OPCOM	OPCOMMAIN OPCOMINI
PRTSMB	PRTSMB SMBSRVSHR

TOPICS

I. For selected VAX/VMS processes, describe their

   A. Primary Functions

   B. Implementation

   C. Methods of communication with other VMS components

   D. Basic internal structure (on a module basis)

II. The selected system processes are:

   A. Job Controller

   B. Symbionts — Whenever a queue is started will support up to 16 devices

   C. Error Logger

   D. OPCOM

OVERVIEW OF SYSTEM PROCESSES

```
VAX/VMS V4.0 on node COMICS  6-NOV-1984 10:40:57.65  Uptime  0 02:22:14
  Pid    Process Name    State  Pri      I/O       CPU       Page flts
00000080 NULL            COM      0        0   0 00:18:42.40         0
00000081 SWAPPER         HIB     16        0   0 00:00:21.10         0
00000103 MARSH           CUR      4      213   0 00:00:04.59       849
00000085 ERRFMT          HIB      7     1165   0 00:00:09.92       140
00000087 OPCOM           LEF      8      202   0 00:00:02.15       181
00000088 JOB_CONTROL     HIB      8     2336   0 00:00:36.37       188
0000008A VAXsim_Monitor  HIB      7      483   0 00:00:06.00       315
0000008D SYMBIONT_0001   COM      4     1377   0 00:08:26.51      2613
0000008E SPIDERMAN       LEF      4     2412   0 00:00:34.72       699
00000090 NETACP          HIB      9     2835   0 00:00:53.49      5800
00000091 EVL             HIB      4       79   0 00:00:02.52      2138
00000092 REMACP          HIB      9       74   0 00:00:00.56       123
00000094 THE_FLASH       LEF      7      947   0 00:00:15.53      2886
0000009A BATMAN          LEF      7     6659   0 00:02:20.76      8142
0000009B CAPT_MARVEL     LEF      7    13420   0 00:08:46.85     32485
0000009D DR_STRANGE      LEF      4    11665   0 00:04:05.12     23536
000000A3 SILVER_SURFER   LEF      4      923   0 00:00:30.45      2075
000000BC KAL-EL          LEF      4     3879   0 00:01:46.67      9493
000000C6 MR_FANTASTIC    LEF      4     6042   0 00:01:07.37      6730
000000C7 SYSTEM          LEF      4     3998   0 00:00:44.44      2375
000000CD DR_XAVIER       LEF      4      702   0 00:00:19.65      2671
000000D9 BATCH_891       COM      4     4033   0 00:03:25.23     13888
000000E6 BRUCE_BANNER    LEF      4      259   0 00:00:05.79       952
000000E7 JON_JONES       LEF      4     1030   0 00:00:16.58      2718
000000ED BATCH_924       COM      4      862   0 00:00:36.38      2646
```

Example 1 SHOW SYSTEM Output

Table 1 VMS System Processes

Process Name	Base Priority	Image Name	Comments
NULL	0	part of SYS.EXE	
SWAPPER	16	part of SYS.EXE	System-wide memory manager
ERRFMT	7	ERRFMT	Cleans up error log buffer
OPCOM	6	OPCOM.EXE	Operator communication manager
JOB_CONTROL	8	JOBCTL.EXE	Queue and accounting manager
SYMBIONT_n	4	PRTSYMB.EXE	Output symbionts
NETACP	8	NETACP.EXE	DECnet ACP
EVL	4	EVL.EXE	Network event logger
REMACP	8	REMACP.EXE	Remote ACP

Default

Sysgen>Set ACP_BASE_PRIO n

Table 2 Processes Created by STARTUP.COM

Process Name Image	Error Log File	Base Priority	Privileges	UIC
ERRFMT				
ERRFMT	errfmt_error	7	BYPASS, CMKRNL, WORLD	[1,6]
OPCOM				
OPCOM	opcom_error	6	CMKRNL,EXQUOTA, OPER,SYSPRV, WORLD,NETMBX, SETPRV	[1,4]
JOB_CONTROL				
JOBCTL	job_control_error	8	SETPRV	[1,4]

- All images reside in SYS$SYSTEM
- All error log files reside in SYS$~~MANAGER~~ Errorlog

THE JOB CONTROLLER

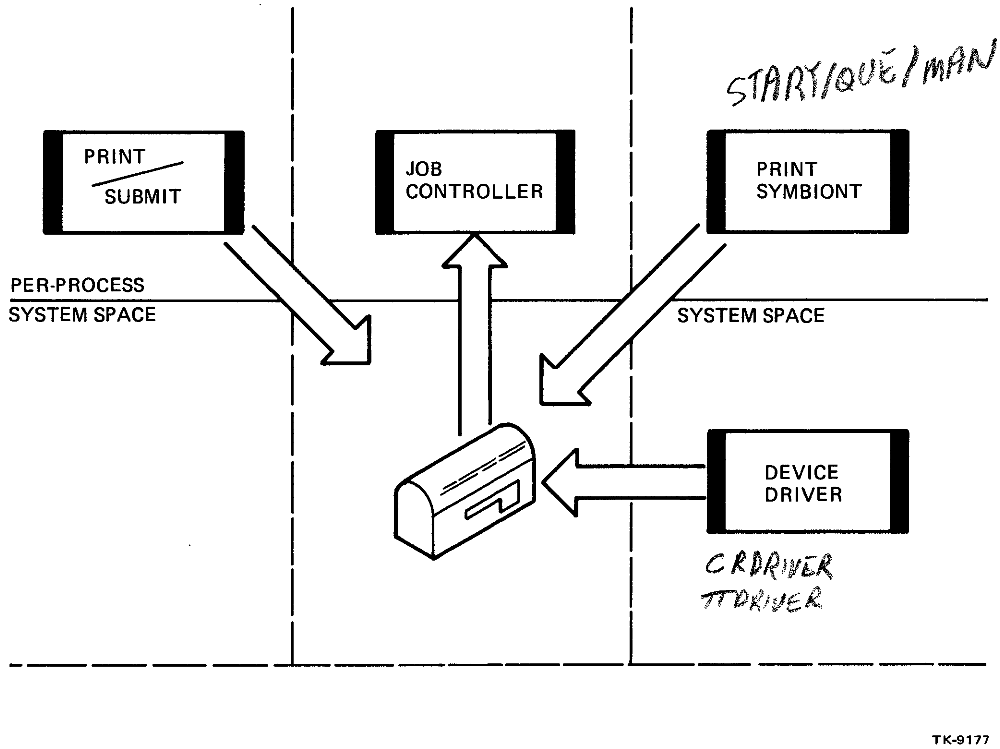


Figure 1 Communication with the Job Controller

- Job Controller is a full process
  - Event-driven
  - Responds to information placed in mailbox
  - Outstanding $QIO on mailbox
- Mailbox communication with
  - User processes
  - Card readers
  - Symbionts

Job Controller Functions

- Interactive and Batch Jobs
  - Creation

    Responds to unsolicited input message
    Process created running LOGINOUT.EXE (for terminals) and INPSMB.EXE (for cardreaders)
  - Activities

    Responds to messages from CLI (for example, PRINT, SUBMIT)
  - Deletion

    Records accounting information
- Symbiont Manager part of JobCtl
  - Creation

    Symbionts created by means of operator action
  - Activities

    Mailbox messages sent to symbiont assign jobs to print; symbionts do not see queue
  - Deletion

    Symbionts deleted by means of operator action
- Accounting Manager
  - Activities

    Interactive or batch job termination
    Print job completion
    Login failure
  - Additional DCL commands ($SET ACCOUNTING) invoke the Accounting Manager

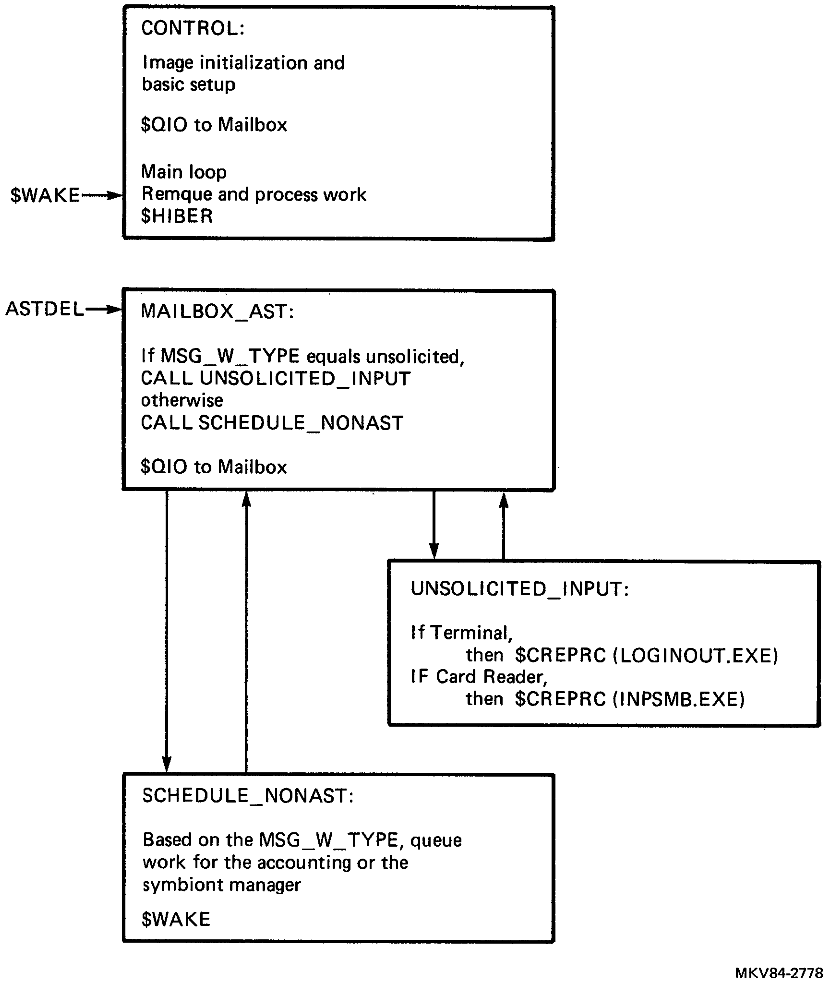


Figure 2 Job Controller Code Flow

- Initialization
- Main Routine Loop
- Mailbox AST
  - If unsolicited TTY or CR, issue $CREPRC
  - Else issue $WAKE

SYMBIONTS

- VAX/VMS transfers data between slow devices and high-speed devices

  Card Reader ---> Disk

  Disk ----> Line Printer

- Controlled by a process called a symbiont.
- The creation, task scheduling, and dismissal of symbionts is controlled by the VMS Job Controller.
- There are three types of symbionts

  1. input symbionts
  2. output symbionts
  3. server symbionts — lost

VMS supplies no server symbionts.

- Print Symbiont facility is bundled with VMS and packaged as a shareable image and an executable symbiont.
- It is designed to allow programmers to implement synchronous single-threaded symbionts using any high-level language that supports the VAX-11 calling standard.
- It also allows asynchronous, multi-threaded symbionts to be implemented.

Output Symbionts

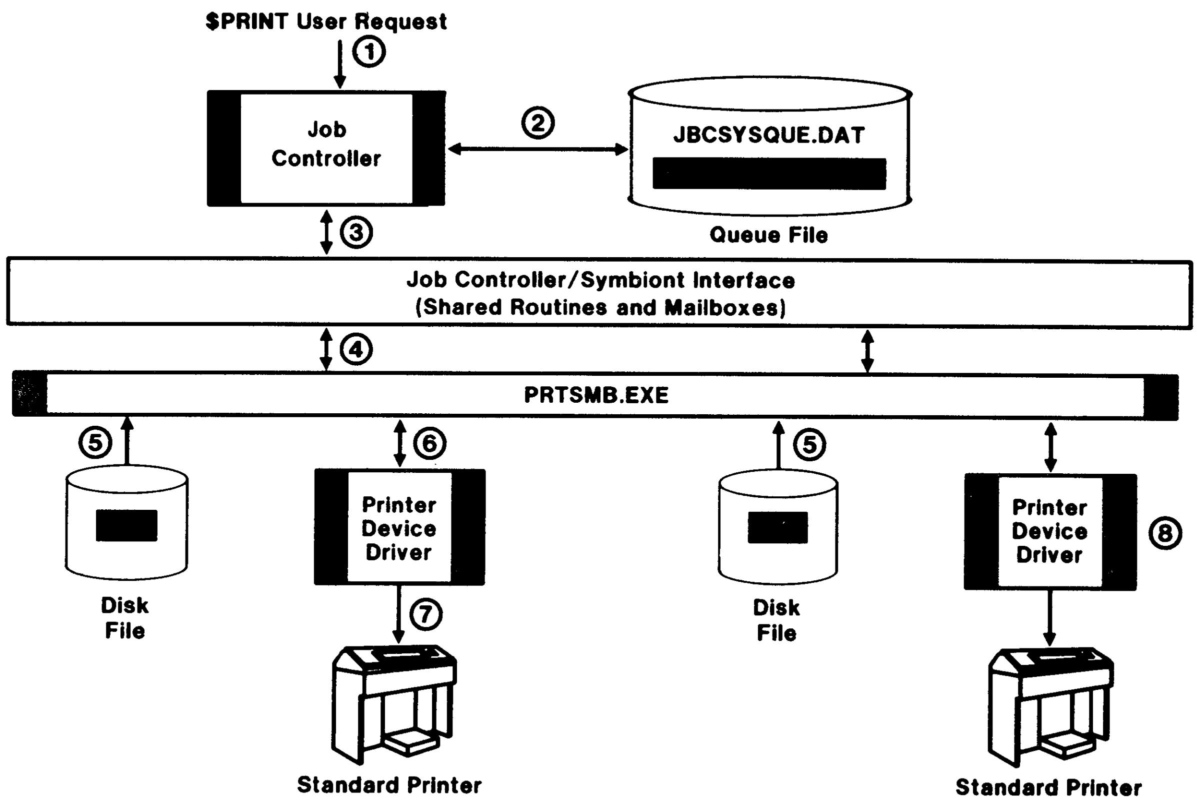


Figure 3 Output Symbiont Flow Diagram

- JOB_CONTROL process receives "PRINT" command
- Task(s) given to symbiont
- Symbiont reports to JOB_CONTROL process when finished

Notes on Figure 3

1. Issue a PRINT request.
2. The Job Controller enters the print request in the appropriate queue and assigns the request a job number.
3. The Job Controller breaks the print job into a number of tasks.
4. PRTSMB inteprets the information it receives from the interface.
5. PRTSMB locates the file it is to print.
6. PRTSMB submits the file to the printer device driver.
7. The file is printed.
8. If written properly, the symbiont can be multi-streamed.

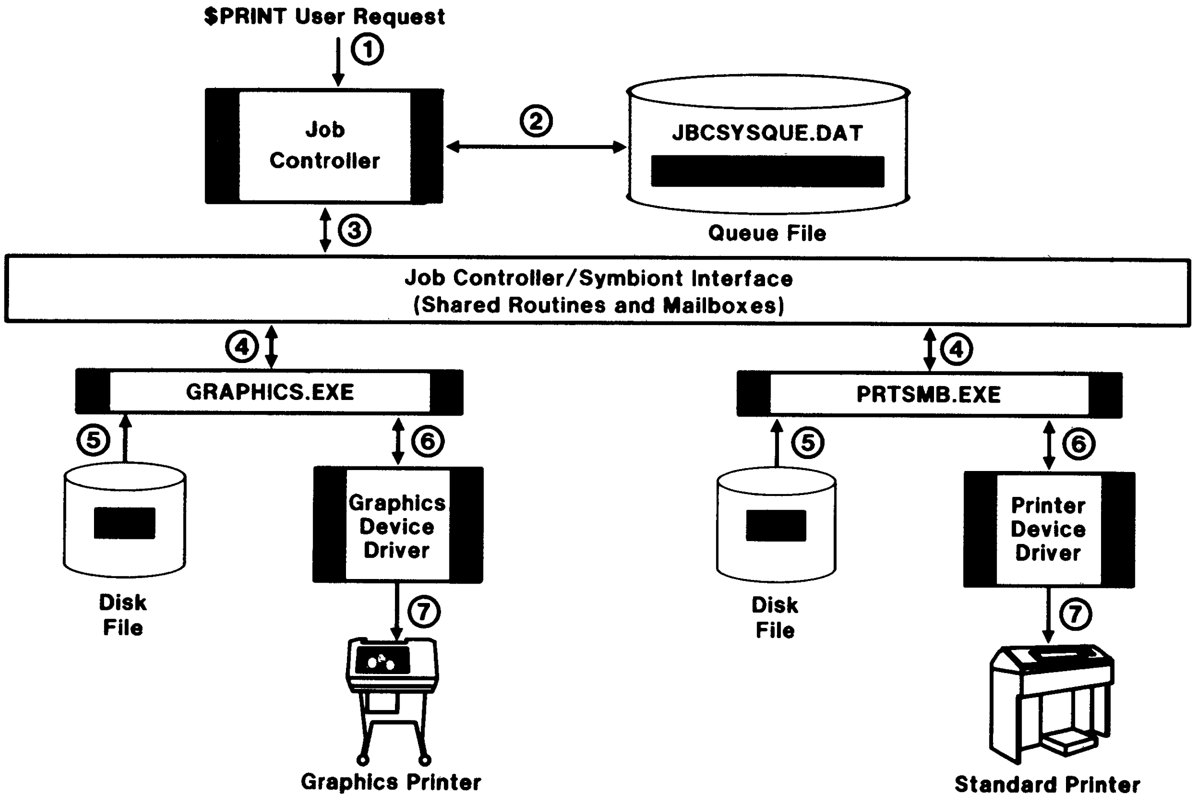


Figure 4 User Symbiont Flow Diagram

- The user symbiont GRAPHICS.EXE is "connected" to the graphics device with the DCL command

  $ INITIALIZE/QUEUE/START/PROCESSOR=GRAPHICS.EXE device

- The GRAPHICS.EXE symbiont is written using VMS-supplied routines.

Symbiont Services

Services supplied by the VMS shared symbiont include:

- A message interface between a symbiont process and its controlling process.
- A set of routines that implement the message interface.
- A set of routines to control and support a multi-streamed, asynchronous symbiont environment.
- A standard print symbiont allowing user-supplied routines for common functions.

User-Supplied Symbionts

- User-supplied output symbionts replace or complement standard symbionts.
- User-supplied input symbionts are not supported.
- There are two ways of creating user symbionts

  1. User-modified symbionts
  2. User-written symbionts

- You choose between user-written symbionts and user-modified symbionts based on how closely the standard symbiont matches your needs.
- Since user-modified symbionts are generally easier to write and debug than user-written symbionts, it is advisable to choose this technique when possible.

THE ERROR LOGGER

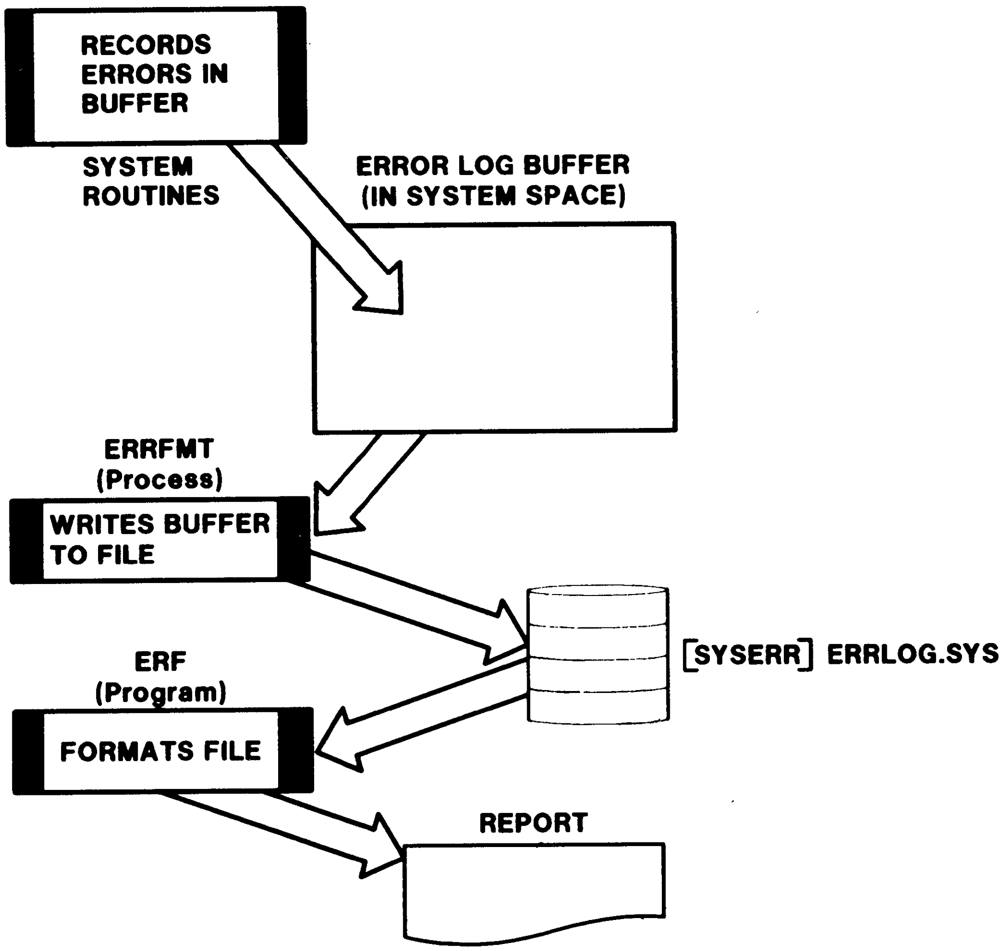


Figure 5 Overview of Error Logging

- Events are reported to VMS
- The information is stored in memory
- The ERRFMT process moves the information to disk when
  - the buffer contains 10 messages
  - the buffer is full
  - 30 seconds has elapsed

Error Logging

- System-wide buffers store the logged information

```
BUFF1:      .blkb 512
BUFF2:      .blkb 512

ERL$AL_BUFADDR:
            .long BUFF1
            .long BUFF2
```

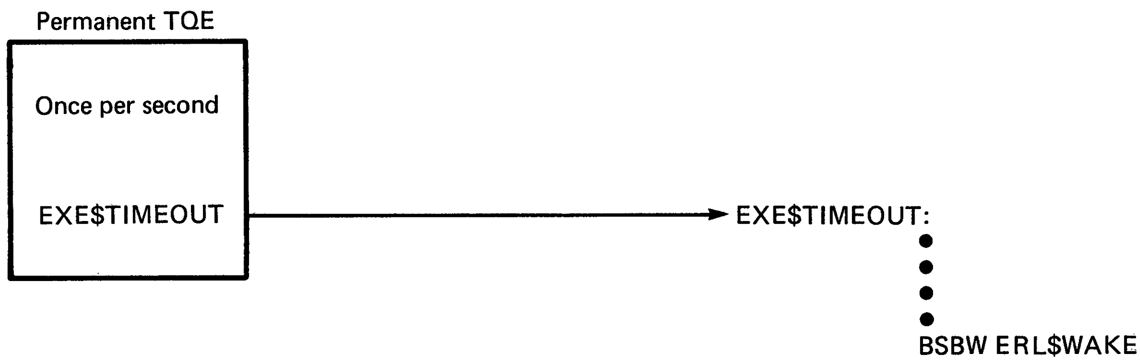
Example 2 Listing Buffer Declarations for Error Logger

- ERRORLOG Portion of the VMS Executive

  VMS has programs called by drivers and other programs to log errors.

DEVICE TIMEOUT	Called by drivers to log a device timeout
DEVICE ERROR	Called by drivers
ERL$WAKE	Called by EXE$TIMEOUT to see if the ERRFMT process should be awakened
ERL$ALLOCEMB	Called by programs to allocate a portion of the message buffer
ERL$RELEASEMB	Called by programs to release the message buffer

Waking the Error Logger




```
ERL$WAKE:
        DECB    ERL$GB_BUFTIM                   ; DECREMENT 30 SECOND TIMER
        BNEQ    10$                             ;
        BICB    #ERL$M_TIMER,ERL$GB_BUFFLAG;CLEAR TIMER ACTIVE FLAG
        MOVB    #MAXTIM,ERL$GB_BUFTIM           ;RESET TIMER VALUE
        MOVL    ERL$GL_ERLPID,R1                ;GET ERROR LOG PROCESS ID
        BSBW    SCH$WAKE                        ;WAKE ERROR LOG PROCESS
        RSB                                     ;
```

MKV84-2706

Figure 6 Code Flow for Error Logger Wake

- EXE$TIMEOUT runs once every second
- ERL$WAKE is called by EXE$TIMEOUT
- If necessary, ERRFMT is awakened to flush the buffer(s)

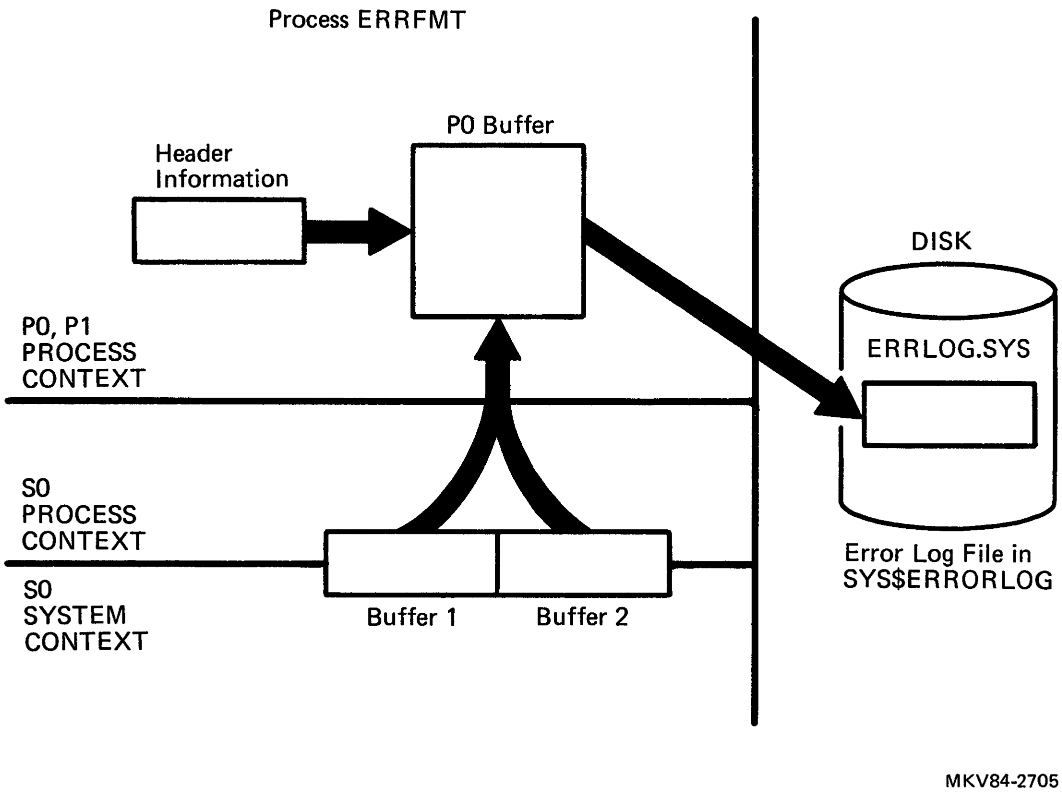


Figure 7 Code Flow for Error Logger

- ERRFMT will transfer the information from the correct S0 buffer to its own P0 buffer space
- Add information to the messages
- If the file is open, use it
- If the file is not open, open it and use it
- If not available, open a new version
- Send the information to the ERRLOG.SYS file

OPERATOR COMMUNICATION

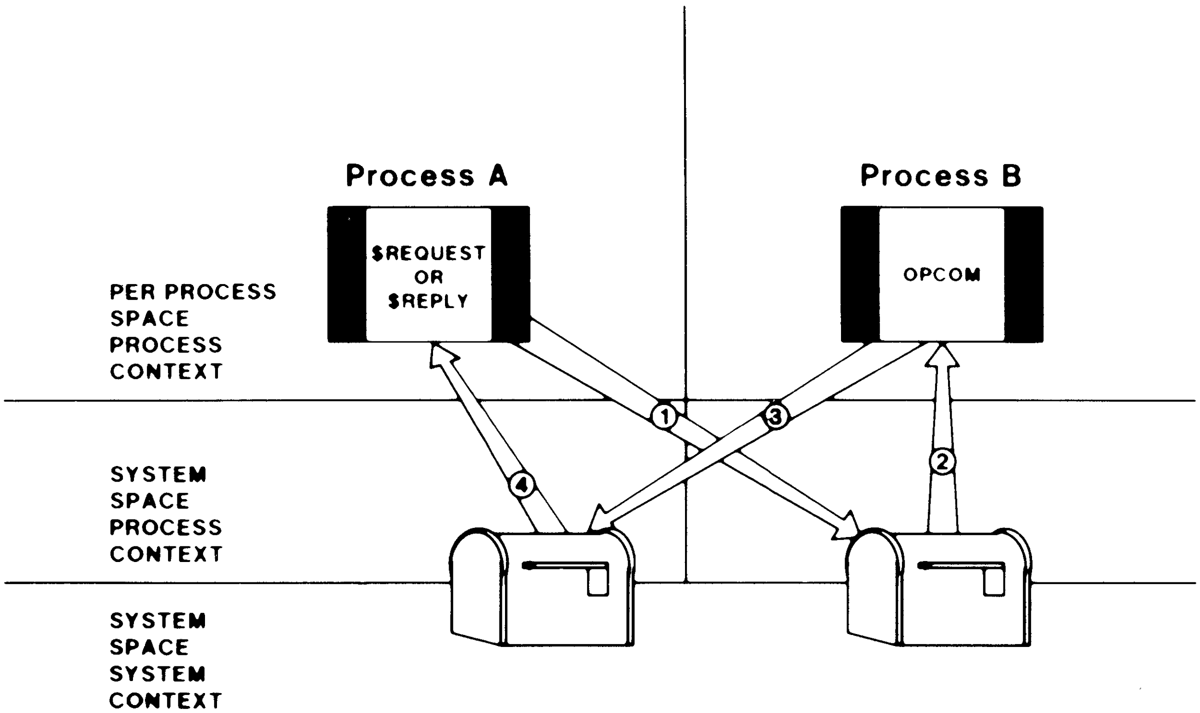


Figure 8 Overview of OPCOM

- Event-driven
- Communication with user processes

```
!Necessary Initialization
OPCOM_INIT ()

!Enter main loop
!Issue time stamp if pending
TIME_STAMP ()

!Issue $QIOW to mailbox
STATUS = $QIOW(............)

!CASE on message type
CASE  RQSTCODE

!End loop
```

Example 3 OPCOM Main Code

- Process start-up goes through initialization
- $QIOW to OPCOM's mailbox
- Case/Select to routine based on message type
- OPCOM Functions
  - Enable/Disable operator terminals
  - Handle user requests and operator replies
  - Device messages
  - Security messages

SUMMARY

- Overview of System Processes
- Job Controller
  - Communication by means of mailboxes
  - Primary functions

    Creation of interactive and batch jobs
    Symbiont manager
    Accounting manager
- Symbionts
  - Workhorses for the job controller
  - Uses:

    Input
    Output
    User-selectable (user symbionts)
- Error Logger
  - Format and store error information
  - Interacts with other VMS components
- OPCOM
  - Communication between users, operators, and the system

Forming, Activating and Terminating Images

INTRODUCTION

An image consists of procedures and data bound together by the linker. Each image executes within the context of a process, and performs various operations for a user.

This module discusses how images are formed on VMS systems. Understanding how images are built can help you create images that execute more efficiently.

The steps in image activation and termination and the related data structures are also discussed. If an image is frequently used, and the speed of its activation is important, the INSTALL utility can be used to partially activate the image in advance.

OBJECTIVES

To write programs that execute more efficiently, the student must understand:

1. How an executable image is formed from source code, especially the structures that are built by the linker.

2. How an image is mapped into the virtual address space of a process, and how it is invoked.

3. The steps in image termination.

RESOURCES

Reading

1. VAX/VMS Internals and Data Structures, chapter on image activation and termination.

2. VAX/VMS Linker Reference Manual, chapters on linker operations and shareable images.

Source Modules

Facility Name	Module Name
DCL	HANDLE IMAGECTRL, IMAGEXECT COMMAND
INSTAL	INSMAIN INSCREATE and others
SYS	SYSIMGACT SYSIMGSTA SYSEXIT SYSRUNDWN

FORMING, ACTIVATING, AND TERMINATING IMAGES

TOPICS

I. Forming an Image

   A. PSECTs in source and object modules

   B. Format and use of the image header

II. Image Activation and Start-Up

   A. Mapping virtual address space

   B. Overview of related data structures

   C. Image start-up (SYS$IMGSTA)

   D. Installing known files

III. Image Exit and Rundown

   A. $EXIT system service

   B. Termination handlers

   C. DCL sequence

IV. SYSGEN Parameters Relating to Image Formation, Activation, and Termination

FORMING AN IMAGE

Program Sections

- Object code is organized into program sections (PSECTs)
  - By VAX-11 MACRO assembler
  - By high-level language compilers
  - Depending on properties of the code, or explicit PSECT directives
- PSECT attributes are assigned by
  - MACRO programmers
  - Some defaults applied by the MACRO assembler
  - High-level language compilers

Table 1 PSECT Attributes

Mnemonic	Attribute	Mnemonic	Attribute
WRT	Writable	NOWRT	Not Writable
RD	Readable	NORD	Not Readable
EXE	Executable	NOEXE	Not Executable
PIC	Position-Independent	NOPIC	Not Position-Independent
LCL	Local	GBL	Global
CON	Concatenated	OVR	Overlaid
SHR	Potentially Shareable	NOSHR	Not Shareable
VEC	Protected (vector)	NOVEC	Nonprotected

Format of an Image File

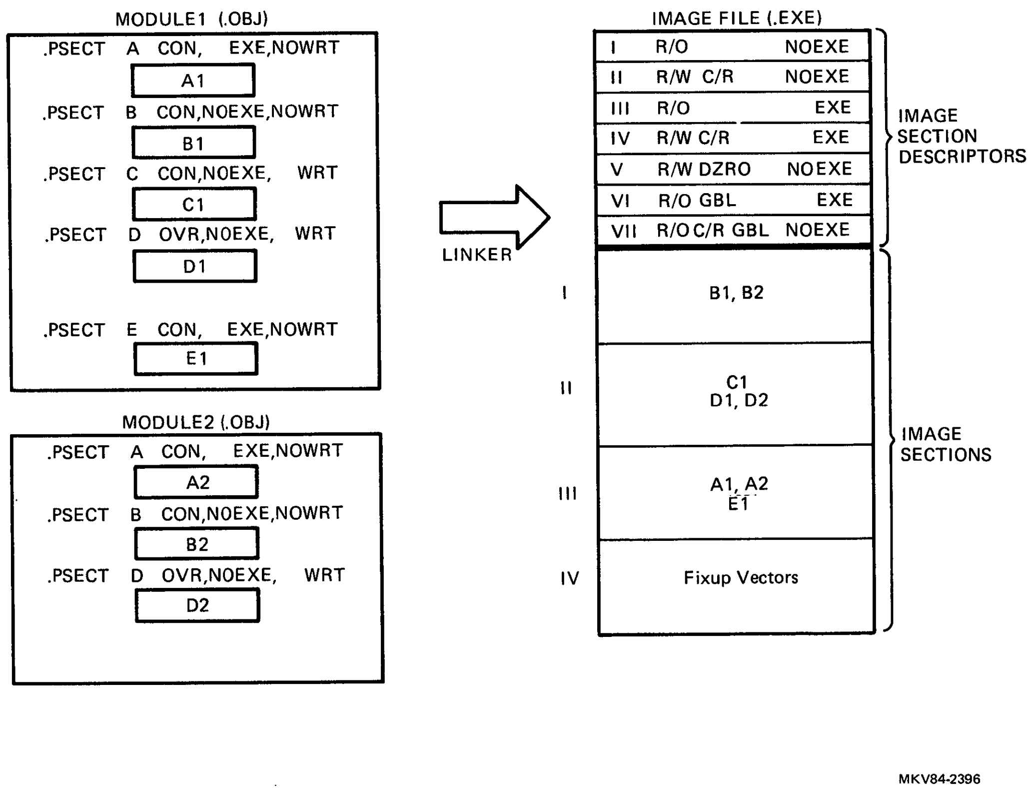


Figure 1 Creating an Image File

- Image sections stored in the image file

  I. Read-only data
  II. Read/write data (copy-on-reference)
  III. Executable code
  IV. Fixup vectors

- User stack is demand zero (image section V)

- Additional image sections because LIBRTL shareable image was referenced

  VI. Transfer vectors and code
  VII. Private impure data (copy-on-reference)

Image Section Descriptor Formats

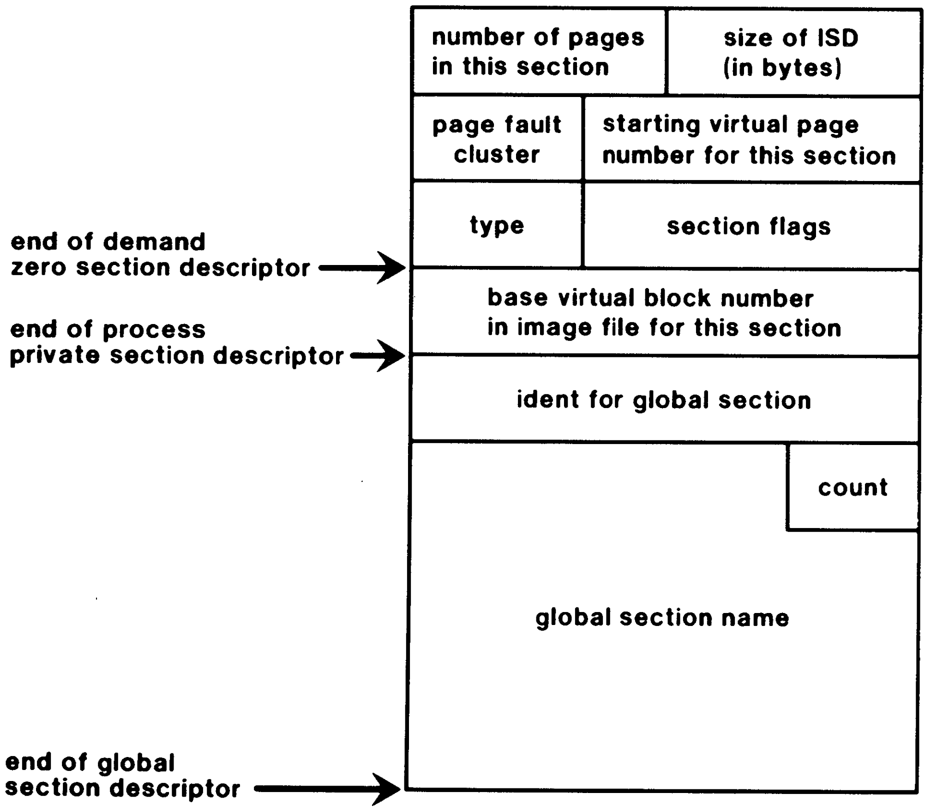


Figure 2 Image Section Descriptor Formats

- Image section descriptors (ISDs) are built by the linker
- One ISD for each image section
- Three kinds of ISDs
- Stored in the image header
- Common TYPE field values are ISDK_NORMAL, ISDK_USRSTACK and ISD$K_SHRPIC

*SYSGEN -

PFCDEFAULT

Format of the Image Header

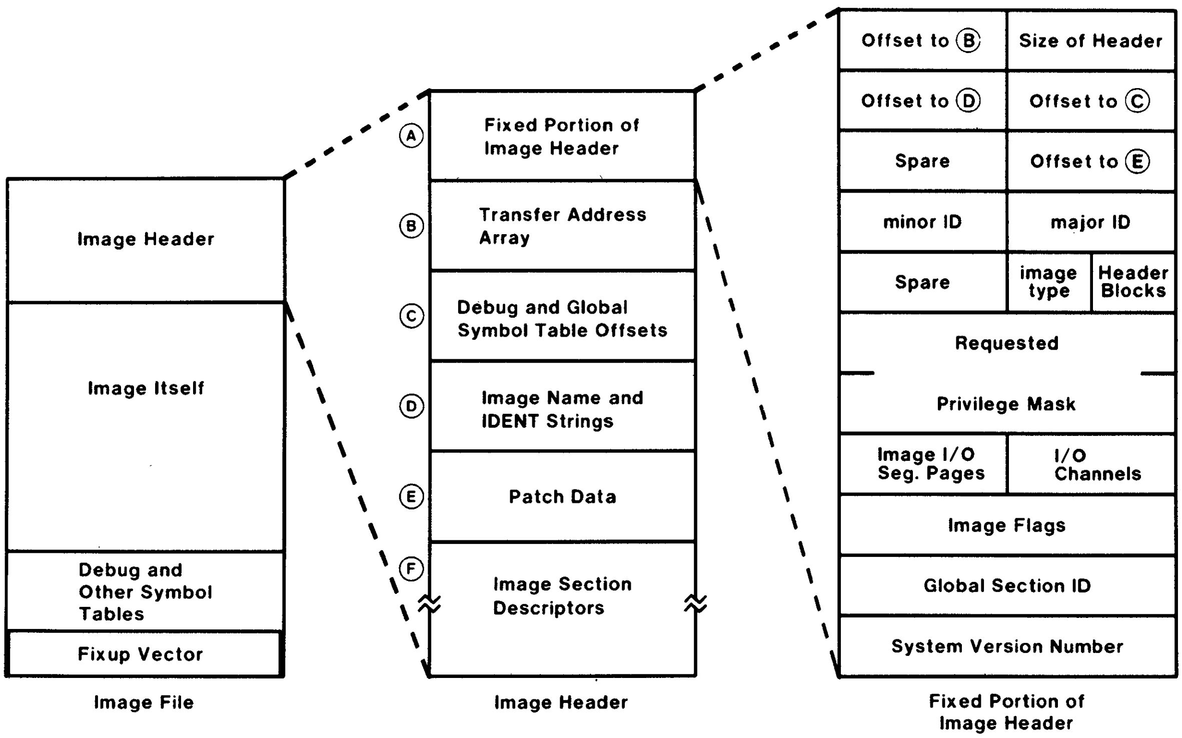


Figure 3 The Image Header

- Image header is at beginning of .EXE file (usually 1 block)
- Contains a description of the image
- Information is used when activating the image

IMAGE ACTIVATION AND START-UP

1. DCL RUN command issued
   - Calls the image activator (SYS$IMGACT)
   - Calls the image
2. Image activator
   - Executive mode system service
   - Opens image file
   - Reads image header
   - Maps image to virtual address space
   - Returns to caller (DCL in this case)
3. Pages of image are brought into physical memory by the VMS pager, as needed

*SYSGEN -

IMGIOCNT

Mapping an Image to Virtual Address Space

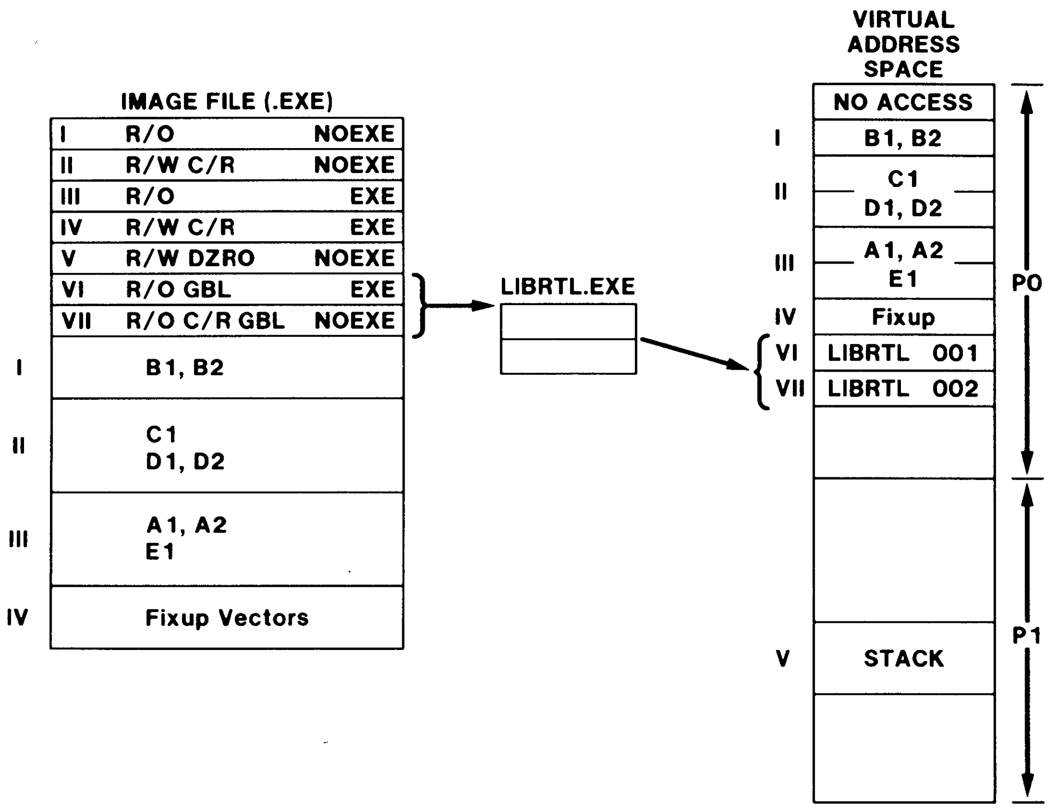


Figure 4 Mapping to Virtual Address Space

- Image activator maps image to process virtual address space
- Code for any shareable images is located
- DCL calls entry point of image
- Program references virtual addresses
- Pages of code brought into physical memory by the pager

Bringing Pages of Image into Physical Memory

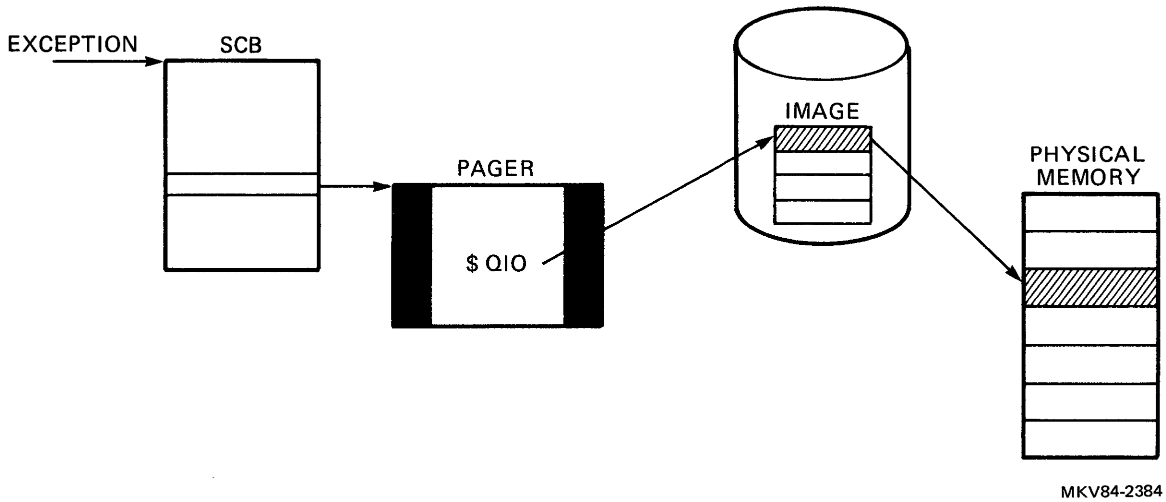


Figure 5 Bringing Pages into Physical Memory

- Referencing image page not in physical memory generates an exception
- Hardware locates address of pager routine via the SCB
- Pager brings pages of image into physical memory
- Image instructions now execute
- Image references virtual addresses

Translating Virtual to Physical Addresses

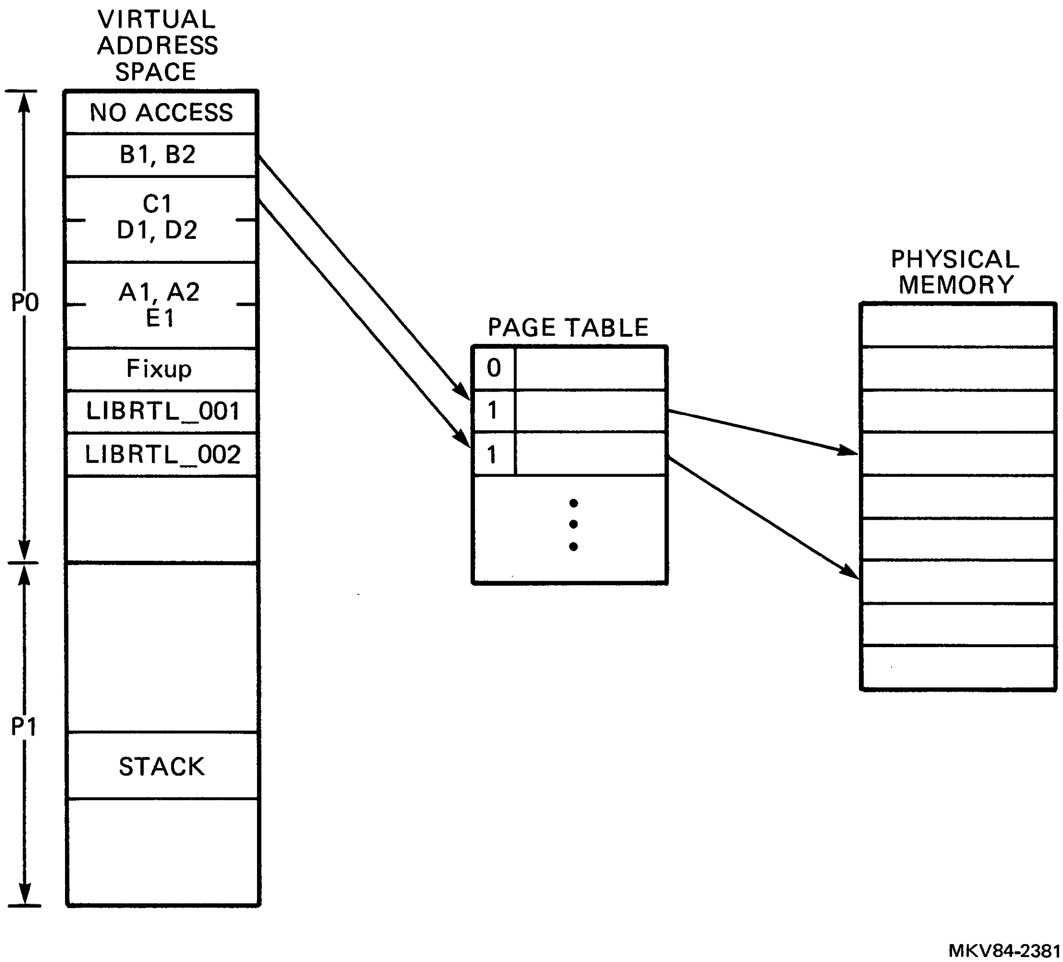


Figure 6 Translating Virtual to Physical Addresses

- Hardware uses page tables to translate virtual addresses
- Page tables are filled in by the image activator
- One page table entry maps one virtual page to one physical page
- High bit of PTE determines whether page is in memory

Locating Image Pages on Disk

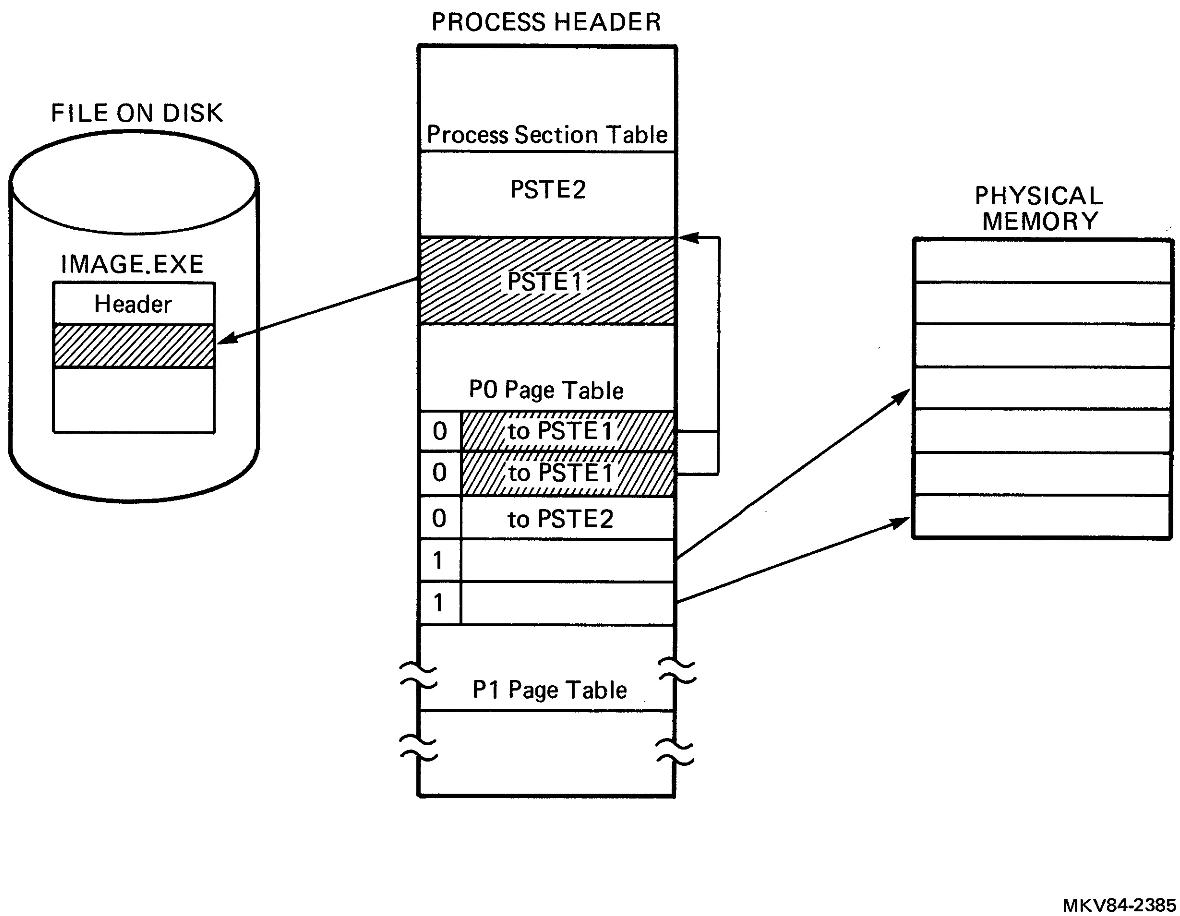


Figure 7 Locating Image Pages on Disk

- Process Section Table (PST) locates image sections on disk
- PST entries are built by the image activator
- Most PST entry information is copied from ISDs

*SYSGEN -

PROCSECTCNT

Summary of Image Formation and Activation

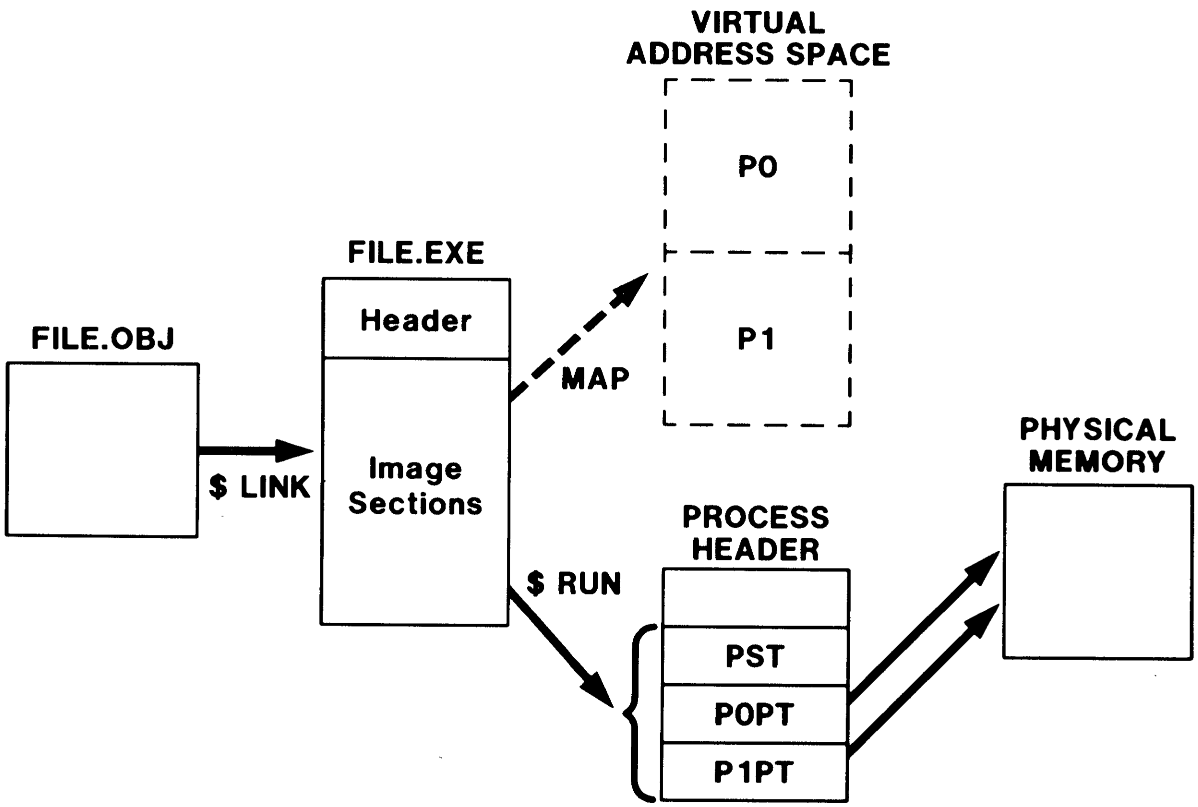


Figure 8 Summary of Image Formation and Activation

Image Start-Up

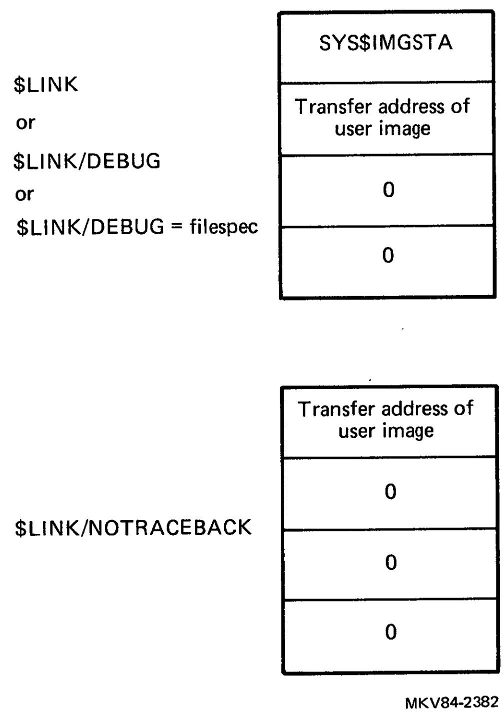


Figure 9 Transfer Address Array Formats

- SYS$IMGSTA system service
  - Map the debugger, if referenced
  - Establish traceback handler
  - Alter argument list to point to next transfer vector address
- LIB$INITIALIZE
- Transfer address obtained from image header

Installing Files

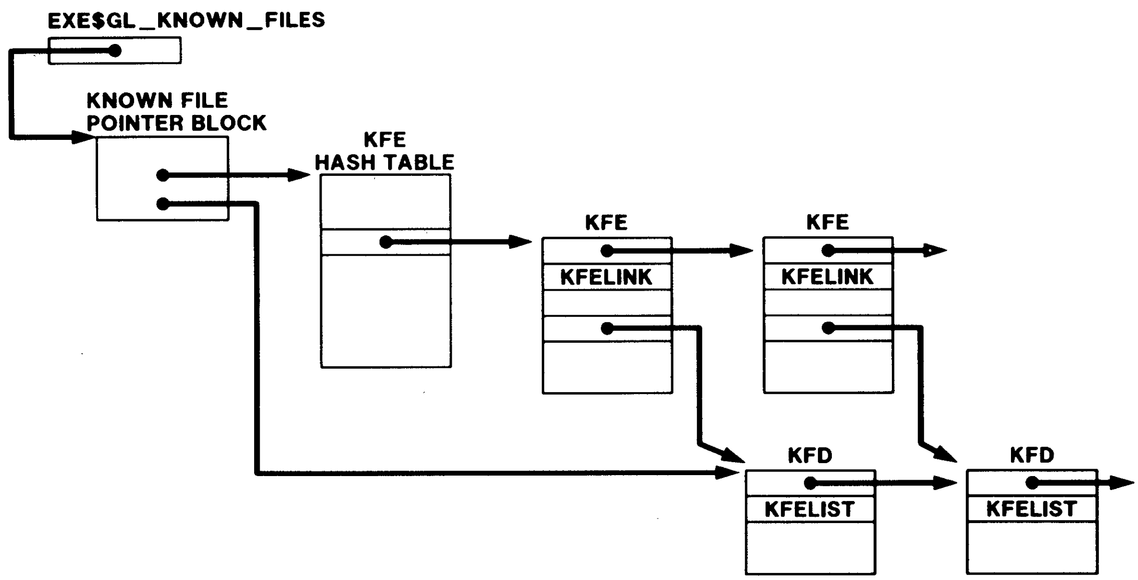


Figure 10 The Known File Database

- Can INSTALL a file with various attributes
- One Known File Entry (KFE) for each file
- One KFD for each unique device, directory, and file-type combination

*SYSGEN -

GBLSECTIONS
GBLPAGES

The Known File Entry

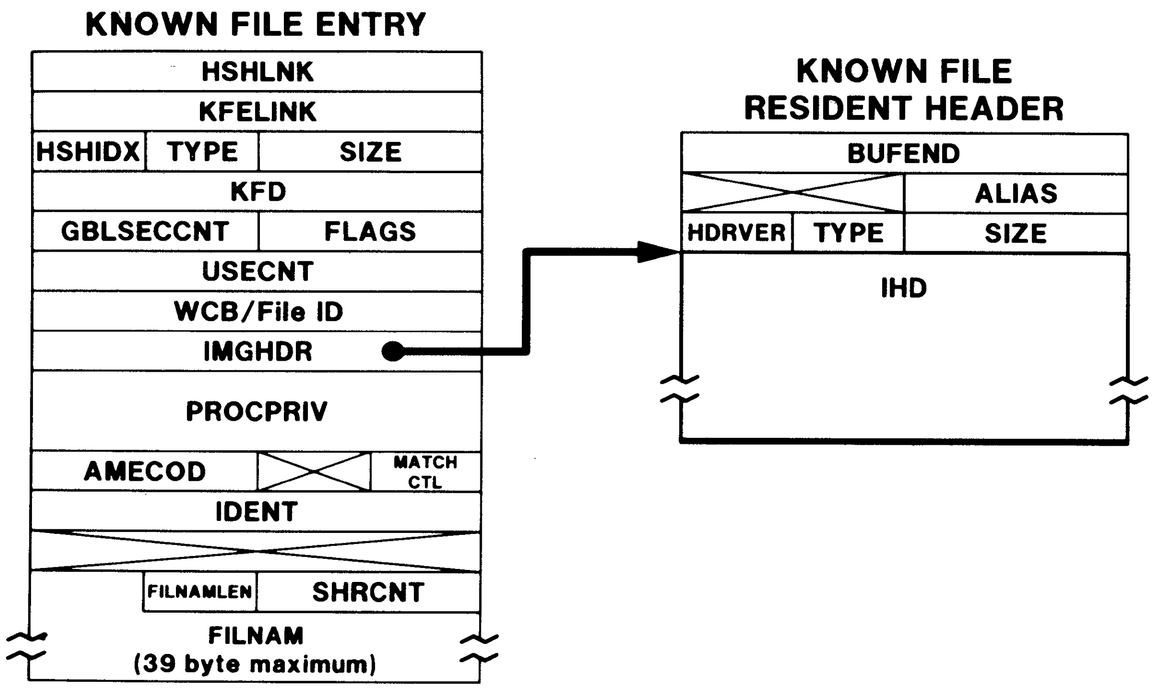


Figure 11 KFE with a Resident Header

- KFE contains information about the installed file's attributes
- If header-resident, KFE points to the image header

IMAGE EXIT AND RUNDOWN

Exit System Service

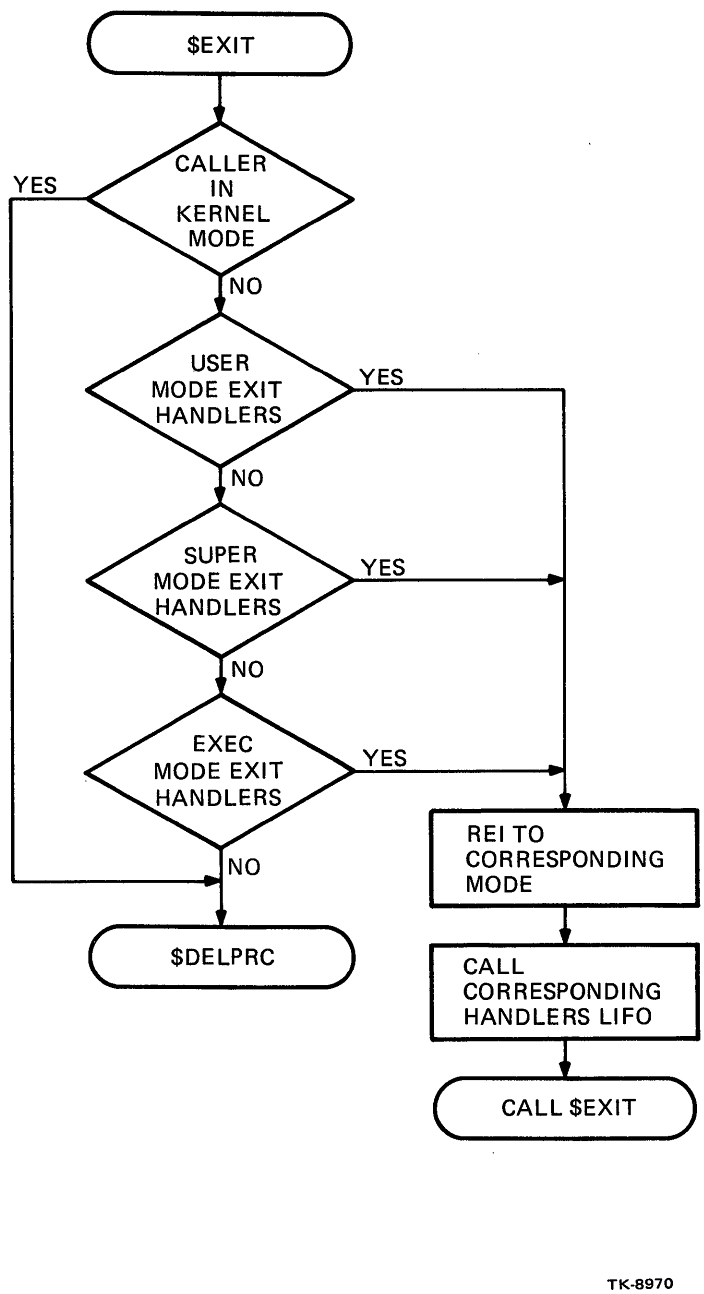


Figure 12 Exit System Service

Termination Handlers

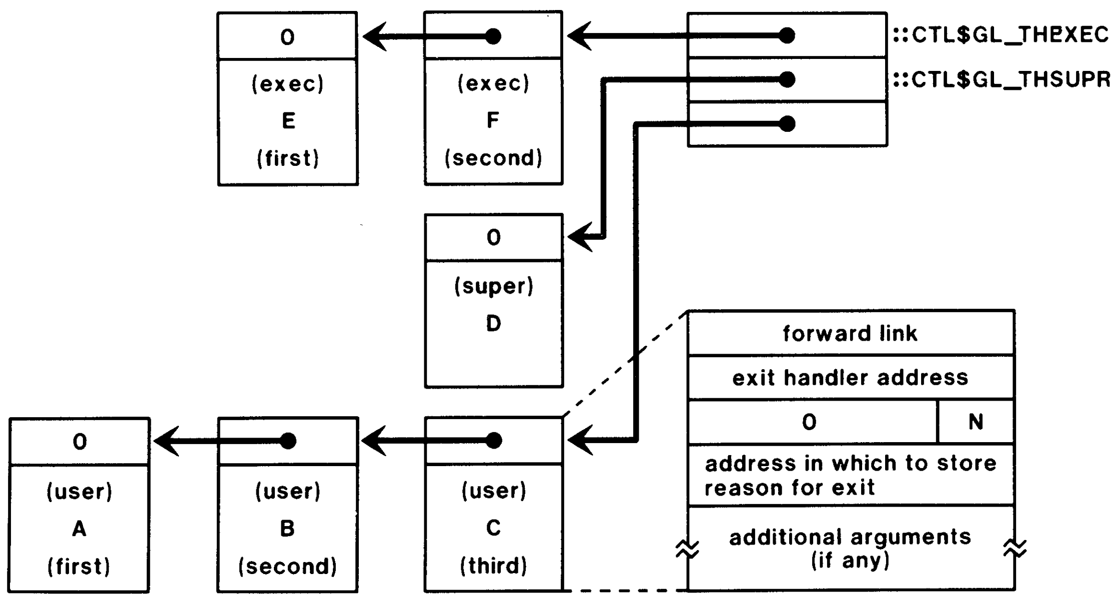


Figure 13 Termination Handlers

Table 2 How Termination Handlers Are Established for Each Access Mode

Mode	Established By
User	User image
Supervisor	DCL (or other CLI)
Executive	PROCSTRT for RMS rundown
Kernel	(No handlers; EXIT causes process deletion)

DCL Operation

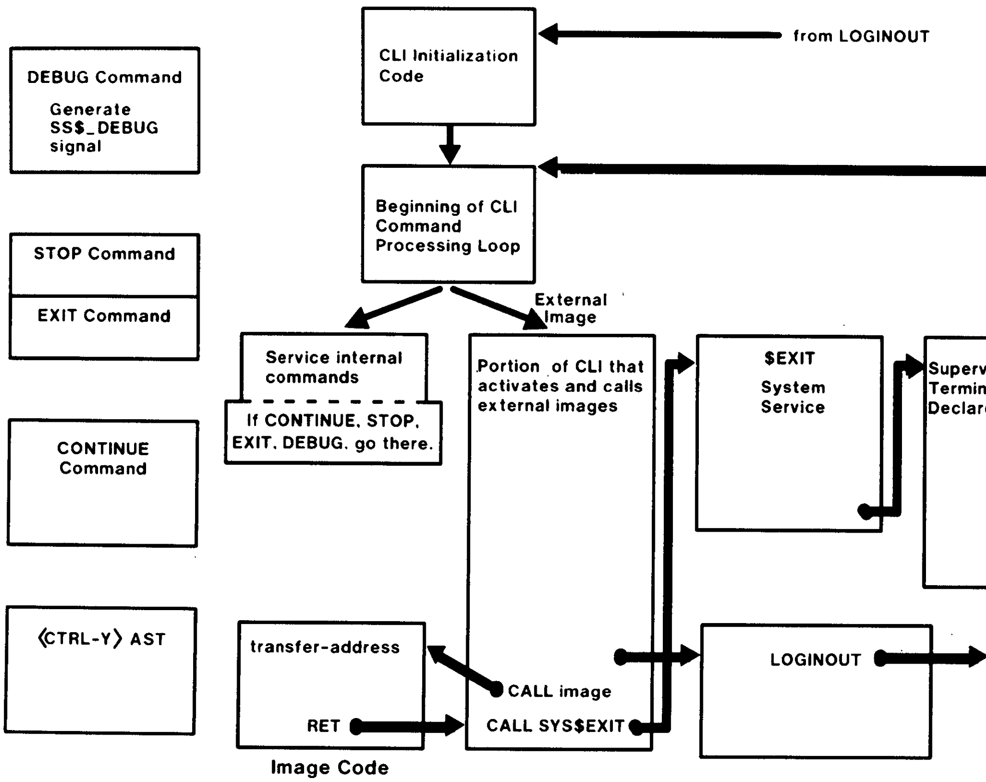


Figure 14 DCL Operation

- Glorified exit handler
- Main command loop
  - Prompts for command
  - Uses DCL tables to decide image or internal routine
- Command code
  - Images run in P0 space
  - Internal routines run in P1 space
- CTRL/Y AST
- $EXIT (image)
- LOGOUT (LOGINOUT.EXE)

SUMMARY

Table 3 Summary of Image Formation, Activation, and Termination

Operation	Component
Form image from object modules	Linker
Map image to virtual address space (create page tables, PST)	Image Activator (SYS$IMGACT)
Partially activate known images	INSTALL
Establish default condition handlers	SYS$IMGSTA
Bring image pages into physical memory	Pager
Invoke termination handlers; eventually cause image to be removed	SYS$EXIT

Table 4 SYSGEN Parameters Related to Image Formation, Activation, and Termination

Function	Parameter
Maximum number of global pages (size of Global Page Table)	GBLPAGES
Maximum number of global sections that can be made known to system (size of GST)	GBLSECTIONS
Default page fault cluster factor for images	PFCDEFAULT
Determine size of process section table (PST)	PROCSECTCNT
Default amount of image I/O address space used by the image activator	IMGIOCNT (*)

(*) special parameter

APPENDIX
LINKER CLUSTERS

- All input files (object, library, and shareable images) are organized into clusters.
- By default
  - All object files are put in the default cluster
  - Separate cluster for each shareable image
- Cluster for a shareable image only contains descriptor for the image, not the whole image (conserve disk space).
- Linker writes contents to image file a cluster at a time.
- Programmer can control linker organization of image
  - Use CLUSTER and/or COLLECT option on LINK command
  - Possibly creates images that execute more efficiently
  - Most effective with large programs (larger than process working set)

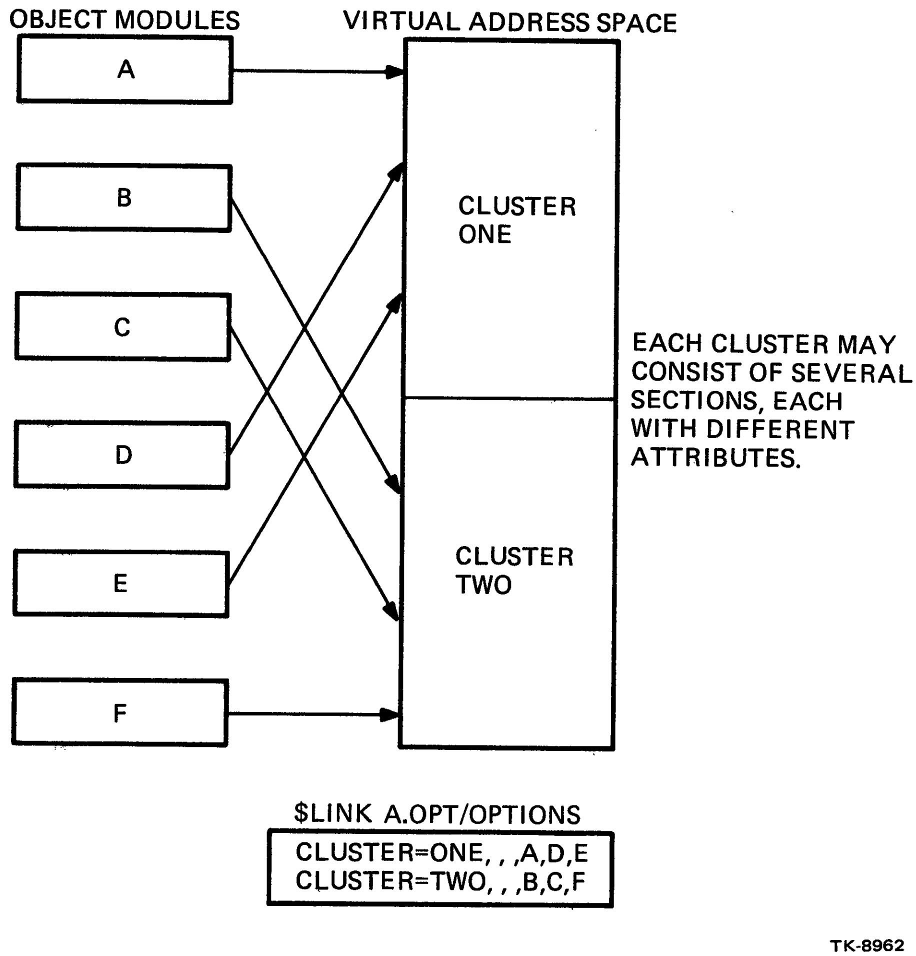


Figure 15 Linker Clusters

Paging

INTRODUCTION

There are two functions required of the memory management subsystem of the operating system. The first gives each user program the impression that it is running in contiguous physical memory, starting at address zero. The second divides the available physical memory equitably among the users of the system.

The first function requires that the user's virtual address be translated to a physical address. If the data is already in memory, the translation is done by hardware. When a program refers to data that is on disk, software is invoked to bring the data into memory. This software is an exception service routine called the pager.

VMS implements the second function by using working sets and paging. Each process is required to execute with a limited amount of its data in memory. To avoid fragmentation of physical memory, this data is divided into 512 byte pieces, called pages. The valid pages a process has in memory at any time are called the working set.

Because the working set limit represents the amount of physical memory "owned" by a process, processes at their working set limits must replace pages in the working set with newly demanded ones (rather than simply acquiring more physical memory). This replacement is performed by the pager.

OBJECTIVES

To understand, and make efficient use of, the Paging system on VMS, the student must be able to:

- Describe the effects of changing working set size, creating and deleting virtual address space, and creating and mapping a global section.
- Discuss the programming considerations that affect paging overhead.
- Given a set of initial conditions and a page request, describe the changes in the status and locations of pages and the changes in process states.
- Discuss the effects of altering SYSGEN parameters governing paging.

RESOURCES

Reading

1. VAX/VMS Internals and Data Structures, chapters on memory management data structures, paging dynamics, and memory management system services.

Additional Suggested Reading

1. VAX/VMS Internals and Data Structures, chapter on image activation and termination.

Source Modules

Facility Name	Module Name
SYS	PAGEFAULT ALLOCPFN SVAPTE
	SYSADJWSL,SYSLKWSET, SYSPURGWS SYSCRMPSC,SYSDGBLSC SYSCREDEL RSE IOCIOPOST
RTL	LIBVM

TOPICS

I. Basic Virtual Addressing

   A. Virtual and physical memory

   B. Page table mapping

II. Overview of Page Fault Handling

   A. Resolving page faults

   B. Data structures in the process header

III. More on Paging

   A. Free and modified page lists

   B. The paging file

   C. Cataloging pageable memory (the PFN database)

IV. Global Paging Data Structures

V. Summary of the Pager

BASIC VIRTUAL ADDRESSING

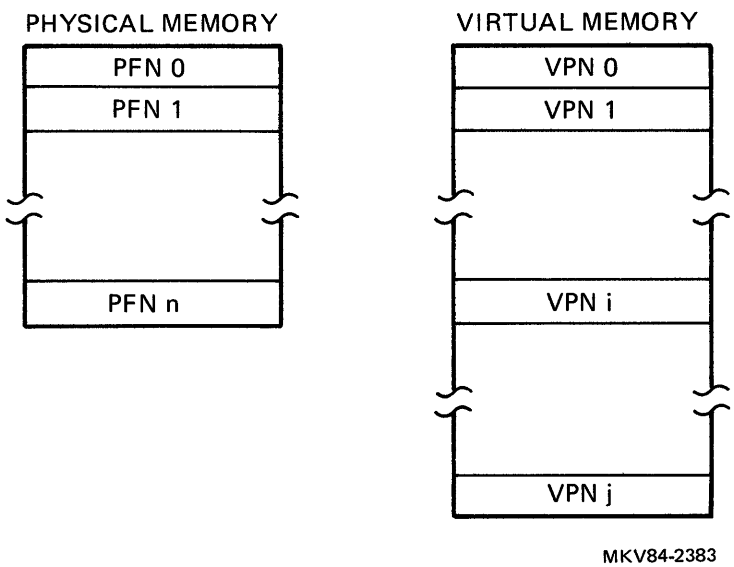


Figure 1 Physical and Virtual Memory

- Physical memory is divided into 512-byte page frames
- Virtual memory is divided into 512-byte pages
- Virtual memory has three areas (P0, P1, S0)

Virtual Address Space

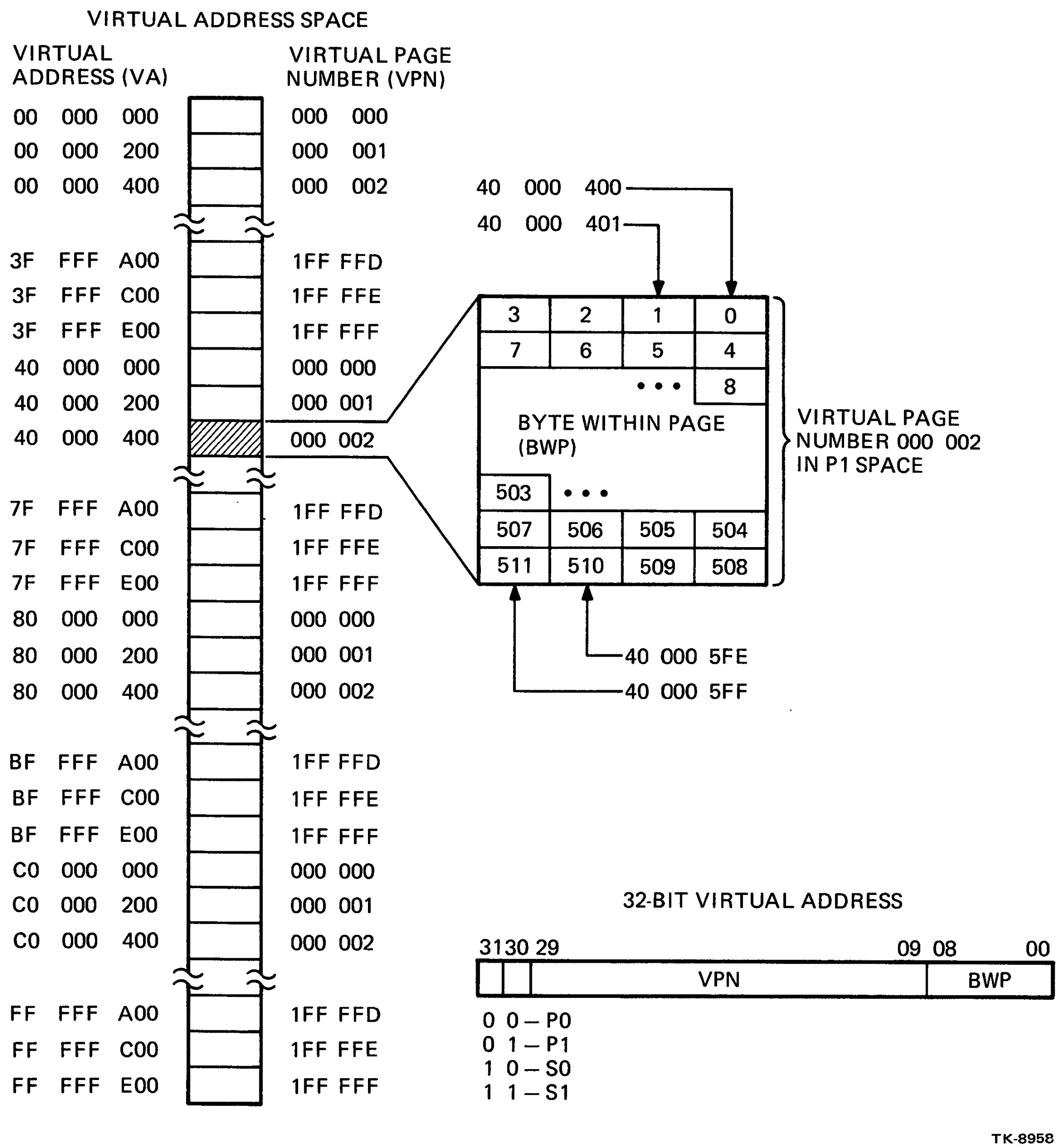


Figure 2 Virtual Address Space

Associating Virtual and Physical Addresses

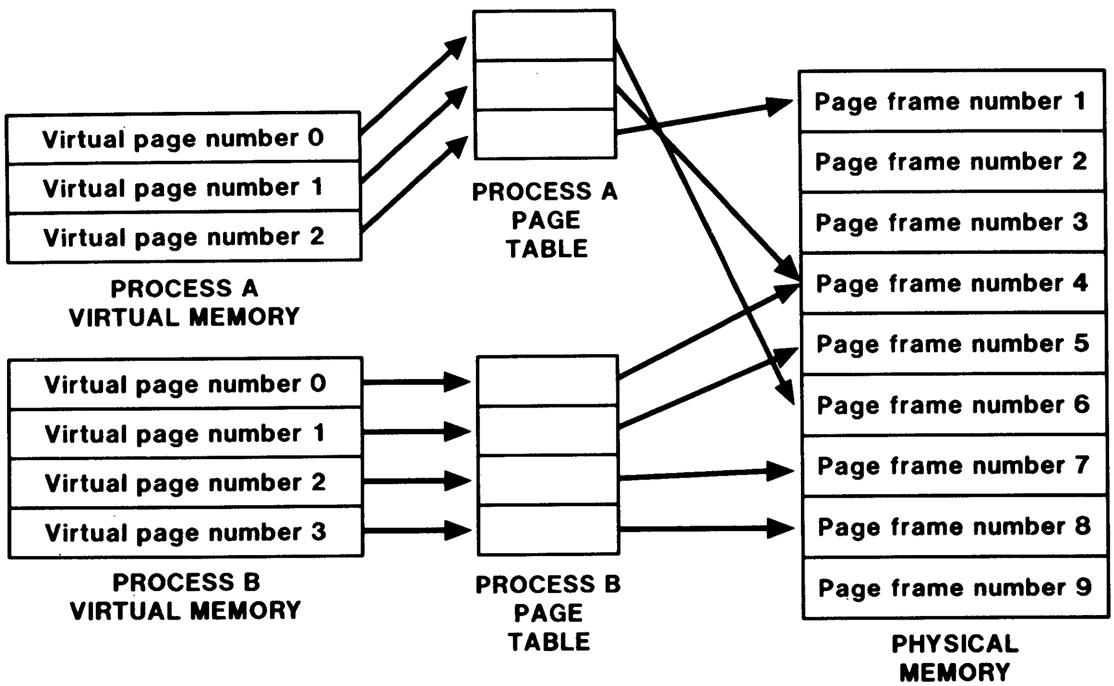


Figure 3 Associating Virtual and Physical Addresses

- Process A has allocated
  PFN 1
  PFN 4
  PFN 6

- Process B has allocated
  PFN 4
  PFN 5
  PFN 7
  PFN 8

- PFN 4 is being shared by Process A and Process B

- Translation from virtual to physical address is done using page tables

S0 Virtual Address Translation

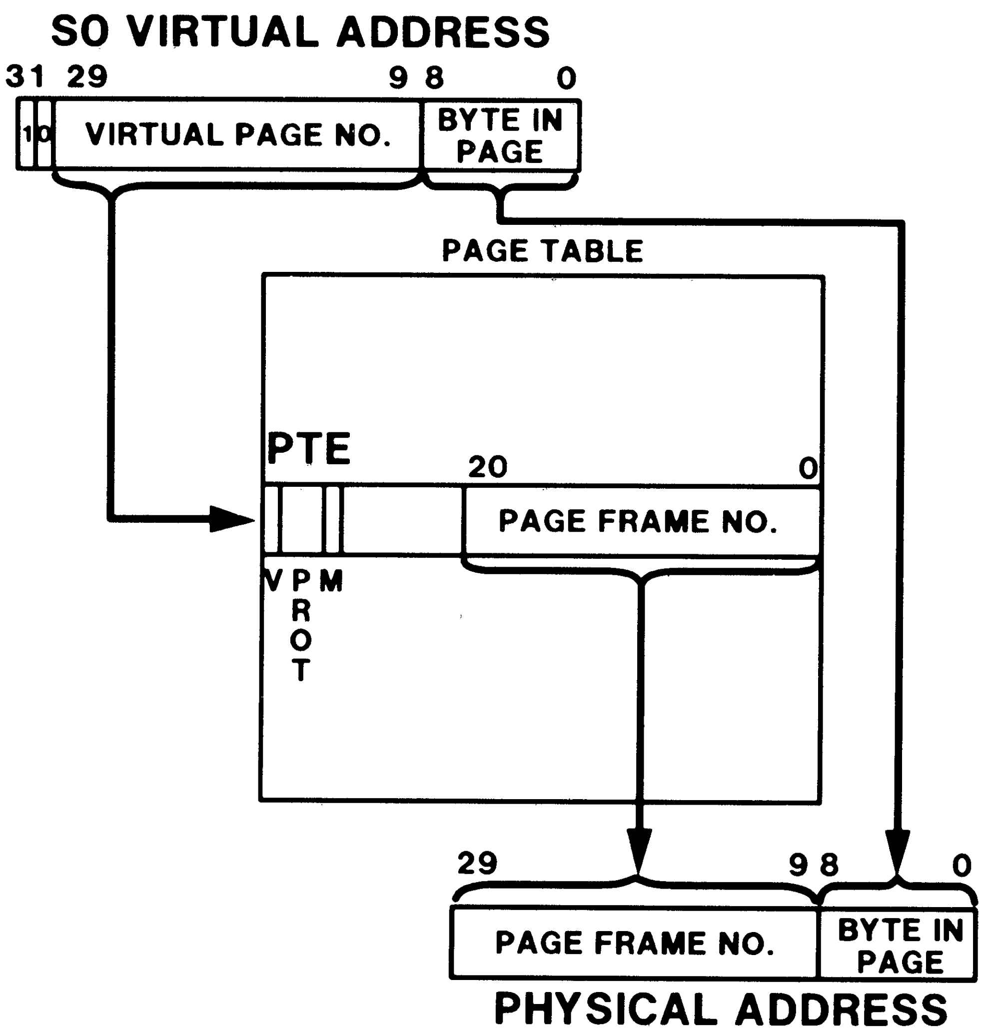


Figure 4 S0 Virtual Address Translation

Hardware Checks

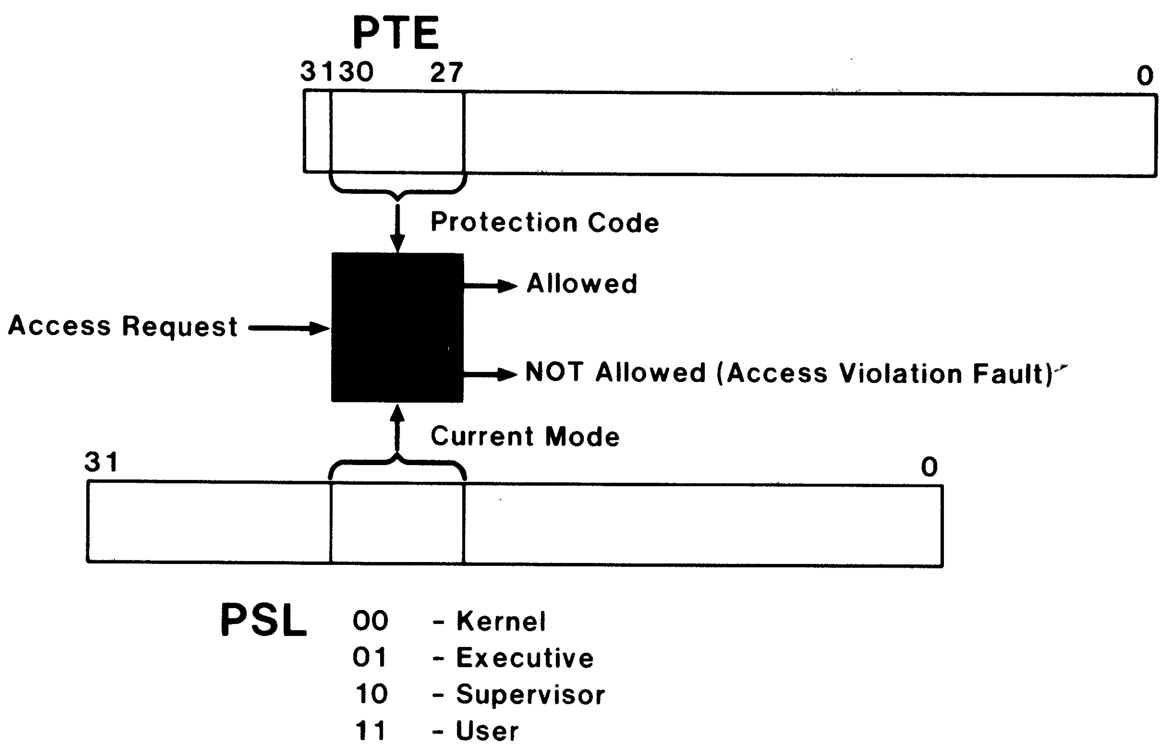


Figure 5 Hardware Checks if Access Allowed

Before address translation occurs, the hardware checks the type of request (read or modify/write) against:

- The protection field of the corresponding page table entry (PTE)
- The current access mode field of the processor status longword (PSL)

If access is denied, no address translation occurs and an access violation condition is signaled.

Process Headers in S0 Space

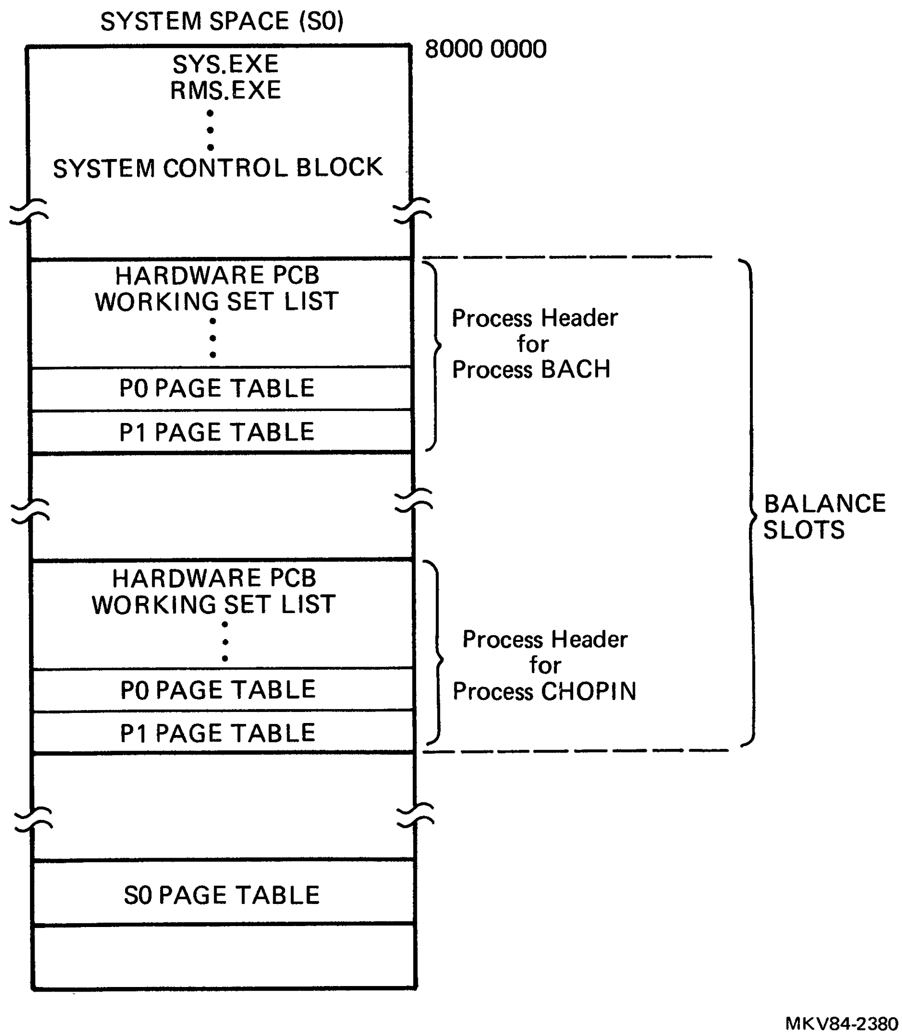


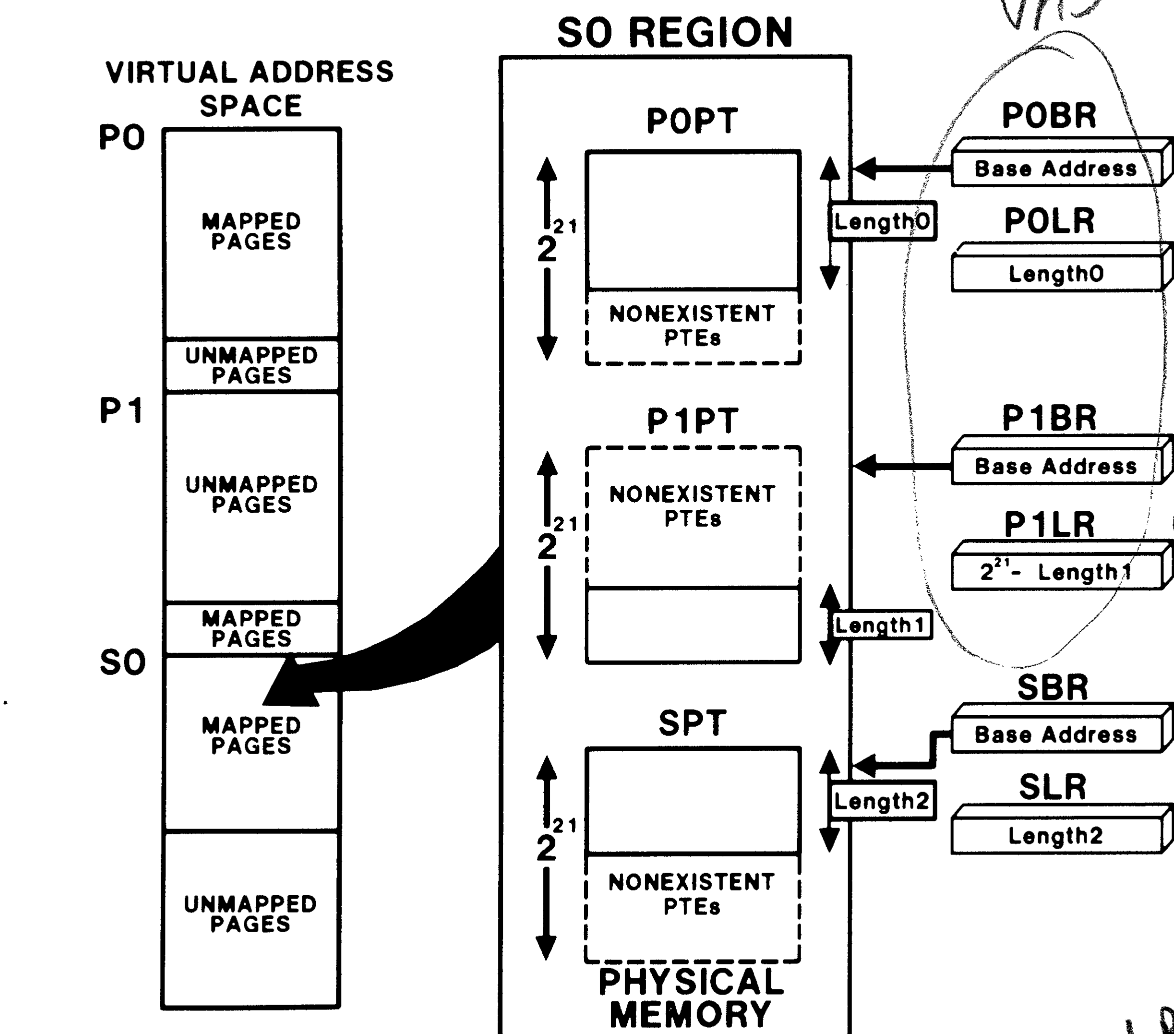
Figure 6 PHDs and S0 Page Table in S0 Space

Page Tables

- All page tables are mapped into system space
- Each page table has
  - Processor base register
  - Processor length register
- System Page Table (SPT) is
  - Permanently resident in memory & is physically contiguous
  - Located by a physical address in System Base Register (SBR)

Page Table Mapping

VA's



{count of P0 map

{count of unused

Figure 7 Page Table Mapping

80000000
7fbe4400
31bbff

example {P1BR = 7FBE440Ø
P1BL = 1FFA4A

PØBR = 803C6EØØ
PØBL = 5B6

invalid PTE's have a "systemtable index" for PSTE in P1 space

Referencing a P0 Virtual Address

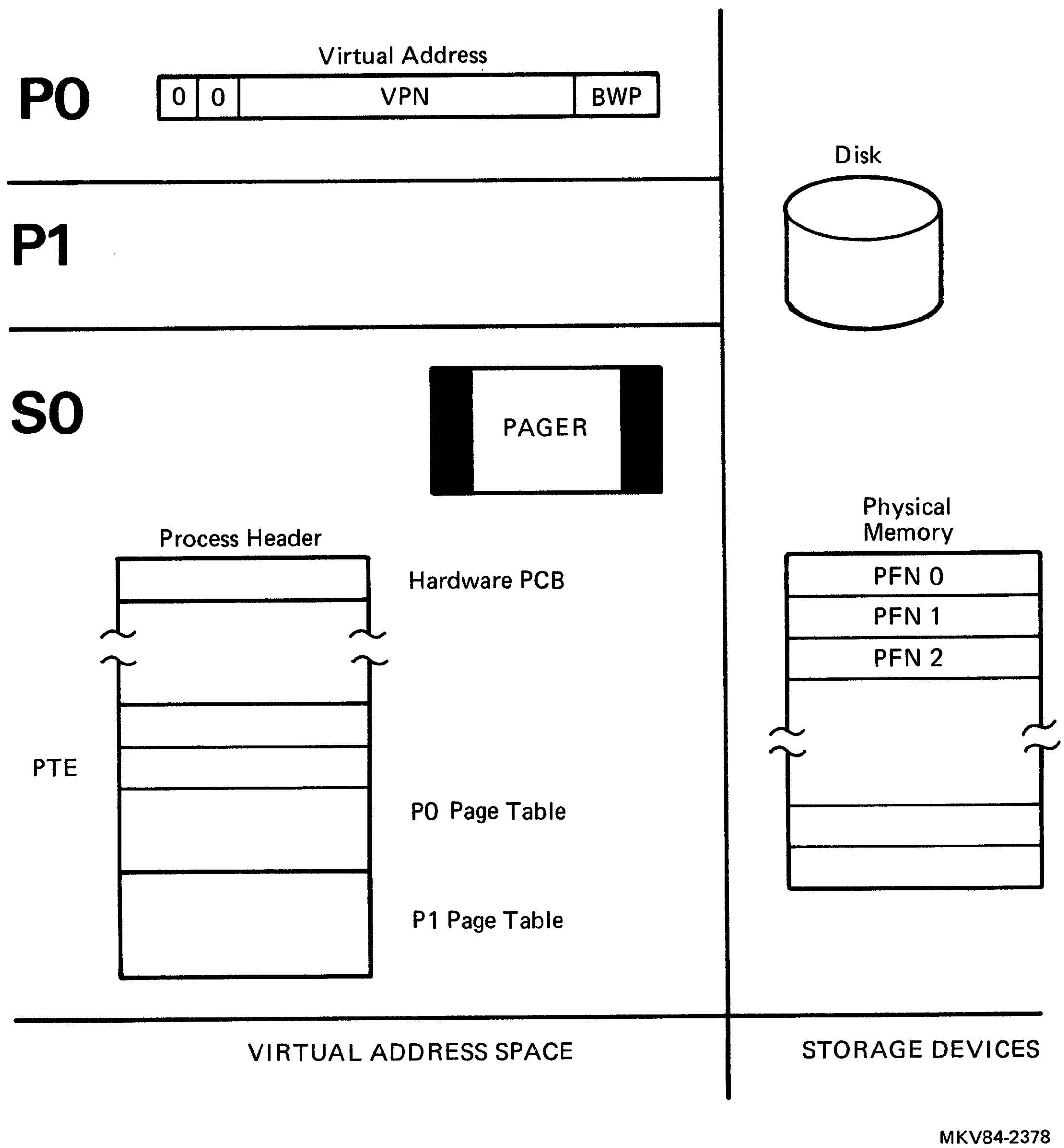


Figure 8 Referencing a P0 Virtual Address

OVERVIEW OF PAGE FAULT HANDLING

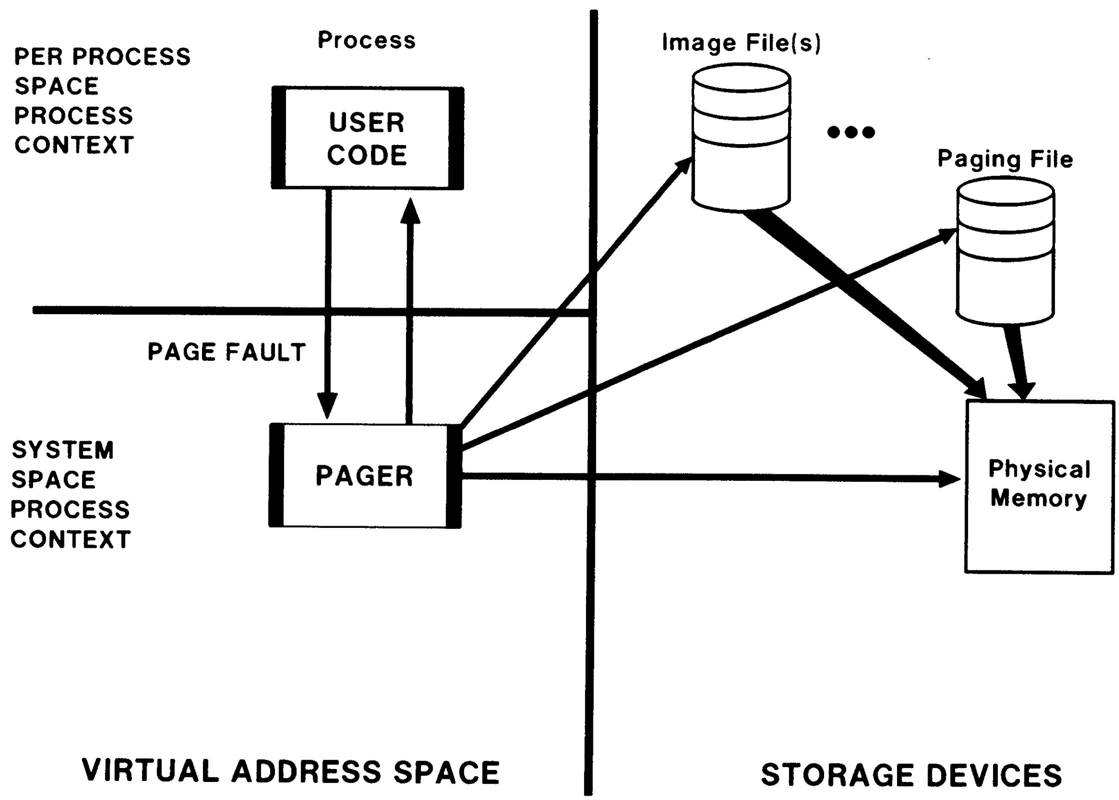


Figure 9 Resolving Page Faults

- Pager is an exception service routine executing within the context of the process that incurred the page fault
- Page not in memory - read I/O issued to image file or page file
- Page in memory - taken from free or modified page list, or valid global page

Working Set List

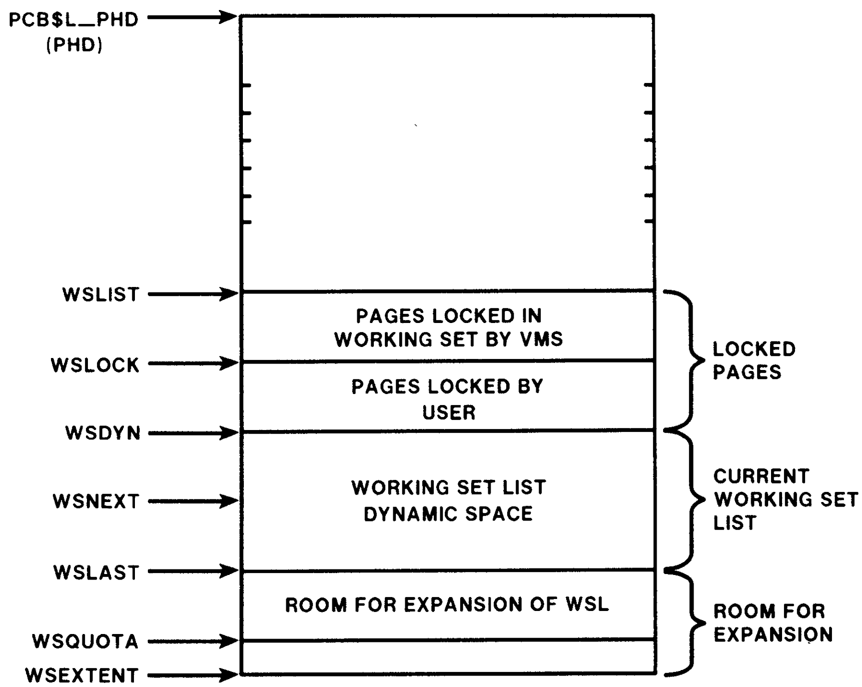


Figure 10 Working Set List

- WSLAST can move to
  - WSQUOTA if few free pages (free page count <BORROWLIM)
  - WSEXTENT if many free pages (free page count >BORROWLIM)
- WSNEXT - latest entry put in working set list
- Page replacement scheme is close to first-in/first-out

*SYSGEN -

BORROWLIM
WSMAX
PQL_DWSDEFAULT, PQL_MWSDEFAULT
PQL_DWSEXTENT, PQL_MWSEXTENT
PQL_DWSQUOTA, PQL_MWSQUOTA
MINWSCNT

Image Section Descriptor Formats

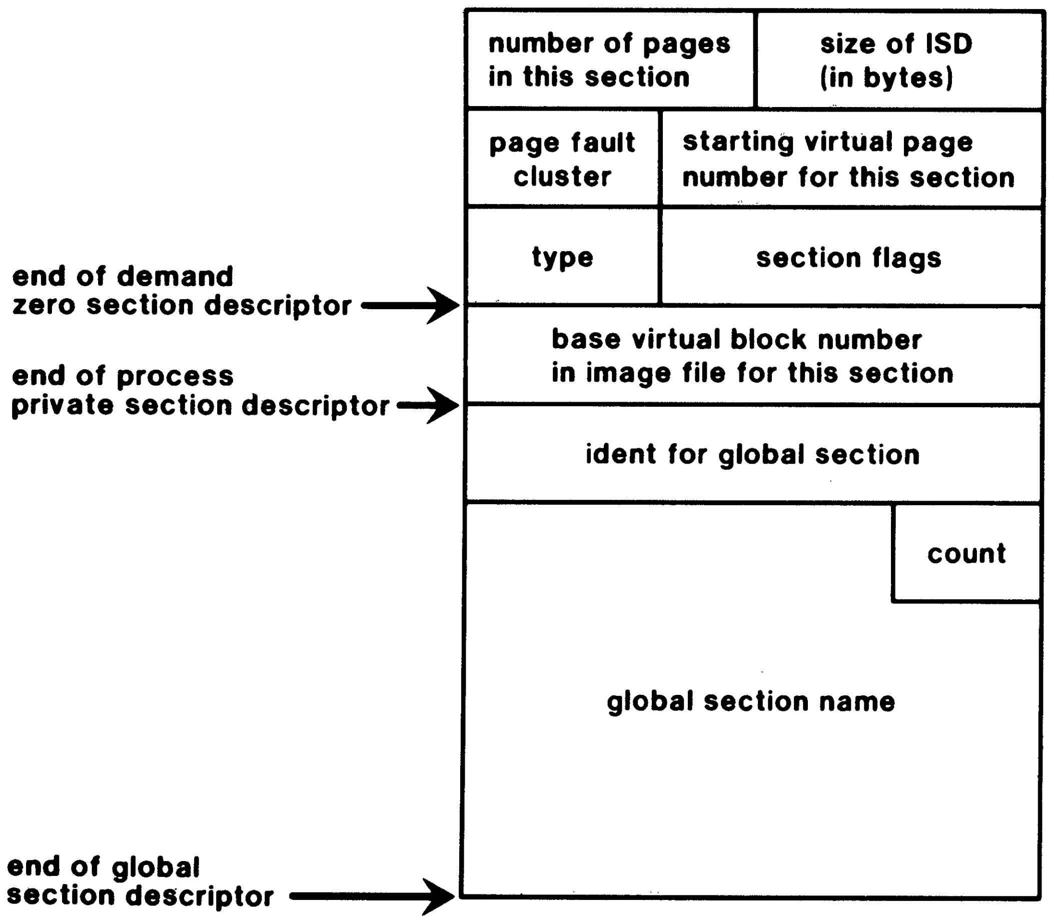


Figure 11 Image Section Descriptor Formats

Process Section Table (PST)

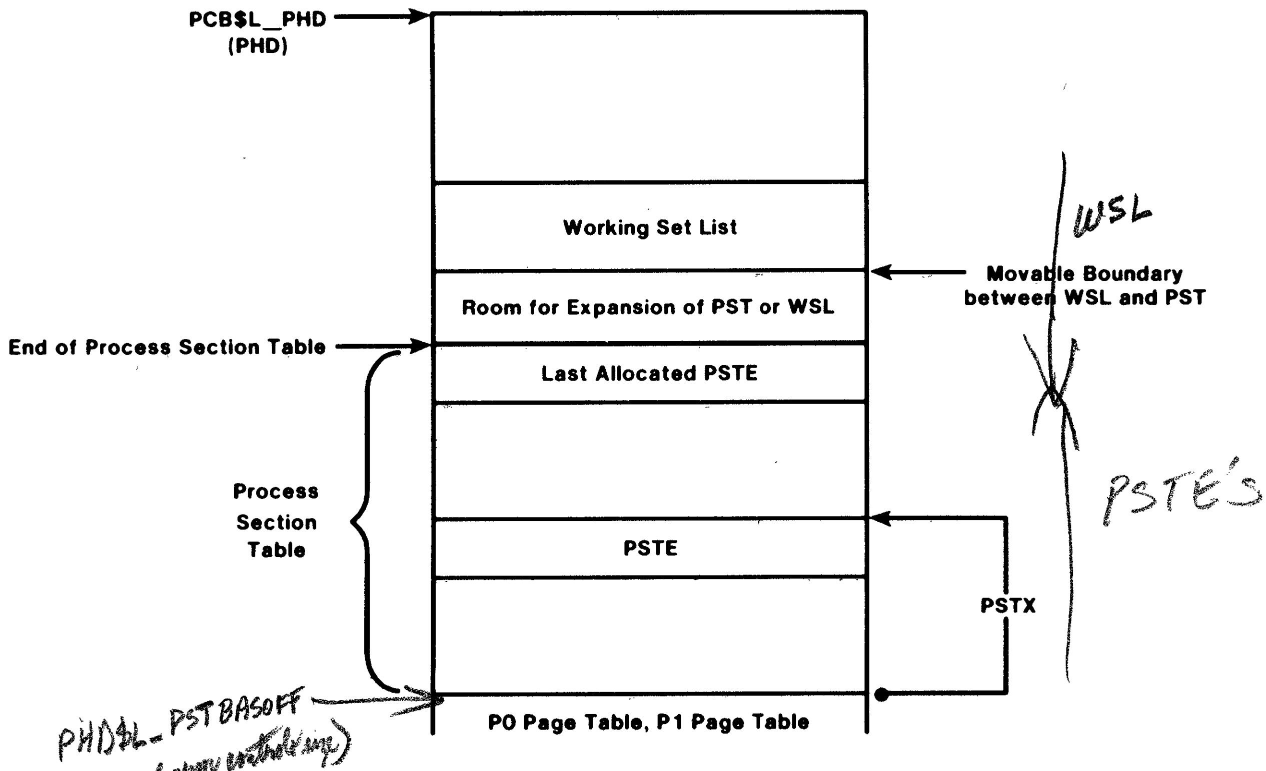


Figure 12 Process Section Table

- Contains entries that locate image sections on disk
- Grows toward lower offsets in the variable portion of the process header

*SYSGEN -

PROCSECTCNT

Process Section Table Entry

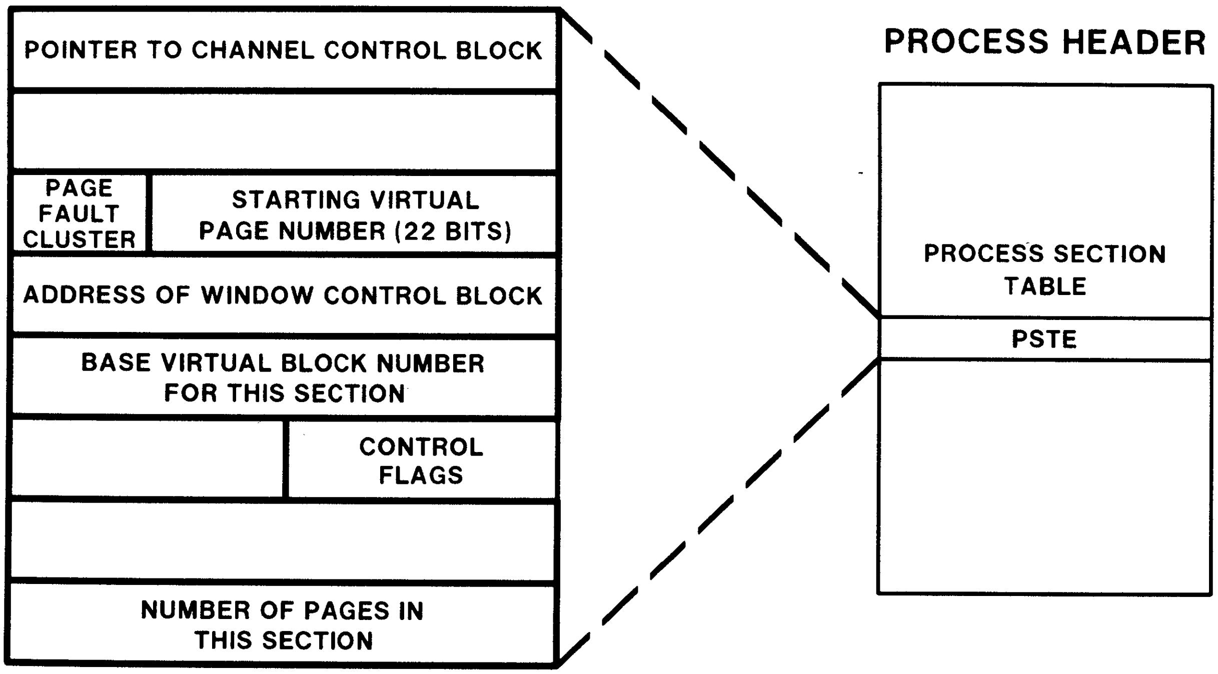


Figure 13 Process Section Table Entry

- Relates virtual pages to virtual blocks in image file
- Is filled in using linker information in Image Section Descriptor (ISD)
- Has control flags to describe attributes of the section (GBL, CRF, DZRO, WRT, etc.)

HOW PTEs, PSTEs ARE FILLED IN by image activator

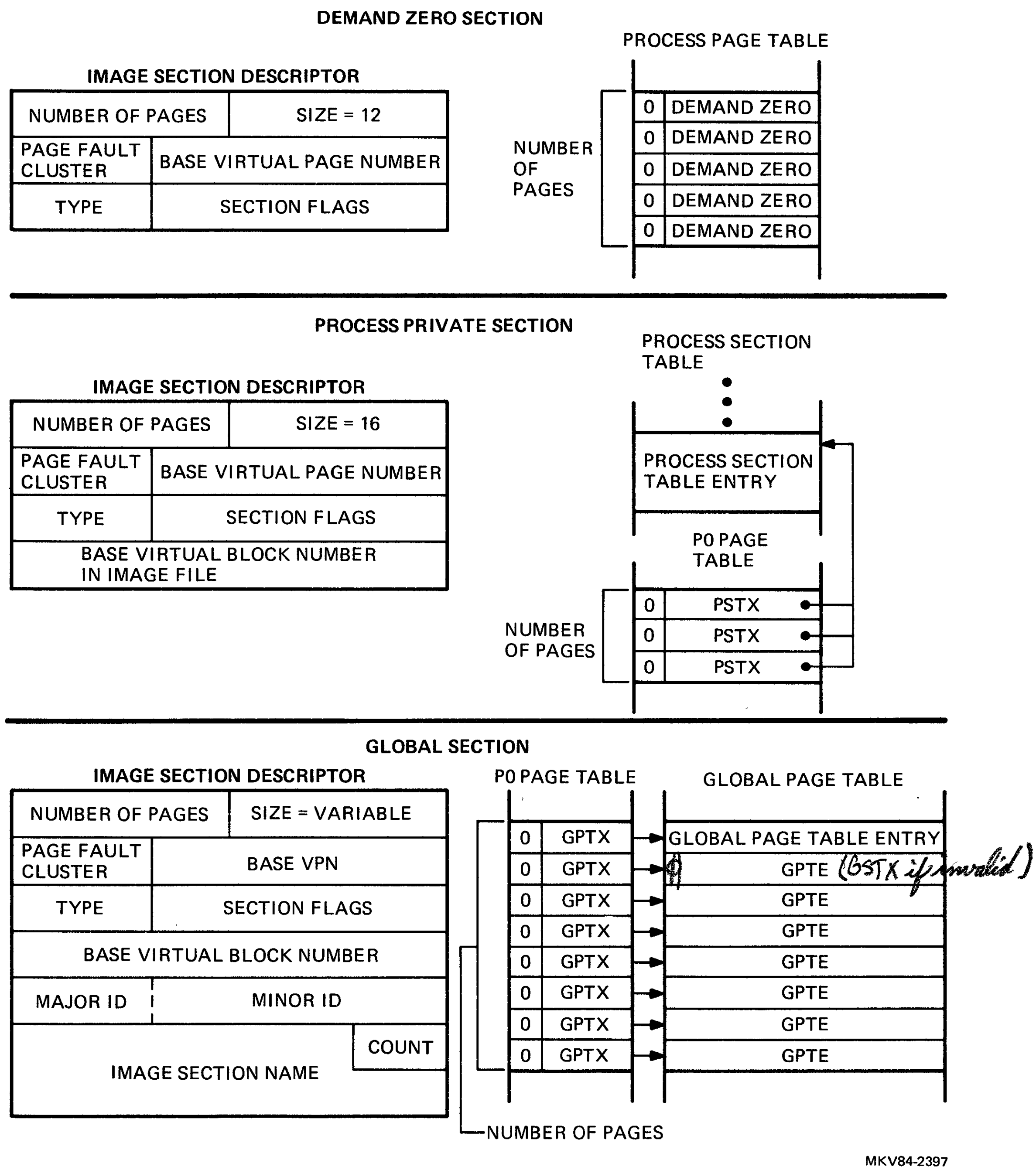


Figure 14 How PTEs, PSTEs Are Filled In

Process Header

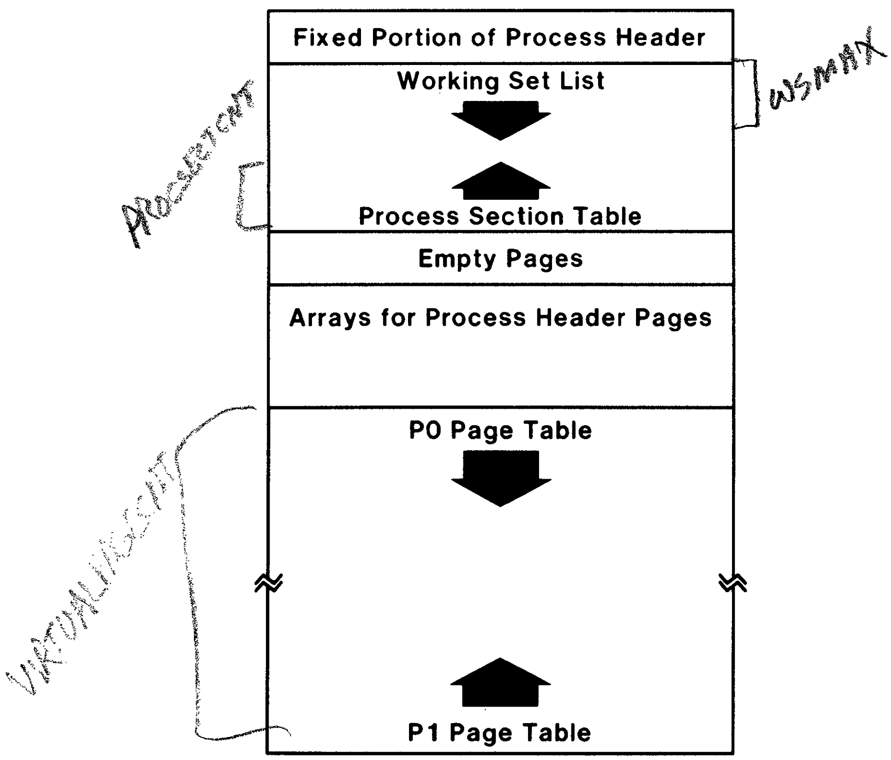


Figure 15 The Process Header

Four major areas of the process header are used in paging operations:

Area	SYSGEN Parameters
• P0 page table • P1 page table	VIRTUALPAGECNT
• Process Section Table	PROCSECTCNT
• Working set list	WSMAX

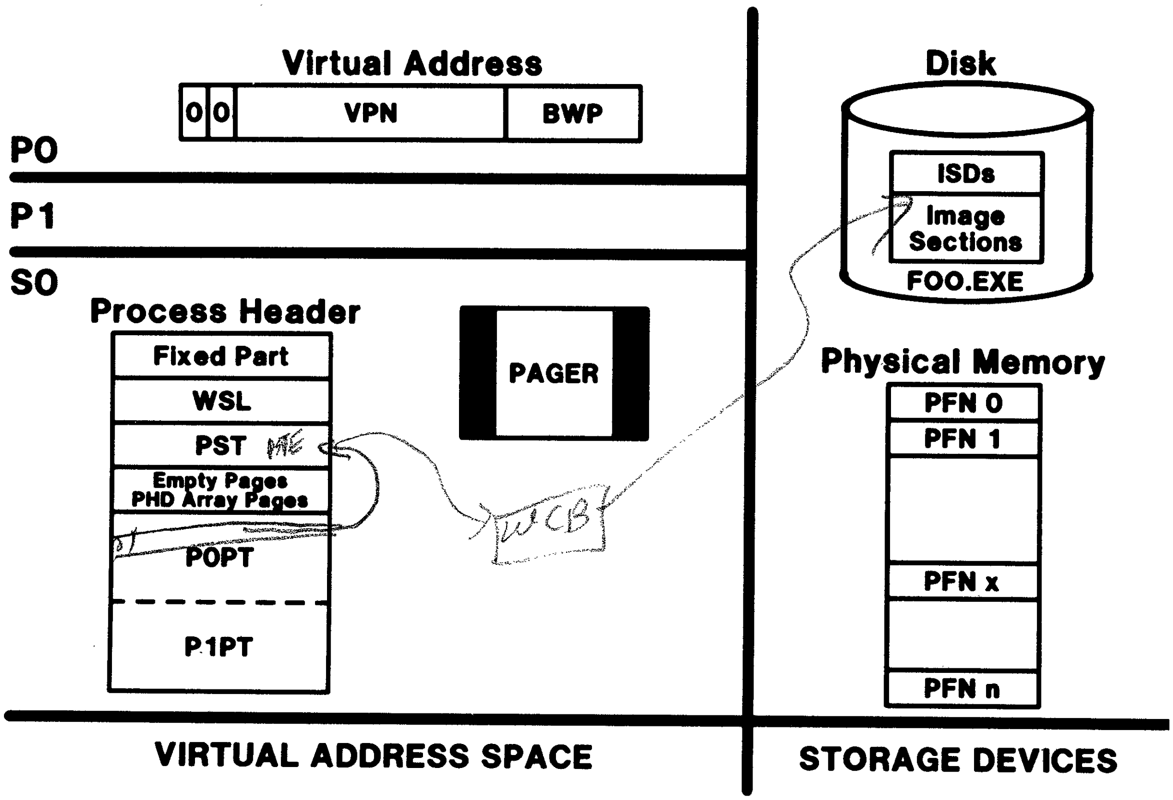


Figure 16 Overview of Page Fault Handling

Sample Program

```
        .TITLE  testing
;       program to illustrate memory management topics
;       it doesn't do much

        .PSECT  $RO$      RD,NOWRT, NOEXE          PSTE1

AA:     .LONG   10
BB:     .LONG   25

        .PSECT  $RW$      RD, WRT, NOEXE           PSTE2

CC:     .LONG   15
DD:     .LONG   0
                                                    DZRO ISD
        .PSECT  $RWDZRO$  RD, WRT, NOEXE

EE:     .BLKB   6*512

        .PSECT  $CODE$    RD, NOWRT, EXE           PSTE3
        .ENTRY  foo,  ^M<R3, R4>
        MOVL    AA, R4
        MOVL    CC, R3
        MOVL    #10, EE
        MOVL    #SS$_NORMAL, R0
        RET
       .END     foo
```

Example 1 MACRO Program with Four PSECTs

Template for Process Paging Example

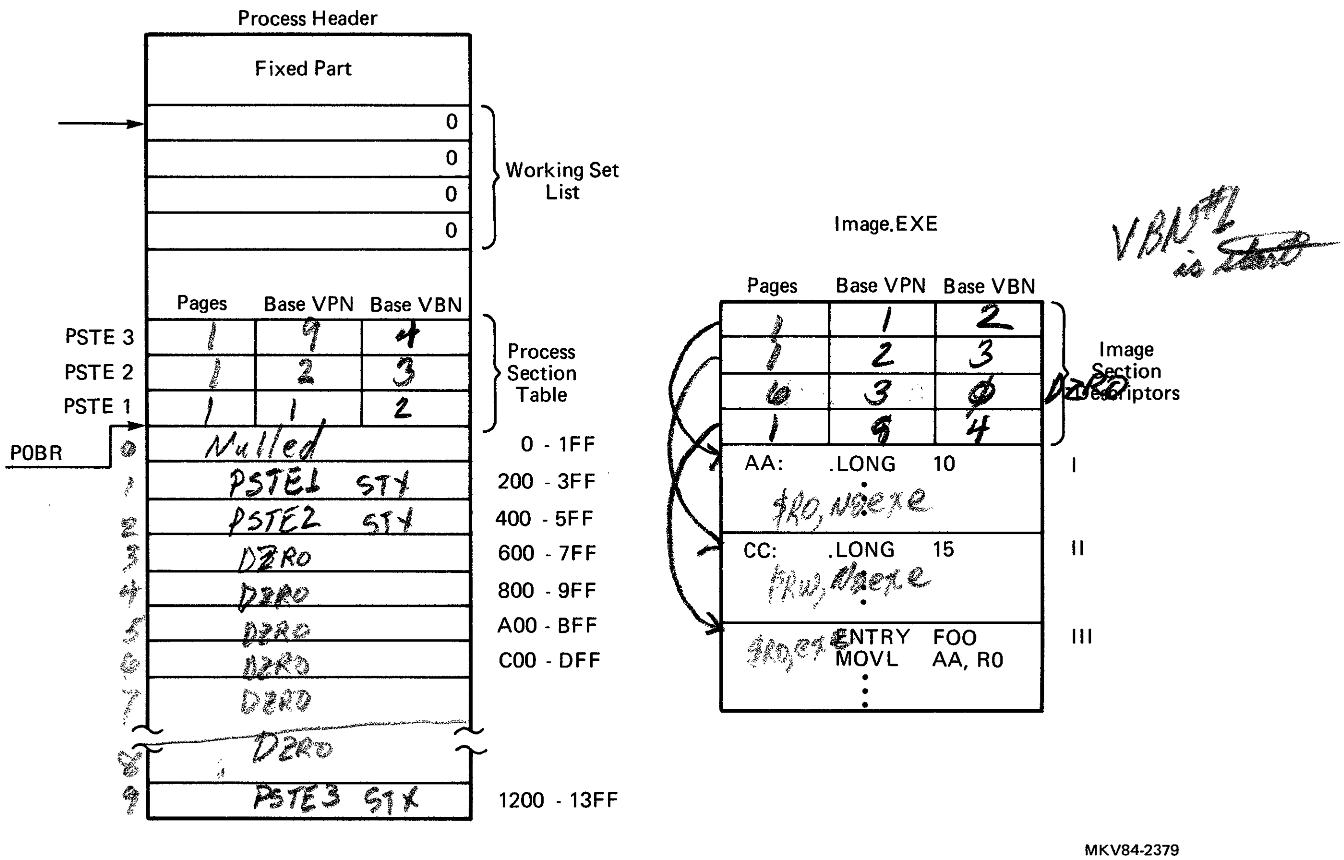


Figure 17 Template for Process Paging Example

initial state after image activation

MORE ON PAGING

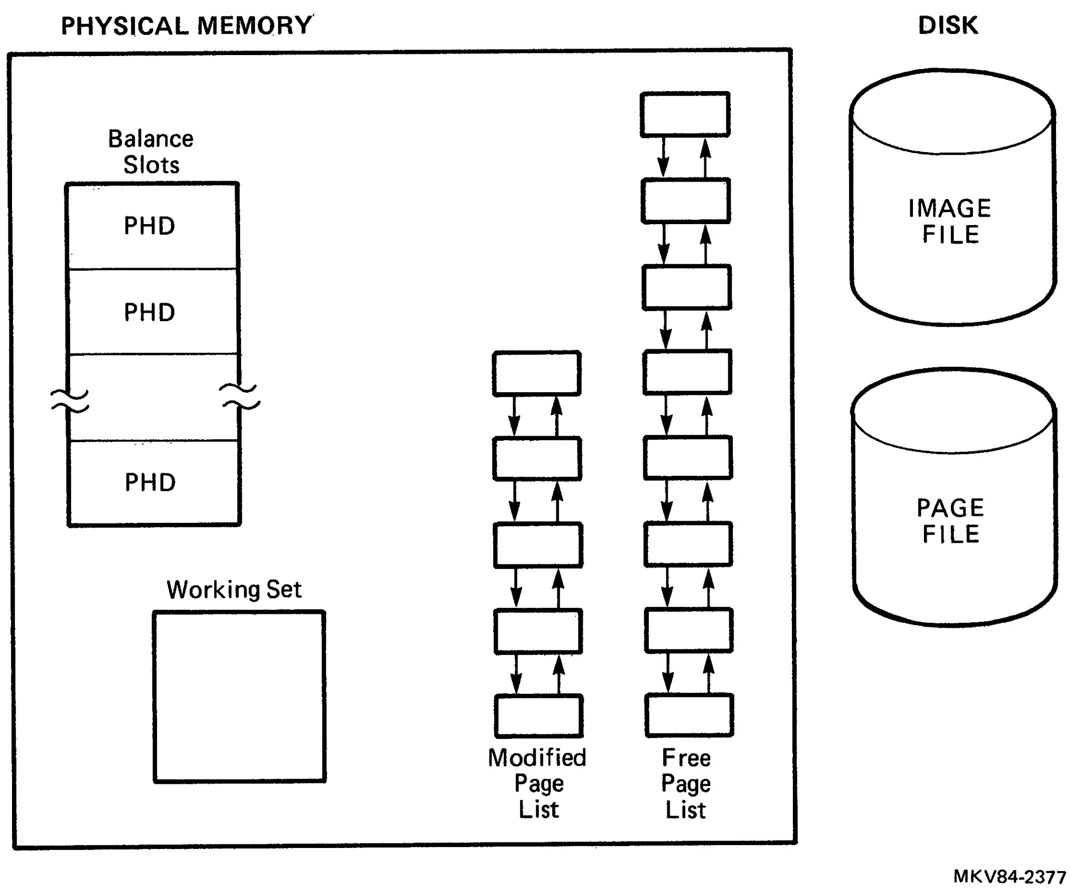


Figure 18 Free and Modified Page Lists

Different Forms of Page Table Entry

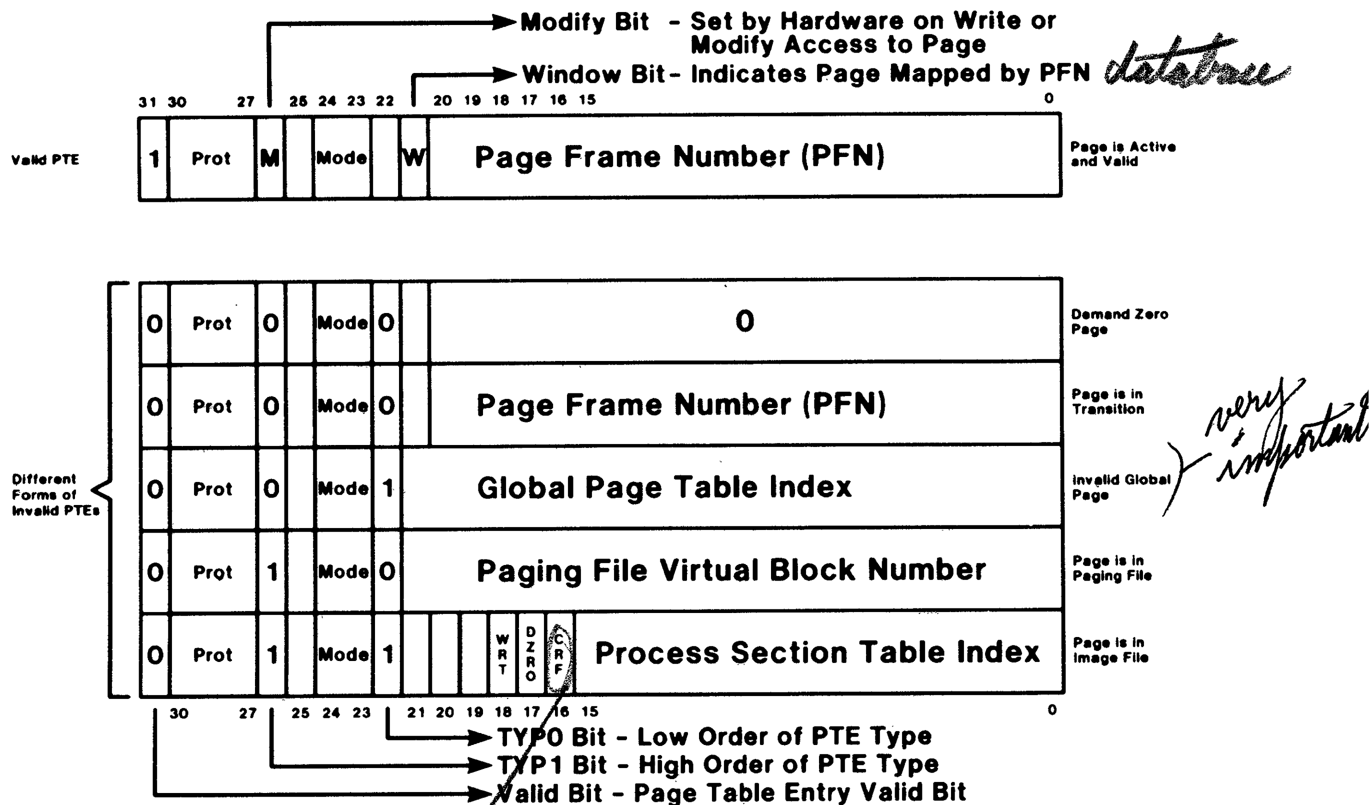


Figure 19 Different Forms of Page Table Entry

copy on reference page
M bit set by software (linker?)

The Paging File

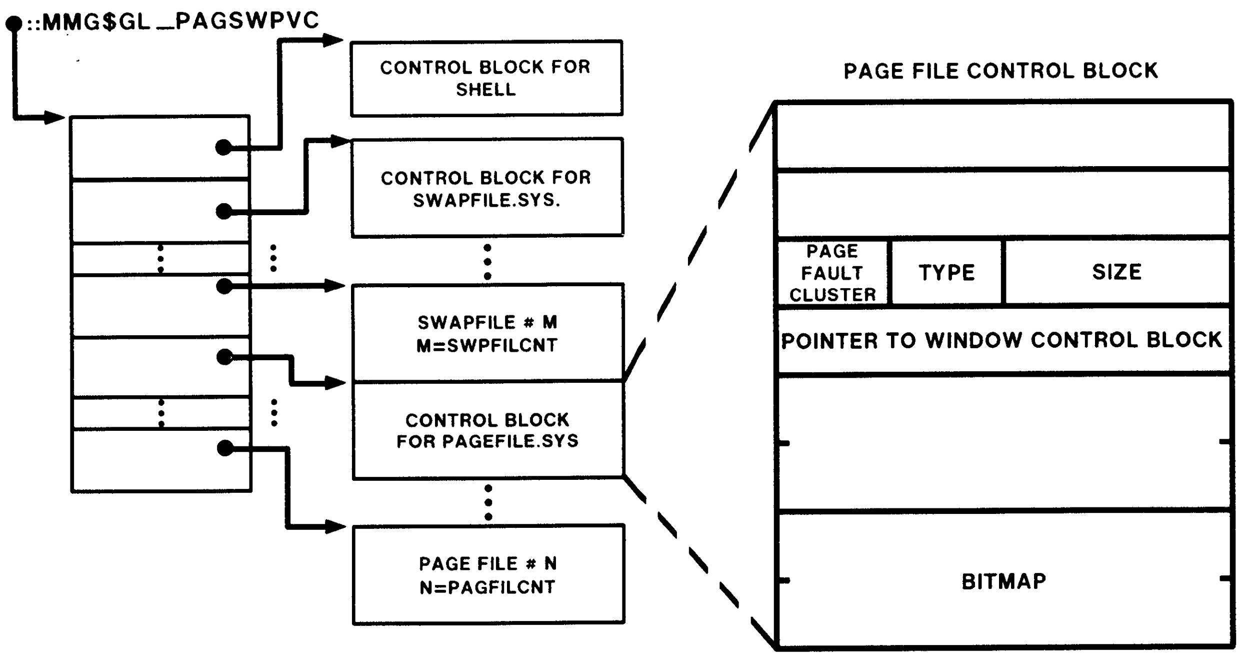


Figure 20 Page File Control Block

(PFL)

Control Block

- Address of bitmap
- Page fault cluster
- Pointer to window control block
- Base virtual block number
- Pages that may be allocated

Bitmap

- One bit per block in the page file
- Bit set implies block available

*SYSGEN -

SWPFILCNT
PAGFILCNT

when assigning from freelist:
1. check PTE invalid old "soft fault" PTE
2. use BAK to repoint that PTE

PFN DATABASE

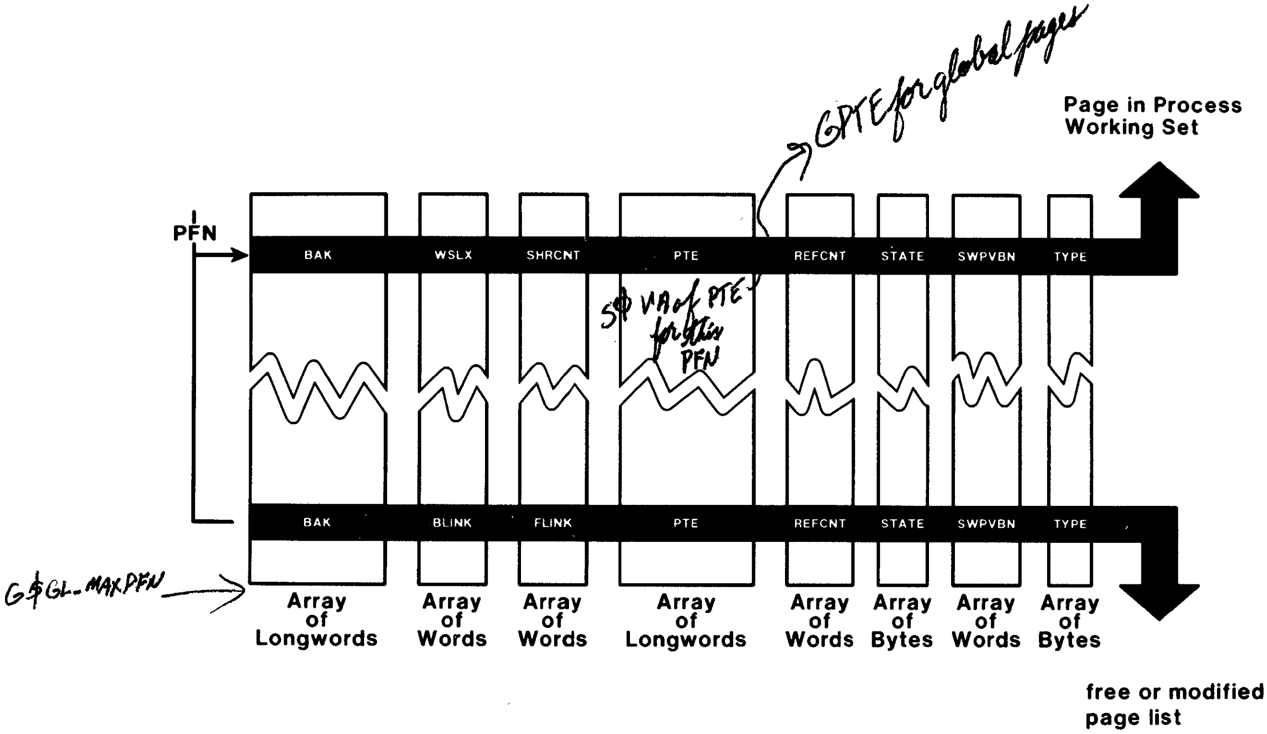


Figure 21 PFN Database

BAK	where page should go if it must leave memory if not valid
WSLX, BLINK	index into working set list, backward link for free & mod list
SHRCNT,FLINK	number of processes sharing page, forward link " "
PTE	points to virtual (S0) address of PTE that maps this page
REFCNT	number of reasons not to put page on free or modified page list ~~(global pages)~~ (I/O outstanding, in WS...)
STATE	specifies list or activity
SWPVBN	virtual block number in swap file or page file
TYPE	type of page - for example, process, system global

Note: PFN is index into arrays.
FLINK and BLINK arrays may be longwords for large physical memory.

Using a Process CRF Page

- Contains read/write data, for example
- Initial copy of page is from image file
- If leaves working set, placed on MPL
- Backed up to paging file

Initial Status of Process CRF Page

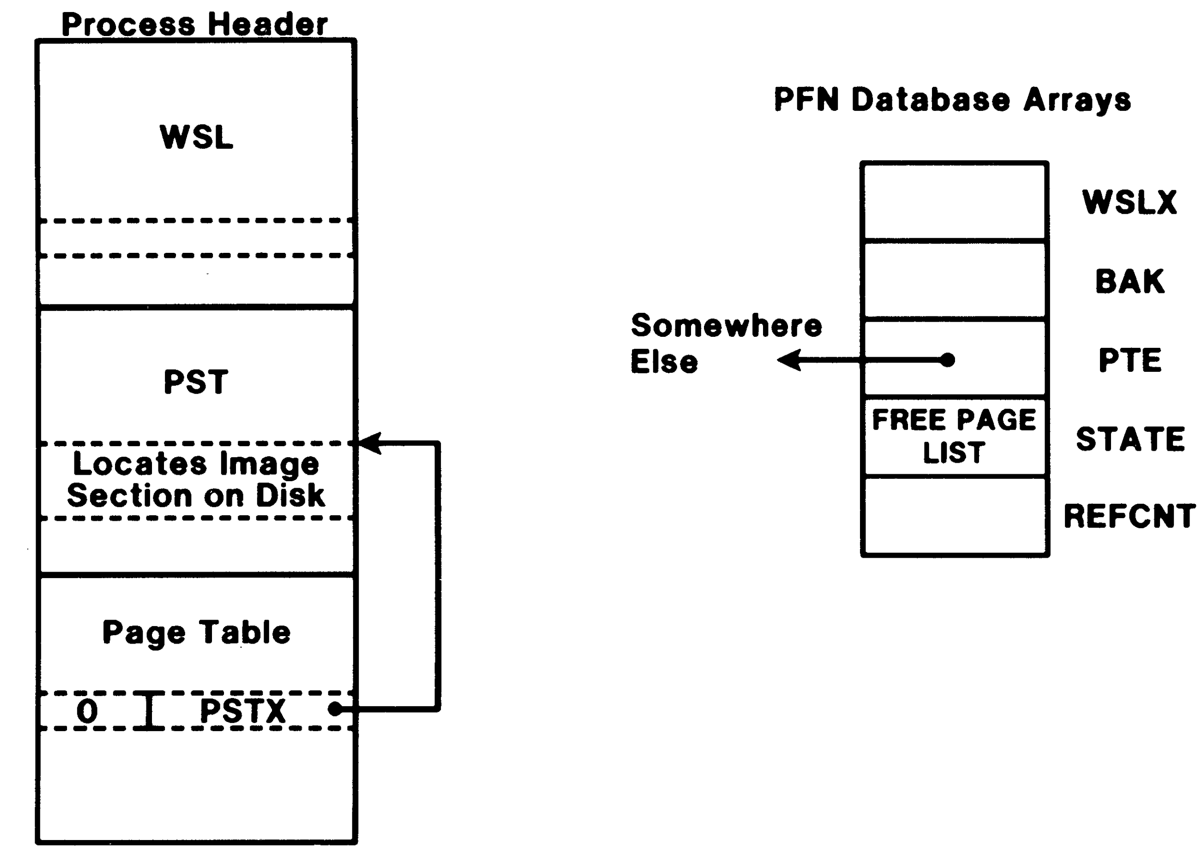


Figure 22 Initial Status of Process CRF Page

Page is invalid and

- PTE has index into Process Section Table
- CRF bits are set in PTE and PSTE
- No connections as yet to PFN database

Page Fault on Process CRF Page (Step 1)

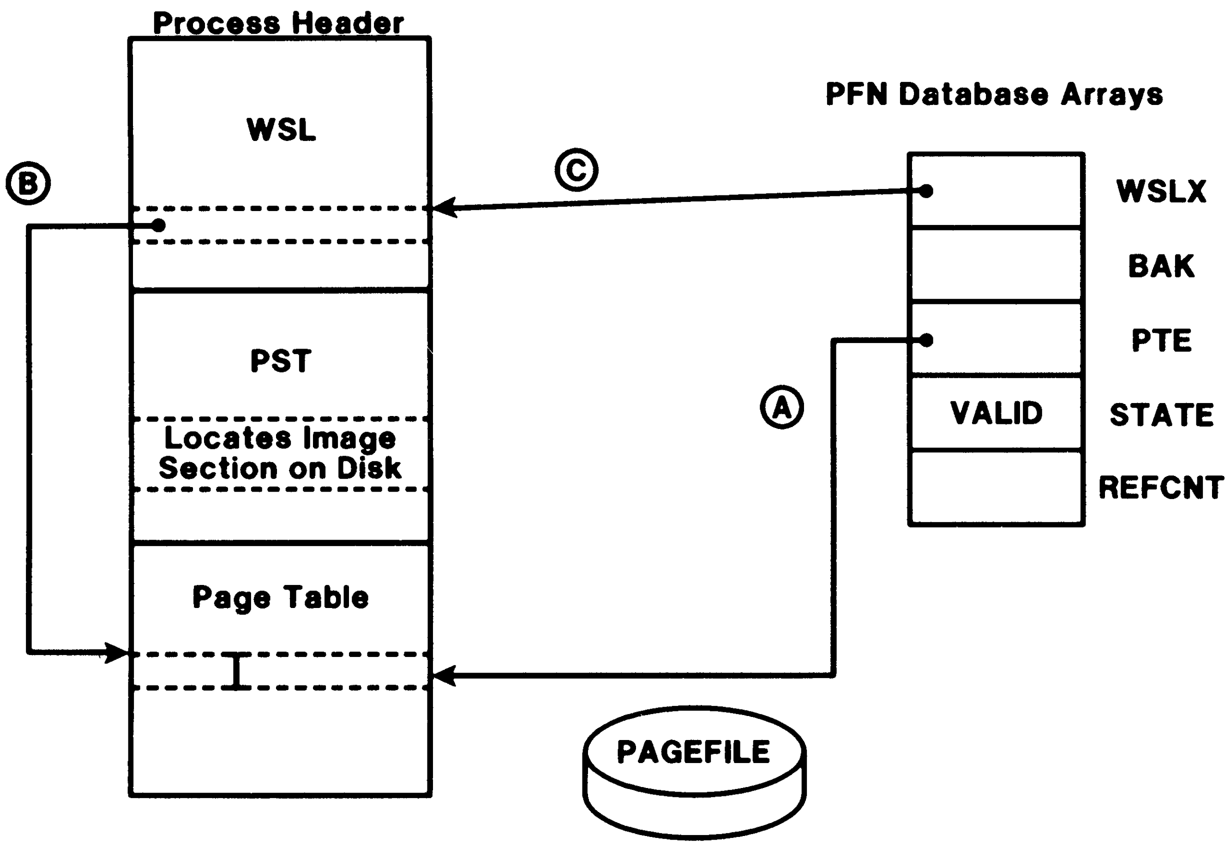


Figure 23 . Page Fault on Process CRF Page (Step 1)

A. PFN PTE array points to the Page Table Entry

B. WSLE is filled with pointer to PTE

C. PFN WSLX array entry contains PHD$W_WSNEXT

Page Fault on process CRF Page (Step 2)

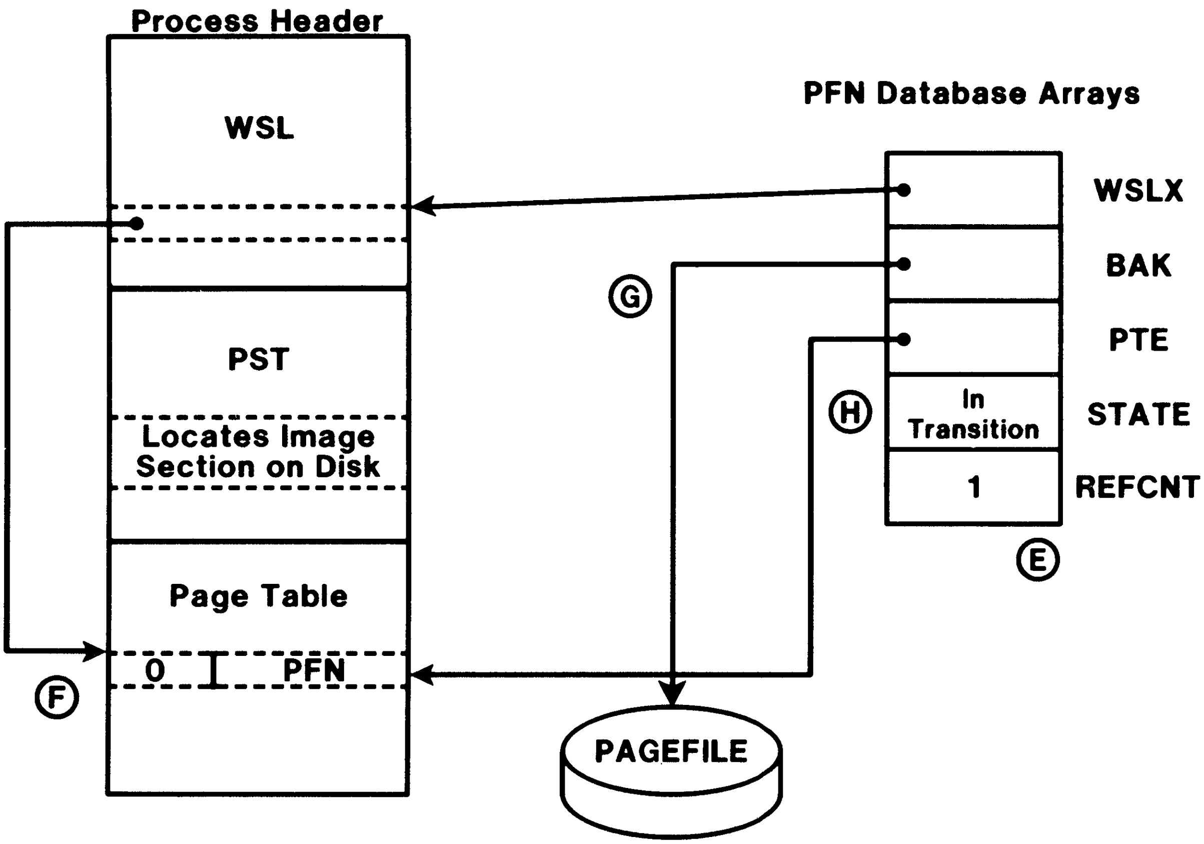


Figure 24 Page Fault on Process CRF Page (Step 2)

E. Increment PFN REFCNT to represent I/O in progress

F. PTE represents a page in transition

G. PFN BAK array is filled with Page File pointer

H. PFN STATE array represents page in transition

Page Fault on Process CRF Page (Step 3)

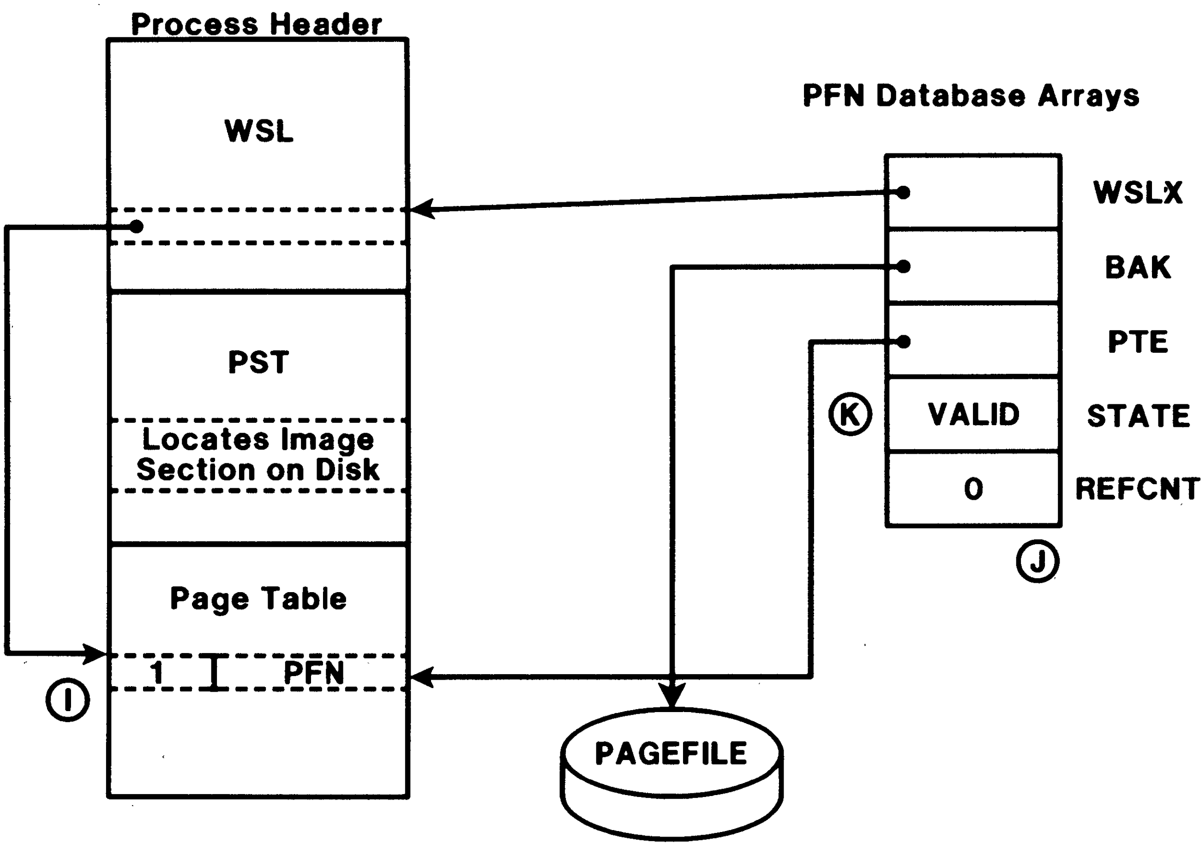


Figure 25 Page Fault on Process CRF Page (Step 3)

When page read completes:

I. PTE is made valid

J. PFN REFCNT is decremented (I/O complete)

K. PFN STATE is changed to valid

Removing Process CRF Page from Working Set

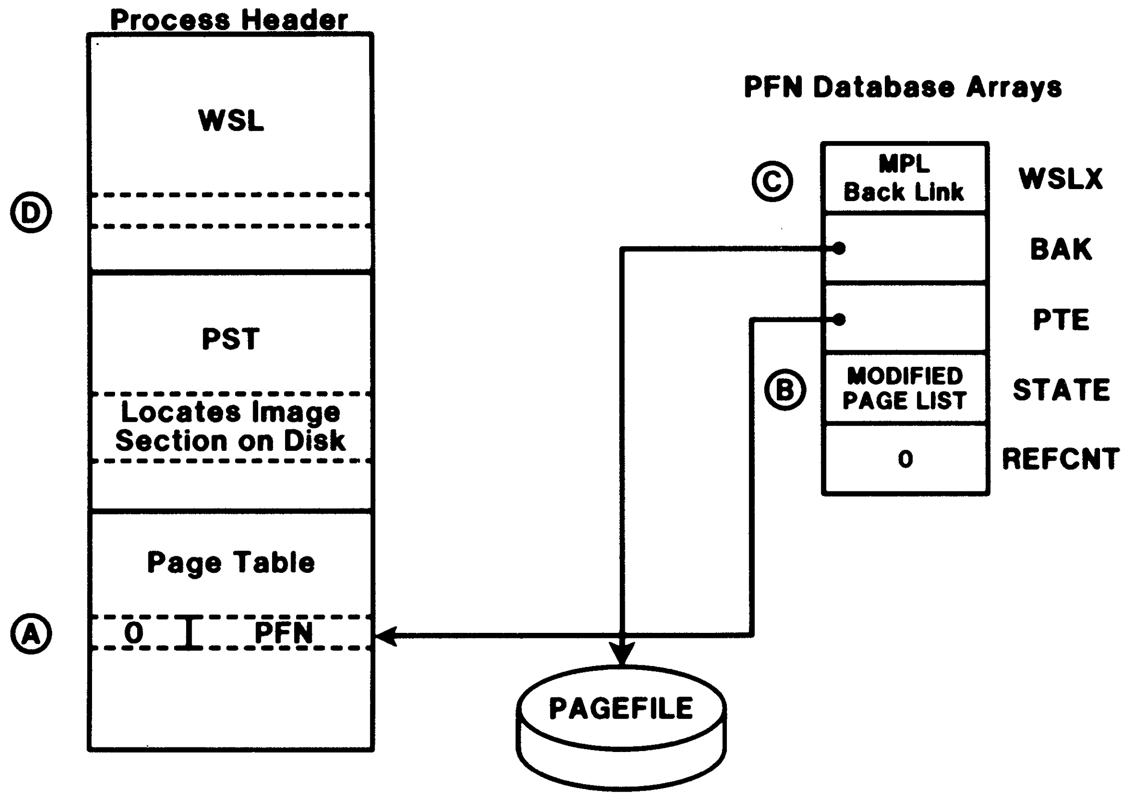


Figure 26 Removing Process CRF Page from Working Set

Page is placed on Modified Page List (MPL)

A. PTE is invalid, but retains PFN

B. PFN STATE array shows page on MPL

C. PFN WSLX array entry has MPL backward pointer

D. Working Set List entry is freed

Process CRF Page Moved from MPL to FPL

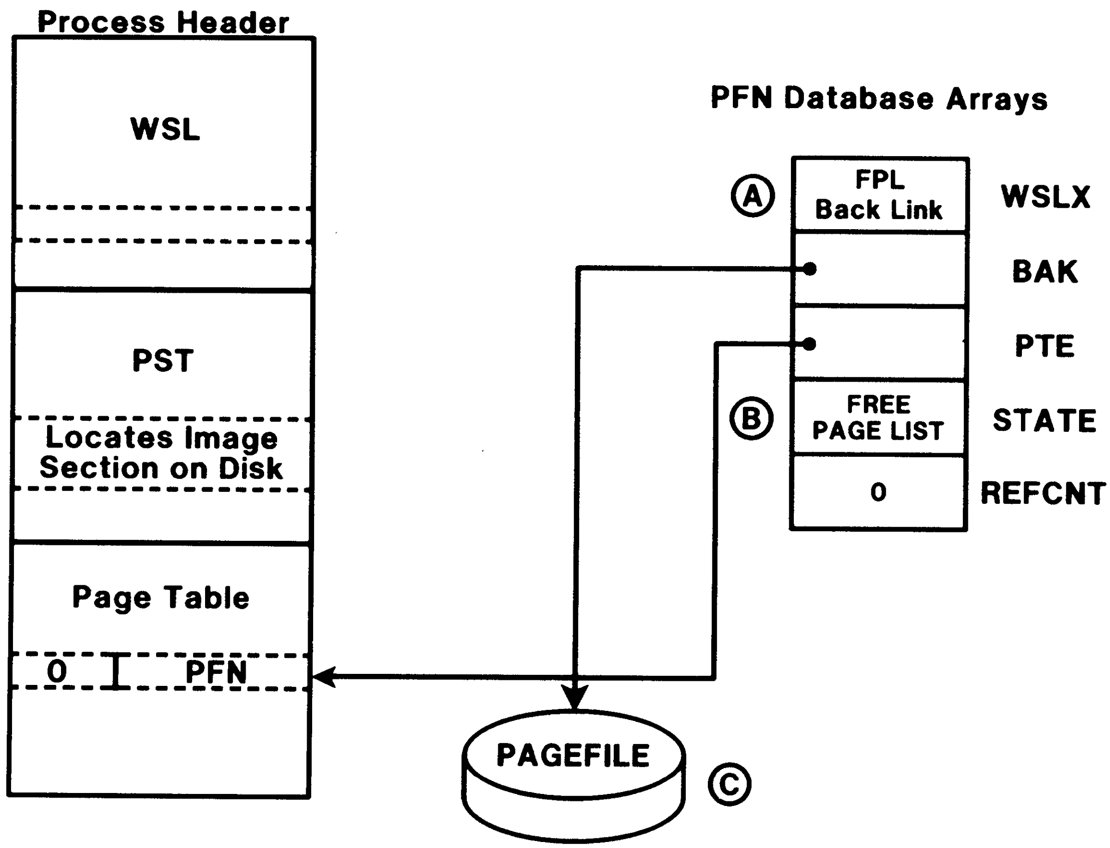


Figure 27 Moving Page from MPL to FPL

A. PFN database contains FPL pointers

B. PFN STATE array shows page on FPL

C. Copy of modified page was written to page file

done 1st ?.

Removing Process CRF Page from Memory

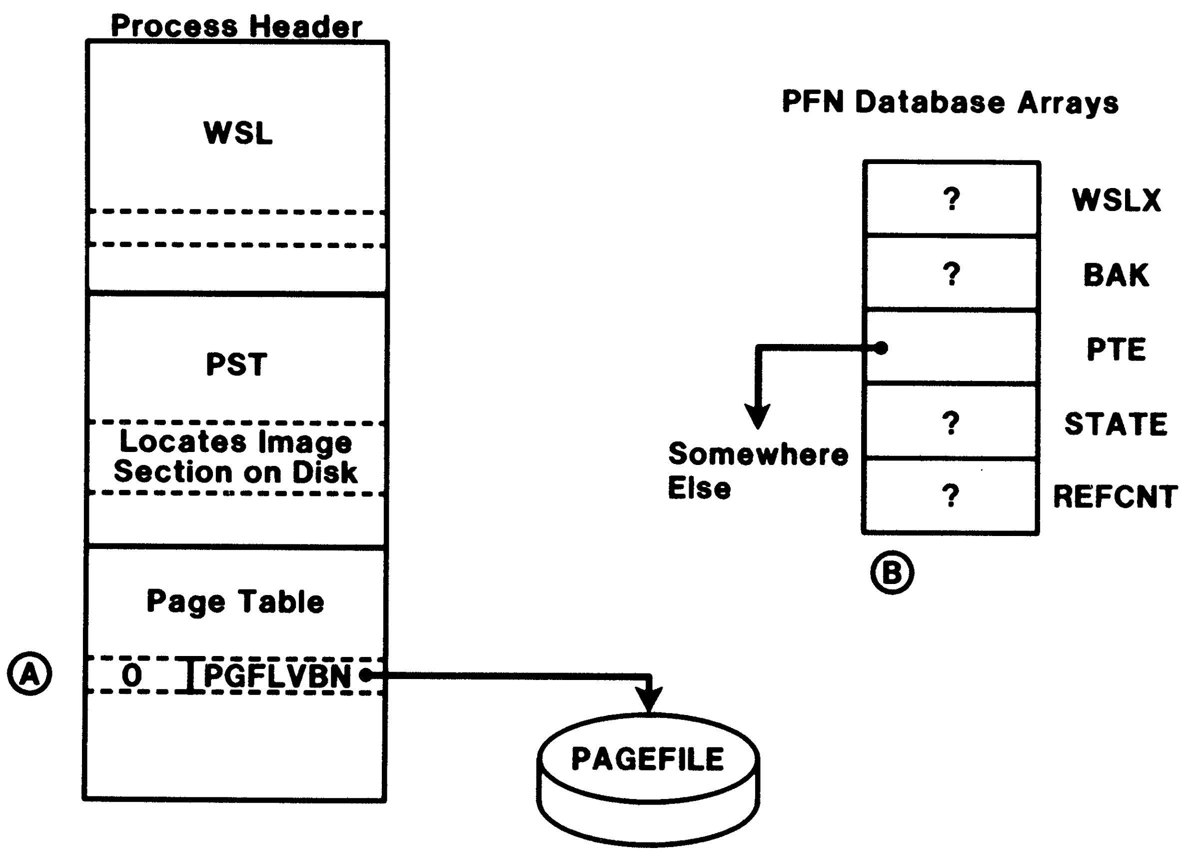


Figure 28 Removing Process CRF Page from FPL

A. Backing store is copied to PTE (and TYP1 bit set)

B. All links to PFN database are broken

DATA STRUCTURES USED BY THE PAGER

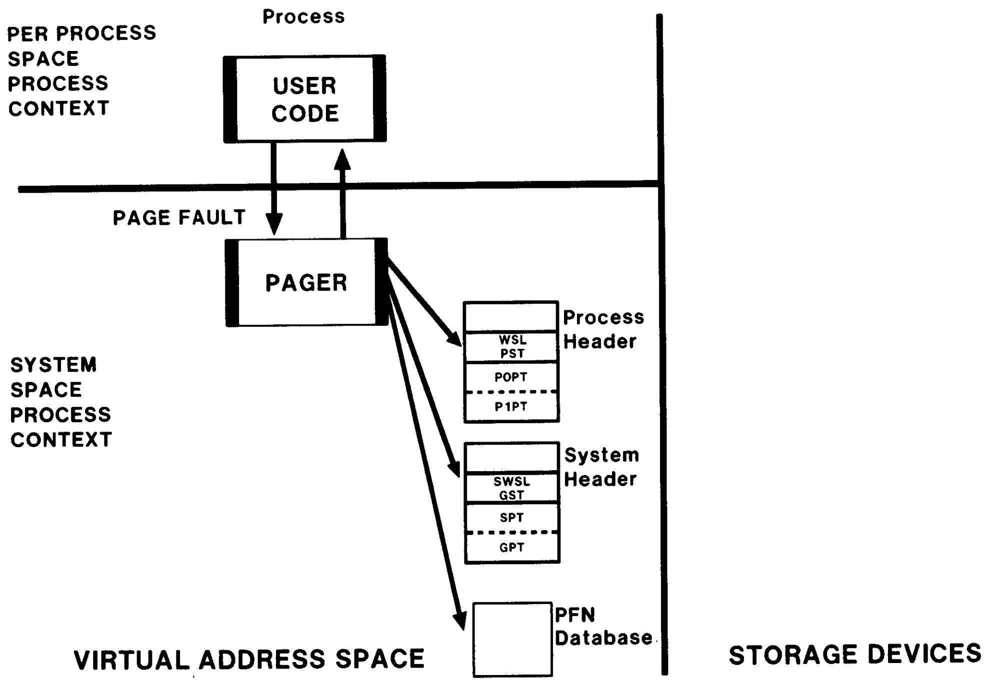


Figure 29 Data Structures Used by the Pager

Table 1 Where Memory Management Information is Stored

Memory Management Information	Data Structure	
Process (P0 and P1 space)	Process Header	- Process Section Table - Page Tables - Working Set List
System (S0 space)	System Header	- System Page Table
Global Sections	System Header	- Global Page Tables - Global Section Table
Physical Memory	PFN Database	

GLOBAL PAGING DATA STRUCTURES

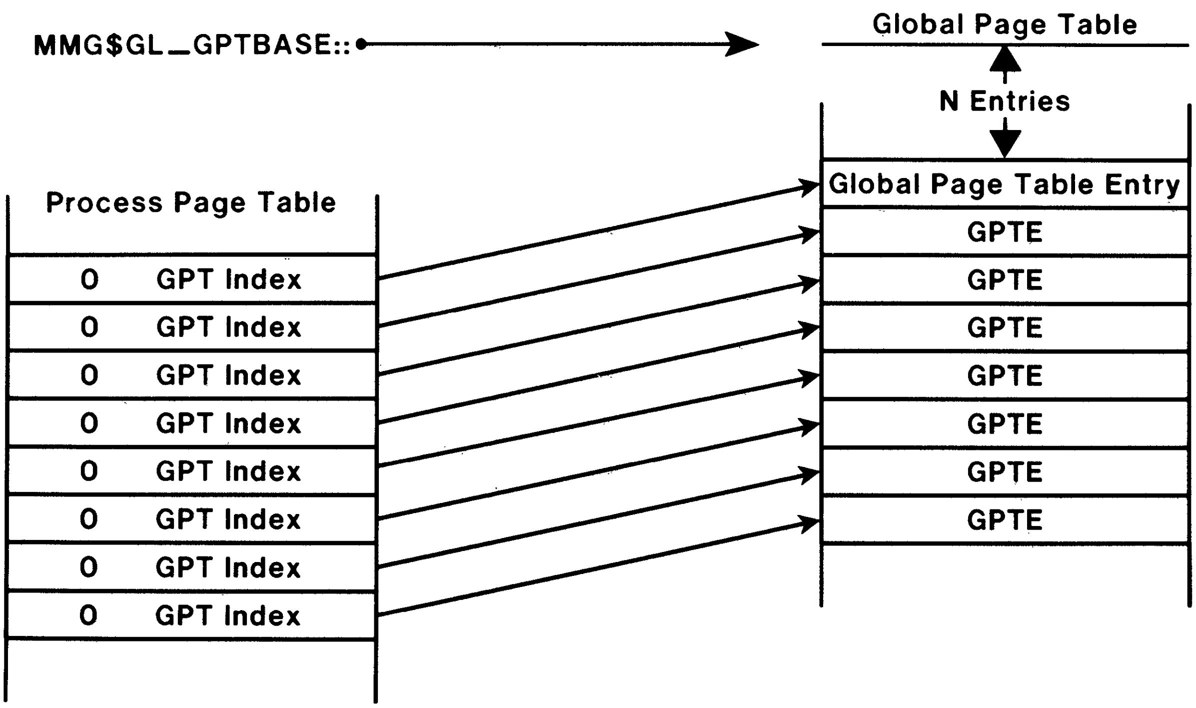


Figure 30 Process PTEs Map to Global PTEs

Global Page Table

- Central location for global page information
- Mapped into S0 space (S0PT)

*SYSGEN -

GBLPAGES
GBLPAGFIL

Relationship Among Global Section Data Structures

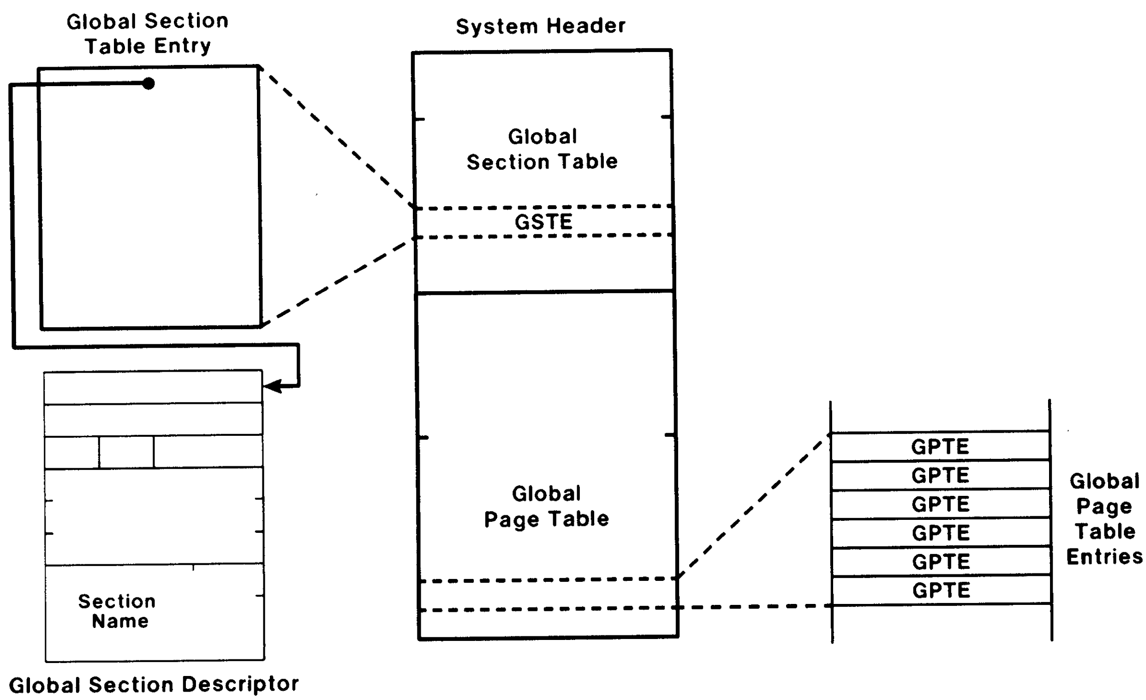


Figure 31 Relationship Among Global Section Data Structures

Three data structures contain global section information:

1. Global page table
2. Global section table (similar to process section table)
3. Global section descriptors (allow the location of global section information by name)

GSDs are placed in either a system queue or a group queue

*SYSGEN -

GBLSECTIONS

Using a Global Read/Write Page

- Result of a $CRMPSC (global), for example
- Higher probability of "cheap" page faults on global pages
- Uses global structures in the system header

Initial Status of Global Read/Write Section Page

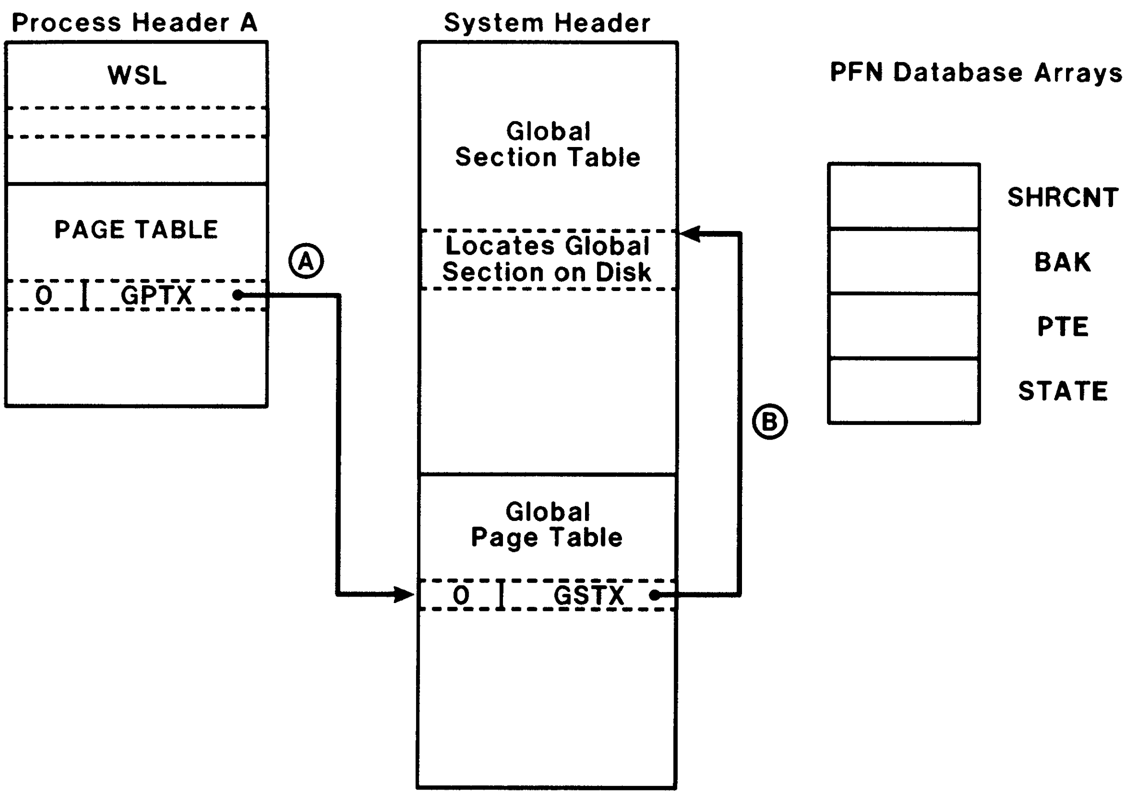


Figure 32 Initial Status of Global Read/Write Section Page

A. Process A PTE points to the Global Page Table Entry (GPTE)

B. GPTE points to Global Section Table Entry (page is not in physical memory)

Adding Global Read/Write Section Page to Working Set

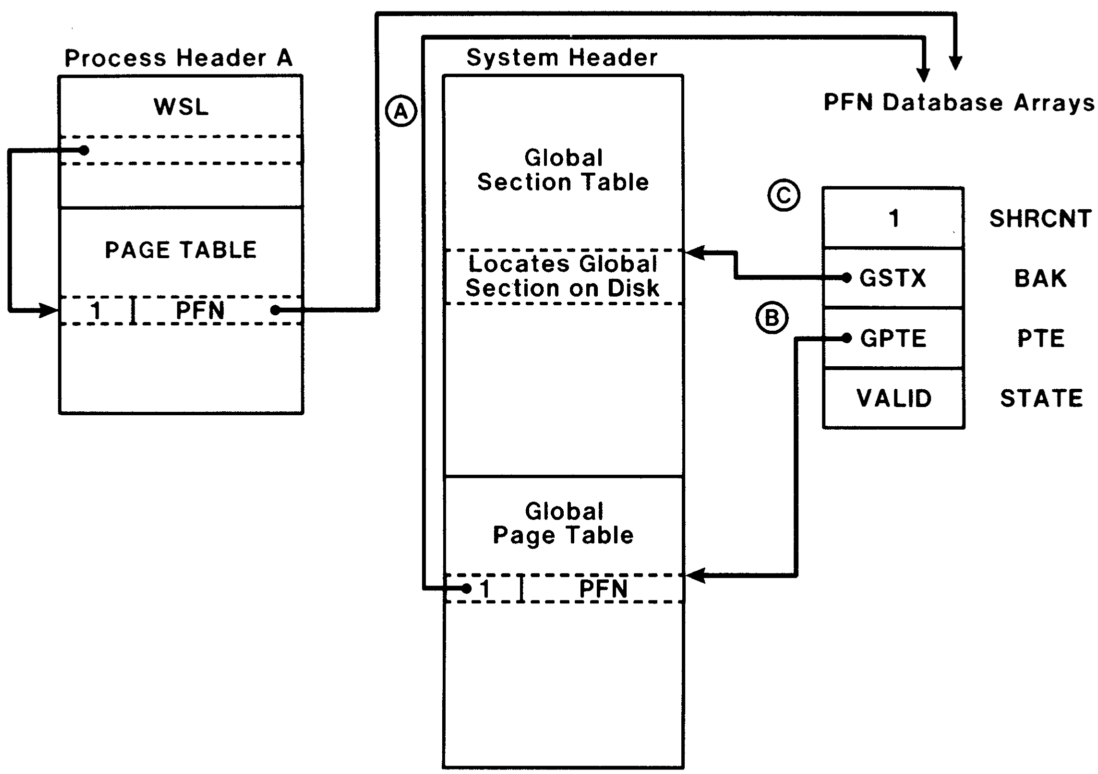


Figure 33 Adding Global Read/Write Section Page to Working Set

When Process A faults the global page:

A. Both the process PTE and the GPTE contain the page frame number

B. The PFN database points only to the system header data structures (GSTE and GPTE)

C. The SHRCNT is initialized to 1

Initial Status of PTE of Second Process

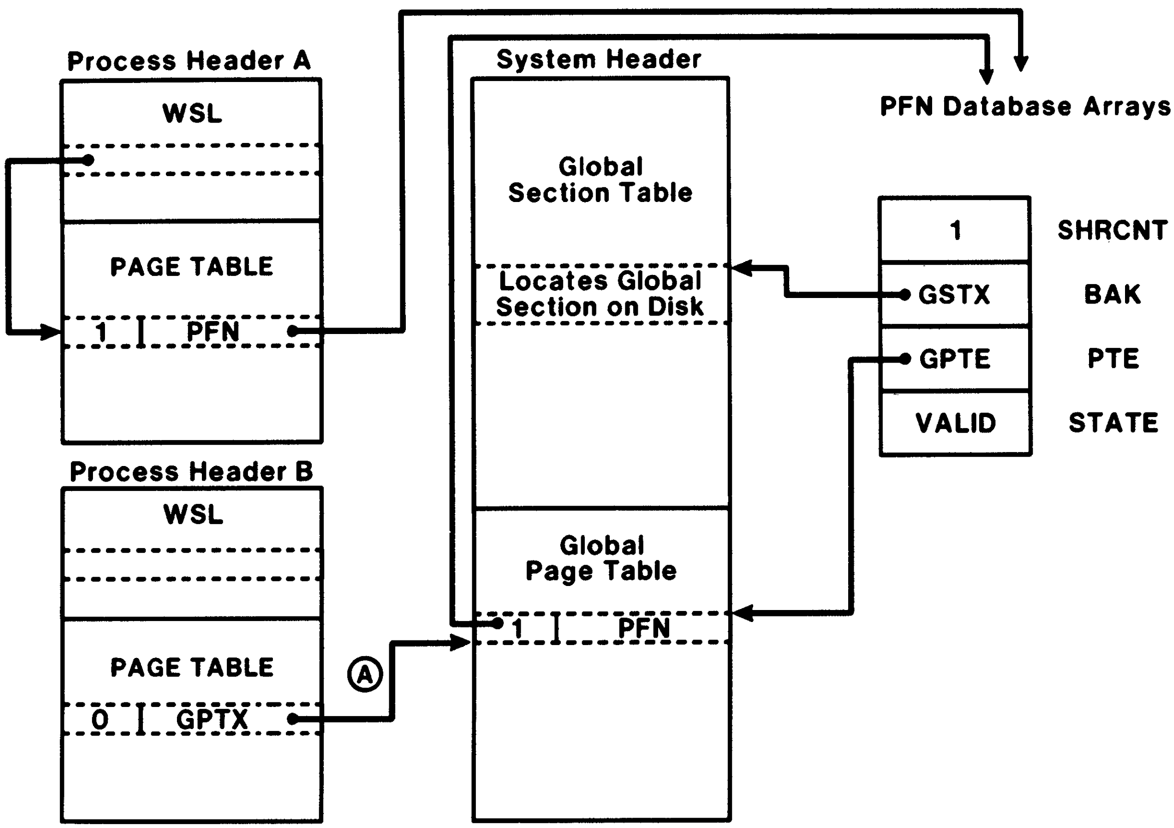


Figure 34 Initial Status of PTE of Second Process Mapping the Same Global Section

A. When Process B maps the same global section, its PTE contains the GPTX.

Adding Global Read/Write Section Page to Second Working Set

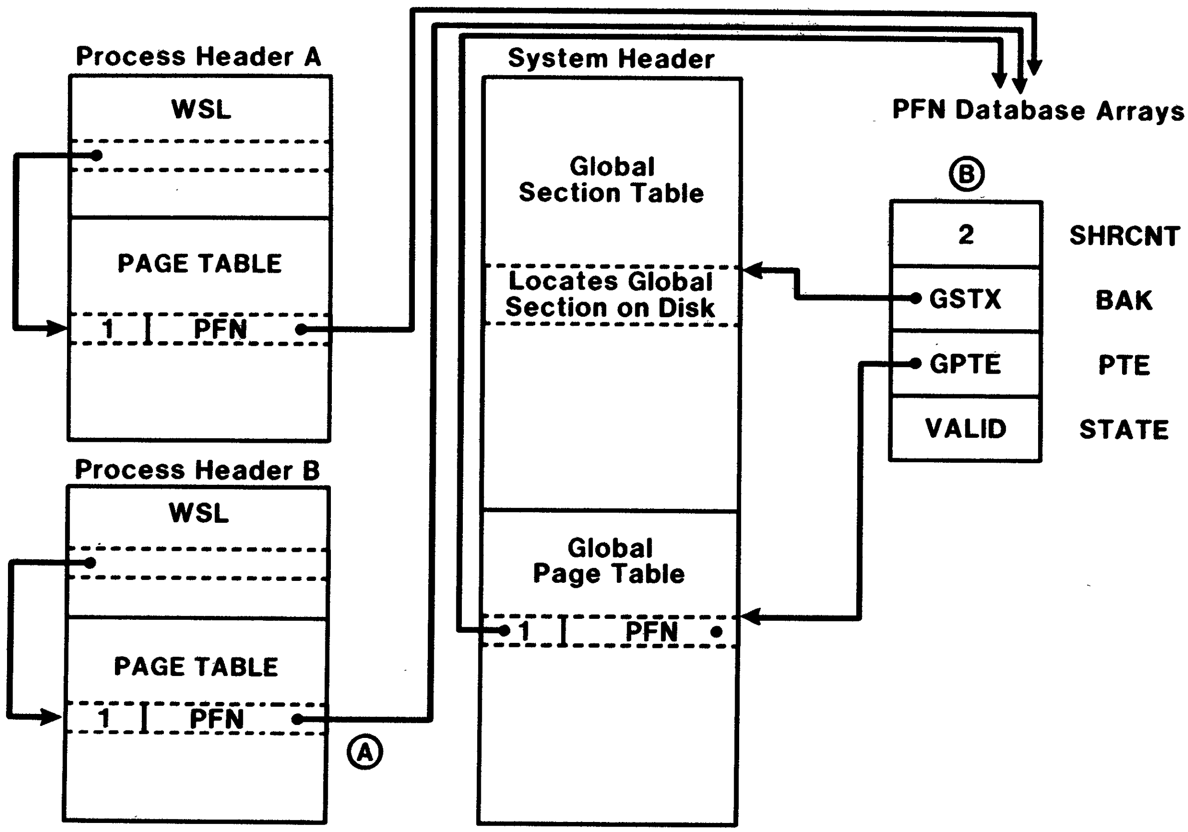


Figure 35 Adding Global Read/Write Section Page to Second Working Set

A. When Process B faults the same global page as Process A, the PTE of Process B also points to the page frame.

B. The only change in the system data structures is the incrementing of the SHRCNT value to two.

Removing Global Read/Write Section Page from Working Set

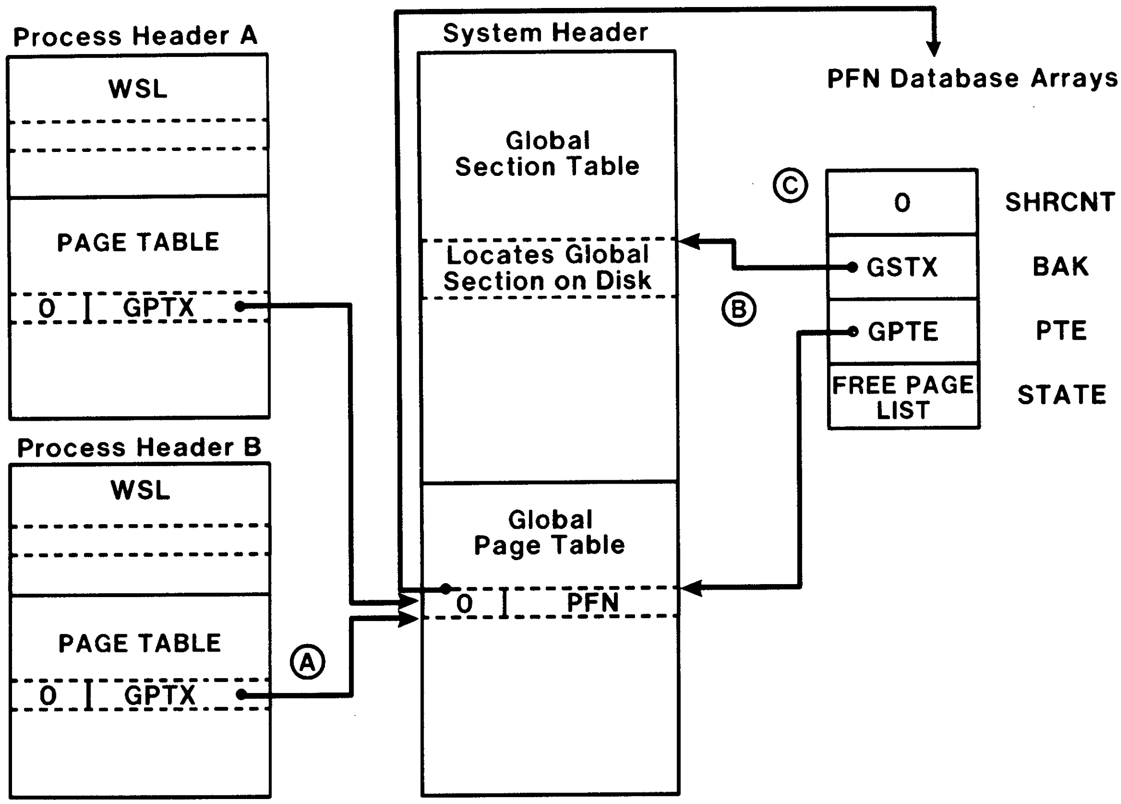


Figure 36 Removing Global Read/Write Section Page from Working Set

Eventually both processes release the global pages from their working sets.

A. As each process loses page from working set, the PFN in the process PTE is overwritten by GPTX.

B. The relationships between the system header data structures and the PFN database are similar to those for a process private page on the free list.

C. The global page is placed on the free or modified page list only after SHRCNT is decremented to zero.

Removing Global Read/Write Section Page from Memory

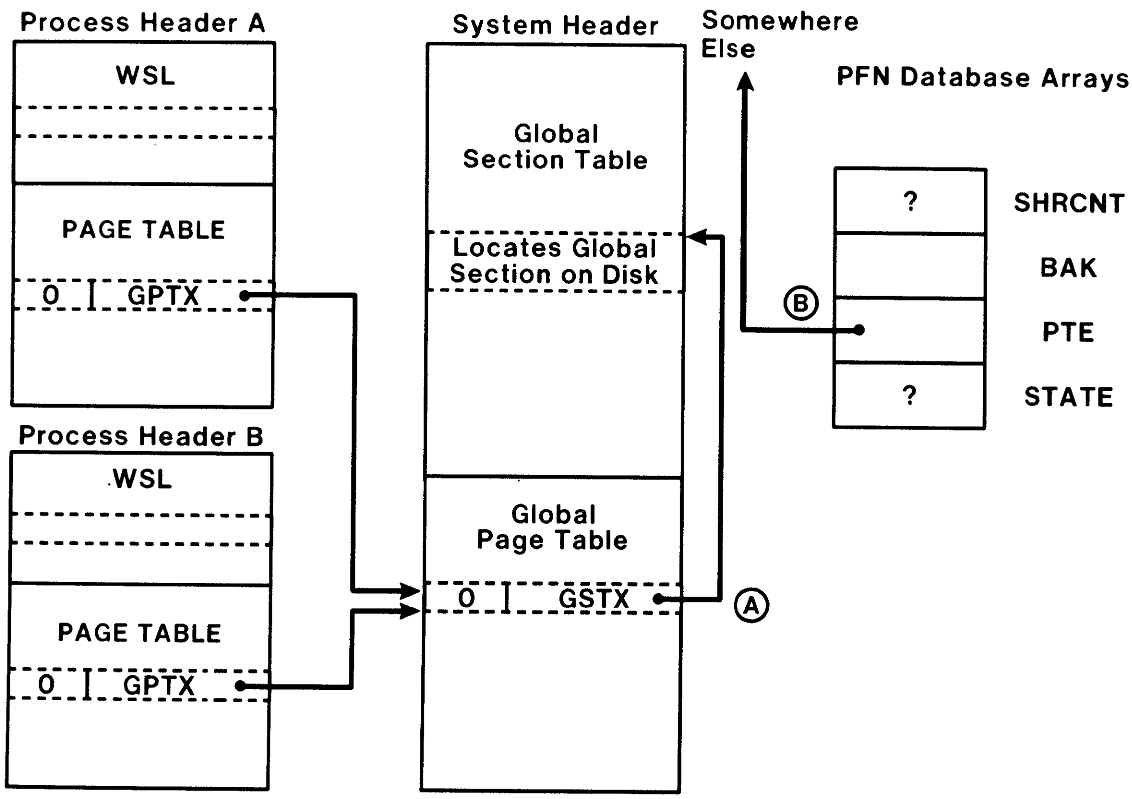


Figure 37 Removing Global Read/Write Section Page from Memory

When the page is allocated to another process from the head of the free list:

A. The system header data structures are returned to their initial states.

B. All links to the PFN database are destroyed.

SUMMARY OF THE PAGER

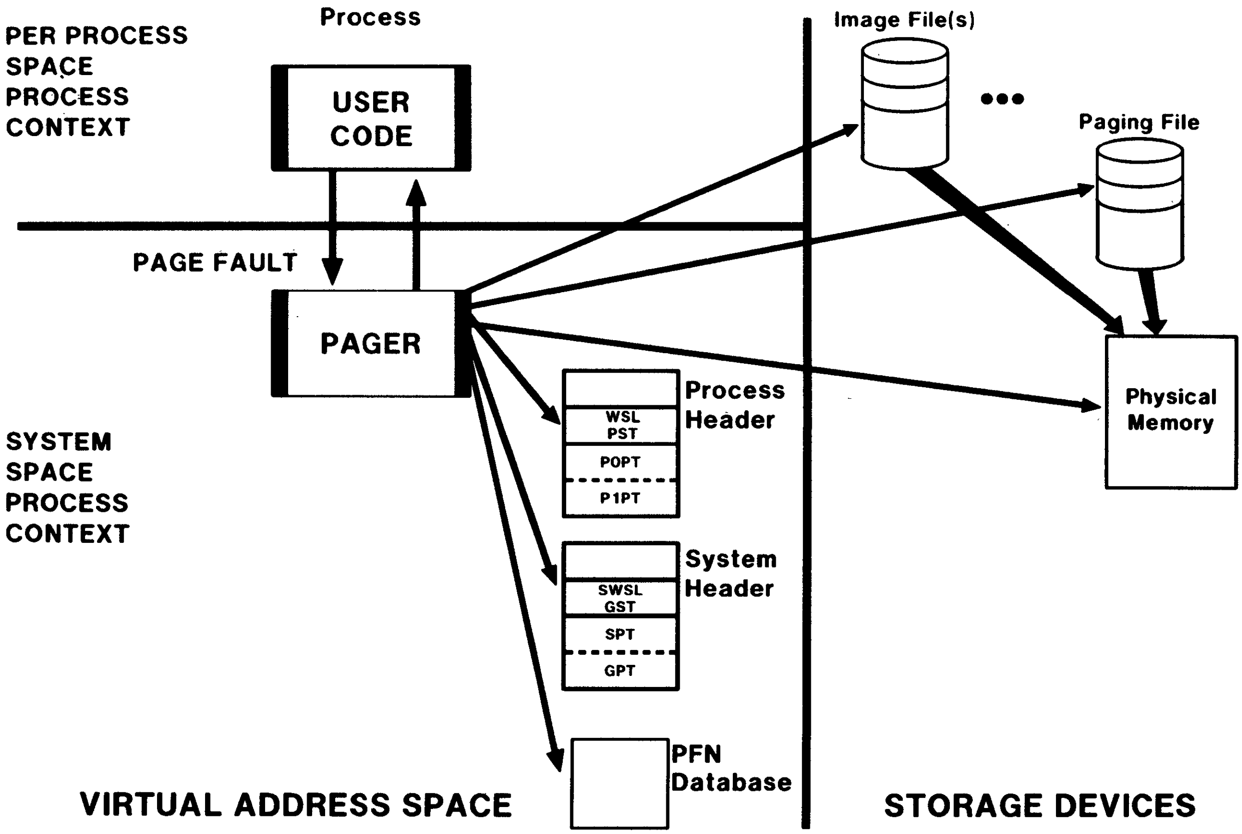


Figure 38 Summary of the Pager

Table 2 SYSGEN Parameters Related to Paging

Function	Parameter
Number of pages required on FPL for processes to grow beyond WSQUOTA (checked at quantum end)	BORROWLIM
Maximum number of global pages (size of Global Page Table)	GBLPAGES
Maximum number of global pages with page file backing store	GBLPAGFIL
Maximum number of global sections that can be made known to system (size of GST)	GBLSECTIONS
Number of pages required on FPL for processes to grow beyond WSQUOTA (checked on page fault)	GROWLIM
Minimum number of fluid pages in a working set	MINWSCNT
Inhibits all page read clustering	NOCLUSTER (*)
Maximum number of paging files	PAGFILCNT
Default page table page fault cluster size	PAGTBLPFC (*)
Default page fault cluster factor for images	PFCDEFAULT
Determine size of process section table (PST)	PROCSECTCNT
Default working set size for processes	PQL_DWSDEFAULT
Minimum default working set size	PQL_MWSDEFAULT
Default working set extent for processes	PQL_DWSEXTENT
Minimum default working set extent	PQL_MWSEXTENT
Default working set quota	PQL_DWSQUOTA
Minimum default working set quota	PQL_MWSQUOTA

(*) special parameter

Table 2 SYSGEN Parameters Related to Paging (Cont)

Function	Parameter
Quota for the size of system working set	SYSMWCNT
Maximum number of working set list entries that may be skipped while freeing an entry	TBSKIPWSL (*)
Number of pages in process virtual address space (P0 plus P1)	VIRTUALPAGECNT
Maximum number of pages in a working set	WSMAX

(*) special parameter

APPENDIX
SUPPLEMENTARY INFORMATION

PROCESS VIRTUAL ADDRESS TRANSLATION

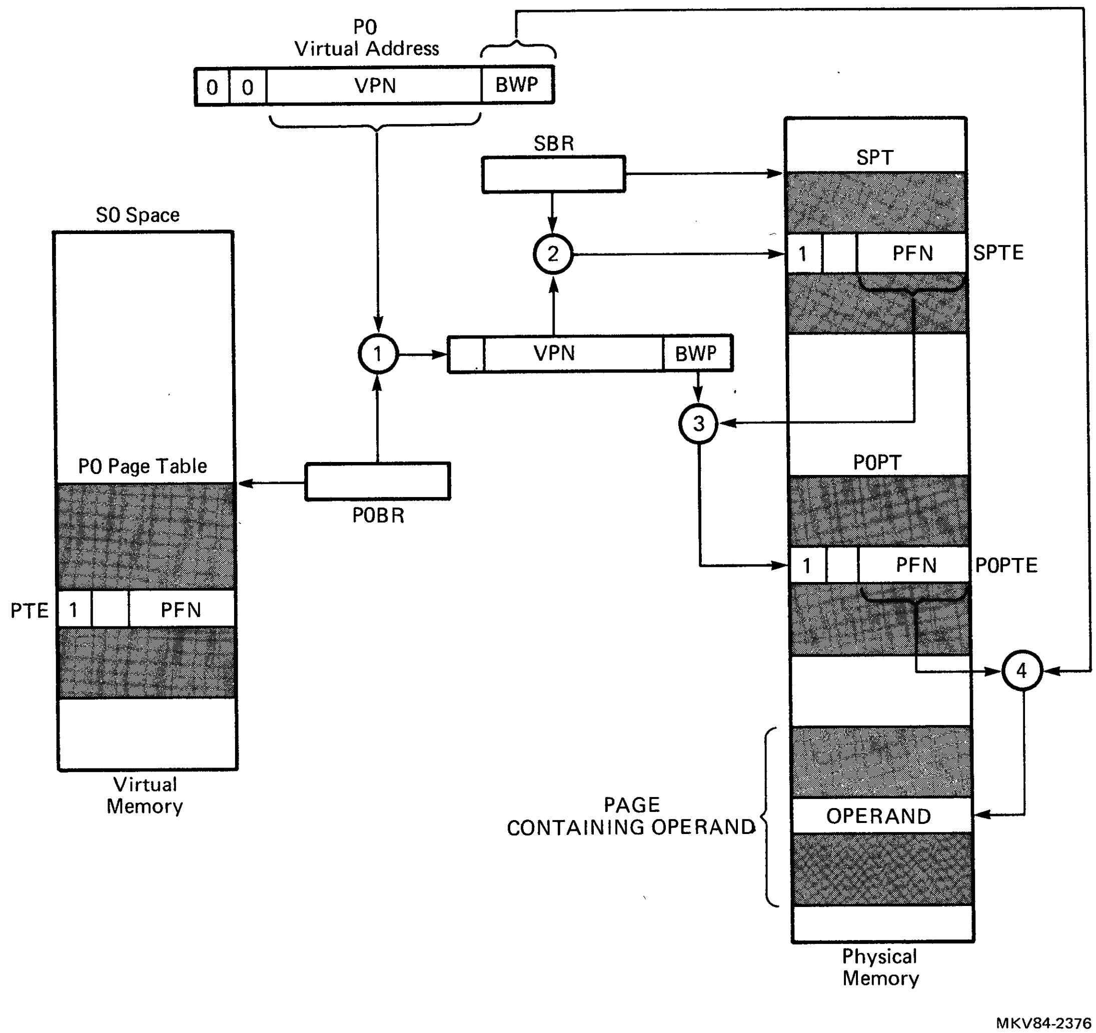


Figure 39 Process Virtual Address Translation

PHYSICAL ADDRESS SPACE

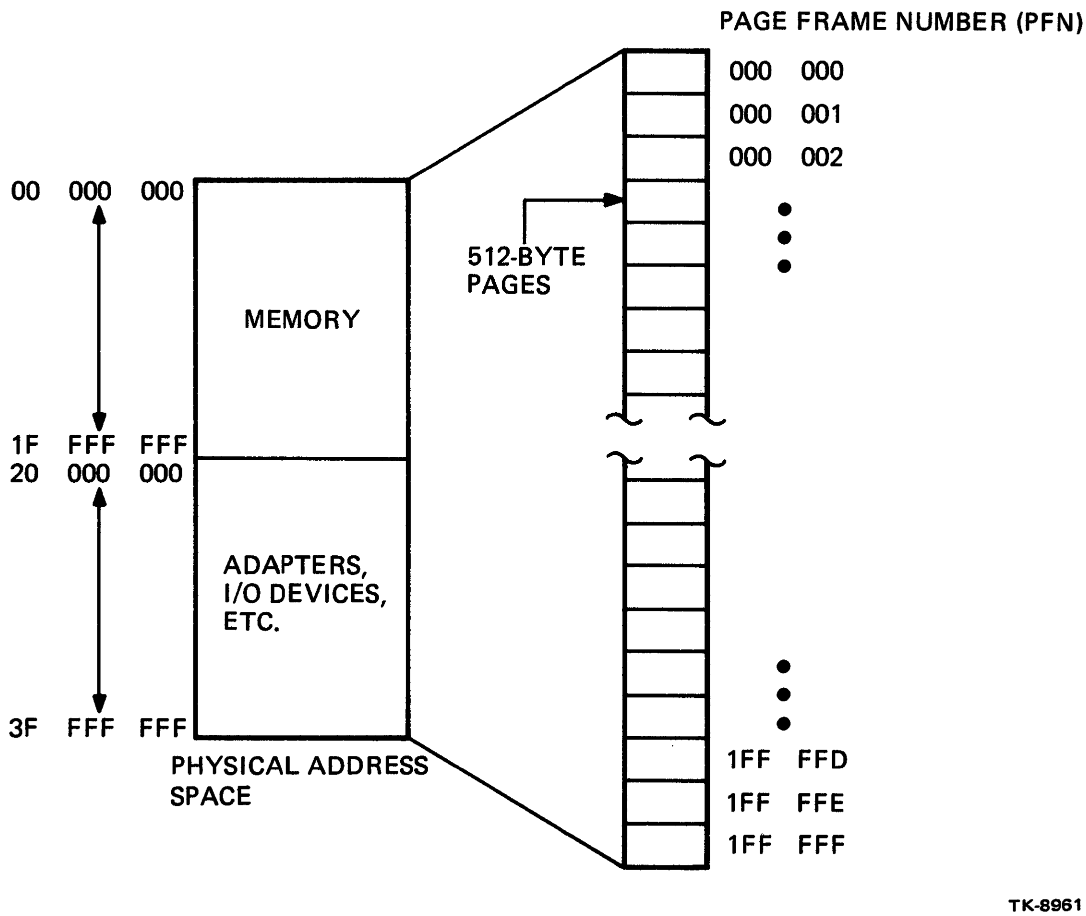


Figure 40 Physical Address Space

*SYSGEN -

PHYSICALPAGES

IMAGE ACTIVATOR AND PROCESS HEADER

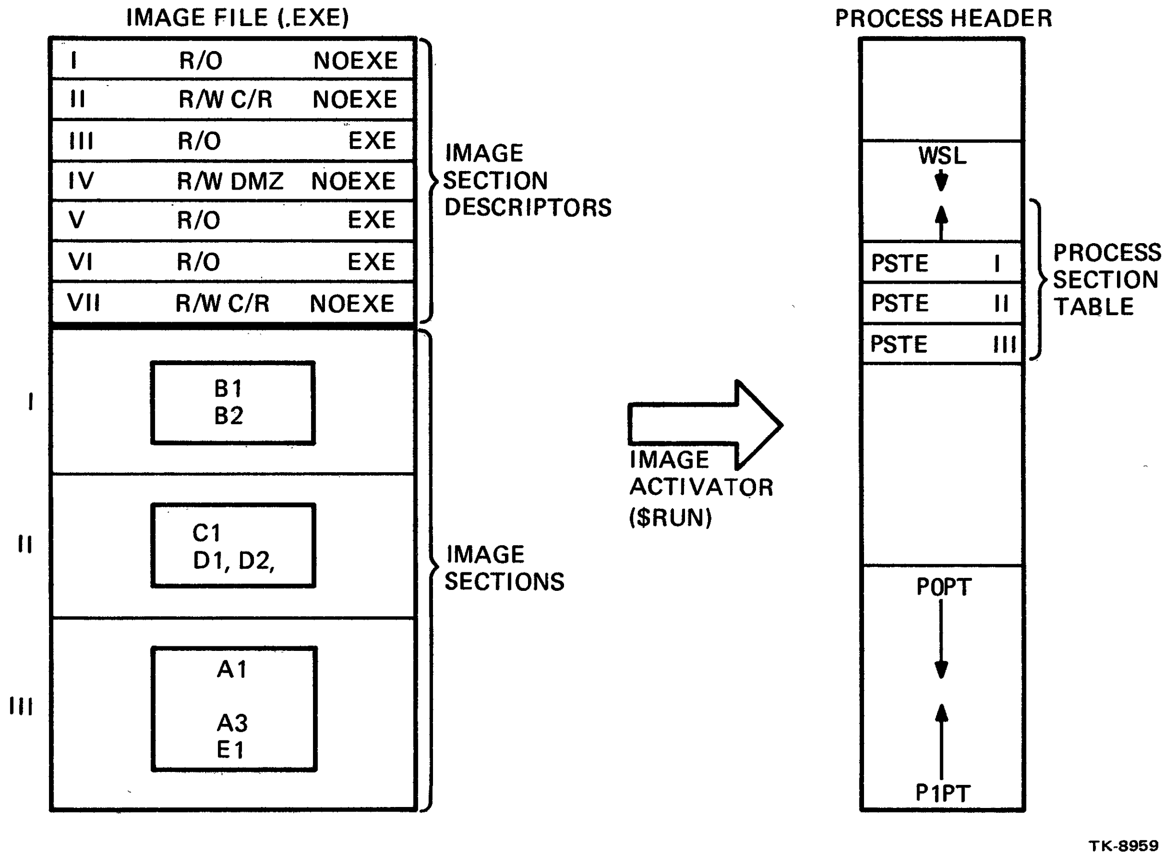


Figure 41 Image File and Process Header

Image Activator

- Fills in Process Section Table entries from the image section descriptors.
- Fills in the Page Table entries.
- Resolves any shared addresses.

PAGE READ CLUSTERING

Why Cluster Pages

- More efficient $QIO.
- Brings into working set pages that potentially will be referenced.

How a Cluster is Made

Pager scans successive PTEs for the same backing store address.

Examples:

- PSTX (from image file)
- Consecutive pagefile VBNs (in page file)
- Consecutive GPTEs with same GSTX (global section)

Pager scans until:

- No more WSLEs are available
- No physical pages available
- Page table page for PTE not valid
- Maximum cluster size reached

If no page can be clustered, previous PTEs are scanned using above rules.

Maximum Cluster Size Determination

Table 3 Cluster Sizes and Where They are Stored

Page	Cluster Size
Global Page Tables	1
Process Page Tables	PAGTBLPFC
Paging File Pages	PFL$B_PFC/PFCDEFAULT
Process, Global Sections	SEC$B_PFC/PFCDEFAULT

Changing/Controlling Cluster Size

- SYSGEN parameters
  - PFCDEFAULT
  - PAGTBLPFC
- PFC argument in $CRMPSC
- Linker option
  (cluster=cluster_name,[base_adr],[pfc],file_spec[,...]
- Inhibit all page read clustering
  - NOCLUSTER SYSGEN parameter

Swapping

INTRODUCTION

The swapper is a process. The code of the swapper is part of the system image and executes in kernel access mode in S0 space.

The swapper is responsible for memory management on a system-wide basis. While the pager is the component servicing the demands within a process, the swapper balances the demands for physical memory by all of the processes in the system and the pageable portion of the operating system. To accomplish this, three operations are performed by the swapper:

- Inswap and outswap
- Modified page writing
- Shrinking working sets

Inswap and outswap operations are transfers of entire working sets between memory and disk.

Outswapping operations typically release over 100 pages at a time, and provide a rapid way to replenish the free page list. Included in these transfers are:

- The P0 and P1 space pages that are memory-resident and valid
- The process headers (including the hardware context, accounting information, and all of the memory management data structures of the process).

The only information normally retained in physical memory after a process has been outswapped is found in data structures allocated from nonpaged dynamic memory, particularly the software process control block (PCB) and the job information block (JIB).

Modified page writing is also performed by the swapper process. When pages are needed, they are always allocated from the free page list.

Pages are provided for allocation by writing modified pages to their backing storage locations and then inserting the pages on the free page list.

Before the swapper outswaps a process, it will attempt to replenish the free page list by recovering pages from a process by shrinking the working set.

The swapper is also involved in both process creation and system initialization.

OBJECTIVES

To properly manage system-wide memory usage, the student must be able to:

1. Explain why swapping and paging are both implemented in VAX/VMS.

2. Describe the swapping operation (inswap/outswap, handling I/O in progress, and global pages).

3. Discuss the effects of altering SYSGEN parameters relating to swapping.

RESOURCES

Reading

1. VAX/VMS Internals and Data Structures, chapter on swapping

Additional Suggested Reading

1. VAX/VMS Internals and Data Structures, chapters on memory management data structures, paging dynamics, and memory management system services

Source Modules

Facility Name	Module Name
SYS	PAGEFILE SWAPPER OSWPSCHED SDAT SHELL WRTMFYPAG IOCIOPOST SYSUPDSEC

TOPICS

I. Comparison of Paging and Swapping

II. Overview of the Swapper, the System-Wide Memory Manager

III. Maintaining the Free Page Count

A. Write modified pages

B. Shrink working sets

C. Outswap processes

IV. Waking the System-Wide Memory Manager

V. Outswapping a Process

A. Swap files

B. Scatter/gather

C. Partial outswaps

VI. Inswapping a Process

COMPARISON OF PAGING AND SWAPPING

Table 1 Differences Between Paging and Swapping

	Paging	Swapping
Function	Supports programs with very large address spaces.	Supports a large number of concurrently active processes.
Implementation	Moves pages into and out of process working sets.	Moves entire processes into and out of physical memory.
How Invoked	Exception service routine that executes in the context of the process that incurred the page fault.	Separate process that is awakened from its hibernating state by components that detect a need for swapper activity.
Unit	The page	A process working set

Similarities

1. The pager and swapper work from a common database.
2. The pager and swapper do conventional I/O.
3. Both components attempt to maximize the number of blocks read or written with a given I/O request.

OVERVIEW OF THE SWAPPER, THE SYSTEM-WIDE MEMORY MANAGER

- Description of Code
  - Located in S0 space
  - Separate process
  - Part of system image (SYS.EXE)
  - Executes in kernel mode only
- Primary function is to control memory for the entire system through:
  - Modified page writing
  - Shrinking of working sets
  - Inswapping and outswapping of working sets
- Also involved in process creation
  - Swaps in SHELL of a process
- One-time initialization code executes during system initialization
  - Creates SYSINIT process

Swapper - Main Loop

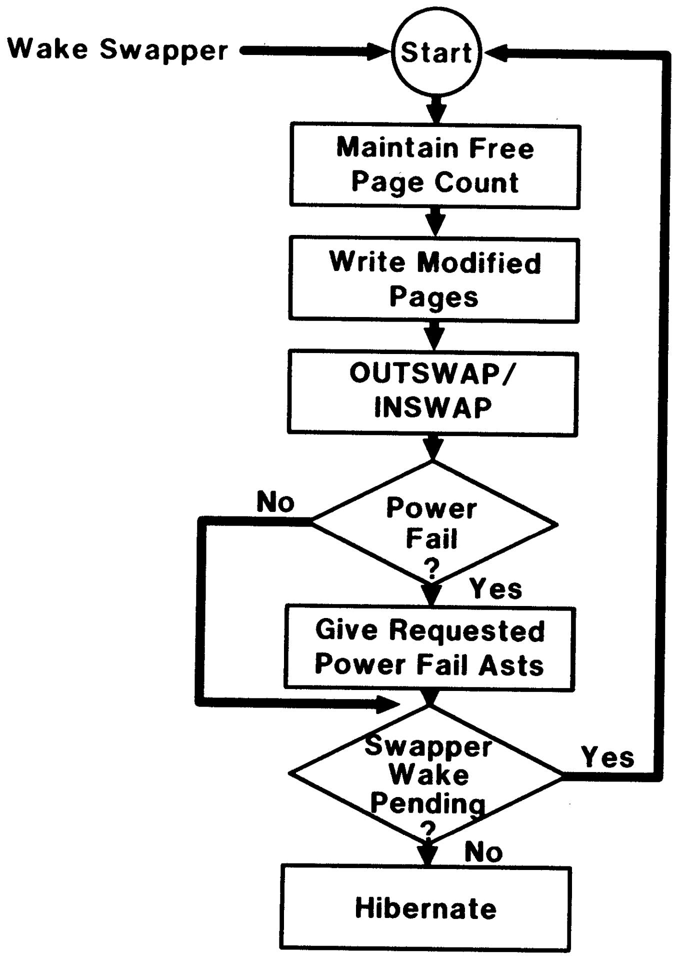


Figure 1 Swapper - Main Loop

*SYSGEN -

FREELIM
MPW_HILIMIT
MPW_LOLIMIT

MAINTAINING THE FREE PAGE COUNT

To maintain at least FREELIM free pages, swapper will attempt to:

1. Reclaim pages from deleted process headers
2. Write modified pages
   - If modified page write alone will satisfy need for free pages
     - If (FREEGOAL minus number on free list) ≤ (number on modified list minus MPW_LOLIMIT)
     - If at least MPW_THRESH pages on modified list
   - Will stop writing at MPW_LOLIMIT
3. Shrink working sets to SWPOUTPGCNT pages
4. Outswap processes

*SYSGEN -

FREEGOAL
FREELIM
MPW_LOLIMIT
MPW_THRESH
SWPOUTPGCNT

get a handle on this!

How Modified Page Writer Gathers Pages

inserted to SWAPPER PT

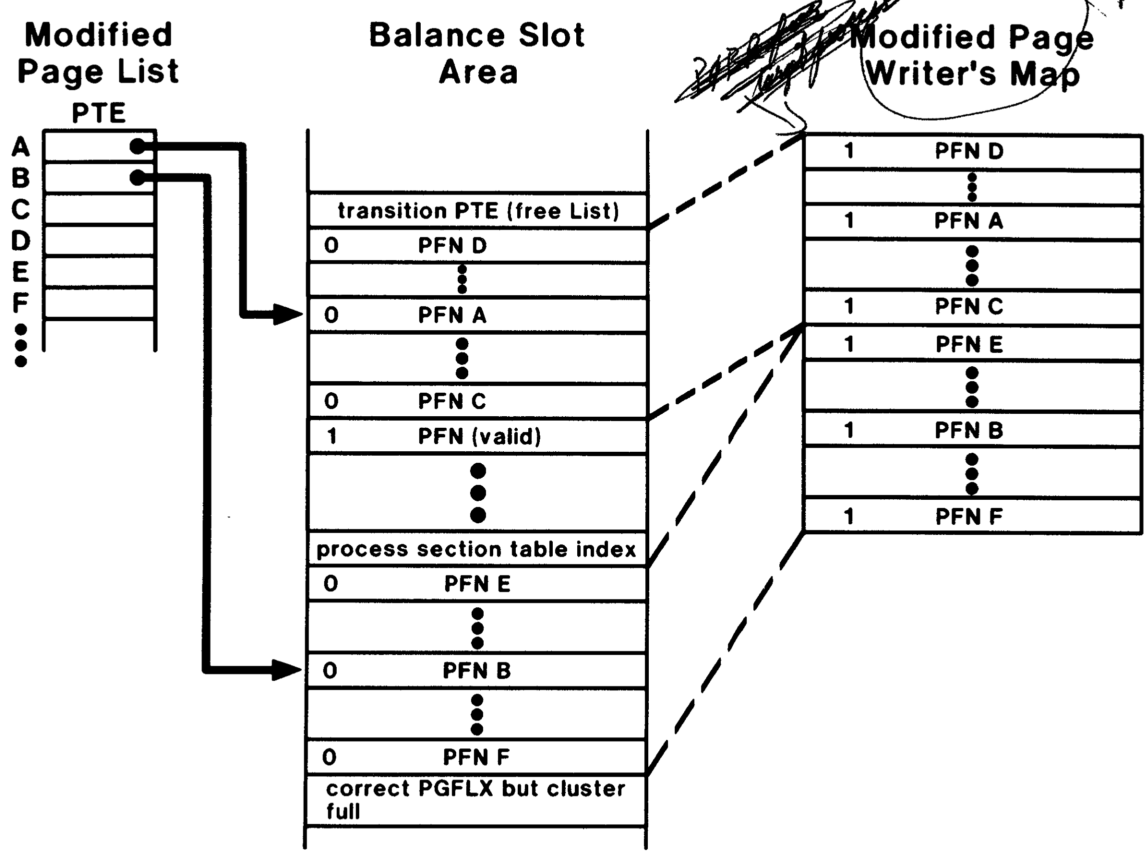


Figure 2 How Modified Page Writer Gathers Pages

Gathers pages around selected PTE from modified pages list until PTE is:

- Free page PTE
- Valid PTE
- PSTX in PTE
- PGFLX in PTE but cluster is full

*SYSGEN -

MPW_WRTCLUSTER

Modified Page Write Clustering

- Scans PTEs in reverse order from page read clustering
- Can write clusters to
  - Page file
  - Image file
  - Mapped data file
- If SWPVBN > 0, page going to swap file, no clustering.
- When building clusters
  - Cluster size determined by SYSGEN parameter MPW_WRTCLUSTER
  - Scan terminated if:
    1. PTE indicates page not on modified page list
    2. PTE points to page in shared memory, or page mapped by PFN
    3. PSTX or GSTX does not match that of first PTE in scan
- When writing to page file
  - Build up several mini-clusters into one larger cluster
  - Use one I/O to write larger cluster to disk
  - Note that on later page read, mini-clusters may be read separately.

Trimming and Swapping Working Sets

- If modified page writing does not gain enough free pages, swapper will trim and swap working sets.
- Outswap is expensive, so try trimming first
- Swapper uses a table for deciding which processes to trim and swap (in module OSWPSCHED)
- Processes on the system are divided into groups depending on
  - Their scheduling state
  - Special swapper considerations, for example, whether or not

    Experienced initial quantum
    Have direct I/O in progress
- General steps for trimming and swapping:
  - Trim all processes in all groups to WSQUOTA (call in "loans")
  - For each group of processes:

    Perhaps trim each process to SWPOUTPGCNT
    Then outswap each process
- Swapper goes on to next task when sufficient free pages on free page list

Table 2 Order of Search for Trim and Swap Candidates

Group	Process State	Special Flags for Swapper
I	SUSP	SWAPASAP
II	COM	DORMANT, SWAPASAP, COMPUTE
III	HIB LEF	LONGWAIT, SWPOGOAL NDIOCNT, LONGWAIT, SWPOGOAL
IV	CEF	NDIOCNT, SWPOGOAL, CEF
V	HIB LEF	SHORTWAIT, SWPOGOAL NDIOCNT, SHORTWAIT, SWPOGOAL
VI	FPG COLPG	PRIORITY PRIORITY
VII	MWAIT	
VIII	CEF LEF	PRIORITY, INQUAN, DIOCNT, CEF PRIORITY, INQUAN, DIOCNT
IX	PFW COM	PRIORITY, INQUAN INQUAN, COMPUTE

dormant process ≜ below DEFPRI (or =) & not scheduled for DORMANT.WAIT [illegible]

Table 3 Description of Special Swapper Flags

Flag	Description
CEF	Marks the CEF state (queues are different from other wait states)
COMPUTE	Marks the computable (COM) state
DIOCNT	Process must have nonzero direct I/O count
DORMANT	Only consider process if dormant
INQUAN	Ignore process if PCB$V_INQUAN is set
LONGWAIT	Only consider processes in a long wait
NDIOCNT	Process must have zero direct I/O count
PRIORITY	Observe priority of process relative to inswap candidate
SHORTWAIT	Only consider processes in a short wait
SWAPASAP	Swap process right after trimming to WSQUOTA
SWPOGOAL	Reduce process past WSQUOTA before swapping

- Characteristics of a DORMANT process
  - Computable state
  - Current priority less than or equal to DEFPRI
  - Has not had a significant event in DORMANTWAIT seconds

*SYSGEN -

DEFPRI
DORMANTWAIT
LONGWAIT
SWPFAIL

Expanding and Shrinking Working Sets

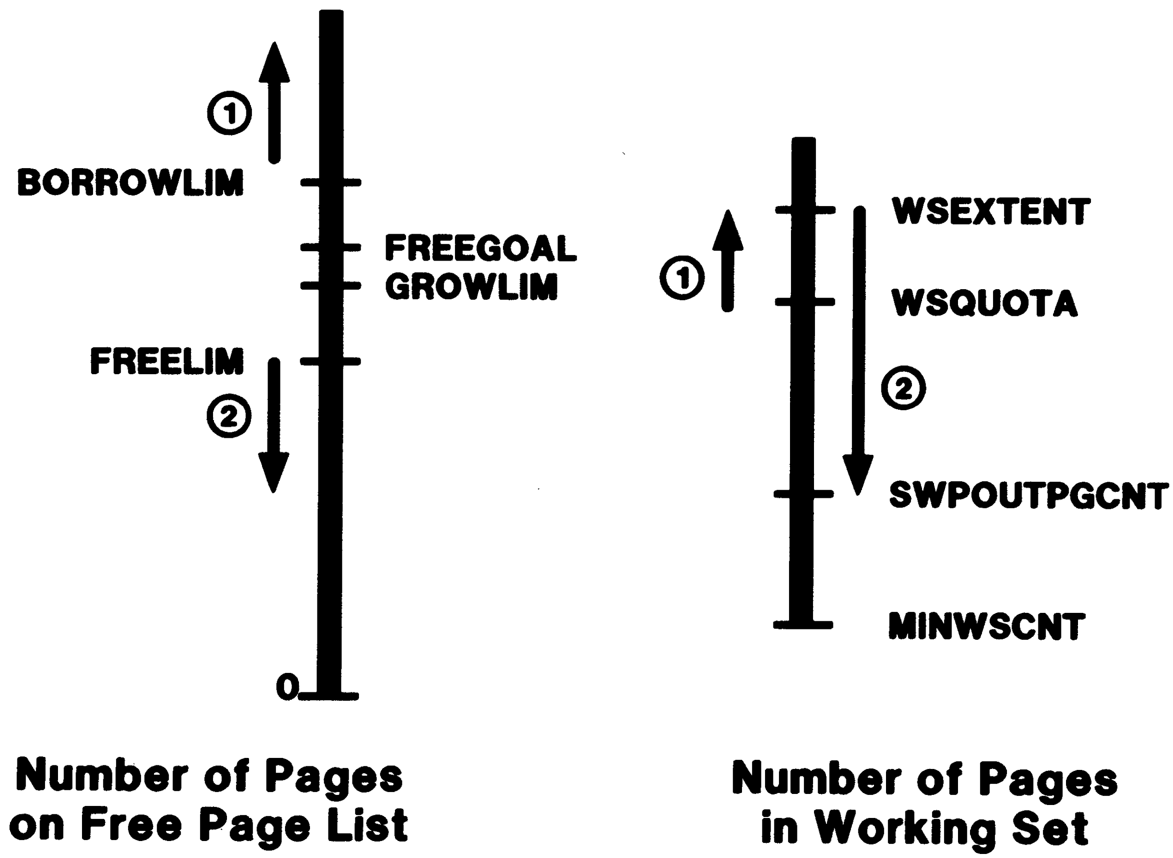


Figure 3 Expanding and Shrinking Working Sets

1. If free page count > BORROWLIM, working set may grow past WSQUOTA to WSEXTENT.

2. If free page count < FREELIM, swapper will attempt to

   - Shrink working sets from WSEXTENT to WSQUOTA

   - Shrink working sets from WSQUOTA to SWPOUTPGCNT

WAKING THE SYSTEM-WIDE MEMORY MANAGER

Table 4 Selected Events That Cause the Swapper or Modified Page Writer to Be Awakened

Event	Module	Additional Comments
Process that is outswapped becomes computable	RSE	Swapper will attempt to make this process resident
Quantum end	RSE	Outswap previously blocked by initial quantum flag setting may now be possible
CPU time expiration	RSE	Process may be deleted, allowing previously blocked inswap to occur
Modified page list exceeds upper limit threshold	ALLOCPFN	Modified page writing is performed by swapper
Free page list drops below low limit threshold	ALLOCPFN	Swapper must balance free page count
Balance slot of deleted process becomes available	SYSDELPRC	Previously blocked inswap may now be possible
Process header reference count goes to zero	PAGEFAULT	Process header can now be outswapped to join previously outswapped process body
System timer subroutine executes	TIMESCHDL	The swapper is potentially awakened every second if there is any work for it to do

OUTSWAPPING A PROCESS

- Outswap a process for two reasons
  - Reclaim free pages
  - Free up a balance slot
- Processes are moved from physical memory to the swap file (or paging file)
- Outswap a process in two stages
  - Process body (P0 and P1 pages)
  - Process header
- More difficult to outswap process header

Outswap Rules

Table 5 Rules for Scan of Working Set List on Outswap

Type of Page	Valid	Action of Swapper for This Page
1. Process Page	Valid	Outswap page. If there is outstanding I/O and the page is modified, load SWPVBN array element with block in swap/page file where the updated page contents should be written when the I/O completes.
2. System Page		Impossible for system page to be in process working set. Swapper generates an error.
3. Global Read-Only	Valid	If SHRCNT = 1, then outswap. If SHRCNT > 1, DROP from working set. It is highly likely that process can fault page later without I/O. This check avoids multiple copies of same page in swap page file.
4. Global Read/Write		DROP from working set. It is extremely difficult to determine whether page in memory was modified after this copy was written to the swap page file.
5. Page Table Page		Not part of process body. However, while scanning process body, VPN field in WSL is modified to reflect offset from beginning process header because page table pages will probably be located at different virtual addresses following inswap.

The scan of the working set list on outswap is keyed off a combination of the physical page type (WSL<3:1>) and the valid bit (PTE<31>).

Locating Disk Files for Swap

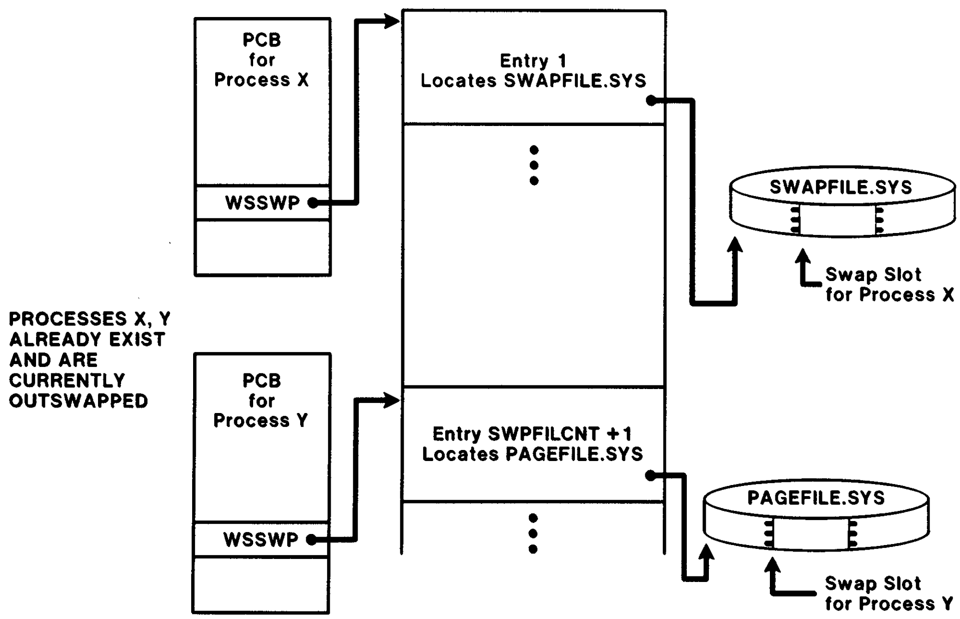


Figure 4 Locating Disk Files for Swap

- Choice of swap file or page file is determined by a field in the PCB called WSSWP
- Swap slots are assigned dynamically in increments of SWPALLOCINC, up to WSQUOTA pages
- If swapping is occurring on system, more efficient to swap file than page file
  - If sufficient memory, may not need separate swap file
  - On small systems, may not want separate swap file

*SYSGEN -

SWPFILCNT
PAGFILCNT
MPW_WRTCLUSTER
SWPALLOCINC

How Swapper's P0 Page Table is Used to Speed Swap I/O

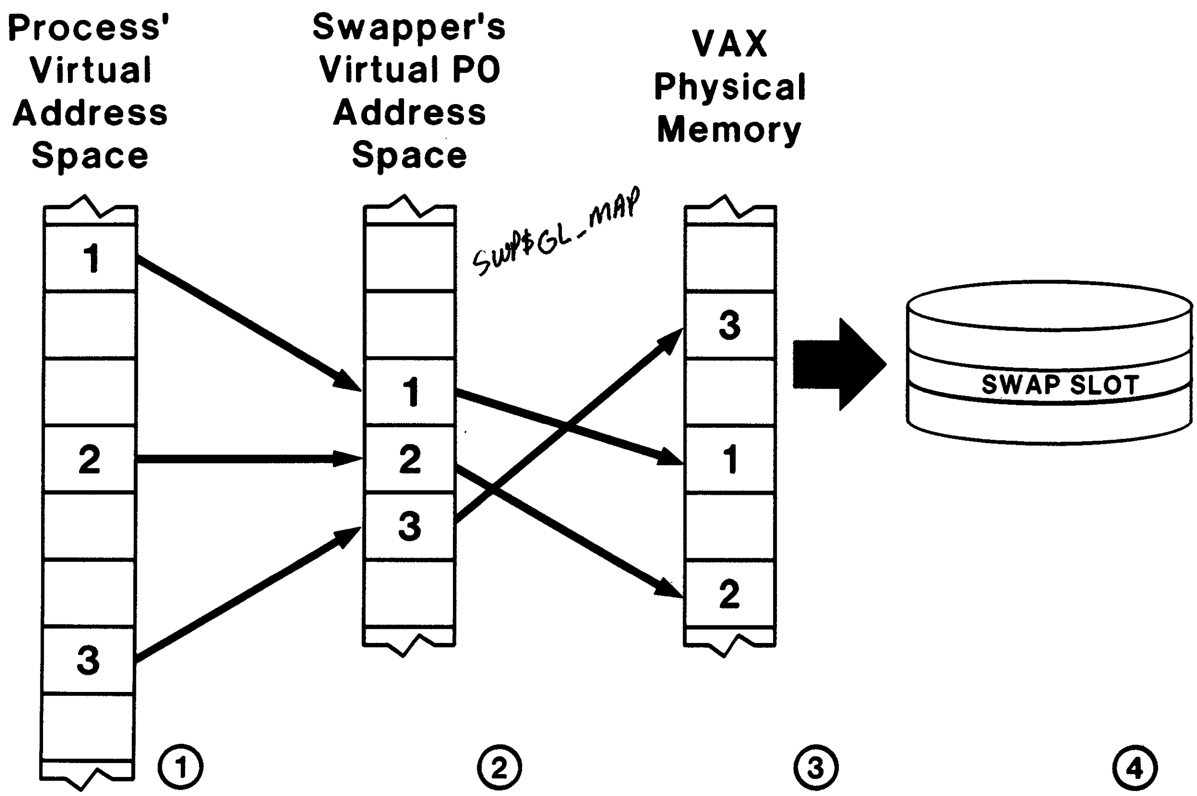


Figure 5 How Swapper's P0 Table is Used to Speed Swap I/O

1. Working set pages usually virtually discontiguous in process address space.

2. Mapped to virtually contiguous addresses in swapper's P0 space.

3. Both virtual pages correspond to same PFNs in physical memory.

4. $QIO on swapper's contiguous virtual addresses --> one I/O to disk (QIO issued with base virtual address and byte count).

Swapper's Pseudo Page Tables

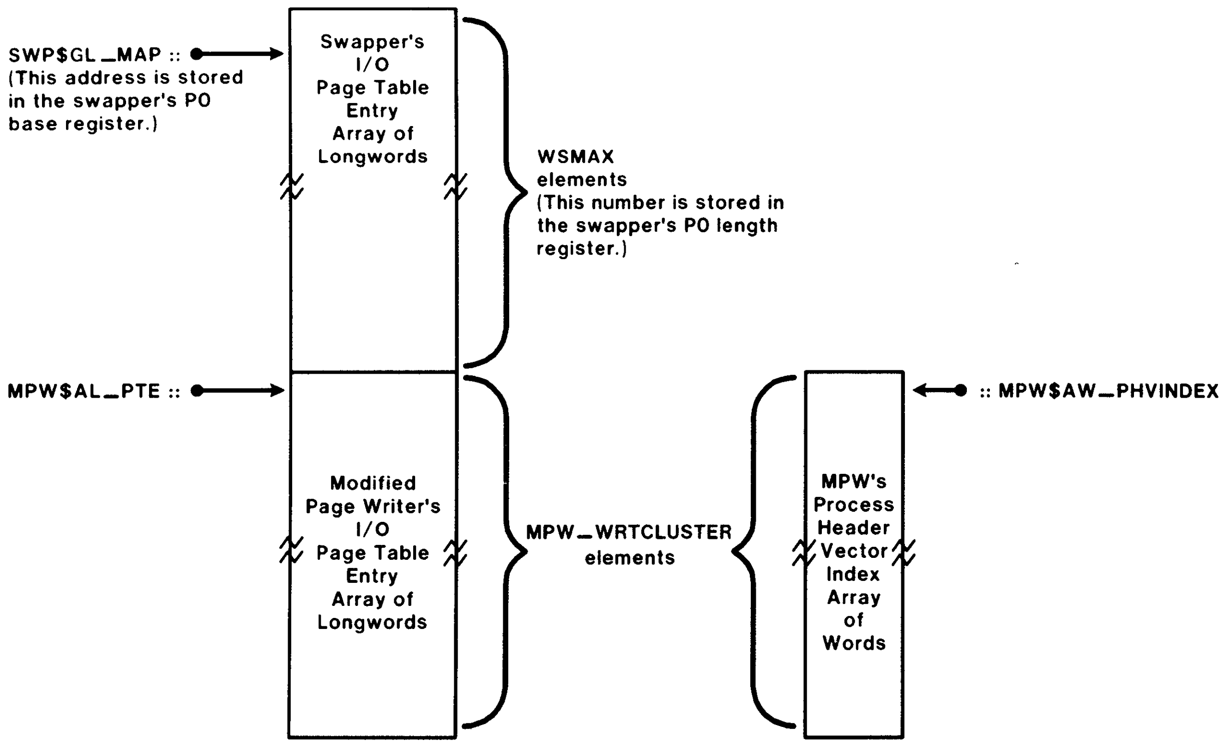


Figure 6 Swapper's Pseudo Page Tables

Swapper can have one swap I/O and one modified page write I/O in progress at the same time.

V5 has made modified page writer asynchronous

*SYSGEN -

MPW_PRIO
SWP_PRIO

Partial Outswaps and the Process Header

- In partial outswap, process body outswapped, process header remains resident
- Reason for partial outswap - pages locked in memory
  - $LCKPAG or direct I/O
  - Locked pages and PHD stay in memory
- Note that $LKWSET has no effect on PHD being outswapped
- Effects of partial outswap
  - Balance slot still occupied, preventing another process getting inswapped (BALSETCNT = maximum number of resident processes, MAXPROCESSCNT >= BALSETCNT)
  - On inswap, if PHD still resident, only process body is inswapped, so process page tables are rebuilt, but not system page table entries mapping PHD.
- PHD size depends on
  - PHD$K_LENGTH
  - WSMAX
  - PROCSECTCNT
  - VIRTUALPAGECNT

INSWAPPING A PROCESS

- Inswap is the opposite of outswap
- Inswap in two stages
  - Process header
  - Process body
- Only inswap from computable outswapped (COMO) state

INSWAP RULES

Table 6 Rules for Rebuilding the Working Set List and the Process Page Tables at Inswap

Type of Page Table Entry	Action of Swapper for this Page
1. PTE is valid	Page is locked into memory and was never outswapped.
2. PTE indicates a transition page (probably due to outstanding I/O when process was outswapped)	Add transition page to process working set. Release duplicate page that was just swapped in.
3. PTE contains a global page table index (GPTX)	Swapper action is based on the contents of the global page table entry (GPTE).
(Page must be global read-only because global read/-write pages were dropped from the working set at outswap time)	a. If the global page table entry is valid, add the PFN in the GPTE to the process working set and release the duplicate page.
	b. If the global page table entry indicates a transition page, make the global page table entry valid, add that physical page to the process working set, and release the duplicate page.
	c. If the global page table entry indicates a global section table index, then keep the page just swapped in, and make that the master page in the global page table entry, as well as the slave page in the process page table entry.

Table 6 Rules for Rebuilding the Working Set List and the Process Page Tables at Inswap (Cont)

Type of Page Table Entry	Action of Swapper for this Page
4. PTE contains a page file index or a process section table index	This is the usual content for pages that did not have outstanding I/O or other page references when the process was outswapped. The PFN in the swapper map is inserted into the process page table. The PFN arrays are initialized for that page.

At inswap time the swapper uses the contents of the page table entry to determine what action to take for each particular page.

SUMMARY

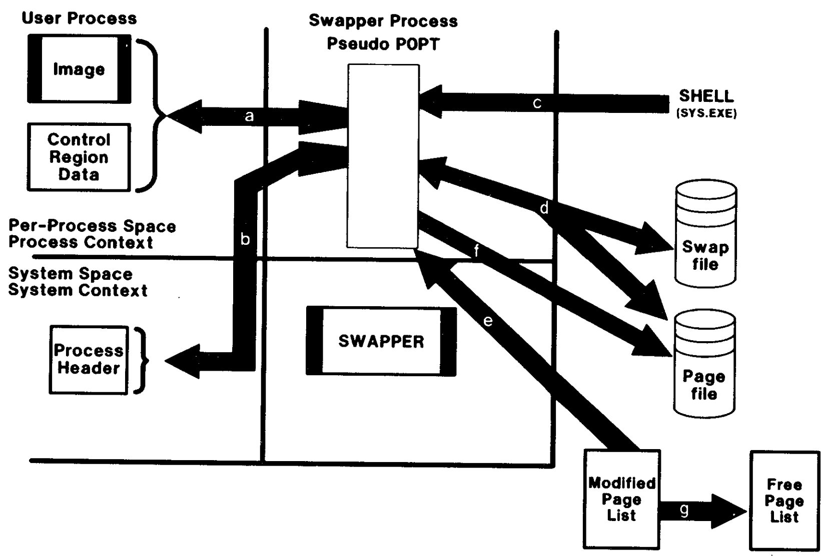


Figure 7 Overview of Swapper Functions

Outswap
- a P0 and P1 pages are "adopted" into swapper's P0 space
- d Process outswapped to swap file/page file
- b PHD pages are "adopted" into swapper's P0 space
- d PHD outswapped to swap file/page file

Inswap
- d Reverse of outswap

Modified Page Writing
- e Selected modified pages "adopted" to swapper's P0 space
- f Modified pages written to page file
- g "Modified" pages transferred to free page list

Process Creation
- c SHELL copied to swapper's P0 space
- a,b SHELL code and data transferred to P1 and PHD of new process

Table 7 SYSGEN Parameters Relevant to the Swapper

Function	Parameter
Number of pages required on free page list for processes to grow beyond WSQUOTA (checked at quantum end)	BORROWLIM
Base default process priority	DEFPRI
Real time that must elapse before swapper considers a COM process dormant	DORMANTWAIT
Swapper's goal for pages on free page list	FREEGOAL
Low threshold of free page list	FREELIM
Number of pages required on FPL for processes to grow beyond WSQUOTA (checked by the pager)	GROWLIM
Real time that must elapse before swapper considers a process temporarily idle	LONGWAIT
Upper limit for modified page list	MPW_HILIMIT
Low limit for modified page list	MPW_LOLIMIT
Priority of modified page write I/O	MPW_PRIO (*)
When regaining free pages, minimum number of pages on modified page list before swapper can write modified pages	MPW_THRESH (*)
Maximum number of pages in one MPW I/O	MPW_WRTCLUSTER
Maximum number of paging files	PAGFILCNT
Priority of swap I/O	SWP_PRIO (*)
Swap file allocation increment value	SWPALLOCINC (*)
Maximum number of processes swapper will skip when searching for a trim/swap candidate	SWPFAIL (*)
Maximum number of swap files	SWPFILCNT
Number of pages swapper tries to trim a process to before outswapping it	SWPOUTPGCNT

(*) = special parameter

APPENDIX
SWAPPER - MAIN LOOP

```
        .SBTTL  SWAPPER - MAIN LOOP

 ;++
 ; FUNCTIONAL DESCRIPTION:
 ;    THE MAIN LOOP OF THE SWAPPER IS EXECUTED WHENEVER THE SWAPPER IS AWAKENED
 ;    FOR ANY REASON.  EACH OF THE FUNCTIONAL ROUTINES WILL CHECK TO SEE IF
 ;    THEY HAVE ANY ACTION TO PERFORM.
 ;--

         .PSECT  $AEXENONPAGED         ; NON-PAGED PSECT
LOOP:    BSBB    BALANCE               ; BALANCE FREE PAGE COUNT
         BSBW    MMG$WRTMFYPAG         ; WRITE MODIFIED PAGES
         BSBW    SWAPSCHED             ; SCHEDULE SWAP
         TSTL    W^EXE$GL_PFATIM       ; CHECK FOR POWER FAIL TIME
         BEQL    15$                   ; BRANCH IF NO POWERFAIL
         JSB     EXE$POWERAST          ; GIVE ANY REQUIRED POWER FAIL ASTS
15$:     MOVL    W^SCH$GL_CURPCB,R4    ; GET PROPER PCB ADDRESS
         MOVAQ   W^SCH$GQ_HIBWQ,R2     ; AND ADDRESS OF WAIT QUEUE HEADER
         SETIPL  #IPL$_SYNCH           ; BLOCK SYSTEM EVENTS WHILE CHECKING
         BBSC    #PCB$V_WAKEPEN,PCB$L_STS(R4),20$ ; TEST AND CLEAR WAKE PENDING
         PUSHL   #0                    ; NULL PSL
         BSBW    SCH$WAITK             ; WAIT WITH STACK CLEAN
20$:     SETIPL  #0                    ; DROP IPL
         BRB     LOOP                  ; CHECK FOR WORK TO DO
         .DISABLE LSB
```

Example 1 Swapper - Main Loop

I/O Concepts and Flow

INTRODUCTION

When attempting to understand how input and output are handled under VMS, it is necessary to examine a series of sources including the code itself. Prior to reading the code, you must understand the steps that are taken to handle an I/O operation. Before reading the code, the names of the modules must also be known.

This module will illustrate and define the concepts and flow of how I/O is handled under VMS. It will also outline the pieces of VMS code that are involved in I/O, and how they are related.

OBJECTIVES

To trace a given I/O function through the VMS system, the student must be able to:

- Briefly describe the components of the I/O system, including system services, RMS, drivers, and XQPs.
- Describe the elements of, and uses of, the database maintained by the I/O system.

RESOURCES

Reading

- Guide to Writing a Device Driver for VAX/VMS, chapters on I/O overview and driver functions.
- VAX/VMS Internals and Data Structures, chapters on ' I/O System Services and device drivers.

Source Modules

Facility Name	Module Name
SYS	SYSQIOREQ IOCIOPOST
F11X	F11XQP

TOPICS

I. Overview of I/O Components and Flow

   A. Example of flow for $QIO request

II. Components of the I/O System

   A. RMS

   B. I/O system services

   C. XQPs, ACPs

   D. Device drivers

III. The I/O Database

   A. Driver tables

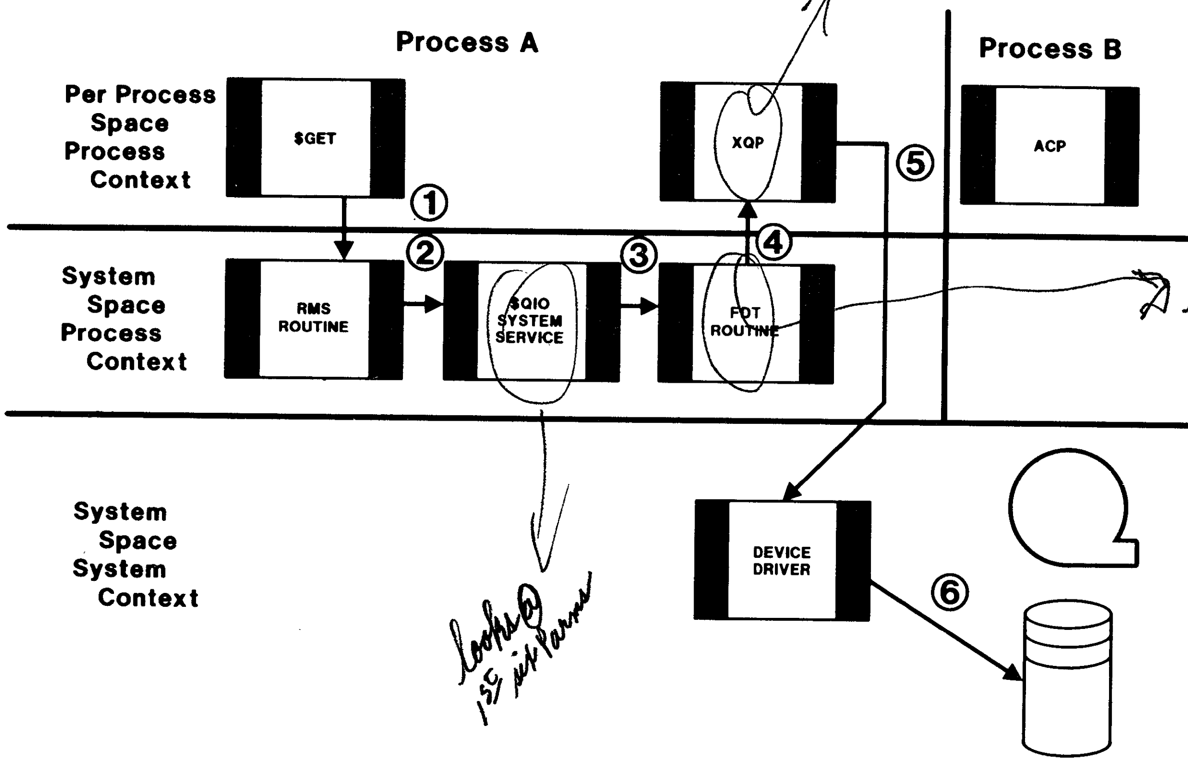
   B. IRPs

   C. Control blocks

IV. Methods of Data Transfer

OVERVIEW OF I/O COMPONENTS AND FLOW

runs as an AST in process context



looks @ 2nd le Parms

Figure 1 Input/Output Flow (Brief)

- Initiated by User Process
- Preprocessed by
  - RMS
  - $QIO
  - FDT (Driver-related routines) Function Decision Table
- ACP
  - Disk Structure (ODS-1)
  - Tape Structure
- XQP for ODS-2 Disks

AQB (from IRP's) forms worklist for device driver

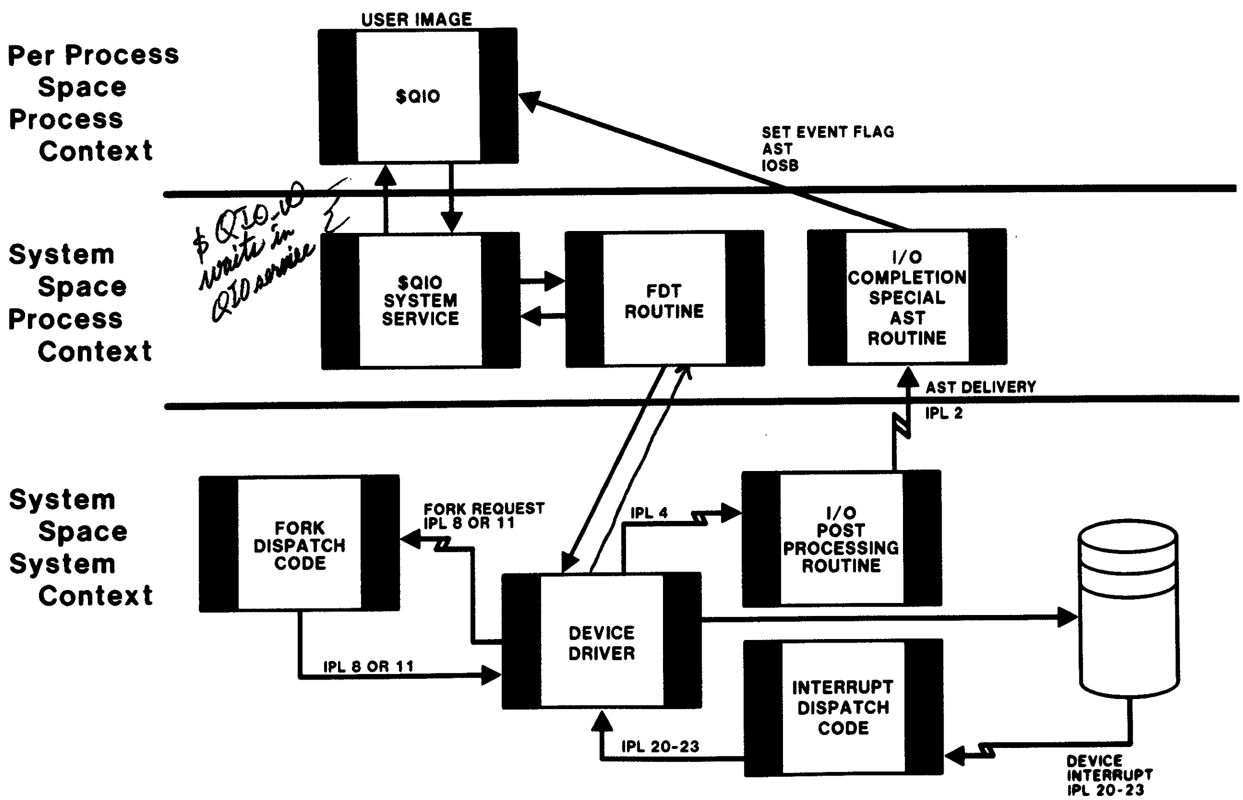


Figure 2 Input/Output (Full)

- Preprocessing done by RMS, $QIO and FDT routines
- Device control and data manipulation done by driver
- Final clean up done by I/O post and I/O completion routines

COMPONENTS OF THE I/O SYSTEM

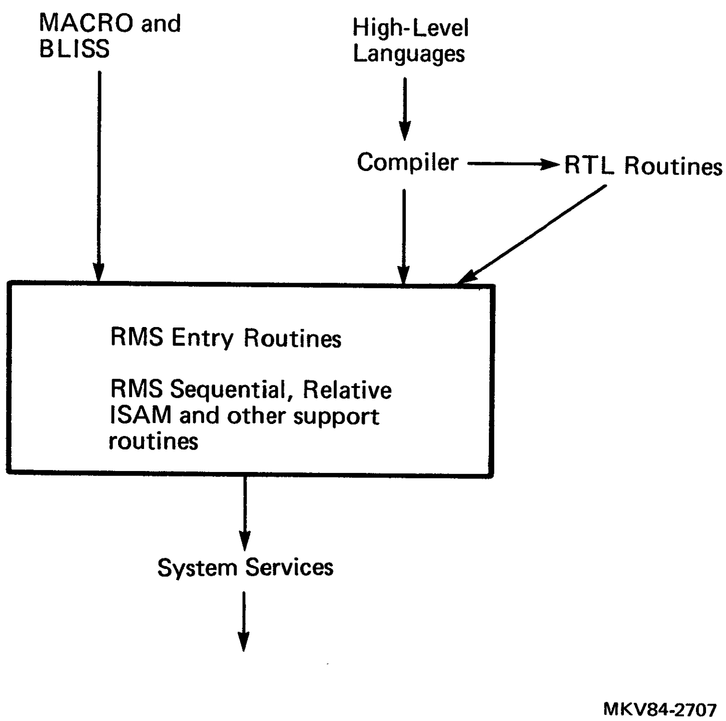


Figure 3 RMS Interfaces

- MACRO and BLISS programs can:
  - Call RMS directly
  - Access RMS data structures directly
- High-level language programs:
  - Use language I/O statements, which the compiler translates to RMS calls
  - Call Run-Time Library routines (which translate to RMS calls)
  - Most languages cannot access RMS data structures directly

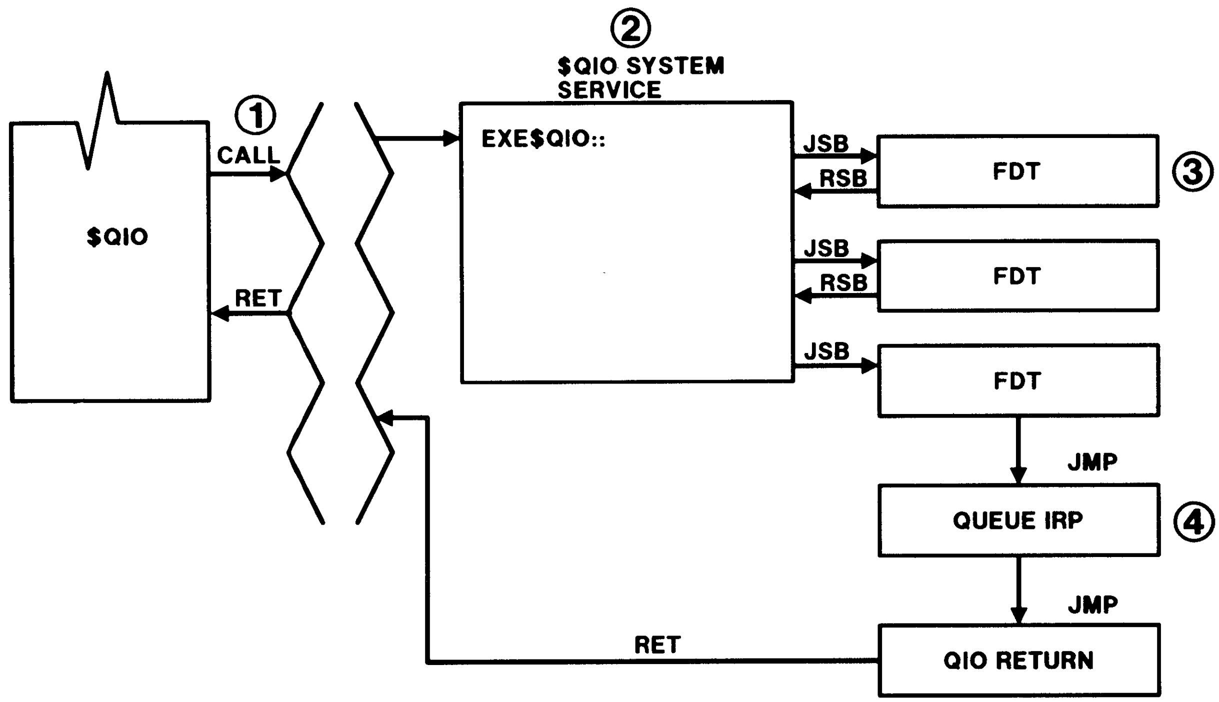


Figure 4 $QIO and FDT Routines

- $QIO
  - Handles device-independent $QIO parameters
  - Selects and JSB's to the proper FDT routines

- FDT Routines
  - Handle the device-dependent $QIO parameters
  - RSB to $QIO if other FDT routines are needed
  - JMP to system routines when FDT routines are finished

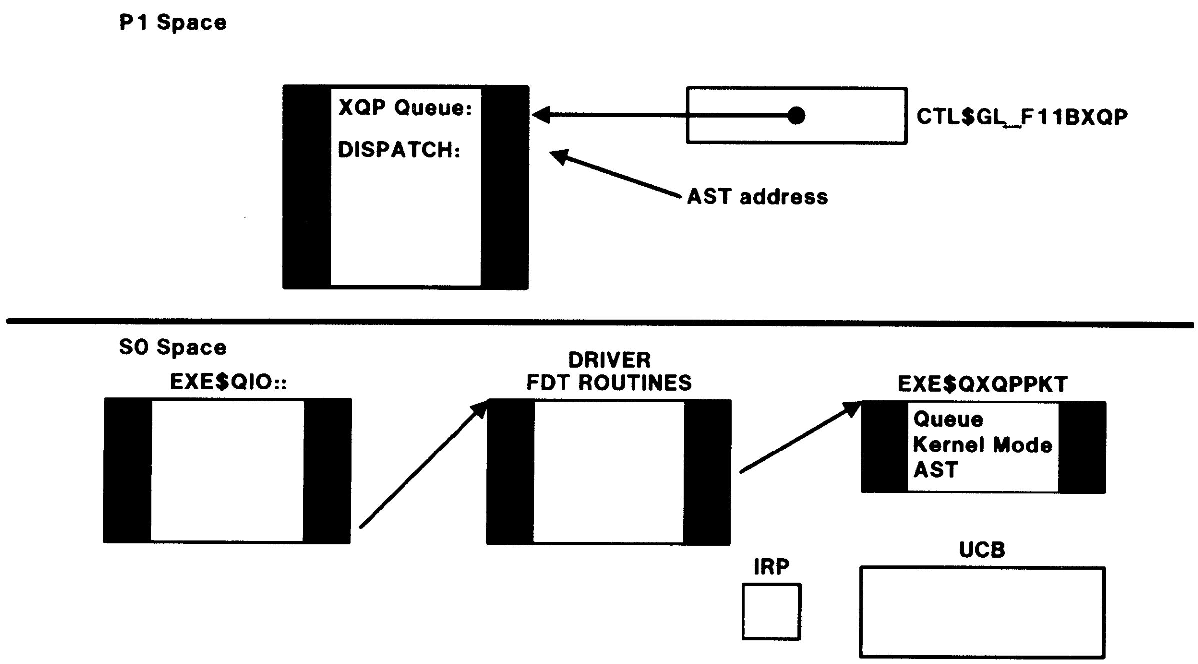


Figure 5 XQPs

- Reside in P1 space
- Are used in Process Context
- FDT routines JSB to EXE$QXQPPKT (AT IPL$_ASTDEL)
- EXE$QXQPACP invokes the XQP by means of an AST
- When finished, XQP queues IRP to the driver's Unit Control Block (UCB)

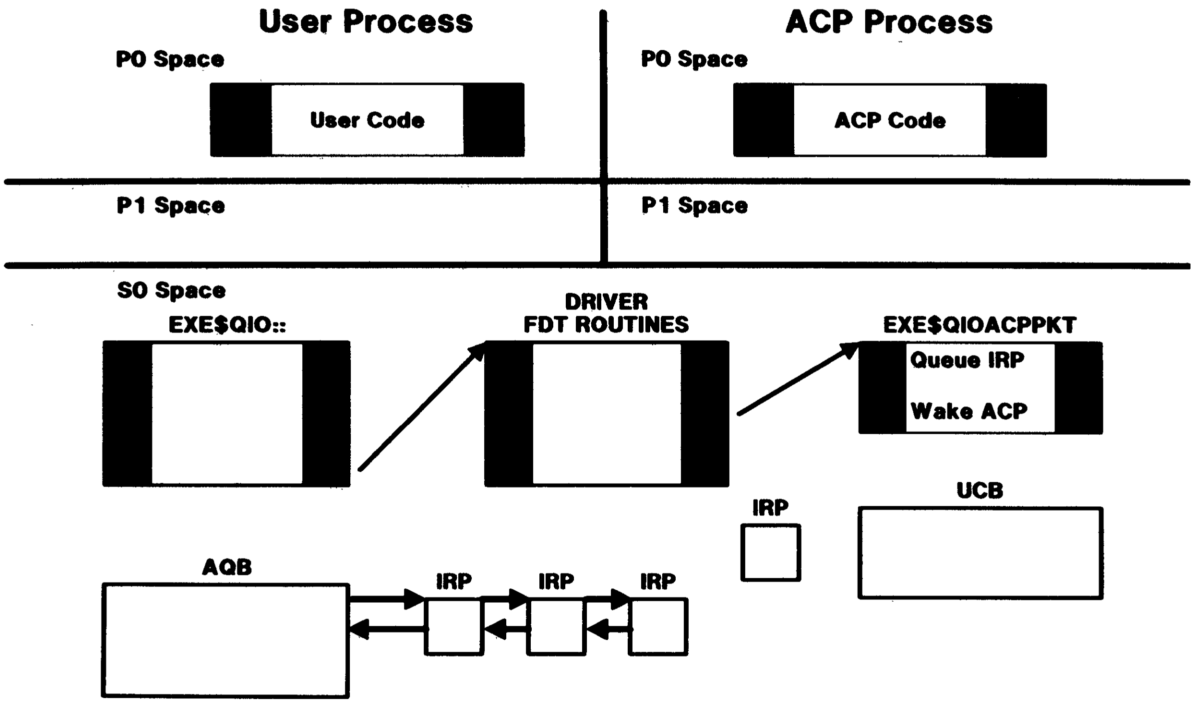


Figure 6 ACPs

- Are separate processes
- FDT routines JSB to EXE$QIOACPPKT (=< IPL$_SYNCH)
- EXE$QIOACPPKT queues IRP to ACP Queue Block (AQB) and wakes ACP
- When finished, ACP queues IRP to proper UCB

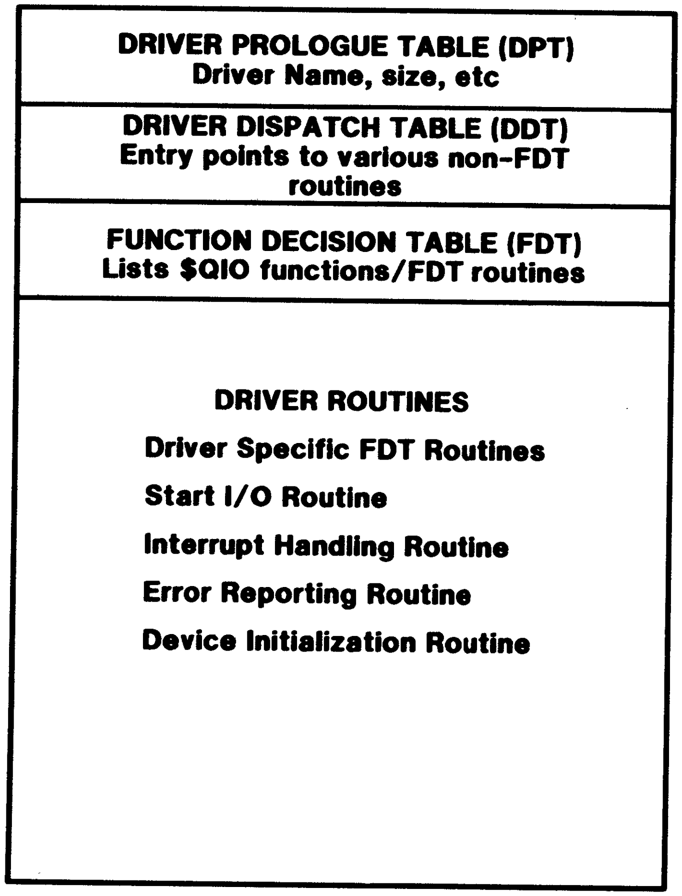


Figure 7 Components of Device Drivers

- Driver Tables
  - Used by SYSGEN to load in the driver and set up the tables
  - Used by VMS to get to the correct routine
  - Used by $QIO to find valid functions and their associated FDT routines

- Driver Routines
  - FDT routines written for this driver, called by $QIO
  - START I/O routine to initiate the I/O when proper
  - Interrupt routine for the driver
  - Error routine for reporting problems to VMS
  - Initialization routine on the device or the controller, called by VMS.

THE I/O DATABASE

Table 1 The I/O Database

Name	Function	Comments
IRP	Carries information for a specific I/O request	Created by $QIO in nonpaged pool
CCB	Links a 'channel' to a specific device unit	Created by $ASSIGN in P1 space
DDB	Contains information common to all devices on a controller	One per device type (one for DBA, etc.)
UCB	Contains information for a device unit. Used as a listhead for storage by the driver	One per device unit (one for DBA1:, etc.)
DDT	Contains entry point addresses for driver routines	Used by VMS to select the correct routine
FDT	Contains list of valid functions and their FDT routine addresses	Used by $QIO to select routines for the prope I/O functions
CRB	Contains information and listheads for a particular controller	Used especially by devices that share a controller (for exampl DBA1: and DBA2: share the controller DBA)
IDB	Contains information including a table of UCB addresses for units under a controller	Used by drivers and VM
ADP	Contains information including mapping registers and data paths	Used by drivers and VM

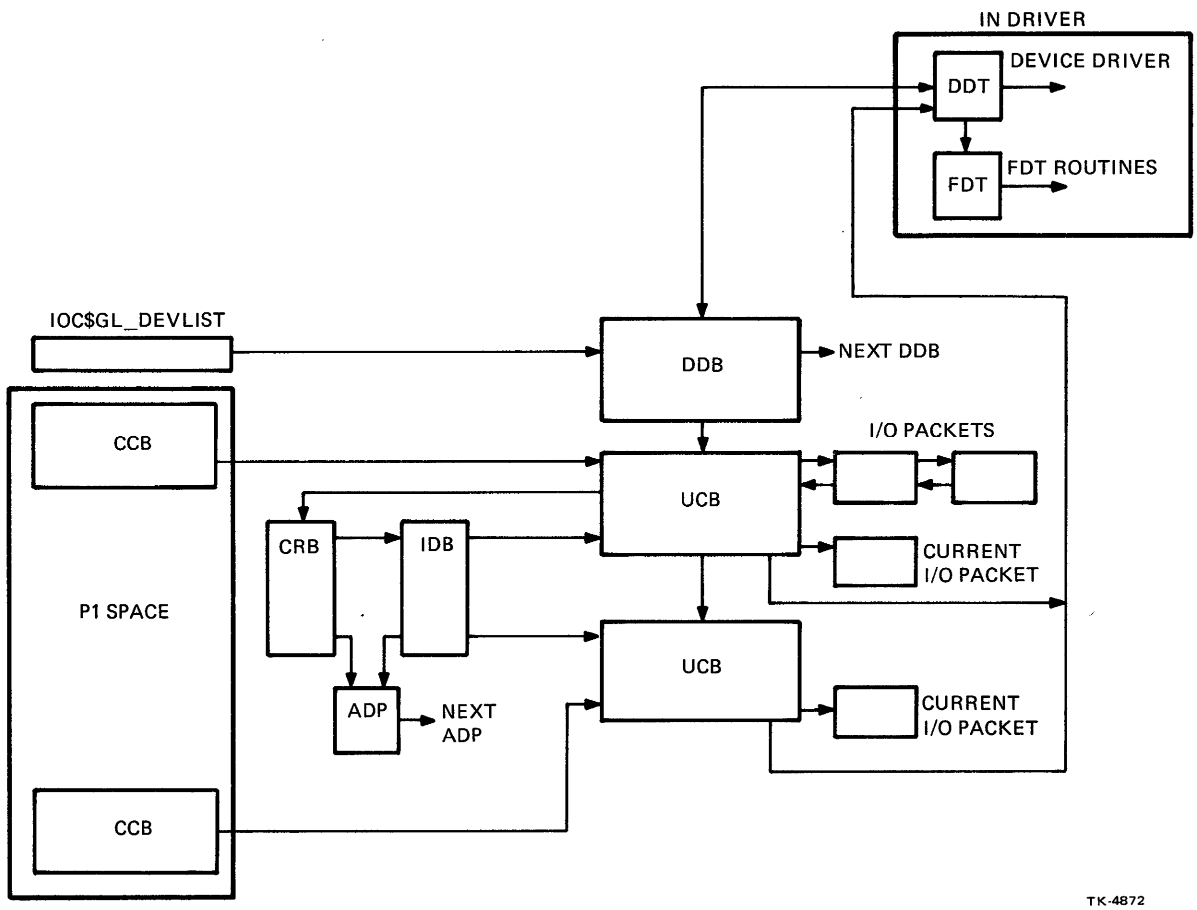


Figure 8 Summary Layout of I/O Database

METHODS OF DATA TRANSFER

- Buffered I/O
  - Data is transferred through an 'intermediate' buffer in S0 space.
  - User process is completely swappable
  - Normally used with slow devices (for example, terminals)

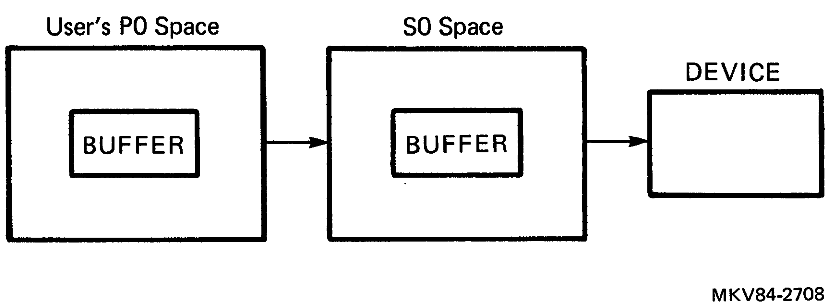


Figure 9 Buffered I/O

- Direct I/O
  - Data is transferred directly between the user's buffer and the device
  - Buffer pages are locked down until I/O completes

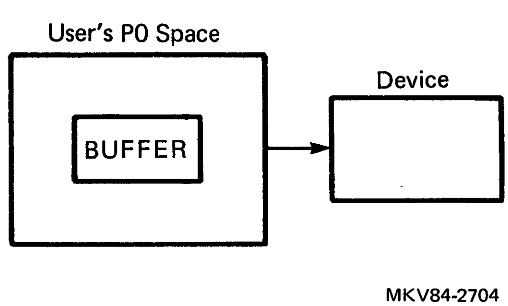
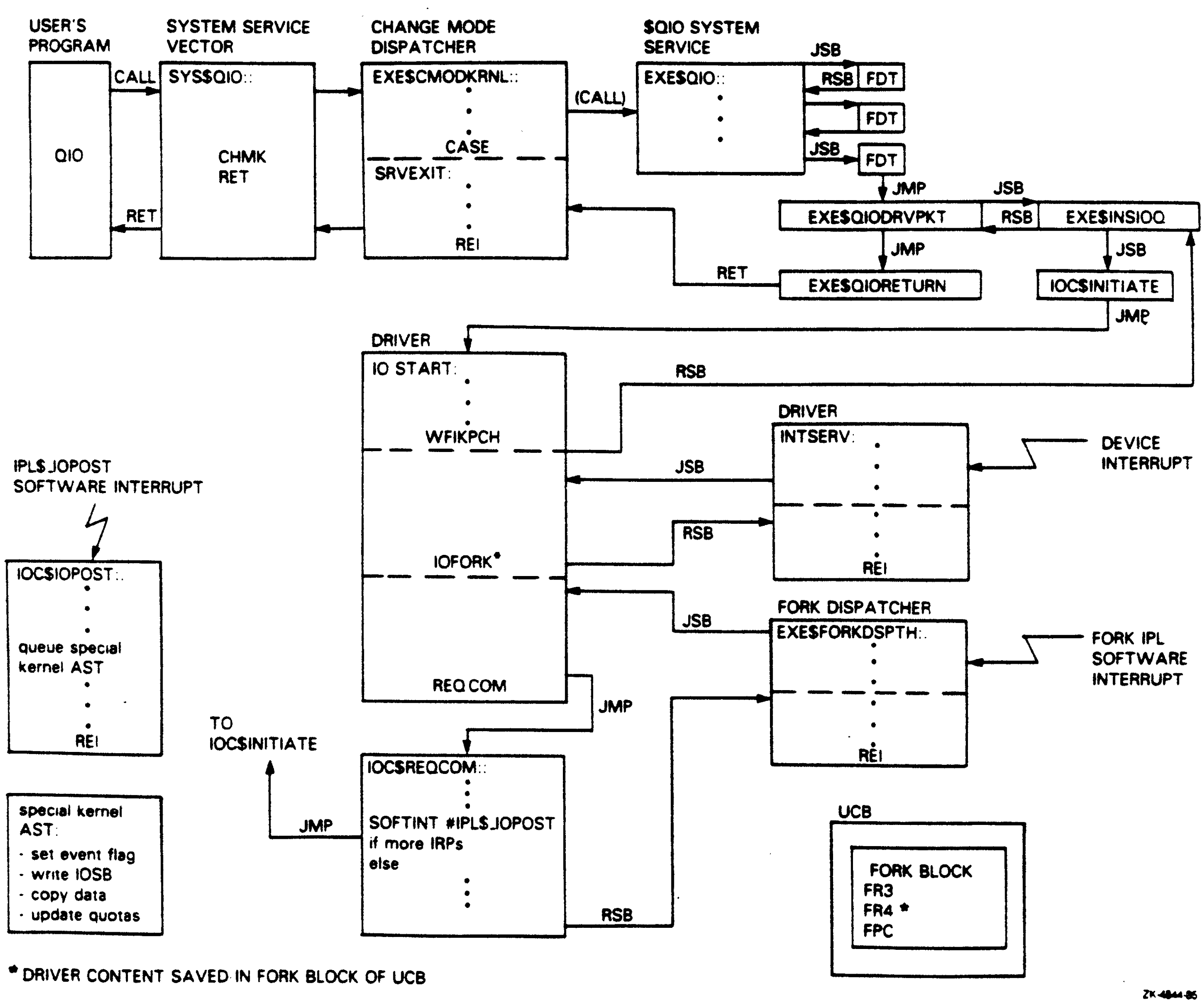


Figure 10 Direct I/O

SUMMARY

- Overview of I/O components and flow
  - Example of flow for $QIO request
- Components of the I/O system
  - RMS
  - I/O system services
  - XQPs, ACPs
  - Device drivers
- The I/O database
  - Driver tables
  - IRPs
  - Control blocks
- Methods of data transfer

Detailed Sequence of VAX/VMS I/O Processing

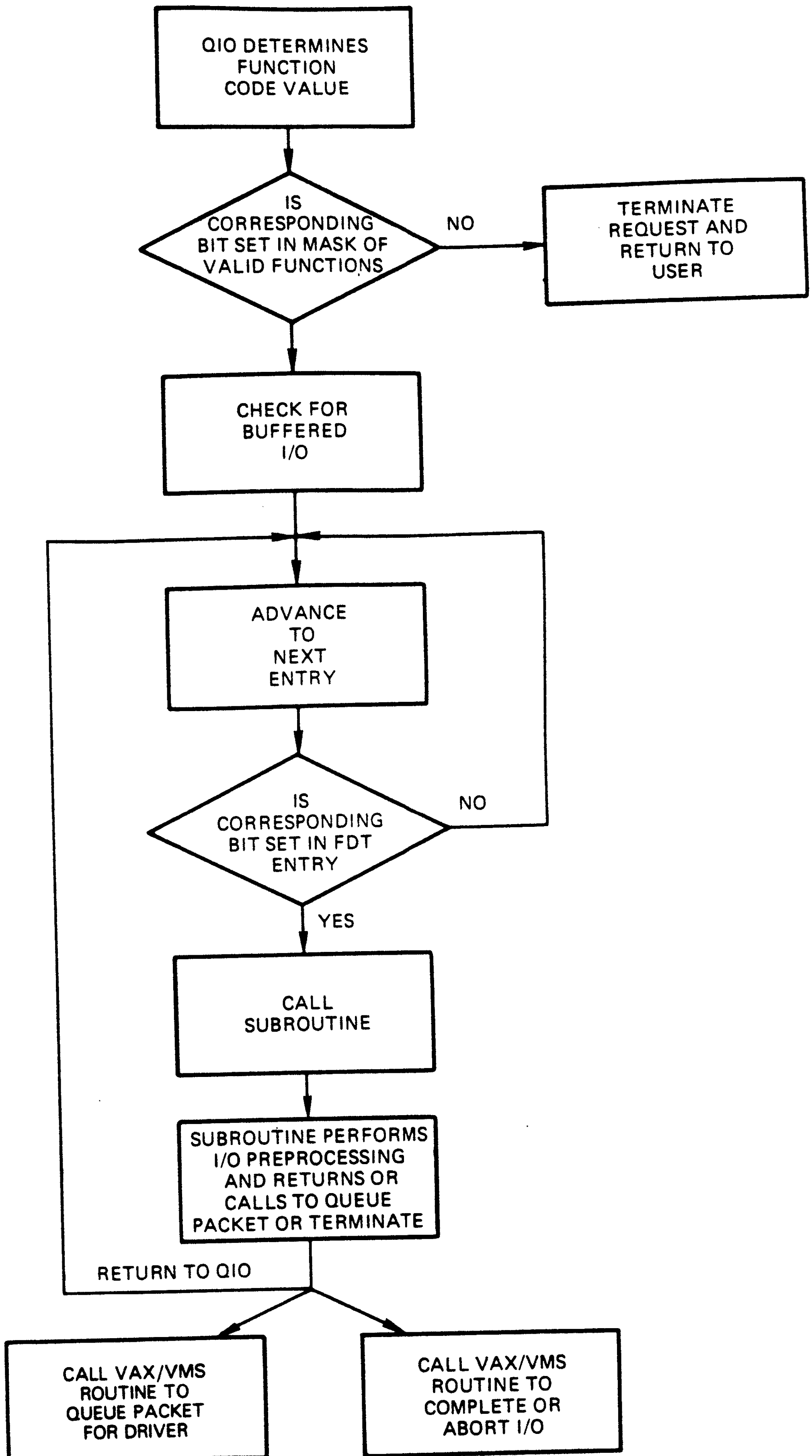


Process I/O Channel Assignment

The first step in preprocessing an I/O request is to verify that the I/O reque: specifies a valid process I/O channel. The process I/O channel is an entry in a system-maintained process table that describes a path of reference from a process to a peripheral device unit. Before a program requests I/O to a device, the program identifies the target device unit by issuing an Assign-I/O-Channel ($ASSIGN) system service call. The $ASSIGN system service performs the following functions:

- Locates an unused entry in the table of process I/O channels
- Creates a pointer to the device unit in the table entry for the channel
- Returns a channel-index number to the program

When the program issues an I/O request, EXE$QIO verifies that the channel number specified is associated with a device and locates the unit-control bloc associated with the specified channel using the field CCB$L_UCB.


QIO DETERMINES FUNCTION CODE VALUE
IS CORRESPONDING BIT SET IN MASK OF VALID FUNCTIONS
NO
TERMINATE REQUEST AND RETURN TO USER
CHECK FOR BUFFERED I/O
ADVANCE TO NEXT ENTRY
IS CORRESPONDING BIT SET IN FDT ENTRY
NO
YES
CALL SUBROUTINE
SUBROUTINE PERFORMS I/O PREPROCESSING AND RETURNS OR CALLS TO QUEUE PACKET OR TERMINATE
RETURN TO QIO
CALL VAX/VMS ROUTINE TO QUEUE PACKET FOR DRIVER
CALL VAX/VMS ROUTINE TO COMPLETE OR ABORT I/O

ZK-922-82

Creating a Driver Fork Process to Start I/O

EXE$INSIOQ creates only one driver fork process at a time for each device unit on the system. As a result, only one IRP per device unit is serviced at one time. EXE$INSIOQ determines whether a driver fork process exists for the target device, as follows:

- If the device is idle, no driver fork process exists for the device; in this case, the EXE$INSIOQ immediately calls IOC$INITIATE to create and transfer control to a driver fork process to execute the driver's start-I/O routine.
- If the device is busy, a driver fork process already exists for the device, servicing some other I/O request. In this case, EXE$INSIOQ calls EXE$INSERTIRP to insert the IRP into a queue of IRPs waiting for the device unit. The routine queues the IRP according to the base priority of the caller. Within each priority, IRPs are in first-in/first-out order. The completion of the current I/O request triggers the servicing of the I/O request that is first in the queue, according to the procedure described in Section 12.1.2.3.

In the latter case, by the time the driver's start-I/O routine gains control to dequeue the IRP, the originating user's process context is no longer available. Because the context of the process initiating the I/O request is not guaranteed to a driver's start-I/O routine, the driver must execute in the reduced context available to a fork process.

IOC$INITIATE always initiates the driver's start-I/O routine with a context that is appropriate for a fork process. VAX/VMS establishes this context by performing the following steps:

1. Raising IPL to driver fork IPL (UCB$B_FIPL)
2. Loading the address of the IRP into R3
3. Loading the address of the device's UCB into R5
4. Transferring control (with a JMP instruction) to the entry point of the device driver's start-I/O routine

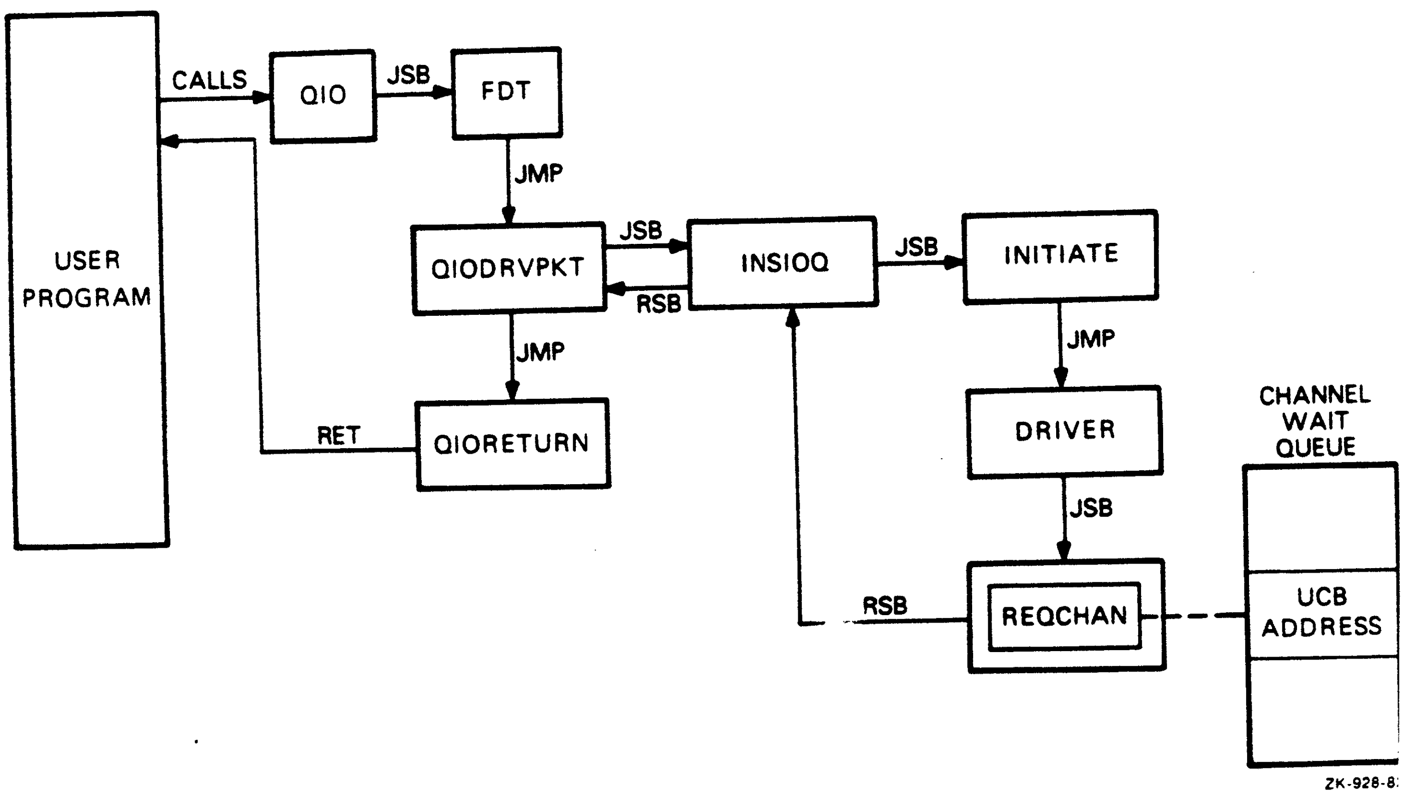
The newly activated driver fork process executes under the following constraints:

- It cannot refer to the address space of the process initiating the I/O request.
- It can use only R0 through R5 freely. It must save other registers before use and restore them after use.
- It must clean up the stack after use. The stack must be in its original state when the fork process relinquishes control to any VAX/VMS routine.
- It must execute at IPLs between driver fork level and IPL$_POWER. It must not lower IPL below fork IPL, except by creating a fork process at a lower IPL.

Each driver fork process executes until one of the following events occurs:

- Device-dependent processing of the I/O request is complete.
- A shared resource needed by the driver is unavailable, as described in Section 3.3.
- Device activity requires the fork process to wait for a device interrupt.

Inserting a UCB into the Channel-Wait Queue



Activating a Device and Waiting for an Interrupt

Depending on the device type supported by the driver, the start-I/O routine performs some or all of the following steps:

1. Analyzes the I/O function and branches to driver code that prepares the UCB and the device for that I/O operation
2. Copies the contents of fields in the IRP into the UCB
3. Tests fields in the UCB to determine whether the device and/or volume mounted on the device are valid
4. If the device is attached to a multiunit controller, obtains the controller data channel
5. If the I/O operation is a DMA transfer, obtains a I/O adapter resources such as mapping registers and a UNIBUS adapter data path
6. Loads all necessary device registers except for the device's control and status register (CSR)
7. Raises IPL to IPL$_POWER (saving the value of fork IPL on the stack) and confirms that a power failure that would invalidate the device operation has not occurred
8. Loads the device's CSR to activate the device
9. Invokes a VAX/VMS routine (using either the WFIKPCH or WFIRLCH macro) to suspend the driver fork process until a device interrupt or timeout occurs

As it suspends the driver, IOC$WFIKPCH or IOC$WFIRLCH saves the driver's context in the UCB. This context consists of the following information

- A description of the I/O request and the state of the device
- The contents of R3 and R4 (UCBL_FR3, UCBL_FR4)
- The implicit contents of R5 as the address of the UCB
- A driver return address (UCB$L_FPC)
- The address of a device timeout handler (at UCB$L_FPC)
- The time at which the device will time out (UCB$L_DUETIM)

Buffered and Direct I/O

Because the buffer specified in the original user I/O request is in process space, it is not automatically accessible to the driver fork process that executes in system context. As a result, for any function that involves data transfer, the driver must select a strategy that supplies a buffer that the fork process can address. The VAX/VMS operating system allows FDT routines a choice between allocating a system buffer (buffered I/O) or locking the process buffer (direct I/O).

A driver employs *buffered I/O* to allocate a buffer from nonpaged pool. It can later refer to the buffer using addresses in system space. For a write request, the driver FDT routine must move data from the user buffer to the allocated system buffer. For a read request, the system ultimately delivers the data from the system buffer to the user buffer by means of a special kernel-mode AST at driver postprocessing. Drivers most often use buffered I/O for PIO devices such as line printers and card readers.

With *direct I/O*, the driver locks the pages of the user buffer in physical memory and refers to them using page-frame numbers (PFNs). Normally, a driver uses direct I/O for DMA transfers.

The trade-off between buffered I/O and direct I/O is the time required to move the data into the user's buffer versus the time required to lock the buffer pages in memory. Sections 7.3.2 and 8.4 provide additional information.

Programmed I/O

Drivers for relatively slow devices, such as printers, card readers, terminals, and some disk and tape drives, must transfer data to a device register a byte or a word at a time. These drivers must themselves keep a record of the location of the data buffer in memory, as well as a running count of the amount of data that has been transferred to or from the device. Thus, these devices perform *programmed I/O* (PIO) in that the transfer is largely conducted by the driver program. This type of transfer is also known as *buffered I/O* because the data registers of certain PIO devices can buffer several bytes or words and transfer those bytes to the device as a group. When this is the case, the driver monitors a device status register to determin when the device buffer is full.

Examples of UNIBUS devices that do PIO transfers are the LP11 and the DZ11. Corresponding Q22 bus devices that perform PIO transfers are the LPV11 and the DZV11.

Section 2 outlines the action of the LP11 driver. The LP11 driver transfers data from a system buffer to the line printer data buffer register a byte at a time, while maintaining a count of the number of bytes left to transfer. When the line printer data buffer is full, the line printer sets a "not ready" bit in its status register. If the driver, while examining this register, sees this bit set, it enables interrupts from the printer, and then suspends itself in the expectation that the printer will post an interrupt to the processor. While the driver remains suspended, the printer prints the data from its buffer and interrupts the processor when it is done. With the interrupt handled by the system interrupt dispatcher and the driver interrupt-servicing routine, driver execution resumes. The driver repeats both its byte-by-byte transfer to the printer data buffer, as well as the entire routine described above, until it determines that all the data has been transferred as requested.

Drivers performing PIO transfers are generally not concerned with the operation of I/O adapters. However, drivers that perform direct-memory-access (DMA) transfers must take into account I/O adapter functions, as discussed below.

Buffered Data Paths

In contrast to the direct data path, the buffered data paths transfer data much more efficiently between the UNIBUS and the backplane interconnect by decoupling the UNIBUS transfer from the backplane interconnect transfer. Buffered data paths read or write multiple words of data in a transfer, and buffer the unrequested portions of the data in UNIBUS adapter buffers. Thus, several UNIBUS read functions can be accommodated with a single backplane interconnect transfer.

A UNIBUS device may choose to use a buffered data path rather than a direct data path to perform the following functions:

- Fast DMA block transfers to or from consecutively increasing addresses
- Word-oriented block transfers that begin and end on an odd-numbered byte of memory; note, however, that these transfers can be quite slow because the UNIBUS adapter might need to perform multiple transfers to complete a one-word transfer
- 32-bit data transfers from random longword-aligned physical addresses

A single buffered data path cannot be assigned to more than one active transfer at a time. When a driver fork process is preparing to transfer data to or from a UNIBUS device on a buffered data path, it performs a sequence of steps similar to those performed by a driver that uses the direct data path, with the exception that it uses a macro that calls a VAX/VMS routine that allocates a free buffered data path. The following are among the actions of the driver fork process:

1. Uses the REQMPR macro to allocate a set of mapping registers.
2. Uses the REQDPR macro to allocate a free buffered data path.
3. Uses the LOADUBA macro to load the mapping registers with physical address mapping data and the number of the allocated buffered data path. The VAX/VMS routine called in the expansion of the LOADUBA macro (IOC$LOADUBAMAP) also sets the valid bit in every mapping register except the last, which remains invalid to prevent a wild transfer.
4. Load the starting address of the transfer in a device register.
5. Load the transfer byte or word count in a device register.
6. Set bits in the device control register to initiate the transfer.

The UNIBUS adapter hardware of certain processors restricts normal buffered data paths to referring only to consecutively increasing addresses. Through a special mode of operation, these UNIBUS adapters can also refer to 32-bit data at randomly-ordered, longword-aligned locations in physical memory. Other processors do not impose this restriction. In order for a device driver to run on both types of processors, it must observe three rules:

- All transfers within a block must be of the same function type (DATI or DATO/DATOB).
- Normal buffered data paths must always transfer data to consecutively increasing addresses.
- To reference 32-bit data at random, longword-aligned locations in physical memory, the longword-access-enable bit (LWAE) must be set.

A buffered data path stores data from the UNIBUS in a buffer until multiple words of data have been transferred (except in longword-aligned, 32-bit, random-access mode as discussed in Section 4.3.5). Then, the UNIBUS adapter transfers the contents of the buffer to the appropriate physical address in a single backplane interconnect operation. The procedure for a UNIBUS write operation that transfers data from a device to memory is broken into individual steps.

1 The UNIBUS device transfers one word of data to the buffered data path.

2 The buffered data path stores the word of data and completes the UNIBUS cycle.

3 The buffered data path sets its buffer-not-empty flag to indicate that the buffer contains valid data.

4 The UNIBUS device repeats the first three steps until the buffer is full.

5 When the UNIBUS device addresses the last byte or word in the buffer, the UNIBUS adapter recognizes a complete data-gathering cycle.

6 The buffered data path requests a backplane-interconnect-write function to write the data from the buffered data path to memory.

7 When the backplane interconnect transfer is complete, the buffered data path clears its flag to indicate that the buffer no longer contains valid data.

The procedure for a UNIBUS read operation that transfers data from main memory to a device varies according to the type of UNIBUS adapter. Those adapters that can perform a prefetch function complete UNIBUS reads from memory more quickly than those that cannot. The prefetch feature accomplishes this improved performance by automatically filling the data path buffer after the buffer's contents are transferred to the UNIBUS.

The following paragraphs discuss the UNIBUS read operation with and without the prefetch function. Device drivers that adhere to the conventions outlined in this manual will execute properly whether or not the device is associated with a UNIBUS adapter that provides prefetch functionality.

1 The UNIBUS device initiates a read operation from a buffered data path.

2 The buffered data path checks to see if its buffers contain valid data.

3 If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data from main memory. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.

4 The buffered data path transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.

5 The buffered data path sets its buffer-not-empty flag to indicate that the buffers contain valid data.

6 When the UNIBUS device empties the buffers of the buffered data path with a UNIBUS read function that accesses the last word of data, the buffered data path clears the buffer-not-empty flag to indicate that the buffer no longer contains valid data.

7 The buffered data path then initiates a read function to prefetch data from memory.

8 When the prefetch is complete, the buffered data path sets the buffer-not-empty flag to indicate that the buffers now contain valid data.

The prefetch might attempt to read data beyond the address mapped by the final mapping register. To avoid referring to memory that does not exist, the VAX/VMS routines that allocate and load mapping registers always allocate one extra mapping register and clear the mapping-register-valid bit before initiating the transfer. When the UNIBUS adapter notices that the mapping register for the prefetch is invalid, the UNIBUS adapter aborts the prefetch without reporting an error.

The steps of a UNIBUS read function without prefetch are listed below.

1. The UNIBUS device initiates a read operation from a buffered data path.
2. The buffered data path checks to see if its buffers contain valid data.
3. If the buffers do not contain valid data, the buffered data path initiates a read function to fill the buffers with data. The transfer completes before the UNIBUS adapter begins a UNIBUS transfer.
4. The buffered data path transfers the requested bytes to the UNIBUS. Bytes of data that were not transferred to the UNIBUS remain in the buffer.

Direct Data Path

Since the direct data path performs a backplane interconnect transfer for every UNIBUS transfer, it can be used by more than one UNIBUS device at a time. The UNIBUS adapter arbitrates among devices that wish to use the direct data path simultaneously. The device driver is unaffected by this UNIBUS adapter arbitration.

The direct data path is slower than buffered data paths because each UNIBUS transfer cycle corresponds to a backplane interconnect cycle. One word or byte is transferred for each backplane interconnect cycle. On some hardware configurations, the direct data path is unable to transfer a word of data to an odd-numbered physical address. Therefore, an FDT routine for a DMA device that uses the direct data path should check that the specified buffer is on a word boundary.[5]

A UNIBUS device may choose to use a direct data path rather than a buffered data path to perform the following functions:

- Execute an interlock sequence to the backplane interconnect (DATIP-DATO/DATOB)
- Transfer to randomly ordered addresses instead of consecutively increasing addresses
- Mix read and write functions

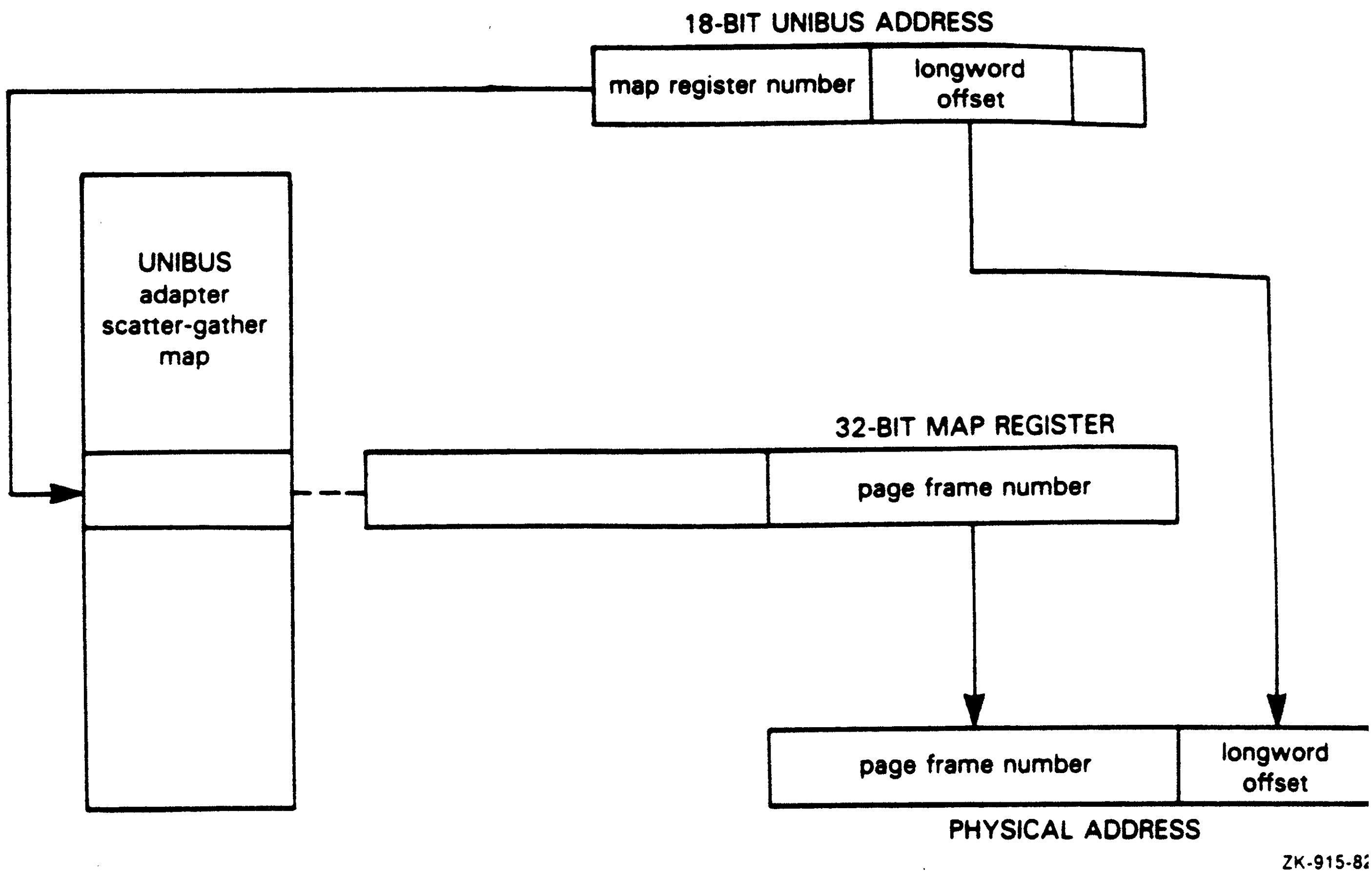
The direct data path is the simplest data path to program. Since the direct data path can be shared simultaneously by any number of I/O transfers, the device driver does not need to call the VAX/VMS routine that allocates the data path. It performs the following actions:

1. Uses the REQMPR macro to allocate a set of mapping registers
2. Uses the LOADUBA macro to load the mapping registers with physical address mapping data and the number of the direct data path (0). The VAX/VMS routine called in the expansion of the LOADUBA macro (IOC$LOADUBAMAP) also sets the valid bit in every mapping register except the last, which remains invalid to prevent a wild transfer.
3. Loads the starting address of the transfer in a device register.
4. Loads the transfer byte or word count in a device register.
5. Sets bits in the device control register to initiate the transfer.

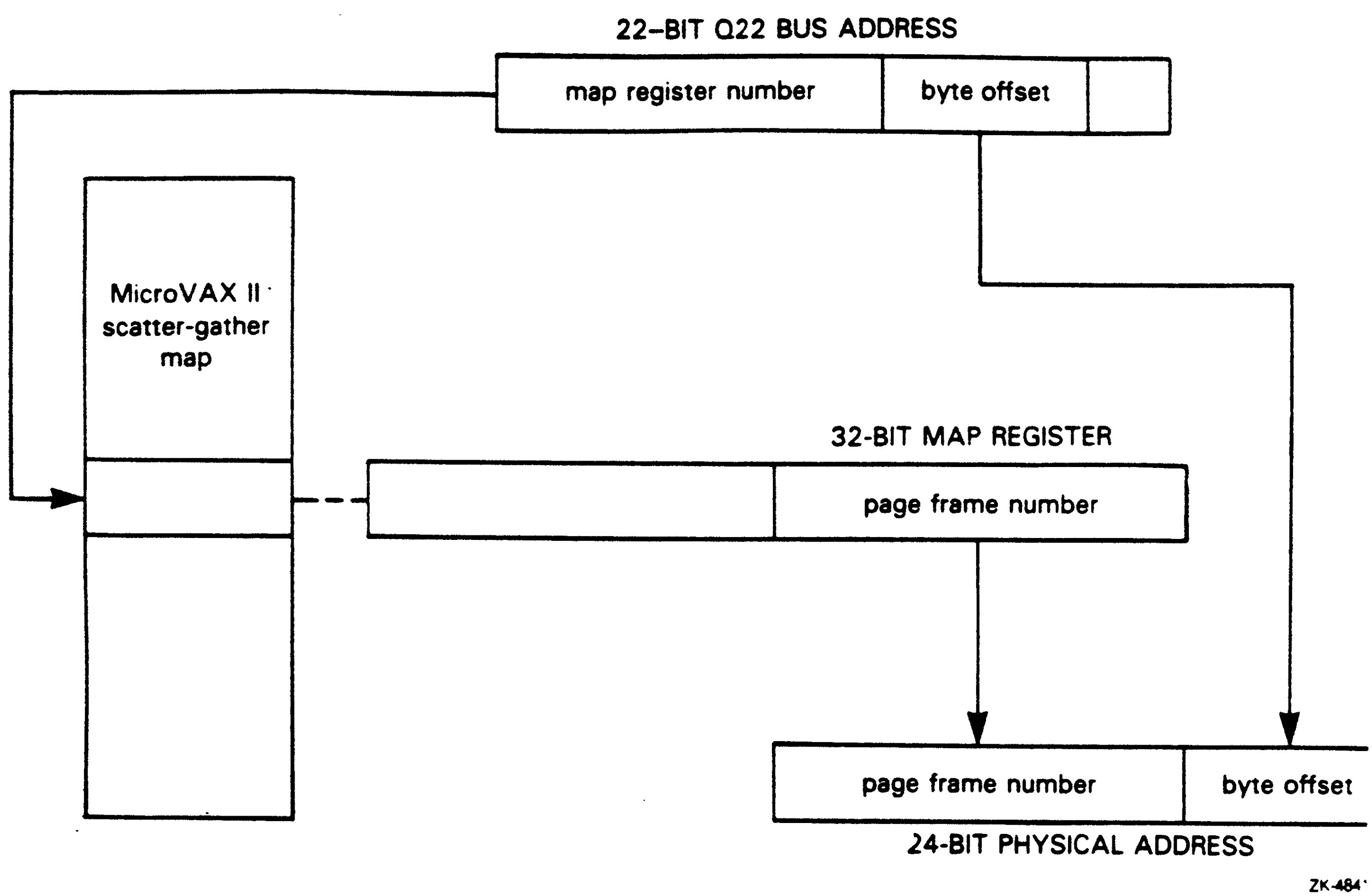
---

5 The MicroVAX II and MicroVAX I implementations of the Q22 bus provide no byte-offset register. As a result, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

Mapping a UNIBUS Address to a Physical Address



Mapping a Q22 Bus Address to a Physical Address



Features of the I/O Bus Adapters of the VAX Processors

Processor	Adapter	Memory References (Physical Address)	Direct Data Path	Buffered Data Paths	Mapping Registers	Interrupt Dispatcher
VAX–11/780 **VAX–11/782** **VAX–11/785** **VAX 8600** **VAX 8650**	UBA	30-bit (via SBI)	1, no byte-aligned transfers	15, 8-byte buffer, byte-aligned transfers, LWAE,[3] prefetch	496	Nondirect vecto
VAX–11/750	UBI	24-bit (via CMI)	1, byte-aligned transfers	3, 4-byte buffer,[2] byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct vector

[2]Buffered data paths on the VAX–11/750 only buffer four bytes of data. Because the data paths do not perform a prefetch, they can always reference longwords at random.

[3]LWAE (longword access enable) refers to the capability to reference random longword aligned data in a bus transfer.

[4]The VAX/VMS operating system makes only 496 of these mapping registers available.

Features of the I/O Bus Adapters of the VAX Processors

Processor	Adapter	Memory References (Physical Address)	Direct Data Path	Buffered Data Paths	Mapping Registers	Interrupt Dispatcher
VAX–11/730 **VAX–11/725**	UBA	24-bit	1, byte-aligned transfers	None	512[4]	Direct vector
VAX 8200 **VAX 8800**	BUA	30-bit (via VAXBI)	1, byte-aligned transfers	5, 8-byte buffer, byte-aligned transfers, LWAE,[3] no prefetch	512[4]	Direct vector
MicroVAX I	—	22-bit	1, no restrictions on data alignment[1]	None	None	Direct vector
MicroVAX II	—	24-bit	1, no restrictions on data alignment[1]	None	8192[4]	Direct vector

[1]The MicroVAX II and MicroVAX I implementations of the Q22 bus provide no byte-offset register, so, on Q22 bus devices that are only capable of word-aligned transfers, only word-aligned transfers are possible.

[3]LWAE (longword access enable) refers to the capability to reference random longword aligned data in a bus transfer.

[4]The VAX/VMS operating system makes only 496 of these mapping registers available.

Competing for a Controller's Data Channel

A controller's data channel is a VAX/VMS synchronization mechanism that guarantees for multiunit controllers that one unit uses the controller at a time. A device's fork process can read and write a device's registers whenever the device unit owns the controller's data channel.

Devices that share a controller, such as disk units, own the controller's data channel only when a VAX/VMS routine assigns the channel to the unit's fork process. In contrast, a single device unit on a controller always owns the controller's data channel. Therefore, if VAX/VMS transfers control to such a driver's start-I/O routine, the driver can immediately address the device's registers without first obtaining the controller's data channel.

An LP11 printer, such as the one discussed in Section 2, has a dedicated (single-unit) controller attached to the UNIBUS. When VAX/VMS finds the device idle and creates a printer driver's fork process to write data to the printer's data buffer, the controller's data channel is guaranteed not to be busy. Because the data channel is not busy, the driver's start-I/O routine can perform the following:

1. Retrieve the virtual address of the data to be written and the number of bytes to transfer from the device's UCB
2. Retrieve the virtual address of the device's CSR from the IDB
3. Calculate the address of the line printer's data buffer register by adding a constant offset to the CSR address
4. Write data, one byte at a time, to the line printer's data buffer until all bytes of data have been written

In contrast, a device unit on a multiunit controller must compete for the controller's data channel with other devices attached to that controller.

Synchronization of I/O-Request Processing

An RK611 controller, for example, controls as many as eight RK06/RK07 devices. The disk driver's fork process must gain control of the controller's data channel before starting an I/O operation on the unit associated with the fork process. The disk driver's start-I/O routine uses the following sequence to start a seek operation on an RK07 device:

1. The start-I/O routine requests the controller's data channel by invoking a VAX/VMS channel arbitration routine.
2. The VAX/VMS routine tests the CRB mask field to determine whether the controller's data channel is available.
3. If the channel is available, the VAX/VMS routine allocates the channel to the fork process and returns the address of the device's CSR to the fork process.

   If the channel is busy, the VAX/VMS routine saves the driver fork context in the UCB fork block and inserts the fork block address in the controller's channel-wait queue.
4. When the fork process resumes execution, the process owns the controller channel. The fork process can then modify the device's registers to activate the device.
5. The driver's start-I/O routine then requests the VAX/VMS operating system to suspend driver processing in anticipation of an interrupt or timeout and to release the channel.
6. The VAX/VMS channel-releasing routine assigns channel ownership to the next fork process in the channel-wait queue, loads the CSR address into a general register, and reactivates the suspended fork process.
7. The reactivated fork process continues execution as though the channel had been available in the first place.

The VAX/VMS channel-arbitration routines keep track of controller availability using a flag field in the CRB. The fork process must always request and release the controller's data channel by invoking these routines. Once the driver owns a controller's data channel, the driver is free to read and modify the device's registers.

The CRB's interrupt-dispatching field (CRB$L_INTD+2) contains executable code that the driver-loading procedure has associated with the interrupting vector. Interrupt-dispatching fields for nondirect vectors contain the following executable instruction:

```
JSB @#address-of-driver-isr
```

On a configuration that uses direct vector interrupts—such as the MicroVAX I, MicroVAX II, VAX 8200, VAX 8800, VAX-11/750, and VAX-11/730—the following sequence occurs:

1 The processor saves, on the interrupt stack, the PC and PSL of the currently executing code and acknowledges the device's interrupt.

2 The device supplies its vector address, which the processor uses as an index into a table in the second (or third) page of the SCB This table contains a list of addresses in the CRB that point to the interrupt-servicing routines for devices attached to the first UNIBUS or an optional second UNIBUS (for the VAX-11/750).

3 When the processor locates the address in the SCB that corresponds to the vector address, it transfers control to an interrupt-dispatching field in the CRB.

4 The CRB's interrupt-dispatching field (CRB$L_INTD) contains executable code that the driver-loading procedure has associated with the interrupt vector. Interrupt-dispatching fields of direct vectors contain the following executable instructions:

```
PUSHR <R0,R1,R2,R3,R4,R5>
JSB   @#address-of-driver-isr
```

The driver-loading procedure determines how many interrupt-dispatching fields to build within the CRB from the number of vectors specified in the /NUMVEC qualifier to the SYSGEN command CONNECT The driver-loading procedure obtains the address of the interrupt-servicing routine for each interrupt-dispatching field from the reinitialization portion of the driver-prologue table This section of the DPT contains one or more DPT_STORE macros that identify the addresses of the interrupt-servicing routines. The number of DPT_STORE macros that identify interrupt-servicing routines must equal the number of vectors given in the /NUMVEC qualifier to avoid errors in device initialization or interrupt handling.

Immediately following the JSB instruction in the CRB is the address of the interrupt-dispatch block (IDB) associated with the CRB. When the JSB instruction executes, a pointer to the address of the IDB is pushed onto the top of the stack as though it were a return address. The driver interrupt-servicing routine can use this IDB address as a pointer into the I/O database. Figure 11-2 illustrates the portion of a CRB that contains the address of the interrupt-servicing routine.

Channel-Request Block

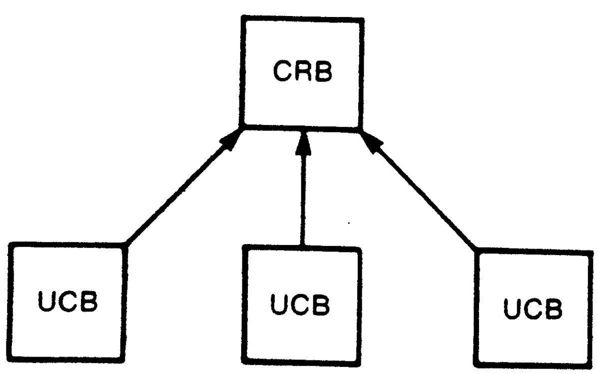
The channel-request block (CRB) allows the operating system to manage the controller data channel. Among its contents are:

- Code that transfers control to a driver's interrupt-servicing routine (CRB$L_INTD)
- Addresses of a driver's unit and controller initialization routines (CRB$L_INTD+VEC$L_UNITINIT, CRB$L_INTD+VEC$L_INITIAL)
- A pointer to the interrupt-dispatch block (IDB), which further describes the controller (CRB$L_INTD+VEC$L_IDB)

Controllers can be either multiunit or dedicated.

All UCBs describing device units attached to a single *multiunit controller* contain a pointer to a single CRB (UCB$L_CRB). For these controllers, a VAX/VMS routine uses fields in the CRB (CRBL_WQFL, CRBB_MASK) and IDB (IDB$L_OWNER) to arbitrate pending driver requests for the controller. When the system grants ownership of a multiunit controller data channel to a driver fork process, the fork process can initiate an I/O operation on a device attached to that controller. Figure 5–3 illustrates the data structures required to describe three devices on a multiunit controller.

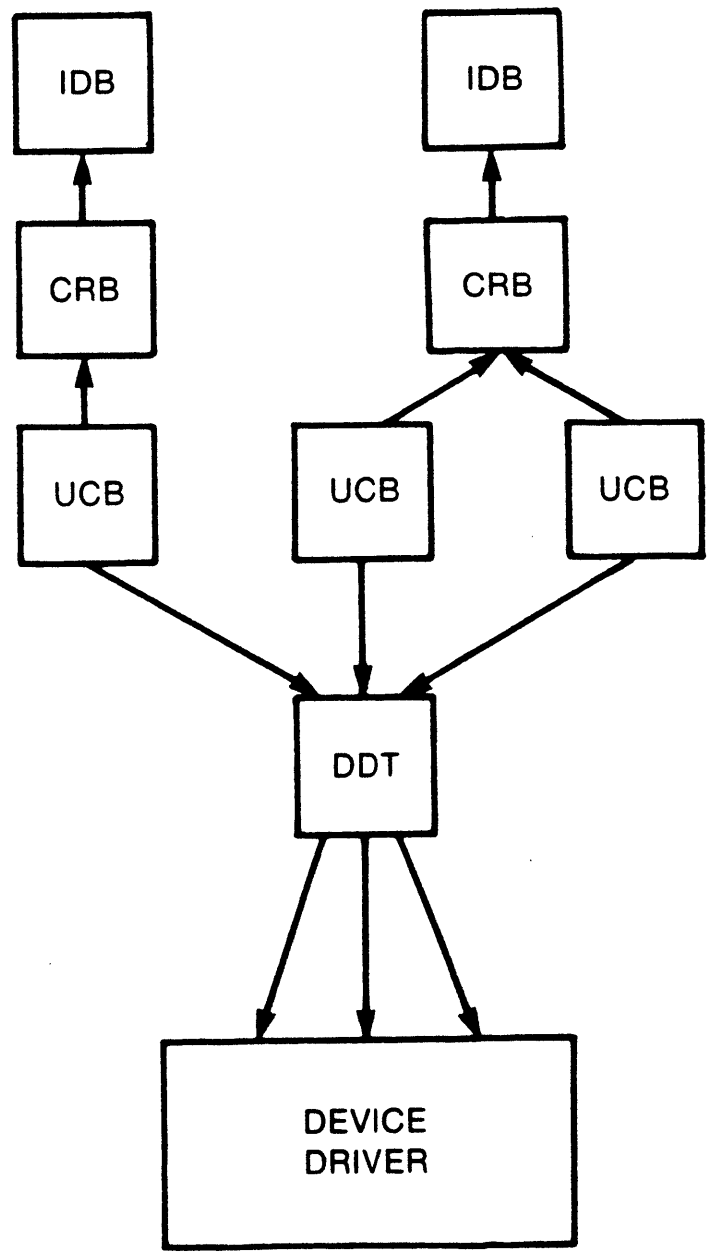
Data Structures for Three Devices on One Controller



ZK-920-82

The VAX/VMS operating system does not use the CRB to synchronize I/O operations for a *dedicated controller*, as the controller manages but a single device. Nevertheless, the CRB still is present and used by drivers and operating system routines.

I/O Database for Two Controllers



ZK-1765-84

Interrupt-Dispatch Block

The CRB contains a pointer to an interrupt-dispatch block (IDB) (CRB$L_INTD+VEC$L_IDB). The IDB contains the addresses of these three critical data structures:

- The UCB of the device unit, if any, that currently owns the controller data channel (IDB$L_OWNER)
- The control and status register (IDB$L_CSR); it is the key to access to device registers
- The adapter-control block (IDB$L_ADP) that describes the adapter of the I/O bus to which the controller is attached

A detailed description of the fields in the IDB appears in Table A-9; Figure A-9 shows its structure.

Figure 5–4 illustrates the relationship between the data structures that describe a group of equivalent devices on two separate controllers. In this figure, one controller has a single device unit, and the other controller has two device units. Devices on both controllers share the same driver code.

Completing an I/O Request

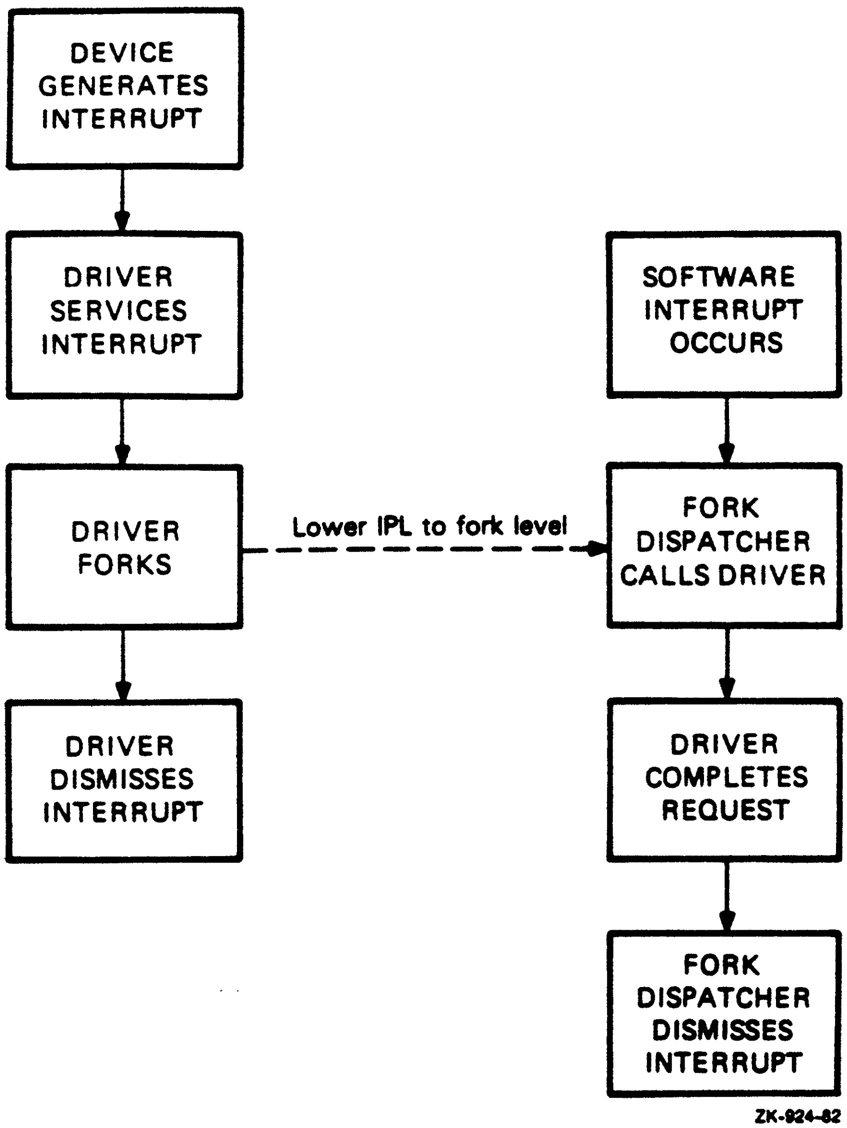
Once reactivated, a driver fork process completes the I/O request as follows

1. Releases shared driver resources, such as I/O adapter mapping registers, UNIBUS adapter data path, and controller ownership
2. Returns status to the VAX/VMS I/O completion routine

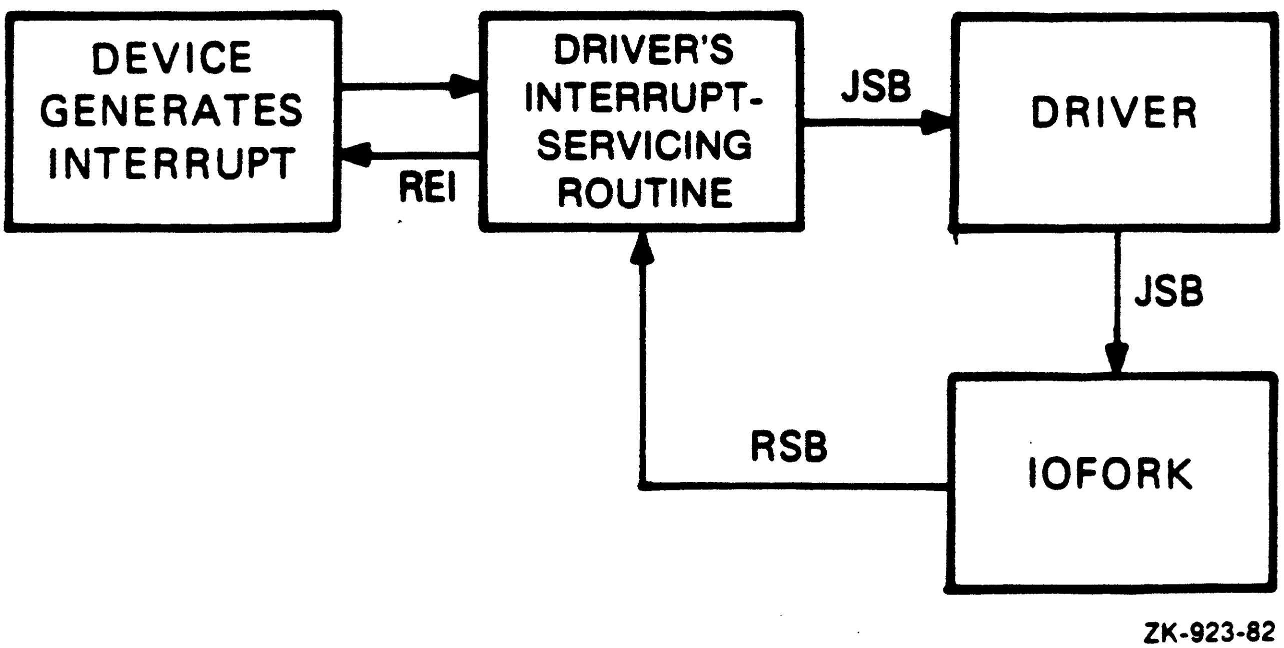
The I/O-completion routine performs the following steps to start postprocessing of the I/O request and to start processing the next I/O reque in the device's queue:

1. Writes return status from the driver into the IRP
2. Inserts the finished IRP in the I/O-postprocessing fork-queue and reques an interrupt at IPL$_IOPOST
3. Creates a new fork process for the next IRP in the device's pending I/O queue
4. Activates the new driver fork process

Reactivation of a Driver Fork Process



Creating a Fork Process After



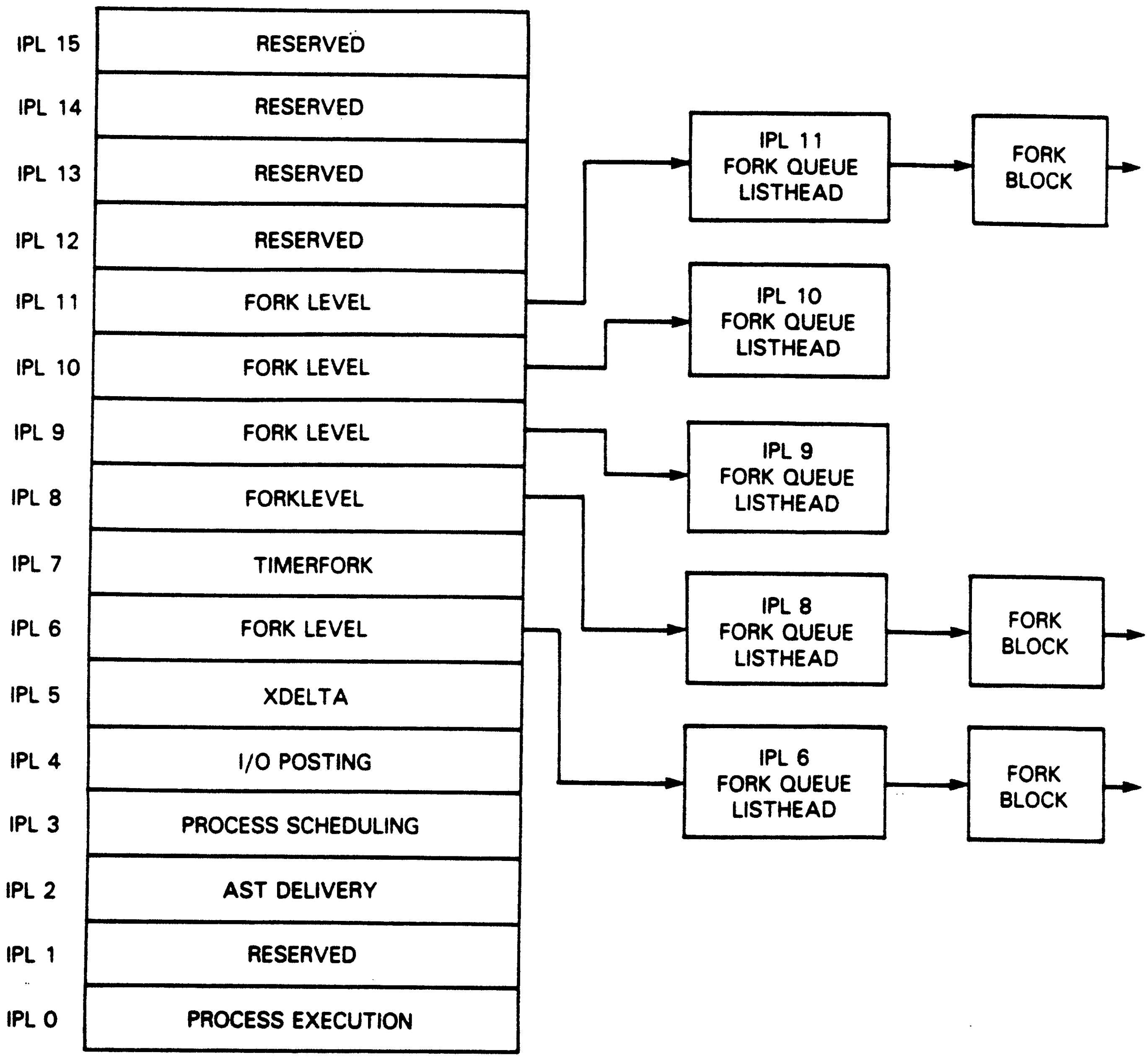
from Interrupt to Fork Process Context

To lower its priority, the driver calls a VAX/VMS fork process queuing routi (by means of the IOFORK macro) that performs the following steps:

1. Disables the timeout that was specified in the wait-for-interrupt routine
2. Saves R3 and R4 (these are the registers needed to execute as a fork process) (UCBL_FR3, UCBL_FR4)
3. Saves the address of the instruction following the IOFORK request in the UCB fork block (UCB$L_FPC)
4. Places the address of the UCB fork block from R5 in a fork queue for the driver's fork level
5. Returns to the driver's interrupt-servicing routine

The interrupt-servicing routine then cleans up the stack, restores registers, and dismisses the interrupt. Figure 5–7 illustrates the flow of control in a driver that creates a fork process after a device interrupt.

Fork Dispatching Queue Structure



ZK-584-81

Activating a Fork Process from a Fork Queue

When no hardware interrupts are pending, the software interrupt priority arbitration logic of the processor transfers control to the software interrupt fork dispatcher. When the processor grants an interrupt at a fork IPL, the fork dispatcher processes the fork queue that corresponds to the IPL of the interrupt. To do so, the dispatcher performs these actions:

1. Removes a driver fork block from the fork queue
2. Restores fork context
3. Transfers control back to the fork process

Thus, the driver code calls VAX/VMS code that coordinates suspension and restoration of a driver fork process. This convention allows VAX/VMS to service hardware device interrupts in a timely manner and reactivate driver fork processes as soon as no device requires attention.

When a given fork process completes execution, the fork dispatcher removes the next entry, if any, from the fork queue, restores its fork process context, and reactivates it. This sequence is repeated until the fork queue is empty. When the queue is empty, the fork dispatcher restores R0 through R5 from the stack and dismisses the interrupt with an REI instruction.

Postprocessing

When processor priority drops below the I/O postprocessing IPL, the processor dispatches to the I/O postprocessing interrupt-servicing routine. This VAX/VMS routine completes device-independent processing of the I/O request.

Using the IRP as a source of information, the IPL$_IOPOST dispatcher executes the sequence below for each IRP in the postprocessing queue:

1. Removes the IRP from the queue
2. If the I/O function was a direct I/O function, adjusts the recorded use of the issuing process' direct I/O quota and unlocks the pages involved in the I/O transfer
3. If the I/O function was a buffered I/O function, adjusts the recorded use of the issuing process' buffered I/O quota and, if the I/O was a write function, deallocates the system buffers used in the transfer
4. Posts the event flag associated with the I/O request
5. Queues a special kernel-mode-AST routine to the process that issued the $QIO system service call

The queuing of a special kernel-mode-AST routine allows I/O postprocessing to execute in the context of the user process but in a privileged access mode. Process context is needed to return the results of the I/O operation to the process' address space. The special kernel-mode-AST routine writes the following data into the process' address space:

- Data read in a buffered I/O operation
- If specified in the I/O request, the contents of the diagnostic buffer
- If specified in the I/O request, the two longwords of I/O status

If the I/O request specifies a user-mode-AST routine, the special kernel-mode-AST routine queues the user-mode AST for the process. When VAX/VMS delivers the user-mode AST, the system AST delivery routine deallocates the IRP. The first part of an IRP is the AST-control block for user requested ASTs.

RMS Implementation and Structure

INTRODUCTION

Programmers on VAX/VMS can access the I/O system on a variety of levels. One method of performing I/O is through the Record Management Services (RMS). RMS provides greater flexibility than most high-level language I/O statements, and can be easier to use than the I/O system services.

RMS may be invoked directly by a programmer, or indirectly through high-level language statements. It is important for the Internals student to understand the module structure and flow of the RMS routines.

Some RMS data structures, such as the record access block (RAB), can be specified by the user, and RMS uses additional internal data structures. This invisible part of RMS affects both the process and the system. Understanding some of the details of RMS implementation enables a better understanding of your process and the system as a whole.

OBJECTIVES

1. To trace a standard RMS read or write through the proper code modules.

2. To describe RMS's entry and exit points as seen by other VMS facilities.

RESOURCES

Reading

- VAX Record Management Services Reference Manual

Source Modules

Facility Name	Module Name
SYS	SHELL
RMS	all RMS0xxx modules

TOPICS

I. User-Specified Data Structures (FAB, RAB, etc.)

II. RMS Internal Data Structures

   A. Process I/O control page (for example, default values, I/O segment area)

   B. File-oriented and Record-oriented data structures (IFAB, IRAB, BufDescBlk, I/O Buffer)

III. RMS Processing

   A. RMS dispatching

   B. RMS routines and data structures

   C. Example - flow of a GET operation

USER-SPECIFIED DATA STRUCTURES

- File Access Block (FAB) 1 per file
  - File organization
  - File access
  - Pointers or indices to other data blocks
  - Space allocation

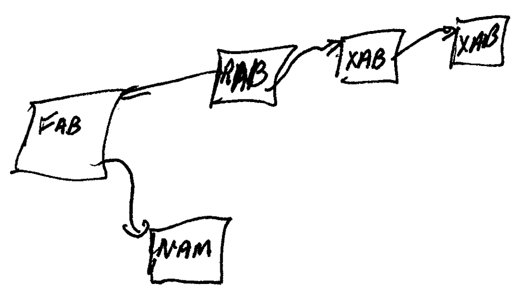
- Record Access Block (RAB) 1 per access mode
  - Record size
  - Block length
  - Pointers or indices to other data blocks → connect to FAB "internal stream ID"
  - Buffer address

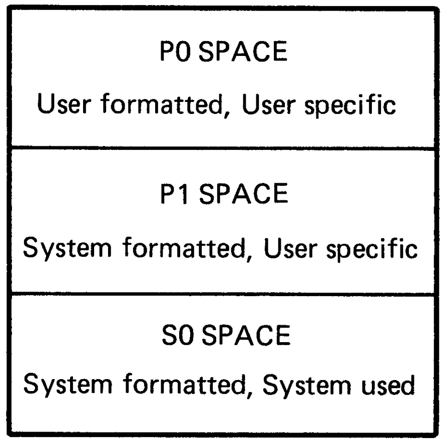
- Name Block (NAM)
  - File name information
  - Directory ID
  - File ID and sequence number
  - Additional file information

- Extended Attribute Blocks (XAB) 1 per key

  They carry additional information on:

  - Header characteristics
  - Allocation
  - Date/time
  - Protection
  - Terminal control





MKV84-2776

Figure 1 Virtual Address Space

- P0 Space
  - User program
  - User data structures
  - User buffers, FABs, RABs, NAMs, XABs
- P1 Space
  - Code (DCL)
  - Data

    DCL symbols
    Process logical name tables
    RMS data structures
- S0 Space
  - SYS.EXE
  - RMS.EXE
  - SYSMSG.EXE
  - Paged pool
  - Nonpaged pool
  - RMS shared file data structures

RMS INTERNAL DATA STRUCTURES

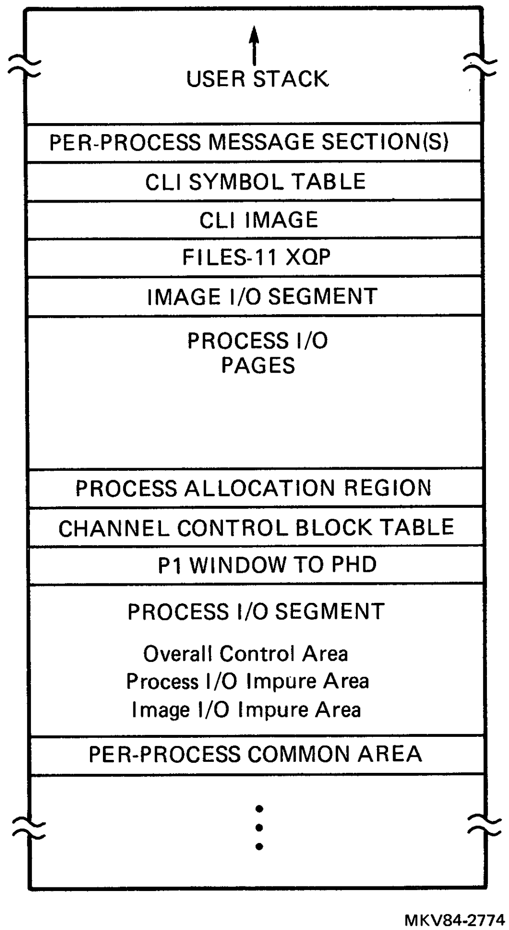


Figure 2 Process I/O Segment in P1 Space

RMS stores some of its information in the Process I/O Segment. The area consists of:

- Overall Control Area
- Process I/O Impure Area
- Image I/O Impure Area

PIO pages pageable

Process I/O Segment

- Overall Control Area
  - Listheads for free space
  - Default values
  - RMS status flags

- Process I/O Impure Area
  - Pointers to process I/O structures
  - Buffer page protection information
  - Status Flags

- Image I/O Impure Area (for image activator)
  - Pointers to image I/O structures
  - Buffer page protection information
  - Status flags

The Overall Control Information Area

- Free memory listhead
- Free list header for image I/O segment (2 longwords)
- RMS overall status
- End of data string
- Default Information
  - File protection
  - Multiblock count
  - Multibuffer counts for

      Sequential disk files
      Magtape files
      Unit record devices
      Relative files
      Indexed files
- Network block count transfer size
- Structure level for RMS files
- Extend quantity for RMS files
- Directory cache list head
- Free list for directory cache nodes (Singly linked list)
- List of locks held (Singly linked list)
- Next sequence number for IRB$L_IDENT

Impure Data Areas

Process Impure Data Area (PIO$GW_PIOIMPA)

- I/O buffer protection (PRT$C_UREW)
- Process I/O segment
  - Set up by PROCSTRT
- Free page listhead
- Free list header
- SP saved longword
- IFAB table address
- IRAB table address
- Number of slots per table (IMP$C_NPIOFILES)

Image Impure Data Area (PIO$GW_IIOIMPA)

- Protection set on pages
- Length of image I/O segment in bytes
- IFAB table address
- IRAB table address
- Number of slots per table (IMP$C_ENTPERSEG)
- IFAB table slots
  - Length is IMP$C_ENTPERSEG longwords
- IRAB table slots
  - Length is IMP$C_ENTPERSEG longwords

File-Oriented and Record-Oriented Data Structures

- IFAB
  - Contains pointers to all data structures associated with a file
  - Many user FAB fields duplicated here but protected user read, executive write

- IRAB
  - Performs similar functions to the IFAB
  - Record pointer (RP) and next record pointer (NRP) stored here

- Buffer Descriptor Block
  - One required for each buffer
  - Contains:

    Status information
    Address fields
    Pointer to associated IRAB
    Pointer to queue of other BDBs

- I/O Buffer
  - Used as the actual source/destination of memory-device transfers
  - The storage is used directly with $GET and LOCATE mode

- Asynchronous Context Block (not ASICB)

  - One is associated temporarily with each IFAB and permanently with each IRAB

  - Contains fields corresponding to the caller's argument list (if the caller is also asynchronous) and register and stack contents

- Record Lock Block

  - One RLB for each record locked at any one time

  - Stores the owner process of a record and the record address (RFA)

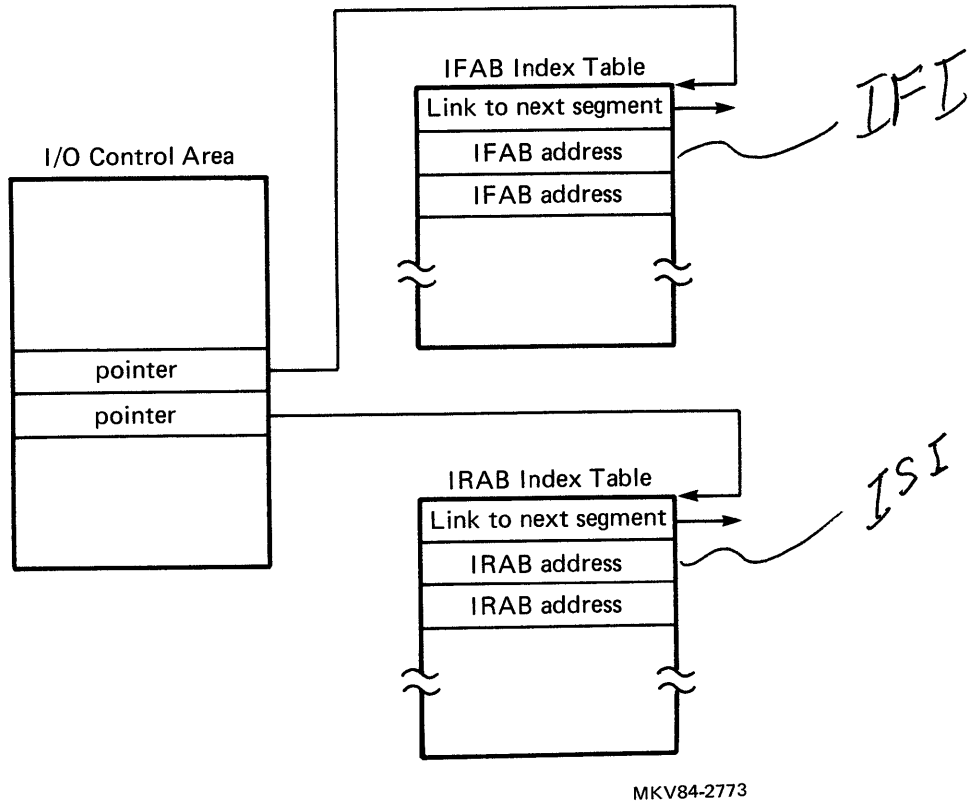


Figure 3 IFAB and IRAB Tables

- IFAB is found by indexing into the IFAB table with the IFI (IFI = Internal File Identifier stored in the FAB)

- IRAB is found by indexing into the IRAB table with the ISI (ISI = Internal Stream Identifier stored in the RAB)

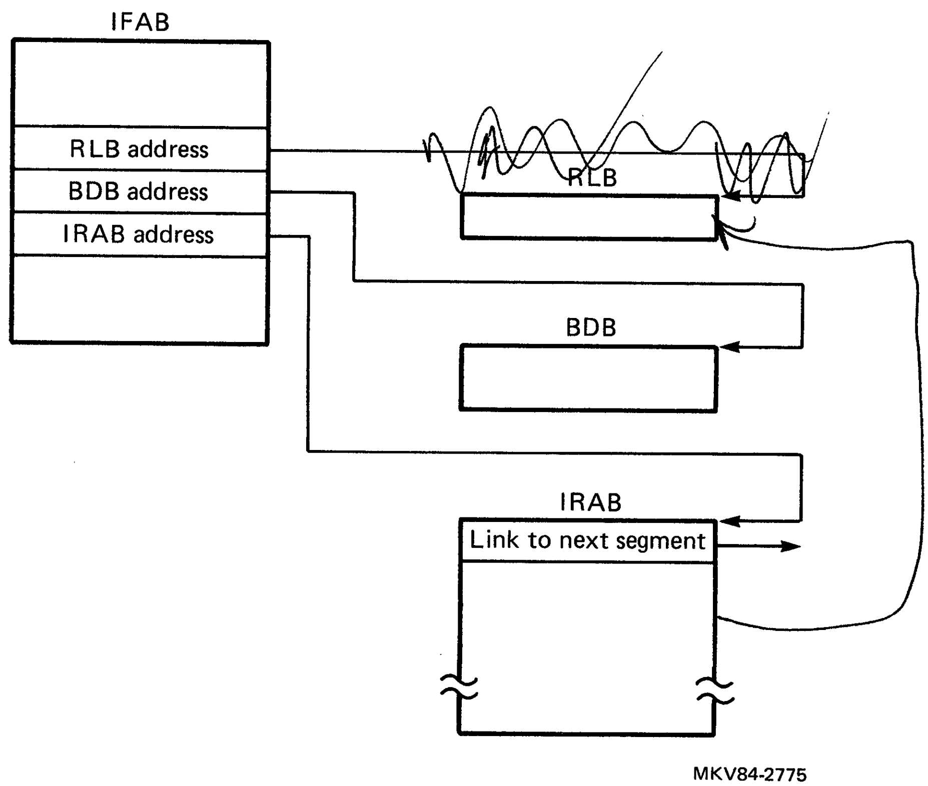


Figure 4 IFAB and Associated Blocks

The IFAB contains pointers to:

- Record Lock Block (RLB)
- Buffer Descriptor Block (BDB)
- Internal Record Access Block (IRAB)

RMS PROCESSING

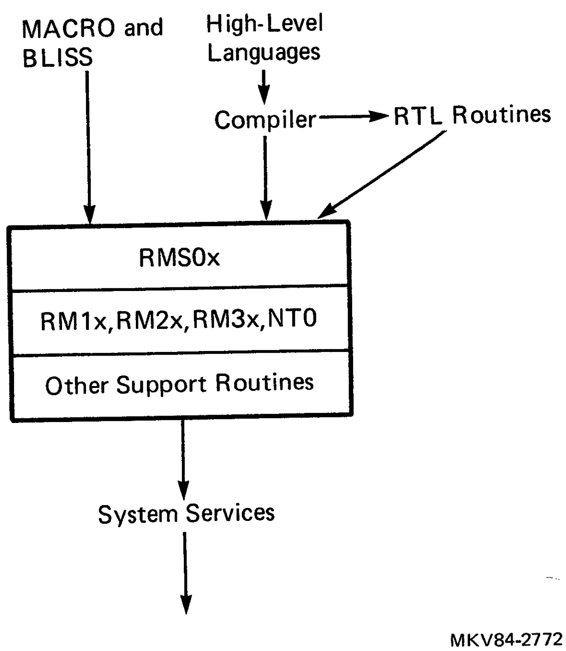


Figure 5 RMS Interfaces

- MACRO and BLISS programs can:
  - Call RMS directly
  - Access RMS data structures directly

- High-level language programs:
  - Use language I/O statements, which the compiler translates to RMS calls
  - Call Run-Time Library routines (which translate to RMS calls)
  - Most cannot access RMS data structures directly
  - VAX-11 PASCAL can access RMS data structures directly

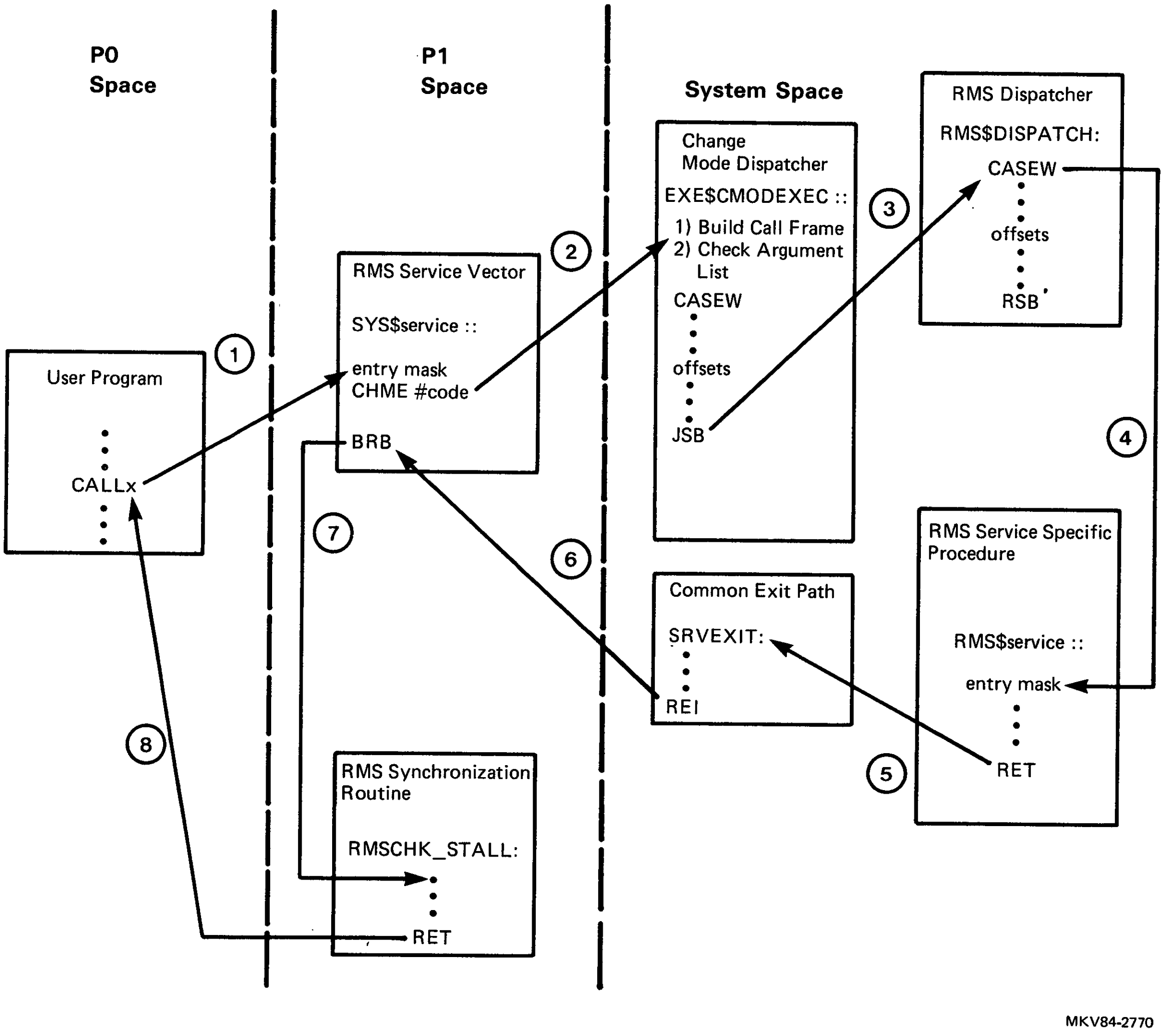


Figure 6 RMS Dispatching

- RMS dispatches through executive mode
- System service-like vectors
- A common exit path
- Synchronization routines

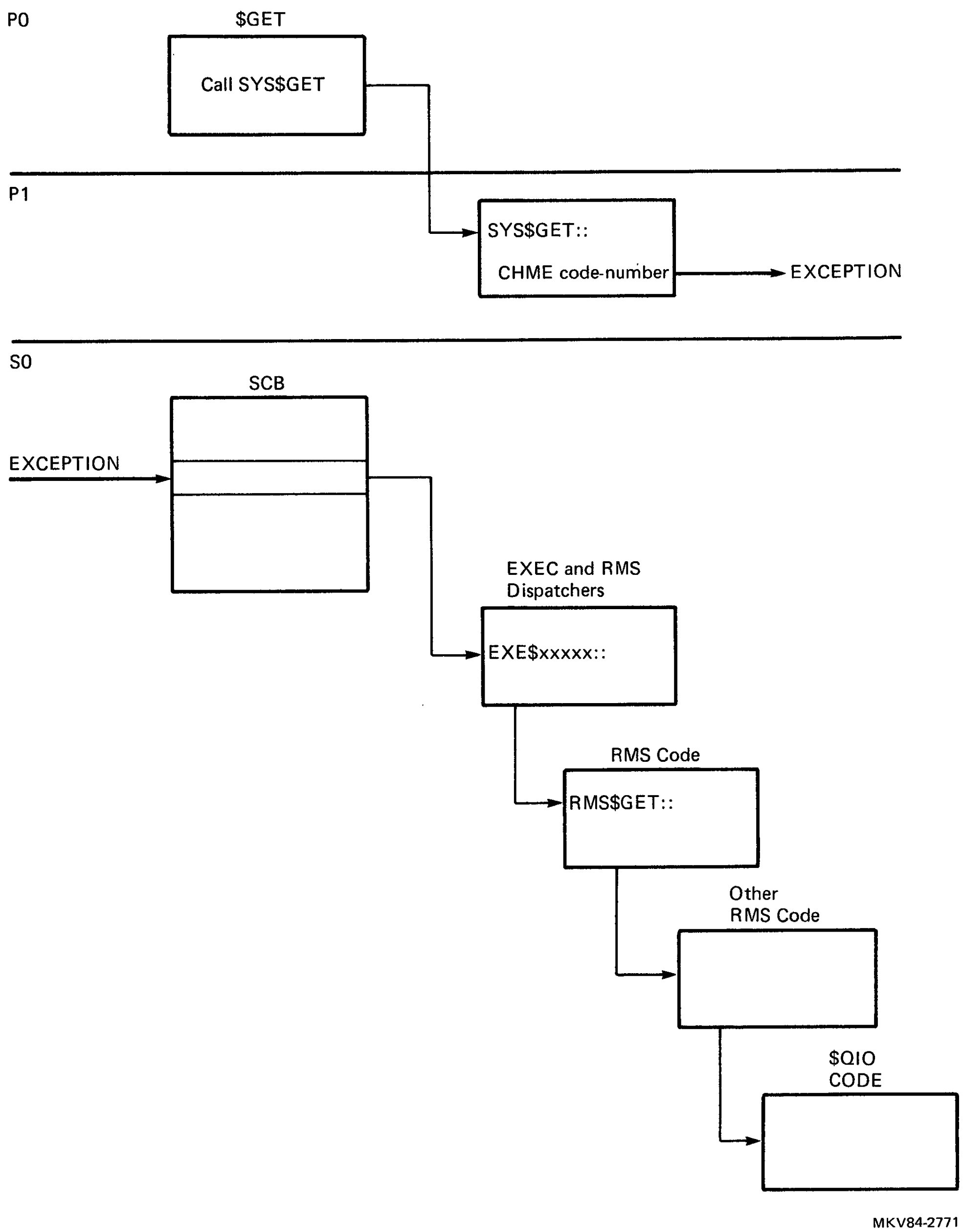


Figure 7 RMS Components in a GET Operation

- Dispatcher (CMODSSDSP.MAR)
- RMS$GET (RMS0BLKIO.MAR)
- EXE$QIO (SYSQIOREQ.MAR)

Table 1 RMS Calling MACROS and the Resulting Code

RMS Macro	P1 Entry Vector	RMS Module
$CLOSE	SYS$CLOSE	RMS0CLOSE
$CONNECT	SYS$CONNECT	RMS0CONN
$CREATE	SYS$CREATE	RMS0CREAT
$DELETE	SYS$DELETE	RMS0DELET
$DISCONNECT	SYS$DISCONNECT	RMS0DISC
$DISPLAY	SYS$DISPLAY	RMS0DISPL
$ERASE	SYS$ERASE	RMS0ERASE
$EXTEND	SYS$EXTEND	RMS0EXTEN
$FIND	SYS$FIND	RMS0FIND
$FILESCAN	SYS$FILESCAN	RMS0FSCN
$FREE	SYS$FREE	RMS0MISC
$FLUSH	SYS$FLUSH	RMS0MISC
$GET	SYS$GET	RMS0GET
$MODIFY	SYS$MODIFY	RMS0MODFY
$OPEN	SYS$OPEN	RMS0OPEN
$PUT	SYS$PUT	RMS0PUT
$READ	SYS$READ	RMS0BLKIO
$RELEASE	SYS$RELEASE	RMS0MISC
$RENAME	SYS$RENAME	RMS0RENAM
$REWIND	SYS$REWIND	RMS0REWIN
$SPACE	SYS$SPACE	RMS0MAGTA
$TRUNCATE	SYS$TRUNCATE	RMS0TRUNC
$UPDATE	SYS$UPDATE	RMS0UPDAT
$WAIT	SYS$WAIT	RMS0WAIT
$WRITE	SYS$WRITE	RMS0BLKIO

SUMMARY

- User-specified data structures (FAB, RAB, XAB, NAM)
- RMS internal data structures in Process I/O Segment
  - Overall control area
  - Process I/O impure area
  - Image I/O impure area
  - File-oriented and record-oriented data structures
- RMS processing
  - RMS dispatching
  - RMS routines and data structures

APPENDIX
RMS FUNCTIONS AND MODULES

Table 2 RMS Functions and Primary Module Names

Function	Entry Module	Comments
High Use Record Operations		
DELETE	RMS0DELET	;DELETE A RECORD
FIND	RMS0FIND	;FIND RECORD
FREE	RMS0MISC	;RELEASE LOCK ON ALL RECORDS
GET	RMS0GET	;GET A RECORD
PUT	RMS0PUT	;PUT A RECORD
READ	RMS0BLKIO	;READ A BLOCK
RELEASE	RMS0MISC	;RELEASE LOCK ON NAMED RECORD
UPDATE	RMS0UPDAT	;REWRITE EXISTING RECORD
WAIT	RMS0WAIT	;STALL FOR RECORD OPERATION COMPLETE
WRITE	RMS0BLKIO	;WRITE BLOCK
Low Use Record Operations		
CLOSE	RMS0CLOSE	;CLOSE FILE
CONNECT	RMS0CONN	;CONNECT RAB
CREATE	RMS0CREAT	;CREATE FILE
DISCONNECT	RMS0DISC	;DISCONNECT RAB
DISPLAY	RMS0DISPL	;DISPLAY FILE INFORMATION
ERASE	RMS0ERASE	;ERASE (DELETE) FILE
EXTEND	RMS0EXTEN	;EXTEND FILE ALLOCATION
FLUSH	RMS0MISC	;FINISH I/O ACTIVITY FOR STREAM
MODIFY	RMS0MODFY	;MODIFY FILE ATTRIBUTES
NXTVOL	RMS0MAGTA	;NEXT VOLUME
OPEN	RMS0OPEN	;OPEN FILE
REWIND	RMS0REWIN	;REWIND FILE
SPACE	RMS0MAGTA	;POSITION FOR TRANSFER
TRUNCATE	RMS0TRUNC	;TRUNCATE FILE
ENTER	RMS0ENTER	;ENTER FILENAME INTO DIRECTORY
PARSE	RMS0PARSE	;PARSE FILENAME SPECIFICATION

Table 2 RMS Functions and Primary Module Names (Cont)

Function	Entry Module	Comments
Low Use Record Operations (cont)		
REMOVE	RMS0SRCH	;REMOVE FILENAME FROM DIRECTORY
RENAME	RMS0RENAM	;RENAME A FILE
SEARCH	RMS0SRCH	;SEARCH A FILE DIRECTORY
SETDDIR	RMS0SETDD	;SET DEFAULT DIRECTORY
SETDFPROT	RMS0SDFP	;SET DEFAULT FILE PROTECTION MASK
RMSRUNDWN	RMS0RNDWN	;PERFORM RUNDOWN ON RMS FILES
RMSRUHNDLR	RMS0RUHND	;RMS Recovery Unit Handler
FILESCAN	RMS0FSCN	;Perform syntax check for file spec
SSVEXC		;GENERATE SYS SERV EXCEPTION

Function Names are used for several symbols

- SYS$function - the symbol used for the RMS vector entry point in P1 space
- RMS$function - the symbol used for the RMS code entry point in S0 space

```
                         F A B
                                        +----+
FAB$B_BID                               |    |       0 (   0)
                                   +----+----+
FAB$B_BLN                          |    |            1 (   1)
                         +---------+----+
FAB$W_IFI                |         |                 2 (   2)
                         +---------+---------+
FAB$L_FOP                |                   |       4 (   4)
                         +-------------------+
FAB$L_STS                |                   |       8 (   8)
                         +-------------------+
FAB$L_STV                |                   |       C (  12)
                         +-------------------+
FAB$L_ALQ                |                   |      10 (  16)
                         +---------+---------+
FAB$W_DEQ                          |         |      14 (  20)
                              +----+---------+
FAB$B_FAC                     |    |                16 (  22)
                         +----+----+
FAB$B_SHR                |    |                     17 (  23)
                         +----+--------------+
FAB$L_CTX                |                   |      18 (  24)
                         +--------------+----+
FAB$B_RTV                               |    |      1C (  28)
                                   +----+----+
FAB$B_ORG                          |    |           1D (  29)
                              +----+----+
FAB$B_RAT                     |    |                1E (  30)
                         +----+----+
FAB$B_RFM                |    |                     1F (  31)
                         +----+          +----+
FAB$B_JOURNAL                            |    |     20 (  32)
                                    +----+----+
FAB$B_RU_FACILITY                   |    |          21 (  33)
                         +---------+----+
unused                   |         |                22 (  34)
                         +---------+---------+
FAB$L_XAB                |                   |      24 (  36)
                         +-------------------+
FAB$L_NAM                |                   |      28 (  40)
                         +-------------------+
FAB$L_FNA                |                   |      2C (  44)
                         +-------------------+
FAB$L_DNA                |                   |      30 (  48)
                         +--------------+----+
FAB$B_FNS                               |    |      34 (  52)
                                   +----+----+
FAB$B_DNS                          |    |           35 (  53)
                         +---------+----+
FAB$W_MRS                |         |                36 (  54)
                         +---------+---------+
FAB$L_MRN                |                   |      38 (  56)
                         +---------+---------+
FAB$W_BLS                          |         |      3C (  60)
                              +----+---------+
FAB$B_BKS                     |    |                3E (  62)
                         +----+----+
FAB$B_FSZ                |    |                     3F (  63)
                         +----+--------------+
FAB$L_DEV                |                   |      40 (  64)
                         +-------------------+
FAB$L_SDC                |                   |      44 (  68)
                         +---------+---------+
FAB$W_GBC                          |         |      48 (  72)
                              +----+---------+
FAB$B_ACMODES                 |    |                4A (  74)
                         +----+----+
FAB$B_RCF                |    |                     4B (  75)
                         +----+
```

VAX-11 RMS OVERVIEW

CONTENTS

Units of Input/Output
A. Blocks
B. Multiblocks
C. Buckets

Terminology
A. Multiple Buffers
B. Global Buffers
C. Window Size
D. READ/AHEAD and WRITE/BEHIND
E. Deferred Write
F. Prologues
G. Fill Factor
H. Bucket Split
I. Segmented Keys
J. Key of Reference
K. Record Stream

RMS Utilities
A. EDIT/FDL
B. CONVERT
C. CONVERT/RECLAIM
D. CREATE/FDL
E. ANALYZE/RMS
F. RMSSHARE

Common Problems and Questions
A. Prologue 3 Indexed File Corruption
B. Global Buffers Cause Performance Degradation
C. Multiple CONVERT's From Same Directory
D. CONVERT/NOFAST Takes Too Long
E. Corrupted Output Files From CONVERT/FAST/NOSORT With Unsorted
F. Meaning of CONVERT/NOSORT
G. SORT Problems With CONVERT's
H. Numeric Sequence Is Not Maintained For Integer Fields Defined as String Keys
I. Convert of Multiple Fixed-Length Inputs Fails With RTB Error
J. CONVERT With Segmented Keys Fails With SEQ or DUP Errors
K. Assorted EDIT/FDL Bugs
L. The RMSSHARE Utility is Obsolete on V4
M. CONVERT/RECLAIM May Corrupt ISAM Files
N. Global Buffers May Cause File Corruption
O. V4 CONVERT Does Not Reduce an ISAM File's Size
P. EDIT/FDL Shows Different Number of Areas Than Final FDL
Q. FDL CONNECT Clauses Do Not Establish Permanent File Attributes
R. EDIT/FDL Does Not Allow Description of Segmented Keys

Units of Input/Output

A. Blocks
A block consists of 512 contiguous bytes. It is the basic unit of disk I/O. Although a program may request a single small record, VAX-11 RMS does not access units smaller than a block. The block is the default I/O unit for sequential files.

B. Multiblocks
A multiblock is two or more blocks that VAX-11 RMS treats as a single I/O unit. Multiblocks are only used by files with sequenti organization. Multiblock count is a run-time attribute. It is se via the SET RMS_DEFAULT/BLOCK_COUNT= comamnd or via the MBC field of the RAB.

C. Buckets
The bucket is the I/O unit for relative and indexed files. A buck consists of from 1 to 32 blocks for V3 releases or from 1 to 63 blocks for V4 releases. Bucket size is set at file creation time by the FDL attributes FILE BUCKET_SIZE or AREA BUCKET_SIZE. Bucke size for a particular file cannot be changed without re-writing th file.

Terminology

A. Multiple Buffers
VAX-11 RMS allows you to use multiple I/O buffers to form a cache memory. The number of I/O buffers is a run-time parameter. It is set via the SET RMS_DEFAULT/BUFFER_COUNT= command or via the MBF field of the RAB.

B. Global Buffers
A global buffer is an I/O buffer that two or more processes can access. If two or more processes are requesting the same information from a file, each process can use the global buffers instead of allocating its own. Global buffers are a creation-time attribute of the file itself. The FDL attribute FILE GLOBAL_BUFFER_COUNT sets the number of global buffers. The SET FILE/GLOBAL_BUFFERS= command also allows the setting of the global buffer count.

C. Window Size
A disk file may be comprised of a variable number of non-contiguou extents. A pointer to each extent resides in the file header. Fo retrieval purposes, the pointers are gathered together in a structure called a window. The default window size is 7 pointers, but it can be set as high as 127 pointers. When an extent is accessed whose pointer is not in the current window, the system ha to read the file header and fetch a new window. This is called a window turn, and requires an I/O operation. Window size is a run-time parameter. It can be set via the RTV field of the FAB. The window size is charged to your buffered I/O byte count quota.

D. READ/AHEAD and WRITE/BEHIND
READ/AHEAD and WRITE/BEHIND operations can be used with sequential files. These operations require two I/O buffers. This allows RMS to overlap processing in one buffer with an I/O request for anothe buffer. These operations are run-time options and can be set via the RAH and WBH options in the ROP field of the RAB.

E. Deferred Write
Deferred write operations are applicable to relative and indexed files. In a deferred write, VAX-11 RMS delays the writing to disk of a modified bucket until the buffer is needed by another bucket. If a subsequent operation makes further modifications to the same buffer while the buffer remains in the cache, performance will improve because fewer I/O operations will occur.

F. Prologues
VAX-11 RMS places certain information about an indexed file in the prologue. The information includes file attributes, key descriptors, and area descriptors. There are three types of prologues--Prologue 1, Prologue 2, and Prologue 3. Prologue 1 fil may have multiple string-type keys. Prologue 2 files also may hav multiple keys, but not all are string keys. Prologue 3 files can have only a primary string-type key on V3 releases but can have al key types on V4 releases. Prologue 3 files allow for file compression.

G. Fill Factor
When an indexed file is loaded, space can be reserved for future record insertions by specifying a fill factor. A fill factor of less than 100% will cause part of each bucket to be left vacant by the initial load operation. The intent is to reduce bucket splits

H. Bucket Split
With indexed files, an attempt to insert a record into a full buck causes a bucket split. RMS tries to keep half of the records in t original bucket and moves the other records to a newly created bucket. Each of the moved records leaves behind a pointer to the new bucket. These pointers are called Record Reference Vectors (RRV). When the system searches for one of the records that moved it must first go to the bucket where the record used to reside, re the RRV, and then move to the new bucket. Bucket splits cause ext I/O operations to retrieve records, and the RRV's may use significant disk space.

I. Segmented Keys
A key which is composed of two or more non-contiguous sub-fields i a segmented key. Each key may have up to eight segments. Segment keys must be string type.

J. Key of Reference
The key of reference is indicated in the KRF field of the RAB. It is only applicable to indexed files. It specifies the key (primar first alternate, etc.) to which the operation applies.

K. Record Stream
A record stream is the logical association of a RAB with a FAB. T record stream is established by storing the address of the FAB in the RAB and issueing a $CONNECT macro.

[. RMS Utilities

A. EDIT/FDL
EDIT/FDL creates and modifies FDL files. FDL files provide specifications for VAX-11 RMS data files; these specifications can then be used by certain utilities to create data files.

B. CONVERT
The CONVERT utility copies records from one or more files to an output file, changing the record format and file organization to that of the output file as specified by an FDL. CONVERT does not change file data. Since a new file is written, RFA access is not preserved.

C. CONVERT/RECLAIM
The CONVERT/RECLAIM utility reclaims empty buckets in Prologue 3 indexed files so that new records can be written in them. Since no new file is written, RFA access to the file is preserved.

D. CREATE/FDL
The CREATE/FDL utility uses the specifications in an existing FDL file to create a new, empty data file.

E. ANALYZE/RMS
The ANALYZE/RMS_FILE utility provides a set of facilities for analyzing the internal structure of a VAX-11 RMS file. The following four functions are available:
1. Check the structure of a file for errors.
2. Generate a statistical report on the file's structure and use.
3. Enter an interactive mode through which the file structure can be explored.
4. Generate an FDL file from an existing data file.

F. RMSSHARE
The RMSSHARE utility was available on V3 releases; it is no longer available on V4 releases. It is used primarily as a system management tool to perform the following functions:
1. Enable the VAX-11 RMS file sharing capability by initializing file sharing structures in paged dynamic memory and set the maximum number of pages that the structures can occupy. The file sharing capability must be enabled each time the system is booted.
2. Display figures on allowable and actual usage and increase the maximum number of pages that the file sharing structures can occupy.

Common Problems and Questions

A. Prologue 3 Indexed File Corruption
When RMS attempts to add a record into a Prologue 3 data bucket, i seeks to avoid a bucket split by retrieving space within the bucke This space retrieval is performed by compressing the bucket to recover space occupied by deleted records. When DATA_KEY_COMPRESSION is enabled, the size of the compressed keys m vary among records. This possible size difference is not taken in account, and as a result, the bucket compression can result in fil corruption. The most common symptom is that ANALYZE/RMS reports "Data record spills over into free space of bucket." The problem can be avoided either by disabling DATA_KEY_COMPRESSION or by converting to a Prologue 1 file.

B. Global Buffers Cause Performance Degradation
A frequent complaint is that after enabling global buffers, system performance suffers. Most frequently this is observed as increase system-wide paging. It is possible to dedicate so much memory to global buffers that system performance suffers. In such case the customer will have to compromise between what he would like to hav and what his hardware complement will accomodate.

C. Multiple CONVERT's From Same Directory
CONVERT creates a workfile (CONVWORK.TMP) in the default directory for the process. The file is used to communicate with SORT-32. When SORT-32 is finished, it uses the file to communicate with CONVERT. SORT-32 does not return either a file ID or a complete filespec to CONVERT. If multiple CONVERT's are being run simultaneously using the same default directory, there will be multiple versions of the workfile. After the SORT completes, CONVERT may access the wrong version of the workfile. Symptoms are BADLOGIC, ISI, and IFI errors. This problem was corrected by V4.

D. CONVERT/NOFAST Takes Too Long
CONVERT/FAST loads a file by building a complete bucket in memory and then issueing block I/O writes to the file. This saves time b reducing the number of I/O requests necessary to load the file. A CONVERT/NOFAST operation actually adds records to a file using RMS calls to add each record. The number of I/O's required is much higher for a /NOFAST operation. The difference in time required f the two techniques differs by a factor of 10. Customers will frequently feel that something is wrong because a /NOFAST operatio takes so long.

E. Corrupted Output Files From CONVERT/FAST/NOSORT With Unsorted Input
File corruption will result when CONVERT/FAST/NOSORT is specified if the input is not sorted in ascending sequence by primary key. The resulting file will have a corrupted index structure.

F. Meaning of CONVERT/NOSORT
The /NOSORT qualifier on the CONVERT command applies only to the primary key. This can cause confusion when CONVERT/NOSORT is run against a file with multiple keys. There is no way to specify tha sorting will not be done on alternate keys.

G. SORT Problems With CONVERT's
CONVERT invokes SORT-32. A CONVERT may fail with sort errors. In this case the problem should be treated as a SORT problem. There a SID entry which discusses reasons for SORT failures and suggeste approaches for resolution.

H. Numeric Sequence Is Not Maintained For Integer Fields Defined as String Keys
Customers will sometimes define an integer field as a string key o define a string key that overlaps an integer field. This is acceptable to RMS. However, records will not be retrieved accordi to the numeric sequence as may be expected. The reason is that th most significant byte of an integer field is the leftmost byte whi the most significant byte of an ASCII string is the rightmost byte

I. Convert of Multiple Fixed-Length Inputs Fails With RTB Error
On V4.0 and V4.1 releases, CONVERT may fail with an RTB error when it should not. The problem is always seen when multiple input files have been provided for the CONVERT. The record format is fixed-length records. Use of the /TRUNCATE qualifier on the CONVE command will allow the CONVERT to succeed without lost data. The problem is corrected in the V4.2 release.

J. CONVERT With Segmented Keys Fails With SEQ or DUP Errors
A CONVERT to a prologue 1 or 2 file format will usually fail with SEQ or DUP errors if the output file has segmented keys. The problem occurs because CONVERT compares each key segment seperatel without considering the result of comparisons on previous key segments. A workaround is to use prologue 3 output causing CONVER to use different code for the key comparisons. The problem is corrected in V4.2.

K. Assorted EDIT/FDL Bugs
The EDIT/FDL utility contains many errors and frequently does not behave as expected. In general, the DESIGN scripts are reliable, but much of the rest of the utility is not reliable. The utility is being completely re-written.

L. The RMSSHARE Utility is Obsolete on V4
The RMSSHARE utility is no longer used on V4 releases. The lock manager has been re-written to accomodate cluster-wide locking. The new lock manager uses whatever it needs from dynamic memory. It doesn't require RMSSHARE to establish limits for it. As far as I know, there is no documentation explaining this.

M. CONVERT/RECLAIM May Corrupt ISAM Files
The CONVERT/RECLAIM utility may cause corruption of prologue 3 ISAM files with DATA_KEY_COMPRESSION enabled. The utility does not fully understand the DATA_KEY_COMPRESSION algorithm. This problem was first discovered on V4 releases but apparently existed on V3 releases also. The problem is corrected in V4.2.

N. Global Buffers May Cause File Corruption
The use of global buffers may result in corrupted files. The problem can occur when RMS needs to copy a global buffer to a process local buffer. RMS releases the lock on the global buffer before moving the data. It is possible that the buffer will be re-used and the data modified before it is copied. Later, data from the corrupted buffer can be written back to the file. The problem is corrected on V4.2.

O. V4 CONVERT Does Not Reduce an ISAM File's Size
On V3 releases, the CONVERT utility could re-organize an ISAM file into a smaller file if the original file had unused space. This was often the desired behavior. The V4 CONVERT behaves differentl It produces an output file the same size as the input file. The behavior can be avoided by generating an FDL, deleting or adjustin the allocation clauses, and using CONVERT/FDL to re-organize the file.

P. EDIT/FDL Shows Different Number of Areas Than Final FDL
After designing an ISAM file, the resulting FDL may be viewed within the EDIT/FDL utility. The FDL displayed will indicate 2 areas per key. However, the FDL file that is written by the utili may have a different number of areas. This is the result of the GRANULARITY setting in the EDIT/FDL session. By default it will cause an FDL with 3 areas to be written. It is reasonable to expe that the FDL written to the disk is identical to the one displayed at the end of the DESIGN script. However, this is apparently intentional behavior and is not regarded as a bug.

Q. FDL CONNECT Clauses Do Not Establish Permanent File Attributes
The file definition language for V4 includes a CONNECT section whi allows the FDL to specify runtime parameters for the program using the FDL. This has caused confusion with customers who thought the CONNECT section was specifying permanent file attributes.

R. EDIT/FDL Does Not Allow Description of Segmented Keys
The EDIT/FDL DESIGN script does not allow the description of segmented keys. The workaround is to describe the key as though i is not segmented and then use EDT to add the segment descriptions to the resulting FDL.

VAX/VMS V4.0
PAUL SENN, VMS PERFORMANCE GROUP, ZKO1-1/D19, 264-8312

RMS FILE AND RECORD PROCESSING OPTIONS WHICH AFFECT BUFFER FLUSHING

The following is a summary of the way various RMS features interact to determine when RMS buffers are flushed. The relevant RMS options (bits in the FOP and the ROP) with with a short definition are listed below. Some examples are also presented that illustrate how these options interact.

Read-ahead/Write-behind

- Applies only to sequential files

- Set by RAH and WBH bits in the ROP

Read-ahead:

When the user issues a request that causes RMS to switch to a new I/O buffer (for instance, a $GET for a record that is not currently buffered), RMS issues asynchronous QIOs to read data into as many available buffers as the user has. Thus, the user is allowed to process the records in the first buffer while reads into the other buffers are completing.

Write-behind:

When the user issues a request that results in RMS writing a buffer to disk, RMS issues an asynchronous QIO to do the write. This allows the user to begin processing on a second buffer while the first buffer write is completing.

Asynchronous I/O

- Applies to all file organizations

- Set by ASY bit in the ROP

Setting this bit allows the user to get control back immediately from RMS, rather than RMS waiting for I/O completion before returning to the user. Notice that write-behind and read-ahead also cause asynchronous I/O.

Except for some special cases, when write-behind and read-ahead are enabled, setting the ASY bit has no effect, since asynchronous I/O is taking place anyway. The relationship between write-behind and the ASY bit is further explained later in the sequential file example.

Deferred-write

- o Set by DFW bit in the FOP for relative and indexed files
- o Always enabled for sequential files (the DFW bit has no effect)
- o The meaning of deferred write is slightly different for sequential files than it is for relative and indexed files, as described below:

Deferred write allows the user to take maximum advantage of the RMS I/O buffers when performing operations that add, delete, or modify records. Without deferred write, every $PUT, $UPDATE, or $DELETE results in at least one direct I/O operation. If deferred write is turned on, it is possible for the user to, for instance, perform multiple sequential $PUTs while incurring only one direct I/O (assuming that multiple records can fit in one I/O buffer).

If deferred write is turned on, RMS must make a decision as to when to write modified I/O buffers to disk. In general, for any file organization, modified buffers are written to disk if:

- o User issues a $FLUSH
- o File is closed

The other conditions under which buffers are written vary, depending on the file organization. For sequential files, a modified buffer is written as soon as RMS moves on from that buffer to a new buffer. For instance, if the user is doing sequential $PUTs, as soon as RMS finds the first buffer is full and it is necessary to start using the second buffer, the first buffer is written out. For relative and indexed files, on the other hand, a modified buffer is not written to disk until the buffer is needed for another operation. The examples below clarify this. (Also, a modified buffer can be written to disk as a result of a blocking AST being delivered, in the case of shared files.)

Deferred write can result in substantial performance gains, and it is usually best that it be turned on. However, it is not always appropriate. For example, in a high-contention environment where frequent concurrent updating is occurring, turning on deferred write can actually cause a performance degradation because of the extra load introduced by blocking AST activity. There is also an interaction between the use of global buffers and the use of deferred write. Enabling deferred write causes local buffers, rather than global buffers to be used for modified buckets. This introduces extra processing overhead, and turning on deferred write with global buffers can cause a performance degradation for processes which do not frequently reaccess buffers after modifying them. Those who are afraid of losing data as a result of a system crash might want to turn off deferred write (for relative and indexed files), thereby ensuring that every update to the file is written to disk immediately. For sequential files, the same effect can be achieved by issuing a $FLUSH, which causes all modified I/O buffers to be written out. The effect of having deferred write enabled when a system crash occurs is further discussed in the indexed file examples below.

The following examples all assume that eight records can fit in one RMS I/O buffer. Also, a multibuffer count of two is assumed, unless otherwise stated.

Sequential File Examples

- $PUTs to a sequential file

No write-behind, synchronous:

After eight $PUTs, the first buffer is full (assuming block spanning isn't allowed). The ninth record added goes into the second buffer. At this time, the first buffer is written out. RMS does not return control to the user until the write is complete. The next eight $PUTs go into the second buffer, with no direct I/O taking place. On the 17th $PUT, RMS moves back to the first buffer and writes the second buffer out, again not returning control until the write is complete.

Write-behind, synchronous:

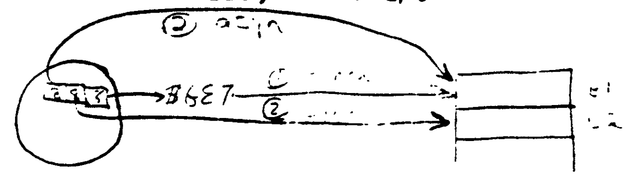
This scenario is the same as above, except that on the ninth and 17th $PUTs, RMS returns control to the user immediately, instead of waiting for the write to complete. This allows the user to begin filling a new buffer while a buffer write is completing behind his back. Note that for this example, turning on write-behind means that the user never has to stall for I/O completion (as long as the I/O device is fast enough and the system load is such that the buffer writes can complete before the buffers need to be reused).

Write-behind, asynchronous:

Because write-behind causes asynchronous I/O, setting the ASY bit in this example has no effect, except for one special case. This is the case where the write of an I/O buffer is not completed before the buffer needs to be reused. Referring to the example above, after eight $PUTs, the first buffer is full. The ninth $PUT goes into the second buffer. At this time, an asynchronous request is issued to write the first buffer out. After eight more $PUTs, the second buffer is full, and RMS issues another asynchronous request to write this buffer out. The 17th record needs to go back into the first buffer. But, because of the speed of the I/O device or the system load, suppose the first asynchronous buffer write is not yet completed. In this case, there is no place for the 17th record to go, since I/O is in progress on both buffers. The setting of the ASY bit affects what action RMS takes in this situation. If the ASY bit is set, RMS returns control to the user immediately, even though the 17th $PUT is not yet completed. Note that the burden is on the user to refrain from modifying the local storage (containing the data for the 17th record) until the asynchronous $PUT completes. If the ASY bit is not set, the user does not get control back until the 17th record is successfully moved into the I/O buffer.

No write-behind, asynchronous:

In this case, after eight $PUTs, RMS issues a request to write the first buffer and return control immediately to the user. However, because write-behind is not turned on, the user is not able to begin filling the second buffer until the first buffer write completes. If another $PUT is issued before the write completes, RMS returns a "Record Stream Active" error code. This illustrates the conceptual difference between write-behind and the ASY bit. The purpose of write-behind is, for a given record stream, to allow the user to make use of one RMS buffer at the same time as I/O is in progress on another buffer. Setting the ASY bit does not provide this capability. Instead, setting the ASY bit gives the user the chance to perform operations totally unrelated to the record stream, such as computation or reads of other files, while I/O is in progress.



o $GETs from a sequential file

Read-ahead, synchronous:

The first $GET causes eight records to be read into the first buffer. RMS stalls until this read is complete. On the ninth $GET, RMS reads records 9 to 16 into the second buffer. Again RMS stalls until the read is complete. However, in addition to this synchronous read that occurs on the ninth $GET, RMS also performs an asynchronous read to fill the first buffer. This request completes while the user is reading the second buffer, so that by the time the user does the 17th $GET, the read into the first buffer is completed and the record is available to the user. The only $GETs the user has to wait for in this example are the first two that are needed to fill the buffers initially); the rest of the I/O can be done asynchronously while the user is reading another buffer (as in the case of write-behind, this is assuming the I/O device is fast enough and the system load is such that the reads complete before the buffers are needed)

The relationship between read-ahead and ASY is analogous to that of write-behind and ASY. If read-ahead is set, only in special cases will setting the ASY bit make a difference. These cases are the initial two buffer reads, for which RMS stalls unless the ASY bit is set, and where the user returns to a buffer before the asynchronous read into the buffer is completed. In this case, similarly to write-behind, if the ASY bit is set, the user gets control back immediately, even though the $GET is not complete. Again, like write-behind, the purpose of read-ahead is for a given record stream, to allow the user to make use of one RMS buffer at the same time as I/O is in progress on another buffer. Normally read-ahead and write-behind are enabled together.

Relative File Examples

o Sequential $PUTs to a relative file

Synchronous, no deferred write:

After each $PUT, RMS writes the modified buffer to disk. Control does not return to the user after the $PUT until the write is complete.

Asynchronous, no deferred write:

After each $PUT, RMS writes the modified buffer to disk, as in the example above. However, rather then waiting for the write to complete before returning to the user, RMS returns control immediately.

Synchronous, deferred write:

After a total of 16 $PUTs, both buffers are full. At this point, nothing is yet written to disk--all 16 records are sitting in the two RMS buffers. On the 17th $PUT, the first buffer needs to be reused. Therefore, at this time, it is written out. The user does not get control back from the 17th $PUT until the write to disk completes.

Asynchronous, deferred write:

This is identical to the example above, except that on the 17th $PUT, rather then waiting for the write to complete before returning to the user, RMS returns control immediately. Again, as in the asynchronous, write- behind sequential file example above, the burden is on the user to refrain from modifying the local storage (containing the data for the 17th record) until the user's asynchronous $PUT completes.

Indexed File Examples

In terms of the interaction of the ASY bit and deferred write, the example above for relative files applies to indexed files also. But the choice of which I/O buffer to flush is made more complicated by the presence of index as well as data buckets. The following example gives some idea of the extra considerations involved in indexed file buffer flushing.

- o Sequential $PUTs (on the key of reference) to an indexed file
- o Multibuffer count = 3
- o The index structure required for the transaction can fit in one bucket, no sharing

Synchronous, deferred write:

Assume that the file was previously loaded with a low fill factor, so now no bucket splits are occurring. Also assume that because of previous activity, the index bucket necessary for the transaction is already cached. (Once read in, the index bucket remains cached for the rest of the transaction, since index buckets are less likely to be thrown out than data buckets.) Buffer 1 contains the index, and buffers 2 and 3 are used for data. After eight $PUTs, buffer 2 is full. The ninth record added goes into buffer 3. After a total of 16 $PUTs, all buffers are full. At this point, nothing is written to disk yet; the modified index bucket and the 16 records are sitting in three RMS buffers. On the 17th $PUT, a buffer needs to be reused. Therefore, at this time, buffer 2 is written out (not buffer 1 since this is an index bucket). Control is not returned to the user from this $PUT until the buffer is written. Note that in this example, the index bucket is not written to disk until the file is closed, since the caching algorithm prevents it from being thrown out of the cache as long as a data bucket is available to be thrown out. However, even if the system crashes before the index bucket is written out, file integrity is maintained because of the existence of pointers at the data bucket level.

VAX RMS File Sharing Internals

VAX/VMS Development
Digital Equipment Corporation
110 Spit Brook Road
Nashua, New Hampshire 03062

WHY FILE SHARING?

- goal: to allow multiple accessors of a file, where at least one accessor is modifying the file, to obtain a consistent view of the file at all times

- RMS objects locked for file sharing

normal	w/global buffers
file	global section
bucket	global buffer
record	

- backround: read Chapter 6, "File Sharing and Buffering", in Guide to VAX/VMS File Applications

VMS V4.4 ENHANCEMENTS

- full support for shared sequential files
  - pre-V4.4: fixed, 512-byte sequential files only
  - V4.4: any type of sequential file
  - includes multiple appenders

- improved file and global buffer section locking performance
  - reduced lock manager usage

OVERVIEW OF RMS SOURCE FILES

- conventions for code modules (RMxxxxxxx.MAR/.B32)

RMS0	RMS service entry points (RMS$xxx)
RM0	org.-independent support routines
RM1	sequential file organization
RM2	relative file organization
RM3	indexed sequential file organization
NT0	network support (NT$xxx)

  e.g. RMS0PUT.MAR, RM1PUT.MAR, RM2PUT.MAR, RM3PUT.B32

- in-memory data structure definitions

RMSINTSTR.SDL	general internal structures
RMSSHR.SDL	file sharing structures
RMSFWADEF.SDL	filename processing structures
DAPDEF.SDL	DAP support structures

- on-disk structures (internal file formats)

RMSFILSTR.SDL	RMS on-disk file structures

- RMS user interface (part of STARLET)

RMS32MAC.MAR	MACRO-32 macros
RMSCALLS.MAR	" "
RMSMAC.REQ	BLISS-32 macros
RMSUSR.SDL	RMS user structures (FAB, RAB, XAB, etc.)

- message files

RMSDEF.MSG	general RMS error message
RMSFALMSG.MSG	DAP error messages

- miscellaneous internal definition files

  RMSIDXDEF.R32
  RMSIDXLNK.R32
  RMSIDXMAC.R32
  RMSMSCMAC.MAR

RMS FILE SHARING MODULES

RM0SHARE.MAR (.B32 for V4.4)	file sharing routines
RM0CACHE.MAR	bucket cache routines
RM0RELEAS.MAR	bucket release routines
RM0RECLCK.MAR	record locking support
RM0RCLCK2.B32	"" ""

FILE LOCKING

- what it is and why it's there
    - to synchronize FILE level operations
- relevant user interface fields

FAB$B_SHR	SHRGET - allow readers SHRPUT,SHRUPD,SHRDEL - allow writers NIL - prohibit sharing by others UPI - user provides interlocking (no sharing) MSE - multistreaming
FAB$B_FAC	GET,PUT,UPD,DEL,TRN

- when is sharing done?

	FAC read	FAC write
SHR read	NO	YES
SHR write	YES	YES

FILE LOCK DETAILS

- root resource

- resource name

  RMS$ + file-id + DEVLOCKNAM (unique volume id)

  - RMS$
    facility code - convention for all resource names
  - file-id
    unique file identification across a volume set
  - DVI$_DEVLOCKNAM
    16-byte cluster-wide unique name for a volume set

- lock modes used

PW	exclusive access to file (open, close, extend, truncate)
CR (currently) NL (V4.4)	read access to file

  concurrency:
  - sequential and relative lock file for all operation
  - indexed files only lock file around extends

- data structures

  SFSB (shared file synchronization block)

  - allocated at $OPEN/$CREATE time

- how to examine SFSB

  SDA> SHOW PROCESS/RMS=IFB

  ...get SFSB address from IFB$L_SFSB_PTR above

  SDA> READ SYS$SYSTEM:RMSDEF.STB

  SDA> FORMAT/TYPE=SFSB address

FILE LOCK VALUE BLOCK

```
+------------------+----------+----------+
|    future use    |   MBC    | LVB_VER  |   0
+------------------+----------+----------+
|       LRL        |          FFB        |   4
+------------------+---------------------+
|         HBK - Hi VBN allocated         |   8
+----------------------------------------+
|         EBK - end of file VBN          |  12
+----------------------------------------+
```

- contents (16 bytes)
    - critical file header information (HBK/EBK/FFB/LRL)
    - shared sequential file common multi-block count value (MBC)
    - file lock protocol version number (LVB_VER)

- how used:

    1) raise to PW - current value block returned

    2) modify file (write, extend)

    3) lower to CR (NL in V4.4) - new value block stored

- V4.4 enhancement
    - file lock NOT lowered -- held with blocking AST
    - blocking AST routine gives up lock if not in use
      else marks file lock "wanted"
    - benefit: reduced lock manager operations (in most cases)
    - example:
      - process A holds file lock in PW
      - process B requests file lock in PW

      1. blocking AST occurs in process A
      2. if A is done, converts file lock to NL
      3. process B's PW lock is granted

- if value block is zero, must be first accessor
    - fill in lock value block: HBK, EBK, etc.

APPEND LOCK

- what it is and why it's there
    - new in V4.4
    - append = $PUT to EOF on a stream $CONNECTed to EOF
    - used only for shared sequential files

- synchronizes append operations:
    - helps ensure temporal ordering of records
    - avoids wasted bucket locking:

        process B holds bucket lock on EOF bucket

        process A
            requests file lock; determines EOF
            requests bucket lock -- waits

        process B writes records past EOF, releases bucket lo

        process A gets file lock, finds EOF has moved,
            now must release old bucket lock and
            get new one

- lock usage
    - sublock of file lock
    - resource name: "APPENDER"
    - lock modes: EX, NL
    - no value block

- how/when used
    - created on first append
    - when done, held with blocking AST (like file lock in V4.4)

    1. acquire file lock
    2. acquire append lock

        (may stall here, causing file lock to be lowered to NL)

    3. re-acquire file lock
    4. based on current EOF, acquire bucket lock

+

EXTEND LOCK

- what it is and why it's there
  - synchronizes file extend operations
  - needed because file lock cannot be held during stall for $EXTEND I/O
  - must prevent other extenders
- lock usage
  - special convention: uses standard bucket lock, VBN=1
- how/when used
  - RM$CACHE request for a lock, no read, no buffer on VBN 1
    - allocates a BLB, but no BDB
  - VBN 1 (prologue) bucket cannot be previously locked

    (ISAM and relative ensure this)

BUCKET LOCKING

- synchronizes access to buckets
- relevant user interface fields

  RAB$B_MBF multi-buffer count
- data structures

  BDB (buffer descriptor block) - one per I/O buffer

  BLB (buffer lock block) - one per bucket lock

  - #BLB = #BDB + 1 [+ #GBPB]
  - allocated at $CONNECT time
- bucket lock details
  - resource name is VBN of bucket (4 bytes)
  - when modifying, held in EX
  - when done, lowered to NL
  - value block is cache sequence number

BUCKET CACHE

- each bucket in cache has a sequence number

- to get a bucket

  1) search local cache for bucket

  2) enqueue for bucket lock in EX
     - sequence # returned in value block

  3) if found, then
     if sequence #'s match
     then use it
     else read new copy from disk

- goal: keep index buckets in cache

  - each bucket has a cache "value"

    - derived from depth of bucket in index structure

  - if cache flush required, data buckets thrown out first

DEFERRED WRITE

- buffer written if owner process needs the buffer
  or when another process needs the bucket

- incompatible lock retained on dirty buffers

  - when done with a bucket, lower to PW with a blocking AST

  - blocking AST causes writeback and then lock lowered to NL

RECORD LOCKING

- synchronizes access to records

- relevant user interface fields

  RAB$B_ROP RLK, REA, RRL, ULK, NLK

- data structures

  RLB (record lock block)

  - describes one locked record

  - allocated on demand (demand usually = 1 unless ULK)

- record lock details

  - resource name is RFA
    (plus 2 bytes pad to longword align)

  - no value block

  - lock modes

EX	exclusive access to record (default)
PW	lock record for write, allow readers (RLK)
PR	lock record for read, allow readers (REA)
CR	used for query locking

GLOBAL BUFFERS

- shared, processor-wide RMS buffer cache

  - buffers shared by processes on that node

  - transparent to application program

- how to use

  - mark file: SET FILE/GLOBAL_BUFFERS=n MYFILE.DAT

  - specify count in FAB$W_GBC at $CONNECT

GLOBAL BUFFER CACHE

- one global buffer cache per file, per CPU

- implemented as a read/write, system page file global section

    - created (using $CRMPSC) by first (shared) accessor to file (see RM1CONN.MAR)

    - section name is "RMS$" + virt. address of FCB of file

        e.g. RMS$804038A0

- data structures

    GBH (global buffer header)

        - one per section

    GBD (global buffer descriptor)

        - one per buffer in section

        - linked list (self-relative)

- buffer flush

    - uses "short scan"

    - scans 8 buffers; lowest cache value kicked out

    - last buffer is "aged" (cache value decremented, min=0)

- no unowned dirty buffers exist in global cache

    - high update environment may lose

    - however, if request is to cache a bucket for update, and bucket not in global cache, local cache is used

GLOBAL SECTION LOCK

- parallel of file lock

- if global buffers specified, enqueue for global section lock
  - resource name, lock modes same as file lock
  - child of EXE$GL_SYSID_LOCK

- if lock value block zero, must be first accessor
  - create global buffer section
  - process FILE lock converted to a system lock
    - used as parent for global buffer system locks
  - process acquires a new file lock in PW

- two different resources, (# processes * 2) + 1 locks

  1. global section, child of CPU-specific resource

     1 lock per process accessing section

  2. file

     1 lock per process accessing file
     plus 1 system lock to act as parent for global buffer

- global section lock value block

```
+------------------+-------------------+
|       GBC        |       GBREF       |
+------------------+-------------------+
|    GS_SIZE - size of GS in bytes     |
+--------------------------------------+
|                spare                 |
+--------------------------------------+
|                spare                 |
+--------------------------------------+
```

  - number of global buffers in section (GBC)
  - number of accessors to global section (GBREF)
  - size of global section in bytes (GB_SIZE)

- data structures

  GBSB (global buffer synchronization block)

  - parallel to SFSB

GLOBAL BUFFER LOCKING

- parallel to bucket locks
  - same resource name, lock modes, value block
  - but, when first released, converted to system lock
- data structures

  GBPB (global buffer pointer block)

  - parallel to BDB (looks like one)
  - 2 allocated per stream at $CONNECT time

  BLB - same as for local buffer locking
- caching a bucket in the global buffer cache

  1. enqueue for global section lock in PW
     - scan global buffer cache
     - if bucket in cache
then
    increment use count in GBD
else
    find a free bucket in cache (may have to flus
    initialize GBD
  2. lower global section lock to NL
  3. initialize GBPB from GBD
  4. enqueue for bucket lock in EX
     - note: if bucket was in cache, 2 locks now exist
  5. read bucket from disk into global buffer if not in cache
- release of bucket to cache

```
if first accessor
then
    convert process bucket lock to system lock in NL,
        parent = system file lock
else
    simply dequeue process lock on buffer
        (NL system lock remains)
```

GLOBAL BUFFER STATISTICS

- find some process with file marked for global buffers open

  SDA> SHOW PROCESS/RMS=GBH

  - cache hits, misses, reads, writes

  - see RMSINTSTR.SDL for details

GLOBAL BUFFER QUOTA

- why needed?

  - global buffers use system locks

  - no quota on system locks

- new SYSGEN parameter: RMS_GBLBUFQUO

  - total number of global buffers allowed to be cached at once
    (= # of system locks used by global buffers)

- decremented when a new buffer is released to cache (RM$RELEASE)

- incremented when a buffer is kicked out of cache (RM$CACHE)

- maintained with ADAWI instruction (so it works w/multi-processors)

FAILOVER

- failure case: change on disk completes, then node fails
  - lock value block not yet written
  - hence, data in lock value block not valid
- next enqueue for file lock gets SS$_VALNOTVALID
- file lock
  - re-read file header and maximizes with lock value block
  - correct value written to value block when file locked lower
  - summary: no loss of file integrity
- bucket lock
  - don't use cached bucket
  - forces a re-read from disk
- global section lock: cannot occur
- record locks: don't have a value block

DEADLOCKS

- file and global section locks: cannot occur
- bucket locks: RMS retries
- record locks
  - RMS$_DEADLOCK returned to user
  - can only happen with manual record locking

VMS in a Multiprocessing Environment

INTRODUCTION

VAX/VMS systems have evolved from a single DECnet node system to a complex set of multiprocessor computers. DIGITAL offers two such multiprocessor systems:

1. VAX-11/782

2. VAXcluster

Each configuration offers a different set of features and environments. In this module we will discuss these two types of multiprocessors, their primary hardware components, and an overview of how VMS operates in each.

OBJECTIVES

1. To describe the different multiprocessing implementations.

2. To describe the trade-offs for each environment.

RESOURCES

Reading

- Guide to VAXclusters

Source Modules

Facility Name	Module Name
MP	MPINIT MPSCBVEC MPLOAD

TOPICS

I. Loosely Coupled Processors

II. Tightly Coupled Processors (VAX-11/782)

   A. MP.EXE structures

   B. Scheduling differences

   C. Start-up/shutdown

III. Clustered Processors

MULTIPROCESSING ENVIRONMENTS

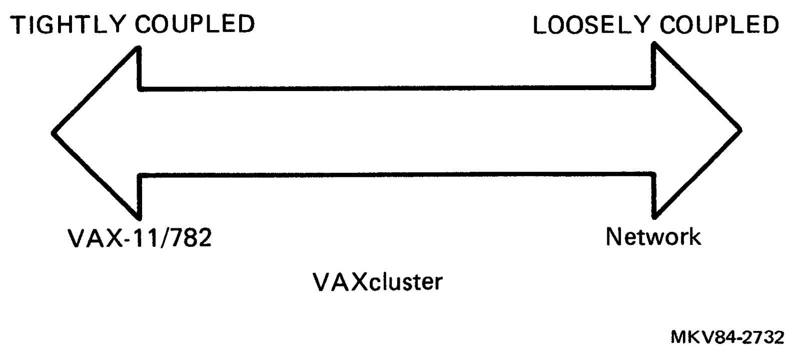


Figure 1 Relationship Between Different Multiprocessing Configurations

Loosely Coupled:	Each processor executes a separate copy of the operating system.
Tightly Coupled:	Both processors share the same copy of the operating system.
VAXcluster:	Each processor executes its own copy of the operating system but is coordinating certain activities with the other processors.

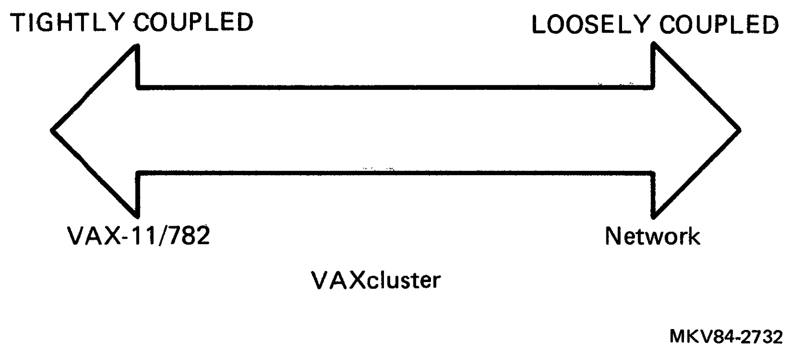


Figure 2 Relationship Between Different Multiprocessing Configurations

Table 1 Different Multiprocessing Implementations

System Characteristic	VAX-11/782	VAXcluster	Network
CPU booting	Together	Separate	Separate
CPU failure	Together	Separate	Separate
CPU cabinet location	Single or adjacent	Same computer room	Can be widely separated
Security/Management domain	Single	Single	Multiple
File system	Integrated	Integrated	Separate
Growth potential	Limited	Very great	Very great

NETWORKS

- Each system in a network is an independent system; it boots and fails separately from the other systems in the network.
- Failure to enter the network does not effect the system's ability to perform local processing.
- Systems in a network are in a separate protection and management domain.
- Systems on a network have their own file system.
- Networks have a powerful growth potential. You can generally add many, many additional nodes in a network, very loosely coupled to each other.

THE VAX-11/782

- Systems boot and fail together
- A single protection and management domain
- No concept of assigning a different UIC to a user based on which processor they are actually running on
- Integrated file system
- Limited growth potential
- The processors tend to be a single cabinet or several cabinets near each other in one machine room.

Definitions

Loosely Coupled:	Each processor executes a separate copy of the operating system. This is good for high-availability.
Tightly Coupled:	Both processors share the same copy of the operating system.
Symmetric:	All processors execute all the operating system code.
Asymmetric:	All processors cannot execute all the operating system code.
Primary Processor:	The CPU that executes kernel mode code as well as executive, supervisor, and user.
Secondary Processor:	The CPU that cannot execute kernel mode code. It executes executive, supervisor, and user mode code.

The VAX-11/782 is a tightly coupled multiprocessing system that is asymmetric for kernel mode and symmetric for the other modes.

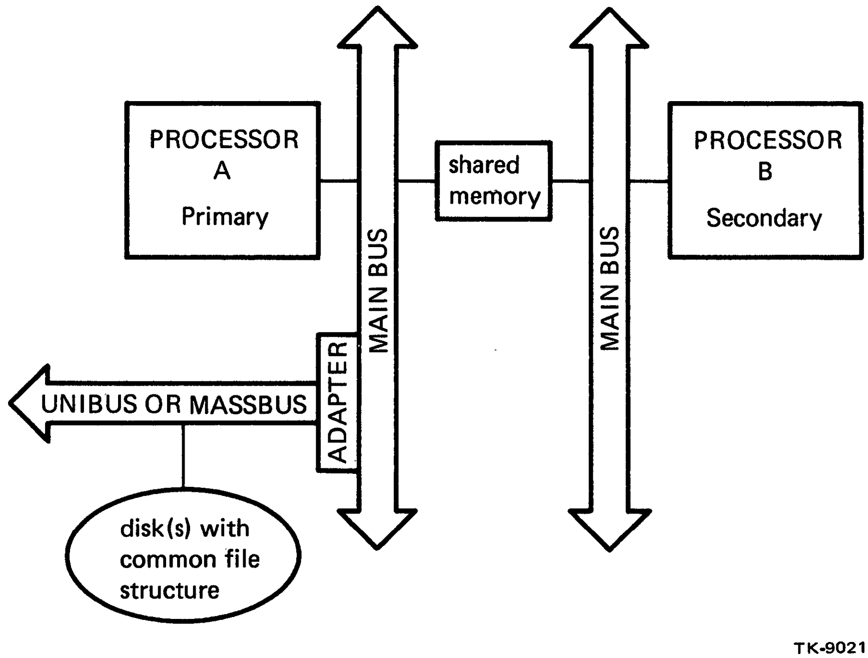
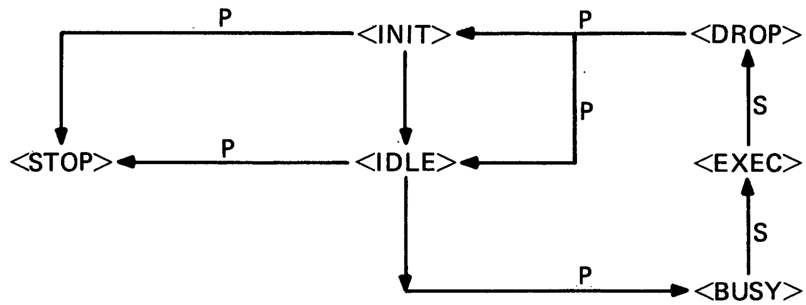


Figure 3 Sample VAX-11/782 Configuration

- Initialization
  - Start primary processor.
  - After the normal system is booted, a privileged program (MP.EXE) is run in the site-specific command file. MP.EXE is activated by the START/CPU DCL command.
  - Start secondary processor. Accomplished by booting with an abbreviated command file in CSA1.



P = PRIMARY MAKES TRANSITION
S = SECONDARY MAKES TRANSITION

TK-9013

Figure 4 Secondary Processor States

```
<INIT>      Processor state when MP.EXE runs.
  |
  v
<IDLE>      After MP.EXE (initialization code runs)
  |
  v
<BUSY>      When a process is found for CPU2
  |
  v
<EXECUTE>   After a LDPCTX instruction is issued
  |
  v
<DROP>      At quantum end or kernel mode request
            by CPU2, a SVPCTX issued, the state
            changed to DROP, interrupt primary.
  |
  v
<IDLE>      After CPU1 takes back process

<STOP>      Requested by system manager ($STOP/CPU)
            Requested by CPU1
```

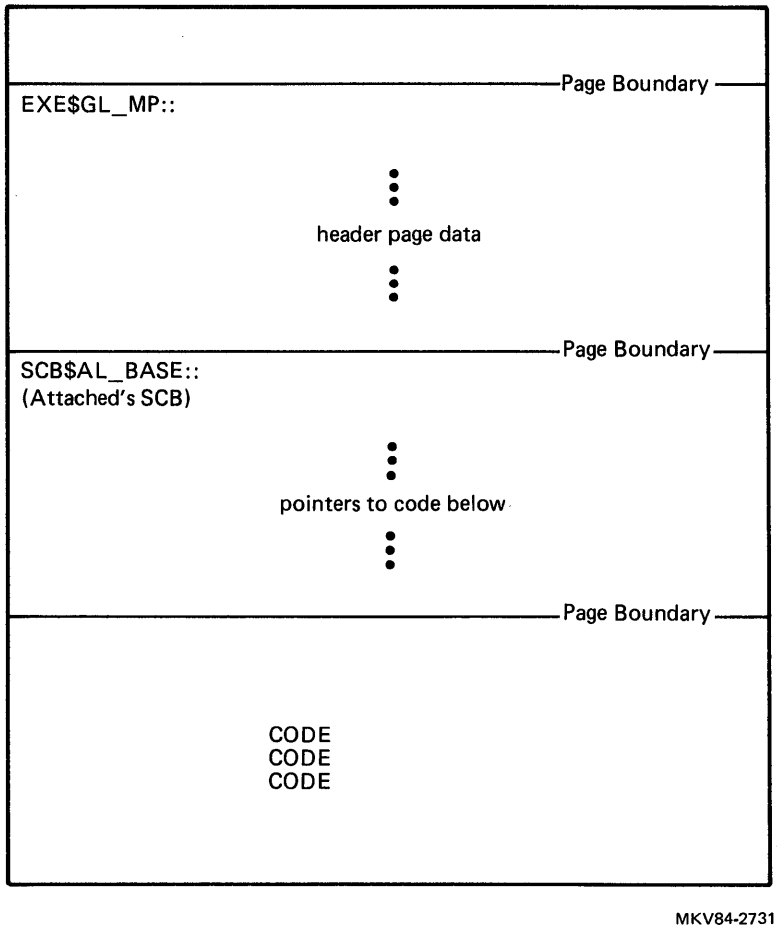


Figure 5 MP.EXE in Nonpaged Pool

CODE section from figure contains:

- Addresses being pointed to by almost all of the SCB vectors
- System locations that are jumped to as a result of being modified by the MP.EXE code.

VAXclusters

- A new system organization that combines features of both multiprocessors and networks.
- A system organization that is positioned in the middle of the spectrum, in between tightly coupled systems and loosely coupled systems.
- Systems boot and fail separately.
- A single protection and management domain.
- The SYSUAF.DAT files must be coordinated.
- VAXclusters have a powerful growth potential.
- Typically in a single computer room, due to the hardware restrictions on the length of a Computer Interconnect (CI) cable.

VAXcluster Benefits

- Incremental system expansion
- System availability
- Data Sharing

  Public Volumes shared across the cluster
- Broader cost/performance range
- Usage of existing equipment

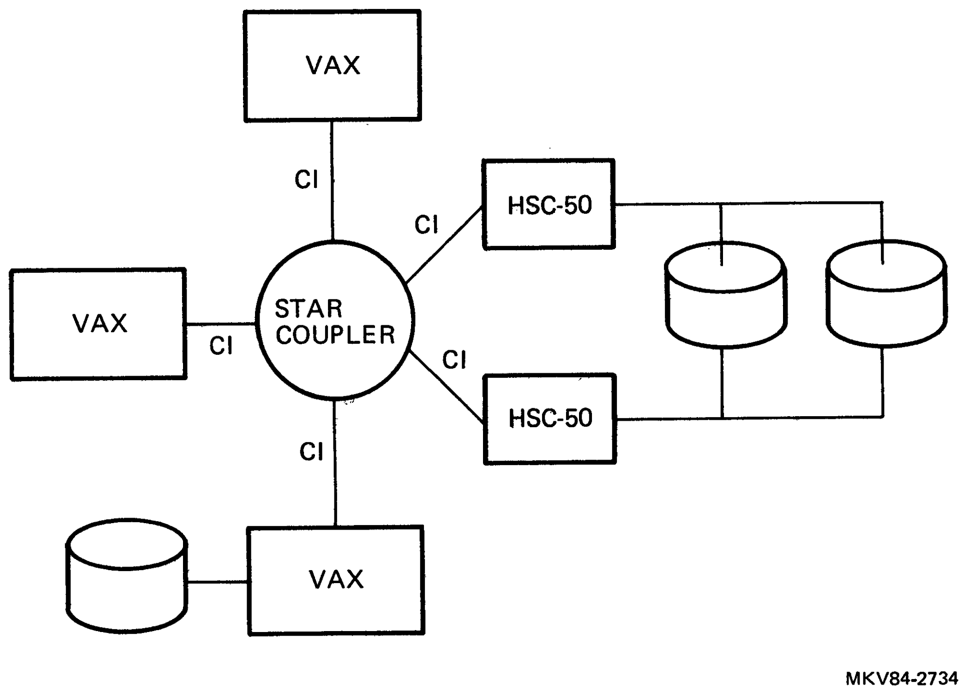


Figure 6 VAXcluster Hardware Configuration

- Two or more VAXen connected together
- CI cables connect each VAX
- Star Coupler
- HSC-50
- RA Disks
- Local Disk(s)

SUMMARY

Table 2 Different Multiprocessing Implementations

System Characteristic	VAX-11/782	VAXcluster	Network
CPU booting	Together	Separate	Separate
CPU failure	Together	Separate	Separate
CPU cabinet location	Single or adjacent	Same computer room	Can be widely separated
Security/Management domain	Single	Single	Multiple
File system	Integrated	Integrated	Separate
Growth potential	Limited	Very great	Very great

GLOSSARY

CLUSTER

This term is commonly used in the VMS context in an imprecise manner to denote a VAXcluster.

VAXcluster

Loosely coupled collection of VMS Systems and HSC-50s where the entire VMS Cluster forms a single domain for the purposes of integrity and security. A high degree of coordination and sharing is supported across nodes in the VAXcluster. A VAXcluster contains two types of nodes. An "active node" is a VAX/VMS System that is cognizant of its membership in the VAXcluster. A "passive node" is exemplified by an HSC-50 that is not cognizant of the VAXcluster.

ACTIVE NODE

Node participating in cluster connection management, therefore having knowledge of the VAXcluster. This is in contrast to a Passive Node which has no knowledge of the VAXcluster.

CI (Computer Interconnect)

High-speed (70 megabits/second), highly available communications system for interconnecting up to 16 VAX-11/780, VAX-11/782, VAX-11/750, HSC-50.

CLUSTER NODE

The unit of a VAXcluster.

HSC-50 (Hierarchical Storage Controller)

Intelligent disk and tape controller interfaced to the CI. This controller supports the MSCP protocol for access to these devices.

MSCP (Mass Storage Control Protocol)

Protocol for logical/physical access to disks and tapes implemented in the HSC-50 and the VMS MSCP server.

MSCP SERVER

VMS component that supports VAXcluster access to disks connected to a VAX using the MSCP protocol from any node within a VAXcluster.

NETWORK

Loosely coupled collection of machines (nodes) where each network node is an independent domain for the purposes of integrity and security.

NETWORK NODE

The unit of a network.

PASSIVE NODE

Node not participating in cluster connection management. A passive node has no knowledge of the VAXcluster. This is in contrast to an active node that is aware of the VAXcluster.

VMS SYSTEM

A single VAX CPU (including VAX-11/782) running under the control of the VAX/VMS operating system.

APPENDIX
THE VAX-11/782

Definitions

Loosely Coupled:	Each processor executes a separate copy of the operating system. This is good for high-availability.
Tightly Coupled:	Both processors share the same copy of the operating system.
Symmetric:	All processors execute all the operating system code.
Asymmetric:	All processors cannot execute all the operating system code.
Primary Processor:	The CPU that executes kernel mode code as well as executive, supervisor, and user.
Secondary Processor:	The CPU that cannot execute kernel-mode code. It executes executive, supervisor, and user mode code.

The VAX-11/782 is a tightly coupled multiprocessing system that is asymmetric for kernel mode and symmetric for the other modes.

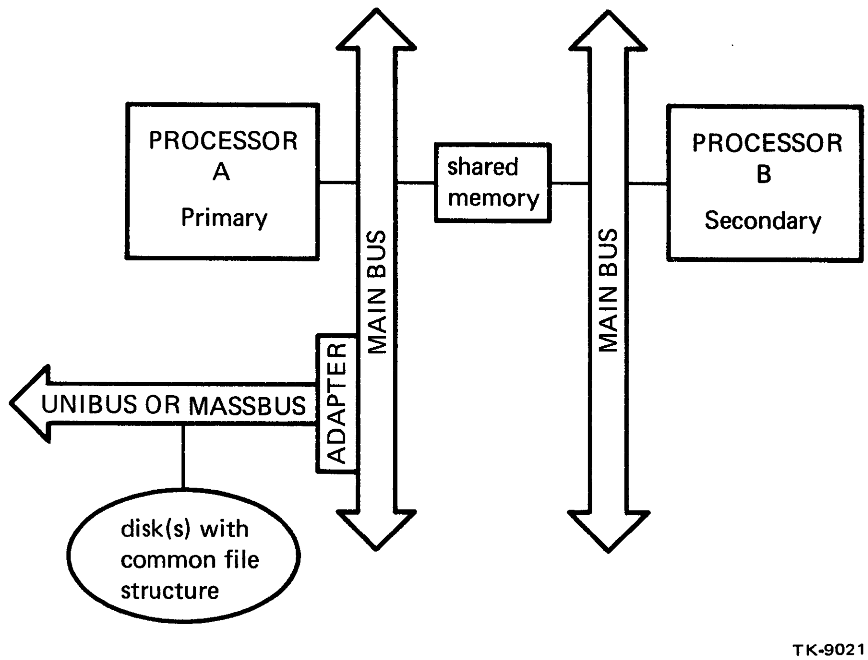


Figure 7 Sample VAX-11/782 Configuration

- Two VAX-11/780s connected to the same shared memory.
- The primary (on the left) has the I/O devices. The secondary or attached processor (on the right) has just the CPU.
- Minimum local physical memory (256Kb) on each CPU for diagnostics only.
- All information in the shared memory.
- Eight Meg maximum physical memory for the shared memory.
- Primary processor runs all interrupt and kernel mode code. Both processors run executive, supervisor, and user mode code.
- Multiprocessing code takes approximately 8K bytes (16 pages) in nonpaged pool.

Initialization

- Start primary processor.

  The DEFBOO.CMD file used to boot the primary processor "requests" that the MA780 memory be used instead of the local physical memory. The memory on MA780 #1 starts at physical address 0.

- After the normal system is booted, a privileged program (MP.EXE) is run in the site-specific command file. MP.EXE is activated by the START/CPU DCL command. MP.EXE does the following:

  - Allocates nonpaged pool and loads in the MP code.
  - Connects the 'hooks' into the VMS code (discussed later).
  - New SCB initialized for the secondary CPU.
  - Primary SCB slightly modified to handle

    Scheduling code for secondary processor
    MA780 interrupt communication

- Start secondary processor. Accomplished by booting with an abbreviated command file in CSA1. This results in:

  - Initialization of memory configuration
  - Starting execution at address in RPB.

Hooks into VMS

- Naming Conventions

  - MPH$samename

    Indicates a routine that will be entirely replaced by a MP routine of the same name.

  - MPH$newnameHK

    Indicates a location of a hook to additional MP code (instead of a replacement).

  - MPH$newnameCONT

    Indicates a location where additional MP code will return to normal flow of code.

  Locations of Hooks (Executive Module Names)

  - ASTDEL — AST delivery and queuing
  - BUGCHECK — BUGCHECK for both processors
  - PAGEFAULT — Translation buffer invalidations
  - SCHED — Process scheduling and rescheduling

SCB Changes

- CPU2

  New SCB created for the secondary processor in nonpaged pool. This SCB points to different routines than those used by the primary CPU.

- CPU1

  - MA780 vectors redirected to point to new MP primary CPU interrupt routine.

  - IPL=5 SCB interrupt vector now contains address of MP secondary scheduling routine.

  - XDELTA interrupt is moved from IPL=5 to IPL=F.

Secondary Processor States

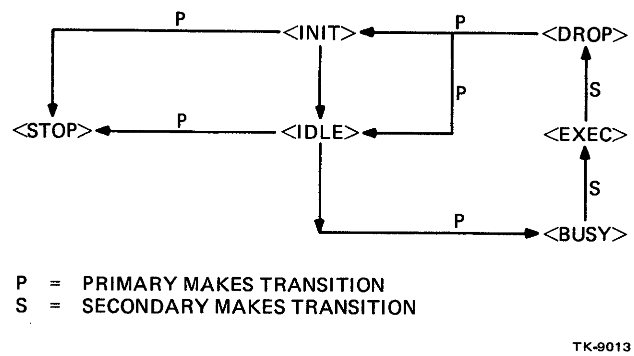


Figure 8 Secondary Processor States

The current state of the secondary processor is recorded in the state variable in nonpaged pool. The contents of this variable (the state) is used by the primary processor to determine whether to schedule work for the secondary processor or not.

```
<INIT>      Processor state when MP.EXE runs.
  |
  v
<IDLE>      After MP.EXE (initialization code runs)
  |
  v
<BUSY>      When a process is found for CPU2
  |
  v
<EXECUTE>   After a LDPCTX instruction is issued
  |
  v
<DROP>      At quantum end or kernel mode request
            by CPU2, a SVPCTX issued, the state
            changed to DROP, interrupt primary.
  |
  v
<IDLE>      After CPU1 takes back process

<STOP>      Requested by system manager ($STOP/CPU)
            Requested by CPU1
```

If there is no process for CPU2 to run, an AST is used to indicate when a process falls below the kernel mode level. The AST delivery is turned into a rescheduling interrupt.

Exceptions for CPU2

- If there is a transition to kernel mode:
  - A SVPCTX is issued
  - Process is "handed" to CPU1
- AST delivery and quantum end both execute special code.
- A separate SCB for the secondary processor allows the enforcement of the rules.

MA780

- Has the ability to interrupt either processor.
- Reasons for CPU1 to interrupt CPU2
  - To request an invalidation of a System Virtual Address
  - AST has arrived for process on CPU2
  - BUGCHECK
- Reasons for CPU2 to interrupt CPU1
  - To request rescheduling
  - Log an error
  - Request a BUGCHECK

Faults

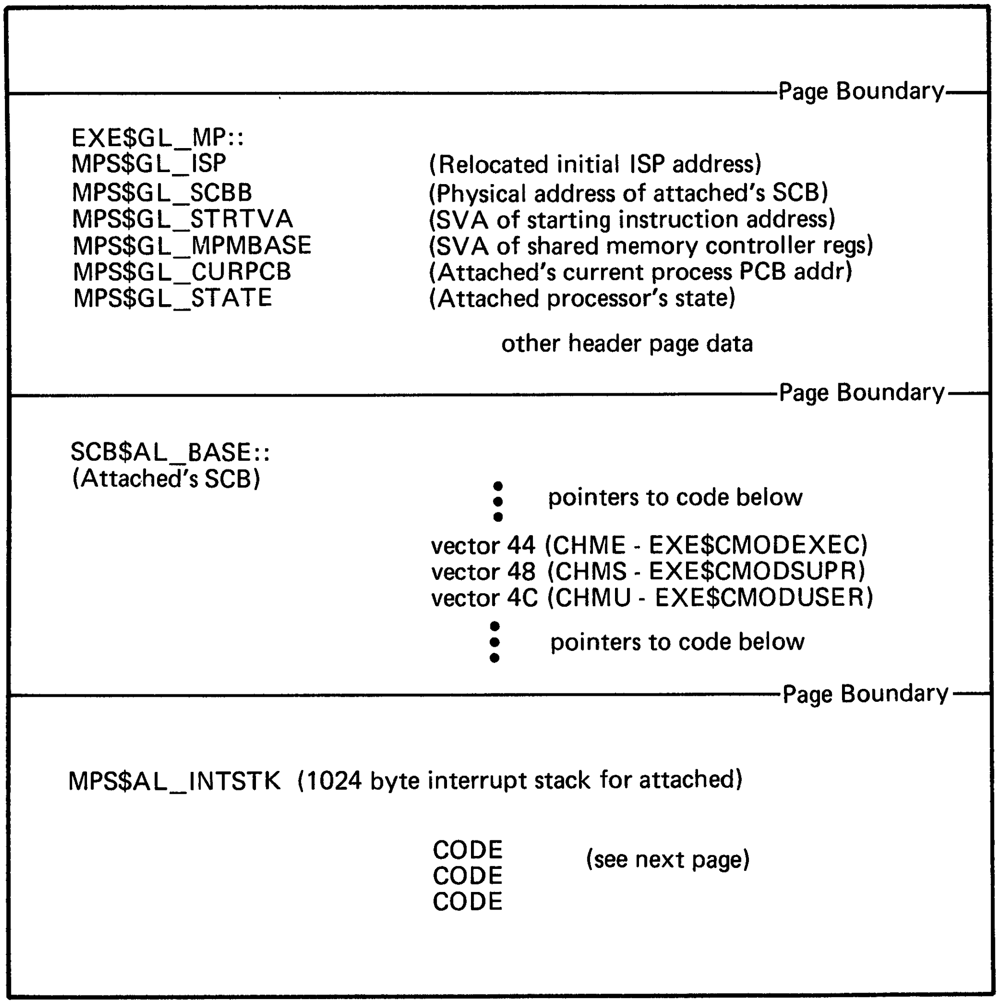
- POWERFAIL
  - If CPU2 goes, CPU1 continues
  - If CPU1 goes, CPU2 waits
- BUGCHECK
  - If a BUGCHECK occurs, CPU2 goes IDLE while CPU1 writes the sysdump file and reboots.
- MACHINE CHECK
  - Like normal VMS

Restrictions

- The processors must be "twins" (ECO,WCS level, FPA)
- First MA780 must be at physical address 0
- Same TR # for the MA780s on both CPUs
- No DR780
- No high-speed RP07s (2.2 MB), 1.3 MB allowed

NOTE

Before MP.EXE runs, the RPB contains a self-jump loop. After MP.EXE runs, RPB contains the address of the secondary CPU start-up code. In this way the secondary CPU (CPU2) can be started before the primary CPU (CPU1) is finished booting.



MKV84-2730

Figure 9 MP.EXE Loaded into Nonpaged Pool

CODE Section from figure on previous page contains:

1. Addresses being pointed to by almost all of the SCB vectors

2. System locations that are jumped to as a result of being modified by the MP.EXE code (See Table 3).

Table 3 System Locations and the Resulting MP Locations

System Locations	MP Locations
SCH$SCHED	SCH$MSCHED
SCH$RESCHED	SCH$MRESCHED
MPH$QAST	MPS$QAST
MMG$INVALIDATE	MPS$INVALID
MPH$BUGCHKH	MPS$BUGVHECK
MPH$ASTDELHK	MPS$ASTSCHEDCHK
MPH$NEWLVLHK	MPS$ASTNEWLVL

VMS in a VAXcluster Environment

INTRODUCTION

VAXclusters, like DECnet, give a whole new dimension to VAX/VMS. This module gives an overview of the important topics in a VAXcluster. It also discusses how some of the VMS features covered earlier in this course are extended to the VAXcluster environment, including:

- Cluster processes
- Cluster synchronization and communication mechanisms
- Cluster start-up
- Cluster shutdown

We do not propose to cover the internals of a VAXcluster in this section.

OBJECTIVES

To assist in managing a VAXcluster, the student must understand:

1. The differences between performing operations, such as the following, on a single VAX system, and performing them on a VAX in a cluster:

   - System start-up
   - System shutdown

2. The additional VMS synchronization and communication mechanisms used on a system in a VAXcluster.

RESOURCES

Reading

- VAX/VMS Cluster Technical Summary
- Guide to VAXclusters

Source Modules

Facility Name	Module Name
SYSLOA	CLUSTRLOA SCSLOA
OPCOM	OPCCRASH

TOPICS

I. Cluster Synchronization and Communication Mechanisms

   A. Distributed lock manager

   B. Distributed Job Controller

   C. Interprocessor communication

II. System Initialization and Shutdown Differences

   A. VMB, INIT and SYSINIT differences

   B. Joining a cluster

   C. Leaving a cluster

III. Additional Considerations in a VAXcluster Environment

IV. SYSGEN Parameters Relevant to a VAXcluster

OVERVIEW OF VAXcluster FEATURES

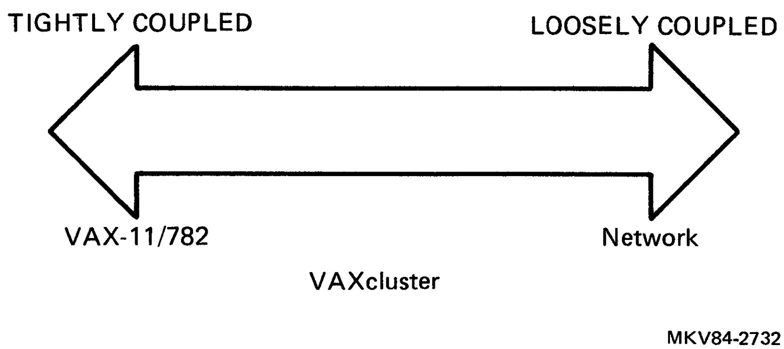


Figure 1 Relationships Between Different Multiprocessor Configurations

Table 1 Different Multiprocessing Implementations

System Characteristic	VAX-11/782	VAXcluster	Network
CPU booting	Together	Separate	Separate
CPU failure	Together	Separate	Separate
CPU cabinet location	Single or adjacent	Same computer room	Can be widely separated
Security/Management domain	Single	Single	Multiple
File system	Integrated	Integrated	Separate
Growth potential	Limited	Very great	Very great

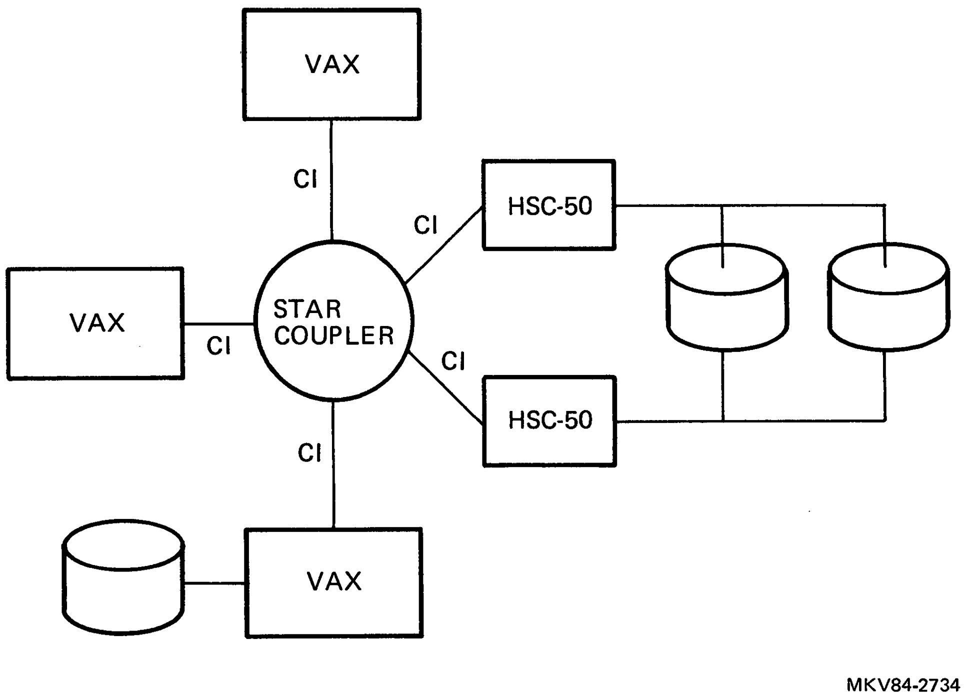


Figure 2 Sample VAXcluster Hardware Configuration

```
VAX/VMS V4.0 on node COMICS  6-OCT-1984 10:40:57.65  Uptime  0 02:22:14
  Pid      Process Name     State  Pri      I/O        CPU       Page flts
20800080 NULL             COM      0        0   0 00:18:42.40          0
20800081 SWAPPER          HIB     16        0   0 00:00:21.10          0
20800085 ERRFMT           HIB      7     1165   0 00:00:09.92        140
20800086 CACHE_SERVER     HIB     16      213   0 00:00:04.59         56
20800087 CLUSTER_SERVER   HIB     10       23   0 00:00:00.34        133
20800088 OPCOM            LEF      8      202   0 00:00:02.15        181
20800089 JOB_CONTROL      HIB      8     2336   0 00:00:36.37        188
2080008B CONFIGURE        HIB      9       55   0 00:00:00.44        137
2080008D SYMBIONT_0001    COM      4     1377   0 00:08:26.51       2613
2080008E SPIDERMAN        LEF      4     2412   0 00:00:34.72        699
20800090 NETACP           HIB      9     2835   0 00:00:53.49       5800
20800091 EVL              HIB      4       79   0 00:00:02.52       2138
20800092 REMACP           HIB      9       74   0 00:00:00.56        123
20800094 THE_FLASH        LEF      7      947   0 00:00:15.53       2886
2080009A BATMAN           LEF      7     6659   0 00:02:20.76       8142
2080009B CAPT_MARVEL      LEF      7    13420   0 00:08:46.85      32485
2080009D DR_STRANGE       LEF      4    11665   0 00:04:05.12      23536
208000A3 SILVER_SURFER    LEF      4      923   0 00:00:30.45       2075
208000BC KAL-EL           LEF      4     3879   0 00:01:46.67       9493
208000C6 MR_FANTASTIC     LEF      4     6042   0 00:01:07.37       6730
208000C7 SYSTEM           LEF      4     3998   0 00:00:44.44       2375
208000CD DR_XAVIER        LEF      4      702   0 00:00:19.65       2671
208000D9 BATCH_891        COM      4     4033   0 00:03:25.23      13888
208000E6 BRUCE_BANNER     LEF      4      259   0 00:00:05.79        952
208000E7 JON_JONES        LEF      4     1030   0 00:00:16.58       2718
208000ED BATCH_924        COM      4      862   0 00:00:36.38       2646
```

Example 1 SHOW SYSTEM Output for a VAXcluster

- Additional system processes
  - CACHE_SERVER
  - CLUSTER_SERVER
  - CONFIGURE
- In a VAXcluster, high bits of PIDs are nonzero

System Processes in a VAXcluster

Table 2 System Processes Specific to a VAXcluster

Process Name	Priority	Image Name	Comments
CACHE_SERVER	16	FILESERV.EXE	Flushes the system-wide caches
CLUSTER_SERVER	8	CSP.EXE	Envelope for cluster jobs
CONFIGURE	8	CONFIGURE.EXE	Dynamic device configuration manager

Table 3 VAXcluster Processes Created by STARTUP.COM

Process Name	Error Log File	Privileges	UIC
CACHE_SERVER	cache_server_error	all	[1,4]
CLUSTER_SERVER	cluster_server_error	all	[1,4]
CONFIGURE	configure_error	CMKRNL,PRMMBX, BYPASS,SHARE	[1,4]

- CACHE_SERVER and CLUSTER_SERVER are only created if system is member of a VAXcluster
- CONFIGURE is only created if device PAA0: exists
- All images reside in SYS$SYSTEM
- All error log files reside in SYS$MANAGER

Cache Server Process

- Agent to flush the system-wide XQP caches
- Cache flush must be done by a process because XQPs execute in process context.

Cluster Server Process

- Provides a process context for use by other systems in the VAXcluster
- Can be used by any application
- Currently used to do cluster operator communication (OPCOM), and cluster broadcast

Configure Process

- Created if there is a CI780/CI750 on your VAX
- Configures the I/O data base on-line, if and when disks are added to your HSCs

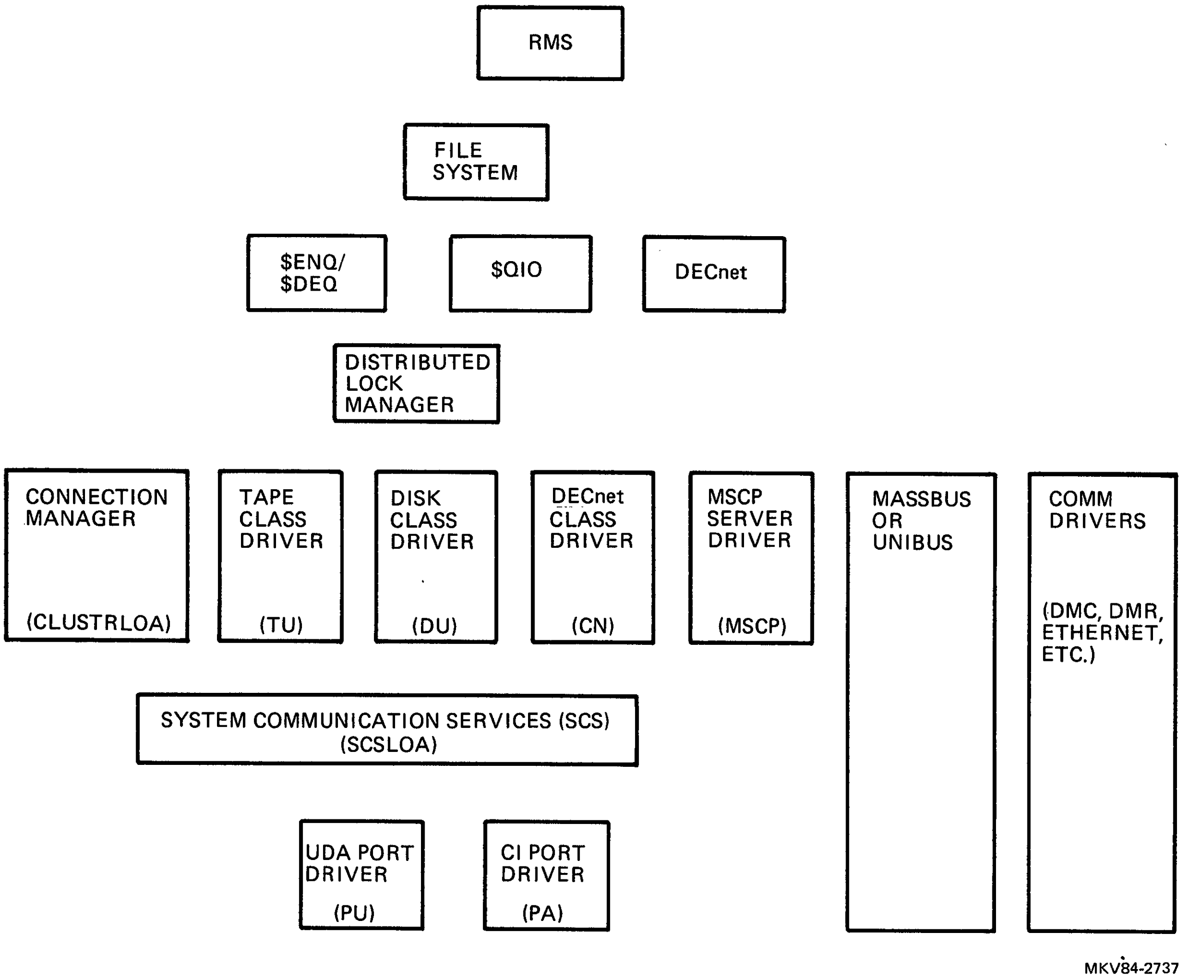


Figure 3 VAXcluster Software Components

- Connection manager
- Distributed lock manager
- File system and RMS
- Class driver
- SCS
- Port driver

The Connection Manager

- Part of the executive but is loadable code
- Primary purpose is to determine and maintain cluster membership
- Synchronizes state changes
- Provides an acknowledged message delivery service
- Prevents partitioning
- Also provides a cluster system ID (CSID)
- No user level access to the connection manager, only an internal interface

Distributed Lock Manager

- Provides cluster-wide synchronization for many VMS components
- The lock manager implements the ENQ and DEQ services
- Used by:
  - File system
  - RMS
  - Job controller
  - Some system services (ALLOCate, MOUNT, ASSIGN)
- Also available to user applications

Distributed File System

- Allows files to be accessed on any system as if they were local to that system
- Files-11 ODS-2 ACP is procedure based and called XQP
  - Resides in P1 space
  - No context switches required for file operations
  - Some additional overhead for ENQ and DEQ
  - File system becomes multithreaded
  - File caches are now perprocess

Record Management Services (RMS)

- All file sharing capabilities available on a Version 3 system are now available in a VAXcluster
- RMS uses the lock manager for its synchronization

Class Driver

- A class driver is a device-independent driver that has been written for a particular series of devices (for example, disks, tapes, terminals). It contains the user interface to the $QIO routines.

SCS (Systems Communications Services)

- SCS provides the process and system addressing, connection management, and flow control necessary to multiplex the basic port-to-port data services among multiple users.

Port Drivers

- A port driver contains the device-specific portions of the driver. It is written to communicate with a specific device under the class it is in (for example, Disks DR, DB, DM).

Distributed Batch and Print Services

- The Job Controller is now distributed across the cluster
- Job controllers in a VAXcluster coordinate operations using the lock manager
- Batch and print services are distributed across a VAXcluster
- Queue file is implemented as an RMS shared file
- Can have a generic cluster print queue that feeds specific printer queues on different systems
- Can also have a generic cluster batch queue
  - Batch queue chosen is the one that will have the lowest percentage of jobs_running/job_limit

I/O in a VAXcluster Environment

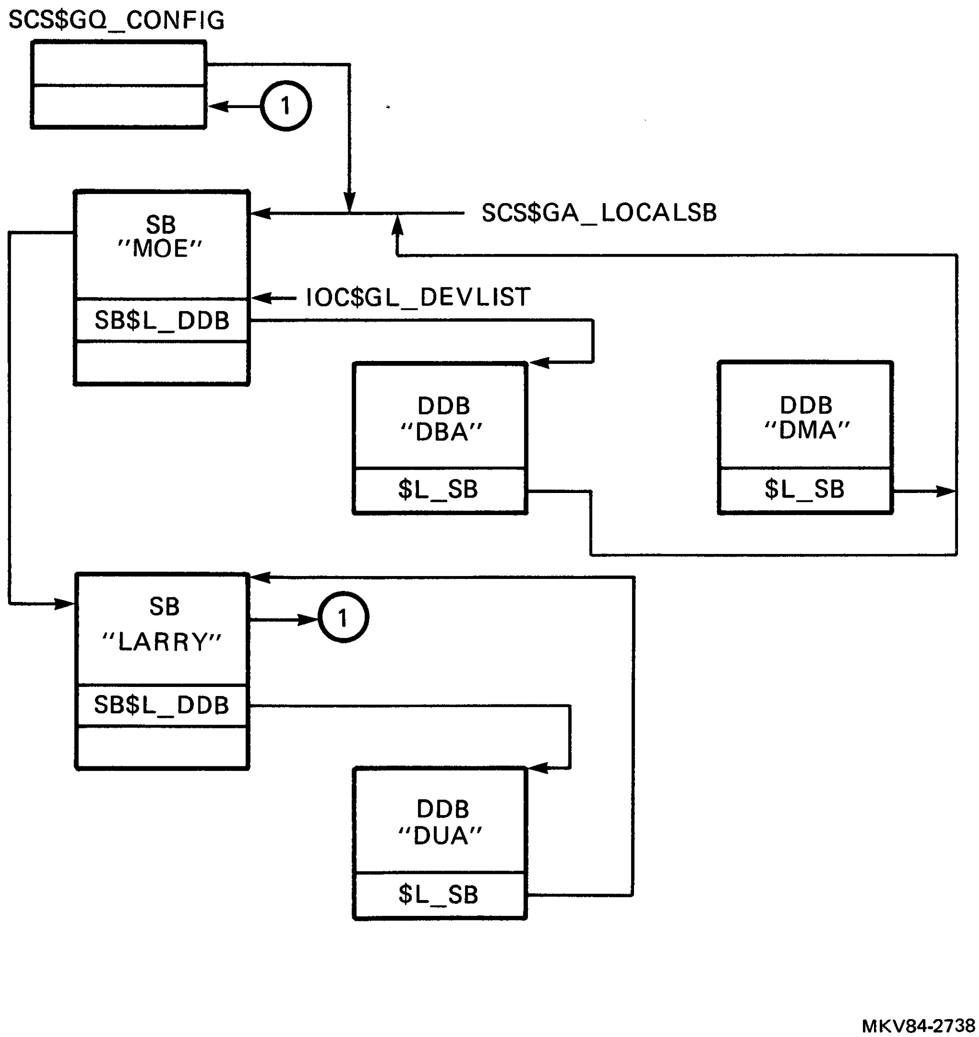


Figure 4 Cluster I/O Database

- Each cluster node has a System Block (SB)
- Local SB points to regular I/O database
- Remote SBs point to remote DDBs that are locally accessible

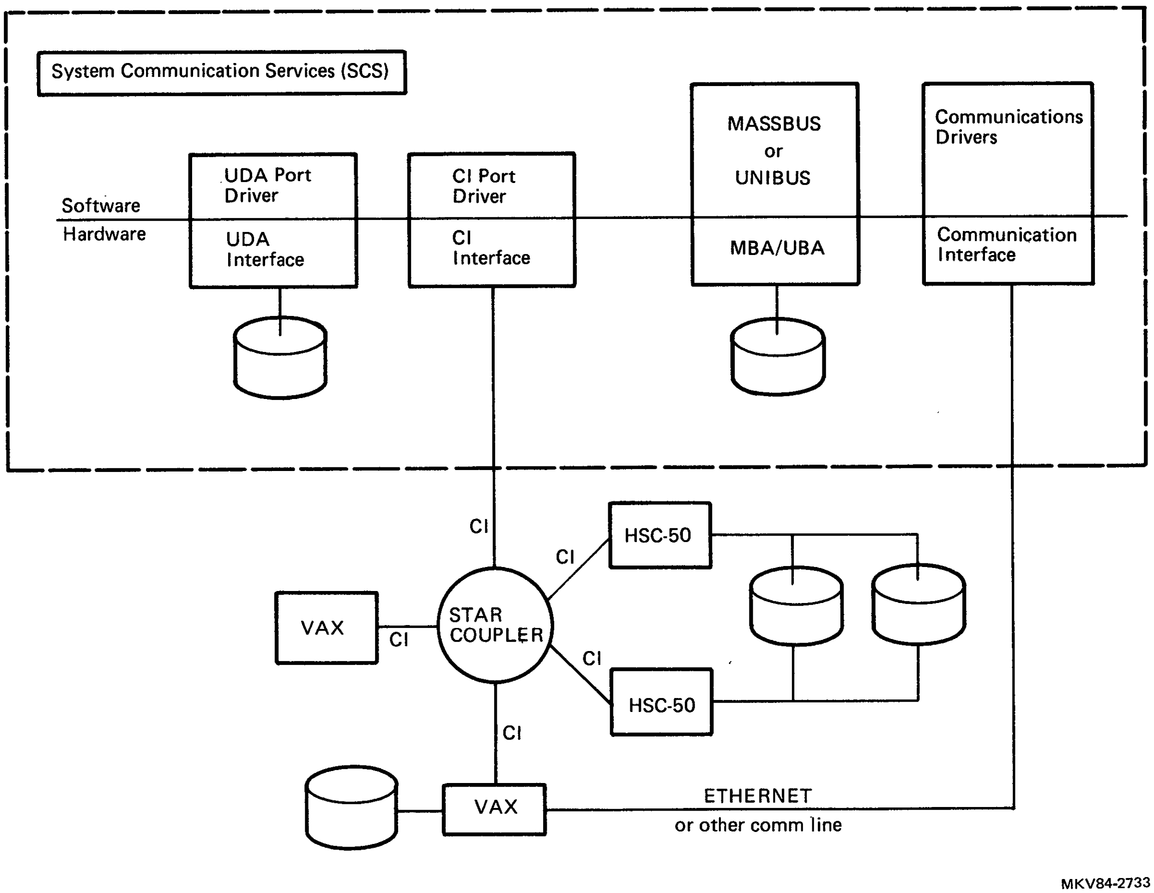


Figure 5 VAXcluster Hardware/Software Block Diagram

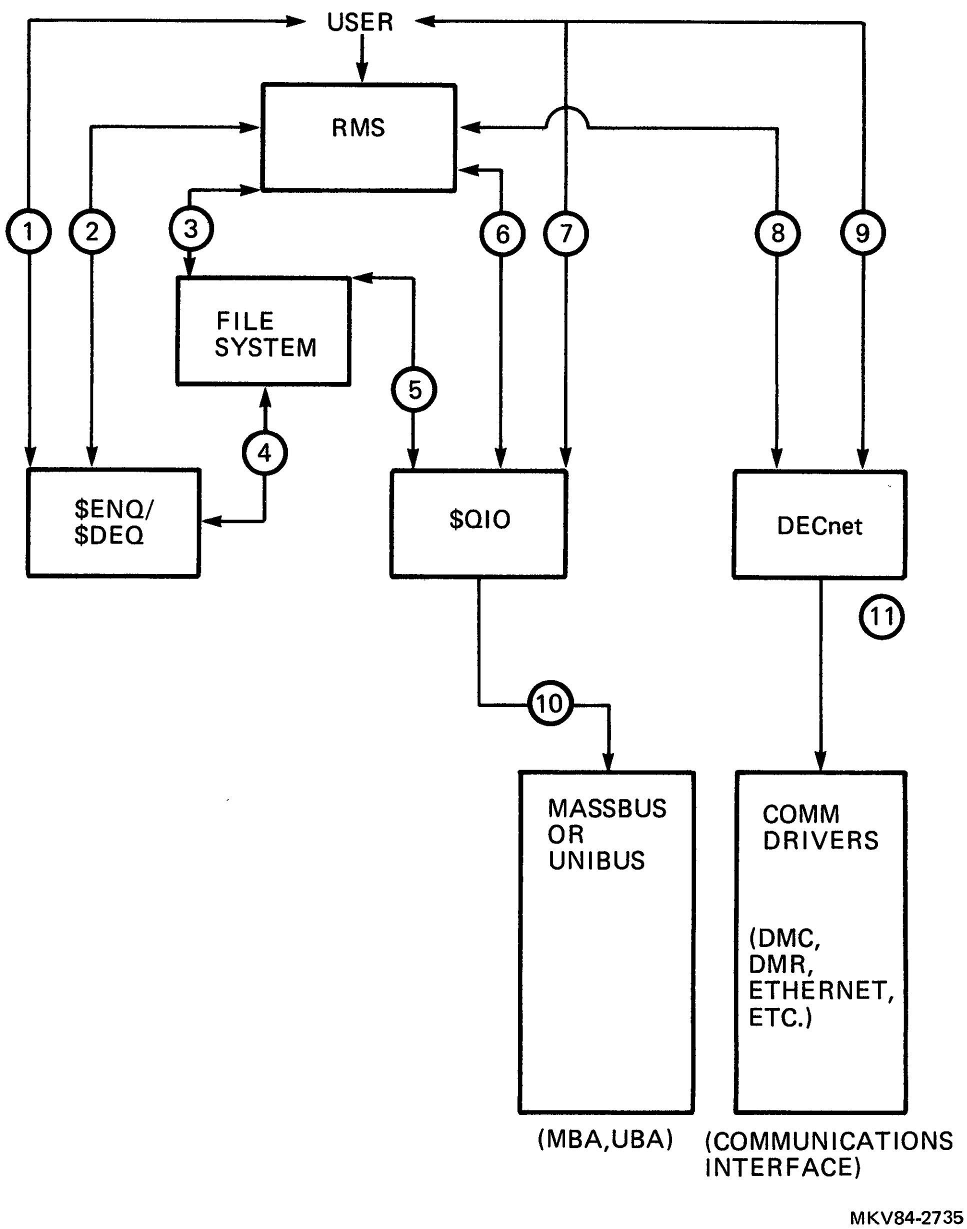


Figure 6 Flow of Standard I/O Operations

Notes on Figure 6

The "normal" connection is from the USER to RMS, letting RMS take care of all the work.

1. USER to ENQ/DEQ Services

2. RMS to ENQ/DEQ Services

3. RMS to File System Access

   - Under Version 3, this is the Disk ACP

   - For Version 4.0, the ACP is now an XQP that resides in P1 space

4. File System to ENQ/DEQ Services

5. File System to $QIO Service

   - Under V3, the ACP sends the IRP directly to the driver

6. RMS to $QIO Service

7. USER to $QIO Service

8. RMS to DECnet (NETDRIVER and NETACP)

9. USER to DECnet (NETDRIVER and NETACP) via $QIO

10. $QIO to Drivers

11. DECnet to Communication Drivers

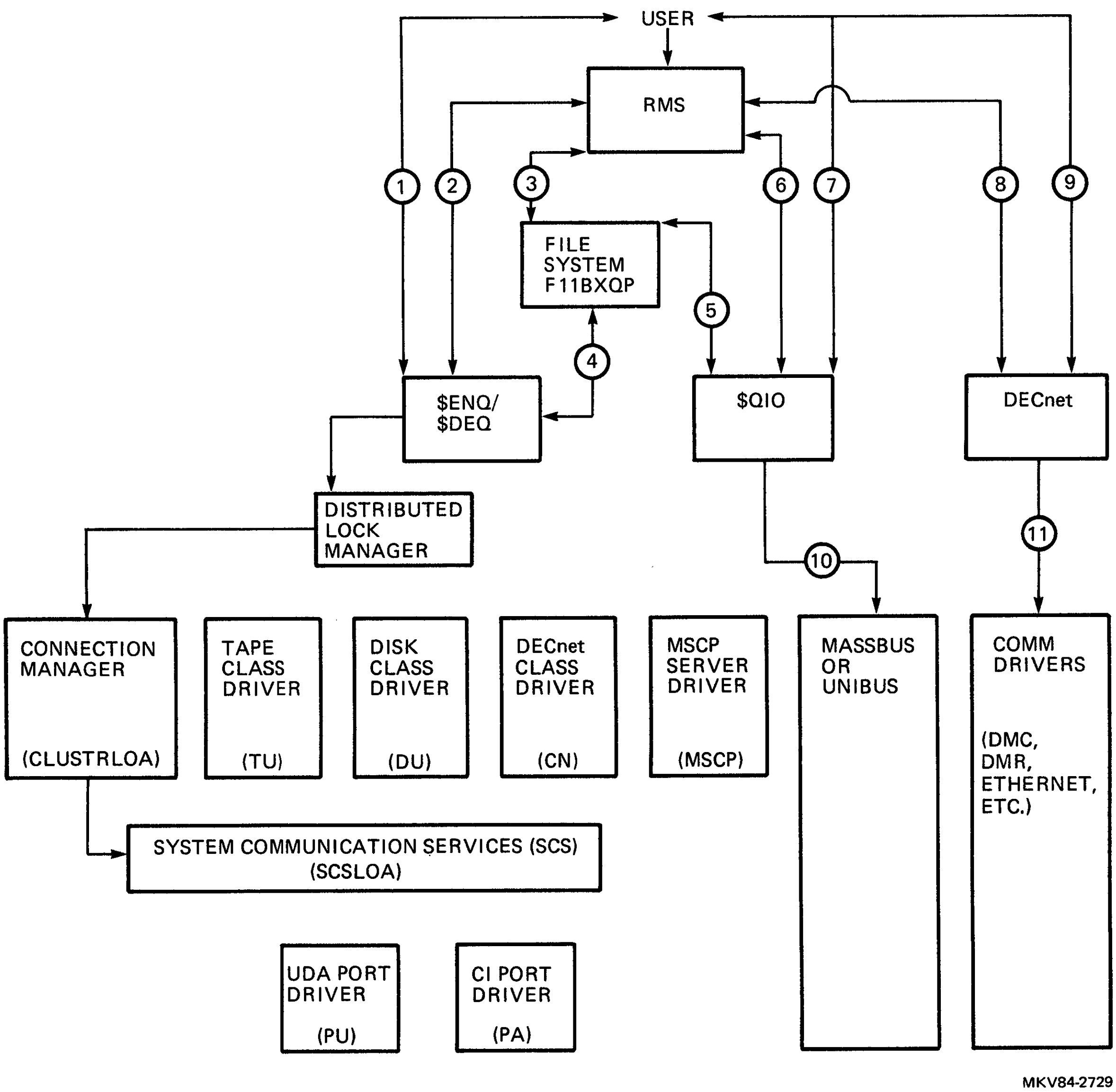


Figure 7 Cluster I/O Available on Version 4.0

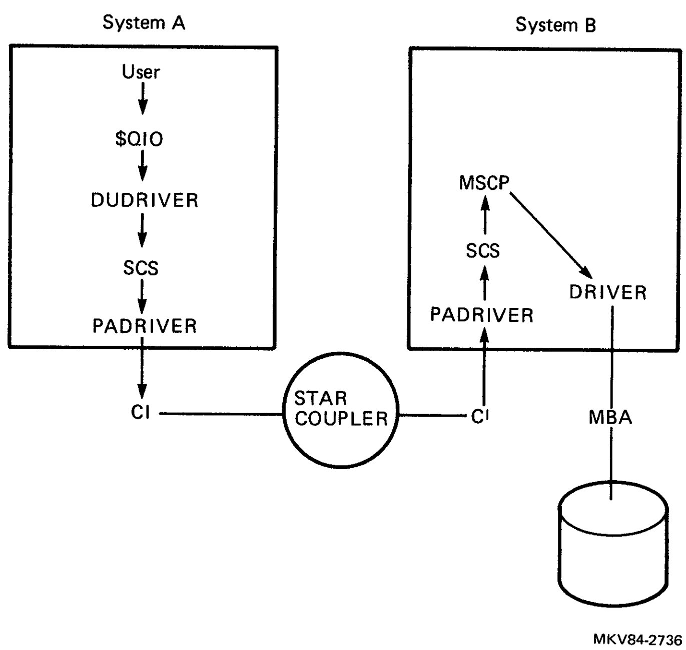


Figure 8 Data Flow for an MSCP Request

MSCP Server

- HSC emulator
- Queues directly to driver
- Pool requirements - buffers and control blocks
- Structure and control flow

JOINING A VAXcluster

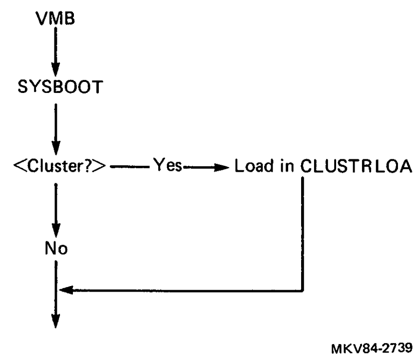


Figure 9 VAXcluster System Start-Up Flow

VMB

- Searches for the start-up files first in [SYSn.SYSEXE] then looks in [SYSn.SYSCOMMOM.SYSEXE]; this allows both nonshared and shared file systems to function
- Identifies all NEXUS adapters (including the CI)
- If CI port found CI microcode loaded

SYSBOOT

- Based on the existence of a CI, SYSBOOT verifies need for, allocates nonpaged pool, and loads SCSLOA.EXE and CLUSTRLOA.EXE.

INIT

- Connects the self-relative vectors from CLUSTRLOA to the system-side vectors and JSBs to CNX$INIT

SYSINIT

- Output message indicating node is waiting to join cluster
- Set cluster initialization flag
- Loops until cluster is formed

LEAVING A VAXcluster

SYS$SYSTEM:SHUTDOWN.COM

- If system is a member of a cluster, ask for shutdown options
  - REMOVE_NODE
  - CLUSTER_SHUTDOWN
  - REBOOT_CHECK
- Quorum disk (if used) is not dismounted
- Define logicals used by OPCCRASH.EXE
  - OPC$UNLOAD
  - OPC$REBOOT
  - OPC$CLUSTER_SHUTDOWN
  - OPC$REMOVE_NODE
- RUN SYS$SYSTEM:OPCCRASH.EXE
  - Flush caches for system disk (mark it for dismount)
  - Set reboot flag according to the logical OPC$REBOOT
  - If member of a VAXcluster

```
If OPC$SHUTDOWN defined,
    SETIPL to IPL$_SYNCH
    JSB    G^CNX$SHUTDOWN
```

  - If OPC$REMOVE_NODE defined,

```
SETIPL to IPL$_SYNCH
Calculate new quorum value
JSB G^CNXCHANGE_QUORUM
When quorum is reset, BUGCHECK
```

```
waiting to form or join VAXcluster
%CNXMAN,  Discovered system MOE
%CNXMAN,  Established connection to system MOE
%CNXMAN,  Sending VAXcluster membership request to system MOE
%CNXMAN,  Now a VAXcluster member -- system LARRY
%PAA0, HSC Error Logging Datagram Received - REMOTE PORT   0


%%%%%%%%%%%  OPCOM   2-SEP-1984 21:01:31.46  %%%%%%%%%%%
Logfile has been initialized by operator _LARRY$LPA0:
Logfile is SYS$SYSROOT:[SYSMGR]OPERATOR.LOG;50
```

Example 2 Booting a VAXcluster System

```
%CNXMAN,  Lost connection to system MOE
%CNXMAN,  Timed-out lost connection to system MOE
%CNXMAN,  Proposing reconfiguration of the VAXcluster
%CNXMAN,  Removed from VAXcluster system MOE
%CNXMAN,  Quorum lost, blocking activity
%CNXMAN,  Completing VAXcluster state transition
%PAA0, HSC Error Logging Datagram Received - REMOTE PORT   0


%PAA0, HSC Error Logging Datagram Received - REMOTE PORT   0


%PAA0, HSC Error Logging Datagram Received - REMOTE PORT   0

%CNXMAN,  Quorum regained, resuming activity
```

Example 3 Leaving a VAXcluster

- Unexpected node crash
- Expected node shutdown using SHUTDOWN.COM

ADDITIONAL CONSIDERATIONS IN A VAXcluster ENVIRONMENT

- Coordination of
  - SYSUAF.DAT
  - NETUAF.DAT
  - VMSMAIL.DAT
  - RIGHTSLIST.DAT
  - JBCSYSQUE.DAT (the queue file)
- Shared Disks ($SET DEVICE/SERVED)
- $MOUNT/CLUSTER disk

SUMMARY

Table 4 Selected VAXcluster SYSGEN Parameters

Function	Parameter
Used to give a unique name to devices that are accessible from more than one node or HSC.	ALLOCLASS
ASCII name of the quorum disk, if one is used. Example, 255DUA0, where $255 is the allocation class and DUA0 is an HSC disk.	DISK_QUORUM
Minimum number of CI nodes (VAXen and HSC) needed to make up a given VAXcluster.	QUORUM
The ID number used by SCS to identify the node. Must be the same as the DECnet node number.	SCSSYSTEMID SCSSYSTEMIDH
The name of the node used by SCS. Must be the same as the DECnet node name.	SCSNODE
Controls the start-up actions of the system. - If 0, do not participate in a cluster. - If 1, should participate in a cluster if hardware exists - If 2, should participate in a cluster	VAXCLUSTER
The number of votes this node has in the VAXcluster.	VOTES

APPENDIX
VAXcluster SYSGEN PARAMETERS

ALLOCLASS	Specifies a numeric value to be assigned as the allocation class for the node.
DISK_QUORUM	The name, in ASCII, of an optional quorum disk. ASCII spaces indicate that no quorum disk is being used.
QDSKVOTES	Specifies the number of votes contributed to the cluster votes total by a quorum disk. The maximum is 127, the minimum is 0, and the default is 1.
QDISKINTERVAL	Specifies the disk quorum polling interval, in seconds. The maximum value is 32767, the minimum value is 1, and the default is 10. Lower values trade increased overhead cost for greater responsiveness. DIGITAL recommends that this parameter be set to the same value on each cluster node.
QUORUM	Specifies an initial setting for the dynamic quorum value. This setting is a numeric value that is an estimate of the correct quorum value to be used and should be greater than half of the total expected votes. By default, the value is 1.
RECNXINTERVAL	Specifies in seconds the interval during which the connection manager attempts to reconnect a broken connection to another VMS system. If a new connection cannot be established during this period, the connection is declared irrevocably broken, and either this system or the other must leave the cluster. This parameter trades faster response to certain types of system failures against the ability to survive transient faults of increasing duration. DIGITAL recommends that this parameter be set to the same value on each cluster node.

VAXcluster — Controls whether the system should join or form a VAXcluster. This parameter accepts the following three values:

- 0 -- Specifies that the system will not participate in a VAXcluster.
- 1 -- Specifies that the system should participate in a VAXcluster if hardware supporting SCS is present (CI, UDA, HSC-50).
- 2 -- Specifies that the system should participate in a VAXcluster.

You should always set this parameter to 2 on systems intended to run in a VAXcluster, 0 on systems that boot from a UDA and are not intended to be part of a VAXcluster, and 1 (the default) otherwise.

VOTES — Specifies the number of votes towards a quorum to be contributed by the node. By default, the value is 1.

PANUMPOLL — Specifies the number of ports to poll at each interval. DIGITAL recommends that this parameter be set to the same value on each cluster node.

PASTIMOUT — Specifies the basic interval at which the CI port driver wakes up to perform time-based bookkeeping operations. It is also the period after which a start handshake datagram is assumed to have timed out. Note that the value obtained by multiplying the values of PASTRETRY and PASTIMOUT must be greater than, or equal to, the value of PAPOLLINTERVAL.

Normally the default value is adequate. DIGITAL recommends that this parameter be set to the same value on each VAXcluster node.

PASTDGBUF — Specifies the number of datagram receive buffers to queue for the CI port driver's configuration poller; that is, the maximum number of start handshakes that can be in progress simultaneously.

Normally the default value is adequate. DIGITAL recommends that this parameter be set to the same value on each cluster node.

PAMAXPORT — Specifies the maximum number of CI ports the CI port driver polls for a broken port-to-port virtual circuit, or a failed remote node.

You can decrease this parameter in order to reduce polling activity if the hardware configuration has fewer than 16 ports. For example, if the configuration has a total of five ports assigned port numbers 0-4, then you should set PAMAXPORT to 4. Note that ports should be assigned contiguously starting at 0.

The default for this parameter is 15 (poll for all possible ports 0 through 15). DIGITAL recommends that this parameter be set to the same value on each cluster node.

PANOPOLL — Disables polling if set to 1 (the default is 0). Disabling polling enables you to boot a system from a private system disk and isolate it from CI activity. You may want to do this following repairs to verify that the system runs properly before introducing it into the hardware cluster. Never set PANOPOLL to 1 while a system is participating in a cluster, if a system is being booted from an HSC, or if it is being booted in order to join a cluster.

PAPOLLINTERVAL — Specifies in seconds the polling interval the Computer Interconnect (CI) port driver uses to poll for a newly booted system, a broken port-to-port virtual circuit, or a failed remote node.

This parameter trades polling overhead against quick response to virtual circuit failures. DIGITAL recommends that you use default value for this parameter.

PAPOLLINTERVAL is a dynamic parameter with a minimum value of 1, a maximum value of 32767, and a default value of 15.

DIGITAL recommends that this parameter be set to the same value on each cluster node.

PAPOOLINTERVAL — Specifies in seconds the interval at which the PA port driver checks for available nonpaged pool after a failure to allocate.

Normally the default value is adequate.

PASANITY	Controls whether the port sanity timer is enabled to permit remote systems to detect a system that has been halted or hung at IPL 7 or above for 99 seconds. This parameter is normally set to 1 and should only be set to 0 when debugging with XDELTA. PASANITY is a dynamic parameter (altered the next time the port is initialized) and has a default value of 1.
PRCPOLINTERVAL	Specifies in seconds the polling interval used to look for SCS applications, such as the Connection Manager and MSCP disks, on other nodes. Each node is polled, at most, once each interval. This parameter trades polling overhead against quick recognition of new systems or servers as they appear. DIGITAL recommends that you set this parameter to 15, which is the default.
SCSBUFFCNT	Specifies the number of computer interconnect (CI) buffer descriptors configured for all CI ports on the system.
SCSCONNCNT	Specifies the total number of SCS connections that are configured for use by all System Applications, including the one used by the directory service listen. Normally, the default value is adequate.
SCSFLOWCUSH	Specifies a lower limit for receive buffers at which point SCS starts to notify the remote SCS of new receive buffers. For each connection, SCS tracks the number of receive buffers available. SCS communicates this number to the SCS at the remote end of the connection. However, SCS does not need to do this for each new receive buffer added. Instead, SCS notifies the remote SCS of new receive buffers if the number of receive buffers falls as low as the SCSFLOWCUSH value. Normally the default value is adequate.

SCSSYSTEMID	Specifies the low-order 32 bits of the 48-bit system identification number. This parameter is not dynamic and must be the same as the DECnet node number.
SCSSYSTEMIDH	The high-order 16 bits of the 48-bit system identification number. This parameter is not dynamic and must be the same as the DECnet node number. Note that once a node has been recognized by another node in the cluster, you cannot change the SCSSYSTEMIDH or SCSNODE parameter without changing both.
SCSNODE	Specifies the SCS system name. This parameter is not dynamic. You should use a name that is the same as the DECnet node name (limited to six characters) since the name must be unique among all systems in the cluster. Note that once a node has been recognized by another node in the cluster, you cannot change the SCSSYSTEMIDH or SCSNODE parameter without changing both.
SCSRESPCNT	Specifies the total number of response descriptor table entries configured for use by all System Applications.

EXERCISES

System Processes

EXERCISES

1. List three functions of the Job Controller.

2. How do print symbionts receive their information from the Job Controller?

3. How much does a VMS print symbiont understand about print queues?

4. What VMS component transfers errors logged in system memory to disk?

System Processes

SOLUTIONS

1. The Job Controller performs these functions:

   - Manages batch queues and batch jobs
   - Symbiont manager
   - Has a part in creation of interactive process initiated by unsolicited terminal input
   - Accounting manager

2. Print Symbionts receive their information from the Job Controller through mailboxes.

3. A VMS print symbiont knows nothing about print queues. A print symbiont is concerned with its current file and nothing else.

4. The ERRFMT process transfers errors logged in system memory to disk.

EXERCISES

VMS provides places where users or layered products can hook into pieces of VMS software. The code for the ERRFMT process is one of these places.

Write a program that obtains a copy of all errors handled by ERRFMT (and ERF), and displays them on the terminal.

To determine how to hook into the ERRFMT process, examine the code for ERRFMT provided in Example 1.

```
;++
; FACILITY:  ERROR LOG FORMAT PROGRAM
;
; ABSTRACT:  THIS PROGRAM EMPTIES THE ERROR LOG BUFFERS AND CREATES
;              A FILE, ERRLOG.SYS, IN A FORMAT ACCEPTABLE TO ERF.
;
; MACROS:
;
; EQUATED SYMBOLS:
;
       $PRDEF                             ; DEFINE PROCESSOR REGISTERS
       $DCDEF                             ; DEFINE DEVICE CLASS TYPES
       $DIBDEF                            ; DEVICE INFORMATION BUFFER
       $DVIDEF                            ; $GETDVI MESSAGE CODES
       $EMBETDEF                          ; ERROR MESSAGE ENTRY TYPES
       $EMBDEF                            ; DEFINE ERROR MESSAGE BF HDR
       $EMBTSDEF                          ; DEFINE TIME STAMP DEFINITION
       $ERFHDDEF                          ; ERROR FORMAT HEADER DEFINIT
       $ERFTSDEF                          ; ERROR FORMAT TIME STAMP DEF
       $ERFVMDEF                          ; ERROR FORMAT VOLUME MOUNT D
       $ERLDEF                            ; SYSTEM ERROR LOGGING DEFINI
       $OPCDEF                            ; OPERATOR MESSAGE DEFINITION
       $PCBDEF                            ; PROCESS CONTROL BLOCK DEFIN
       $SSDEF                             ; DEFINE STATUS CODES

ERM$C_FORMAT = 2                          ; FORMAT NUMBER FOR VAX
ERF$C_LOOP_CNT = 255                      ; TIMES TO WAIT FOR BUFFER
ERF$K_DLTA_STMP = <60*10>                 ; TIME STAMP DELTA IN SECS
ERF$K_CLK_TICK = -<10*1000*1000>          ; CONVERSION TO CLOCK TICKS/S
;
; OWN STORAGE:
```

Example 1 Selected ERRFMT Source Code (Sheet 1 of 11)

EXERCISES

```
        .PSECT  DATA,RD,WRT,NOEXE,PAGE

INBUF: .BLKB    512                     ; INPUT BUFFER
OUTFAB:         $FAB    -                       ; RECORD ACCESS BLOCK
                FAC=<PUT,UPD>,-         ; PUT AND UPDATE FILE ACCESS
                FNA=OUTNAM,-            ; FILE NAME ADDRESS
                FNS=OUTNAMSZ,-          ; LENGTH OF FILE NAME
                NAM=NAMEBLOCK, -        ; ASSOCIATED NAME BLOCK
                RFM=VAR, -
                FOP=CIF, -
                SHR=<GET,UPI>, -
                ORG=SEQ,-               ; SEQUENTIAL ORGANIZATION
                MRS=0                   ; MAX RECORD SIZE UNSPECIFIED

OUTRAB:         $RAB    -                       ; RECORD ACCESS BLOCK
                ROP=<EOF,WBH>, -        ; OPEN TO END OF FILE
                MBC=1, -
                MBF=2, -
                RAC=SEQ, -
                FAB=OUTFAB              ; FILE ACCESS BLOCK ADDR

NAMEBLOCK:                              ; NAME BLOCK ASSOCIATED WITH
        $NAM
OUTFID:          .WORD  0[3]            ; SAVED FILE ID
LASTENTRY:      .BYTE 0                 ; ENTRY TYPE OF LAST RECORD W
SID:            .LONG   0               ; SYSTEM ID #
ERF$W_MBXCHN:   .WORD   0               ; DIAGNOSTIC MAILBOX CHANNEL
ERF$W_MBXSIZ:   .WORD   0               ; DIAGNOSTIC MAILBOX SIZE
ERF$W_MBXUNT:   .WORD   0               ; PREVIOUS DIAG MBX UNIT #

DEVFAO: .ASCID  /_!AC!UW:/              ; $FAO control string to form

;
; MESSAGE SENT TO OPERATOR UPON FAILURE TO WRITE TO ERROR LOG FILE.
;
OPRMSG_DSC:
OPRMSG_LEN:
        .LONG   OPRMSG_END-OPRMSG       ; SIZE OF OPERATOR MESSAGE BF
        .LONG   OPRMSG                  ; ADDRESS OF OPERATOR MESSAGE
ROMSG_DSC:
        .LONG   ROMSG_END-ROMSG
        .LONG   ROMSG
OPRMSG:
        .LONG   OPC$_RQ_RQST!-          ; TYPE OF MESSAGE
                <<OPC$M_NM_CENTRL@8>>   ; OPERATOR TO INFORM
```

Example 1 Selected ERRFMT Source Code (Sheet 2 of 11)

EXERCISES

```
        .LONG    0                        ; NOBODY TO RESPOND TO
        .ASCII   /ERRFMT - ERROR ACCESSING ERROR LOG FILE/<13><10>
OPRMSG_END:

ROMSG:
        .BLKB    256                      ; HOLDS TRANSLATED STATUS ME
ROMSG_END:
ROMSG_LEN:                                ; HOLDS TRANSLATED MESSAGE L
        .LONG    0
;
; MESSAGE SENT TO OPERATOR WHEN WE'VE FAILED TOO MANY TIMES TO WRITE
; TO ERROR LOG FILE.
;
BYEMSG_DSC:                               ; MESSAGE DESCRIPTOR
BYEMSG_LEN:
        .LONG    BYEMSG_END-BYEMSG        ; LENGTH
        .LONG    BYEMSG                   ; ADDRESS

BYEMSG:                                           ; MESSAGE
        .LONG    OPC$_RQ_RQST! -          ; TYPE OF MESSAGE
                 <<OPC$M_NM_CENTRL@8>>    ; OPERATOR TO INFORM
        .LONG    0                        ; NOBODY TO RESPOND TO
        .ASCII   /ERRFMT - DELETING ERRFMT PROCESS/<13><10>
        .ASCII   /ERROR LOG FILE UNWRITABLE/<13><10>
         .ASCII /TO RESTART ERRFMT PROCESS, USE "@SYS$SYSTEM:STARTUP
BYEMSG_END:
;
; MOUNT AND DISMOUNT MESSAGE STRINGS
;
MOUNT_FAO:
        .LONG    MOUNT_END-MOUNT_MSG      ; LENGTH OF CONTROL STRING
        .ADDRESS         MOUNT_MSG        ; ADDRESS OF CONTROL STRING
MOUNT_MSG:
        .LONG    OPC$_RQ_RQST             ; TYPE OF MESSAGE (OPERATOR
        .LONG    0                        ; NOBODY TO REPLY TO
        .ASCII   \Volume "!AD"!ASmounted, on physical device !AS\
MOUNT_END:

MOUNT_DSC:
        .LONG    128                      ; MAX SIZE OF THE MESSAGE
        .ADDRESS         MOUNT_BUF        ; ADDRESS OF THE MESSAGE BUF
MOUNT_BUF:
        .BLKB    128                      ; STORAGE FOR FORMATTED MESS
MOUNT_MNT:
        .ASCID   \ \                      ; FOR VOLUME MOUNTED MESSAGE
```

Example 1 Selected ERRFMT Source Code (Sheet 3 of 11)

EXERCISES

```
MOUNT_DMT:
        .ASCID  \ dis\                  ; FOR VOLUME DISMOUNTED MESS
;
; ERROR COUNTERS
;
ERF$B_ERRCNT:                           ; COUNT ERRORS IN WRITING TO
        .BYTE   0                       ; ERRORLOG FILE
ERF$B_MAXERRCNT:                        ; MAXIMUM # ERRORS BEFORE DE
        .BYTE   20                      ; THIS PROCESS

;
; Data structures needed to get the version number and expanded file
; a newly created SYS$ERRORLOG:ERRSNAP.LOG (Venus-specific).
;
        .ALIGN  PAGE
ERRSNAP_FAB:
        $FAB    -                       ; File Access Block.
                FNM=<SYS$ERRORLOG:ERRSNAP.LOG>, -       ; File name.
                NAM=ERRSNAP_NAM, -      ; Associated NAM block.
                XAB=ERRSNAP_XAB         ; Associated XAB block.

ERRSNAP_XAB:                            ; Declare date/time XAB.
        $XABDAT

ERRSNAP_NAM:
        $NAM    -                       ; Name block.
                RSA=ERRSNAP_RSA, -      ; Resultant string area addr
                RSS=NAM$C_MAXRSS        ; Use maximum length of resu

ERRSNAP_RSA:                            ; Resultant string will be r
        .BLKB   NAM$C_MAXRSS
;
; Data structures used when SPAWNing a sub-process to execute ERRSNA
;
ERRSNAP_COM:                            ; Descriptor for command pro
        .ASCID  /SYS$ERRORLOG:ERRSNAP.COM/
ERRSNAP_LOG1:                           ; Initial DCL command if cop
        .ASCID  /$ FILENAME := SNAP1.DAT/
ERRSNAP_LOG2:                           ; Initial DCL command if cop
        .ASCID  /$ FILENAME := SNAP2.DAT/
ERRSNAP_FLAGS:                          ; Set NOCLISYM and NOWAIT fl
        .LONG   6
ERRSNAP_STATUS:                         ; Store the exit status of t
        .LONG   0                       ; command procedure here.
```

Example 1 Selected ERRFMT Source Code (Sheet 4 of 11)

EXERCISES

```
; Definitions needed to communicate with 11/790 logical console inte
;
CON$C_REQERL = ^X30                     ; Console command to request
                                        ; snapshot file status.
CON$C_INVSNP1 = ^X31                    ; Console command to invalid
CON$C_INVSNP2 = ^X32                    ; Console command to invalid
ERRSNAP_CONCMD:                         ; Store command to be sent t
        .BYTE   0
ERRSNAP_DATA:                           ; Store returned data from l
        .LONG   0                       ; console interface here.
;
; PURE DATA - KEPT IN CODE PSECT FOR LOCALITY
;
        .PSECT  CODE,RD,NOWRT,EXE
;
; ARGUMENT LIST FOR FILE CREATE TIME STAMP ENTRY
;
FILCRE:                 .LONG   1               ; ONE ARGUMENT
                .LONG   EMB$K_NF        ; NEW FILE TYPE MESSAGE

ERF$Q_DELTA:                            ; TIME BETWEEM TIME MARKS

; ***   .LONG   ERF$K_CLK_TICK*ERF$K_DLTA_STMP&^X0FFFFFFFF

                .LONG   ^X09A5F4400     ; LOW 1/2 OF DELTA TIME

; ***   .LONG   ERF$K_CLK_TICK*ERF$K_DLTA_STMP@-32

                .LONG   ^X0FFFFFFFE     ; HIGH 1/2 OF DELTA TIME

ERF$Q_WAIT:     .LONG   -<10*1000*500>  ; # OF 10 MILLISEC INTERVALS
                .LONG   -1              ; TO WAIT FOR BUFFER COMPLET
OUTNAM:         .ASCII  \SYS$ERRORLOG:ERRLOG.SYS\ ; OUTPUT FILE NAME
OUTNAMSZ = . - OUTNAM                   ; LENGTH OF OUTPUT NAME

        .SBTTL  ERRFMT
;++
; FUNCTIONAL DESCRIPTION:
;
;       THIS PROGRAM IS AWAKENED FROM HIBERNATION BY THE ERROR LOGGE
;       WHENEVER AN ERROR LOG BUFFER BECOMES FULL.  THE ERROR FORMAT
;       PROGRAM READS THE FULL BUFFER AND THEN RELEASES IT FOR RE-US
;       THE ERROR LOGGER PROGRAM.  THE DATA JUST READ IS RE-ORGANIZE
;       AND WRITTEN TO A FILE CALLED "ERRLOG.SYS" IN A FORMAT ACCEPT
;       TO SYE.
```

Example 1 Selected ERRFMT Source Code (Sheet 5 of 11)

EXERCISES

```
;       THE ERROR FORMAT PROGRAM ALSO PLACES TIME STAMP ENTRIES INTO
;       ERROR LOG BUFFER.  THESE TIME STAMPS ARE PLACED INTO THE BUF
;       AT REGULAR INTERVALS.  HOWEVER, SEQUENTIAL TIME STAMPS ARE N
;       WRITTEN INTO THE FILE, "ERRLOG.SYS".
;
;       THE FILE, "ERRLOG.SYS", IS UPDATED, OR A NEW VERSION CREATED
;       THE MOST RECENT VERSION IS BEING ACCESSED OR DOES NOT EXIST.
;--
        .PSECT   CODE,RD,NOWRT,EXE
        .ENABL   LSB
        .ENTRY   ERF$START,0                ;
        $CMKRNL_S       W^ERF$INIT          ; INITIALIZE THE ERR FORMATE
        CMPB     #PR$_SID_TYP790, -         ; ARE WE EXECUTING ON A VENU
                 G^EXE$GB_CPUTYPE           ;
        BNEQ     PRCBUF                     ; BRANCH IF NO
        CALLS    #0,W^ERF$ERRSNAP           ; CALL VENUS-SPECIFIC ERROR
PRCBUF: $CMKRNL_S       W^ERF$GETBUF        ; GET THE FULL ERROR LOG BUF
        BLBS     R0,PRCNXT                  ; BR IF MESSAGE(S) TO PROCES
        $CLOSE   FAB=W^OUTFAB               ; CLOSE THE OUTPUT
        $HIBER_S                            ; WAIT FOR SOMETHING TO DO
        BRB      PRCBUF                     ;
;
; PROCESS NEXT MESSAGE - COME HERE WHEN A BUFFER HAS BEEN COPIED FRO
; THE SYSTEM INTO THE LOCAL BUFFER.  IF THE FILE IS NOT OPEN,
; OPEN THE OUTPUT FILE OR CREATE ONE IF MOST RECENT IS BEING ACCESSE
;
PRCNXT: CLRL      R3                        ; R3=0 => OPEN EXISTING FILE
                                            ; R3~=0 => CREATE NEW ERRLOG
PRCNXT1:
        MOVAB    W^INBUF,R8                 ; GET ADDR OF FIRST MSG
        ADDB3    ERL$B_BUSY(R8),ERL$B_MSGCNT(R8),R6 ; GET COUNT OF ME
        BEQL     PRCBUF                     ; BR IF NO MESSAGES TO PROCE
        ADDL     #ERL$C_LENGTH,R8           ; POINT TO START OF MESSAGES
        MOVAB    W^OUTFAB,R2                ; SET ADDRESS OF FAB
        TSTW     FAB$W_IFI(R2)              ; IS THE FILE OPEN?
        BEQL     2$                         ; BRANCH TO OPEN OR CREATE F
        BRW      NXTMSG                     ; FILE ALREADY OPEN; CONTINU
2$:
        CLRL     FAB$L_ALQ(R2)              ; CLEAR ALLOCATION
        TSTL     R3                         ; OPEN OR CREATE ERRLOG.SYS?
        BNEQ     5$                         ; BR TO CREATE NEW FILE
        $OPEN    FAB=(R2)                   ; OPEN MOST RECENT VERSION
        BLBC     R0,4$                      ; OPEN FAILED; GO CREATE A N
```

Example 1 Selected ERRFMT Source Code (Sheet 6 of 11)

EXERCISES

```
; IF THE OPEN WAS SUCCESSFUL, CHECK THAT THIS IS THE SAME ERRLOG.SYS
; ONE WE WROTE TO LAST TIME.  IF NOT, CREATE A NEW VERSION OF ERRLOG
;
        TSTL    W^OUTFID                    ; HAS SYSTEM JUST RE-BOOTED?
        BEQL    10$                         ; YES; DON'T CREATE A NEW ER
        MOVAL   W^NAMEBLOCK,R4              ; GET ADDRESS OF NAME BLOCK
        CMPL    NAM$W_FID(R4),W^OUTFID      ; CHECK FIRST TWO WORDS OF F
        BNEQ    3$                          ; FIDS DIFFER; CREATE A NEW
        CMPW    NAM$W_FID+4(R4),W^OUTFID+4      ; CHECK 3RD WORD OF
        BEQL    10$                         ; FIDS MATCH; GO CONNECT RAB
3$:
        $CLOSE  FAB=(R2)                    ; CLOSE OLD FILE AND CREATE
4$:
        INCL    R3                          ; SIGNAL CREATING NEW FILE
5$:
        $CREATE FAB=(R2)                    ; CREATE NEW VERSION
        BLBS    R0,10$                      ; BRANCH ON SUCCESS
        BRW     WRITE_FAILURE               ; NOTIFY OPERATOR OF CREATE
10$:    MOVAB   W^OUTRAB,R9                 ; SET ADDRESS OF OUTPUT RAB
        CLRW    RAB$W_ISI(R9)               ; PERFORM A FAST DISCONNECT
        $CONNECT RAB=(R9)                   ; CONNECT RAB TO FAB
        BLBS    R0,12$                      ; BRANCH ON SUCCESS
        BRW     WRITE_FAILURE               ; ELSE BRANCH ON FAILURE
12$:
        TSTL    R3                          ; WAS A NEW FILE JUST CREATE
        BEQL    NXTMSG                      ; BR IF NOT NEW FILE
        CLRL    R3                          ; SIGNAL SUCCESSFUL FILE CRE
                                            ; AND INITIALIZATION
        CLRB    W^LASTENTRY                 ; CLEAR SAVED MESSAGE ENTRY
        MOVAL   W^NAMEBLOCK,R4              ; GET ADDRESS OF NAME BLOCK
        MOVL    NAM$W_FID(R4),W^OUTFID      ; SAVE FIRST TWO WORDS OF FI
        MOVW    NAM$W_FID+4(R4),W^OUTFID+4      ; SAVE 3RD WORD OF F
        SUBL    #EMB$K_HD_LENGTH,SP         ; ALLOCATE A BUFFER (ONLY HE
        MOVL    SP,R2                       ; COPY ADDRESS OF BUFFER
        MOVL    R2,RAB$L_RBF(R9)            ; SET BUFFER ADDRESS IN RAB
        MOVW    #EMB$K_HD_LENGTH,RAB$W_RSZ(R9) ; AND SET LENGTH FOR
        MOVL    W^SID,EMB$L_HD_SID(R2)      ; SET SYSTEM IDENT
        MOVW    #EMB$K_NF,EMB$W_HD_ENTRY(R2) ; SET ENTRY TYPE
        MOVQ    EMB$Q_HD_TIME+EMB$K_LENGTH(R8),- ; COPY TIME AND DAT
                EMB$Q_HD_TIME(R2)           ; FIRST ENTRY IN THE ERROR L
        CLRW    EMB$W_HD_ERRSEQ(R2)         ; SET ERROR SEQUENCE NUMBER
        $PUT    RAB=(R9)                    ; WRITE FILE CREATED MARK
        BLBS    R0,15$                      ; BR IF SUCCESSFUL
        BRW     WRITE_FAILURE               ; ELSE BRANCH ON FAILURE
```

Example 1 Selected ERRFMT Source Code (Sheet 7 of 11)

EXERCISES

```
15$:
        ADDL     #ERF$K_TS_LENGTH,SP       ; CLEAR THE STACK
;
; PROCESS A MESSAGE IN THE ERROR BUFFER.
;
;       R6 = NUMBER OF MESSAGES IN THE BUFFER
;       R7 = IS USED TO HOLD THE FORMATTED RECORD
;       R8 = THE START OF THE NEXT MESSAGE IN THE LOCAL BUFFER
;       R9 = ADDRESS OF THE OUTPUT RAB
;
NXTMSG: DECB     R6                        ; IS THERE ANOTHER MSG?
        BGEQ     30$                       ; BRANCH TO FORMAT ANOTHER M
20$:    BRW      PRCBUF                    ; TRY FOR ANOTHER BUFFER
        ASSUME   EMB$W_HD_ENTRY  EQ ERF$W_HD_ENTRY
        ASSUME   EMB$Q_HD_TIME   EQ ERF$Q_HD_TIME
        ASSUME   EMB$W_HD_ERRSEQ EQ ERF$W_HD_ERRSEQ
30$:    ADDL     #EMB$K_LENGTH,R8          ; POINT PAST MESSAGE HEADER
        MOVZWL   EMB$W_SIZE(R8),R1         ; GET SIZE OF MESSAGE TEXT
        SUBL     #EMB$K_LENGTH,R1          ; SUBTRACT SIZE OF MESSAGE H
        MOVW     R1,RAB$W_RSZ(R9)          ; AND SET INTO RAB
        MOVAL    (R8),RAB$L_RBF(R9)        ; AND THE ADDRESS OF THE BUF
        TSTB     EMB$B_VALID(R8)           ; IS RECORD VALID?
        BNEQ     40$                       ; BRANCH ON YES
        BISB     #ERF$M_HD_INVALD,ERF$W_HD_ENTRY(R8) ; FLAG INVALID B
40$:    MOVL     R8,R7                     ; COPY START OF CURRENT RECO
        ADDL     R1,R8                     ; ADVANCE TO NEXT RECORD

        .DSABL   LSB
;
; OUTPUT ERROR MESSAGE.  R1=SIZE.
;
MSGOUT: CMPB     W^LASTENTRY,#EMB$C_TS     ; LAST REC = TIME STA
        BNEQ     10$                       ; BRANCH ON NO
        CMPB     ERF$W_HD_ENTRY(R7),#EMB$C_TS ; THIS REC = TIME STAMP?
        BNEQ     10$                       ; BRANCH ON NO
        MOVB     #RAB$C_RFA,RAB$B_RAC(R9)  ; SET RANDOM FILE ACCESS
        $FIND    RAB=(R9)                  ; FIND LAST RECORD WRITTEN
        MOVB     #RAB$C_SEQ,RAB$B_RAC(R9); SET TO SEQUENTIAL ACCESS
        BLBC     R0,WRITE_FAILURE          ; BR IF ERROR
        $UPDATE  RAB=(R9)                  ; UPDATE LAST RECORD
        BLBC     R0,WRITE_FAILURE          ; BR IF ERROR
        BRW      MBX                       ; BRANCH TO MAILBOX PROCESSI
10$:    MOVB     ERF$W_HD_ENTRY(R7),W^LASTENTRY ; SAVE MSG ENTRY TYPE
        CMPB     ERF$W_HD_ENTRY(R7),#EMB$C_VM   ; VOLUME MOUNTED?
        BEQL     20$                       ; XFER IF SO
```

Example 1 Selected ERRFMT Source Code (Sheet 8 of 11)

EXERCISES

```
        CMPB    ERF$W_HD_ENTRY(R7),#EMB$C_VD    ; OR VOLUME DISMOUNT
        BNEQ    30$                             ; XFER IF NOT
20$:    PUSHL   R7                              ; ELSE SAVE ADDRESS OF THE B
        CALLS   #1,ERF$MOUNT                    ; GO FORM OPERATOR MESSAGE A
30$:    $PUT    RAB=(R9)                        ; OUTPUT MSG
        BLBS    R0,MBX                          ; BR IF SUCCESSFUL $PUT
;
; COME HERE IF AN ACCESS TO THE ERRORLOG FILE FAILED.
;
WRITE_FAILURE:
        $GETMSG_S -                             ; TRANSLATE REASON FOR FAILU
                MSGID=R0, -
                MSGLEN=W^R0MSG_LEN, -
                BUFADR=W^R0MSG_DSC
        MOVL    W^OPRMSG_LEN,R4                 ; SAVE BASIC MESSAGE LENGTH
        ADDL2   W^R0MSG_LEN,W^OPRMSG_LEN;       ; COMBINE OPRMSG WITH STATUS
        $SNDOPR_S -                             ; INFORM OPERATOR OF ERROR I
                MSGBUF=W^OPRMSG_DSC             ; WRITING ERRORLOG FILE
        MOVL    R4,W^OPRMSG_LEN                 ; RESTORE BASIC MESSAGE LENG
        $CLOSE  FAB=W^OUTFAB                    ; CLOSE FILE AS CAN'T WRITE
        ACBB    W^ERF$B_MAXERRCNT,#1, -         ; INC ERROR COUNT AND BRANCH
                W^ERF$B_ERRCNT,10$              ; <= MAX ERROR COUNT.
        $SNDOPR_S -                             ; ELSE NOTIFY OPERATOR THAT
                MSGBUF=W^BYEMSG_DSC             ; PROCESS WILL BE DELETED.
        BRB     MBX                             ; BRANCH TO MAILBOX PROCESSI
10$:                                            ; ERROR COUNT <= MAX ERROR C
        INCL    R3                              ; SIGNAL ACCESS FAILURE
        MOVAB   W^OUTFAB,R2                     ; MUST CREATE NEW FILE
        CLRW    FAB$W_IFI(R2)                   ; CLEAR INDICATOR TO OPEN NE
        $FAB_STORE -                            ; REINITIALIZE FAB
                FAB=(R2), -
                ORG=SEQ, -                      ; SEQUENTIAL ORGANIZATION
                MRS=#0, -                       ; NO MAX ON RECORD SIZE
                FOP=CIF, -
                SHR=<GET,UPI>, -
                RFM=VAR                         ; VARIABLE LENGTH RECORDS
        BRW     PRCNXT1                         ; GO TRY TO OPEN A NEW FILE
MBX:                                            ; MAILBOX MESSAGES
        MOVL    SP,R11                          ; MARK THE STACK
        MOVZWL  W^ERF$W_MBXCHN,R0               ; MBX CHANNEL ALREADY?
        BEQL    30$                             ; BRANCH ON NONE
        CMPW    G^EXE$GQ_ERLMBX,W^ERF$W_MBXUNT ; SAME AS LAST TIME?
        BEQL    50$                             ; YES, GO MAIL THE MSG
```

Example 1 Selected ERRFMT Source Code (Sheet 9 of 11)

EXERCISES

```
        $DASSGN_S       CHAN=R0             ; NO, DEASSIGN OLD CHANNEL
        CLRW    W^ERF$W_MBXCHN              ; CLEAR OLD CHANNEL
30$:
        MOVZWL  G^EXE$GQ_ERLMBX,R0          ; GET NEW MAIL BOX UNIT
        MOVW    R0,W^ERF$W_MBXUNT           ; SET NEW UNIT TO USE
        BEQL    40$                         ; BRANCH IF NONE
        SUBL    #32-4,SP                    ; ALLOCATE BUFFER IN THE STA
        MOVL    SP,R2                       ; MARK START OF MAIL BOX UNI
        PUSHL   #^A/_MBA/                   ; SET PROTOTYPE NAME
        PUSHL   SP                          ; SET START OF BUFFER
        BSBW    100$                        ; SET UNIT OF MAILBOX
        SUBL3   (SP),R2,-(SP)               ; FIND LENGTH OF NAME
        MOVL    SP,R2                       ; SAVE POINTER TO NAME
        $ASSIGN_S       DEVNAM=(R2),-       ; ASSIGN A CHANNEL TO
                        CHAN=W^ERF$W_MBXCHN; THE DIAGNOSTIC MAILBOX
        BLBS    R0,45$                      ; BRANCH ON SUCCESS
40$:    BRW     65$                         ; SKIP THE QIO IF FAILED
45$:
        MOVL    #32,(SP)                    ; RESET LENGTH OF BUFFER
        $GETCHN_S       CHAN=W^ERF$W_MBXCHN,-; GET SIZE OF MAILBOX
                        PRIBUF=(R2)         ; I.E., THE MAXIMUM MSG SIZE
        MOVL    4(R2),R2                    ; GET ADDRESS OF DEV CHAR BU
        MOVW    DIB$W_DEVBUFSIZ(R2),W^ERF$W_MBXSIZ ; GET MAILBOX SIZ
50$:    MOVZWL  RAB$W_RSZ(R9),R0            ; GET SIZE OF MESSAGE
        CMPW    R0,W^ERF$W_MBXSIZ           ; MSG TOO LARGE?
        BLEQU   55$                         ; BRANCH ON OK
        MOVW    W^ERF$W_MBXSIZ,R0           ; TRUNCATE MSG
55$:    $QIO_S  CHAN=W^ERF$W_MBXCHN,-       ; CHANNEL FOR DIAG MBX
                FUNC=#<IO$_WRITEVBLK!IO$M_NOW>,- ; DONT WAIT FOR SUC
                P1=(R7),-                   ; ADDR OF ERROR MSG
                P2=R0                       ; SIZE OF MSG
        CMPB    W^ERF$B_ERRCNT, -           ; HAVE WE EXCEEDED THE ERROR
                W^ERF$B_MAXERRCNT           ; THRESHHOLD?
        BLEQ    65$                         ; BRANCH IF NO
        MOVZWL  W^BYEMSG_LEN,R0             ; GET LENGTH OF GOODBYE MESS
        CMPW    R0,ERF$W_MBXSIZ             ; MESSAGE TOO LARGE?
        BLEQU   60$                         ; BRANCH ON OK
        MOVW    W^ERF$W_MBXSIZ,R0           ; TRUNCATE MESSAGE
60$:
        $QIO_S -                            ; NOTIFY MAILBOX THAT PROCES
                CHAN=W^ERF$W_MBXCHN, -      ; BEING DELETED.
                FUNC=#<IO$_WRITEVBLK!IO$M_NOW>, -
                P1=W^BYEMSG, -
                P2=R0
```

Example 1 Selected ERRFMT Source Code (Sheet 10 of 11)

```
65$:    MOVL    R11,SP                  ; RESET THE STACK POINTER
        CMPB    W^ERF$B_ERRCNT, -       ; HAVE WE EXCEEDED THE ERROR
                W^ERF$B_MAXERRCNT       ; THRESHHOLD?
        BGTR    70$                     ; BRANCH IF YES
        BRW     NXTMSG                  ; ELSE GO PROCESS NEXT MESSA
;
; IF ERRCNT > MAXERRCNT, DELETE THIS PROCESS TO PREVENT INFINITE LOO
; THE ERRFMT PROCESS CAN BE RESTARTED VIA AN OPERATOR COMMAND FILE.
;
70$:    $DELPRC_S                       ; DELETE THIS PROCESS

;
; LOCAL SUBROUTINE TO CONVERT BINARY TO ASCII AND STORE RESULT
; IN BUFFER POINTED TO BY R2
;

100$:   CLRL    R1                      ; ZERO HI 1/2 OF QUADWORD
110$:   EDIV    #10,R0,R0,-(SP)         ; GET NEXT DIGIT
        ADDL    #^A/0/,(SP)             ; FIND THE DIGIT IN ASCII
        TSTL    R0                      ; ANY THING LEFT
        BEQL    120$                    ; BR IF NO MORE TO CONVERT
        BSBB    110$                    ; GET NEXT DIGIT
120$:   CVTLB   (SP)+,(R2)+             ; STORE A BYTE
        RSB                             ;
```

Example 1 Selected ERRFMT Source Code (Sheet 11 of 11)

SOLUTIONS

To obtain a copy of the errors handled by ERRFMT, create a mailbox and place its unit number in EXE$GQ_ERLMBX. Then continuously read the mailbox and output the information from ERRFMT (see Example 2).

```
;                                                       SYSPLAB1.MAR
        .TITLE  GETERROR
        $DVIDEF

BUFSIZ=512
;
;       This program obtains a copy of the errors handled by
;       ERRFMT and displays them on the terminal.
;
        .PSECT  NONSHARED_DATA  PIC, NOEXE, LONG
STAT_BLOCK1:     .BLKW  1               ;Status Block
LEN1:            .BLKW   1
INFO1:           .BLKL   1
BUF_1:          .BLKB   BUFSIZ          ;Error record buffer
TTCHAN:                 .BLKW    1              ;Terminal channel
TTDEV:          .ASCID  \SYS$COMMAND\   ;Terminal Device
GET_LIST:
                .WORD   4               ;Item list for Unit No.
                .WORD   DVI$_UNIT
                .LONG   MBX_UNIT
                .LONG   0
                .LONG   0
MBX_UNIT:
                .BLKL   1               ;MBX Unit Number
        .PSECT  NONSHARED_DATA  PIC, NOEXE, LONG
CHANNEL:        .BLKW   1
MAILBOX_NAME:   .ASCID  /ERROR/

        .PSECT  CODE            PIC, SHR, NOWRT, LONG
        .ENTRY  BEGIN ^M<>
;
;       Open a channel to the terminal
        $ASSIGN_S       CHAN=TTCHAN,DEVNAM=TTDEV
;
;       Create mailbox
        $CREMBX_S  CHAN=CHANNEL, LOGNAM=MAILBOX_NAME,-
                     MAXMSG = #512, BUFQUO = #4096
        BLBC R0,ERR1
```

Example 2 Solution to Lab Exercise (Sheet 1 of 2)

SOLUTIONS

```
;
;       Get the unit number for the mailbox
        $GETDVIW_S      CHAN=CHANNEL, IOSB=STAT_BLOCK1,-
                         ITMLST=GET_LIST
        BLBC R0,ERR1
        BRW     MORE
ERR1:
        BRW     ERR
MORE:
        $CMKRNL_S       GETINFO         ;Record MBX Unit Number
        BLBC    R0,ERR1
;
;       Read message from the mailbox
        $QIO_S  EFN=#1, CHAN=CHANNEL,-
                FUNC=#IO$_READVBLK,IOSB=STAT_BLOCK1,-
                 P1 =BUF_1, P2 = #256
        BLBC R0, ERR
;
;       Wait for information
        $SYNCH_S        EFN=#1,IOSB=STAT_BLOCK1
        BLBC R0, ERR
;
;
;       Output info to terminal
        $QIO_S  EFN=#1, CHAN=TTCHAN,-
                FUNC=#IO$_WRITEVBLK,IOSB=STAT_BLOCK1,-
                 P1 =BUF_1, P2 = #256,P4 = #32
;
        BLBC R0, ERR
40$:    BRW MORE
        MOVL    #SS$_NORMAL, R0
ERR:    RET
        .ENTRY GETINFO,0
30$:
        MOVW    MBX_UNIT,G^EXE$GQ_ERLMBX ; SET MBX UNIT NUMBER

40$:    RET
        .END BEGIN
```

Example 2 Solution to Lab Exercise (Sheet 2 of 2)

Forming, Activating, and Terminating Images

EXERCISES

1. Explain how each of the following INSTALL options affects the start-up time in using an image.

   a. INSTALL

   b. INSTALL/OPEN

   c. INSTALL/OPEN/HEADER

2. Using the linker map in Example 1 below, answer the following questions about the executable image named CALC.

   a. How many image sections are in this image (including sections for the user stack and any shareable images)?

   b. What is the base virtual address of the PSECT named MAIN_CODE?

   c. In which module is the symbol LIB$GET_INPUT defined?

   d. In which module is the symbol SUBTRACT defined?

EXERCISES

e. How many pages of P0 virtual address space, excluding the pages for the RTL, will this image use?

f. One of the image sections starts at virtual address 400 (hex). List the PSECTs that contribute to this image section.

```
                                                    25-OCT-1984 14:02        VAX-11 Linker V04-00            Page    1

                                         +-------------------------+
                                         ! Object Module Synopsis !
                                         +-------------------------+

Module Name      Ident              Bytes     File                              Creation Date       Creator
-----------      -----              -----     ------                            -------------       -------
CALCULATOR       0                   1048 [HUNT.OSI.MODS.IMAGE]CALC.OBJ;2      25-OCT-1984 12:49  VAX/VMS Macro V04-00
SYS$P1_VECTOR    V04-000                0 SYS$SYSROOT:[SYSLIB]STARLET.OLB;2    16-SEP-1984 00:40  VAX/VMS Macro V04-00
LIBRTL           V04-000                0 SYS$SYSROOT:[SYSLIB]LIBRTL.EXE;1     16-SEP-1984 04:00  VAX-11 Linker V04-00


DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.EXE;3          25-OCT-1984 14:02        VAX-11 Linker V04-00            Page    2

                                         +-------------------------+
                                         ! Image Section Synopsis !
                                         +-------------------------+

    Cluster      Type Pages   Base Addr  Disk VBN PFC Protection and Paging     Global Sec. Name   Match      Majorid   Minorid
    -------      ---- -----   ---------  -------- --- --------------------     ----------------   -----      -------   -------

DEFAULT_CLUSTER    0     1    00000200          2   0 READ WRITE   COPY ON REF
                   0     2    00000400          3   0 READ ONLY
                   0     1    00000800          5   0 READ WRITE   FIXUP VECTORS
                 253    20    7FFFD800          0   0 READ WRITE DEMAND ZERO

LIBRTL             3   111    00000000-R        0   0 READ ONLY                 LIBRTL_001        LESS/EQUAL        1        11
                   4     1    0000DE00-R        0   0 READ WRITE DEMAND ZERO    LIBRTL_002        LESS/EQUAL        1        11


          Key for special characters above:
                 +-------------------+
                 ! R  - Relocatable !
                 ! P  - Protected    !
                 +-------------------+

DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.EXE;3          25-OCT-1984 14:02        VAX-11 Linker V04-00            Page    3

                                        +--------------------------+
                                        ! Program Section Synopsis !
                                        +--------------------------+

Psect Name       Module Name        Base     End           Length             Align              Attributes
----------       -----------        ----     ---           ------             -----              ----------

MAC__PSECT                        00000200 000002FE 000000FF (        255.) LONG 2 NOPIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,  WRT,NOVEC
                 CALCULATOR       00000200 000002FE 000000FF (        255.) LONG 2
MAIN_DATA                         000002FF 0000039C 0000009E (        158.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                 CALCULATOR       000002FF 0000039C 0000009E (        158.) BYTE 0
ADD_CODE                          00000400 00000409 0000000A (         10.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 CALCULATOR       00000400 00000409 0000000A (         10.) BYTE 0
DIV_CODE                          0000040A 00000413 0000000A (         10.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 CALCULATOR       0000040A 00000413 0000000A (         10.) BYTE 0
MAIN_CODE                         00000414 00000666 00000253 (        595.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 CALCULATOR       00000414 00000666 00000253 (        595.) BYTE 0
MULT_CODE                         00000667 00000670 0000000A (         10.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 CALCULATOR       00000667 00000670 0000000A (         10.) BYTE 0
SUB_CODE                          00000671 0000067A 0000000A (         10.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                 CALCULATOR       00000671 0000067A 0000000A (         10.) BYTE 0
```

Example 1 Full Linker Map of CALC.EXE
(Sheet 1 of 3)

```
DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.EXE;3                25-OCT-1984 14:02         VAX-11 Linker V04-00            Page    4

                                        +--------------------------+
                                        ! Symbol Cross Reference !
                                        +--------------------------+

Symbol          Value            Defined By          Referenced By ...
------          -----            ----------          -----------------
ADD             00000400-R        CALCULATOR
BEGIN           00000414-R        CALCULATOR
DIVIDE          0000040A-R        CALCULATOR
LIB$GET_INPUT   00000854-RX       LIBRTL               CALCULATOR
LIB$PUT_OUTPUT  00000858-RX       LIBRTL               CALCULATOR
MULTIPLY        00000667-R        CALCULATOR
OTS$CVT_L_TI    0000084C-RX       LIBRTL               CALCULATOR
OTS$CVT_TI_L    00000850-RX       LIBRTL               CALCULATOR
STR$CONCAT      00000848-RX       LIBRTL               CALCULATOR
SUBTRACT        00000671-R        CALCULATOR
SYS$IMGSTA      7FFEDF68          SYS$P1_VECTOR


DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.EXE;3                25-OCT-1984 14:02         VAX-11 Linker V04-00            Page    5

                                           +-------------------+
                                           ! Symbols By Value !
                                           +-------------------+

Value                                  Symbols...
-----                                  ----------
00000400     R-ADD
0000040A     R-DIVIDE
00000414     R-BEGIN
00000667     R-MULTIPLY
00000671     R-SUBTRACT
00000848    RX-STR$CONCAT
0000084C    RX-OTS$CVT_L_TI
00000850    RX-OTS$CVT_TI_L
00000854    RX-LIB$GET_INPUT
00000858    RX-LIB$PUT_OUTPUT
7FFEDF68       SYS$IMGSTA


          Key for special characters above:
                +-------------------+
                ! *  - Undefined    !
                ! U  - Universal    !
                ! R  - Relocatable  !
                ! X  - External     !
                +-------------------+
```

Example 1 Full Linker Map of CALC.EXE
(Sheet 2 of 3)

```
DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.EXE;3                25-OCT-1984 14:02      VAX-11 Linker V04-00           Page    6

                                                     +-----------------+
                                                     ! Image Synopsis !
                                                     +-----------------+

Virtual memory allocated:                             00000200 000009FF 00000800 (2048. bytes, 4. pages)
Stack size:                                                 20. pages
Image header virtual block limits:                           1.          1. (     1. block)
Image binary virtual block limits:                           2.          5. (     4. blocks)
Image name and identification:                        CALC 0
Number of files:                                             4.
Number of modules:                                           3.
Number of program sections:                                 11.
Number of global symbols:                                  251.
Number of cross references:                                 16.
Number of image sections:                                    6.
User transfer address:                                00000414
Debugger transfer address:                            7FFEDF68
Number of code references to shareable images:               5.
Image type:                                           EXECUTABLE.
Map format:                                           FULL WITH CROSS REFERENCE in file DEMON$DUA3:[HUNT.OSI.MODS.IMAGE]CALC.MAP;3
Estimated map length:                                 59. blocks
                                                   +---------------------+
                                                   ! Link Run Statistics !
                                                   +---------------------+

Performance Indicators                                Page Faults   CPU Time       Elapsed Time
----------------------                                -----------   --------       ------------
    Command processing:                                        96   00:00:00.14    00:00:02.66
    Pass 1:                                                   182   00:00:00.77    00:00:03.89
    Allocation/Relocation:                                     32   00:00:00.16    00:00:00.66
    Pass 2:                                                   108   00:00:00.36    00:00:01.67
    Map data after object module synopsis:                     15   00:00:00.22    00:00:00.27
    Symbol table output:                                        5   00:00:00.03    00:00:00.15
Total run values:                                             438   00:00:01.68    00:00:09.30

Using a working set limited to 300 pages and 49 pages of data storage (excluding image)

Total number object records read (both passes):   136
    of which 19 were in libraries and 2 were DEBUG data records containing 266 bytes
235 bytes of DEBUG data were written,starting at VBN 6 with 1 blocks allocated

Number of modules extracted explicitly            = 0
    with 1 extracted to resolve undefined symbols

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

LINK/MAP/FULL/CROSS CALC
```

Example 1 Full Linker Map of CALC.EXE
(Sheet 3 of 3)

Forming, Activating, and Terminating Images

EXERCISES

1.

a. INSTALL allows a file to be opened by file ID and sequence number. This improves the speed of the OPEN operation since directory lookup I/O operations need not be performed.

b. INSTALL/OPEN makes the file permanently opened. There is no wait for the OPEN operation, since a channel to the file has been created, and I/O can be issued immediately.

c. INSTALL/OPEN/HEADER makes the file open and the file header permanently resident. Not only is there no OPEN processing, but one less disk read operation is required during the image activation.

2.

a. This image has 6 image sections; 4 in the default cluster and 2 in the cluster for LIBRTL.

b. The base virtual address of the PSECT named MAIN_CODE is 414 (hex).

c. The symbol LIB$GET_INPUT is defined in module LIBRTL.

d. The symbol SUBTRACT is defined in module CALCULATOR.

e. This image will use 4 pages of P0 virtual address space. (The pages for the user stack are not included in this count because they are P1 pages.)

f. The PSECTs that contribute to the image section with base address 400 are:

- ADD_CODE
- DIV_CODE
- MAIN_CODE
- MULT_CODE
- SUB_CODE

Forming, Activating, and Terminating Images

EXERCISES

1. The DCL ANALYZE/IMAGE command formats and displays the information in an image header. Analyze the image SYS$SYSTEM:MONITOR.EXE and answer the following questions.

   a. How large is the header for this image (in blocks)?

   b. There may be 1 to 3 addresses in the transfer address array. How many transfer addresses are there for this image, and why?

   c. The third image section descriptor (ISD) is only 12 bytes long. Explain the purpose of this type of ISD.

   d. How many pages will be mapped for the user stack when this image is activated? (Remember: One of the image sections describes the user stack.)

   e. Some of the ISDs are between 26 and 64 bytes long. Explain the purpose of this type of ISD.

EXERCISES

2. The known file database describes the installed files on a system. Use the System Dump Analyzer to obtain the following information about the known file database on the system recorded in the dump file OSI$LABS:CRASH1.DMP.

   It will be helpful to read the file OSI$LABS:GLOBALS.STB into your SDA session.

   You may also want to obtain copies of the .PIC (picture) files for the known file data structures (KFPB, KFD, and KFE).

   a. Because the DCL SHOW command is issued frequently, the image that implements this command is installed /OPEN/HEADER/SHARE to improve performance. In addition, the image is installed with privileges.

      To find the Known File Entry (KFE) describing the SHOW image, first calculate the index into the KFE hash table for SHOW.

      Consult your instructor for the hash algorithm, and calculate the hash index.

   b. Locate the KFE hash table.

      (HINTS: There is a pointer to the KFE hash table in the KFPB, and the KFPB can be located with a global system symbol.)

EXERCISES

c. Locate the first KFE in the hash chain containing the KFE for the SHOW image. (The hash index from part (a) is the number of longwords you should offset into the KFE hash table.)

   Search the singly linked hash chain until you find the KFE for the SHOW image.

   (NOTE: When you format a KFE, the last field, containing the ASCII file name string, is not shown. To examine that field, examine the bytes after the last field formatted -- KFE$B_FILNAMLEN.)

d. Locate the mask of privileges with which SHOW is installed, and verify that it is non-zero.

e. Give the device, directory, and file type for the SHOW image.

   (HINT: This information is stored in another data structure for the image, called the KFD. Locate the KFD using the address at offset KFE$L_KFD in the KFE.)

f. Some images are installed from different directories than the SHOW image. List at least two of these different device/directory/file-type combinations.

Forming, Activating, and Terminating Images

SOLUTIONS

1. Analyze the monitor image with the command:

   ```
   $ ANALYZE/IMAGE  SYS$SYSTEM:MONITOR
   ```

   a. The image header is 1 block long.

   b. There is only one transfer address for this image. It is the entry point of the image.

      The other two possible entries in the transfer address array are SYS$IMGSTA and LIB$INITIALIZE. Image start-up is not required for this image, therefore, the address of SYS$IMGSTA does not appear in the transfer address array.

      LIB$INITIALIZE is not referenced by MONITOR, therefore, there is only one transfer address for this image.

   c. The third image section descriptor (ISD) describes a demand-zero section. Note that the ISD flag ISD$V_DZRO is set.

      Demand-zero (DZRO) sections of a program consist of uninitialized pages that do not reside in the disk file. Rather, they are allocated from physical memory as needed.

      Because the DZRO pages do not take up space in the disk file, there is no starting virtual block number (VBN) for a DZRO section. Therefore, a DZRO ISD is 12 bytes long, as compared to a 16-byte ISD for a process private section.

   d. Twenty (20) pages will be mapped for the user stack when this image is activated.

      (You can find the ISD that describes the user stack by locating the ISD whose section type is ISD$K_USRSTACK.)

   e. An ISD that is 26-64 bytes long describes a global image section. A global ISD contains the IDENT and name of the global section.

Forming, Activating, and Terminating Images

SOLUTIONS

2. Enter SDA with the command ANAL/CRASH OSI$LABS:CRASH1.DMP.

   a. Consult your instructor for the hash index for the SHOW image.

   b. The following command displays the address of the KFE hash table:

      ```
      SDA> EXAMINE @EXE$GL_KNOWN_FILES + KFPB$L_KFEHSHTAB
      ```

      The symbol EXE$GL_KNOWN_FILES locates the Known File Pointer Block (KFPB). The address of the KFE hash table is stored at offset KFPB$L_KFEHSHTAB in the KFPB.

   c. First locate the hash chain containing the KFE for the SHOW image.

      Take the address of the KFE hash table from part (b), and offset n longwords, where n is the hash index from part (a). That entry in the hash table contains the address of the first KFE in the hash chain. Format the first KFE with a command such as:

      ```
      SDA> FORMAT  @(tableaddr_from_b + index_from_a * 4)
      ```

      Examine the ASCII string at offset KFE$T_FILNAM in the KFE. If the value of the string is "SHOW", you have located the KFE for the SHOW image.

      If the ASCII string is something other than SHOW, you must search the hash chain. The value at offset KFE$L_HSHLNK is the address of the next KFE in the chain.

   d. The mask of privileges with which SHOW is installed is located at offset KFE$Q_PROCPRIV in the KFE.

   e. SHOW is normally installed from SYS$SYSROOT:[SYSEXE] with the file type .EXE.

      This information is stored at the end of the KFD for SHOW. Locate the KFD using the value at offset KFE$L_KFD in the KFE.

   f. Locate the other device/directory/file-type combinations by searching the list of KFDs. Two of the combinations will be:

      ```
      SYS$SYSROOT:[SYSLIB].EXE
      SYS$SYSROOT:[SYSMSG].EXE
      ```

Paging

EXERCISES

1. What is the page replacement algorithm used by the pager? Explain how this algorithm combines with the page cache (free and modified page lists) to effectively implement a least-recently-used algorithm.

2. What limitation is imposed on the size of a page file by memory management data structures? Is this a restrictive limitation at the present time?

EXERCISES

3. The VAX/VMS operating system uses page table entries of invalid pages to locate those pages in secondary storage. In view of the sequence of steps followed in address translation, what must the contents of PTE<30:27> be? Why?

4. A common programming error involves inadvertently transferring control to location 0. This problem is neatly caught in VAX native images using the memory management protection mechanism. What is the mechanism used? How can the programmer distinguish this exception from other problems?

EXERCISES

5. In translating a process virtual address, two address translations are potentially involved, and thus, two distinct translation-not-valid faults can occur.

   a. What are the translations?

   b. Why are two translations not always required?

   c. How can one distinguish the two faults?

   d. What is the difference between the state of the stack in the two cases?

6. Explain how the VAX hardware uses the modify bit in the page table entry.

7. State one instance when the VAX/VMS operating system must invalidate a single entry in the translation buffer.

Paging

EXERCISES

Questions 8 through 12 represent a sequence of operations involving the interactions among three user processes and VAX/VMS. The processes have the following initial characteristics:

Process Name	Software Priority	Scheduling State
LOW	4	CUR
MEDIUM	10	LEF
HIGH	15	LEF

8. Process LOW causes a page fault in referencing a page in VAX-11 RMS (a mapped system section in the system region). The corresponding page table entry (PTE) points to the image file (SYS$SYSTEM:RMS.EXE).

   a. What is the action of the pager?

   b. Into what scheduling state is process LOW placed?

   c. Into what page state is the physical page (PFN database entry) placed?

9. While the paging operation is in progress, Process HIGH becomes computable and also makes a reference to the same page in RMS as Process LOW referenced.

   a. Into what scheduling state is process HIGH placed?

   b. What page state is the physical page (PFN database entry) in now?

EXERCISES

10. While the paging operation continues, process MEDIUM also becomes computable and also refers to the same RMS page as processes HIGH and LOW.

    a. Into what scheduling state is process MEDIUM placed?

    b. What page state is the physical page (PFN database entry) in now?

11. The paging read operation completes. Further processing is performed at IPL 4 by the I/O post processing routine.

    a. Into what scheduling state is process LOW placed?

    b. Into what scheduling state is process MEDIUM placed?

    c. Into what scheduling state is process HIGH placed?

    d. Into what page state is the physical page (PFN database entry) placed?

EXERCISES

12. IOPOST completes its processing and dismisses the IPL 4 interrupt. A scheduling interrupt (IPL 3) occurs as a result of the IOPOST operations.

    a. Which of the three processes will be scheduled first?

    b. Why is this process selected for execution?

13. Several components and utilities of VAX/VMS are required to cooperate in the implementation of shared sections.

    a. How does this feature contribute to reducing the consumption of disk storage and physical memory?

    b. A shareable image requires the use of the global page table and the global section table to resolve page faults within the image. What does this fact imply about the speed of an individual page fault resolution within a global section? What is the implication of page fault resolution considering all of the processes on the system? Why?

EXERCISES

14. To answer the following question, you will need access to a set of VMS Version 4.x microfiche. See your instructor for the microfiche and a microfiche reader.

    a. The code for the pager is part of the SYS facility. Locate the module on the fiche that contains the pager code, and record the name of the module below.

    b. If a process incurs a page fault when its working set is full, the pager must remove a page from the working set to make room for the new page.

       Locate the routine within the pager that is responsible for freeing a working set list entry.

    c. If a faulted page is not resident, the pager queues a read request for the page. Locate the section of code in the pager that queues a page read request.

    d. In the routine mentioned in part (c), the pager

       - Queues an I/O request
       - Puts the process on the PFW state queue
       - Issues a SVPCTX instruction
       - Branches to SCH$WAITM

       The module that defines SCH$WAITM is part of the SYS facility.

       Using the symbols cross-reference section of SYS.MAP, determine the name of the module that defines the SCH$WAITM routine.

EXERCISES

e. Locate the SCH$WAITM routine in the microfiche. After doing some bookkeeping, the routine branches to SCH$SCHED.

   What is the name of the module that defines SCH$SCHED?

f. The pager is invoked as the result of an exception. If the pager queues a page read request, it branches to SCH$WAITM. SCH$WAITM branches to the scheduling code, and does not return to the pager.

   How is the page fault exception dismissed?

Paging

SOLUTIONS

1. The pager replaces the oldest page in the process working set. The process working set list is a circular buffer, with a single pointer advancing to the next replacement candidate.

   The contents of the physical page are not discarded when the page is removed from the working set. Rather, the physical page is placed on either the free page list or the modified page list. If a page fault occurs while the page is on either of these lists, the pager simply removes the page from the list and puts it back into the process working set.

   Virtual pages that are frequently referenced will occasionally be removed from the process working set. However, it is highly likely that the page will still be on one of the lists when a subsequent page fault occurs.

2. The page file control block imposes no limitation on the size of the page file. The form of page table entry that indicates that a virtual page is in the page file allows 22 bits for virtual block number. This requires that the page file be less than four megablocks. Because disks do not normally exceed one megablock, the maximum size of a single page file is much larger than the available disks. No limitation is currently imposed by the data structures.

3. PTE<30:27> must contain a protection code, even for invalid pages. Because the access check is performed before the valid bit is tested, the PTE for each page in process or system virtual space (specified by the contents of the appropriate region length register) must contain a protection code in these four bits.

4. The VAX linker sets up the first page of a native image as NO ACCESS for any access mode (PTE<30:27> = 0). A transfer of control to location 0 (via a CALLX, JMP, BRx, JSB, or BSBx instruction) causes a protection code access violation.

   The top two longwords on the stack will both be zero. The reason mask is a zero and the virtual address causing the exception is also a zero. This is the key to this type of programming error.

   The third longword on the stack is the PC of the offending instruction; the fourth longword is the PSL at the time of the exception.

Paging

SOLUTIONS

5.

a. Both the process virtual address and the system virtual address of the corresponding PxPTE must be translated.

b. If a translation buffer hit occurs on the process page table entry, the physical address can be formed immediately.

   Note that if a translation buffer hit occurs on the SPTE that maps the PxPTE, two translations are still required.

c. If the translation-not-valid fault occurs on the associated page table entry, bit 1 in the reason mask (on the top of the kernel stack) will be set. The second longword will contain the process virtual address in both cases.

d. The only difference in the state of the stack is bit 1 in the reason mask. The faulting virtual address is the process virtual address in both cases.

6. When a page is brought into a process working set, the modify bit is initially clear. Each time a write or modify access is made to a page, the modify bit is checked. If the bit is clear, it will be set by hardware both in the translation buffer and in the page table entry in physical memory.

   Thus, the first write or modify access will cause the bit to be set. All subsequent accesses (until the page is removed from the working set) will have no effect on the modify bit.

   The state of the modify bit will be checked when the page is removed from the working set. If the bit is set, the page must be put on the modified page list and written to secondary storage before the physical page can be reused by another process.

Paging

SOLUTIONS

7. The most common example of invalidating a single page table entry in the translation buffer is when the page is removed from the working set. If virtual addresses are deleted from a process (as a result of $CNTREG, $DELTVA, $DGBLSC system services, or at image exit) their associated translation buffer entries must be invalidated.

   If page protection is changed by using the $SETPRT system service, the corresponding translation buffer entries are invalidated.

8.

   a. The pager

      - Determines that the page is in an image file
      - Allocates a physical page
      - Allocates a working set list entry (WSLE) from the system working set list
      - Initiates the read operation
      - Sets the process scheduling state to page fault wait (PFW).

   b. Page fault wait state (PFW) (see question 9)

   c. Read-in-progress

9.

   a. Collided page wait state (COLPG)

   b. Read-in-progress (as in question 8) but with the collided page bit set

Paging

SOLUTIONS

10.

a. Collided page wait state (COLPG)

b. No further change from answer 8b. The collided page bit is already set.

11.

a. Computable (COM)

b. Computable (COM)

c. Computable (COM)

d. Active and valid

12.

a. Process HIGH

b. Scheduling is based strictly upon the relative priorities of computable processes, and not upon circumstances such as which process caused the initial page fault. Thus, although process LOW caused the initial page fault, and most of the work was performed by the pager in its context, process HIGH is likely to be the first process to use the valid page as a result of its higher priority.

13.

a. Disk storage is reduced because each image file does not require a separate copy of the shared sections. Physical memory requirements are reduced because only one copy of a shared section needs to exist in the system (and only those pages of a section actually used by one or more processes occupy physical memory).

SOLUTIONS

b. Although there is an additional level of indirection involved in resolving addresses within a shared image, address resolution only seems longer. With several processes referring to the section, there is a higher probability that the global page table entry (GPTE) is active and valid. If this is the case, page fault resolution is rapid. The working set list must be modified, the contents of the GPTE copied into the process PxPTE, and the share count for the physical page incremented in the PFN database.

14.

a. Using the index page of the microfiche, locate the directory for the SYS facility, and the entry for PAGEFAULT.LIS. Use the page number and page coordinates on the directory entry to locate the page(s) of fiche containing PAGEFAULT.LIS.

b. The routine responsible for freeing a working set list entry is called MMG$FREWSLE. If you are not sure of the name of a routine in a piece of code, the Table of Contents at the beginning of the listing may be helpful.

   MMG$FREWSLE is listed in the Table of Contents of PAGEFAULT.LIS. The second column in the contents contains the starting line number of the routine.

c. The piece of code in the pager that queues a page read request is listed as "Page Not Resident, Queue a Read Request" in the table of contents.

d. The SCH$WAITM routine is defined in the module SYSWAIT.

e. The SCHED module defines the routine SCH$SCHED.

f. The REI done by the scheduler dismisses the page fault exception, in this case. Note that the scheduler is invoked with a branch instruction, not with an interrupt. Therefore, the number of exceptions/interrupts equals the number of REIs, which is as it should be.

Paging

EXERCISES

Use the system recorded in the dump file OSI$LABS:CRASH1.DMP to answer the following questions.

1. The working set list, located in the process header, is one of the perprocess memory management data structures.

   a. Locate the process header of the current process, and record its address.

   b. Locate the top of the working set list for the current process.

   c. The entries at the top of the working set list catalog pages that are locked in the working set.

      Verify that the first few entries in the working set list for the current process catalog pages locked in the working set. (Consult Figure 14-5 in <u>VAX/VMS Internals and Data Structures</u> for the format of a working set list entry.)

2. Virtual address space is implemented with page tables.

   a. Locate the process header for the JOB_CONTROL process, and record its address.

b. Obtain the contents of the P0 base register for the job controller. (Remember that the process memory management registers, including the P0BR, are stored in the hardware PCB, which is part of the process header.)

c. Using the value in the P0BR, display the first 20 page table entries in the job controller's P0 page table. You should be able to do this with one SDA command.

   Look over the page table entries, and choose a valid entry (the high bit is set). Record the address of the entry and its contents below.

d. What is the protection code in the PTE you chose in part (c)?

   (HINTS: The protection code is stored in bits 27-30 of the PTE. Use Table 14-1 in VAX/VMS Internals and Data Structures to decipher this 4-bit code.)

e. Extract the PFN from the PTE in part (c). Display the PFN database information for this page frame using the SDA command

```
SDA>  SHOW PFN_DATA your_pfn
```

EXERCISES

f. One of the pieces of information displayed by SHOW PFN_DATA is the address of the PTE mapping the page.

   Does the PTE address displayed in part (e) match the address of the PTE you chose in part (c)?

3. Sam Wizard was analyzing a crash dump and located a page table entry for a valid page. The PTE was at address 8026302C. He examined the contents of the PTE, and displayed the PFN data for the mapped page frame. A portion of his output is shown below.

```
SDA> EXAMINE  8026302C
8026302C:  F9800523    "#..<U~>"

SDA> SHOW PFN_DATA 523

PFN    PTE ADDRESS    BAK        REFCNT  FLINK BLINK    TYPE
----   -----------  ---------    ------  ----- -----  ----------

0523     8032B6FC   0040FE70        1     0005 0000   02 GLOBAL

  STATE
----------
07 ACTIVE
```

The PTE address in the PFN database does not point back to Sam's page table entry. Why?

EXERCISES

4. Using the System Dump Analyzer (optional)

   a. Using a copy of the system page table obtained via SDA, construct a map of the actual placement in physical memory of the components of the permanently resident portion of the executive. These include:

      the system page table itself
      the PFN database
      the system header
      the nonpaged executive code and data
      the interrupt stack
      nonpaged dynamic memory

      HINT

      The table in VAX/VMS Internals and Data Structures that details the layout of system virtual address space gives the memory access codes for these components. These can be used to identify which pages in the SPT are associated with each component. You might find it easiest to work from the end of the SDA listing of the system page table. The components listed in the table are in the order that they will appear in the SPT. The actual page frame for each page is also listed in the SPT.

   b. Using a copy of the PFN database obtained by using SDA, determine how many pages of physical memory are available for paging. Determine how much memory must be used by the permanently resident executive. Go back to the system page table and determine how many pages are required by each component from question (a) above, and add the values together. Does this agree with the value (computed above) from the PFN database? It should.

Paging

SOLUTIONS

1.

a. Locate the process header of the current process by either:

- Issuing the SHOW SUMMARY command and noting the PHD address of the process in the CUR state.
- Issuing the SHOW PROCESS command and noting the PHD address.

b. The address of the top of the working set list is stored at offset PHD$L_WSL in the process header.

c. To determine whether or not a working set list entry catalogs a page locked in the working set, examine bit 5 of the entry.

The first few pages at the top of the process working set list are locked in the working set. These WSLEs catalog such pages as the kernel stack pages and the P1 pointer page.

2.

a. To locate the address of the process header for the JOB_CONTROL process, issue the SHOW SUMMARY command. The PHD address will appear in the display.

b. The contents of the P0 base register for the job controller are at offset PHD$L_P0BR in the process header.

c. To display the first 20 page table entries in the job controller's P0 page table, use the P0BR value from question (b) and issue the following command:

```
SDA> EXAMINE P0BR_value_from_b ; 50
```

This command will display the first 80 bytes (20 longwords) of the P0 page table. Remember that SDA will, by default, interpret the "50" in the above command as a hexadecimal number.

Paging

SOLUTIONS

d. Many of the pages in P0 space will have the protection code "0100", which means user mode read and write.

e. The PFN is stored in the low 21 bits of a valid PTE. Extract this PFN and issue the SHOW PFN_DATA command.

f. In most cases, the PTE address in the PFN database will match the address of your PTE. If not, see the answer to the next question.

3. The PTE address in the PFN database does not point back to Sam's page table entry. This is because Sam's page table entry maps a global page, as reflected in the TYPE array of the PFN database.

   If a page is global, the PFN PTE array contains the address of the <u>global</u> page table entry mapping the page, not the address of any one process page table entry.

4. See your instructor for the solution to this question.

Swapping

EXERCISES

1. The following figures show the state of the data structures related to a sample process working set at various times during outswap.

   The working set contains the following four virtual pages:

   Y - Global read-only (GRO), in only this process working set
   Z - Process page (PPG), direct I/O in progress
   W - Global read/write (GRW), in four process working sets
   X - Process page (PPG)

   Using the outswap scan table in your student workbook, and the template data structures provided, outswap the process body.

EXERCISES

a. Scan the working set list and decide which pages to write to the swap file. Record those pages in the swapper's I/O map, and drop the others from the working set.

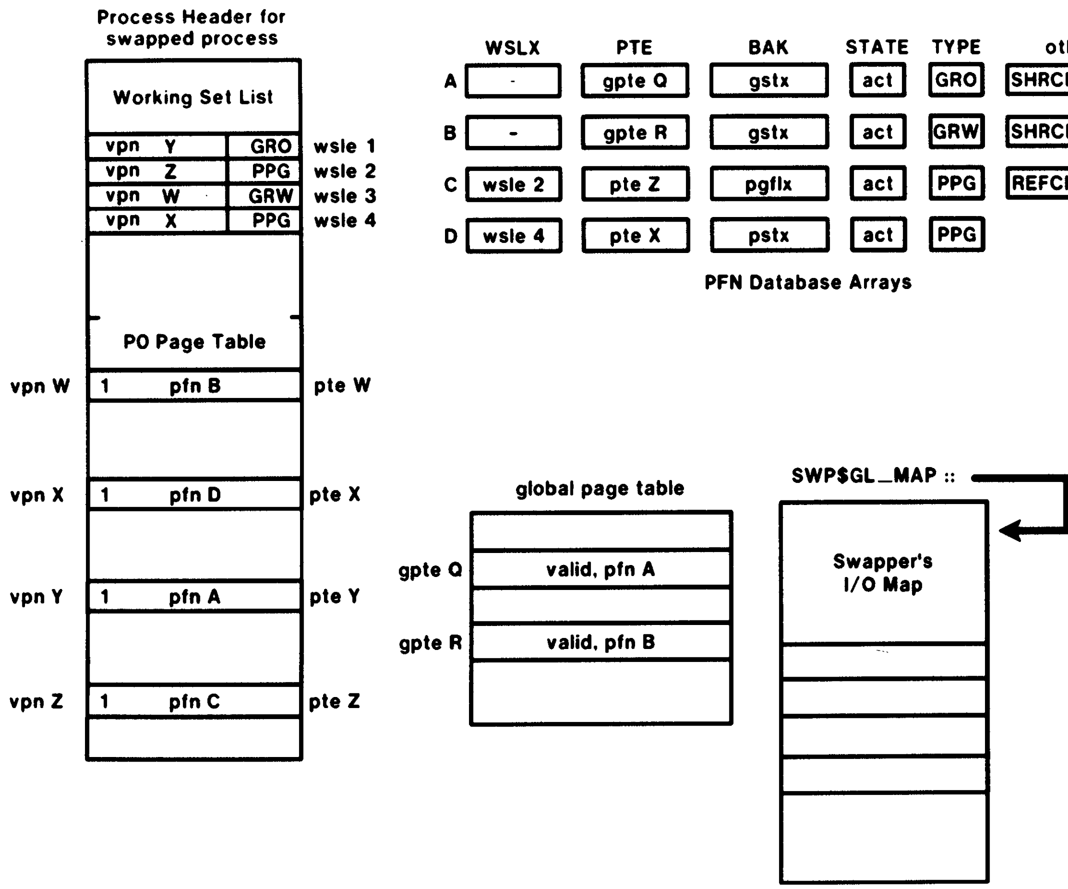


Figure 1 Template for Working Set List Outswap Scan

EXERCISES

b. Figure 2 shows the state of the data structures after the working set list outswap scan.

Record the state of the data structures after the swap I/O completes.

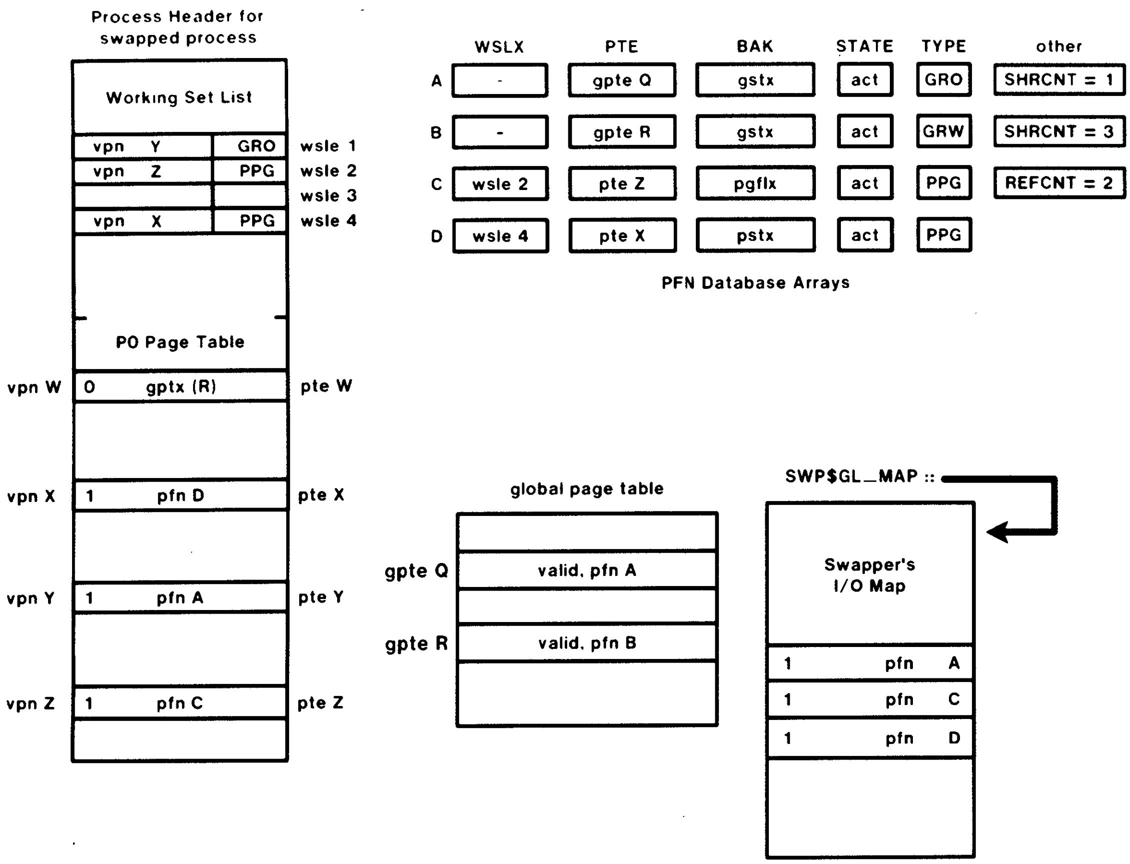


Figure 2 Template for Data Structures After Swap I/O

Swapping

EXERCISES

2. The following figures show the state of the data structures related to a sample process working set. The process header is in memory; the process body needs to be inswapped.

   This is not necessarily the same working set as in the previous exercise.

   The working set contains the following four virtual pages:

   X - Global read-only page (GRO), not in memory
   W - Process page (PPG), not in memory
   Y - Global read-only page (GRO), copy in memory (valid GPTE)
   Z - Process page (PPG), on free page list

   Using the inswap table in your student workbook, and the template data structures provided, inswap the process body. First allocate physical pages for the inswap, then rebuild the process body.

EXERCISES

a. Allocate physical pages for the inswap using the PFN database, and record the PFNs in the swapper's I/O map.

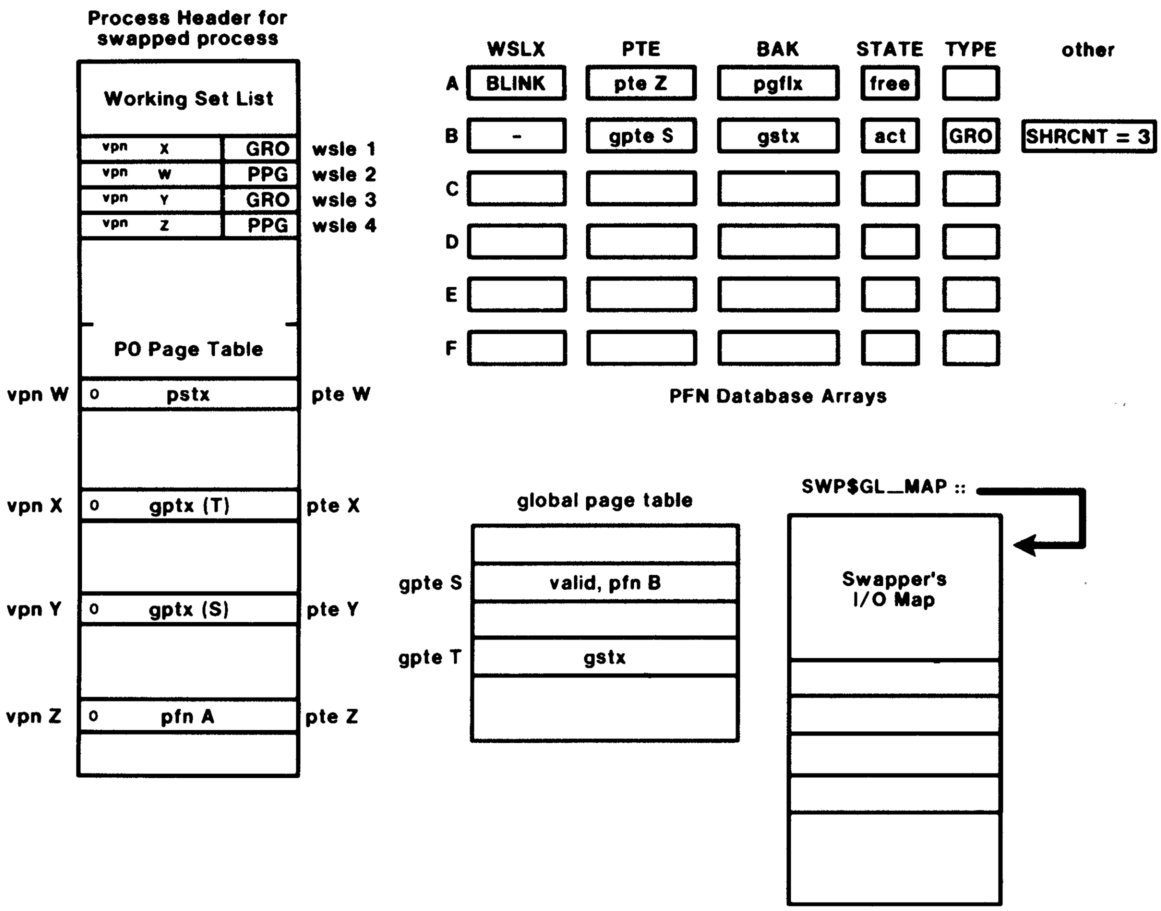


Figure 3 Template for Inswap I/O

EXERCISES

b. Rebuild the process working set, recording the PFNs in the P0 page table, and adjusting the global page table and the PFN database fields accordingly.

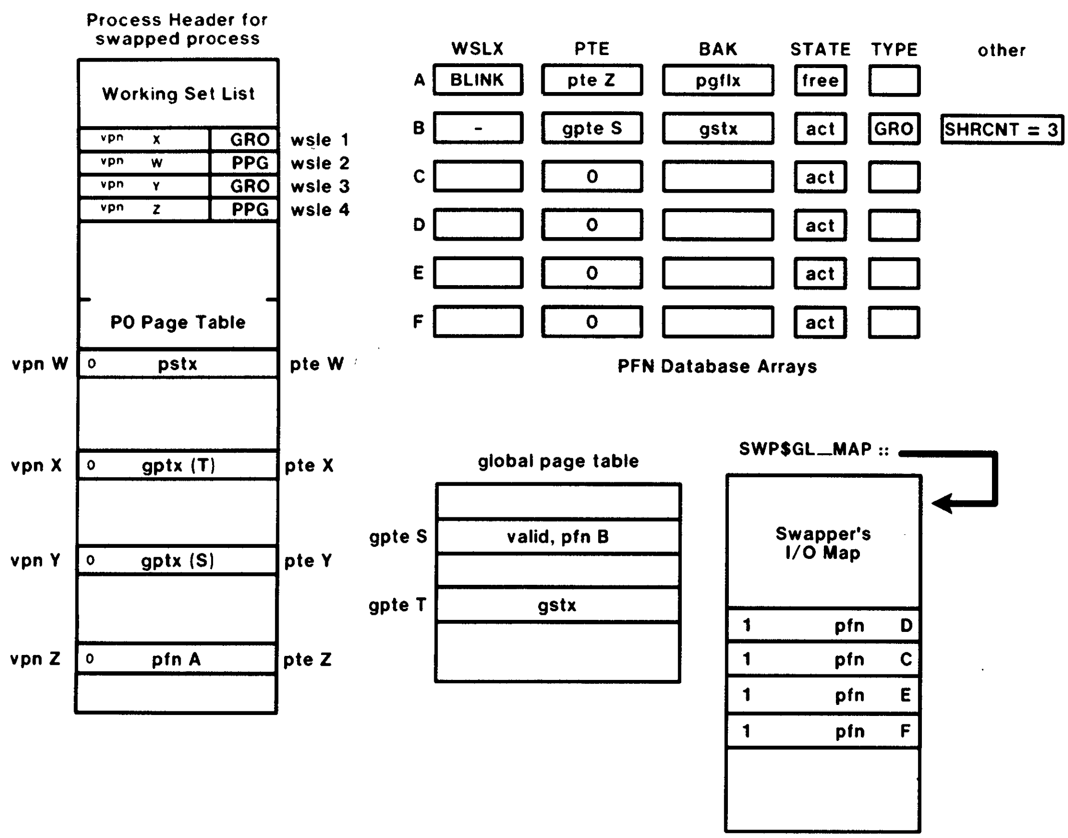


Figure 4 Template for Rebuilding Process Body on Inswap

Swapping

EXERCISES

3. To answer this question, you will need access to a set of VMS Version 4.x microfiche. See your instructor for the microfiche and a microfiche reader.

   VMS allows for dynamic adjustment of the working set list, within defined bounds, using the $ADJWSL system service. Adjustment may be performed by the user, and automatic adjustment is often performed by VMS at quantum end.

   When the working set list is extended, the swap slot currently allocated for the process may need to be traded for a larger slot.

   Find the VMS code responsible for allocating a larger swap slot for processes whose working set has expanded. List the name of the VMS source module, and the name of the routine below.

   (HINT: The code is in the SYS facility.)

4. Describe the special treatment given to pages with direct I/O in progress both at outswap and inswap times. Be sure to include the special case of the inswap occurring before the read or write operation completes.

EXERCISES

5. Discuss the special treatment given global pages by the swapper. Include both global read-only and global read/write pages in your discussion.

6. Why is there a need for a swapper in addition to a pager on VMS?

Swapping

EXERCISES

Questions 7 through 10 describe the interaction of two real-time processes and the swapper over an interval of time. Each question describes a particular event. For each process, indicate which process state will be occupied by that process. If a process does not exist, indicate this instead of a process state.

The initial process characteristics are:

Name	Priority	State
SWAPPER	16	HIB
LOW	20	CUR
HIGH	22	not yet created

7. Process LOW issues a $CREPRC system service request to create process HIGH, and continues to execute.

   a. SWAPPER

   b. LOW

   c. HIGH

EXERCISES

8. Process LOW issues a $HIBER system service request.

   a. SWAPPER

   b. LOW

   c. HIGH

9. The inswap operation completes and is reported to the scheduler. Assume that the SWAPPER performs further operations at IPL SYNCH before dropping the interrupt priority level.

   a. SWAPPER

   b. LOW

   c. HIGH

10. The SWAPPER drops the interrupt priority level from IPL SYNCH to IPL 0.

    a. SWAPPER

    b. LOW

    c. HIGH

Swapping

SOLUTIONS

1.

a. Figure 5 shows the state of the data structures after scanning the working set list.

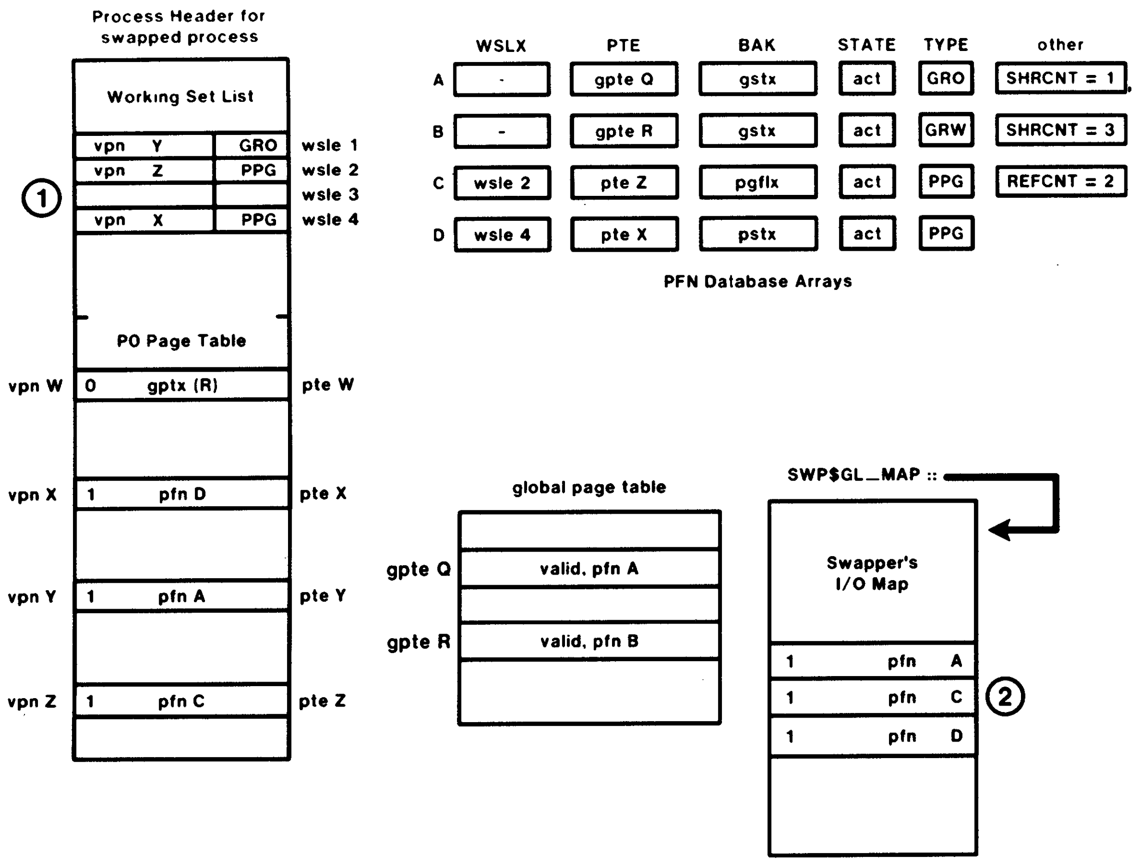


Figure 5 Working Set List After Outswap Scan

1. The global read/write page is removed from the working set.

2. The remaining elements of the working set are mapped by the I/O map, and then the I/O request is made.

SOLUTIONS

b. Figure 6 shows the state of the data structures after the swap I/O completes.

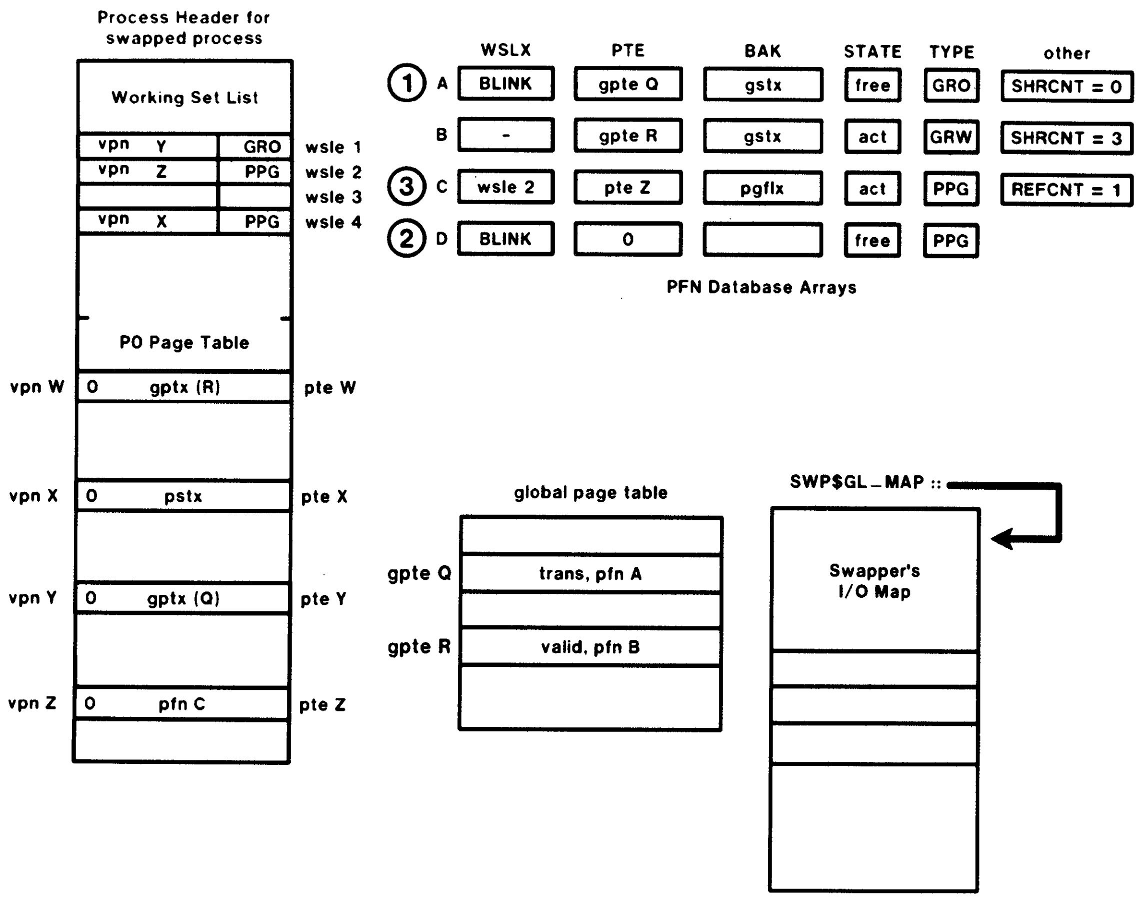


Figure 6 Data Structures After Swap I/O Completes

1. The global read-only page and the process page without I/O are placed on the free list.

2. Same as one.

3. The remaining process page (with I/O) has its REFCNT decremented by one.

Swapping

SOLUTIONS

2.

a. Figure 7 shows the data structures after physical pages are allocated for the inswap.

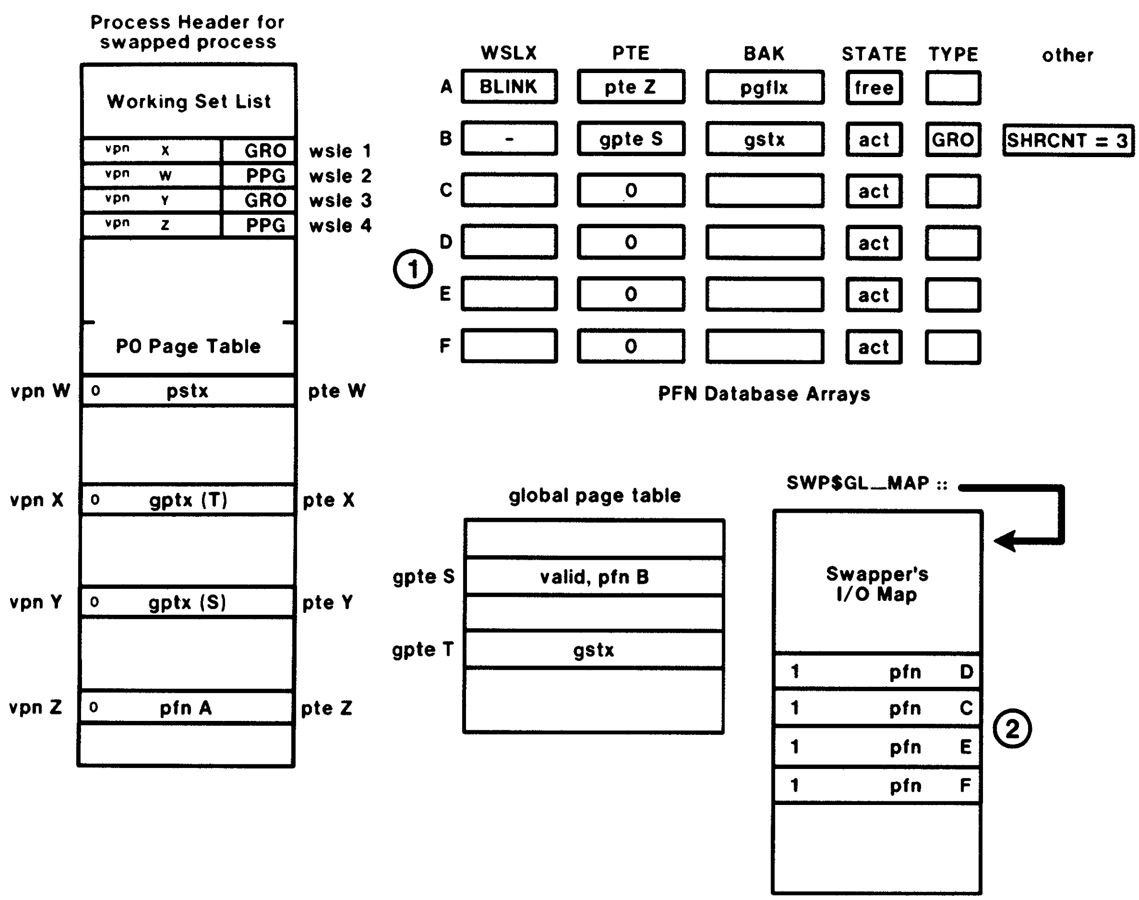


Figure 7 Data Structures After Physical Page Allocation

1. Swapper allocates pages from the free page list for every page in the process working set.

2. Swapper copies PFNs into its P0 space.

   Swapper issues read from disk which copies swapped working set into physical memory.

SOLUTIONS

b. The process working set is rebuilt.

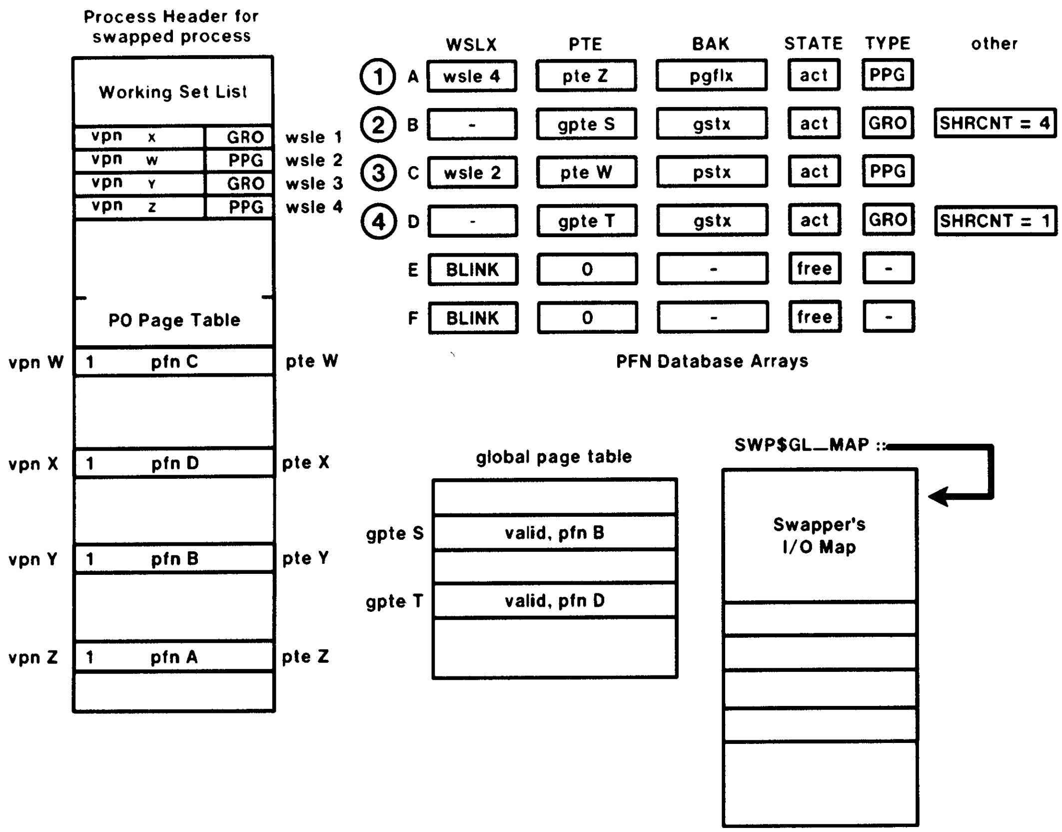


Figure 8 Working Set List and Rebuilt Page Tables

1. - PFN A still on free list so made valid.
   - PFN F released to the free page list.

2. - PFN B still valid so SHRCNT increased to 4
   - PFN copied to PTE Y
   - PFN E released to free page list

3. - PFN C copied to PTE W

4. - PFN D copied to PTE X
   - SHRCNT = 1

The actual order of operations is 4,3,2,1.

Swapping

SOLUTIONS

3. The code responsible for allocating a larger swap slot for processes whose working set has expanded is in module PAGEFAULT, routine MMG$FREWSLX.

   It is more efficient to have the pager allocate a larger slot, rather than a component such as $ADJWSL, because $ADJWSL only increases the size of your working set list. The pager is the code that actually adds pages to your working set. In this way, a larger swap slot is not allocated until the process has actually outgrown the current swap slot.

4. For either read-in-progress or write-in-progress, the pages in question are written to the swap file with the rest of the working set. However, because the reference count will not go to zero at outswap completion if the read or write is still outstanding, the pages will not be released to the free page list.

   If the read or write is still outstanding when the process is swapped back into memory, the swapper will take this into account by putting the page left behind into the rebuilt working set of the process and releasing the page frame from the swapper's special I/O page table.

   If the operation in progress was a write, the contents of the swap file are accurate, and the page is released to the free page list when the write operation completes.

   If the operation in progress was a read, the contents of the swap file are out of date. The write of the page to the swap file merely served to reserve a place in the swap file. This block is noted in the SWPVBN array in the PFN database. When the read operation completes, the page will be released to the modified page list. Subsequently, the modified page writer will write this page not to the page file but to its reserved location in the swap file. (If the inswap occurs before the modified page writer writes this page to the swap file, the page is simply faulted in from the modified page list while the swapper rebuilds the working set.)

Swapping

SOLUTIONS

Note that the only I/O that is relevant here is direct I/O because only direct I/O locks pages in the working set until I/O completion. Buffered I/O uses an intermediate buffer in system virtual address space (nonpaged dynamic memory). Thus, buffered operations do not require the user buffer to be in memory while the request is being processed. On a buffered write, the appropriate FDT routine transfers data (perhaps with modification) from the user buffer to a system buffer. On a buffered read, the I/O completion special kernel mode AST routine transfers the data from the system buffer into the specified user buffer.

5. At outswap time, each global read/write page is removed from the working set of the process. Each page must be refaulted into the working set after inswap only if it is referenced after inswap.

   Each global read-only page is written to the swap file if the PFN database SHRCNT value is one (only this process is using this page). Otherwise, the global page is removed from the working set and will need to be refaulted if it is referenced after inswap.

   At inswap time, global read-only pages are read along with the rest of the working set of the process. If the corresponding global page table entry (GPTE) is either valid or in transition, then the PxPTE points to the existing physical page, and the duplicate page is released to the free page list. If the GPTE is pointing to the global section table entry, the page is retained and both the PxPTE and GPTE are made valid.

6. The swapper manages physical memory on a system-wide basis, whereas the pager manages memory on a perprocess basis.

Swapping

SOLUTIONS

7.

a. COM -- process creation will awaken the swapper process.

b. CUR -- the stated assumption.

c. COMO -- the initial process state for every process.

8.

a. CUR -- the highest priority computable process.

b. HIB -- the stated assumption.

c. COMO -- still in the initial state.

9.

a. CUR -- the swapper is still executing.

b. HIB -- the stated assumption.

c. COM -- the purpose of the inswap operation is to make this process computable.

10.

a. COM -- dropping IPL will enable an IPL 3 scheduling interrupt to occur.

b. HIB -- the stated assumption.

c. CUR -- the highest priority computable process.

Swapping

EXERCISES

1. The values for SYSGEN parameters are stored in S0 space in the system image. These values can be accessed from a program.

   Write a program to modify the values of the FREELIM and FREEGOAL SYSGEN parameters. The program should:

   - Obtain and display the current value of each parameter.
   - Prompt the user for new values for the parameters, and modify them accordingly.
   - When the user desires, restore the original values of the parameters.

   If you need to use an elevated access mode, be aware of the dangerous implications of program errors.

EXERCISES

2. All students in the class should work together on this exercise.

   The purpose of the exercise is to analyze the interrelationships between memory management data structures, the swapper, the pager, and SYSGEN parameters.

   You will modify some SYSGEN parameters governing the free page list, and analyze the effects on the system.

   Please read through the entire exercise before beginning.

   a. Choose a few terminals close to each other to be used as monitoring stations.

      - From one terminal, watch the activity of the swapper process (using the SHOW PROCESS/CONTINUOUS command).

        Note that the PC of the swapper is always in S0 space. The swapper is a process, but its image resides in system space.

        Note also that the swapper spends most of its time in the HIB state.

      - From a second terminal, watch the processes that are consuming most of the CPU. (Use MONITOR PROCESSES/TOPCPU)

      - On a third terminal, watch the sizes of the free and modified page lists (MONITOR PAGE).

        Note the average size of each page list.

   b. Run your program from exercise (1).

      - Increase the values of FREELIM and FREEGOAL so they are within 100 pages of the average size of the free page list.

        The swapper should spend considerably more time in the CUR state than before you started the program. Why?

      - Restore the original values of the FREELIM and FREEGOAL parameters.

SOLUTIONS

1. Example 1 shows a program that modifies the values of FREELIM and FREEGOAL.

```
        .TITLE  SWAPLAB1
;++
;
; ABSTRACT:
;
;       This program allows you to change the values of
;       the FREELIM and FREEGOAL sysgen parameters.
;
; ENVIRONMENT:
;
;       Changes mode to exec. and to kernel.
;       CMKRNL privilege required.
;
;       Linked with SYS.STB:
;       $ LINK  SWAPLAB1, SYS$SYSTEM:SYS.STB/SELECTIVE
;
; SIDE EFFECTS:
;
;       The above mentioned sysgen parameters are changed, and
;       restored to their original values before program exit.
;
;--
        .LIBRARY        /OSI$LABS:OSIMACROS/    ; for I/O

; ***************************************************************
        .PSECT  DATA    NOEXE,WRT,NOSHR

ASCII_DESC:     .LONG   10                      ; fixed length string
                .ADDRESS  ASCII_STR             ;  descriptor
ASCII_STR:      .BLKB   10
DYNDESC:        .WORD   20                      ; dynamic string for
                .BYTE   14,2                    ;  converting integers
                .LONG   0                       ;  to strings; and dummy
CONCAT_DESC:    .LONG   80                      ; descriptor for
                .ADDRESS  CONCAT_STR            ;  concatenating strings
CONCAT_STR:     .BLKB   80
FREELIM_STR:    .ASCID  /The value of FREELIM is:  /
FREEGOAL_STR:   .ASCID  /The value of FREEGOAL is:  /
FREELIM_PROMPT: .ASCID  /Enter new value for FREELIM: /
```

Example 1 Program to Modify Value of Some Parameters
(Sheet 1 of 3)

SOLUTIONS

```
FREEGOAL_PROMPT:.ASCID  /Enter new value for FREEGOAL: /
RESTORE_PROMPT:
        .ASCID  /Press RETURN to restore original parameter values:/
E_ARG_LIST:     .LONG    2                  ; for $cmexec call.
                .ADDRESS  OLD_FREELIM       ; 2 writeable arguments,
                .ADDRESS  OLD_FREEGOAL      ;  passed by reference
K_ARG_LIST:     .LONG    2                  ; for $cmkrnl call
                .BLKL    1                  ;  FREELIM, passsed by value
                .BLKL    1                  ;  FREEGOAL, passsed by value
OLD_FREELIM:    .LONG    0
OLD_FREEGOAL:   .LONG    0
NEW_FREELIM:    .LONG    0
NEW_FREEGOAL:   .LONG    0
; ***************************************************************
        .PSECT  CODE    EXE,NOWRT,PIC,SHR
START:  .WORD   ^M<R3,R4,R5,R6>

;       read current values of parameters in exec. mode
        $CMEXEC_S       routin= 100$, arglst= E_ARG_LIST
        CHECK_STATUS

;       convert and display old freelim and freegoal values
20$:    CONV_BIN_INT    OLD_FREELIM, ASCII_DESC
        CONCAT2         CONCAT_DESC, FREELIM_STR, ASCII_DESC
        DISPLAY         CONCAT_DESC

        CONV_BIN_INT    OLD_FREEGOAL, ASCII_DESC
        CONCAT2         CONCAT_DESC, FREEGOAL_STR, ASCII_DESC
        DISPLAY         CONCAT_DESC

;       prompt for, and convert to binary, new freelim value
        PUSHAL  FREELIM_PROMPT
        PUSHAL  DYNDESC
        CALLS   #2, G^LIB$GET_INPUT
        CHECK_STATUS
        CONV_INT_BIN    DYNDESC, NEW_FREELIM

;       prompt for, and convert to binary, new freegoal value
        PUSHAL  FREEGOAL_PROMPT
        PUSHAL  DYNDESC
        CALLS   #2, G^LIB$GET_INPUT
        CHECK_STATUS
        CONV_INT_BIN    DYNDESC, NEW_FREEGOAL
```

Example 1 Program to Modify Value of Some Parameters
(Sheet 2 of 3)

```
;       put new values in k_arg_list and write to s0 space
        MOVL    NEW_FREELIM, K_ARG_LIST+4
        MOVL    NEW_FREEGOAL, K_ARG_LIST+8
        $CMKRNL_S       routin= 200$, arglst= K_ARG_LIST
        CHECK_STATUS

;       stall until ready to restore old parameter values
        PUSHAL  RESTORE_PROMPT
        PUSHAL  ASCII_DESC                  ; dummy value, never used
        CALLS   #2, G^LIB$GET_INPUT
        CHECK_STATUS

;       put OLD values in k_arg_list and write to s0 space
        MOVL    OLD_FREELIM, K_ARG_LIST+4
        MOVL    OLD_FREEGOAL, K_ARG_LIST+8
        $CMKRNL_S       routin= 200$, arglst= K_ARG_LIST
        CHECK_STATUS

        MOVL    #SS$_NORMAL, R0             ; set normal completion
        RET                                 ; all done

; *****************  executive mode code  *********************
100$:   .WORD   ^M<>
;
;       Obtain current values of FREELIM and FREEGOAL
;       Executes in exec. mode to read SGN fields.
;
        MOVL    G^SGN$GL_FREELIM, @4(AP)
        MOVL    G^SGN$GL_FREEGOAL, @8(AP)
        MOVL    #SS$_NORMAL, R0
        RET                                 ; finished in exec. mode

; *****************  kernel mode code  ***********************
200$:   .WORD   ^M<>
;
;       Modify FREELIM and FREEGOAL parameters in S0 space.
;       Must be done in kernel mode.

        MOVL    4(AP), G^SGN$GL_FREELIM
        MOVL    8(AP), G^SGN$GL_FREEGOAL
        MOVL    #SS$_NORMAL, R0
        RET                                 ; finished in kernel mode

        .END    START
```

Example 1 Program to Modify Value of Some Parameters
(Sheet 3 of 3)

Swapping

SOLUTIONS

2.

a. At the first terminal, enter the command:

```
$ SHOW PROCESS/CONTINUOUS swapper-pid
```

At the second terminal, enter the command:

```
$ MONITOR PROCESSES/TOPCPU
```

At the third terminal, enter the command:

```
$ MONITOR PAGE
```

b. When the values of FREELIM and FREEGOAL are increased to almost match the average number of free pages, the swapper is forced to work furiously to maintain the free page count.

I/O Concepts and Flow

EXERCISES

1. Briefly describe the functions of the following components of the I/O system.

   a. Record Management Services (RMS)

   b. I/O System Services

   c. FDT routines

   d. Extended QIO Procedures (XQP)

2. Listed below are acronyms for some data structures in the I/O database. For each structure, give its full name and briefly describe its function.

   a. UCB

   b. CCB

   c. DDB

I/O Concepts and Flow

SOLUTIONS

1.

a. The Record Management Services (RMS) consist of file and record handling routines. All high-level language I/O operations go through RMS.

b. The I/O System Services, for example $ASSIGN and $QIO, are the most primitive I/O routines that have a user interface. These routines create and pre-process the data structures necessary for performing I/O operations.

c. FDT routines process the device-dependent parameters on a call to $QIO. They are written (or selected) by the author of a device driver.

   FDT routines execute in process context, which enables them to access the caller's P0 and P1 space.

d. The Extended QIO Procedures (XQP) interpret and maintain the Files-11 on-disk structure. Prior to Version 4.0 these functions were handled by disk Ancillary Control Processes (ACPs).

2.

a. UCB - The Unit Control Block contains information for a device unit. It is also used as a listhead for storage of IRPs by the driver.

b. CCB - The Channel Control Block links a 'channel' to a specific UCB by storing the virtual address of the UCB. It also contains other channel-related information.

c. DDB - The Device Data Block contains information common to all devices on a controller.

I/O Concepts and Flow

EXERCISES

The purpose of the following exercises is to learn how to find drivers and their data structures in system space. When a system crash occurs, it may be necessary to trace through the I/O database to find information to help debug the crash.

1. The SYSGEN utility is used for several purposes:

   - To configure the I/O database for the known devices on the system, and load in the VMS driver code
   - To configure and load user-written drivers
   - As an editor, to examine and modify the dynamic system parameters, and to create a new set of parameters to be used on the next reboot

   NOTE: You will need CMEXEC privilege to do this exercise.

   a. Enter SYSGEN with the following command:

   ```
   $ RUN SYS$SYSTEM:SYSGEN
   ```

   and examine the HELP for the SET and SHOW commands.

   b. Issue the SHOW/DEVICES command, and compare the names of the drivers with the names of the devices they handle.

   c. Using the SHOW commands, examine some of the system parameters discussed thus far in the course.

I/O Concepts and Flow

EXERCISES

2. Analyze the current system (ANALYZE/SYSTEM) to answer the following questions.

   NOTE: You need CMKRNL privilege to do this exercise.

   a. Starting with IOC$GL_DEVLIST, find the UCB for the terminal you are on.

      You may want to read a symbol table file into your SDA session so you can access $DDBDEF and $UCBDEF. SYS$SYSTEM:SYSDEF.STB or OSI$LABS:GLOBALS.STB should provide those symbols.

      Note that when you format a UCB, SDA does not translate any of the data into ASCII.

   b. Find the UCB for your default disk.

I/O Concepts and Flow

SOLUTIONS

1.

a. Enter SYSGEN with the following command:

```
$ RUN SYS$SYSTEM:SYSGEN
```

b. SYSGEN> SHOW/DEVICES

c. Issue commands such as SYSGEN> SHOW BORROWLIM.

2. $ ANALYZE/SYSTEM

a. Starting with IOC$GL_DEVLIST, search the DDBs until you find the DDB that describes the controller for your terminal. Then search the UCBs off that DDB until you find the UCB for your terminal.

b. First locate the DDB for the controller for your default disk. Then search the UCBs off that DDB until you find the UCB for your disk.

RMS Implementation and Structure

EXERCISES

1. Using the information in your student workbook, your Source Listings book, and the code in Examples 1 and 2, trace an OPEN operation through the relevant code modules.

   Assume it is an open on an existing relative file that is not installed.

   a. The OPEN operation is initiated with a call to SYS$OPEN. In which area of virtual address space does the RMS vector SYS$OPEN reside?

   b. Execution of the CHME instruction in the SYS$OPEN vector causes an exception. The system vectors through the SCB to the executive change mode dispatcher (EXE$CMODEXEC).

      What is the name of the routine that gains control in executive mode?

   c. Using RMS.MAP, determine the RMS code module that contains the routine from part (b).

   d. The routine from part (b) appears in Example 1. After the routine performs the common setup, it dispatches to organization-dependent routines.

      Which organization-dependent routine will be invoked for this file OPEN?

   e. Using RMS.MAP, determine the RMS code module that contains the organization-dependent routine for this file OPEN.

2. When processing files, RMS copies the FAB information into an IFAB. Why does it make a copy of this information?

EXERCISES

```
        $BEGIN   RMS0OPEN,000,RM$RMS,<DISPATCH FOR OPEN OPERATION>
;++
; Facility: RMS32
;
; Abstract:
;                This module is the highest level control routine
;                to perform the $open function.
;--
        .SBTTL   DECLARATIONS
;
; Include Files:
;
; Macros:
;
        $ARMDEF
        $CCBDEF
        $CHPCTLDEF
        $DEVDEF
        $FABDEF
        $FCBDEF
        $FIBDEF
        $FWADEF
        $IFBDEF
        $IPLDEF
        $KFEDEF
        $NAMDEF
        $PCBDEF
        $PRDEF
        $PSLDEF
        $RJBDEF
        $RJRDEF
        $RMSDEF
        $UCBDEF
        $WCBDEF
        $XABALLDEF
        $XABKEYDEF
        $XABSUMDEF
;
; Equated Symbols:
;
        FOP=FAB$L_FOP*8                      ; bit offset to fop
;
; Own Storage:
;
```

Example 1 Excerpt from RMS0OPEN.MAR (Sheet 1 of 16)

EXERCISES

```
RM$XABOPN_ARGS::
        .BYTE   XAB$C_SUM,XAB$C_SUMLEN,XBC$C_OPNSUM3
        .BYTE   XAB$C_KEY,XAB$C_KEYLEN_V2,XBC$C_OPNKEY3
        .BYTE   XAB$C_ALL,XAB$C_ALLLEN,XBC$C_OPNALL3
        .BYTE   0

        .SBTTL  RMS$OPEN - $OPEN ROUTINE
;++
;
; RMS$OPEN -- Open routine.
;
;  This routine performs the highest level $open processing.
;  Its functions include:
;
;       1. Common setup.
;       2. Dispatch to organization-dependent code.
;       3. Dispatch to the display routine.
;
;
; Calling Sequence:
;
;       Entered from exec as a reult of user's calling sys$open
;       (e.g., by using the $open macro).
;
; Input Parameters:
;
;       AP      user's argument list addr
;
; Implicit Inputs:
;
;       The contents of the fab and possible related user interface
;       blocks.
;
; Output Parameters:
;
;       R0      status code
;       R1      destroyed
;
; Implicit Outputs:
;
;       The various fields of the fab are filled in to reflect
;       the status of the open file. (see rms functional spec for
;       a complete list.)
;       An ifab is initialized to reflect the open file.
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 2 of 16)

EXERCISES

```
;       A completion ast is queued if so specified by the user.
;
; Completion Codes:
;
;       Standard rms (see functional spec for list).
;
; Side Effects:
;
;       none
;--
        $ENTRY  RMS$OPEN
        $TSTPT  OPEN
        BSBW    RM$FSETI                        ; do common setup
                                                ; note: does not return on
;
; Alternate entry point for open.  Called from create when the cre
; was a restart operation.
;
RM$OPEN_ALT::
;
; An ifab has been set up
;
        CLRL    R10                             ; no FWA to start w
        BSBW    RM$PRFLNM                       ; process file name

        BBC     #FAB$V_BRO,IFB$B_FAC(R9),10$    ; branch if bro not
        CSB     #FAB$V_BIO,IFB$B_FAC(R9)        ; clear bio (implie
                                                ;  by bro without r
10$:    BLBC    R0,50$                          ; exit on error fro
20$:    BBC     #FAB$V_KFO+FOP,(R8),22$         ; branch if kfo not
        MOVL    FAB$L_NAM(R8),R7                ; get name block
        BSBW    RM$CHKNAM                       ; can we use it?
        BLBC    R0,22$                          ; nope
        BBS     #FWA$V_NODE,(R10),25$           ; branch on network
        BBS     #FWA$V_EXP_VER,(R10),25$        ; explicit version,
;
; INS$KF_SCAN returns R0:
;
;       SS$_NORMAL      - known file found but not open
;       RMS$_KFF        - known file found and it was open
;       RMS$_FNF        - known file not found
;
        PUSHAL  FAB$L_CTX(R8)                   ; return KFE in fab
        PUSHL   R7                              ; filled in name bl
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 3 of 16)

EXERCISES

```
        CALLS   #2,INS$KF_SCAN                  ; go try known file
        BLBC    R0, 30$                         ; not installed

        CMPL    #RMS$_KFF,R0                    ; was the file inst
        BNEQ    40$                             ; installed, but no

        PUSHL   R0                              ; preserve status
        $CMKRNL_S -                             ; kernel mode routi
                ROUTIN=RM$KNOWNFILE             ; modify system ref
        MOVL    (SP)+,R1                        ; recover status
        BLBC    R0,30$                          ; can't access this
        MOVL    R1,R0                           ; set appropriate s
        BRW     RM$CREATEXIT                    ; exit from open im
                                                ;  if found/open or

22$:    BRB     100$                            ; helper branch

;
; Couldn't find this file spec as a known file,
; try to get another if searchlist is present
;
25$:    RMSERR  FNF                             ; setup appropriate
30$:    BBC     #FWA$V_SLPRESENT,(R10),100$     ; if no searchlist
        BSBW    RM$CHK_SLIST                    ; try again
        BLBS    R0,20$                          ; did it work?
        BLBC    R1,60$                          ; should we try aga
        BSBW    RM$DEALLOCATE_FWA               ; release exhausted
        CSB     #FAB$V_KFO+FOP,(R8)             ; Make sure not KFO
        BRB     RM$OPEN_ALT                     ; try for non-KFO o
;
; Try to open the knownfile normally; if this fails, then go thru t
; file searchlist lookup logic
;
40$:    SSB     #FAB$V_NAM+FOP,(R8)             ; Force NAM block o
        BSBW    RM$SETDID                       ; process the direc
        BLBC    R0,30$                          ; check search list
        BSBW    RM$ACCESS                       ; access the file
        BLBC    R0,30$                          ; check search list
        BRB     RM$OPEN_CIF                     ; continue with ope

60$:    BRW     ERROR                           ; no, return error
;
; There was a problem with the file spec, try to get another if sea
; are present
```

Example 1 Excerpt from RMS0OPEN.MAR (Sheet 4 of 16)

EXERCISES

```
50$:    TSTL    R10                             ; have a FWA?
        BEQL    60$                             ; if not don't chec
        BBC     #FWA$V_SLPRESENT,(R10),60$      ; if no search list
        BSBW    RM$CHK_SLIST                    ; try again
        BLBC    R0,60$                          ; did it work?

100$:   BSBW    RM$SETDID                       ; process the direc
        BLBC    R0,50$                          ; check search list
        BSBW    RM$ACCESS                       ; access the file
        BLBC    R0,50$                          ; check search list
;
;  Return point for create turned into open via 'cif' bit.
;
RM$OPEN_CIF::
        BLBC    R0,ERROR                        ; exit on error
        BSBW    RM$FILLNAM                      ; fill in nam block
        BLBC    R0,ERROR                        ; exit on error
;
;  Copy the DID from the NAM block into the FIB if this is an OPEN
;
        BBC     #FAB$V_NAM+FOP,(R8),5$          ; skip if not open
        TSTL    R7                              ; have a NAM block
        BEQL    5$                              ; ce la vie'
        MOVL    FWA$Q_FIB+4(R10),R1             ; get addr of FIB
        BEQL    5$                              ; no FIB, no DID
        TSTW    FIB$W_DID(R1)                   ; have a DID?
        BNEQ    5$                              ; continue if so
        MOVL    NAM$W_DID(R7),FIB$W_DID(R1)     ; copy DID
        MOVW    NAM$W_DID+4(R7),FIB$W_DID+4(R1) ; copy DID last wor
;
;  Make sure eof info is in "eof blk + 1, 0 offset" form.
;
5$:     CMPW    IFB$W_FFB(R9),-                 ; is last block ful
                IFB$L_DEVBUFSIZ(R9)
        BLSSU   10$                             ; branch if not
        INCL    IFB$L_EBK(R9)                   ; bump eof block
        CLRW    IFB$W_FFB(R9)                   ; and zero offset
10$:    BBS     #IFB$V_DAP,(R9),DAPRTN          ; branch if network
;
;  Dispatch to organization-dependent open code.
;
        BBC     #DEV$V_SQD,IFB$L_PRIM_DEV(R9),- ; branch if not mag
                20$
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 5 of 16)

EXERCISES

```
        BBC     #DEV$V_MNT,IFB$L_PRIM_DEV(R9),- ; error, if magtape
                ERRDNR
        BBS     #DEV$V_DMT,IFB$L_PRIM_DEV(R9),- ; error, if magtape
                ERRDNR
20$:    CASE    TYPE=B,-                        ; pick up correct r
                SRC=IFB$B_ORGCASE(R9),-
                DISPLIST=<RM$OPEN1,RM_OPEN2_BR,RM$OPEN3>
;++
;  Error returns
;--

; Unknown file organization  -  verify bio (or bro) accessed.
;
        RMSERR  ORG                             ; org not supported
        BITB    #FAB$M_BIO!FAB$M_BRO,-          ; either bio or bro
                IFB$B_FAC(R9)
        BEQL    ERROR                           ; branch if not (er
        RMSSUC
        BRW     RM$COPRTN                       ; all finished open

RM_OPEN2_BR:
        JMP     RM$OPEN2                        ; branch aid

ERRRFM: RMSERR  RFM                             ; bad rfm field
        BRB     ERROR

ERRDNR: RMSERR DNR                              ; device not mounte
         BRB     ERROR

ERRIRC: RMSERR  IRC                             ; illegal fixed rec

ERROR:  CSB     #IFB$V_ACCESSED,(R9)            ; don't write file
        CMPB    #IFB$C_IDX,IFB$B_ORGCASE(R9)    ; indexed file?
        BNEQ    5$                              ; branch if not...c
        BBS     #IFB$V_DAP,(R9),5$              ; branch if network
        PUSHL   R0                              ; push error code o
        MOVL    R9,R10                          ; RM$CLOSE3 expects
        JSB     G^RM$CLOSE3                     ; close indexed fil
        POPL    R0                              ; pop error code fr
5$:     BRW     RM$CLSCU                        ; clean up and retu
```

Example 1 Excerpt from RMS0OPEN.MAR (Sheet 6 of 16)

```
;  Return here from org-dependent routines.
;
RM$COPRTN::
       BLBC     R0,ERROR
       BBS      #IFB$V_DAP,(R9),DAPRTN          ; branch if network
;
; Now handle summary, allocation, and key xab's.
;
       MOVAB    RM$XABOPN_ARGS,AP               ; move addr of xab
       BSBW     RM$XAB_SCAN                     ; scan the xab chai
       BLBC     R0,ERROR                        ; get out on error
;
;  Override run-time deq with user value, if any.
;
DAPRTN:         MOVW    FAB$W_DEQ(R8),IFB$W_RTDEQ(R9)
       BNEQ     5$                              ; branch if speced
       MOVW     IFB$W_DEQ(R9),IFB$W_RTDEQ(R9)   ; otherwise pick up
;
; From file header.
;
       MOVW     IFB$W_DEQ(R9),FAB$W_DEQ(R8)     ; and put in fab
;
;  Return bdb and i/o buffer to free space and page lists.
;
5$:    BBC      #IFB$V_AT,IFB$B_JNLFLG(R9),7$   ; skip if not AT jo
       BSBW     WRITE_AT_JNL                    ; write AT record -
7$:    BSBW     RM$RELEASALL                    ; return bdb and bu
;
;  Validate rfm.
;
       ASSUME   IFB$V_RFM       EQ      0
       ASSUME   IFB$S_RFM       EQ      4

       BICB2    #^XF0,IFB$B_RFMORG(R9)          ; leave only rfm in
;
;  Check for rfm in supported range.
;
       BBS      #IFB$V_BIO,IFB$B_FAC(R9),10$    ; don't check if bi
       CMPB     IFB$B_RFMORG(R9),#FAB$C_MAXRFM
       BGTRU    ERRRFM
;
;  If fixed length record format, then set mrs from lrl in case this
;  is an fcs-11 file.
;
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 7 of 16)

EXERCISES

```
10$:    CMPB     IFB$B_RFMORG(R9),#FAB$C_FIX       ; fixed len rec?
        BNEQ     20$                               ; branch if not
        MOVW     IFB$W_LRL(R9),IFB$W_MRS(R9)       ; set record length
        BLEQ     ERRIRC                            ; branch if invalid
;
; force stream format files to appear to have RAT non-null,
; even if they don't.
;
        ASSUME   FAB$C_STM        LT      FAB$C_STMLF
        ASSUME   FAB$C_STM        LT      FAB$C_STMCR

20$:    CMPB     IFB$B_RFMORG(R9),#FAB$C_STM       ; stream format?
        BLSSU    RM$COPRTN1                        ; nope
        BITB     #<FAB$M_CR!FAB$M_FTN!FAB$M_PRN>,-
                 IFB$B_RAT(R9)                     ; carriage control
        BNEQ     RM$COPRTN1                        ; ok
        BISB2    #FAB$M_CR,IFB$B_RAT(R9)           ; force RAT=CR
;
;  Return point for indirect open of process permanent file.
;
;  Set the rfm, rat, org, and mrs fields into the fab.
;
RM$COPRTN1::
        MOVB     IFB$B_RFMORG(R9),FAB$B_RFM(R8)    ; set rfm
        MOVB     IFB$B_RAT(R9),FAB$B_RAT(R8)       ; set rat
;
;  Return point for indirect open of process permanent file and rfm
;  rat already set.
;
RM$COPRTN2::
        INSV     IFB$B_ORGCASE(R9),-               ; set org
                 #FAB$V_ORG,#FAB$S_ORG,-
                 FAB$B_ORG(R8)
        BBC      #IFB$V_SEQFIL,(R9),10$            ; branch if not seq

        ASSUME   FAB$C_SEQ        EQ      0

        CLRB     FAB$B_ORG(R8)                     ; this is really a
                                                   ;  Orgcase says rel
10$:    MOVW     IFB$W_MRS(R9),FAB$W_MRS(R8)       ; set mrs
        MOVW     IFB$W_GBC(R9),FAB$W_GBC(R8)       ; set gbc
;
;  If vfc record format, check for 0 fixed header size and if
;  found make it 2 bytes.
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 8 of 16)

```
        CMPB    IFB$B_RFMORG(R9),#FAB$C_VFC
        BNEQ    20$                                 ; omit check if not
        TSTB    IFB$B_FSZ(R9)                       ; check for default
        BNEQ    30$                                 ; branch if value s
        MOVB    #2,IFB$B_FSZ(R9)                    ; set default value
        BRB     30$                                 ; continue
20$:    CLRB    IFB$B_FSZ(R9)                       ; guarantee 0 fsz f
                                                    ;  (note: fixes rms
30$:    RMSSUC                                      ; indicate successf

;++
;
;  Common exit for $create and $open.
;
;--

CREOPEN_EXIT:
        BLBS    R0,2$                               ; branch if no erro
1$:     BRW     ERROR                               ; otherwise go to e
;
;  Save the various close option bits in ifab
;
2$:     CSB     #IFB$V_CREATE,(R9)                  ; clear the "doing
        BBS     #IFB$V_PPF_IMAGE,(R9),5$            ; don't save option

        ASSUME  FAB$V_RWC+1     EQ      FAB$V_DMO
        ASSUME  FAB$V_DMO+1     EQ      FAB$V_SPL
        ASSUME  FAB$V_SPL+1     EQ      FAB$V_SCF
        ASSUME  FAB$V_SCF+1     EQ      FAB$V_DLT

        EXTZV   #FAB$V_RWC+FOP,#5,(R8),R1           ; get option bits

        ASSUME  IFB$V_RWC+1     EQ      IFB$V_DMO

        ASSUME  IFB$V_SPL+1     EQ      IFB$V_SCF
        ASSUME  IFB$V_SCF+1     EQ      IFB$V_DLT

        INSV    R1,#IFB$V_RWC,#5,(R9)               ; and save them
;
;  If this is foreign magtape, rewind the tape if rwo is set.
;
        BBC     #DEV$V_FOR,IFB$L_PRIM_DEV(R9),5$; branch if not for
        BBC     #DEV$V_SQD,IFB$L_PRIM_DEV(R9),5$; or if not magtape
        BBC     #FAB$V_RWO+FOP,(R8),5$              ; or if rwo not spe
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 9 of 16)

```
        BSBW    RM$REWIND_MT                    ; rewind the tape
        BLBC    R0,1$                           ; branch on error
;
;   Set 'blk' bit in ifab for magtape.
;
5$:     BBC     #DEV$V_SQD,IFB$L_PRIM_DEV(R9),8$; branch if not mag
        BISB2   #FAB$M_BLK,IFB$B_RAT(R9)        ; set no spanning b
;
;   Set the fsz, bks, stv, alq, dev, and sdc fields into fab.
;
8$:     MOVB    IFB$B_FSZ(R9),FAB$B_FSZ(R8)     ; set fsz
        MOVB    IFB$B_BKS(R9),FAB$B_BKS(R8)     ; set bks
        BBC     #IFB$V_SEQFIL,(R9),9$           ; branch not seq fi
        CLRB    FAB$B_BKS(R8)                   ; always zero for s
9$:     MOVW    IFB$W_CHNL(R9),FAB$L_STV(R8)    ; set stv to chan #
        MOVL    IFB$L_HBK(R9),FAB$L_ALQ(R8)     ; set alq
;
;   Move device characteristics bits into the fab.
;
        BSBW    RM$RET_DEV_CHAR                 ; set DEV and SDC f
;
;   Check for user file open option.
;
20$:    BBS     #FAB$V_UFO+FOP,(R8),40$         ; branch if ufo opt
        BRW     RM$EXRMS                        ; return to user
;
;   Leave file open for user but remove ifab
;   (no further rms operations available on this file).
;
40$:    BRW     RM$RETIFB

;   Common create clean up and exit
;   Return all bdb's and buffers to free space list, causing unlock

RM$CREATEXIT::
        PUSHL   R0                              ; save status code
;
;   Entry point with status already pushed on the stack.
;
RM$CREATEXIT1::
        BBC     #IFB$V_AT,IFB$B_JNLFLG(R9),10$  ; skip if not AT jo
        BSBW    WRITE_AT_JNL                    ; write AT record -
10$:    BSBW    RM$RELEASALL                    ; release all bdb's
        POPL    R0                              ; restore status
        BRW     CREOPEN_EXIT                    ; join open finish
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 10 of 16)

EXERCISES

```
        .SUBTITLE RM$CHK_SLIST - Process the searclist loop
;++
;
;       These routines are called when a file access failed and a s
;       is present.  It evaluates whether the error allows the list
;       continue and updates the list to the next searchlist elemen
;
;       RM$CHK_SLIST     - Normal file access failures can continue
;       RM$CHK_SLIST1    - File access failures that create-if can c
;
; Inputs:
;       R0 - failure status of previous access operation.
;
; Outputs:
;       R0 - success/fail
;            (if searchlist is exhausted, status is previous access
;       R1 - undefined if R0 = success
;            if R0 = fail, success if processing may continue, fail
;
; Implicit inputs:
;       R11 - impure ptr
;       R10 - FWA ptr
;       R9  - IFB ptr
;       R8  - FAB ptr
;
; Implicit outputs:
;       FWA and IFB fields modified.
;
; Saved stack:
;       ERR(SP) =>       R0 error code
;       STV(SP) =>       FAB$L_STV(R8)
;       FNB(SP) =>       NAM$L_FNB
;       RSL(SP) =>       NAM$B_RSL
;       ESL(SP) =>       NAM$B_ESL
;--

; Stack offsets for saved context
;
ESL     = 0                        ;       NAM$B_ESL
RSL     = 1                        ;       NAM$B_RSL
FNB     = 4                        ;       NAM$L_FNB
STV     = 8                        ;       FAB$L_STV
ERR     = 12                       ;       R0
STACK_SIZE = 16                    ;       Size of stack to allocate
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 11 of 16)

EXERCISES

```
;  Errors that searchlist processing should continue from
;
RM$SLIST_ERRS::
        RMSERR_WORD     DEV                         ; invalid device na
        RMSERR_WORD     DNF                         ; directory not fou
        RMSERR_WORD     DNR                         ; device not ready
        RMSERR_WORD     FNF                         ; file not found
        RMSERR_WORD     NMF                         ; no more files fou
RM$SLIST_ERR_CNT1 == .-RM$SLIST_ERRS
        RMSERR_WORD     ACC                         ; ACP file access e
        RMSERR_WORD     FND                         ; ACP file lookup e
        RMSERR_WORD     PRV                         ; privilege violati
RM$SLIST_ERR_CNT == .-RM$SLIST_ERRS

RM$CHK_SLIST1::
        PUSHL   S^#<RM$SLIST_ERR_CNT1/2>            ; get number of err
        BRB     CHK_SLIST                           ; and check

RM$CHK_SLIST::
        PUSHL   S^#<RM$SLIST_ERR_CNT/2>             ; get number of err

CHK_SLIST:
        MOVL    (SP),R1                             ; get count
10$:    CMPW    R0,B^RM$SLIST_ERRS-2[R1]            ; continue from thi
        BNEQ    40$                                 ; nope
        BSBB    S_LOOP                              ; try next element
        BLBS    R0,30$                              ; got a good one
        BLBC    R1,CHK_SLIST                        ; get any kind of e
30$:    MOVL    #1,R1                               ; processing can co
        ADDL2   #4,SP                               ; discard count
        RSB                                         ; return

40$:    SOBGTR  R1,10$                              ; try another
        ADDL2   #4,SP                               ; discard count
        RSB                                         ; return don't cont
                                                    ; previous input st

S_LOOP: SSB      #FWA$V_SL_PASS,(R10)               ; flag search list
        PUSHL   R0                                  ; save error status
        PUSHL   FAB$L_STV(R8)                       ; save stv secondar
        CLRQ    -(SP)                               ; room for NAM bloc
        MOVL    FAB$L_NAM(R8),R7                    ; get nam address
        BEQL    22$                                 ; branch if none
        BSBW    RM$CHKNAM                           ; check nam validit
        BLBC    R0,22$                              ; branch if illegal
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 12 of 16)

EXERCISES

```
        MOVL     NAM$L_FNB(R7),FNB(SP)             ; save file name st
        CLRL     NAM$L_FNB(R7)                     ; clear file name s
        MOVB     NAM$B_RSL(R7),RSL(SP)             ; save result strin
        CLRB     NAM$B_RSL(R7)                     ; clear size
        MOVB     NAM$B_ESL(R7),ESL(SP)             ;  and expanded str
        CLRB     NAM$B_ESL(R7)                     ; clear size
        CLRB     NAM$T_DVI(R7)                     ; clear device ID

        ASSUME   NAM$W_DID EQ NAM$W_FID+6

        CLRQ     NAM$W_FID(R7)                     ; and file IDs
        CLRL     NAM$W_DID+2(R7)                   ;

22$:    BBCC     #IFB$V_ACCESSED,(R9),25$          ; deaccess any open
        BSBW     RM$DEACCESS                       ; network links
25$:    $DASSGN_S       CHAN=IFB$W_CHNL(R9)        ; deassign old chan
        CLRW     IFB$W_CHNL(R9)                    ; clear it
        BSBW     RM$PRFLNMALT                      ; try again
        BLBS     R0,30$                            ; try next element
        TSTL     R0                                ; end of list?
        BEQL     40$                               ; no, so return the
30$:    CLRL     R1                                ; found an element
        ADDL2    #STACK_SIZE,SP                    ; discard saved con
        RSB
;
; XPFN exited with RMS$_NOMLIST, no more search list to parse, so r
; the original error code and name block string lengths
;
40$:    MOVL     FAB$L_NAM(R8),R7                  ; get nam address
        BEQL     42$                               ; branch if none
        BSBW     RM$CHKNAM                         ; check nam validit
        BLBC     R0,42$                            ; branch if illegal
        MOVB     ESL(SP),NAM$B_ESL(R7)             ; restore expanded
        MOVB     RSL(SP),NAM$B_RSL(R7)             ;  and result strin
        MOVL     FNB(SP),NAM$L_FNB(R7)             ; restore file name
42$:    ADDL2    #8,SP                             ; discard NAM space
        MOVL     (SP)+,FAB$L_STV(R8)               ; set stv secondary
        MOVL     (SP)+,R0                          ; restore error sta
        MOVL     #1,R1                             ; end-of-list encou
        RSB                                        ; exit
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 13 of 16)

EXERCISES

```
        .SUBTITLE RM$KNOWNFILE - Kernel Mode Known FILE Support
;++
;
;       This routine is called, in kernel mode, when an open known
;       is found.  The following operations are performed:
;
;       1.  Check the volume and file protection to see if the user
;           has access to the file.
;       2.  If the user has read access in addition to execute,
;           report that fact as well.
;       3.  Increment the refcnt on the shared file
;           window and to set the channel appropriately.
;
;       These operations must be interlocked against process deleti
;       by executing at IPL 2.
;
; Inputs:
;       none
;
; Outputs:
;       R0  -   SS$_NORMAL if access allowed
;               SS$_NOPRIV if access denied
;
; Implicit inputs:
;       R11 - impure ptr
;       R10 - FWA ptr
;       R9  - IFB ptr
;       R8  - FAB ptr
;
; Implicit outputs:
;       channel and window control blocks modified.
;--
        ASSUME  CHPCTL$L_ACCESS EQ 0
        ASSUME  CHPCTL$L_FLAGS  EQ 4
        ASSUME  CHPCTL$B_MODE   EQ 8
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 14 of 16)

EXERCISES

```
RM$KNOWNFILE::
        .WORD   ^M<R2,R3,R4,R5>                     ; save registers
        PUSHL   #0                                  ; null access mode
        PUSHL   #<CHPCTL$M_READ!CHPCTL$M_USEREADALL> ; set CHPCTL f
        PUSHL   #ARM$M_READ                         ; set CHPCTL access
        MOVL    SP,R2                               ; point to CHPCTL
        BBC     #IFB$V_WRTACC,(R9),10$              ; write access?
        BISL2   #ARM$M_WRITE,CHPCTL$L_ACCESS(R2);   check for it too
        XORL2   #<CHPCTL$M_WRITE!CHPCTL$M_USEREADALL>,-
                CHPCTL$L_FLAGS(R2)                  ; set WRITE and cle

10$:    SETIPL  #IPL$_ASTDEL                        ; prevent process d
        MOVL    FAB$L_CTX(R8),R0                    ; get KFE
        MOVL    KFE$L_WCB(R0),R5                    ; get WCB
        MOVL    @#CTL$GL_PCB,R4                     ; get PCB addr
        CLRL    R3                                  ; no CHPRET
;
; Check the volume protection in the UCB.
;
        MOVL    WCB$L_ORGUCB(R5),R1                 ; get UCB
        MOVL    UCB$L_ORB(R1),R1                    ; get ORB addr
        MOVL    PCB$L_ARB(R4),R0                    ; get ARB addr
        JSB     G^EXE$CHKPRO_INT                    ; check for volume
        BLBC    R0,100$                             ; give up if no acc
;
;  Now see if the user has requested access  (READ implies EXECUTE)
;
        MOVB    IFB$B_MODE(R9),-
                CHPCTL$B_MODE(R2)                   ; set CHPCTL access
        MOVL    WCB$L_FCB(R5),R1                    ; get FCB
        PUSHL   FCB$L_FILESIZE(R1)                  ; save high block f
        MOVAL   FCB$R_ORB(R1),R1                    ; get ORB addr
        MOVL    PCB$L_ARB(R4),R0                    ; get ARB addr
        JSB     G^EXE$CHKPRO_INT                    ; check for read ac
        BLBC    R0,20$                              ; nope
        BISB2   #FAB$M_GET,FAB$B_FAC(R8)            ; tell user if so
        BRB     30$                                 ; and continue
```

Example 1 Excerpt from RMSOOPEN.MAR (Sheet 15 of 16)

EXERCISES

```
;
;   See if the user has execute-only access
;
20$:    BBC      #FAB$V_EXE,IFB$B_FAC(R9),100$     ; execute access?
        BBS      #IFB$V_WRTACC,(R9),100$           ; no special check
        CMPB     IFB$B_MODE(R9),#PSL$C_SUPER       ; can he ask for ex
        BGTRU    100$                              ; nope
        MOVL     #ARM$M_EXECUTE,-
                 CHPCTL$L_ACCESS(R2)               ; try execute-only
        MOVL     PCB$L_ARB(R4),R0                  ; get ARB addr
        JSB      G^EXE$CHKPRO_INT                  ; check for execute
        BLBC     R0,100$                           ; nope, then return
30$:    MOVZWL   IFB$W_CHNL(R9),R0                 ; get channel numbe
        JSB      G^IOC$VERIFYCHAN                  ; R1 contains CCB a
        MOVL     R5,CCB$L_WIND(R1)                 ; store WCB address
        INCW     WCB$W_REFCNT(R5)                  ; and count this us
        MOVL     (SP),IFB$L_HBK(R9)                ; stuff high block
        MOVL     #1,R0                             ; success!
100$:   SETIPL   #0                                ; restore IPL
        RET

        .END
```

Example 1 Excerpt from RMS0OPEN.MAR (Sheet 16 of 16)

EXERCISES

```
        $BEGIN  RM2OPEN,000,RM$RMS2,<RELATIVE SPECIFIC OPEN>
;
; Facility: RMS32
;
; Abstract:
;       this module provides the organization-specific
;       open processing for relative files.
;
;--
        .SBTTL  DECLARATIONS
;
; Include Files:
;
; Macros:
        $BDBDEF
        $FABDEF
        $IFBDEF
        $PLGDEF
        $RMSDEF
;
; Equated Symbols:
;
; Own Storage:

        .SBTTL  RM$OPEN2 - PROCESS RELATIVE FILE PROLOG
;++
;  RM$OPEN2    -
;
;       this routine performs the file open functions that are
;       specific to the relative file organization, including:
;
;       1  -  verify inter-process record locking not specified
;             since not yet implemented
;       2  -  reading in the prolog and setting the ebk,dvbn,
;             and mrn ifab fields based upon its contents.
;       3  -  setting the mrn fab field.
;
; Calling sequence:
;
;       entered via case branch from RMS$OPEN. returns by
;       jumping to RM$COPRTN.
```

Example 2 Excerpt from RM2OPEN.MAR (Sheet 1 of 4)

EXERCISES

```
; Input Parameters:
;
;       R11       impure area address
;       R9        ifab address
;       R8        fab address
;
; Implicit Inputs:
;
;       the contents of the ifab
;
; Output Parameters:
;
;       R0                status code
;       R10               ifab addr
;       R1-R5,AP          destroyed
;
; Implicit Outputs:
;
;       various fields in the ifab and fab are initialized.
;
; Completion Codes:
;
;       standard rms, in particular suc,plg,shr,rpl, and ver.
;
; Side Effects:
;
;       may wait quite some time for prolog to become
;       free initially. leaves prolog locked.
;--
RM$OPEN2::
        TSTB    IFB$B_BKS(R9)           ; make sure bks nonzero
        BEQL    ERRIFA                  ; if yes, is error
        BITB    #FAB$C_REL,-            ; really relative?
                IFB$B_RFMORG(R9)
        BEQL    EXIT                    ; aha - a bogus seq file p
                                        ; as relative for sharing
;
; if bio access, then prolog read is not required.
```

Example 2 Excerpt from RM2OPEN.MAR (Sheet 2 of 4)

```
        BBS     #IFB$V_BIO,-                ; leave successfully
                IFB$B_FAC(R9),SEXIT
;
; read and process prolog
;
        MOVL    R9,R10                      ; set ifab addr
        MOVZWL  #512,R5                     ; ask for one block to read
        BSBW    RM$ALDBUF                   ; allocate bdb and buffer
        BLBC    R0,EXIT                     ; get out on error
        INCW    IFB$W_AVLCL(R9)             ; count BDB & buffer
        $CACHE  VBN=#1,-                    ; read the prolog
                SIZE=#512,-                 ; (R5=buffer addr)
                FLAGS=LOCK,-
                ERR=ERRRPL
        BSBW    RM$CHKSUM                   ; validate its checksum
        BLBC    R0,EXIT                     ; get out on error
        CMPW    PLG$W_VER_NO(R5),-          ; supported version?
                #PLG$C_VER_NO
        BNEQ    ERRPLV                      ; branch if not
;
; set up ifab values
;
        MOVL    PLG$L_EOF(R5),-             ; copy eof vbn
                IFB$L_EBK(R9)
        MOVZWL  PLG$W_DVBN(R5),-            ; copy vbn of first data bu
                IFB$L_DVBN(R9)
        MOVL    PLG$L_MRN(R5),-             ; copy max. record number
                IFB$L_MRN(R9)
        CLRW    IFB$W_FFB(R9)               ; set blk offset=0
;
; set mrn, gbc in fab
;
SET:    MOVL    IFB$L_MRN(R9),-             ; set mrn
                FAB$L_MRN(R8)
SEXIT:  RMSSUC                              ; show success
EXIT:   JMP     RM$COPRTN                   ; & rejoin common open code
                                            ; note: the bdb will
                                            ; be released there
```

Example 2 Excerpt from RM2OPEN.MAR (Sheet 3 of 4)

EXERCISES

```
;
; handle errors
;
ERRIFA:
        MOVL     #RMS$_BKS,FAB$L_STV(R8) ; set secondary error info
        RMSERR   IFA                     ; illegal file attributes
        BRB      ERRXIT

ERRORG:
        RMSERR   ORG                     ; trying to open a ppf
        BRB      ERRXIT

ERRRPL:
        TSTL     FAB$L_STV(R8)           ; do we have an stv?
        BNEQ     10$                     ; okay use it
        BISL3    #^X1000,R0,FAB$L_STV(R8); else set the RMS error t
10$:    RMSERR   RPL                     ; prolog read error
;
; (stv has ss error code)
;
ERRXIT:          JMP      RM$COPRTN              ; go clean up

ERRPLV:
        RMSERR   PLV                     ; unsupported prolog versi
        BRB      ERRXIT

        .END
```

Example 2 Excerpt from RM2OPEN.MAR (Sheet 4 of 4)

RMS Implementation and Structure

SOLUTIONS

1.

a. The RMS vector SYS$OPEN resides in P1 space.

b. Execution of the CHME instruction in the SYS$OPEN vector causes an exception. The system vectors through the SCB to the executive change mode dispatcher (EXE$CMODEXEC).

The routine named RMS$OPEN gains control in executive mode.

c. The Symbol Cross-Reference section of RMS.MAP indicates that module RMS0OPEN contains the routine RMS$OPEN.

d. RM$OPEN2 will be invoked for this file OPEN.

The file organization stored in the IFAB is 2 (meaning relative), so RMS$OPEN cases to label RM_OPEN2_BR. That label is simply a branch aid to the routine RM$OPEN2.

The CASE macro used here can be found in SYS$LIBRARY:LIB.MLB.

e. Module RM2OPEN contains the routine RM$OPEN2.

2. RMS wants to be sure the information accurately reflects the file being processed, so it copies the FAB information into an IFAB. The user has read and write access to the FAB, but the IFAB is protected against user write.

RMS Implementation and Structure

EXERCISES

1. Analyze the crash dump in OSI$LABS:CRASH1.DMP and gather some information about the RMS internal data structures being used in the JOB_CONTROL process.

   a. Set process context to that of the JOB_CONTROL process.

   b. Display information about the RMS data structures for JOB_CONTROL using the command SHOW PROCESS/RMS.

   c. What is the address of the IFAB for the file with Internal File Identifier (IFI) 01?

   d. What is the address of an IRAB for the file with IFI 01?

RMS Implementation and Structure

SOLUTIONS

1. Issue the DCL command ANALYZE/CRASH OSI$LABS:CRASH1.DMP.

   a. SDA> SET PROCESS JOB_CONTROL

   b. SDA> SHOW PROCESS/RMS

   c. The address of the IFAB for the file with IFI 01 appears in the subheading and is labeled "IFAB address: ."

   d. The address of the IRAB for the first record stream appears in the description of the IFAB, and is labeled "IRAB_LNK: ."

VMS in a Multiprocessing Environment

EXERCISES

1. Complete the table below describing the characteristics of the different multiprocessor implementations.

System Characteristic	VAX-11/782	VAXcluster	Network
CPU booting			Separate
CPU failure			Separate
CPU cabinet location			Can be widely separated
Security/Management domain			Multiple
File system			Separate
Growth potential			Very great

2. What piece of hardware is the high-speed, highly available communications medium for connecting VAXcluster nodes?

3. What VAXcluster software component can provide access from any VAXcluster node, to a disk connected to one VAX (in other words, not connected to an HSC)?

4. What piece of hardware is the central connection point for all nodes in a VAXcluster?

5. What is the difference between an active node and a passive node in a VAXcluster? Give an example of each kind of node.

EXERCISES

6. Why is it useless to run ANALYZE/MEDIA on a disk connected to an HSC-50 or UDA?

7. What is the maximum number of HSC-50s and VAXen that can be connected in a VAXcluster?

8. The HSC-50 is called a "disk server." Explain what this term means.

VMS in a Multiprocessing Environment

SOLUTIONS

1.

System Characteristic	VAX-11/782	VAXcluster	Network
CPU booting	Together	Separate	Separate
CPU failure	Together	Separate	Separate
CPU cabinet location	Single or adjacent	Same computer room	Can be widely separated
Security/Management domain	Single	Single	Multiple
File system	Integrated	Integrated	Separate
Growth potential	Limited	Very great	Very great

2. The Computer Interconnect (CI) provides a high-speed, highly available communications medium for connecting VAXcluster nodes.

3. The MSCP Server can provide access from any VAXcluster node, to a disk connected to one VAX (in other words, not connected to an HSC).

4. The Star Coupler is the central connection point of all nodes in a VAXcluster.

5. An active node, such as a VAX system, actively participates in cluster connection management, and therefore knows the current configuration of the VAXcluster.

   A passive node, such as an HSC, is connected to the cluster, but does not actively participate in cluster connection management.

SOLUTIONS

6. Disks that are connected to an HSC-50 or a UDA are RA-type disks. These disks have automatic revectoring of bad blocks, and therefore provide a contiguous perfect LBN space. ANALYZE/MEDIA, which locates bad blocks, need not be used on RA-type disks.

7. On VMS Version 4.0 the total number of nodes in a VAXcluster, HSC-50s plus VAXen, must be less than or equal to 16.

8. As a disk server, the HSC-50 knows nothing about the ODS-2 file system. It is only concerned about reading and writing logical disk blocks.

VMS in a Multiprocessing Environment

EXERCISES

There are no lab exercises for this module.

VMS in a VAXcluster Environment

EXERCISES

1. For each of the following system processes, specify whether it is created on <u>any</u> VAX system, or only created on a system in a VAXcluster.

   a. ERRFMT

   b. CACHE_SERVER

   c. CLUSTER_SERVER

   d. OPCOM

   e. SWAPPER

2. Briefly describe the functions of the following software components of a VAXcluster.

   a. Distributed Lock Manager

   b. Connection Manager

   c. Distributed File System

VMS in a VAXcluster Environment

SOLUTIONS

1.

a. ERRFMT - created on any VAX

b. CACHE_SERVER - only created on system in a VAXcluster

c. CLUSTER_SERVER - only created on system in a VAXcluster

d. OPCOM - created on any VAX

e. SWAPPER - created on any VAX

NOTE: The above is not necessarily true for a MicroVAX system.

2.

a. The Distributed Lock Manager provides cluster-wide synchronization for many VMS components through the $ENQ and $DEQ system services. It is also available to user applications.

b. The primary function of the Connection Manager is to determine and maintain VAXcluster membership. The connection managers in a VAXcluster work together to synchronize state changes and prevent partitioning.

The connection manager also provides cluster system IDs (CSIDs), and an acknowledged message delivery service.

c. The Distributed File System allows files to be accessed on any system in a VAXcluster as if they were local to that system.

The Files-11 ODS-2 XQP is procedure-based, and mapped into P1 space.

VMS in a VAXcluster Environment

EXERCISES

There are no lab exercises for this module.

Tests

VMS Internals II

PRE-TEST

Circle the best choice for each of the following questions.

1. Which stack is being used when code is running in system context?

   a. User stack
   b. Kernel stack
   c. Interrupt stack
   d. Could be any stack

2. Which of the following is a characteristic of an exception?

   a. Asynchronous to the execution of a process
   b. Changes the IPL to that of the interrupting device
   c. Serviced on the process local stack in process context

3. Which of the following components maintains disk and file structure for Files-11 ODS-2 disks?

   a. Job Controller
   b. XQP
   c. ACP
   d. Pager

4. How is the software timer implemented?

   a. Interrupt service routine
   b. Exception service routine
   c. Process
   d. User-level routine

5. Which interprocess communication technique is used between device drivers and the Error Logger?

   a. Mailbox
   b. Buffer in memory
   c. Common event flags
   d. Locks

6. Symbols for locations in P1 space typically start with a prefix of:

   a. EXE$
   b. MMG$
   c. SCH$
   d. CTL$

7. In which data structure is the hardware PCB located?

   a. JIB
   b. Software PCB
   c. PHD
   d. IRP

8. Which of the following types of information would NOT be found in a process header?

   a. Working set list
   b. P0 and P1 page tables
   c. Process section table
   d. Process scheduling state

9. Into which section of virtual address space is the Files-11 XQP mapped?

   a. P0
   b. P1
   c. S0

10. Forking is used by system routines and drivers to:

    a. Lower IPL
    b. Select which routine to transfer control to next
    c. Respond to exceptions or interrupts
    d. Start up several independent concurrent operations

PRE-TEST

11. AST control blocks that are waiting to be delivered to a process are queued to which of the following data structures?

    a. JIB
    b. Software PCB
    c. PHD
    d. IRP

12. Which of the following operations is NOT performed by the hardware clock interrupt service routine?

    a. Update quantum information for current process
    b. Check for device timeouts
    c. Update system time
    d. Check timer queue to see if software timer interrupt service routine needs to be invoked

13. What mechanism is used to go from a more privileged access mode to a less privileged access mode?

    a. CHMx instruction
    b. REI instruction
    c. Change mode system service
    d. MOVPSL instruction

14. Which data structure contains the information that tells the system how to respond to every possible exception or interrupt?

    a. System Header
    b. SCB
    c. RPB
    d. PCB

15. What IPL value is used to synchronize access to the scheduler's database?

    a. IPL$_ASTDEL
    b. IPL$_SCHED
    c. IPL$_SYNCH
    d. IPL$_POWER

PRE-TEST

16. Why does the SRVEXIT section of the change mode dispatcher issue an REI instruction?

    a. To match the CHMx instruction in the SYS$service code
    b. To match the CALLx instruction to the system service in the user's program
    c. To match the CASE instruction in the change mode dispatcher
    d. To signal that an error has occurred in the execution of the system service

17. What does G mean in SDA or XDELTA?

    a. Go to a particular location
    b. Add 80000000 hex
    c. Repeat the previous command
    d. Display a global symbol

18. Which SDA command(s) can be used to duplicate the information found on a console bugcheck output?

    a. SHOW CRASH and SHOW STACK
    b. SHOW SUMMARY
    c. SHOW CRASH and SHOW SUMMARY
    d. SHOW PFN and SHOW STACK

19. What information is usually placed in the first two longwords of system data structures?

    a. Forward and backward links
    b. Structure size and type
    c. ASCII representation of structure name
    d. Access mode and IPL at which synchronization should occur on the data structure

20. Which of the following tools would you use to change the contents of a field in a data structure?

    a. SDA
    b. INSTALL
    c. MONITOR
    d. XDELTA

PRE-TEST

21. Which of the following tools would be the easiest to use to examine the value of the symbolic location EXE$GL_SITESPEC?

    a. SDA
    b. INSTALL
    c. MONITOR
    d. XDELTA

22. Process FRED has a base priority of 4. It is currently executing at a priority of 8. An event occurs that has an associated priority boost of 2. What will be the resulting priority of Process FRED?

    a. 4
    b. 6
    c. 8
    d. 10

23. Process LIZ and Process MO both want to access the same RTL routine. Process LIZ references the routine first, and causes a page fault. Then Process MO references the page which has not yet been read into memory (i.e., faults the same page). What scheduling state will Process LIZ and Process MO be in?

    a. Process LIZ = PFW, Process MO = PFW
    b. Process LIZ = PFW, Process MO = COLPG
    c. Process LIZ = COLPG, Process MO = COLPG
    d. Process LIZ = COLPG, Process MO = PFW

24. Which component is responsible for moving processes into and out of wait states?

    a. RSE
    b. Scheduler
    c. Swapper
    d. Pager

25. Which component is responsible for building most of the virtual address space of a new process?

    a. $CREPRC system service
    b. Swapper process
    c. PROCSTRT routine
    d. A user routine

26. Which component is responsible for allocating a PCB for a new process?

    a. $CREPRC system service
    b. Swapper process
    c. PROCSTRT routine
    d. A user routine

27. From which state is a process deleted?

    a. CUR
    b. COMO
    c. PFW
    d. MWAIT

28. Which of the following pieces of information can the extended PID contain?

    a. Process index into PCB and sequence vectors
    b. Process sequence number
    c. Cluster node index
    d. Node sequence number
    e. All of the above

29. In the system initialization sequence, which component is responsible for sizing S0 space and setting up the system page table?

    a. VMB
    b. SYSBOOT
    c. INIT
    d. CONSOLE.SYS

PRE-TEST

30. During the execution of which software component is VMS memory management enabled during system initialization?

    a. VMB
    b. SYSBOOT
    c. INIT
    d. CONSOLE.SYS

VMS Internals II

SOLUTIONS TO PRE-TEST

1. Which stack is being used when code is running in system context?

   a. User stack
   b. Kernel stack
   c. Interrupt stack
   d. Could be any stack

2. Which of the following is a characteristic of an exception?

   a. Asynchronous to the execution of a process
   b. Changes the IPL to that of the interrupting device
   c. Serviced on the process local stack in process context

3. Which of the following components maintains disk and file structure for Files-11 ODS-2 disks?

   a. Job Controller
   b. XQP
   c. ACP
   d. Pager

4. How is the software timer implemented?

   a. Interrupt service routine
   b. Exception service routine
   c. Process
   d. User-level routine

5. Which interprocess communication technique is used between device drivers and the Error Logger?

   a. Mailbox
   b. Buffer in memory
   c. Common event flags
   d. Locks

VMS Internals II

SOLUTIONS TO PRE-TEST

6. Symbols for locations in P1 space typically start with a prefix of:

   a. EXE$
   b. MMG$
   c. SCH$
   d. CTL$

7. In which data structure is the hardware PCB located?

   a. JIB
   b. Software PCB
   c. PHD
   d. IRP

8. Which of the following types of information would NOT be found in a process header?

   a. Working set list
   b. P0 and P1 page tables
   c. Process section table
   d. Process scheduling state

9. Into which section of virtual address space is the Files-11 XQP mapped?

   a. P0
   b. P1
   c. S0

10. Forking is used by system routines and drivers to:

    a. Lower IPL
    b. Select which routine to transfer control to next
    c. Respond to exceptions or interrupts
    d. Start up several independent concurrent operations

SOLUTIONS TO PRE-TEST

11. AST control blocks that are waiting to be delivered to a process are queued to which of the following data structures?

    a. JIB
    b. Software PCB
    c. PHD
    d. IRP

12. Which of the following operations is NOT performed by the hardware clock interrupt service routine?

    a. Update quantum information for current process
    b. Check for device timeouts
    c. Update system time
    d. Check timer queue to see if software timer interrupt service routine needs to be invoked

13. What mechanism is used to go from a more privileged access mode to a less privileged access mode?

    a. CHMx instruction
    b. REI instruction
    c. Change mode system service
    d. MOVPSL instruction

14. Which data structure contains the information that tells the system how to respond to every possible exception or interrupt?

    a. System Header
    b. SCB
    c. RPB
    d. PCB

15. What IPL value is used to synchronize access to the scheduler's database?

    a. IPL$_ASTDEL
    b. IPL$_SCHED
    c. IPL$_SYNCH
    d. IPL$_POWER

SOLUTIONS TO PRE-TEST

16. Why does the SRVEXIT section of the change mode dispatcher issue an REI instruction?

    a. To match the CHMx instruction in the SYS$service code
    b. To match the CALLx instruction to the system service in the user's program
    c. To match the CASE instruction in the change mode dispatcher
    d. To signal that an error has occurred in the execution of the system service

17. What does G mean in SDA or XDELTA?

    a. Go to a particular location
    b. Add 80000000 hex
    c. Repeat the previous command
    d. Display a global symbol

18. Which SDA command(s) can be used to duplicate the information found on a console bugcheck output?

    a. SHOW CRASH and SHOW STACK
    b. SHOW SUMMARY
    c. SHOW CRASH and SHOW SUMMARY
    d. SHOW PFN and SHOW STACK

19. What information is usually placed in the first two longwords of system data structures?

    a. Forward and backward links
    b. Structure size and type
    c. ASCII representation of structure name
    d. Access mode and IPL at which synchronization should occur on the data structure

20. Which of the following tools would you use to change the contents of a field in a data structure?

    a. SDA
    b. INSTALL
    c. MONITOR
    d. XDELTA

SOLUTIONS TO PRE-TEST

21. Which of the following tools would be the easiest to use to examine the value of the symbolic location EXE$GL_SITESPEC?

    a. SDA
    b. INSTALL
    c. MONITOR
    d. XDELTA

22. Process FRED has a base priority of 4. It is currently executing at a priority of 8. An event occurs that has an associated priority boost of 2. What will be the resulting priority of Process FRED?

    a. 4
    b. 6
    c. 8
    d. 10

23. Process LIZ and Process MO both want to access the same RTL routine. Process LIZ references the routine first, and causes a page fault. Then Process MO references the page which has not yet been read into memory (i.e., faults the same page). What scheduling state will Process LIZ and Process MO be in?

    a. Process LIZ = PFW, Process MO = PFW
    b. Process LIZ = PFW, Process MO = COLPG
    c. Process LIZ = COLPG, Process MO = COLPG
    d. Process LIZ = COLPG, Process MO = PFW

24. Which component is responsible for moving processes into and out of wait states?

    a. RSE
    b. Scheduler
    c. Swapper
    d. Pager

SOLUTIONS TO PRE-TEST

25. Which component is responsible for building most of the virtual address space of a new process?

    a. $CREPRC system service
    b. Swapper process
    c. PROCSTRT routine
    d. A user routine

26. Which component is responsible for allocating a PCB for a new process?

    a. $CREPRC system service
    b. Swapper process
    c. PROCSTRT routine
    d. A user routine

27. From which state is a process deleted?

    a. CUR
    b. COMO
    c. PFW
    d. MWAIT

28. Which of the following pieces of information can the extended PID contain?

    a. Process index into PCB and sequence vectors
    b. Process sequence number
    c. Cluster node index
    d. Node sequence number
    e. All of the above

29. In the system initialization sequence, which component is responsible for sizing S0 space and setting up the system page table?

    a. VMB
    b. SYSBOOT
    c. INIT
    d. CONSOLE.SYS

SOLUTIONS TO PRE-TEST

30. During the execution of which software component is VMS memory management enabled during system initialization?

    a. VMB
    b. SYSBOOT
    c. INIT
    d. CONSOLE.SYS

VMS Internals II

POST-TEST

Circle the best choice for each of the following questions.

1. All of the following are functions of the Job Controller EXCEPT

   a. Manages batch queue and batch jobs
   b. Has a part in creating interactive processes associated with terminals
   c. Accounting manager
   d. Issues $QIOs to print files

2. Which interprocess communication technique is used between the Job Controller and symbionts?

   a. Global sections
   b. Mailboxes
   c. Common event flags
   d. Locks

3. How much does a VMS print symbiont understand about print queues?

   a. Nothing
   b. Knows how many print queues there are on the system
   c. Understands the structure of JBCSYSQUE.DAT

4. Which software component is responsible for logging/reporting device errors?

   a. RSE
   b. ERRFMT
   c. Device drivers
   d. Oscar the Grouch

5. A command language interpreter (CLI) is essentially

   a. A condition handler
   b. An exit handler
   c. An interrupt service routine
   d. A procedure called by the operating system

POST-TEST

6. All of the following are valid types of image section descriptors EXCEPT

   a. Demand zero section
   b. Process private section
   c. Global section
   d. Paging file section

7. What information in the image file tells the image activator where to map each portion of an image into process virtual address space?

   a. Base VPN in image section descriptor
   b. Base VBN in image section descriptor
   c. Size of image section descriptor
   d. VBN of block on disk

8. Which data structure entry in the PHD tells the pager where to find a process private page on disk?

   a. Page table entry
   b. Process section table entry
   c. Working set list entry

9. Which element of the PFN database indicates where a page should be placed if it has to leave physical memory?

   a. PTE
   b. STATE
   c. BAK
   d. TYPE

10. A page table entry can contain each of the following EXCEPT

   a. Physical page frame number (PFN)
   b. Process section table index (PSTX)
   c. Page file virtual block number
   d. Swap file virtual block number

POST-TEST

11. Which SYSGEN parameter limits the size of the P0 and P1 page tables?

    a. NPAGEDYN
    b. WSMAX
    c. VIRTUALPAGECNT
    d. PROCSECTCNT

12. Where is the protection code for a page of physical memory stored?

    a. In the first four bits of the physical page
    b. In the PFN database
    c. In the PTE(s) mapping the page
    d. In a file on the system disk

13. When a page is removed from a process working set, it does not leave memory right away. If that page was written to, it will go to the

    a. Free page list
    b. Modified page list
    c. Bad page list
    d. Home for wayward pages

14. The swapper must be involved in all of the following system activities EXCEPT

    a. Modified page writing
    b. Process creation
    c. System initialization
    d. Process scheduling

15. The swapper is able to issue a single $QIO to read/write entire process working sets because it makes the pages appear virtually contiguous using:

    a. An intermediate buffer in system space
    b. Modifications to the process's page table entries
    c. Its own P0 page table
    d. The SWPVBN elements in the PFN database

VMS Internals II

POST-TEST

16. In its attempts to regain free pages, the swapper will do all of the following EXCEPT

    a. Write modified pages
    b. Delete processes of low priority
    c. Shrink working sets
    d. Outswap processes

17. Which of the following components consists of the most primitive I/O routines that have a user interface?

    a. RMS
    b. I/O system services
    c. FDT routines
    d. Device drivers

18. Which of the following components processes the device-dependent parameters on a call to $QIO?

    a. RMS
    b. I/O system services
    c. FDT routines
    d. Device drivers

19. Which of the following data structures contains information common to all devices on a controller?

    a. UCB
    b. CCB
    c. DDB
    d. IRP

20. Which of the following data structures contains information for a device unit?

    a. UCB
    b. CCB
    c. DDB
    d. IRP

POST-TEST

21. Process-specific RMS internal data structures are stored in which area of P1 space?

    a. Image I/O segment
    b. Process I/O segment
    c. P1 window to the PHD
    d. Per-process common area

22. What software component is used just before you enter the RMS specific procedure?

    a. EXE$CMODEXEC
    b. RMS$DISPATCH
    c. RMS synchronization routine

23. What data structures used by RMS are stored in P1 space?

    a. FAB and RAB
    b. DDB and UCB
    c. IFAB and IRAB
    d. MIC and KEY

24. Which of the following is a characteristic of a tightly coupled VAX-11/782 configuration?

    a. The file systems are separate
    b. It is a multiple management domain
    c. The CPUs boot and fail together
    d. The CPU cabinets can be widely separated

25. Which piece of hardware is the high-speed, highly available communications medium for connecting VAXcluster nodes?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

POST-TEST

26. What VAXcluster software component can provide access from any VAXcluster node, to a non-HSC disk connected to just one VAX?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

27. What piece of hardware is the central connection point for all nodes in a VAXcluster?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

28. Which of the following processes is present on a system in a VAXcluster, and NOT on a single, non-clustered VAX?

    a. SWAPPER
    b. OPCOM
    c. JOB_CONTROL
    d. CACHE_SERVER

29. Which of the following VAXcluster components provides cluster-wide synchronization for many VMS components?

    a. Distrubuted lock manager
    b. Connection manager
    c. Distrubuted file system
    d. Systems communications services (SCS)

30. Which of the following VAXcluster components determines and maintains VAXcluster membership?

    a. Distrubuted lock manager
    b. Connection manager
    c. Distrubuted file system
    d. Systems communications services (SCS)

VMS Internals II

SOLUTIONS TO POST-TEST

1. All of the following are functions of the Job Controller EXCEPT

   a. Manages batch queue and batch jobs
   b. Has a part in creating interactive processes associated with terminals
   c. Accounting manager
   d. Issues $QIOs to print files

2. Which interprocess communication technique is used between the Job Controller and symbionts?

   a. Global sections
   b. Mailboxes
   c. Common event flags
   d. Locks

3. How much does a VMS print symbiont understand about print queues?

   a. Nothing
   b. Knows how many print queues there are on the system
   c. Understands the structure of JBCSYSQUE.DAT

4. Which software component is responsible for logging/reporting device errors?

   a. RSE
   b. ERRFMT
   c. Device drivers
   d. Oscar the Grouch

5. A command language interpreter (CLI) is essentially

   a. A condition handler
   b. An exit handler
   c. An interrupt service routine
   d. A procedure called by the operating system

SOLUTIONS TO POST-TEST

6. All of the following are valid types of image section descriptors EXCEPT

   a. Demand zero section
   b. Process private section
   c. Global section
   d. Paging file section

7. What information in the image file tells the image activator where to map each portion of an image into process virtual address space?

   a. Base VPN in image section descriptor
   b. Base VBN in image section descriptor
   c. Size of image section descriptor
   d. VBN of block on disk

8. Which data structure entry in the PHD tells the pager where to find a process private page on disk?

   a. Page table entry
   b. Process section table entry
   c. Working set list entry

9. Which element of the PFN database indicates where a page should be placed if it has to leave physical memory?

   a. PTE
   b. STATE
   c. BAK
   d. TYPE

10. A page table entry can contain each of the following EXCEPT

    a. Physical page frame number (PFN)
    b. Process section table index (PSTX)
    c. Page file virtual block number
    d. Swap file virtual block number

SOLUTIONS TO PRE-TEST

11. Which SYSGEN parameter limits the size of the P0 and P1 page tables?

    a. NPAGEDYN
    b. WSMAX
    c. VIRTUALPAGECNT
    d. PROCSECTCNT

12. Where is the protection code for a page of physical memory stored?

    a. In the first four bits of the physical page
    b. In the PFN database
    c. In the PTE(s) mapping the page
    d. In a file on the system disk

13. When a page is removed from a process working set, it does not leave memory right away. If that page was written to, it will go to the

    a. Free page list
    b. Modified page list
    c. Bad page list
    d. Home for wayward pages

14. The swapper must be involved in all of the following system activities EXCEPT

    a. Modified page writing
    b. Process creation
    c. System initialization
    d. Process scheduling

15. The swapper is able to issue a single $QIO to read/write entire process working sets because it makes the pages appear virtually contiguous using:

    a. An intermediate buffer in system space
    b. Modifications to the process's page table entries
    c. Its own P0 page table
    d. The SWPVBN elements in the PFN database

SOLUTIONS TO POST-TEST

16. In its attempts to regain free pages, the swapper will do all of the following EXCEPT

    a. Write modified pages
    b. Delete processes of low priority
    c. Shrink working sets
    d. Outswap processes

17. Which of the following components consists of the most primitive I/O routines that have a user interface?

    a. RMS
    b. I/O system services
    c. FDT routines
    d. Device drivers

18. Which of the following components processes the device-dependent parameters on a call to $QIO?

    a. RMS
    b. I/O system services
    c. FDT routines
    d. Device drivers

19. Which of the following data structures contains information common to all devices on a controller?

    a. UCB
    b. CCB
    c. DDB
    d. IRP

20. Which of the following data structures contains information for a device unit?

    a. UCB
    b. CCB
    c. DDB
    d. IRP

SOLUTIONS TO POST-TEST

21. Process-specific RMS internal data structures are stored in which area of P1 space?

    a. Image I/O segment
    b. Process I/O segment
    c. P1 window to the PHD
    d. Per-process common area

22. What software component is used just before you enter the RMS specific procedure?

    a. EXE$CMODEXEC
    b. RMS$DISPATCH
    c. RMS synchronization routine

23. What data structures used by RMS are stored in P1 space?

    a. FAB and RAB
    b. DDB and UCB
    c. IFAB and IRAB
    d. MIC and KEY

24. Which of the following is a characteristic of a tightly coupled VAX-11/782 configuration?

    a. The file systems are separate
    b. It is a multiple management domain
    c. The CPUs boot and fail together
    d. The CPU cabinets can be widely separated

25. Which piece of hardware is the high-speed, highly available communications medium for connecting VAXcluster nodes?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

SOLUTIONS TO POST-TEST

26. What VAXcluster software component can provide access from any VAXcluster node, to a non-HSC disk connected to just one VAX?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

27. What piece of hardware is the central connection point for all nodes in a VAXcluster?

    a. Computer interconnect (CI)
    b. MSCP server
    c. Star coupler
    d. HSC-50

28. Which of the following processes is present on a system in a VAXcluster, and NOT on a single, non-clustered VAX?

    a. SWAPPER
    b. OPCOM
    c. JOB_CONTROL
    d. CACHE_SERVER

29. Which of the following VAXcluster components provides cluster-wide synchronization for many VMS components?

    a. Distrubuted lock manager
    b. Connection manager
    c. Distrubuted file system
    d. Systems communications services (SCS)

30. Which of the following VAXcluster components determines and maintains VAXcluster membership?

    a. Distrubuted lock manager
    b. Connection manager
    c. Distrubuted file system
    d. Systems communications services (SCS)