International Technical Support Organization

SG24-4640-00

The OS/2 Debugging Handbook - Volume I
Basic Skills and Diagnostic Techniques

February 1996

IBM

International Technical Support Organization
Boca Raton Center

International Technical Support Organization

SG24-4640-00

The OS/2 Debugging Handbook - Volume I
Basic Skills and Diagnostic Techniques

February 1996

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xiii.

First Edition (February 1996)

This edition applies to IBM OS/2 Warp Version 3.0.

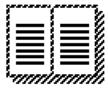
Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader′s feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. JLPC Building 014-1 Internal Zip 5220
1000 NW 51st Street
Boca Raton, Florida 33431-1328

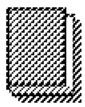
The OS/2 Debugging Handbook Library

The following information describes the four volumes that comprise the OS/2 Debugging Handbook library. The graphic of the opened book denotes the volume that you are currently reading.



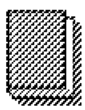
Volume I, *Basic Skills and Diagnostic Techniques, SG24-4640.*

This volume introduces the concepts of debugging with practical examples. Also contained in this book is a CDROM version of the entire library, which is viewable using the OS/2 INF View utility.



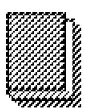
Volume II, *Using the Debug Kernel and Dump Formatter, SG24-4641.*

This volume provides necessary information to set up and use the Kernel Debug and Dump Formatter tools. Also this guide serves as a command reference for these products.



Volume III, *System Trace Reference, SG24-4642.*

This volume includes all system tracepoints contained within OS/2.



Volume IV, *System Diagnostic Reference, SG24-4643.*

This volume provides details of internal structures used by OS/2.

Abstract

This publication is one of four volumes which together provide information and reference materials intended to help perform OS/2 debugging.

This volume provides an introduction to OS/2 debugging with a section of worked examples. The later worked examples provide an aid to the understanding of the debug process.

This document is intended for use by service personnel, system programmers and software developers.

Note that this book has been used in conjunction with a hands-on OS/2 debugging class run by IBM.

(361 pages)

Contents

Figures

Special Notices

This publication is intended to help service personnel, system programmers and software developers to understand the concepts and application of debugging techniques. The information in this publication is intended as a supplement to already published specifications of any programming interfaces that are provided by IBM Warp OS/2 Version 3. See the PUBLICATIONS section of the IBM Programming Announcement for IBM Warp OS/2 Version 3 for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM	OS/2
Presentation Manager	Workplace Shell

The following terms are trademarks of other companies:

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Windows is a trademark of Microsoft Corporation.

MicroFocus Cobol	MicroFocus Corporation

Other trademarks are trademarks of their respective companies.

Preface

Debugging problems is essentially an iterative process of hypothesis, test and conclusion that aims to eliminate the irrelevant and therefore focus on the probable causal area.

To engage this process successfully one needs to be equipped with an innate ability to think laterally coupled with sufficient knowledge of the environment in which the problem persists and above all else to be able to use the tools that extract information from the system under diagnosis.

This scenario applies as much to first level problem determination (PD) as it does to the software developer who is engaged in detailed analysis of his programs′ behavior.

Information and tools to aid first level problem determination is relatively accessible. Technical literature is available from IBM and book stores that will fulfill the needs of the first level PD analyst. For example, the reader is invited to consult the following IBM redbook publications to achieve an all-round high-level technical appreciation of the OS/2 environment:

The Technical Compendium Volume 1 - Control Program

The Technical Compendium Volume 2 - DOS and Windows Environment

The Technical Compendium Volume 3 - Presentation Manager and Workplace Shell

The Technical Compendium Volume 4 - Application Development

The Technical Compendium Volume 5 - The Print Sub-system

The problem analysis level that is less well provided for is that which involves internal knowledge of the OS/2 operating system and its diagnostic tools. This is the level at which service personnel, system programmers and software developers work. It is this audience to which the OS/2 Debugging Handbooks are directed.

An inevitable consequence of working at a deep technical level is that the amount of information one could amass is vast. Given time constraints and the need to publish useable material before it became obsolete we had to make certain compromises for the first edition. The following principles guided us in making decisions about which material to include:

Material that is adequately documented elsewhere is referenced, but not included.

Accurate reference documentation for the diagnostic tools and facilities available for OS/2 has been given priority over worked examples and OS/2 Internals reference material.

Internals information has centered around the base operating system, that is, the kernel.

It is hoped to remedy some of these short-comings in future revisions of this book and in companion volumes.

The current printed edition contains full reference material for the following OS/2 System diagnostic facilities:

System Trace

System Dump

Kernel Debugger

In addition to these topics, included is an introductory guide to problem determination. This provides a resumé of the hardware and software environment and an introduction to using the dump formatter and kernel debugger.

Throughout this book it is assumed the availability and familiarity with two co-requisite publications:

The Intel Pentium Family User′s Manual, Volume 3: Architecture and programming manual, ISBN 1-55512-227-2, Intel order number 241430-003.

This should be consulted as the authoritative source for hardware architectural information.

The Design of OS/2 by H.M. Deitel and M.S. Kogan.

This should be consulted for an overview of the internal operation and architecture of OS/2.

This book is supplied with a CD-ROM whose contents are:

- Sample exercises to accompany Chapter 1, "Approach to Problem Solving" on page 1. These take the form of system dumps of typical problems in application programs.
- Online version of this book. This is slightly more advanced than the printed version and includes more worked examples. This is an .INF file and should be viewed using the OS/2 VIEW.EXE program. Much use has been made of hypertext links, which direct the user to the glossary. From the glossary it is possible to link to related material in other sections of the book.
- The OS/2 Problem Determination Package (OS2PDP), which includes the dump formatter, symbol files and trace customizer (TRCUST).

Unless otherwise stated the material in this book may be assumed to be applicable to OS/2 Warp version 3.0 (ALLSTRICT Kernel).

As indicated above, work on this subject matter can never be complete. It is intended to build on and update the material in this edition. In order to address the areas in most need of attention you the reader are invited to fill in the Reader′s Comment Form with your suggestions.

How This Document is Organized

The document is organized as follows:

- Chapter 1, "Approach to Problem Solving"

  This chapter provides an introduction to debugging and an approach to problem solving.

- Chapter 2, "The Address Space Picture"

  This chapter describes the address space picture of OS/2.

- Chapter 3, "OS/2 Implementation Details"

  This section details the memory implementation and information about thunking.

- Chapter 4, "Stacks"

  This chapter describes how most OS/2 applications use the stack.

- Chapter 5, "Application Documentation"

  Documentation that the compiler can optionally generate is described in this chapter.

- Chapter 6, "Steps to Diagnose a Trap"

  A brief introduction to diagnosing a trap.

- Chapter 7, "The OS/2 System Trace"

  This chapter talks about the trace facility that OS/2 has and how it is enabled via the CONFIG.SYS file.

- Chapter 8, "TRCUST, the Dynamic Trace Customizer"

  Shows how OS/2 provides a mechanism by which developers may dynamically apply trace points in their application modules.

- Chapter 9, "The Layout of a Trace Source File"

  This Chapter describes the source code file in detail.

- Chapter 10, "Steps to Diagnose a Hang"

  A brief introduction to diagnosing a hang.

- Chapter 11, "Serialization and Priorities in OS/2"

  This section describes the various ways to serialize access to resources.

- Chapter 12, "A Form to Use for Unwinding Stacks"

  A form for documenting the unwinding of stacks.

- Chapter 13, "Advanced Guide to Hang Analysis"

  This section describes the various features of a hang.

- Chapter 14, "Worked Examples"

  This chapter contains worked examples to illustrate the use of some of the debugging tools.

- Appendix A, "Minimal Command Reference"

  This appendix contains a small command reference.

Related Publications

Throughout this book we assume the availability and familiarity with two co-requisite publications:

- *The INTEL486 Microprocessor Programmer's Reference Manual*, ISBN 1-55512-159-4
- *The Intel Pentium Family User's Manual, Volume 3: Architecture and Programming Manual*, ISBN 1-55512-227-2
- *The Design of OS/2 by H.M. Deitel and M.S. Kogan*, ISBN 0-201-54889-5

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *The OS/2 Technical Library Control Program Programming Reference Version 2.00*, S10G-6263-00
- *OS/2 2.0 Proc Lang 2/REXX Ref*, S10G-6268-00
- *OS/2 2.0 Proc Lang 2/REXX User Guide*, S10G-6269-00
- *OS/2 WARP Control Program Programming Guide*, G25H-7101-00
- *OS/2 WARP Control Program Programming Ref*, G25H-7102-00
- *OS/2 WARP PM Basic Programming Guide*, G25H-7103-00
- *OS/2 WARP PM Advanced Programming Guide*, G25H-7104-00
- *OS/2 WARP GPI Programming Guide*, G25H-7106-00
- *OS/2 WARP GPI Programming Ref*, G25H-7107-00
- *OS/2 WARP Workplace Shell Programming Guide*, G25H-7108-00
- *OS/2 WARP Workplace Shell Programming Ref*, G25H-7109-00
- *OS/2 WARP IPF Programming Guide*, G25H-7110-00
- *OS/2 WARP Tools Reference*, G25H-7111-00
- *OS/2 WARP Multimedia App Programming Guide*, G25H-7112-00
- *OS/2 WARP Multimedia Subsystem Programming*, G25H-7113-00
- *OS/2 WARP Multimedia Programming Ref*, G25H-7114-00
- *OS/2 WARP PM Programming Ref Vol I*, G25H-7190-00
- *OS/2 WARP PM Programming Ref Vol II*, G25H-7191-00
- *Technical Reference - Personal Computer AT*, Part Number 1502494
- *PS/2 and PC BIOS Interface Technical Reference*, Part Number 68X2341

International Technical Support Organization Publications

- *OS/2 Warp Connect*, GG24-4505
- *OS/2 Warp Generation, Vol.1*, SG24-4552
- *OS/2 Warp Version 3 and BonusPak*, GG24-4426
- *Multimedia in Warp*, GG24-2516
- *The Technical Compendium Volume 1 - Control Program*, GG24-3730

- *The Technical Compendium Volume 2 - DOS and Windows Environment*, GG24-3731
- *The Technical Compendium Volume 3 - Presentation Manager and Workplace Shell*, GG24-3732
- *The Technical Compendium Volume 4 - Application Development*, GG24-3774

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, may be found in:

> *International Technical Support Organization Bibliography of Redbooks,* GG24-3070.

To get a catalog of ITSO redbooks, VNET users may type:

```
TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
```

A listing of all redbooks, sorted by category, may also be found on MKTTOOLS as ITSOCAT TXT. This package is updated monthly.

How to Order ITSO Redbooks

IBM employees in the USA may order ITSO books and CD-ROMs using PUBORDER. Customers in the USA may order by calling 1-800-879-2755 or by faxing 1-800-445-9269. Most major credit cards are accepted. Outside the USA, customers should contact their local IBM office. Guidance may be obtained by sending a PROFS note to BOOKSHOP at DKIBMVM1 or E-mail to bookshop@dk.ibm.com.

Customers may order hardcopy ITSO books individually or in customized sets, called GBOFs, which relate to specific functions of interest. IBM employees and customers may also order ITSO books in online format on CD-ROM collections, which contain redbooks on a variety of products.

ITSO Redbooks on the World Wide Web (WWW)

Internet users may find information about redbooks on the ITSO World Wide Web home page. To access the ITSO Web pages, point your Web browser (such as WebExplorer from the OS/2 3.0 Warp BonusPak) to the following URL:

```
http://www.redbooks.ibm.com/redbooks
```

IBM employees may access LIST3820s of redbooks as well. Point your web browser to the IBM Redbooks home page:

```
http://w3.itsc.pok.ibm.com/redbooks/redbooks.html
```

Acknowledgments

The authors of this book are:

Pete Guy
IBM SDO, Austin

Richard Moore
IBM PSP EMEA

Redbook project developed by:

Tim Sennitt
ITSO Boca Raton, Center

This book could not have reached publication without the encouragement, help and support from a number of colleagues and friends. In particular we would like to thank the following:

Tim Sennitt for his help in preparing the printed material and doing much of the donkey-work to bring this to publication.

Joanne Rearnkham, Barry Bryan and David Jaramillo for their support in enabling access to the materials necessary to produce this book.

Chris Perritt and Glen Brew for making available the original Design Workbook and Functional Specifications for OS/2 2.0.

Charlie Schmitt for his original work on converting the kernel debugger code into a dump formatter.

Jeff Mielke and David Jaramillo for their work on PMDF, the structure compiler and continued work on the dump formatter.

Allen Gilbert for making available documentation on System Trace, which has been reproduced in an edited form in this book. Also, for making available an early version of the dump formatter without which it would not have been possible to develop the original Dump Formatter class.

Doug Azzarito for supplying the material on Kernel Debugger Remote Debug Setup.

James Taylor for providing the basis of the lab exercises relating to PM hangs.

Marie Jazynka, one of the first OS/2 debuggers, for patient encouragement of a great many OS/2 debugging people.

Our management teams, without whose foresight and support none of this work would ever have started. These include:

- Hermann Lamberti General Manager for PSM EMEA; Gordon Bell - director PSM EMEA Technical Marketing; Chris Brown - manager PSM OEM and Enterprise Technical Marketing and Brian Rose - manager PSM Project Office; Roy Aho - Director of the Solution Developer Technical Support Center, for encouraging the beginnings of this several years ago; Terry Gray, manager of Platform Competency and Operation, within Solution Developer Technical Support, Austin.

Finally to Sarah-Jane and Shelly, for supporting many very extended working days and weeks.

Chapter 1. Approach to Problem Solving

In order to succeed at low-level program problem diagnosis, one must have several skills. None of these is particularly difficult, but many are foreign to today′s programmers.

At first, it will appear that each problem is solved with a different technique. Study of the methods used to solve problems yields the fact that the several skills are used as appropriate, virtually as subroutines, and without thought, by experienced analysts.

The intent of this material is to provide the basic knowledge and to illustrate each of the skills separately, to aid understanding. Trying to solve problems without the basic skills can be extremely frustrating, at best.

The fundamentals include knowledge of hardware operation, software conventions, and basic use of tools to display the data sought. Once the fundamentals are understood, it is time to begin using them to solve problems, because one can then concentrate on building the problem solving skill.

Application traps are perhaps the easiest problems to approach, so they are explained after the basic skills. Similarly, traps in privileged code are only incrementally more difficult.

Once some experience in solving traps has been gained, it is reasonable to extend one′s skills by exploring reasons for waits and loops, collectively known as hangs, or to learn the additional functions provided by Symmetric Multiprocessor (SMP) systems, as well as the challenges in properly serializing them when needed.

1.1 List of Necessary Skills

The following are fundamental skills needed:

- A good knowledge of how the hardware protection mechanisms work.
- A good knowledge of what any instruction actually does.
- A good knowledge of a few primary software conventions:
  - How a stack is used and what information is in it.
  - How to use the stack data for debugging.
- How to use optional program documentation to get from a failing instruction to the actual line of the program which contains it.
- How to find the program′s variables in storage.
- How to obtain the above documentation for some IBM languages.
- How to collect a dump of a system at the point of failure.
- How to use the available analysis tools.
- How to determine the owner of a part of storage, and which processes have access to that storage.

And that′s what this material is designed to teach!

1.2 Collecting Documentation

If the problem can be reliably reproduced in a development environment, do it. This is the fastest way to get the problem fixed. When you cannot, try to get a good set of starting documentation.

It is possible to acquire and install a replacement for the OS/2 kernel which is the same as the one being replaced, except that it has debugging facilities and a debug interface to a serial port, COM2. If you install the wrong debug kernel, no one can predict the results. If you install the correct version, you will need to have a terminal emulation program (or ASCII terminal) to access the debug interface. The capabilities of this debug tool are essentially unlimited, and there is no protection from accidental entry errors. Its use is not a trivial task, nor one to be lightly undertaken.

It is often possible to collect enough information about a problem to diagnose its cause by creating customized trace entries specifically for that particular problem. For this to work well, the problem must be reproducible, and the trace buffer must be captured while the data gathered is still present.

Most people who have worked in a technical support role will agree that often the largest obstacle to solving a problem is collecting enough useful information about it. We will briefly discuss how to get enough useful data that problem solving can start in most cases. Be aware that frequently there will be some additional useful information, which can be gathered when the need for it is discovered, and that what is outlined here is not a complete list, by any means.

It is important to collect as complete a set of volatile data as possible from a single failure. If it is not gathered, it will be lost, perhaps requiring another occurrence of the problem in order to get needed information.

It is generally possible to use either an interactive debugger or a dump to diagnose either traps or hangs in an application.

For application problems, particularly traps, a good set of documentation includes the following:

- A statement of what sequence of events leads to the problem
- The trap screen, if a trap is involved
- A storage dump, with system trace data
- All the executable modules involved in the failure
- Optional application documentation, including:
  - all source files
  - .map files, produced by the linker
  - .LST and .COD or .ASM files, produced by the compiler

The storage dump is the only thing which is volatile. The rest can be collected whenever the need is discovered. To collect the first item, perform the following steps:

Warning

This will drastically change OS/2 behavior when a trap occurs. OS/2 will not control the failure, but will instantly and irrevocably stop the system, and initiate a storage dump. There will be no shutdown of the Workplace shell, databases, file systems (or lazy-write buffers) or anything else. It can be as disruptive as a power failure. It is possible to lose files or parts of files, but unlikely.

Prior to WARP: execute the command `CREATEDD A:`
This will prepare a diskette for taking a dump. The diskette will work only once. This is not required for WARP, nor for later levels of 2.11. A quick way to discover if it is required is to read the prompt which asks for the diskette at the beginning of the process. If CREATEDD is required, the prompt asks for the diskette prepared by CREATEDD, otherwise it asks for a formatted diskette.

Preparation:

1. Save the current CONFIG.SYS
2. Edit CONFIG.SYS
   a. If the line is not already present, add a line which reads `TRAPDUMP=ON`
   b. Add a line which reads `TRACEBUF=63` to enable the system trace
   c. Add a line which reads `TRACE=ON` to turn on the system trace
   d. Optionally, add a line which reads `TRACE=OFF,4,6,7`
   e. Optionally, turn LAZYWRITE off, so data goes directly to disk.
3. Locate some formatted diskettes to use for a storage dump.

   Estimate about 2MB of RAM per diskette; usually one diskette more than that number is needed. For very large systems, estimate 1.5MB per diskette. The dump process *will not* format.
4. Reboot the system so that the changes take effect.
5. Restore the original CONFIG.SYS, so you do not have to reboot an extra time to put things back to normal, after collecting the dump.

Acquiring the storage dump:

1. Cause the application to trap, that is, reproduce the problem.
2. Insert the CREATEDD diskette, if created, otherwise insert the first formatted diskette.
3. If you can read the screen, follow directions every time you hear one or more beeps.
4. If you cannot read the screen, you can still successfully get a dump, by listening for a beep. Insert the next diskette every time you hear a single short beep. When the dump is almost complete, there will be a very distinctively different series of beeps. At this point, reinsert the first diskette.
5. Very soon after the first diskette is reinserted, the dump will complete. Remove the diskette.
6. OS/2 will reboot automatically in most cases. Expect autocheck to run on HPFS drives during the boot.

7. Run CHKDSK on the drives as soon as convenient.

1.3 Hardware Architecture

This section explains how the hardware operates in protected mode, what forms of protection exist, how they operate, and what happens when a program attempts to violate one or more of the protection mechanisms.

The three protection mechanisms in 32-bit OS/2 are:

1. Privilege
2. Description
3. Address mapping

All three are active at all times when 32-bit OS/2 is running protected mode programs. Only address mapping is active when 32-bit OS/2 is running a VDM in V86 mode.

1.3.1 Address Components

All addresses in x86 processors are composed of two parts:

Addresses are usually written with a colon separating the two parts, for example, selector:offset.

1. A segment or selector
2. An offset

   The offset part will be covered during the review of typical machine instructions, because it is straightforward, and the same in real and protected modes.

These two parts are implicitly or explicitly specified by every instruction that references memory for either or both operands. Generally, the selector is implied and the offset is specified but there are exceptions to this.

1.3.1.1 NEAR and FAR Addresses

Because there are two parts of an address and an item may or may not be in a current segment, there are two ways to specify the address of a data item.

A NEAR address is an offset without specifying a selector. This is a very efficient way to address data because the overhead of loading a selector register and fetching the descriptor is avoided. The selector to use is implied, and is normally already loaded.

A FAR address contains both a selector and an offset in protect mode. This is slower and more cumbersome because both address components must be specified as well as causing the overhead of altering a selector register. When a far address is displayed from storage (as two words), the offset will be seen in the left word, and the segment or selector in the right word.

A FAR address contains a segment and an offset in real or V86 mode. The overhead is not so bad as in protect mode, because there are no descriptors to fetch when a segment register is loaded.

1.3.1.2 Real Mode and V86 Mode

```
                  Real and V86 Modes

  CS = Code Segment               SS = Stack Segment
  DS = Data Segment               ES = Extra Segment

                  for 386 and later,

  FS = another data segment       GS = another data segment


     In REAL or V86 modes, say 'segment registers'
     In PROTECT MODE, say 'selector registers'
```

Note: In real mode each segment register has a 16-bit number. The segment number is shifted left 4 bits, then added to the offset value. There is no checking of any kind.

```
DS=1234,   offset=5678

                 12340
                  5678
                 -----
                 179B8
```

This is equivalent to any of the following:

segment 179B, offset 8;

segment 1790, offset B8;

segment 1267, offset 5348;

or many other possibilities.

1.3.2 Protected Mode

In protected mode, all storage is described by the hardware, using tables maintained by the software. The description includes the location, and size of the storage segment, as well as the type of storage. The storage type further constrains how it may be used.

This section concentrates on the selector part of the address because the offset is handled in a very simple and consistent fashion once the memory segment has been located and the validity of the access has been verified.

1.3.2.1 Descriptors

A selector specifies a descriptor, which describes a memory segment. The attributes described include the base or starting address of the memory segment, the size of the segment and what accesses are allowed.

Protected mode addressing in a 386 or later begins with Descriptor Tables which are described by hardware registers. There are three Descriptor Tables, each of which is discussed below after supplying the format of individual descriptors. The tables contain the descriptors and the descriptors are selected by an interrupt number or by the content of a selector register.

An application descriptor is required for all accesses to instructions and to data. For most segments, the limit is the largest valid offset. If the offset is larger than the limit, a general protection exception occurs. The exception to this rule occurs for data segments which are expand down. In this case, the offset must be greater than the content of the limit field. The system stack (ring 0) is an example of an expand down segment.

To find the linear address of the data element, the processor adds the offset (obtained from the instruction) to the base address of the segment. That's the end of the discussion for offsets!

There are three distinct kinds of data recognized by the processor:

Stack, which holds temporary data, parameters and return addresses.

Code, which is instructions for the processor to execute.

Data, which is used to hold data that is available for longer than the lifetime of any one function or routine.

The primary distinction between stack and data is that data segments begin at offset zero and expand upward (to the limit) while stack segments begin at the highest offset and expand downward (to just greater than the limit). Many language implementations use data segments for their stack, which is perfectly acceptable, but it makes it impossible to grow the stack.

The descriptor for a memory reference is found by using the appropriate selector as the index to a table or, if you ignore the 3 lower bits, as an offset to the table, since descriptors are 8 bytes long.

1.3.3 Selector Format

In protect mode, a Selector has three fields:

1. Index, the left 13 bits, bits numbered 15-3

   This is an index into a descriptor table

2. Table indicator, one bit, bit number 2

   0 means GDT

   1 means LDT

3. RPL, the right 2 bits, numbered 1 and 0.

   Requested Privilege Level.

   Perceived as a two bit value, range 0 to 3

   00=most privileged, or ring 0; 11=least privileged, or ring 3.

The position of the bits makes a selector (with its 3 low order bits turned off) the offset into the table.

```
┌──────────────────────────────────────────────────┬───┬───┬───┐
│          I N D E X                               │TI  RPL    │ fields
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
 15  14  13  12  11  10  09  08  07  06  05  04  03  02  01  00   bit numbers
```

1.3.4 Privilege Levels

The point of privilege levels is to prevent a program from accessing a storage object that is more privileged than the program itself. Generally, this means that application programs are not able to access storage used by supervisory programs in any way. This also means it is safe to keep descriptions of storage used by the system in a descriptor table that can be accessed by applications, because the application cannot use those descriptors.

There are actually three distinct privilege levels associated with every storage access, and testing privilege level is a two-step process. The privilege level used to access a storage operand is the less privileged of CPL and RPL. The first step is to determine the actual privilege level with which to attempt the access. The second step is to compare the privilege level of the storage object (from the descriptor) to the result of the first step.

DPL	Descriptor Privilege Level.	Bits 45 and 46 of descriptor.
RPL	Requested Privilege Level.	2 low order bits of selector.
CPL	Current Privilege Level.	2 low order bits of CS.

A more privileged (lower numbered) program may access the storage objects of a less privileged program. This is how the operating system returns structures and fills in data areas for an application.

Any attempt by a less privileged (higher numbered) program to access in any way a storage object which is more privileged generates a general protection exception.

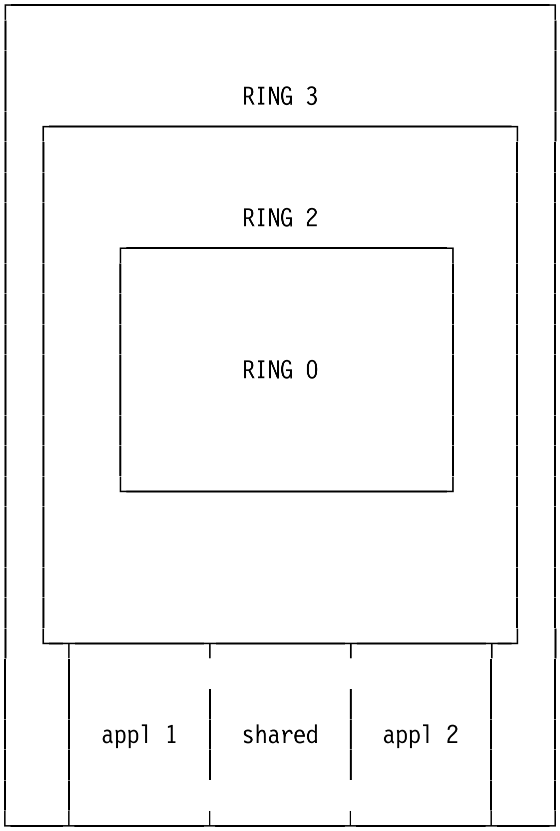


Figure 1. Ring Protection Diagram

1.3.5 Descriptor Tables

There are three tables which hold descriptors.

The three tables are:

1. The Global Descriptor Table or GDT, describes memory objects which are accessible to all processes.

   The GDT is located by means of a hardware register called the GDTR which contains the linear address and length of the GDT.

2. The Local Descriptor Table or LDT, describes memory objects which are unique to one process or are shared among a few processes by design.

   The LDT is located by means of a hardware register called the LDTR which contains a selector. The descriptor referenced by this selector must be a system descriptor which describes an LDT.

3. The Interrupt Descriptor Table or IDT, has gates that specify interrupt handler entry points.

   The IDT is located by means of a hardware register called the IDTR which contains the linear address and length of the IDT. The interrupt number is used to index into this table when an interrupt occurs.

1.3.5.1 Descriptor Fields

Type Tells what kind of object is described

Application types: Code, Data

System types: LDT, TSS, Call Gate, Irpt Gate

Base Linear address of object

Limit Defines the size of a storage object

DPL Privilege level defines which ring(s) can access the described object

```
┌──────┬──────┬───┬────┬───┬───────────┬─────────────────┬──────────┐
 LIMIT 00-15   BASE 0-23   TYPE S DPL P LIMIT 16-19 FLAGS BASE 24-31
└──────┴──────┴───┴────┴───┴───────────┴─────────────────┴──────────┘
   0      1     2   3   4       5               6              7
                            byte offsets
Display a descriptor with 'DB' to see it in this form.

Notes:
 TYPE is what kind of object is described
    S is descriptor category; 0=system, 1=code or data
   PL is privilege level of object described
    P is the present bit; 1=present, 0=not present
```

1.3.5.2 Descriptor Flags

Bit 55 Granularity: (G) 0=limit is in bytes, 1=limit is in 4K pages

Bit 54 Default address size: 0=16 bit, 1=32 bit

Bit 53 Default operand size: 0=16 bit, 1=32 bit

Bit 52 Unused by hardware, used by OS/2 to indicate UVirt

Bit 47 Present: (P) 1=segment is present, 0=segment is not present

Bits 46 and 45 Privilege Level: 00=most, 11=least

Bit 44 Segment type: 0=system segment, 1=application segment

Bit 40 Accessed: (A) 0=not accessed, 1=accessed

Note: If application segment, (Bit 44 = 1), used to store program code and data.

Bit 43=0 is Data Segment

Bit 42: Expansion: 0=Expand Up, 1=Expand Down

Bit 41: Writeable: 0=Read Only, 1=Read/Write

Bit 43=1 is Code Segment

Bit 42: Conforming: 0=Nonconforming, 1=Conforming

Bit 41: Readable: 0=Execute Only, 1=Read/Execute

Note: If system segment, (Bit 44 = 0)

Bits 39-42 Type of segment

00 RESERVED

01 Available 286 TSS (16-bit)

02 LDT

03 Busy 286 TSS (16-bit)

04	286 Call Gate (16-bit) (Parm Count is words)
05	Task Gate
06	286 Interrupt Gate (16-bit)
07	286 Trap Gate (16-bit)
08	RESERVED
09	Available 386 TSS (32-bit)
10	RESERVED
11	Busy 386 TSS (32-bit)
12	386 Call Gate (32-bit) (Parm count is doublewords)
13	RESERVED
14	386 Interrupt Gate (32-bit)
15	386 Task Gate (32-bit)

1.3.5.3 Descriptor Table Summary

There are three descriptor tables at any instant:

1. Global Descriptor Table

   Located via GDTR

   1 per system

   Accessible to all processes

   Describes objects common to all processes

2. Local Descriptor Table

   LDTR is selector

   GDT Descriptor Locates LDT

   1 per process except VDM

   Describes data unique to one process

3. Interrupt Descriptor Table

   Located via IDTR

   1 per VDM + 1 per system for protect mode

   Describes interrupt routine entry points

1.3.6 The Selector Registers

Each selector register appears to be 16 bits long. The six application selector registers and a brief description of the use for each follow:

SS: Stack Selector, specifies the descriptor used for stack references.

CS: Code Selector, specifies the descriptor used for instruction references.

DS: Data Selector, specifies the descriptor used for most data references.

ES: Extra Selector, specifies another descriptor used for data references.

FS: This is a selector which can be used for data references if explicitly specified.

GS: This is a selector which can be used for data references if explicitly specified.

The two system selector registers and a brief description of the use for each follow:

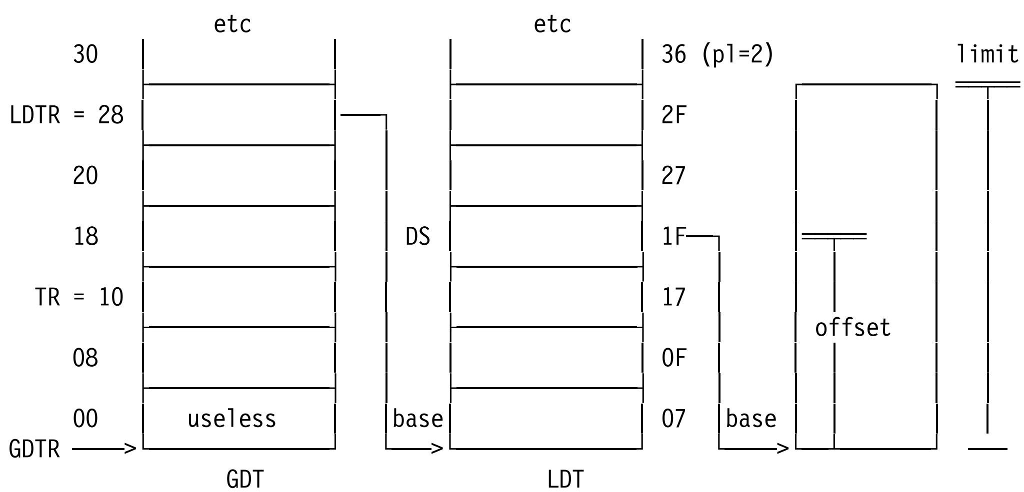
LDTR: The LDT register selects the LDT descriptor from the GDT.

TR: The Task Register selects the descriptor used for the TSS.

1.3.7 When Checking is Done

When a program moves data into a selector register, that data becomes a selector and the processor fetches the content of the appropriate entry from the specified table into onboard registers which are not accessible to the programmer. The processor verifies the validity of the attempted access to the memory whenever a selector register is updated. This makes the protection overhead occur as part of the instruction which modifies a selector register, but eliminates it for further use of the selector.

If the RPL of the SS register is not the same as CPL, or if an attempt is made to move the null selector into SS, a general protection exception occurs.



Note: The first descriptor in the GDT is reserved, by definition, and cannot be used. Any selector which would reference it is called the NULL selector; possible values are 0000, 0001, 0002, and 0003.

By definition, the null selector may be placed in DS, ES, FS, or GS, but any attempt to form an address with it is a general protection fault.

The LDTR is a register that contains a selector. It can be accessed only by privilege level 0 instructions. It must contain a selector that references the GDT, and a descriptor whose type is LDT.

It is not unusual for a GDT selector to describe the same storage as an LDT selector does. In OS2 2.x, application selectors in the GDT happen to describe one 448 Meg segment, not just a 64K segment like the LDT selectors describe. The linear address assigned to each LDT descriptor is extremely convenient for changing one form of an address to another, called thunking, which will be discussed later.

1.3.8 Descriptor Examples

These examples come from DUMP01, which is used for several exercises.

```
DL 7 37
0007  Data    Bas=ac6d7000 Lim=0000ffff DPL=3 P  RO
000f  Code    Bas=00010000 Lim=00002e77 DPL=3 P  RE    A
0017  Data    Bas=00020000 Lim=0000290f DPL=3 P  RW    A
001f  Data    Bas=00030000 Lim=000018af DPL=3 P  RW    A
0027  Data    Bas=00040000 Lim=0000030a DPL=3 P  RW    A
002f  Data    Bas=00050000 Lim=00000fff DPL=3 P  RW
0036  Data    Bas=00060000 Lim=00000fff DPL=2 P  RW    A

DL BECF
bece  Code    Bas=17d90000 Lim=00000010 DPL=2 P  RE    A

DL BFD7 BFEF
bfd7  Data    Bas=17fa0000 Lim=0000ffff DPL=3 P  RW    A
bfdf  Data    Bas=17fb0000 Lim=0000ffff DPL=3 P  RW    A
bfee  Code    Bas=17fd0000 Lim=00000aa2 DPL=2 P  RE    A

DG 20 78
0020  Data    Bas=ffe5b000 Lim=000003ff DPL=0 P  RW       UV
0028  LDT     Bas=ac6d7000 Lim=0000ffff DPL=0 P
0030  Data    Bas=ffe09de4 Lim=0000421b DPL=0 P  RW ED A  UV
003b  Data    Bas=ff4cbe2c Lim=00000073 DPL=3 P  RW
0040  Data    Bas=ffe5a400 Lim=000003bf DPL=0 P  RW       UV
004a  Data    Bas=00000000 Lim=1bffffff DPL=2 P  RW    A  G4k BIG UV
0053  Data    Bas=00000000 Lim=1bffffff DPL=3 P  RW    A  G4k BIG UV
005a  Code    Bas=00000000 Lim=1bffffff DPL=2 P  RE C  A  G4k C32 UV
0063  Data    Bas=00000000 Lim=1fffffff DPL=3 P  RW       G4k BIG UV
006b  Data    Bas=00000000 Lim=1bffffff DPL=3 P  RW    A  G4k BIG UV
0070  Data    Bas=ffe22000 Lim=000074e4 DPL=0 P  RO    A
0078  Data    Bas=ffe22000 Lim=000074e4 DPL=0 P  RW

DG 148 L 4
0148  Code    Bas=fff39000 Lim=00009262 DPL=0 P  RE    A
0150  Code    Bas=fff43000 Lim=0000e137 DPL=0 P  RE    A
0158  Data    Bas=00000000 Lim=ffffffff DPL=0 P  RW    A  G4k BIG
0160  Code    Bas=00000000 Lim=ffffffff DPL=0 P  RE    A  G4k C32
```

The top section of the above output was created by entering the command
`DL 7 37`

By inspecting the type, base, and limit fields in the above output, we can see the following about the descriptor referenced by 002F:

The storage is described as data having a base or linear, address of 00050000. The linear address is not normally written with leading zeros. If there were any chance that the address might be mistaken for physical, a percent sign would be used, for example, %50000. The limit is FFF, which means that the segment is 4K or 1000(hex) long. The privilege level is 3, the segment is present, and the flags indicate Read/Write storage. It has *not* been accessed, because the ′A′ flag is not present and OS/2 no longer uses this flag; once set by the hardware, it remains set.

Examples related to privilege level protection follow below:

CS:IP	CPL	DS:xxxx	RPL	lesser privilege CPL and RPL	DPL (from descriptor)	Access allowed?
000F:xxxx	3	17:xxxx	3	3	3	Yes
000F:xxxx	3	16:xxxx	2	3	3	Yes
000F:xxxx	3	14:xxxx	0	3	3	Yes
000F:xxxx	3	37:xxxx	3	3	2	No
000F:xxxx	3	36:xxxx	2	3	2	No
000F:xxxx	3	34:xxxx	0	3	2	No
000F:xxxx	3	43:xxxx	3	3	0	No
000F:xxxx	3	42:xxxx	2	3	0	No
000F:xxxx	3	40:xxxx	0	3	0	No
BECE:xxxx	2	17:xxxx	3	3	3	Yes
BECE:xxxx	2	16:xxxx	2	2	3	Yes
BECE:xxxx	2	14:xxxx	0	2	3	Yes
BECE:xxxx	2	37:xxxx	3	3	2	No
BECE:xxxx	2	36:xxxx	2	2	2	Yes
BECE:xxxx	2	34:xxxx	0	2	2	Yes
BECE:xxxx	2	43:xxxx	3	3	0	No
BECE:xxxx	2	42:xxxx	2	2	0	No
BECE:xxxx	2	40:xxxx	0	2	0	No
0150:xxxx	0	17:xxxx	3	3	3	Yes
0150:xxxx	0	16:xxxx	2	2	3	Yes
0150:xxxx	0	14:xxxx	0	0	3	Yes
0150:xxxx	0	37:xxxx	3	3	2	No
0150:xxxx	0	36:xxxx	2	2	2	Yes
0150:xxxx	0	34:xxxx	0	0	2	Yes
0150:xxxx	0	43:xxxx	3	3	0	No
0150:xxxx	0	42:xxxx	2	2	0	No
0150:xxxx	0	40:xxxx	0	0	0	Yes

In each case, as you read across you will see that CPL comes from the value of the CS register, RPL comes from the two low-order bits of the selector, and DPL comes from the descriptor. The column titled 'lesser privilege' is calculated remembering that higher numbers are lower privilege. The final column is obtained by following the access rules, a short way back.

1.4 Exercise 1: Selectors and Descriptors

Objectives:

1. Learn how to load a dump for analysis.
2. Introduction to the dump formatter.
3. Learn how to display descriptors.

Start the lab at a full-screen or windowed command prompt.

A full-screen session is faster, but a windowed session can be made 100 lines high by entering

MODE CO80,100
This can be very useful, because you can look back quite a ways by using the scroll bar.

Change to directory HANDS-ON\UTILS

Make diskette one by typing OS2IMAGE ..\IMAGES.162\LAB01.001 A:

Make diskette two by typing OS2IMAGE ..\IMAGES.162\LAB01.002 A:

Load the dump into a new file which will be named DUMP01.DMP by typing

DCOMP A: X:\DUMP01.DMP and pressing enter, then following the prompts.

When the dump is loaded, it should have a file size of 4194816.

Start the dump formatter by typing DF_RET X:\DUMP01.DMP

or by (X is the CD-ROM drive)
X:HANDS-ON8162.DFDF_RET X:HANDS-ON\DUMPS.162\DUMP01.DMP

You should see 6 or 7 informational lines at the top, followed by a pair of lines which start ″Slot″, and ″*0023#″, followed by a set of registers.
*** We are not yet concerned with any of these. ***

You should get a prompt, which is the character #

Note: You can always document what you are thinking by simply typing it in as an evaluation for the dump formatter to perform. You can access the evaluation function by typing ? followed by whatever you want echoed to the screen and to the log. You can also type in ? and any expression to have it evaluated and output in hex, decimal, octal, binary, character and Boolean forms.

? by itself is a simple request for what commands are recognized.

Use the dump formatter to look at descriptors and answer these questions.

The dump formatter is *not* case sensitive.

Descriptors may be displayed using DG or DL followed by the selector. Try it both ways for several selectors, such as F, 160, DFFF, 158.

Use the miniature command reference in the appendix, if necessary.

There are a great many things we will *not* do in this exercise. We are using only a tiny part of the dump formatter′s capabilities for this class. For example, we will ignore the IDT in this class; one can enter DI followed by the interrupt number to see the descriptor from the IDT.

Questions to answer:

1. Which table contains the descriptor data for selector 000F?

2. Which command is preferred to display only the descriptor for 000F?

3. What alternative command will also display only the descriptor for 000F?

4. What type of memory is described by selector 000F?

   Hint: It is one of the first things displayed in the output for each descriptor.

5. What is the largest valid offset within segment 000F?

6. What is the size of segment 000F?

   Hint: Not quite the same as the previous answer.

7. What is the linear (virtual) address of segment 000F?

8. What privilege level is segment 000F?

9. What is the Requested Privilege Level of selector 000F?

   Hint: RPL is not in the descriptor.

10. What is the type and limit of segment 0017?

11. What is the linear (virtual) address of segment 0017?

12. Which table contains descriptor 0017?
13. Will the aplication program be able to access the segment selected by 000F?

    Explain.____

14. Will the program be able to store into segment 000F?

    Explain.____

15. Will the application program be able to access storage using selector 0037?

    Explain.____

16. Will the program be able to write into storage using selector 0038?

    Explain.____

17. Will the program be able to write into storage using selector 0007?

    Explain.____

18. Enter the following command: `DG 70 L 2`

    Compare and contrast the base, limit, privilege level and flags for each.

19. Enter the following command: `DG 5A;DG 5B`

    Compare and contrast the base, limit, privilege level and flags for each.

20. Enter the following command: `DG 28;DL 7`

    Compare and contrast the base, limit, privilege level and flags for each.

The dump formatter will exit in response to the command `Q`

1.5 Address Mapping

This section describes the method used to transform addresses from linear addresses to physical addresses.

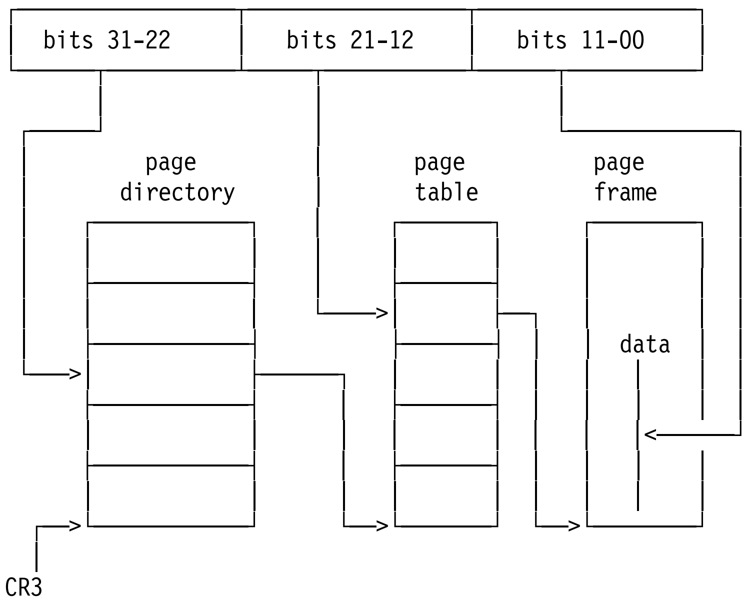
1.5.1 Paging Overview

OS/2 V2 uses paging in addition to the above logical addressing. Paging is a mechanism which converts linear addresses to physical addresses and allows a consistent size (4K) to be moved back and forth to auxiliary storage (SWAPPER.DAT) when the demand for virtual memory exceeds the physical memory installed on the machine. Another hardware register, Control Register 3 or CR3, is used to locate a page directory which contains table entries that locate page tables. The page tables are used to locate the physical memory where the data really resides. Physical memory is sometimes referred to by page number. A page number is simply the twenty high-order bits of an address. The twelve low-order bits of a page address are all zero. One can convert a page number to an address by simply appending three hex zeros to it.

The result of combining a segment number and an offset, or the addition of an offset to the base address from a descriptor, is a linear address. Under OS/2 1.x, these would be physical addresses. Under OS/2 2.0 and following, these are linear or virtual addresses.
The picture below shows how linear addresses are converted to physical addresses. Only the top line in the picture below is a linear address - the rest are physical.

The ten high order bits of the linear address are used to index into the Page Directory which has the twenty high order bits of the page table's physical address (page number). The next ten bits of the linear address are used to index into the page table. The twenty high order bits of the page frame's physical address (page number) are retrieved. The twelve low order bits of the linear address are also the twelve low order bits of the physical address. Therefore, the physical address is the twenty bits from the page table entry, followed by the 12 low-order bits from the linear address.

LINEAR ADDRESS



CR3 and the table entries all have physical addresses!

1.5.2 Page Table Entries

The page directory entries are identical to the page table entries.

Each entry is 4 bytes, making 1K entries in each page table.

Bits 31-12 Physical address of page or page frame address

Bits 11-09 Ignored by hardware, used by OS2. See Note.

Bits 08-07 Reserved, must be zero

Bit 6 Dirty (D) 0=not changed (clean), 1=changed (dirty)

Bit 5 Accessed (A) 0=not accessed, 1=accessed

Bit 4 Page Cache Disable 0=allow cache use, 1=bypass cache

Bit 3 Page Write-Through 0=cache write-into, 1=write through to RAM

Bit 2 Supervisor (S/U) 0=Supervisor (PL=0,1,2), 1=user (PL=3)

Bit 1 Write enable (RO/RW) 0=Read Only, 1=Read/Write

Bit 0 Present (P) 0=not present, 1=present

Note: The left 5 hex digits of the entry are the left 5 hex digits of the physical page; while the right 3 hex digits are mostly flags.

If Bit 0 is zero, (page invalid) the remaining bits are *not* inspected by the hardware. OS/2 uses them to identify the virtual page associated with this address.
Bits 09 and 10 are used to track the state of the page frame.

Three of the possible four combinations are used:

0 - Pageable

1 - UVirt

2 - Resident

1.5.3 Page Table Contents

To look at the contents of the page directory and page table(s), use the command DP, followed by the address of interest.

```
DP F:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00010000* 001e0  frame=0009e  0    0  c  A        U  r  P  pageable
%00010000  0009e  frame=0009e  0    0  c  A        U  r  P  pageable
%00011000         vp id=012ae  0    0  c  u        U  r  n  pageable
%00012000  00292  frame=00292  0    0  c  A        U  r  P  pageable

DP 17:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00020000* 001e0  frame=00181  0    0  D  A        U  W  P  pageable
%00020000  00181  frame=00181  0    0  D  A        U  W  P  pageable
%00021000  003d4  frame=003d4  0    0  D  A        U  W  P  pageable
%00022000  0005a  frame=0005a  0    0  D  A        U  W  P  pageable

DP 1F:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00030000* 001e0  frame=003ae  0    0  D  A        U  W  P  pageable
%00030000  003ae  frame=003ae  0    0  D  A        U  W  P  pageable
%00031000  001b5  frame=001b5  0    0  D  A        U  W  P  pageable

DP 27:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00040000* 001e0  frame=00052  0    0  c  A        U  W  P  pageable
%00040000  00052  frame=00052  0    0  c  A        U  W  P  pageable

DP 2F:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00050000* 001e0  frame=00075  0    0  D  A        U  W  P  pageable
%00050000  00075  frame=00075  0    0  D  A        U  W  P  pageable

DP 37:0
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00060000* 001e0  frame=002ae  0    0  D  A        U  W  P  pageable
%00060000  002ae  frame=002ae  0    0  D  A        U  W  P  pageable
```

In each case, the first line of output is the data from the page directory.

The field labelled ′frame′ is the physical page frame which holds the data at the referenced address.

The ′vp id′ is the virtual page identifier for the entry %11000.

′Dc′ is Dirty or Clean. ′Au′ is Accessed or unaccessed.

′Us′ is User (Ring 3) or supervisor (Rings 0 and 2).

′rW′ indicates read-only or Writeable. ′Pn′ indicates Present or not present.

1.6 Data Format in Storage

Data format is least significant byte at lowest address!

This arrangement is not intuitive for many people, because when you read bytes, the data placement seems reversed. The tools will let you display storage as bytes, words, and doublewords; the data will be re-arranged to suit the format requested. This can be good or bad.

For example:

```
DB 1F:1608 L 20
001f:00001608 42 4f 4f 4b 53 48 45 4c-46 3d 43 3a 5c 4f 53 32 BOOKSHELF=C:\OS2
001f:00001618 5c 42 4f 4f 4b 3b 00 43-4f 4d 53 50 45 43 3d 43 \BOOK;.COMSPEC=C

DA 1F:1608
001f:00001608 BOOKSHELF=C:\OS2\BOOK;

DB 17:0 L40
0017:00000000 02 00 03 00 05 00 07 00-0b 00 0d 00 11 00 13 00 ................
0017:00000010 17 00 1d 00 1f 00 25 00-29 00 2b 00 2f 00 35 00 ......%.).+./.5.
0017:00000020 3b 00 3d 00 43 00 47 00-49 00 4f 00 53 00 59 00 ;.=.C.G.I.O.S.Y.
0017:00000030 61 00 65 00 67 00 6b 00-6d 00 71 00 7f 00 83 00 a.e.g.k.m.q.....

DW 17:0 L20
0017:00000000  0002 0003 0005 0007 000b 000d 0011 0013
0017:00000010  0017 001d 001f 0025 0029 002b 002f 0035
0017:00000020  003b 003d 0043 0047 0049 004f 0053 0059
0017:00000030  0061 0065 0067 006b 006d 0071 007f 0083

DW 17:1 L 20
0017:00000001  0300 0500 0700 0b00 0d00 1100 1300 1700
0017:00000011  1d00 1f00 2500 2900 2b00 2f00 3500 3b00
0017:00000021  3d00 4300 4700 4900 4f00 5300 5900 6100
0017:00000031  6500 6700 6b00 6d00 7100 7f00 8300 8900

DD 17:0 L 10
0017:00000000  00030002 00070005 000d000b 00130011
0017:00000010  001d0017 0025001f 002b0029 0035002f
0017:00000020  003d003b 00470043 004f0049 00590053
0017:00000030  00650061 006b0067 0071006d 0083007f

DD 17:1 L 10
0017:00000001  05000300 0b000700 11000d00 17001300
0017:00000011  1f001d00 29002500 2f002b00 3b003500
0017:00000021  43003d00 49004700 53004f00 61005900
0017:00000031  67006500 6d006b00 7f007100 89008300

DD 17:2 L10
0017:00000002  00050003 000b0007 0011000d 00170013
0017:00000012  001f001d 00290025 002f002b 003b0035
0017:00000022  0043003d 00490047 0053004f 00610059
0017:00000032  00670065 006d006b 007f0071 00890083

You need to know what you are looking at!
```

1.7 Exercise 2: Paging, Addresses, Data

Objectives:

1. Reinforce the knowledge from exercise 1.
2. Learn how to display page table data.
3. Learn how to convert a logical address to a linear address.
4. Learn how to convert a linear address to a physical address.
5. Learn how to display storage as ASCII, bytes, words and doublewords.

Startup directions:

1. Start the dump formatter by typing (X is the CD-ROM drive)
   `X:HANDS-ON8162.DFDF_RET X:HANDS-ON\DUMPS.162\DUMP01.DMP`

2. You should see the standard startup messages.

3. The initial register display is what the application registers were at the time the application (ring 3) program trapped.

4. You can see these at any time by entering the `.R` command.

5. Use the dump formatter to look at the dump and answer these questions. The dump formatter is NOT case sensitive.

Note: Paging data may be displayed using the `DP` command, followed by the address.
The dump process *destroys* the first entry of the page directory. You will get quite confused if you try to follow the hardware method to look at paging information for addresses 0 - 3FFFFF.
If you must, use the `.N` command to find "savepage", which will tell you the physical address of the page table for that address range.

This may well be the last time you use a physical address in an OS/2 debugging session. With the notable exceptions of physical memory management and physical device drivers, OS/2 is almost completely unaware of physical addresses. The 32-bit virtual address, also called a linear address, and a flat address, is what is used in general throughout OS/2.

Assuming these registers, answer the following questions:

```
eax=0000c8cf ebx=00002910 ecx=000000df edx=00000000 esi=00000030 edi=00000060
eip=000000be esp=000014be ebp=000014e6 iopl=2 rf -- -- nv up ei pl zr na pe nc
cs=000f ss=001f ds=001f es=0017 fs=150b gs=0000  cr2=00000000  cr3=001a7000
```

1. What are the base and limit fields for selector 000F?
   (the base is the linear address)

2. How many 4K pages are in this segment? Hint: Look closely at the limit field.

3. How many physical pages are allocated for the virtual memory segment starting at F:0?

   Hint: `DP 0F:0` or `DP %10000`

4. Why are the above two answers different?

5. What is the physical address of the data at F:0?

   Observation: You now have three ways to address the data.

   a. Real or V86 (&selector:offset)
   b. Logical (#selector:offset)
   c. Linear (%address)
   d. Physical (%%address)

   We will now display the same storage many ways, to confirm we know how.

6. What is the command to display the storage at SS:BP in words using a logical address?

7. What is the command to display the storage at SS:BP in words using a linear address?

8. What is the command to display the storage at SS:BP in words using a physical address?

   For each of the following, study the results until you understand.

9. Display the data at 7:0 as bytes, and words.

10. Display the data at 7:1 as bytes and words.

11. Display the data at 7:0 and 7:1 as words.

12. Display the data at 7:0 as words and doublewords.

13. Display the data at 1F:15C6 as bytes and ASCII. Also look at 1F:15DA as bytes and ASCII.

1.8 Instruction Set

This section discusses the Intel 86 registers and some common instructions from the instruction set.

1.8.1 Register Review

Registers discussed so far:

CR3 32-bit physical address of the Page Directory

IDTR 32-bit linear address of IDT, 16-bit size of IDT

GDTR 32-bit linear address of GDT, 16-bit size of GDT

LDTR 16-bit selector for an entry (type 2) in the GDT

SS 16-bit selector, used for stack operations

CS 16-bit selector, used to locate instructions

DS 16-bit selector, used to locate data, generally the default

ES 16-bit selector, used to locate data, string destination

FS 16-bit selector, used to locate data explicitly

GS 16-bit selector, used to locate data explicitly

1.8.2 Execution

The 386 execution consists of the classic pattern of fetching an instruction from memory and executing it, then repeating the process. The instructions are always found in a code segment accessed via the descriptor designated by the selector in the CS register. The current privilege level of the program is contained by the two low order bits in the CS register. The offset of the next instruction is contained in the instruction pointer, (IP or EIP) which is incremented as each instruction is fetched. The 386 and following generations recognize a great number of instructions, but compilers generate a very small subset of the whole instruction set. Much of that subset will be discussed here. If you cannot ascertain what an instruction does when you encounter it, look it up in the appropriate reference manual. Instructions are generally executed sequentially, and the processor attempts to fetch instructions well in advance, to increase execution speed. The flow of control departs from sequential when a jump, call, return, interrupt or interrupt return is encountered. Jumps are conditional or unconditional. Conditional jumps are used to implement decisions and contain a relative offset which is combined with IP by signed addition to cause a different instruction in the same segment to be executed next. Calls, returns and unconditional jumps come in two varieties: NEAR and FAR. The NEAR variety update only IP and leave CS untouched. The FAR variety update both CS and IP and are potentially quite complex. CALL, RETurn and interrupts require a stack. Most instructions reference the registers.

1.8.3 General Registers

```
                          EAX                          ALL 32 BITS
  ( part of EAX )                        AX            LOW 16 BITS
  ( part of AX )                  AH                   HIGH 8 BITS
  ( part of AX )                              AL       LOW 8 BITS
```

```
  Registers EBX, ECX and EDX  also subset in the same way.
  There are two byte-sized pieces, which can be collectively
  referenced as a word-sized item.
```

```
                          EIP                          ALL 32 BITS
  ( part of EIP )                        IP            LOW 16 BITS
```

```
IP and EIP are always offsets into CS.
They always contain the address of the next instruction to execute.
```

```
                          ESP                          ALL 32 BITS
  ( part of ESP )                        SP            LOW 16 BITS
```

```
SP and ESP are always offsets into SS.
They contain the address of the last item pushed into the stack.
```

```
  REGISTERS EBP, ESI and EDI also subset in this way.
                             They have no 8 bit parts.
```

1.8.4 Machine Instructions

There are several fields which may be present in an instruction. Additionally, there are a few easy-to-learn generalities which will make understanding what an instruction does much easier. Data definitions will not be covered here.

There are many fields possibly present in an instruction.

1. The label.

   The label is optional, but must be first. It is followed by a colon. It is used so the programmer can refer to the instruction symbolically. A label does not require an instruction.

   Labels which are Public become symbols at link time.

2. The mnemonic operation code, or opcode, is next.

It defines what operation will be attempted, and therefore what operands need to be specified. Instructions have zero to three specified operands; many instructions also imply operands.

3. The operands are next, separated by commas.

   The first operand is always the result, or target, of the operation.

   An operand may be a value, a register, or storage. When the operand is a value, it is called immediate, because the operand is immediately available if the instruction has been fetched. When a register is named, it is the operand. If an expression is contained in brackets, it is evaluated and the result is used as a offset into some segment.

   A storage operand is in some segment by default. Data references default to the data segment or DS, unless (E)BP or (E)SP are present in the address expression. In this case, the default segment is the stack segment (SS). (E)IP is *always* in the CODE segment (instructions). (E)SP is *always* in the STACK segment (data). (E)BP is *usually* in the STACK segment (data).

   The default segment can usually be overridden by specifying the selector as part of the address, for example, DS:[BP+8].

   You will come across helper words within operands, such as byte, word and dword which are there to remind you of the size of the data item referenced. You will also come across the helper word ″ptr″, which is to remind you that the addressed data is in storage, and that the offset, in brackets, is a pointer to the data.

4. The last item you may find is an optional comment.

   A comment is preceded by a semicolon. Anything following is a comment. Comments are sparse in the output of the Unassemble command.

   The debug kernel will use a comment to identify a breakpoint.

   Both the debug kernel and the dump formatter will supply a symbol anytime a number matches the symbol in an active file.

1.8.5 Typical Instructions

MOV CL,DH

The opcode is MOV, the first operand is the CL register, and the second operand is the DH register. This instruction will copy (MOVe) all 8 bits from the DH register to the CL register.

MOV DX,8

The opcode is MOV, the first operand is the DX register, the second operand is the immediate value of 8. This instruction puts the value 8 into the DX register.

MOV EBP,ESP

Again, the opcode is MOV, and the instruction will copy all 32 bits of ESP into EBP.

MOV AX,BX

You should be able to tell by now that this instruction will copy 16 bits from BX to AX. Note that instructions which reference only registers are extremely unlikely to cause an exception.

MOV AX,word ptr [BX]

This instruction is different from the one above because there are brackets around the second operand. This means that the operand, BX in this case, is in storage, and the BX register holds the offset into the DS segment. If BX is outside the limit of the DS segment, a general protection fault will occur.

MOV word ptr [BX],AX

This instruction is similar to the preceding one, but moves data into storage, rather than from storage. The same exceptions might occur, and if the DS segment is read-only, this instruction would also fail.

MOV word ptr ES:[BX],DI

This is an example of overriding the default segment, DS, by explicitly specifying that the offset in the BX register applies to the ES segment.

ADD word ptr DS:[BP],AX

This would add the 16 bits from AX into storage at DS:BP, developing the sum directly in storage. The override is needed because the use of BP means that the default segment is SS.

DEC word ptr [BP-2]

Some instructions have only one operand. In this case it is in storage at an offset calculated by subtracting 2 from the BP value, in the segment defined by the SS register, because BP is used.

Also SUB, CMP, AND, OR, XOR, XCHG, INC, SHL, etc.

It is extremely common for 16-bit code to use FAR addresses. When they are in storage, it would require several instructions to get a FAR address into the registers, if it were not for several instructions whose purpose is specifically to fetch a FAR address from storage into a selector and another register. These instructions may be recognized by the opcode, which is the letter L followed by a selector register name other than CS. The apparent first operand is the general, base, or index register which will hold the offset part of the far address. Both registers will be loaded, with the first operand coming from the address specified, and the selector coming from the following word.

LES BX,dword ptr [BP+6]

This instruction loads both BX and ES. BX comes from BP+6 and ES comes from BP+8, both in the stack segment.

LDS SI,dword ptr [BP-12]

This instruction loads both SI and DS, SI is loaded from BP-12 and DS is loaded from BP-10.

LEA EDI,[EBP+ECX*4-12]

Load Effective Address DOES NOT actually reference storage. Instead, once the offset has been calculated, it is put into the target register, EDI in this case. Address expressions like this are possible, but not often seen while actually debugging. The scale factor can be 1, 2, 4, or 8; not any arbitrary value.

1.8.6 The System Flags

The flags not only control system operation, but also hold the result of instructions such as CMP (compare). At times, you will find the flags have been copied to a register, or to memory. The following figure gives the format of the flags in such cases:

Bit	Hex	Flag name	Comments
18	00040000	AC	Alignment Check, if the alignment mask is 1 (CR0).
17	00020000	VM	V86 mode. Turned on for Virtual DOS Machines.
16	00010000	RF	Resume Flag. Suppress debug exceptions for 1 instruction.
14	00004000	NT	Nested Task. Involved with hardware task switching.
13/12	3000	IOPL	The least privileged code which has unrestricted I/O access.
11	0800	OF	Overflow. An arithmetic result does not fit.
10	0400	DF	Direction of string instructions. 0=up, 1=down.
09	0200	IF	Interrupt flag. 1=enabled, 0=disabled.
08	0100	TF	Trap flag. Generate a debug exception after each instruction.
07	0080	SF	Sign. 1=minus, 0=plus.
06	0040	ZF	Zero or Equal. 1=zero result, 0=non-zero result.
04	0010	AF	Auxiliary flag. Used in BCD arithmetic.
02	0004	PF	Parity flag. 0=even, 1=odd.
00	0001	CF	Carry flag. 0=no carry, 1=carry.

1.8.7 Unassembled Instructions

```
U CS:IP-22 IP-18
000f:0000009c f1            db      f1
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-23 IP-18
000f:0000009b f7f1          div     cx
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-24 IP-18
000f:0000009a ee            out     dx,al
000f:0000009b f7f1          div     cx
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-25 IP-18
000f:00000099 56            push    si
000f:0000009a ee            out     dx,al
000f:0000009b f7f1          div     cx
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-26 IP-18
000f:00000098 8b56ee        mov     dx,word ptr [bp-12]
000f:0000009b f7f1          div     cx
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-27 IP-18
000f:00000097 ec            in      al,dx
000f:00000098 8b56ee        mov     dx,word ptr [bp-12]
000f:0000009b f7f1          div     cx
000f:0000009d 8946fc        mov     word ptr [bp-04],ax
000f:000000a0 f7e1          mul     cx
000f:000000a2 8946f4        mov     word ptr [bp-0c],ax
000f:000000a5 3946f6        cmp     word ptr [bp-0a],ax

U CS:IP-10 IP
000f:000000ae 3976f0        cmp     word ptr [bp-10],si
000f:000000b1 77df          ja      0092
000f:000000b3 3976f0        cmp     word ptr [bp-10],si
000f:000000b6 7510          jnz     00c8
000f:000000b8 c45ede        les     bx,dword ptr [bp-22]
000f:000000bb 8b46f6        mov     ax,word ptr [bp-0a]
000f:000000be 268907        mov     word ptr es:[bx],ax
```

1.8.8 Observations About Unassembling from an Unknown Starting Place

Instructions are of variable length, from one to fifteen bytes long.

This means the address you provided may not actually be the start of an instruction. This also means, therefore, that the first few instructions you see may not actually be what the machine saw.

If you look at the output of several unassemblies starting at sequential addresses, you will see that after typically three to five tries, the unassembly will agree with previous ones, for some point after the unassembly started.

This is typically within four or five lines, but not always. Be cautious, and see if the sequence looks reasonable. If it does, you have most likely found an instruction boundary. Experience will help this process.

Some common sense will help as well. Obviously, an application in ring 3 cannot perform I/O directly. Likewise, the *db* means that the unassembler did not have a way to interpret this as an instruction.

The last command entered looks at a few of the instructions which actually preceded a failure.

Can you discover which instruction put the data into the ES and BX registers?

1.9 Exercise 3: Unassembling and Reading Instructions

Objectives:

1. Reinforce the preceding lab exercises
2. Learn how to unassemble instructions
3. Learn how to read instructions
4. Learn about variable length instructions

We will now look at instructions.

1. In what type of segment are instructions found?

2. Are instructions ever executed in any other segment type?

3. Unassemble the instructions which would have been next to execute (if the application hadn't trapped) by entering `U`. The default address is CS:IP initially. You can unassemble further with repeated use of `U`. To unassemble at a particular place, specify the address; for example CS:IP.

4. What was the next instruction which would have executed?

5. Unassemble using an address range to see some previous instructions. Type `U CS:IP-20 IP-10`. This will unassemble from ip-20 to ip-10. Now type `U CS:IP-21 IP-10` and `U CS:IP-22 IP-10`. Observe what is happening by closely observing the address at which each instruction begins.

6. Now type `U CS:IP-18 IP` to see the two instructions immediately before the failing instruction

   (at CS:IP!). What are they?

7. Which one loaded the address used in the next (failing) instruction?

8. Did the address come from this routine's private data, or was it a parameter passed by the caller?

   This is presented in detail later.

9. Circumstantially at least, what seems to be wrong?

   Also presented later.

of pages will face each other.

1.10 Exceptions

Events sometimes occur which disrupt the normal sequence of instruction. These are called exceptions and interrupts. Intel defines exceptions in relation to an unsuccessful attempt to execute an instruction. Interrupts are defined as a hardware response to a event unrelated to program execution.

Trap Hex	Type	B/C	Error Code	Source Cause
Hex			CODE	CAUSE
0	Fault	C	No	Divide Overflow (perhaps by zero)
1	DR6	B	No	Debug Exception
2	Int	B	No	NMI (Non-Maskable Interrupt), normally hardware fault
3	Trap	B	No	Breakpoint (INT 3 instruction)
4	Trap	B	No	Overflow (INTO instruction)
5	Fault	B	No	Bounds Check (BOUND instruction)
6	Fault	B	No	Invalid Opcode
7	Fault	B	No	Co-processor not available, see note
8	Abort	Abort	Always Zero	Double Fault, any instruction
9	Fault	C	Yes	Co-processor Segment Overrun (286,386 only) (Fault D in 486+)
A	Fault	C	Yes	Invalid TSS
B	Fault	C	Yes	Segment Not Present (swapped out)
C	Fault	C	Yes	Stack Exception
D	Fault	C	Yes	General Protection
E	Fault	PF	Yes	Page Fault (paged out)
F				(reserved)
10	Fault	B	No	Co-processor Error
11	Fault	?	Always Zero	Alignment Check
12	Abort	??	Machine Check	
13-1F				(reserved)
20-FF	Trap	N/A	No	Available for Hardware Interrupts Via ′INTR′ Pin
00-FF	Trap	N/A	No	The INT instruction is actually a trap.

Note:

Co-processor not available may be due to not having one, or because the content of the co-processor belongs to another thread. The co-processor data needs to be saved and restored only when more than one thread is using it.
Bit 3 in CR0 indicates that a thread switch has occurred and will cause a trap 7 when a co-processor instruction is decoded.

Explanation of B/C column

B - Benign, means it is ok with any other exception.

C - Contributary, means it will contribute to a double fault.

PF - Page Fault, means a referenced address is not present.

1.10.1 Definition of Fault, Trap, Aborts and Interrupts

1. Faults

   CS and EIP point to the instruction which generated the fault.

2. Traps

   CS and EIP point to the instruction to be executed after the instruction which caused the trap.

INT3, INTO, BOUND, and INT nn are examples of traps.

3. Aborts

   In general, these exceptions do not permit locating the failing instruction, nor restart of the thread which caused the abort. Aborts are used to report inconsistent or illegal values in system tables, and hardware errors.

4. Interrupts

   Unlike the preceding exceptions, interrupts are not related to the program being executed, but to an external condition.

1.10.2 Hardware Error Codes

Selector Related Error Code

```
Bits 31-15: Reserved.
Bits 15-03: The index part of the selector involved.
Bit     02: The table indicator bit,
            if neither bit 01 nor bit 00 are 1.
Bit     01: IDT selector bit,
            if on, the selector is in the IDT.
Bit     00: External bit,
            if on, not caused by the program
```

Page Fault Error Code

```
Bits 31-04: Reserved.
Bit  03: RSV. A 1 bit was detected in a reserved
         bit of a page directory or page table entry.
Bit  02: U/S.
         0: The program was in supervisor mode.
         1: The program was in user mode.
Bit  01: W/R.
         0: The access was a read.
         1: The access was a write.
Bit  00: Level.
         0: The fault is because of a not-present page.
         1: The fault is because of page-level protection.
```

1.10.3 Simultaneous Exceptions

It is possible for more than one exception to occur while attempting to execute an instruction. In order to determine what will happen if two simultaneous exceptions occur on the same instruction, use the following table:

First Exception	Second Exception	Resulting Action
Benign	Benign	OK
Benign	Contributory	OK
Benign	Page Fault	OK
Contributory	Benign	OK
Contributory	Contributory	Double Fault
Contributory	Page Fault	OK
Page Fault	Benign	OK
Page Fault	Contributory	Double Fault
Page Fault	Page Fault	Double Fault

Note:

OK means the faults are processed consecutively.

Double Fault means the faults are reported together.

If any other exception occurs trying to enter the DoubleFault handler, the processor shuts down until RESET; although, if the NMI handler has not been entered, NMI will be recognized and accepted.

A trap C in Ring 0 is usually a double fault.

When the processor detects a Stack Exception it needs to push an error code and a return address onto the stack of the exception handler. If this happens in Ring 0, there will be no privilege level transition, which includes switching to a new, protected stack. If the exception is due to stack growth, there is no place to push the error code or return address.

RESULT: TRAP 8

Chapter 2. The Address Space Picture

This is a picture of what the address space looks like for several processes.

```
FFFFFFFF  ┌──────────────────────────────────────────────────┐
          │                    KERNEL                        │
          │          OS2 SYSTEM CONTROL BLOCKS               │
          │                                                  │
          │                SYSTEM REGION                     │
          │                                                  │
          │                                                  │
20000000  │                                                  │
          │--------------------------------------------------│
1FFFFFFF  │                                                  │
          │                                                  │
          │                                                  │
          │               SHARED REGION                      │
          │                                                  │
13000000  │                                                  │
          │- - - - - - - - - - - - - - - - - - - - - - - - - │
          │                                                  │
          │              EXPANSION REGION                    │
          │           ( SHARED and PRIVATE )                 │
04000000  │                                                  │
          │- - - - - - - - - - - - - - - - - - - - - - - - - │
03FFFFFF  │         │         │         │          │         │
          │  Pid 1  │         │         │  Pid 18  │         │
PRIVATE   │         │  Pid 2  │         │          │         │
 REGION   │         │         │         │          │  Pid 12 │
          │         │         │  Pid 3  │          │         │
          │         │         │         │          │         │
00010000  ├──────────────────────────────────────────────────┤
          │   I N V A L I D    A D D R E S S    R A N G E    │
00000000  └──────────────────────────────────────────────────┘
```

Within the private region you must know the Process ID, as well as the linear address to define a piece of virtual storage. All regions except the private region are shared among all processes. Above the private region in the shared regions, there is only one version of a given address, so you *do not* need the Process ID.

The boundary at 03FFFFFF is an initial value. If some application allocates over 03FFFFFF of private space, this boundary will move upward. It moves in steps of 00400000, because another page table is allocated.

DLL′s are initially loaded beginning at the 1BFFFFFF boundary, and to successively lower addresses. This water mark moves downward in steps of 00400000, too.

Addresses not assigned to a memory object are invalid. Any attempt to use them will generate an exception.

The address space picture discussed here is a simplified overview. A more detailed description may be found in the Advanced Guide to Hang Analysis chapter, under Memory Management and Ownership Topics.

Chapter 3. OS/2 Implementation Details

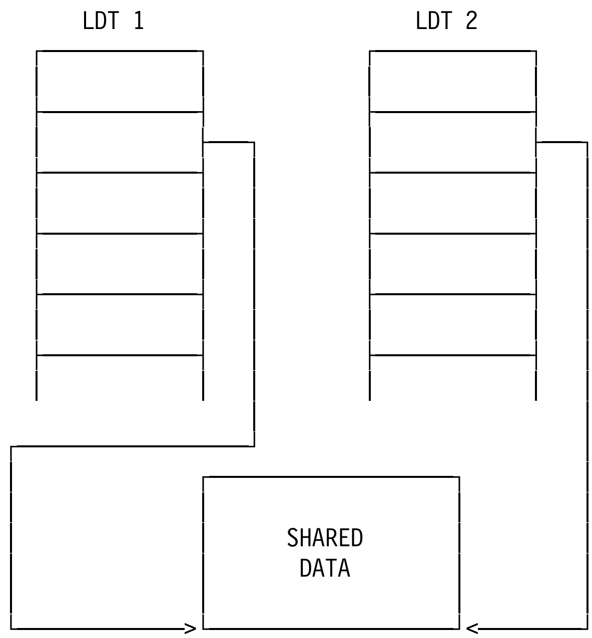
This section discusses some of the implementation details of OS/2 which particularly involve debugging.

3.1 Shared Memory

This highlights how memory is shared among a few processes.

The same selector is allocated in each process that shares the storage. Each process therefore uses the same offset in the LDT, and the LDT entries are the same, so the same linear address is also used.

Note: The page table entries used for the shared storage are the same for both processes, too.



DLLs are a good example of shared storage.

DLLs are loaded into the shared address range. The boundary is dynamic, and moves downward as DLLs are loaded.

The boundary of private addresses move upward as private storage is allocated. There is a guarantee of 64 Meg for private, and 64 Meg for shared.

3.2 Address Tiling

Address tiling refers to the practice of creating a mathematical or algorithmic relationship between an LDT selector and the base, or virtual address in the descriptor.

By using address tiling, OS/2 avoids the need to move memory blocks because of reallocation, and also makes it very fast to convert an LDT Selector:Offset to a flat, or Linear Address. The implementation is simply to allocate 64K of virtual address space to each selector, starting with selector 000F, at virtual address 64K or %10000.

Note: Selector 0007 is used to map the LDT as read-only data.

3.3 Why Thunk?

It is still common to have applications which have some 32-bit parts, and some 16-bit parts. The 32-bit parts try to avoid using 16-bit selector:offset addressing, because of the overhead of loading the selector registers, as well as to avoid the challenge of correctly dealing with storage references in both modes.

A typical example is a 32-bit application calling a 16-bit DLL.

Since storage is (must be) the same for all parts of a process, there has to be a way to convert one form of an address to the other.

Only 16-bit application selectors from the LDT are eligible for this quick form of the conversion, and only linear addresses less than %20000000 can be converted to 16:16 format.

Additionally, addresses in the packed region may *not* be converted by this quick method, but by a search of the LDT descriptor base (linear) addresses, followed by a calculation.

The top of normal application space, at %1BFFFFFF, is mapped to selector DFFF. The top of protected shared addresses at %1FFFFFFF maps to selector FFFF, if used.

3.4 Address Transformations (Thunks)

This section tells you how to change from 16:16 to 0:32-bit mode, or vice versa. There are two parts to thunking, the address transformation, and properly aligning the stack, if necessary. The stack alignment is usually done by a subroutine which detects the need to do this, and builds an extra frame in the new mode, properly aligned by making a copy of the incoming parms, transforming the addresses as part of this process.

This works only because the specific implementation within OS/2 which was designed to use address tiling for LDT selectors.

3.4.1 16:16 to 0:32 Thunk

The selectors which are eligible for this thunk are LDT selectors which are PL=3.
In this case, all three low-order bits are 1. Because of this, one can shift the selector 3 bits to the right, or divide by 8, without loss of information. The resulting number is the high-order word of the 32-bit address because of address tiling. For example, address 000F:00BA can be thunked from 16:16 to 0:32 as follows:

```
0    0    0    F    :        0    0    B    A    <--- Hex Sel:Offset
0000 0000 0000 1111          0000 0000 1011 1010 <--- Binary
shift the selector 3 bits to the right, which gives
0000 0000 0000 0001          0000 0000 1011 1010 <--- Binary
0    0    0    1             0    0    B    A    <--- Linear Address
Note that the lower 16 bits, or offset, are unchanged.
```

A stack may require alignment, because a 32-bit stack is built on doubleword boundaries, with two low order zero bits in the address of each element, whereas a 16-bit stack is aligned only on a word boundary.

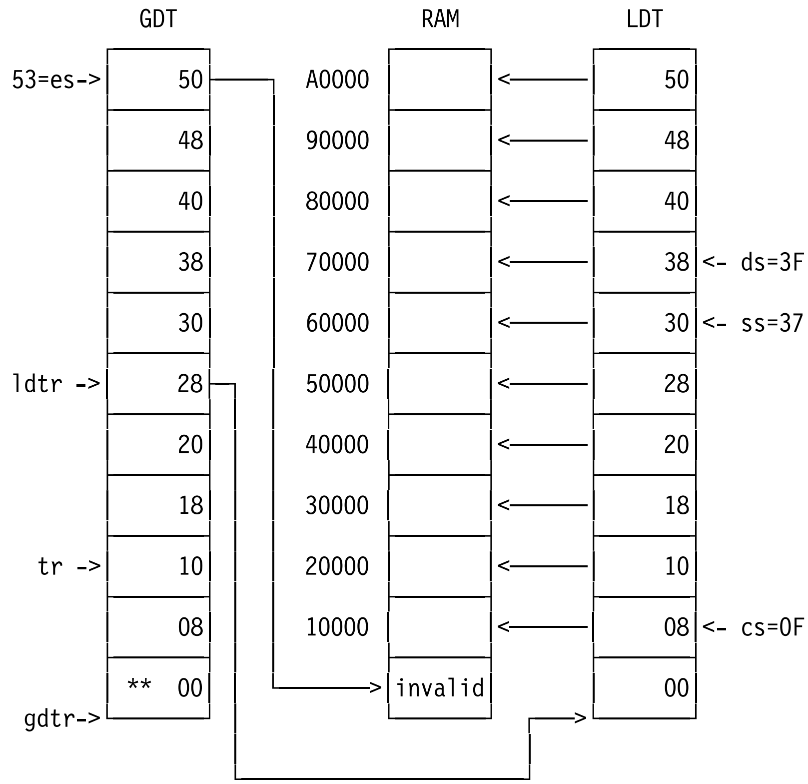
3.4.2 0:32 to 16:16 Thunk

Because the range of LDT selectors is only 512 Meg, addresses less than this can be transformed to use an LDT selector, with restrictions. The transformation is to append three low-order 1 bits to the value, and to discard three high order zero bits. An alternative way of stating this is to multiply by 8, then add 7. The three low order one bits are LDT (table indicator=1) and PL=3. The restrictions are that the storage must be PL=3 application storage, must not span a 64K boundary in the linear address space, and the value must be less than hex 2000 0000.

```
0    0    0    2             1    4    B    0    <--- Linear Address
0000 0000 0000 0010          0001 0100 1011 0000 <--- Binary
shift the left half 3 bits to the left, which gives
0000 0000 0001 0000          0001 0100 1011 0000 <--- Binary
add 7 to the left half ( 0111 binary )
0000 0000 0001 0111          0001 0100 1011 0000 <--- Binary
0    0    1    7    :        1    4    B    0    <--- Hex Sel:Offset

Note that the lower 16 bits, or offset, are unchanged.
```

3.4.3 Simultaneous 16-Bit and 32-Bit Descriptions of Virtual Storage



** useless, null selector

The digits within the tables are the offsets to each descriptor. The selector values (CS=0F) indicate which selector normally accesses the descriptor.

Any selector containing the value 0-3 is the NULL selector which *does not* specify the first entry in the GDT. It is a place holder when a selector does not specify a descriptor. Any attempt to use the null selector results in a general protection exception.

The descriptors in the LDT are 16-bit descriptors. This is one of the reasons that 16-bit programs still execute and fail in exactly the same manner as on previous versions of OS/2.

Chapter 4. Stacks

This section describes how most OS/2 programs use the stack.

Understanding the stack is generally straightforward. The stack is defined by the descriptor selected by the Stack Selector register or SS, and the stack pointer or SP. Stacks are always read/write. There are two basic operations on a stack, PUSH and POP. PUSH decrements the stack pointer and then stores the operand at the offset provided by SP in the stack segment. POP moves the data item at the offset provided by SP to the operand and then increments SP. SP always points to the last item PUSHed. Stacks grow downward from higher addresses to lower addresses.

4.1 Near CALL and RETurn

The near CALL instruction is used to invoke a subroutine. The instruction first pushes IP into the stack and then updates IP so that it contains the offset of the first instruction in the subroutine.

The near form of the RETurn instruction is really just a POP IP, which restores the saved content of IP. Execution continues at the instruction following the CALL.

4.2 Far CALL and RETurn

The far CALL instruction is used to invoke a subroutine. The instruction first pushes CS into the stack, and then pushes IP. Next, it updates CS and IP so that they contain the selector:offset of the first instruction in the subroutine.

The far form of the RETurn instruction first pops IP, which restores the saved content of IP, and then pops CS, restoring it as well. Execution continues at the instruction following the CALL.

4.3 Passing Parameters

Parameters are generally passed to a subroutine by putting them on the stack with PUSH instructions prior to the CALL. Parameters are removed from the stack in one of two ways:

- By the caller (C convention), generally by adding a constant to SP.
- By the subroutine (PASCAL convention), by specifying the operand for the RETurn which is added to SP after the return address is POP′ed.

Note: C convention PUSHes parameters from right to left.
PASCAL convention PUSHes parameters left to right.

Because the NEAR versions of jump (JMP), CALL and RETurn DO NOT touch CS, there can be no change of privilege level during execution of any of them. The FAR versions of them do provide a new value for CS. If the new CS is the same privilege level as the current level, the only change from above is that CALL PUSHes CS before PUSHing IP. RETurn POPs IP before POPing CS.

4.4 Receiving Parameters

There is a register which can be used by a subroutine to access parameters very efficiently. This register is the Base Pointer. When it is used to obtain an offset, the default segment is the STACK segment. If the entry to a subroutine begins with these instructions the stack will look like the picture on the next page.

PUSH BP

MOV BP,SP

SUB SP,sizeof(LOCAL DATA ITEMS)

This sequence is so common that there is a single instruction equivalent:

ENTER sizeof(LOCAL DATA ITEMS), 0

This allows all parameters to be accessed as BP plus the appropriate offset and local data elements to be accessed as BP minus the appropriate offset.

The instructions to exit are either:

MOV SP,BP

POP BP

RET

or:

LEAVE

RET

4.5 Why do we Care About the Pascal Convention?

The Pascal convention was used by OS/2 1.x for those calls that access system functions that are implemented in a higher privilege level (ring) than the application. It is also used to call 16-bit Window Procedures.

Two examples are DosAllocSeg and DosRead. The decision was made to use the Pascal convention because of the way the hardware protects access to instructions and storage which is more privileged. This type of interface, including hardware operation, is discussed in detail after basic stack operation has been discussed.

4.6 Single Stack Frame

```
                 STACK
           ┌─────────────────┐ ( higher addresses )
           │   previous bp   │<──up here ──────┐
           ├─────────────────┤                 │
           │                 │                 │
           ├─────────────────┤                 │
           │                 │                 │
           ├─────────────────┤                 │
           │    parameter    │                 │
           ├─────────────────┤                 │
           │    parameter    │                 │
           ├─────────────────┤                 │
           │     return      │                 │
           ├─────────────────┤                 │
 bp ───>   │   caller's bp   │ ( caller's bp points)
           ├─────────────────┤
           │   local data    │
           ├─────────────────┤
           │   local data    │
           ├─────────────────┤
 sp ───>   │   local data    │  ( lower addresses )
           └─────────────────┘
```

Note: A stack grows downward (expand down).

When this convention is followed the stack can be viewed as a series of stack frames. Each stack frame has parameters and local data for some routine and linkage to the stack frames used by the caller of that routine, etc. The saved BP values create a linked list in the stack segment which has all the information about each call including the return address. The process of following the chain back is referred to as unwinding the stack and is an important aid to diagnosis when working on a problem.

4.7 An Example of Using the Stack

This is a trivial example of how to pass and receive parameters, which is used to document where the stack pointer and base pointer are at the end of each instruction.

The example is 32-bit non-optimized code.

The subroutine, SUB, is designed to return the difference obtained by subtracting the second parameter from the first.

First, the relevant C code:

```
(  main  )                ( sub )
    .
z=sub(A,B);               int sub(int x, int y)
    .                       {
    .                       return x-y;
    .                       }
```

Next, the assembler code

```
    .          ( i ) initial condition
  PUSH B     ; (01)            SUB:    PUSH EBP        ; (04)
  PUSH A     ; (02)                    MOV  EBP,ESP    ; (05)
  CALL SUB   ; (03)                    SUB  ESP,nn     ; (06)
  ADD  ESP,8 ; (12)                     .
  MOV  Z,EAX ; ( f ) final condition    .   ( NOTE )
                                        .
                                       MOV  EAX,[EBP+8] ; (07)
                                       SUB  EAX,[EBP+12] ; (08)
                                        .
                                       MOV  ESP,EBP       (09)
                                       POP  EBP           (10)
                                       RET                (11)
```

Note: At this point, the stack frame is established. If another, lower-level routine is called, the code to do so will look like the code seen in main, and a new stack frame will be established by that routine as soon as it receives control.

The new frame will be just below the current one.

4.8 Stack Example

This example shows a stack, with ESP on the left, and EBP on the right.

```
   ESP                                        EBP
               ( higher addresses )        ( i,10 )
              ________________________
 ( i,12 )    |           ?            |
             |________________________|
 ( 1 )       |           B            |
             |________________________|
 ( 2,11 )    |           A            |
             |________________________|
 ( 3,10 )    |     address of ADD     |
             |________________________|
 ( 4,9 )     |  EBP ( used by main )  |     ( 5 )
    .______  |________________________|
    .        |       automatic        |
    .        |________________________|
nn bytes     |          data          |
    .        |________________________|
    .        |       for 'sub'        |
    .        |________________________|
 ( 6 )       |      ( if used )       |
    _______  |________________________|
             |                        |  <-- where parameters for the
             |________________________|      next subroutine are put
               ( lower addresses )
```

Note: The numbers in parentheses indicate where the register points immediately after the numbered instruction on the previous page completes.

4.9 Multiple Stack Frames

```
                STACK SEGMENT
         +-----------------------+  high addresses
         |                       |
         |                       |
         +-----------------------+  ---------------
         |                       |
         |---PARMS FROM ASTART---|
         |---RETURN TO ASTART----|    MAIN's  FRAME
         |-ASTART'S BASE POINTER-|
         |                       |
         +-----------------------+  ---------------
         |                       |
         |---PARMS FROM MAIN-----|
         |---RETURN TO MAIN------|    SUB 1 FRAME
         |---MAIN's BASE POINTER-|
         |                       |
         +-----------------------+  ---------------
         |                       |
         |---PARMS FROM SUB 1----|
         |---RETURN TO SUB 1-----|    SUB 2 FRAME
         |---SUB 1 BASE POINTER--|
         |                       |
         +-----------------------+  ---------------
         |                       |
         |---PARMS FROM SUB 2----|
         |---RETURN TO SUB 2-----|
 BP -->  |---SUB 2 BASE POINTER--|    SUB 3 FRAME
         |                       |
         +-----------------------+  ---------------
         |                       |
 SP -->  +-----------------------+  low addresses
```

4.10 A Stack From a Dump

In this example you could use DUMP01.DMP in the DUMPS.162 subdirectory.

```
DW SS:BP L10
001f:000014e6  1550 00f1 000f 02e8 18ae 0000 0000 0000
001f:000014f6  0000 0000 0000 0000 0000 0000 0000 0000
```

The first word, which is addressed by the current value in the BP register, is the near address of the next stack frame, 1550.

The next two words are a far return address, with the offset to the left of the selector. The return is to address F:F1.

The words following the return address are the parameters, if any were passed. There is no direct way to tell from the stack how many parameters were passed, or expected. To see the next frame,

```
DW 1550 L 10
001f:00001550  0000 0300 000f 0001 1560 001f 156e 001f
001f:00001560  1568 001f 0000 0000 4544 4f4d 0000 15c6
```

In this stack frame, the BP chain pointer is zero. This usually means that you have found all of the frames on this stack.

The return address for this frame is F:300. The parameters seem to be an integer, 1, and three far addresses, 1F:1560, 1F:156e, and 1F:1568. A little further inspection shows that the third address 1F:1568 is pointed to by the first, which is highly unusual. Actually, this is the stack frame received by 'main'. Main's parameters are as follows:

1. An integer, which tells it how many strings were found on the command line
2. The far address of a list of addresses, each of which points to one of the strings
3. The far address of a second list of addresses, each of which points to an environment variable. This list is terminated with a NULL POINTER, a far address in which both the selector and offset are zero.

Let's look at them.
1F:1560 has the address 1f:1568. Near addresses default to the last selector used, so we are not required to supply it every time.

```
DA 1568
#001f:00001568 DEMO
```

Right, the name of the program was the first string on the command line. The first parameter indicates that there is only one string.

Let's look at a few of the environment variables.

```
DW 156E L8
001f:0000156e  15c6 001f 15da 001f 1608 001f 161f 001f
```

This gets us four far addresses. To see them all with only one input line, use the semicolon as a command delimiter and type away.

```
DA 15C6;DA 15DA;DA1608;DA 161F
001f:000015c6 WP_OBJHANDLE=132739
001f:000015da AUTOSTART=PROGRAMS,TASKLIST,FOLDERS,LAUNCHPAD
001f:00001608 BOOKSHELF=C:\OS2\BOOK;
001f:0000161f COMSPEC=C:\OS2\CMD.EXE
```

Notice that the tools are not very particular about spaces in the commands.

Lastly, to see the local data for the failing routine,

```
DW SS:SP BP-2
001f:000014be  02e8 18ae 00e3 2910 0017 0060 0017 0000
001f:000014ce  0004 0017 c8cf 0000 0030 c949 c85a c8cf
001f:000014de  0002 0000 00e6 1488
```

and now you have it, displayed above.

Chapter 5. Application Documentation

We will briefly discuss what files are optionally generated by most compilers, and how to tell the linker to create the MAP file. After an explanation of the contents, and why some of the numbers are what they are, we will answer some questions using various parts of the optional application documentation.

5.1 The .MAP File

When you look at a 16-bit MAP file, you will discover that it may have at least three sections. A 32-bit MAP file can have at least four.

1. The first section is built in the same sequence as the executable.
2. The second section contains a list of all external symbols, sorted by the name of the symbol.

   This is particularly useful when a programmer wants to find where some particular variable or routine is located.
3. The third section contains a list of the same symbols, sorted by the location of the symbol.

   This is particularly useful when you know where something is, and want to find out if it has an external name, or what routine encompasses the address of interest.
4. The fourth section of a 32-bit MAP file contains a list of locations where the compiled code for each input line begins. This can tell you almost immediately which line of code failed, once you know which program, and where within the program the failing instruction was located.

5.2 The .COD File

Many 16-bit compilers will produce a file similar to a .COD file, although it may have a different file extension. For example, MicroFocus Cobol can produce a .GRP file, which has the same organization as the .COD file.

The format of this file is that of a mixed listing.

The listing generally contains an input line, identified by line number, followed by the machine instructions generated by the compiler, with the address to the left, the hex instruction in the middle, and an assembler form of the instruction on the right. Some of these files will actually be accepted as is by an assembler, but most compilers document the fact that this is not a supported feature of the compiler.

Obviously, if you know the offset of some instruction, perhaps one that caused a failure, you can use this listing to identify which line of the input program caused the generation of the failing instruction.

5.3 Exercise 4: Application Documentation

Some typical files associated with 16- and 32-bit applications

5.3.1 A 16-Bit Map File

Part 1: Same sequence as executable.

```
DEMO

Start     Length     Name                   Class
0001:0000 00292H     DEMO_TEXT              CODE
0001:0292 02BE6H     _TEXT                  CODE
0001:2E78 00000H     C_ETEXT                ENDCODE
0002:0000 02910H     FAR_BSS                FAR_BSS
0003:0000 00042H     NULL                   BEGDATA
0003:0042 007D8H     _DATA                  DATA
0003:081A 0000EH     CDATA                  DATA
0003:0828 00000H     XIFB                   DATA
0003:0828 00000H     XIF                    DATA
0003:0828 00000H     XIFE                   DATA
0003:0828 00000H     XIB                    DATA
0003:0828 00000H     XI                     DATA
0003:0828 00000H     XIE                    DATA
0003:0828 00000H     XPB                    DATA
0003:0828 00004H     XP                     DATA
0003:082C 00000H     XPE                    DATA
0003:082C 00000H     XCB                    DATA
0003:082C 00000H     XC                     DATA
0003:082C 00000H     XCE                    DATA
0003:082C 00000H     XCFB                   DATA
0003:082C 00000H     XCF                    DATA
0003:082C 00000H     XCFE                   DATA
0003:082C 00006H     CONST                  CONST
0003:0832 00008H     HDR                    MSG
0003:083A 000FAH     MSG                    MSG
0003:0934 00002H     PAD                    MSG
0003:0936 00001H     EPAD                   MSG
0003:0938 00226H     _BSS                   BSS
0003:0B5E 00000H     XOB                    BSS
0003:0B5E 00000H     XO                     BSS
0003:0B5E 00000H     XOE                    BSS
0003:0B60 00000H     c_common               BSS
0003:0B60 00A00H     STACK                  STACK

Origin   Group
0003:0   DGROUP
```

Note: The numbers to the left of the colon look like the selector part of a far address, because that is what they will become. The linker has no idea what selectors will be assigned by the loader, so it simply calls the first segment 1, the next segment 2, and so on.

The loader actually builds a table that shows the relationship between the selector assigned and the segment number from the map.

Part 2: Sorted by the name of the symbol

```
 Address         Publics by Name

0000:0000   Imp  DOSALLOCSEG          (DOSCALLS.34)
0000:0000   Imp  DOSCHGFILEPTR        (DOSCALLS.58)
0000:0000   Imp  DOSEXIT              (DOSCALLS.5)
0000:0000   Imp  DOSGETDBCSEV         (NLS.4)
0000:0000   Imp  DOSGETMACHINEMODE    (DOSCALLS.49)
0000:0000   Imp  DOSGETVERSION        (DOSCALLS.92)
0000:0000   Imp  DOSQHANDTYPE         (DOSCALLS.77)
0000:0000   Imp  DOSREAD              (DOSCALLS.137)
0000:0000   Imp  DOSREALLOCSEG        (DOSCALLS.38)
0000:0000   Imp  DOSSETVEC            (DOSCALLS.89)
0000:0000   Imp  DOSWRITE             (DOSCALLS.138)
0003:06E6        STKHQQ
0001:2D3E        _brkctl
0003:0938        _edata
0003:0B60        _end
0003:069B        _environ
0003:0662        _errno
0001:057A        _exit
0001:24E6        _fflush
0001:03F0        _fgets
0001:295C        _flushall
0001:275C        _free
0001:0000        _gen
0001:2836        _isatty
0001:29A0        _lseek
0001:00E2        _main
0001:2771        _malloc
0001:285A        _memset
0002:0000        _prime
0001:0394        _printf
0001:2618        _read
0001:0492        _sscanf
0001:2E64        _stackavail
0001:2024        _strlen
0001:282C        _ultoa
0001:2576        _ungetc
0001:29DE        _write
0003:06E2        __aaltstkovr
0003:04D6        __abrkp
0003:00D6        __abrktb
0003:04D6        __abrktbe
0003:04D8        __acfinfo
0003:00CC        __acmdln
```

```
0000:9876  Abs   __acrtmsg
0000:9876  Abs   __acrtused
0000:D6D6  Abs   __aDBdoswp
0003:06A6        __adbgmsg
0000:D6D6  Abs   __aDBused
0003:00CE        __aenvseg
0003:00D4        __aexit_rtn
0001:2E58        __aFlshl
0001:28A2        __aFNalshl
0000:0000  Imp   __AHINCR                 (DOSCALLS.136)
0001:2BDD        __amalloc
0001:2D1C        __amallocbrk
0003:0816        __amblksiz
0001:2CC0        __amexpand
0001:2CFA        __amlink
0001:0310        __amsg_exit
0003:0042        __anullsize
0003:0810        __aseg1
0003:0806        __asegds
0003:04E6        __aseghi
0003:04E8        __aseglo
0003:0812        __asegn
0003:0814        __asegr
0003:00D0        __asizds
0003:0702        __asizeC
0003:0703        __asizeD
0001:02A2        __astart
0003:00D2        __atopsp
0002:2710        __bufin
0003:06EA        __cfltcvt_tab
0003:06E8        __cflush
0003:06A3        __child
0001:2244        __chkstk
0001:04F0        __cinit
0001:0306        __cintDIV
0001:2DF4        __cltoasub
0001:05CA        __ctermsub
0003:0704        __ctype
0003:0704        __ctype_
0001:2E01        __cxtoa
0003:0669        __doserrno
0003:0668        __dosmode
0001:291F        __dosret
0001:2910        __dosretf
0003:0666        __dosvermajor
0003:0667        __dosverminor
```

```
0001:0591        __exit
0003:065A        __fac
0001:275C        __ffree
0001:05EC        __FF_MSGBANNER
0001:0702        __filbuf
0001:22D0        __flsbuf
0001:2771        __fmalloc
0003:081C        __fpinit
0001:223E        __fptrap
0001:08E0        __ftbuf
0001:2458        __getbuf
0001:098C        __input
0003:04EE        __iob
0003:05DE        __iob2
0003:0656        __lastiob
0003:066B        __nfile
0001:2B82        __nfree
0001:2B94        __nmalloc
0001:069C        __NMSG_TEXT
0001:06CC        __NMSG_WRITE
0001:2268        __nullcheck
0003:0669        __oserr
0003:066D        __osfile
0003:0666        __osmajor
0003:0667        __osminor
0003:0668        __osmode
0003:0666        __osversion
0001:156A        __output
0003:069F        __pgmptr
0003:0681        __pipe
0001:203C        __setargv
0001:0610        __setenvp
0003:0700        __sigintoff
0003:06FE        __sigintseg
0001:07FE        __stbuf
0001:228E        __stdalloc
0003:06AE        __stdbuf
0003:0664        __umaskval
0003:04EC        ___aDBrterr
0003:04EA        ___aDBswpflg
0003:0695        ___argc
0003:0697        ___argv
```

Part 3: Sorted by location in storage

```
 Address         Publics by Value

0000:0000  Imp  DOSGETMACHINEMODE    (DOSCALLS.49)
0000:0000  Imp  DOSGETVERSION        (DOSCALLS.92)
0000:0000  Imp  DOSREAD              (DOSCALLS.137)
0000:0000  Imp  __AHINCR             (DOSCALLS.136)
0000:0000  Imp  DOSEXIT              (DOSCALLS.5)
0000:0000  Imp  DOSALLOCSEG          (DOSCALLS.34)
0000:0000  Imp  DOSREALLOCSEG        (DOSCALLS.38)
0000:0000  Imp  DOSCHGFILEPTR        (DOSCALLS.58)
0000:0000  Imp  DOSWRITE             (DOSCALLS.138)
0000:0000  Imp  DOSSETVEC            (DOSCALLS.89)
0000:0000  Imp  DOSQHANDTYPE         (DOSCALLS.77)
0000:0000  Imp  DOSGETDBCSEV         (NLS.4)
0000:9876  Abs  __acrtmsg
0000:9876  Abs  __acrtused
0000:D6D6  Abs  __aDBdoswp
0000:D6D6  Abs  __aDBused
0001:0000       _gen
0001:00E2       _main
0001:02A2       __astart
0001:0306       __cintDIV
0001:0310       __amsg_exit
0001:0394       _printf
0001:03F0       _fgets
0001:0492       _sscanf
0001:04F0       __cinit
0001:057A       _exit
0001:0591       __exit
0001:05CA       __ctermsub
0001:05EC       __FF_MSGBANNER
0001:0610       __setenvp
0001:069C       __NMSG_TEXT
0001:06CC       __NMSG_WRITE
0001:0702       __filbuf
0001:07FE       __stbuf
0001:08E0       __ftbuf
0001:098C       __input
0001:156A       __output
0001:2024       _strlen
0001:203C       __setargv
0001:223E       __fptrap
0001:2244       __chkstk
0001:2268       __nullcheck
0001:228E       __stdalloc
0001:22D0       __flsbuf
0001:2458       __getbuf
0001:24E6       _fflush
0001:2576       _ungetc
0001:2618       _read
0001:275C       _free
0001:275C       __ffree
```

```
0001:2771        __fmalloc
0001:2771        _malloc
0001:282C        _ultoa
0001:2836        _isatty
0001:285A        _memset
0001:28A2        __aFNalshl
0001:2910        __dosretf
0001:291F        __dosret
0001:295C        _flushall
0001:29A0        _lseek
0001:29DE        _write
0001:2B82        __nfree
0001:2B94        __nmalloc
0001:2BDD        __amalloc
0001:2CC0        __amexpand
0001:2CFA        __amlink
0001:2D1C        __amallocbrk
0001:2D3E        _brkctl
0001:2DF4        __cltoasub
0001:2E01        __cxtoa
0001:2E58        __aFlshl
0001:2E64        _stackavail
0002:0000        _prime
0002:2710        __bufin
0003:0042        __anullsize
0003:00CC        __acmdln
0003:00CE        __aenvseg
0003:00D0        __asizds
0003:00D2        __atopsp
0003:00D4        __aexit_rtn
0003:00D6        __abrktb
0003:04D6        __abrktbe
0003:04D6        __abrkp
0003:04D8        __acfinfo
0003:04E6        __aseghi
0003:04E8        __aseglo
0003:04EA        ___aDBswpflg
0003:04EC        ___aDBrterr
```

```
0003:04EE        __iob
0003:05DE        __iob2
0003:0656        __lastiob
0003:065A        __fac
0003:0662        _errno
0003:0664        __umaskval
0003:0666        __osmajor
0003:0666        __dosvermajor
0003:0666        __osversion
0003:0667        __osminor
0003:0667        __dosverminor
0003:0668        __osmode
0003:0668        __dosmode
0003:0669        __doserrno
0003:0669        __oserr
0003:066B        __nfile
0003:066D        __osfile
0003:0681        __pipe
0003:0695        ___argc
0003:0697        ___argv
0003:069B        _environ
0003:069F        __pgmptr
0003:06A3        __child
0003:06A6        __adbgmsg
0003:06AE        __stdbuf
0003:06E2        __aaltstkovr
0003:06E6        STKHQQ
0003:06E8        __cflush
0003:06EA        __cfltcvt_tab
0003:06FE        __sigintseg
0003:0700        __sigintoff
0003:0702        __asizeC
0003:0703        __asizeD
0003:0704        __ctype
0003:0704        __ctype_
0003:0806        __asegds
0003:0810        __aseg1
0003:0812        __asegn
0003:0814        __asegr
0003:0816        __amblksiz
0003:081C        __fpinit
0003:0938        _edata
0003:0B60        _end

Program entry point at 0001:02A2
```

5.3.2 A 16-Bit Code File

```
;       Static Name Aliases
;
;       $S180_inbuf     EQU     inbuf
        TITLE   DEMO.C
        .286p
        .287
DEMO_TEXT       SEGMENT  WORD PUBLIC 'CODE'
DEMO_TEXT       ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
DGROUP  GROUP   CONST, _BSS, _DATA
        ASSUME  CS: DEMO_TEXT, DS: DGROUP, SS: DGROUP
EXTRN   __acrtused:ABS
EXTRN   _printf:FAR
EXTRN   _sscanf:FAR
EXTRN   _fgets:FAR
_BSS      SEGMENT
COMM NEAR       _prime:  2:      5000
_BSS      ENDS
EXTRN   __iob:BYTE
_DATA      SEGMENT
$SG188  DB      'there are %u primes less than 65536',  0aH,  00H
$SG191  DB      '%u',  00H
$SG195  DB      'Enter number to factor: ',  00H
$SG197  DB      '%u',  00H
$SG198  DB      'Unable to convert number. Please try again',  0aH,  00H
$SG207  DB      '%u is prime',  0aH,  00H
$SG208  DB      '%u=%u',  00H
$SG212  DB      '*%u',  00H
$SG213  DB      0aH,  00H
_DATA      ENDS
_BSS      SEGMENT
$S180_inbuf     DW 028H DUP (?)
_BSS      ENDS
CONST      SEGMENT
$T20004 DW SEG _prime
CONST      ENDS
DEMO_TEXT      SEGMENT
        ASSUME  CS: DEMO_TEXT
;|***
;|*** #include <stdio.h>
```

```
; Line 2
;|*** #define INBUFSIZE 80
;|*** #define NPRIME 5000
;|*** unsigned short prime[NPRIME];
;|***
;|*** int gen(void)
;|***   {
; Line 8
        PUBLIC  _gen
_gen    PROC FAR
        *** 000000      c8 24 00 00                 enter   WORD PTR 36,0
        *** 000004      57                          push    di
        *** 000005      56                          push    si
;       ix = -6
;       l = -16
;       ll = -14
;       npr = -2
;       q = -4
;       t = -10
;       tp = -8
;       tt = -12
;|***   unsigned short ix,l=2,ll=25,npr=3,q,t,tp=2,tt;
; Line 9
        *** 000006      c7 46 f0 02 00              mov     WORD PTR [bp-16],2      ;l
        *** 00000b      c7 46 f2 19 00              mov     WORD PTR [bp-14],25     ;ll
        *** 000010      c7 46 fe 03 00              mov     WORD PTR [bp-2],3       ;npr
        *** 000015      c7 46 f8 02 00              mov     WORD PTR [bp-8],2       ;tp
;|***   prime[0]=2;
; Line 10
        *** 00001a      8e 06 00 00                 mov     es,$T20004
        *** 00001e      26 c7 06 00 00 02 00        mov     WORD PTR es:_prime,2
;|***   prime[1]=3;
; Line 11
        *** 000025      26 c7 06 02 00 03 00        mov     WORD PTR es:_prime+2,3
;|***   prime[2]=5;
; Line 12
        *** 00002c      26 c7 06 04 00 05 00        mov     WORD PTR es:_prime+4,5
;|***   for ( t=7 ; t<65530 ; t+=tp )
; Line 13
        *** 000033      c7 46 f6 07 00              mov     WORD PTR [bp-10],7      ;t
        *** 000038      c7 46 e2 04 00              mov     WORD PTR [bp-30],OFFSET _prime+4
        *** 00003d      c7 46 e4 00 00              mov     WORD PTR [bp-28],SEG _prime
        *** 000042      c7 46 de 06 00              mov     WORD PTR [bp-34],OFFSET _prime+6
        *** 000047      c7 46 e0 00 00              mov     WORD PTR [bp-32],SEG _prime
                                            $L20002:
;|***     {
; Line 14
;|***     tp=6-tp;
; Line 15
        *** 00004c      b8 06 00                    mov     ax,6
        *** 00004f      2b 46 f8                    sub     ax,WORD PTR [bp-8]      ;tp
        *** 000052      89 46 f8                    mov     WORD PTR [bp-8],ax      ;tp
```

```
;|***     if ( ll<=t )
; Line 16
        *** 000055      8b 46 f6                mov     ax,WORD PTR [bp-10]     ;t
        *** 000058      39 46 f2                cmp     WORD PTR [bp-14],ax     ;ll
        *** 00005b      77 15                   ja      $I170
;|***      {
; Line 17
;|***      l++;
; Line 18
        *** 00005d      83 46 e2 02             add     WORD PTR [bp-30],2
        *** 000061      ff 46 f0                inc     WORD PTR [bp-16]        ;l
;|***      ll=prime[l]*prime[l];
; Line 19
        *** 000064      c4 5e e2                les     bx,DWORD PTR [bp-30]
        *** 000067      26 8b 07                mov     ax,WORD PTR es:[bx]
        *** 00006a      89 46 dc                mov     WORD PTR [bp-36],ax
        *** 00006d      f7 e0                   mul     ax
        *** 00006f      89 46 f2                mov     WORD PTR [bp-14],ax     ;ll
;|***      }
; Line 20
;|***     for ( ix=2 ; ix<l ; ix++ )
; Line 21
                                        $I170:
        *** 000072      be 02 00                mov     si,2
        *** 000075      39 76 f0                cmp     WORD PTR [bp-16],si     ;l
        *** 000078      76 39                   jbe     $FB173
        *** 00007a      8b 46 f6                mov     ax,WORD PTR [bp-10]     ;t
        *** 00007d      89 46 ec                mov     WORD PTR [bp-20],ax
        *** 000080      c7 46 ee 00 00          mov     WORD PTR [bp-18],0
        *** 000085      c7 46 e8 04 00          mov     WORD PTR [bp-24],OFFSET _prime+4
        *** 00008a      c7 46 ea 00 00          mov     WORD PTR [bp-22],SEG _prime
        *** 00008f      c4 7e e8                les     di,DWORD PTR [bp-24]
                                        $L20000:
;|***      {
; Line 22
;|***      q=t/prime[ix];
; Line 23
        *** 000092      26 8b 0d                mov     cx,WORD PTR es:[di]
        *** 000095      8b 46 ec                mov     ax,WORD PTR [bp-20]
        *** 000098      8b 56 ee                mov     dx,WORD PTR [bp-18]
        *** 00009b      f7 f1                   div     cx
        *** 00009d      89 46 fc                mov     WORD PTR [bp-4],ax      ;q
;|***      tt=q*prime[ix];
; Line 24
        *** 0000a0      f7 e1                   mul     cx
        *** 0000a2      89 46 f4                mov     WORD PTR [bp-12],ax     ;tt
;|***      if ( t==tt ) break;
; Line 25
        *** 0000a5      39 46 f6                cmp     WORD PTR [bp-10],ax     ;t
        *** 0000a8      74 09                   je      $FB173
        *** 0000aa      83 c7 02                add     di,2
        *** 0000ad      46                      inc     si
        *** 0000ae      39 76 f0                cmp     WORD PTR [bp-16],si     ;l
        *** 0000b1      77 df                   ja      $L20000
                                        $FB173:
;|***      }
```

```
;|***     if ( l==ix ) prime[npr++]=t;
; Line 27
        *** 0000b3      39 76 f0                cmp     WORD PTR [bp-16],si     ;l
        *** 0000b6      75 10                   jne     $I175
        *** 0000b8      c4 5e de                les     bx,DWORD PTR [bp-34]
        *** 0000bb      8b 46 f6                mov     ax,WORD PTR [bp-10]     ;t
        *** 0000be      26 89 07                mov     WORD PTR es:[bx],ax
        *** 0000c1      83 46 de 02             add     WORD PTR [bp-34],2
        *** 0000c5      ff 46 fe                inc     WORD PTR [bp-2] ;npr
;|***     }
; Line 28
                                        $I175:
        *** 0000c8      8b 46 f8                mov     ax,WORD PTR [bp-8]      ;tp
        *** 0000cb      01 46 f6                add     WORD PTR [bp-10],ax     ;t
        *** 0000ce      83 7e f6 fa             cmp     WORD PTR [bp-10],-6     ;t
        *** 0000d2      73 03                   jae     $JCC210
        *** 0000d4      e9 75 ff                jmp     $L20002
                                        $JCC210:
        *** 0000d7      89 76 fa                mov     WORD PTR [bp-6],si      ;ix
;|***   return npr;
; Line 29
        *** 0000da      8b 46 fe                mov     ax,WORD PTR [bp-2]      ;npr
        *** 0000dd      5e                      pop     si
        *** 0000de      5f                      pop     di
        *** 0000df      c9                      leave
        *** 0000e0      cb                      ret
        *** 0000e1      90                      nop

_gen    ENDP
;|***   }
;|***
;|*** int main(int argc, char *argv[])
;|***   {
; Line 33
        PUBLIC  _main
_main   PROC FAR
        *** 0000e2      c8 60 00 00             enter   WORD PTR 96,0
        *** 0000e6      57                      push    di
        *** 0000e7      56                      push    si
;       argc = 6
;       argv = 8
;       ix = -6
;       last = -10
;       nf = -8
;       fact = -76
;       input = -2
;       is = -12
;       q = -4
;|***   static char inbuf[INBUFSIZE];
;|***   int ix,last,nf;
;|***   unsigned short fact[32],input=0,is,q;
; Line 36
        *** 0000e8      c7 46 fe 00 00          mov     WORD PTR [bp-2],0       ;input
```

```
;|***   last=gen();
; Line 37
        *** 0000ed      0e                      push    cs
        *** 0000ee      e8 00 00                call    NEAR PTR _gen
        *** 0000f1      89 46 f6                mov     WORD PTR [bp-10],ax     ;last
;|***   printf("there are %u primes less than 65536\n",last);
; Line 38
        *** 0000f4      50                      push    ax
        *** 0000f5      1e                      push    ds
        *** 0000f6      68 00 00                push    OFFSET DGROUP:$SG188
        *** 0000f9      9a 00 00 00 00          call    FAR PTR _printf
        *** 0000fe      83 c4 06                add     sp,6
;|***   if ( 1<argc )
; Line 39
        *** 000101      83 7e 06 01             cmp     WORD PTR [bp+6],1       ;argc
        *** 000105      7e 25                   jle     $I190
;|***     if ( 0==sscanf(argv[1],"%u",&input) ) argc=1;
; Line 40
        *** 000107      8d 46 fe                lea     ax,WORD PTR [bp-2]      ;input
        *** 00010a      16                      push    ss
        *** 00010b      50                      push    ax
        *** 00010c      1e                      push    ds
        *** 00010d      68 25 00                push    OFFSET DGROUP:$SG191
        *** 000110      c4 5e 08                les     bx,DWORD PTR [bp+8]     ;argv
        *** 000113      26 ff 77 06             push    WORD PTR es:[bx+6]
        *** 000117      26 ff 77 04             push    WORD PTR es:[bx+4]
        *** 00011b      9a 00 00 00 00          call    FAR PTR _sscanf
        *** 000120      83 c4 0c                add     sp,12
        *** 000123      0b c0                   or      ax,ax
        *** 000125      75 05                   jne     $I190
        *** 000127      c7 46 06 01 00          mov     WORD PTR [bp+6],1       ;argc
;|***   while ( 2>argc )
; Line 41
                                        $I190:
        *** 00012c      83 7e 06 02             cmp     WORD PTR [bp+6],2       ;argc
        *** 000130      7d 4b                   jge     $FB194
        *** 000132      8b 76 06                mov     si,WORD PTR [bp+6]      ;argc
                                        $L20005:
;|***     {
; Line 42
;|***     printf("Enter number to factor: ");
; Line 43
        *** 000135      1e                      push    ds
        *** 000136      68 28 00                push    OFFSET DGROUP:$SG195
        *** 000139      9a 00 00 00 00          call    FAR PTR _printf
        *** 00013e      83 c4 04                add     sp,4
;|***     fgets(inbuf,INBUFSIZE,stdin);
; Line 44
        *** 000141      1e                      push    ds
        *** 000142      68 00 00                push    OFFSET __iob
        *** 000145      6a 50                   push    80
        *** 000147      1e                      push    ds
        *** 000148      68 00 00                push    OFFSET DGROUP:$S180_inbuf
        *** 00014b      9a 00 00 00 00          call    FAR PTR _fgets
        *** 000150      83 c4 0a                add     sp,10
```

```
;|***     if ( 0==sscanf(inbuf,"%u",&input) )
; Line 45
        *** 000153      8d 46 fe                lea     ax,WORD PTR [bp-2]      ;input
        *** 000156      16                      push    ss
        *** 000157      50                      push    ax
        *** 000158      1e                      push    ds
        *** 000159      68 41 00                push    OFFSET DGROUP:$SG197
        *** 00015c      1e                      push    ds
        *** 00015d      68 00 00                push    OFFSET DGROUP:$S180_inbuf
        *** 000160      9a 00 00 00 00          call    FAR PTR _sscanf
        *** 000165      83 c4 0c                add     sp,12
        *** 000168      0b c0                   or      ax,ax
        *** 00016a      75 11                   jne     $FB194
;|***       printf("Unable to convert number. Please try again\n");
; Line 46
        *** 00016c      1e                      push    ds
        *** 00016d      68 44 00                push    OFFSET DGROUP:$SG198
        *** 000170      9a 00 00 00 00          call    FAR PTR _printf
        *** 000175      83 c4 04                add     sp,4
;|***     else break;
;|***     }
; Line 48
        *** 000178      83 fe 02                cmp     si,2
        *** 00017b      7c b8                   jl      $L20005
                                        $FB194:
;|***   for ( ix=nf=0,is=input ; ix<last ; ix++ )
; Line 49
        *** 00017d      2b c0                   sub     ax,ax
        *** 00017f      89 46 f8                mov     WORD PTR [bp-8],ax      ;nf
        *** 000182      89 46 fa                mov     WORD PTR [bp-6],ax      ;ix
        *** 000185      8b 46 fe                mov     ax,WORD PTR [bp-2]      ;input
        *** 000188      89 46 f4                mov     WORD PTR [bp-12],ax     ;is
        *** 00018b      83 7e f6 00             cmp     WORD PTR [bp-10],0      ;last
        *** 00018f      7f 03                   jg      $JCC399
        *** 000191      e9 8e 00                jmp     $FB202
                                        $JCC399:
        *** 000194      8b 46 fa                mov     ax,WORD PTR [bp-6]      ;ix
        *** 000197      d1 e0                   shl     ax,1
        *** 000199      05 00 00                add     ax,OFFSET _prime
        *** 00019c      89 46 a6                mov     WORD PTR [bp-90],ax
        *** 00019f      c7 46 a8 00 00          mov     WORD PTR [bp-88],SEG _prime
        *** 0001a4      8d 46 b4                lea     ax,WORD PTR [bp-76]     ;fact
        *** 0001a7      89 46 a2                mov     WORD PTR [bp-94],ax
        *** 0001aa      8c 56 a4                mov     WORD PTR [bp-92],ss
        *** 0001ad      8b 46 f6                mov     ax,WORD PTR [bp-10]     ;last
        *** 0001b0      2b 46 fa                sub     ax,WORD PTR [bp-6]      ;ix
        *** 0001b3      89 46 a0                mov     WORD PTR [bp-96],ax
        *** 0001b6      01 46 fa                add     WORD PTR [bp-6],ax      ;ix
        *** 0001b9      8b 4e fe                mov     cx,WORD PTR [bp-2]      ;input
                                        $L20008:
;|***     {
; Line 50
```

```
;|***     q=input/prime[ix];
; Line 51
        *** 0001bc      c4 5e a6                les     bx,DWORD PTR [bp-90]
        *** 0001bf      26 8b 07                mov     ax,WORD PTR es:[bx]
        *** 0001c2      89 46 aa                mov     WORD PTR [bp-86],ax
        *** 0001c5      8b c1                   mov     ax,cx
        *** 0001c7      2b d2                   sub     dx,dx
        *** 0001c9      f7 76 aa                div     WORD PTR [bp-86]
        *** 0001cc      89 46 fc                mov     WORD PTR [bp-4],ax      ;q
;|***     while ( q*prime[ix]==input )
; Line 52
        *** 0001cf      8b 46 aa                mov     ax,WORD PTR [bp-86]
        *** 0001d2      f7 66 fc                mul     WORD PTR [bp-4] ;q
        *** 0001d5      3b c1                   cmp     ax,cx
        *** 0001d7      75 3d                   jne     $FB205
        *** 0001d9      26 8b 07                mov     ax,WORD PTR es:[bx]
        *** 0001dc      89 46 b2                mov     WORD PTR [bp-78],ax
        *** 0001df      8b f0                   mov     si,ax
        *** 0001e1      8b 46 a2                mov     ax,WORD PTR [bp-94]
        *** 0001e4      8b 56 a4                mov     dx,WORD PTR [bp-92]
        *** 0001e7      89 46 ac                mov     WORD PTR [bp-84],ax
        *** 0001ea      89 56 ae                mov     WORD PTR [bp-82],dx
        *** 0001ed      c4 7e ac                les     di,DWORD PTR [bp-84]
                                        $L20006:
;|***       {
; Line 53
;|***       fact[nf++]=prime[ix];
; Line 54
        *** 0001f0      26 89 35                mov     WORD PTR es:[di],si
        *** 0001f3      83 c7 02                add     di,2
        *** 0001f6      83 46 a2 02             add     WORD PTR [bp-94],2
        *** 0001fa      ff 46 f8                inc     WORD PTR [bp-8] ;nf
;|***       input/=prime[ix];
; Line 55
        *** 0001fd      8b c1                   mov     ax,cx
        *** 0001ff      2b d2                   sub     dx,dx
        *** 000201      f7 f6                   div     si
        *** 000203      8b c8                   mov     cx,ax
;|***       q=input/prime[ix];
; Line 56
        *** 000205      2b d2                   sub     dx,dx
        *** 000207      f7 f6                   div     si
        *** 000209      89 46 fc                mov     WORD PTR [bp-4],ax      ;q
;|***       }
```

```
; Line 57
        *** 00020c      8b 46 b2                mov     ax,WORD PTR [bp-78]
        *** 00020f      f7 66 fc                mul     WORD PTR [bp-4] ;q
        *** 000212      3b c1                   cmp     ax,cx
        *** 000214      74 da                   je      $L20006
                                        $FB205:
;|***     }
; Line 58
        *** 000216      83 46 a6 02             add     WORD PTR [bp-90],2
        *** 00021a      ff 4e a0                dec     WORD PTR [bp-96]
        *** 00021d      75 9d                   jne     $L20008
        *** 00021f      89 4e fe                mov     WORD PTR [bp-2],cx      ;input
                                        $FB202:
;|***   if ( nf<2 ) return printf("%u is prime\n",is);
; Line 59
        *** 000222      83 7e f8 02             cmp     WORD PTR [bp-8],2       ;nf
        *** 000226      7d 14                   jge     $I206
        *** 000228      ff 76 f4                push    WORD PTR [bp-12]        ;is
        *** 00022b      1e                      push    ds
        *** 00022c      68 70 00                push    OFFSET DGROUP:$SG207
        *** 00022f      9a 00 00 00 00          call    FAR PTR _printf
        *** 000234      83 c4 06                add     sp,6
        *** 000237      5e                      pop     si
        *** 000238      5f                      pop     di
        *** 000239      c9                      leave
        *** 00023a      cb                      ret
        *** 00023b      90                      nop
                                        $I206:
        *** 00023c      ff 76 b4                push    WORD PTR [bp-76]        ;fact
        *** 00023f      ff 76 f4                push    WORD PTR [bp-12]        ;is
        *** 000242      1e                      push    ds
        *** 000243      68 7d 00                push    OFFSET DGROUP:$SG208
        *** 000246      9a 00 00 00 00          call    FAR PTR _printf
        *** 00024b      83 c4 08                add     sp,8
;|***   printf("%u=%u",is,fact[0]);
```

```
;|***   for ( ix=1 ; ix<nf ; ix++ )
; Line 61
        *** 00024e      c7 46 fa 01 00          mov     WORD PTR [bp-6],1       ;ix
        *** 000253      83 7e f8 01             cmp     WORD PTR [bp-8],1       ;nf
        *** 000257      7e 29                   jle     $FB211
        *** 000259      8d 46 b6                lea     ax,WORD PTR [bp-74]
        *** 00025c      89 46 a2                mov     WORD PTR [bp-94],ax
        *** 00025f      8c 56 a4                mov     WORD PTR [bp-92],ss
        *** 000262      8b 76 f8                mov     si,WORD PTR [bp-8]      ;nf
        *** 000265      4e                      dec     si
        *** 000266      01 76 fa                add     WORD PTR [bp-6],si      ;ix
                                        $L20010:
;|***     printf("*%u",fact[ix]);
; Line 62
        *** 000269      c4 5e a2                les     bx,DWORD PTR [bp-94]
        *** 00026c      26 ff 37                push    WORD PTR es:[bx]
        *** 00026f      1e                      push    ds
        *** 000270      68 83 00                push    OFFSET DGROUP:$SG212
        *** 000273      9a 00 00 00 00          call    FAR PTR _printf
        *** 000278      83 c4 06                add     sp,6
        *** 00027b      83 46 a2 02             add     WORD PTR [bp-94],2
        *** 00027f      4e                      dec     si
        *** 000280      75 e7                   jne     $L20010
                                        $FB211:
;|***   return printf("\n");
; Line 63
        *** 000282      1e                      push    ds
        *** 000283      68 87 00                push    OFFSET DGROUP:$SG213
        *** 000286      9a 00 00 00 00          call    FAR PTR _printf
        *** 00028b      83 c4 04                add     sp,4
        *** 00028e      5e                      pop     si
        *** 00028f      5f                      pop     di
        *** 000290      c9                      leave
        *** 000291      cb                      ret

_main   ENDP
DEMO_TEXT       ENDS
END
;|***   }
```

5.3.3 Questions

Please answer the following questions, using the preceding listings:

1. How large is segment 2 of DEMO.EXE? _____________

2. What is the segment:offset of the 'fgets' routine? _____________

3. What is the segment:offset of the symbol 'fpinit'? _____________

4. Does DEMO.EXE call DosOpen? ______ How can you tell? _____________

5. Which routine begins at address 0001:29DE? _____________

6. How long is the routine named 'strlen'? _____________

7. Which routine contains address 0001:186A? _____________

8. How far into the routine is the previous address? _____________

9. What is the program′s entry point? _____________

10. What is the name of the routine which is the entry point? _____________

11. What instruction mnemonic is at offset 00C5 in DEMO.EXE? _____________

12. What variable is in AX when the return at 00E0 executes? _____________

13. Which line in DEMO.C generated the above return? _____________

14. Offset 0188 in DEMO.C is in which C function? _____________

15. What variable name is used by the instruction at 0055? _____________

16. What is the purpose of the instruction at offset 0234? _____________

    Note: In the .COD file, the numbers in the assembler instructions are decimal.

    The lines generated in the .COD file between offsets 00E7 and 00E8 tell you where the local variables are stored, relatively speaking.

17. To what are the numbers like 8, -10, -76, -12 relative? _____________

18. If a failure were to occur in routine ′gen′, what command would you use to display only the variable ′npr′? DW _____________

19. How would you display the variable ′t′? DW _____________

20. Is the variable ′tp′ in ′gen′ at the same location as the variable ′nf′ in main? Yes / No Explain. _________________________________

    ____

    ____

21. Check the offsets for ′gen′ and ′main′ between the .MAP and the .COD files. Do they match? _______ Why/why not? ________________ Will the offsets always behave this way? Yes / No

    Explain. ____

    ____

5.3.4 A 32-Bit Map File

```
DEMO

Start         Length     Name                    Class
0001:00000000 000001A68H CODE32                  CODE 32-bit
0001:00001A68 000000030H _MSGSEG32               CODE 32-bit
0002:00000000 00000006CH DDE4_DATA32             DATA 32-bit
0003:00000000 00000005CH DATA32                  DATA 32-bit
0003:0000005C 0000000B0H CONST32                 CONST 32-bit
0003:0000010C 000000000H BSS32                   BSS 32-bit
0003:00000110 000002000H STACK32                 STACK 32-bit

Origin   Group
0000:0   FLAT
0003:0   DGROUP

 Address         Publics by Name

0000:00000000  Imp  DOS32FLATTOSEL       (DOSCALLS.425)
0001:00001A7A       DOS32GETMESSAGE
0001:00001A7A       Dos32GetMessage
0000:00000000  Imp  DOS32IQUERYMESSAGECP (MSG.8)
0000:00000000  Imp  DOS32SELTOFLAT       (DOSCALLS.426)
0000:00000000  Imp  DOS32TRUEGETMESSAGE  (MSG.6)
0000:00000000  Imp  DosAllocMem          (DOSCALLS.299)
0000:00000000  Imp  DosExit              (DOSCALLS.234)
0000:00000000  Imp  DosFreeMem           (DOSCALLS.304)
0001:00001A7A       DosGetMessage
0001:00001A7A       DOSGETMESSAGE
0000:00000000  Imp  DosWrite             (DOSCALLS.282)
0001:00000F94       free
0001:00000000       gen
0001:000000AC       main
0001:000013F0       malloc
0001:00001A68       sig32
0002:00000008       _argc
0002:0000000C       _argv
0001:000001A4       _bufprint
0001:00001850       _DosFlatToSel
0001:00001848       _DosSelToFlat
0003:0000010C       _edata
0001:00000F48       _edcGetMessage
0003:00000110       _end
0002:00000004       _exeentry
0002:00000010       _have_freed
0001:00001010       _heapmin
0001:00001108       _heapmin_int
0001:000012E0       _ilog2
0001:0000162C       _MesgServ
0002:00000014       _pBucketArr
0001:000017FC       _PrintErrMsg
0001:0000017C       _printfieee
0001:00000154       _printf_ansi
```

```
0001:00001858        _setuparg
0001:00001304        _split_chunk
0001:00001A40        _terminate
0001:00001A50        _wfloatfmt
0001:000002BC        __dofmto
0001:000012E8        __isdigit

 Address         Publics by Value

0000:00000000  Imp  DosFreeMem           (DOSCALLS.304)
0000:00000000  Imp  DOS32IQUERYMESSAGECP (MSG.8)
0000:00000000  Imp  DOS32SELTOFLAT       (DOSCALLS.426)
0000:00000000  Imp  DosAllocMem          (DOSCALLS.299)
0000:00000000  Imp  DOS32TRUEGETMESSAGE  (MSG.6)
0000:00000000  Imp  DOS32FLATTOSEL       (DOSCALLS.425)
0000:00000000  Imp  DosWrite             (DOSCALLS.282)
0000:00000000  Imp  DosExit              (DOSCALLS.234)
0001:00000000       gen
0001:000000AC       main
0001:00000154       _printf_ansi
0001:0000017C       _printfieee
0001:000001A4       _bufprint
0001:000002BC       __dofmto
0001:00000F48       _edcGetMessage
0001:00000F94       free
0001:00001010       _heapmin
0001:00001108       _heapmin_int
0001:000012E0       _ilog2
0001:000012E8       __isdigit
0001:00001304       _split_chunk
0001:000013F0       malloc
0001:0000162C       _MesgServ
0001:000017FC       _PrintErrMsg
0001:00001848       _DosSelToFlat
0001:00001850       _DosFlatToSel
0001:00001858       _setuparg
0001:00001A40       _terminate
0001:00001A50       _wfloatfmt
0001:00001A68       sig32
0001:00001A7A       DOSGETMESSAGE
0001:00001A7A       Dos32GetMessage
0001:00001A7A       DOS32GETMESSAGE
0001:00001A7A       DosGetMessage
0002:00000004       _exeentry
0002:00000008       _argc
0002:0000000C       _argv
0002:00000010       _have_freed
0002:00000014       _pBucketArr
0003:0000010C       _edata
0003:00000110       _end
```

```
Line numbers for DEMO.obj(DEMO.C) segment CODE32

         Source       Src File      Flags        Seg:Offset
        Line Num       Index        (0X)            (0X)
        ---------     --------     ------       -------------
              9           1          00         0001:00000000
             11           1          00         0001:00000009
             12           1          00         0001:0000001e
             13           1          00         0001:00000024
             14           1          00         0001:0000002b
             15           1          00         0001:00000032
             17           1          00         0001:00000040
             18           1          00         0001:0000004b
             20           1          00         0001:00000050
             21           1          00         0001:00000051
             23           1          00         0001:0000005a
             27           1          00         0001:00000069
             28           1          00         0001:00000082
             29           1          00         0001:00000087
             30           1          00         0001:00000097
             31           1          00         0001:000000a4
             34           1          00         0001:000000ac
             39           1          00         0001:000000b2
             40           1          00         0001:000000c8
             41           1          00         0001:000000e5
             42           1          00         0001:000000f4
             43           1          00         0001:00000107
             45           1          00         0001:00000114
             46           1          00         0001:00000127
             47           1          00         0001:00000144
             48           1          00         0001:00000149

        Record Number of Start of Source: 9
        Number of Primary Source Records: 26
        Source and Listing Files:
        File 1)DEMO.C
        File 2)C:\PMG\CSET\INCLUDE\os2.h
        File 3)C:\PMG\CSET\INCLUDE\os2def.h
        File 4)C:\PMG\CSET\INCLUDE\bse.h
        File 5)C:\PMG\CSET\INCLUDE\bsedos.h
        File 6)C:\PMG\CSET\INCLUDE\bsememf.h
        File 7)C:\PMG\CSET\INCLUDE\bsesub.h
        File 8)C:\PMG\CSET\INCLUDE\bseerr.h
        File 9)C:\PMG\CSET\INCLUDE\STDIO.H

Program entry point at 0001:00000F6C
```

5.3.5 A 32-Bit .ASM File, Produced by CSET/2

```
        TITLE   DEMO.C
        .386
        .387
        INCLUDELIB OS2386.LIB
        INCLUDELIB dde4nbs.lib
CODE32  SEGMENT DWORD USE32 PUBLIC 'CODE'
CODE32  ENDS
DATA32  SEGMENT DWORD USE32 PUBLIC 'DATA'
DATA32  ENDS
CONST32 SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST32 ENDS
BSS32   SEGMENT DWORD USE32 PUBLIC 'BSS'
BSS32   ENDS
DGROUP  GROUP CONST32, BSS32, DATA32
        ASSUME  CS:FLAT, DS:FLAT, SS:FLAT, ES:FLAT
        EXTRN   DosAllocMem:PROC
        EXTRN   _printfieee:PROC
        EXTRN   _DosFlatToSel:PROC
        EXTRN   _DosSelToFlat:PROC
        EXTRN   _exeentry:PROC
DATA32  SEGMENT
@STAT1  DB "non-zero return code fro"
DB "m DosAllocMem=%d",0aH,0H
        ALIGN 04H
@STAT2  DB "there are %u primes less"
DB " than 65536",0aH,0H
        ALIGN 04H
@STAT3  DB "%6u ",0H
@STAT4  DB 0aH,0H
        DD      _exeentry
DATA32  ENDS
BSS32   SEGMENT
BSS32   ENDS
CONST32 SEGMENT
CONST32 ENDS
CODE32  SEGMENT

;***** 9 int gen(int *prime)
        ALIGN 04H

        PUBLIC gen
gen     PROC
        PUSH    EBP
        MOV     EBP,ESP
        PUSH    EBX
        PUSH    ESI
        PUSH    EDI
        MOV     [EBP+08H],EAX;  prime

;***** 11   int ix,l=2,ll=25,npr=3,q,t,tp=2,tt;
        MOV     DWORD PTR [EBP-018H],019H;      ll
        MOV     DWORD PTR [EBP-014H],03H;       npr
        MOV     DWORD PTR [EBP-010H],02H;       tp
```

```
;***** 12   prime[0]=2;
        MOV     DWORD PTR [EAX],02H
;***** 13   prime[1]=3;
        MOV     DWORD PTR [EAX+04H],03H
;***** 14   prime[2]=5;
        MOV     DWORD PTR [EAX+08H],05H
;***** 15   for ( t=7 ; t<65530 ; t+=tp )
        MOV     ECX,[EBP-01cH]; ix
        MOV     EBX,07H
        MOV     EDI,02H
        ALIGN 04H
FELB6:

;***** 16     {
;***** 17     tp=6-tp;
        MOV     EDX,[EBP-010H]; tp
        NEG     EDX
        ADD     EDX,06H
        MOV     [EBP-010H],EDX; tp
;***** 18     if ( ll<=t )
        CMP     [EBP-018H],EBX; ll
        JG      FELB7
;***** 19       {
;***** 20       l++;
        INC     EDI
;***** 21       ll=prime[l]*prime[l];
        MOV     EDX,DWORD PTR [EAX+EDI*04H]
        IMUL    EDX,EDX
        MOV     [EBP-018H],EDX; ll
;***** 22       }
FELB7:

;***** 23     for ( ix=2 ; ix<l ; ix++ )
        MOV     ECX,02H
        CMP     EDI,02H
        JLE     FELB8
        ALIGN 04H
FELB9:
        MOV     [EBP-020H],EDI; @CBE17
        MOV     ESI,EAX
;***** 24       {
;***** 25       q=t/prime[ix];
;***** 26       tt=q*prime[ix];
;***** 27       if ( t==tt ) break;
        MOV     EDI,DWORD PTR [ESI+ECX*04H]
        MOV     EAX,EBX
        CDQ
        IDIV    EDI
        MOV     EDX,EDI
        MOV     EDI,[EBP-020H]; @CBE17
        XCHG    ESI,EAX
        IMUL    EDX,ESI
        CMP     EDX,EBX
        JE      FELB8
```

```
;***** 28        }
        INC      ECX
        CMP      ECX,EDI
        JL       FELB9
FELB8:

;***** 29      if ( l==ix ) prime[npr++]=t;
        CMP      EDI,ECX
        JNE      FELB12
        MOV      EDX,[EBP-014H]; npr
        MOV      DWORD PTR [EAX+EDX*04H],EBX
        INC      EDX
        MOV      [EBP-014H],EDX; npr
FELB12:
        MOV      EDX,EBX

;***** 30      }
        MOV      EBX,[EBP-010H]; tp
        ADD      EBX,EDX
        CMP      EBX,0ffaH
        JL       FELB6

;***** 31    return npr;
        MOV      EAX,[EBP-014H]; npr
        POP      EDI
        POP      ESI
        POP      EBX
        LEAVE
        RET
gen     ENDP

;***** 34 int main(int argc, char *argv[])
        ALIGN 04H

        PUBLIC main
main    PROC
        PUSH     EBX
        PUSH     ESI
        PUSH     EDI
        SUB      ESP,0cH
```

```
;***** 39   rc=DosAllocMem(&mem,16384,PAG_READ+PAG_WRITE+PAG_COMMIT);
        PUSH    013H
        PUSH    04000H
        LEA     ECX,[ESP+010H]; mem
        PUSH    ECX
        MOV     AL,03H
        CALL    DosAllocMem
        ADD     ESP,0cH

;***** 40   if ( rc ) return printf("non-zero return code from DosAllocMem=%d\n",rc);
        OR      EAX,EAX
        JE      FELB18
        PUSH    EAX
        MOV     EAX,OFFSET FLAT: @STAT1
        SUB     ESP,04H
        CALL    _printfieee
        ADD     ESP,014H
        POP     EDI
        POP     ESI
        POP     EBX
        RET
FELB18:

;***** 41   last=gen(p=mem);
        MOV     EAX,[ESP+08H];  mem
        MOV     [ESP+04H],EAX;  p
        CALL    gen
        MOV     ESI,EAX

;***** 42   printf("there are %u primes less than 65536\n",last);
        PUSH    ESI
        MOV     EAX,OFFSET FLAT: @STAT2
        SUB     ESP,04H
        CALL    _printfieee
        MOV     EAX,ESI
        ADD     ESP,08H

;***** 43   for ( ix=0 ; ix<last ; ix++ )
        OR      EAX,EAX
        JLE     FELB20
        MOV     EBX,EAX
        MOV     EDI,[ESP+04H];  p
        XOR     ESI,ESI
        ALIGN 04H
```

```
FELB21:

;***** 44      {
;***** 45      printf("%6u ",p[ix]);
        PUSH    DWORD PTR [EDI+ESI*04H]
        MOV     EAX,OFFSET FLAT: @STAT3
        SUB     ESP,04H
        CALL    _printfieee
        ADD     ESP,08H

;***** 46      if ( 9==(ix%10) ) printf("\n");
        MOV     EAX,ESI
        MOV     ECX,0aH
        CDQ
        IDIV    ECX
        CMP     EDX,09H
        JNE     FELB22
        MOV     EAX,OFFSET FLAT: @STAT4
        CALL    _printfieee
FELB22:

;***** 47      }
        INC     ESI
        CMP     ESI,EBX
        JL      FELB21
FELB20:

;***** 48    return 0;
        XOR     EAX,EAX
        ADD     ESP,0cH
        POP     EDI
        POP     ESI
        POP     EBX
        RET
main    ENDP
CODE32  ENDS
END
```

5.3.6 Questions

Please answer the following questions, using the preceding listings:

1. How large is segment 1 of DEMO.EXE? _____________

2. What is the segment:offset of the 'bufprint' routine? _____________

3. What is the segment:offset of the symbol 'have_freed'? _____________

4. Does DEMO.EXE call DosWrite? ______ How can you tell? _____________

5. Which routine begins at address 0001:12E8? _____________

6. How long is the routine named 'terminate'? _____________

7. Which routine contains address 0001:1888? _____________

8. What is the program′s entry point? _____________

9. What is the name of the routine which has the entry? _____________

10. How far into this routine is the actual entry point? _____________

11. What is the first instruction mnemonic generated by line 28? _____________

12. What variable is in EAX when the return at line 31 executes? _____________

13. Offset 0124 in DEMO.C is in which C function? _____________

14. What variable name is used by the instruction at 0040? _____________

15. Where does the code for line 34 start? _____________

Note: In the .ASM file, the numbers in the assembler instructions are hex. You can tell because they are suffixed with ′H′.

The assembler code generated has the variable name following each line where it is referenced. This makes it easy to locate the variables, because you simply use the address expression in the instruction.

16. Look at line 11. To what are the numbers -18, -14, -10 relative?

    ____

17. Look at the code generated for line 15.

    Where will the variable ′t′ be found? __________________

18. If a failure were to occur in routine ′gen′, what command would you use to display only the variable ′npr′? DD _____________

19. How would you display the variable ′t′? DD _____________

20. Is the variable ′l′ in ′gen′ at the same location as the variable ′ix′ in main? Yes / No Explain. __________________________________

    ____

    ____

5.4 Exercise 5: Unwinding a 16-Bit Stack

Objectives:

1. To learn how to unwind a stack. This is how to find the calling hierarchy which existed at some particular point, such as at the point of failure.
2. To learn how to mine information from the stack frames.

Normally, every routine which has not returned to its caller will have a stack frame. Each stack frame normally contains the parameters passed to a routine, the return address for the routine, and the data which is local for that routine.

Start the dump formatter just as before, on the same dump by typing (X is the CD-ROM drive)
`X:HANDS-ON8162.DFDF_RET X:DUMPS.162DUMP01.DMP`

Questions:

1. The convention states that BP or EBP will point to the current stack frame. SP will point to the lowest address which is in use.

   Therefore, note the initial values for SP _______ and BP ________. Since SS is the selector that defines the stack, note which it is. Some analysts also note the limit of the SS descriptor, because that value bounds the range of both SP and BP.

   SS ________ SSLIM _______

2. Display the current stack frame using `DW SS:BP`. This will show you only part of the frame, but most analysts do this because it makes following the chain so easy.

   The first word is the offset, or near address, of the next frame. The second word is the offset part of the return address. If the call was a far call, the return must also be a far call. If this is the case, the third word is the selector part of the return address.

   next stack frame ________ return offset ________ selector ________

3. Some number of words following the return address are the parameters passed. There is no way to know for certain how many parameters there are, unless you know how both the caller and the routine are written. Analysts typically write down a few words, as convenient.

   parameter word# 1 ________ 2 ________ 3 ________ 4 ________

4. At this point we have gleaned what we can from this frame. Now you need to repeat the process for the rest of the stack frames.

Many analysts will follow the entire chain of stack frames before going to the system or application documentation to find the names of the routines involved, and the line numbers. Others choose to go back and forth, and put in the routine names and line numbers for each frame as they go.

The application documentation will tell you where variables are stored. Remember that each routine uses its own stack frame, so be certain to use the numeric value rather than the register name 'BP' to look at local data for routines other than the failing one.

If you display from SP to BP-2, or ESP to EBP-4, you will see the entire local data for the routine using the current stack frame. This can be quite nice for locating the individual variables.

Find the routine which failed by looking at the .MAP file.

Find the line number that failed by looking next at the .COD file.

The following variables are involved in the failure: 'npr' and 't'. their locations can be found in the .COD file.

Find the location of npr,________ then display its value ________

Find the location of t,________ then display its value ________

You may want to look at the call to the failing routine, before going away to find the programmer.

5.5 Exercise 6: Unwinding a 32-Bit Stack

Objectives:

1. To learn how to unwind a stack. This is how to find the calling hierarchy which existed at the point of failure.
2. To learn how to mine information from the stack frames.

Normally, every routine which has not returned to its caller will have a stack frame. Each stack frame normally contains the parameters passed to a routine, the return address for the routine, and the data which is local for that routine.

Start the dump formatter by typing (X is the CD-ROM drive)
`X:HANDS-ON8162.DFDF_RET X:DUMPS.162DUMP04.DMP`

Questions:

1. The convention states that BP or EBP will point to the current stack frame. ESP will point to the lowest address which is in use.

   Therefore, note the initial values for ESP _______ and EBP ________. Since SS is the selector that defines the stack, note which it is.

   SS ________ SSLIM _______ (not generally useful when SS is 53)

2. Display the current stack frame using DD SS:EBP. This will show you only part of the frame, but most analysts do this because it makes following the chain so easy.

   The first doubleword is the offset, or near address, of the next frame. The second doubleword is the offset part of the return address. If the call was a far call, the return must also be a far call. If this is the case, the third doubleword is the selector part of the return address.
   It is rare for 32-bit programs to use FAR addresses.

   next stack frame ________ return offset ________

3. Some number of doublewords following the return address are the parameters passed. There is no way to know for certain how many parameters there are, unless you know how both the caller and the routine are written. Analysts typically write down a few doublewords, as convenient.

   parameter doubleword# 1 ________ 2 ________ 3 ________ 4 ________

4. At this point we have gleaned what we can from this frame. Now you need to repeat the process for the rest of the stack frames.

```
eax=00080000 ebx=000097eb ecx=0000002d edx=00001000 esi=000000c5 edi=0000002d
eip=0001008e esp=000320c0 ebp=000320cc iopl=2 rf -- -- nv up ei pl zr na pe nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001a7000
005b:0001008e 891c90         mov dword ptr [eax+edx*27],ebx ds:00084000=invalid

DD SS:ESP EBP-4
0053:000320c0  00000000 00000000 00000000

DD SS:EBP L18
0053:000320cc  000320f8 000100f2 00080000 00080000
0053:000320dc  00080000 00000000 00000000 00000000
0053:000320ec  00010f8e 00000001 00070010 00000000
0053:000320fc  1bfbbf68 0000036d 00000000 00040000
0053:0003210c  0004030b 00000000 00000000 00000000
0053:0003211c  00000000 00000000 00000000 00000000

DD 330F8 L 10
0053:000320f8  00000000 1bfbbf68 0000036d 00000000
0053:00032108  00040000 0004030b 00000000 00000000
0053:00032118  00000000 00000000 00000000 00000000
0053:00032128  00000000 00000000 00000000 00000000

The first parameter passed by OS/2 is the load module handle.

.LMO 36D
hmte=036d pmte=%ff652c6c mflags=00903150 c:\pmg\classes\labs\lab4\demo.exe
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 00001a98 00010000 80002025 00000001 00000002 0361 000f r-x shr big
0002 0000006c 00020000 80002003 00000003 00000001 0000 0017 rw- big
0003 00002110 00030000 80002003 00000004 00000001 0000 001f rw- big

Wonder what the 00040000 and 0004030B are? Display them to see!

DB %40000
%00040000 57 50 5f 4f 42 4a 48 41-4e 44 4c 45 3d 31 33 32 WP_OBJHANDLE=132
%00040010 37 33 39 00 41 55 54 4f-53 54 41 52 54 3d 50 52 739.AUTOSTART=PR
%00040020 4f 47 52 41 4d 53 2c 54-41 53 4b 4c 49 53 54 2c OGRAMS,TASKLIST,
%00040030 46 4f 4c 44 45 52 53 2c-4c 41 55 4e 43 48 50 41 FOLDERS,LAUNCHPA
%00040040 44 00 42 4f 4f 4b 53 48-45 4c 46 3d 43 3a 5c 4f D.BOOKSHELF=C:\O
%00040050 53 32 5c 42 4f 4f 4b 3b-00 43 4f 4d 53 50 45 43 S2\BOOK;.COMSPEC
%00040060 3d 43 3a 5c 4f 53 32 5c-43 4d 44 2e 45 58 45 00 =C:\OS2\CMD.EXE.
%00040070 44 50 41 54 48 3d 43 3a-5c 50 4d 47 5c 4f 53 32 DPATH=C:\PMG\OS2

DB %4030B L 20
%0004030b 4c 41 42 34 5c 44 45 4d-4f 00 00 55 f0 8b c7 e8 LAB4\DEMO..Up.Gh
%0004031b d5 1d 01 00 ff 75 e8 e8-19 1e 01 00 83 c4 14 89 U....uhh.....D..
```

```
Frame at    Next Frame at   Return address    parameters:

_________   _________       ______________    _______ _______ _______ _______


Frame at    Next Frame at   Return address    parameters:

_________   _________       ______________    _______ _______ _______ _______


Frame at    Next Frame at   Return address    parameters:

_________   _________       ______________    _______ _______ _______ _______
```

Many analysts will follow the entire chain of stack frames before going to the system or application documentation to find the names of the routines involved, and the line numbers. Others choose to go back and forth, and put in the routine names and line numbers for each frame as they go.

The application documentation will tell you where variables are stored. Remember that each routine uses its own stack frame, so be certain to use the numeric value rather than the register name BP to look at local data for routines other than the failing one.

If you display from ESP to EBP-2, or ESP to EBP-4, you will see the entire local data for the routine using the current stack frame. This can be quite nice for locating the individual variables.

Find the routine which failed by looking at the .MAP file.

Find the line number that failed by looking again at the .MAP file.

The following variables are involved in the failure: ′npr′ and ′t′. Their locations can be found in the .ASM file.

Find the location of npr,________ then display its value ________

Find the location of t,________ then display its value ________ Hint: t has been optimized, and is in a register.

You may want to look at the call to the failing routine, before going away to find the programmer.

5.6 Requesting Kernel Services

If CALL targets a less privileged CS, or RET (RETURN) a more privileged CS, a general protection exception occurs by definition; a trusted program cannot directly invoke a less trusted one.

If CALL targets a more privileged CS, a general protection exception occurs because a less privileged program cannot access a more privileged object (code segment).
It is *impossible* to *directly* call a code segment which is a different privilege level than the caller.
It *is possible* to *indirectly* call a more privileged code segment.

5.6.1 The Task State Segment (TSS)

This hardware control block is used to control hardware multitasking, I/O access, and privilege transitions.

5.6.1.1 How to Find the TSS

There is a selector register named the task register (TR). This register has a GDT selector that chooses a descriptor whose type is TSS. This descriptor contains the base and limit for the TSS.
The fields from offset 4 to 1F are not changed by the hardware.

Table 1. Task State Segment Format

Offset(size)	Content	Offset(size)	Content
00(2)	link - previous tss selector		
04(4)	Ring 0 ESP	08(2)	Ring 0 SS
0C(4)	Ring 1 ESP	10(2)	Ring 1 SS
14(4)	Ring 2 ESP	18(2)	Ring 2 SS
1C(4)	CR3.	20(4)	EIP
24(4)	EFLAGS	28(4)	EAX
2C(4)	ECX	30(4)	EDX
34(4)	EBX	38(4)	ESP
3C(4)	EBP	40(4)	ESI
44(4)	EDI		
48(2)	ES	4C(2)	CS
50(2)	SS	54(2)	DS
58(2)	FS	5C(2)	GS
60(2)	LDT selector		

5.6.2 The Call Gate

This section gives an explanation of what a call gate provides, and how it works.

5.6.2.1 Why Have a Call Gate?

The CALL GATE is the mechanism by which an application requests services from the operating system. Integrity has several requirements which are not immediately obvious to most people.

1. The caller must be forced to use a designed entry point to prevent entry at an arbitrary location; for example, at a point after the parameters have been validated. This might cause the operating system to violate its own integrity or that of another application.

2. The parameters, as well as the rest of the stack, must be protected from the application while in use by the operating system to prevent changes by another thread in that application.

3. The return address must be protected from the application while the operating system is running to prevent other threads of the application from altering it in a way that would cause a return to the application in a privileged mode.

Note: A CALL GATE implements all of the above requirements.

Note: A CALL GATE is a system descriptor which describes an entry point in a more privileged program which is accessible to less privileged programs.

5.6.3 Another View

A Gate is a service window which describes the entry point of the gate and what size package is passed into the more protected ring.

```
┌────────────────────────────────────────────────┐
│               RING 3                           │
│    ┌───────────────────\   ──────────────┐     │
│    │                    \ <───Gate       │     │
│    │          RING 2     \               │     │
│    │    ┌──────\   ───────────────┐      │     │
│    │    │       \  <───Gate       │      │     │
│    │    │        \                │      │     │
│    │    │       RING 0            │      │     │
│    │    │                         │      │     │
│    │    └─────────────────────────┘      │     │
│    │                                     │     │
│    └─────────────────────────────────────┘     │
│                                                │
└────────────────────────────────────────────────┘
```

5.6.4 Call Gate Contents

CALL GATE

PL of Gate	Can I see this gate?
CS of entry	Where is the entry?
EIP of entry	Where is the entry?
Parm Count WC or DWC	What gets passed?

A Descriptor

Note: Observe that the privilege level of the gate controls which privilege level programs can access the gate; the target privilege level is contained in the entry point CS value.

5.6.4.1 Call Gate Overview

When a FAR CALL contains a target code selector (CS) which is a CALL GATE, the processor ignores the offset (IP) contained in the instruction and gets the true target CS and offset (IP) from the CALL GATE. In addition, if the call is to a more privileged program, the processor locates a fresh stack for it to use, stores the current stack selector and stack pointer in the new, more privileged stack, copies the parameters from the old stack to the new one, and finally saves the return information in the new stack. All this happens during the execution of the call instruction.

Briefly, when the return occurs, all this gets undone.

5.6.4.2 Call Gate Detail

From less to more privileged, for example, Ring 3 to Ring 0.

1. Verify new stack will hold linkage data. If not, stack fault, error code 0.
2. New SS, SP from TSS, based on PL of new CS.
3. Old SS:SP copied to new stack.
4. Parameters (up to 15 words or doublewords) copied from old to new stack.
5. Former CS:IP copied to new stack; SP now points here.
6. New CS, IP from Call Gate.

5.6.4.3 A Ring Transition Picture

The following is how the stacks look at entry to the more privileged program:

```
   RING 3 STACK                RING 0 STACK  <-- INITIAL R0 ESP
 ┌────────────┐    high      ┌────────────┐
 │            │  addresses   │     SS     │
 ├────────────┤              ├────────────┤
 │     P3     │              │    ESP     │
 ├────────────┤              ├────────────┤
 │     P2     │              │     P3     │
 ├────────────┤              ├────────────┤
 │     P1     │              │     P2     │
 ├────────────┤              ├────────────┤
 │     P0     │              │     P1     │
 ├────────────┤              ├────────────┤
 │            │              │     P0     │
 ├────────────┤              ├────────────┤
 │            │              │     CS     │
 ├────────────┤              ├────────────┤
 │            │              │    EIP     │
 ├────────────┤              ├────────────┤
 │            │     low      │            │
                addresses
```

Notes

There is no return address on the less privileged stack.

The two items at the top of the more privileged stack are the less privileged SS and ESP.

Subtract 8 from the SP value found in the TSS to find where the less privileged ESP and SS are stored. The values in the TSS are initial values, not the address of the first item pushed.

A trap C in Ring 0 is usually a double fault.

When the processor detects a Stack Exception it needs to push an error code and a return address onto the stack of the exception handler. If this happens in Ring 0, there will be no privilege level transition, which includes switching to a new, protected stack. If the exception is due to stack growth, there is no place to push the error code or return address.

RESULT: TRAP 8

5.6.4.4 Return Detail

From more to less privileged, for example, Ring 0 to Ring 3.

1. Verify that all steps below will work. If not, general protection fault.
2. Pop IP, CS
3. Add immediate value to old SP
4. Pop SP, SS
5. Add immediate value to new SP
6. Zero every selector which has PL more privileged than the new CPL. This is required to maintain integrity because access validation is done only when a selector register is changed - not when it is used.

5.7 Exercise 7: Looking at a Ring Transition

Objectives:

1. Introduction

   To review previous knowledge of analyzing traps

   To begin getting familiar with the debug kernel

   To learn how to identify the API targeted by a call gate

2. Techniques

   To learn about the PATCH program

   To learn about getting control when a module is loaded

3. Finding the TSS and the privileged stacks

   To learn when you may need to find the TSS

   To learn how to find the TSS

   To learn how to find privileged stacks

4. Watching a ring transition

   Look at the ring 0 stack before

   Look at the ring 3 stack before

   Actually execute a far call from ring 3 to ring 0.

   Look at both stacks afterwards.

5.7.1 Part 1: Introduction to the Debug Kernel

Procedures:

Introduction

1. Change to directory `HANDS-ONPROBLEMS.LABLAB09`

2. Execute `OSPREY`, see the failure, and the trap screen.

   At the failure, record CS:EIP from the trap screen.

   CS _________ EIP _____________

   At this point, it is too late to cause a dump. Dismiss the trap screen.

   We will refer to the system on which the problem occurs at the Machine Under Test, or the MUT. The MUT is connected via a null modem cable to an adjacent machine, which we will call the debug terminal. Most of the debugging actions will occur from the debug terminal, on which we will run a public domain terminal emulation program, LOGICOMM. If you like LOGICOMM and intend to use it frequently, you should register it, which will also get you an improved version.

   Let's use the debug kernel for the first time. First, we need to get its attention. The way to do this is to enter Ctrl-C on the debug terminal, after starting LOGICOMM. The debug kernel defaults to settings

   ```
   9600, N , 8, 1
   ```

3. Start `LOGICOMM` on the debug terminal, then type Ctrl-C.

   The default response of the debug kernel is the registers at whatever point OS/2 was interrupted by the Ctrl-C. This is not generally very useful. We need to get control where we want it, not at a random place.

4. Enter the command VSF*

   This tells the debug kernel that you want control on any interrupt which may be Fatal to a thread. The ′F′ is for fatal, the ′*′ is for ′any′.

   Enter the command G(Go), so OS/2 can continue.

5. On the MUT, rerun OSPREY.

   This time, you should get a group of lines on the debug terminal which tell you that a fatal failure has occurred.

   Enter the command DG CS You will find that this is in ring 0.

   Before we look at ring 0, let us find where ring 3 called ring 0, and also identify the API which was called.

   Enter the command .R (the period is very significant!)

   .R shows you the ring 3 registers, whereas R shows you the current ones.

   CS=__________ EIP=_____________ Does this match the trap screen?

   ```
   eax=00000000 ebx=0000405c ecx=00000000 edx=00000001 esi=00000000 edi=000016b0
   eip=00001bc3 esp=000011e4 ebp=0000120e iopl=2 -- -- -- nv up ei pl zr na pe nc
   cs=000f ss=001f ds=001f es=001f fs=150b gs=0000  cr2=00000000  cr3=001a7000
   000f:00001bc3 0bc0           or      ax,ax
   ```

   We already know this instruction did not trap; the trap is in ring 0.

6. If we unassemble prior to 000f:1bc3, we will find a far call.

   ...1BBE call ___________:0000

   The instruction as hex data is _____ _____ _____ _____ _____

7. If you inquire about the descriptor by entering DG and the selector, You should see something similar to this

   ```
   # DG 1xxx
   1xxx  CallG32 Sel:Off=0148:0000550a     DPL=3 P  DWC=7
   ```

   Write down CS ______ EIP__________ DPL________ DWC________

   If you enter the LN command with the values of CS and EIP from the call gate, you will identify the API which is called via this gate.

8. We know how to find parameters on the ring 3 stack, DW SS:SP

   We can also find them on the ring 0 stack, but at this point, the kernel has already manipulated some of the addresses, so there is not an exact match. We need to get control at the point of the call at 1BBE.

9. Enter the command GT which will GoThrough the trap.

5.7.2 Part 2: Some Techniques

Procedures to get control at a point other than a trap:

One approach is to use clever breakpoints within OS/2.

Stopping at the first executable instruction of a program

1. We will make use of a couple of breakpoint commands

   This command tells the debug kernel that we want control on the debug terminal at some specific point. The problem is that the place where we would like to get control is not loaded into memory until we run the program, and it is difficult at best to type Ctrl-C at just the right time.

2. The initial breakpoint uses the fact that almost all programs use the DOSCALL1 DLL, which appears to have instance initialization.

   Enter the command `BP DosLibIDisp,'.p*'`

   The content of the quoted string is the command to execute when we arrive at the breakpoint. This will assure us that we are in the correct context, because the output of '.p' includes the module name.

   Let the MUT run, and execute `OSPREY` once again.

   You will probably get control in the context of OSPREY. If not, issue `G` again a time or two until you are.

3. At this point, OSPREY has been loaded, so we can set a breakpoint.

   If you simply try the command `BP 0F:1BBE` you will discover that the page is not yet loaded.

   There are two ways around this problem.

   a. Use a register breakpoint, `BR E,0F:1BBE`

   b. Cause OS/2 to bring the page in with `.I 0F:1BBE`

      Then reenter the `BP` command from above.

4. However, this is 'cheating' because we already knew where to stop.

   To find the address of the first instruction at this point, enter the command `.M 0F:0` Find the MTE handle, hmte.

   Issue the `.LMO` command with the HMTE as the parameter.

   Alternatively, try `.LMO 'OSPREY'` which works sometimes.

   The output of the .LMO command includes the linear address of the MTE.

   Display the MTE as doublewords, and get the address of the SMTE from the output; it is in the second doubleword.

   Display the SMTE as doublewords, and you can find the entry point in the second and third words displayed.

   Now you can set a breakpoint at the entry to any module.

   The PATCH program

1. On the MUT, execute the `EXEHDR` utility against OSPREY.EXE.

   EXEHDR is distributed with the developers' toolkit.

   The output will provide you information you need to patch a program successfully. The last part of the output should look like

```
Module:                  OSPREY
Description:             OSPREY.EXE
Data:                    NONSHARED
Initial CS:IP:           seg   1 offset 0088
Initial SS:SP:           seg   3 offset 0000
Extra stack allocation:  0a00 bytes
DGROUP:                  seg   3

no. type address  file  mem   flags
  1 CODE 00000200 0247d 0247e
  2 DATA 00000000 00000 00200
  3 DATA 00002800 007cb 00960
```

There are two things we will need in this listing.

2. The entry point, or initial CS:IP is _________:________
3. The location in the file where that segment begins _________

   The columns labelled 'file' and 'mem' are the sizes of the segment in the file, and in memory. The difference is due to un-initialized data, which is not stored, saving space and reducing program load time.

   To find the location of an instruction in the file, add the offset to the file address.

4. To get control, we will replace a byte with the hex value 'CC', which is a special one-byte instruction, Int 3, or BreakPoint.
5. We will patch the call instruction at 1BBE.

   Add the offset, 1BBE to the file address 0200 _________

   If you cannot add hex, get the debug kernel's attention, and then type in `? 1BBE+0200` ? is a general purpose evaluation command.

6. We now know where we want to patch the program. Let's do it.

   On the MUT, enter the command `PATCH OSPREY.EXE`

   The patch address was calculated above; enter it.

   The byte you are about to replace is hex ______

   Type `CC` then press enter, and complete the confirmations.

   We have now patched the program.

7. Execute the program on the MUT; you get control at the INT 3.

   We need to put back the hex data which was originally there, so as not to introduce another problem. We will use the enter command.

   Type the command `E CS:IP`

   You will see the 'CC', type the original data value and press enter.

   Type the command `.R` and you should see the original far call.

8. This is one way to get control.

   It has problems if the MUT is not where you can touch it.

   Type the commands `G` then `GT` to let OSPREY finish.

9. Patch OSPREY back to its original content if you wish.

5.7.3 Part 3: Finding the TSS

It is relatively simple to find and display this critical control block which is used by the hardware for ring transitions.

1. Get the debug kernel's attention, so you can display data.
2. The TSS is located via the Task Register (TR), which is a selector.

   You can find the value in TR by entering `? TR`

   Entering `RT` toggles register terse mode. Try `R` before and after entering `RT`. You can look for TR in the output.

   You really do not want TR, but the TSS, which is at TR:0.
3. Enter

   `DD TR:0` to display the TSS as doublewords

   `DT TR:0` to format the TSS.
4. The first doubleword is the link field.

   It indicates which TSS called this one through a task gate.

   The next two doublewords are the ESP and SS for entry to ring 0.

   The next pair of doublewords are unused by OS/2; they would have the ESP and SS for entry into ring 1.

   The next pair of doublewords are the ESP and SS for entry to ring 2.
5. To display the stack used at entry to ring 0,

   Use the `DD` command with the SS and ESP values from the TSS; BUT Stacks grow downward, so put -80 after the ESP value. 80 is the number of bytes displayed by default; this will show you the top of the stack for ring 0, with the saved SS value as the last item shown.

5.7.4 Part 4: Watching a Ring Transition

We will watch a ring transition by stopping on an instruction which we know causes a ring transition, display both stacks, then single step the instruction, and look at both stacks again.

Get control in OSPREY so that the next instruction is at 0F:1BBE.

1. Display the call gate by using `DG` and the selector from the call.

   Write down the target CS______ EIP____________ DWC_____ PL____
2. Display the ring 3 stack as WORDS

   so you can see as many DOUBLEWORDS as are passed through the gate.

   Display the ring 0 stack as words, too. It is technically incorrect to do this, but for the purposes of this exercise, it makes things easy.
3. Use the command `T` to execute the call instruction.

   Now, again display the ring 0 stack as words again.
4. Compare the ring 0 content now with the content of the ring 3 stack.

   Do not overlook the ring 3 SS and ESP at the top of the ring 0 stack.

   Do not overlook the return address in the ring 0 stack, following the parameters which were copied by the hardware as it executed the call.

5. TIMESAVER:

   If you know what API will be called, you can simply set the breakpoint at the API, by using its name. A side effect is that every thread which calls the API will stop, so you may want to use something like ′.p*′ as the command to execute at the breakpoint, which makes it easy to see when the thread of interest is there.

This lab is now complete. However, if you let it run to the failing instruction, you will find an additional detail about this API, namely that because only 13 words were pushed, and 7 doublewords are needed to get them all copied into the ring 0 stack, there is one more detail we can see, namely how the difference (two bytes) is handled.

If you display the ring 0 stack once again, it has been changed!

The return will need to add enough to the ring 0 stack pointer that it can find the ring 3 stack successfully; this is also what is added to the ring 3 stack pointer, because both stacks must be cleaned up. In order that this not be destructive of what is already on the ring 3 stack, the ring 0 entry code has adjusted the saved ring 3 ESP downward by 2 before the trap occurs. This is an example of some of the work that has been done within the ring 0 stack by the privileged code.

5.8 Exercise 8: Identifying the Owner of Storage

Objectives:

1. To learn how to find out where a part of storage originated
2. To learn how to find out what module contains it, if not dynamically acquired

Every piece of storage has an owner. Storage owned by OS/2 may not have all the storage accounting information which is kept for storage used by applications. The most common clue that this situation has occurred is the presence of the UVirt flag (bit 52) in the descriptor. The next most common clue is that the procedure below may fail if complete storage accounting has not been done.

Within OS/2, handles are used extensively. Generally, a handle is nothing more than an index into some table or other. For diagnostic purposes, one can treat it as a magic number that can be used as an operand on certain commands.

The initial objective is to find the module table entry which the loader built when the module was loaded. This will relate storage to the far addresses in the link map.

The procedure is slightly different for private and shared storage.

With practice, one learns quickly what selectors are likely to be private, and which are likely to be shared. Refer to the address space picture which appears earlier, to refresh your memory about private and shared storage.

One way to tell is to display the entire LDT (using `DL`), and to look for the gap between low numbered and high numbered selectors.

> If the address is private, there will likely be many processes that define the address, and the data is likely different for each. You will need to find which process is the one you want.
>
> The dump formatter command `.I` will show you not only the handle of the module table entry for the executable which caused this process to exist, but also will show you the handle of the ′PTDA′, which is the key control block for a process.
>
> The command usually used to identify storage is the `.M` command.
>
> If issued with a shared address, the output has the handle of the module table entry. If issued for a private address, you get a set of output lines for every process which contains the address. In this case, you will need to use the hPTDA, or PTDA handle from the `.I` command to determine which set of output lines to use.
>
> Once you have the handle of the module table entry, issue the command `.LMO <handle>`
>
> The command will not only give you the full path name of the module, but will also format a table which has a column (toward the right) titled ′sel′. This is the selector assigned. The first line of output is for the first segment in the link map, the second line is for the second segment, and so on. Thus, you can convert the selector:offset in the dump to a segment:offset in the correct link map.

```
.I
PROCESS slot:23 Pid:0008 Ord:0001
PTDA    handle=032e address=%ad6d97f0
MTE     handle=0363 address=%ff666d4c (DEMO)
SMTE    address=%fe14abe8
LDT     handle=035c address=%ac6d7000
CODE:   user (cs:eip)#000f:000000be cbargs=
STACKS: user (ss:esp)#001f:000014be(active)
        ring2(ss:esp)#0036:00001000(bottom)
        ring0 tcbframe=%fe023f58 bottom=%fe023f9c

.M CS:IP
*har     par      cpg        va     flg next prev link hash hob   hal
 01f5 %ff821b18 00000010 %00010000 1c9 01f6 01f3 00fa 0000 0131 0000 hptda=0240
 00fa %ff820586 00000010 %00010000 1d9 0102 00f9 0000 0000 0131 0000 hptda=0117
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0131  01f5 0000  0838 0132 0132  0000 00  00 00 00 shared    c:pmshell.exe

*har     par      cpg        va     flg next prev link hash hob   hal
 0177 %ff821044 00000010 %00010000 179 0178 0175 0000 0000 01be 0000 hptda=01b9
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 01be  0177 0000  002c 01b9 01bf  0000 00  00 00 00 UNKNOWN

*har     par      cpg        va     flg next prev link hash hob   hal
 01f5 %ff821b18 00000010 %00010000 1c9 01f6 01f3 00fa 0000 0131 0000 hptda=0240
 00fa %ff820586 00000010 %00010000 1d9 0102 00f9 0000 0000 0131 0000 hptda=0117
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0131  01f5 0000  0838 0132 0132  0000 00  00 00 00 shared    c:pmshell.exe

*har     par      cpg        va     flg next prev link hash hob   hal
 02b5 %ff822b98 00000010 %00010000 1d9 02b6 02b2 0000 0000 0322 0000 hptda=031c
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0322  02b5 0000  0838 0327 0327  0000 00  00 00 00 shared    c:cmd.exe

*har     par      cpg        va     flg next prev link hash hob   hal
 02e6 %ff822fce 00000010 %00010000 1d9 02e7 02e5 0000 0000 0362 0000 hptda=032e
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0362  02e6 0000  0838 0363 0363  0000 00  00 00 00 shared    c:demo.exe

.LMO 363
hmte=0363 pmte=%ff666d4c mflags=00803142 c:\pmg\pete\demo16\demo.exe
seg  sect psiz vsiz hob  sel  flags
0001 0001 2e78 2e78 0362 000f 2d20 code shr rel
0002 0000 0000 2910 0000 0017 0c01 data
0003 0019 0937 1560 0000 001f 0d01 data rel
```

Start the dump formatter by typing (X is the CD-ROM drive):
`X:HANDS-ON8162.DFDF_RET X:&bslHANDS-ON\DUMPS.162\DUMP01.DMP`

Procedure:

1. Enter the command `.I`

   The PDTA handle is __________, the module table entry handle is _________

2. Enter the command `.M CS:IP` to identify Memory at CS:IP.

   Which 'har' line is for our process? har=_________

   What is the hmte value from this set of lines? hmte=______________

Note: This is exactly what the .I command showed you, because this is what the .I command does internally.

3. Enter the command `.LMO` followed by the hmte value.

   What is the full path name of the module that contains CS:IP?

   ____

4. What is the segment number which has been assigned selector 000F? ______

5. What address would you look for in the link map to find CS:IP?

   ____

6. Now, repeat the same steps using the data in the next few displays. The address of interest is DFDF:9324

7. What is the privilege level of this segment? ________

8. What is its size? ___________

9. What is the command to identify memory at this address?

   ____

   Issue it. The lines which start hco= are context records, which indicate all of the contexts (processes) that can reference this address. It is extremely likely to be a shared address.

10. Issue the `.LMO` command for the module table entry handle.

11. What module is this? Full path name is ______________________

12. Which segment in the module contains this address? _________

13. Therefore, in the .MAP file, the address will be ______:_______

```
DG DFDF
LDT
dfdf  Code    Bas=1bfb0000 Lim=0000d4ef DPL=2 P  RE C  A

.M DFDF:9324
*har     par      cpg        va    flg next prev link hash hob   hal
 00b7 %ff81ffc4 00000010 %1bfb0000 3d9 0075 00b8 0000 00b4 00c5 0000  hco=0026c
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00c5  00b7 0000  0838 00bf 00bf  0000 00  00 00 00 shared    c:doscall1.dll
 hco=026c pco=ffe62c37  hconext=00184 hptda=032e f=1c  pid=
 hco=0184 pco=ffe627af  hconext=00014 hptda=031c f=1c  pid=
 hco=0014 pco=ffe6207f  hconext=00089 hptda=0240 f=1c  pid=
 hco=0089 pco=ffe622c8  hconext=00021 hptda=01b9 f=1c  pid=
 hco=0021 pco=ffe620c0  hconext=00000 hptda=0117 f=1c  pid=

.LMO BF
hmte=00bf pmte=%ff66ef3c mflags=0498b594 c:\os2\dll\doscall1.dll
obj   vsize    vbase    flags    ipagemap cpagemap hob  sel
0001 00000360 1bf80000 80009025 00000001 00000001 00c8 dfc6 r-x shr alias iopl
0002 0000aa30 1bf90000 80002025 00000002 0000000b 00c7 dfcf r-x shr big
0003 0000d519 1bfa0000 8000d025 0000000d 0000000e 00c6 dfd6 r-x shr alias conf iopl
0004 0000d4f0 1bfb0000 8000d025 0000001b 0000000e 00c5 dfde r-x shr alias conf iopl
0005 00001140 13f90000 80001023 00000029 00000002 00c4 9fcf rw- shr alias
0006 00001af0 13fa0000 80001003 0000002b 00000002 0000 9fd7 rw- alias
0007 00000e44 13fb0000 80001023 0000002d 00000001 00c2 9fdf rw- shr alias
0008 00000550 13fc0000 80001003 0000002e 00000001 0000 9fe7 rw- alias
0009 00001000 13fd0000 80001023 0000002f 00000000 00c0 9fef rw- shr alias
000a 00001000 13fe0000 80001023 0000002f 00000000 00be 9ff7 rw- shr alias
```

Chapter 6. Steps to Diagnose a Trap

The intent of the following material is to illustrate a proven method for finding the cause of a trap in an application program. By first learning how to solve the simplest problems, one will have a much better basis for approaching more difficult problems. Historically, problem solving skills have been largely self-taught. Much can be learned by observing others solve problems. Many problems can be solved quickly by using significant short-cuts and assumptions and then verifying them. When a novice observes an experienced diagnostician, the activities are difficult to understand, and may lead to the opinion that each problem has its own special method for solution, which in turn leads to questions about when to use which method.

The following process will lead you to the cause of a trap.

Remember to take notes as you proceed. This will help if you are interrupted, and want to continue later, or if you need to explain to someone else what you found, and what facts led you to a particular analysis of the situation. You can obviously do this manually, but you can use a log file more easily. Just type ? followed by whatever you wish to log. The tools will evaluate the string, supplying the trailing quote, and show you the string, thus adding your thoughts to the log.

1. Locate the failing instruction.

   If you cannot do this, you have no place to start. Most operating systems will provide at least an excellent clue to the location of the failing instruction, if not its exact address.

2. Determine why the failing instruction will not execute.

   A knowledge of hardware operation, or a reference manual kept handy, is essential for this step. At the very worst, each of the possible exceptions described in the manual can be eliminated one by one until the cause is found.

   Until you know why the instruction will not execute, you do not know what went wrong at the machine level. Conversely, as soon as you do know, you are prepared to begin the diagnosis of how things got into such a state. Observe that this does not require knowledge of C, FORTRAN, COBOL, SMALLTALK, etc. It requires only hardware knowledge.

3. Analyze how the conditions for failure occurred. It may be that an address calculation was done incorrectly, or that the failure was due to an invalid parameter. If the former, you now need only to discover what program has done this, and where in that program the error occurred. Skip the next two steps.

4. If an invalid parameter has been received, you must now update your notion of the cause of the problem. You need to consider the call as the location of the failure, and the specific parameter value as the reason why the called routine did not execute.

5. You must now analyze how the parameter was created, and where it came from. Unwind one stack frame, and return to step 3.

6. You now know what caused the problem, and now it is time to identify the failing program, locate the failing line, find the value of the program's variables, and, in general, collect all the data the programmer would have

had if the failure had occurred at his desk. This step is usually a mechanical one.

Once this is done, go find the programmer, and turn over all you know about the problem. Be prepared to continue helping, or to show the programmer how to get additional information.

Chapter 7. The OS/2 System Trace

The System Trace facility is used to record a sequence of system events, function calls, or data in a fixed-size circular buffer as requested by calls to its API′s. The buffer must be allocated during the processing of CONFIG.SYS.

If you have a TRACEBUF statement in CONFIG.SYS, a trace buffer is allocated, allowing you to use the TRACE command successfully later.

If any valid TRACE statements are in CONFIG.SYS (including TRACE=OFF), a trace buffer will be allocated for the default size of 4K, if TRACEBUF is not specified. This means that the statement TRACE=OFF will actually enable system tracing, which seems counter-intuitive to many people.

If you do not specify TRACE or TRACEBUF statements in CONFIG.SYS, OS/2 does not allocate a trace buffer and system tracing is disabled.

After the trace data is recorded, the trace formatter is used to retrieve the data from the system trace buffer and present it on your display, and optionally, to print it, or save it in a file.

7.1 TRACEBUF and TRACEFMT

TRACEBUF=x

TRACEBUF sets the size of the trace buffer in the CONFIG.SYS file.

The parameter x specifies a circular trace buffer size of up to 63K. If you have a TRACEBUF statement without a TRACE statement in CONFIG.SYS, the trace buffer size requested is specified and tracing is turned off (the same as if you specify TRACE=OFF).

If you need to use the System Trace facility, use the largest size, if possible. TRACEBUF=63

TRACEFMT displays formatted trace records in reverse time-stamp order. It is intended to be used to format the trace data so that you can analyse the content of the trace buffer. The most recent entry is displayed first. TRACEFMT numbers each event as it is formatted. The event numbers are unrelated to the trace data, and are useful when discussing a trace with someone else, for easy reference to events.

TRACEFMT works without a filename only if you have a trace buffer defined in the running system.

TRACEFMT works with a filename only if the file is a hex image of a trace buffer from a system for which you have Trace Formatting Files. If the .TFF file is not correct, the entries which are different will be formatted incorrectly with little or no indication of an error.

The file is typically created by the dump formatter by using the command `.TS filename` but TRACEFMT will also save the trace buffer in unformatted form. This is much smaller than the formatted form.

7.2 TRACE and TRACE Processing

The TRACE command is used to control the system trace.

```
     Command line:                       CONFIG.SYS:

     TRACE ON                            TRACE=ON
     TRACE OFF                           TRACE=OFF
(you can specify only static TRACE in the CONFIG.SYS file).
```

The above is optionally followed by one or more major code specifications, or one or more trace definition file specifications, or keywords. Next, you may optionally specify one or more process identifiers. Finally, you may specify that the trace buffer be cleared, and that trace activity be suspended, or resumed.

OS/2 processes trace statements in the order in which they appear from any source. TRACE commands in CONFIG.SYS are processed in the order they appear. The effects of the statements are cumulative for the duration of OS/2's execution. If any part of a statement is incorrect, OS/2 ignores the statement.

Process Id is specified by /P:nn,nn,nn (where nn is in hex)

Clearing the trace buffer is specified by using /C.

Resuming trace activity is specified by using /R.

Suspending trace activity is specified by using /S.

Major and minor event codes are associated with all trace events. Some of the major codes are the following:

Machine Exceptions Major Code: 3

Hardware Interrupts Major Code: 4

Device Helper Routines Major Code: 6

Disk Device Driver Major Code: 7

Major codes may be specified by listing them separated by commas, or as a range, for example, 2-7 specifies codes 2,3,4,5,6,7 Both methods may be combined, as in 5,7,12-18,2,27-32,9

If you do not specify TRACE in CONFIG.SYS, event tracing is not started by CONFIG.SYS processing, but may be started later if TRACEBUF has been specified.

Records in the buffer are identified by major and minor codes. Some of the data that may be recorded in the circular buffer includes system events such as interrupts, exceptions, and thread switches.

OS/2 contains a mixture of static tracepoints and dynamic tracepoints.

Static tracepoints are implemented as trace function calls within individual software modules. The TRACE command can be used to turn on and off static tracepoints by specifying them by major code and, optionally, by minor code.

Dynamic tracepoints are implemented by implanting an INT 3 instruction at the specified location, and gathering data when the interrupt occurs.

7.3 TRACEFMT Processing

When the trace is complete, you can use the trace formatter (TRACEFMT) to format the data into a report. This helps you to isolate causes of problems in the OS/2 system by making the data in the trace buffer available for analysis.

7.4 Static and Dynamic Trace, and Files Used

Trace format files (.TFF) and trace definition files (.TDF)

Static tracing occurs when a program developer has coded an API call to the system trace interface, which means you cannot specify at what point in the program flow tracing occurs, nor can you control what data is collected.

Trace format files are used by TRACEFMT. They specify how the trace data should be formatted. The filename implies which major code is described, and TRACEFMT generates the filename for the .TFF file from the major code of the event about to be formatted. If no description is found, or if the description does not describe all of the trace entry, TRACEFMT defaults to hexadecimal bytes for a default formatting. This will be covered in detail by hands-on exercises.

Trace definition files are used for dynamic tracing, and specifying one of them requires you to name the .DLL involved, or KERNEL. You may optionally a type or list of types and a group or list of groups. The .TDF file is used by TRACE to define where to collect data, and what data to collect.

Dynamic tracing occurs when trace definition files (.TDF) are used by the TRACE command. The implementation is that OS/2 inserts actual breakpoint instructions at the specified locations, and collects the data specified when the breakpoint is executed. There is no overhead for dynamic tracing when it is not in use, and a technician can be very creative when defining where to collect trace data, and what data to collect. We will create custom dynamic trace entries during hands-on exercises.

The OS/2 static tracepoints do not have associated TDF files, but may have associated TFF files that are used by the TRACEFMT.

7.5 Dynamic Trace Processing

Dynamic tracepoints are implemented as trace definition file (TDF) entries. The TRACE command can be used to insert (and turn on) a dynamic tracepoint by patching it into its corresponding software module. Dynamic tracepoints are specified by the dynamic link library (DLL) filename and minor code.

Individual dynamic tracepoints can be qualified by separate type and group qualifiers. These qualifiers exist so that you can more easily turn on and off sets of related dynamic tracepoints. For example, all the dynamic tracepoints that are associated with pre-invocation events might have a type of PRE. Similarly, all the dynamic tracepoints that are involved in semaphore processing might have a group of SEM. In the TRACE command syntax, group is considered to have a stronger binding than type. This means that you can ask to turn on all events that are of a specified group that are also of one or more specified types. You do not need to use these qualifiers; they are there simply to make it easier to control related sets of dynamic tracepoints.

TDF files are typically found in the \OS2\SYSTEM\TRACE directory. They are identified by .TDF file name extensions. There are also trace formatting files (TFF) found within that directory. These files are used by the OS/2 Trace Formatter (TRACEFMT) utility to format the entries that are logged within the system trace buffer.

Commonly Used Abbreviations for OS/2 Groups and Types:

```
Groups                        Types
FS-  file system              API-  application programming interface
KBD- keyboard I/O             INT-  internal
LDR- resource loader          PRE - pre-processing invocation
LNK- environment management   POST- post-processing invocation
MOU- mouse I/O
MSG- message management
MSP- virtual memory management
NLS- national language support
PIP- pipe support
SEL- selector-related
SEM- semaphore support
SIG- signal handling
TIM- timer support
TK - task management
TSK- monitor support
VIO- video I/O
VM - virtual memory management
```

7.5.1 OS/2 Predefined Dynamic Trace Events

The file SYSTEM.TDF file supports dynamic tracing for the following:

```
TRACE ON KERNEL              Major Code: 5 (decimal) 5 (hex)
      Groups: FS, LDR, NLS, PIP, SEL, SEM, SIG, TIM, TK, VM
      Types: PRE, POST, API, INT
      Purpose: Tracepoint definitions for APIs in the OS/2 kernel

 TRACE ON DOSCALL1           Major Code: 16 (decimal) 10 (hex)
      Groups: FS, LDR, LNK, MSG, MSP, NLS, SEM, TSK
      Types: PRE, POST, API
      Purpose: Tracepoint definitions for APIs in DOSCALL1.DLL

 TRACE ON QUECALLS           Major Code: 22 (decimal) 16 (hex)
      Groups: None
      Types: API, PRE, POST, INT
      Purpose: Tracepoint definitions for APIs in QUECALLS.DLL

 TRACE ON SESMGR             Major Code: 23 (decimal) 17 (hex)
      Groups: None
      Types: API, PRE, POST
      Purpose: Tracepoint definitions for APIs in SESMGR.DLL

 TRACE ON OS2CHAR            Major Code: 24 (decimal) 18 (hex)
      Groups: KBD, MOU, VIO
      Types: API, PRE, POST
      Purpose: Tracepoint definitions for APIs in OS2CHAR.DLL

 TRACE ON PMSHAPI            Major Code: 192 (decimal) C0 (hex)
      Groups: None
      Types: None
      Purpose: Tracepoint definitions for APIs in PMSHAPI.DLL

 TRACE ON PMWIN              Major Code: 194 (decimal) C2 (hex)
      Groups: None
      Types: None
      Purpose: Tracepoint definitions for APIs in PMWIN.DLL

 TRACE ON PMGRE              Major Code: 195 (decimal) C3 (hex)
      Groups: None
      Types: None
      Purpose: Tracepoint definitions for APIs in PMGRE.DLL

 TRACE ON PMGPI              Major Code: 197 (decimal) C5 (hex)
      Groups: None
      Types: None
      Purpose: Tracepoint definitions for APIs in PMGPI.DLL
```

The file SYSTEM.TFF file provides formatting information for OS/2 events.

Chapter 8. TRCUST, the Dynamic Trace Customizer

OS/2 provides a mechanism by which developers may dynamically apply tracepoints in their module at run time. This method eliminates all overhead of tracing when tracing is disabled. It also allows the developer to add tracepoints without modifying source code. This reduces the possibility that adding a tracepoint will induce errors into one′s code. OS/2 needs a binary file, for each module being dynamically traced, which defines the tracepoints for the module.

TRCUST converts tracepoint definitions from a trace source file (TSF) into dynamic tracepoints for the trace definition file (TDF), and into formatting rules in the trace format file (TFF).

```
                   ---> .TDF ---> trace ( set tracepoints )
    .TSF ---> trcust              tracepoint data collected in buffer
                   ---> .TFF ---> tracefmt formats data from the buffer
                                  produces formatted output
```

```
.TSF An ASCII file created by a developer which defines all dynamic
     tracepoints for a given module. TRCUST currently allows at most
     only one major code per TSF.

.TDF A binary file, created by TRCUST, using the .TSF file as input.
     This file defines all tracepoints in the module in a manner
     acceptable to OS/2. This is used by the Trace Utility, TRACE.

.TFF A binary file also created by TRCUST using the .TSF file.
     This file defines how all tracepoints will be formatted.  This
     is used by the Trace Formatter, TRACEFMT.
```

To trace a module do the following:

1. Define the tracepoints and data to be traced in the TSF.
2. Invoke the Trace Customizer using the TSF as input. This produces two files, a TDF and a TFF.
3. Put the TDF file in the current directory, a directory in the DPATH, or \OS2\SYSTEM\TRACE.
4. Invoke the OS/2 TRACE command using the name of the TDF instead of the major code value. This activates the tracepoints, causing the trace data to be saved in the system trace buffer.
5. The OS/2 TRACE command can be used to turn tracing off at any time.
6. Put the TFF file in the directory OS/2 uses, \OS2\SYSTEM\TRACE.

   Many versions of TRACEFMT appear to search directories named in DPATH.
7. To display the contents of the trace buffer, invoke TRACEFMT. TRACEFMT uses the major code to determine the TFF file and uses the formatting string corresponding to the minor code value to format the data in the RAS trace buffer and output it to the screen, file or printer.

8.1 File Naming Convention

The TDF file name is the same as the module to be traced, but has a file extension of TDF. The TFF has a name of the form TRC00xx.TFF where xx is the major code, for example, a module with major code 0xC2 will generate a TFF with the name TRC00C2.TFF. This naming convention is used to allow TRACEFMT to dynamically generate the TFF name given only the major code.

TRCUST can be invoked to process a TSF or to combine several TFF files into a single TFF. For processing a TSF, TRCUST is given the name of a TSF, and optionally the desired name of the resulting TDF, the MAP file name, and the error message level.

For combining TFF files that use the same major code, TRCUST is given the name of the file containing the TFF filenames to combine and the name of the file to contain the combined trace format statements.

Combine TFF files when several modules that use dynamic tracing use the same major code. The Trace formatter can only use one TFF file per major code to get formatting information. After the TSF file for each module is run through TRCUST to produce TDF and TFF files, TRCUST can be invoked again, this time using the combine TFF files option and a file that only contains the paths to all the TFF files using the same major code. TRCUST will read all the TFF files. TRCUST will read each trace format record from the TFF files and write them in ascending order according to minor code to the destination TFF file given.

TRCUST will store the TSF tracepoint formatting specifications in the TFF file and if the tracepoint specified was for a dynamic tracepoint, the TSF tracepoint definition will be converted into the format required by TRACE and stored in the TDF file. On errors, TRCUST will display appropriate messages, skip any tracepoint with errors in its definition, and continue processing the next tracepoint definition.

TRCUST will issue an error message and abort processing under the following conditions:

- The TSF cannot be opened.
- When combining TFF files, if any TFF input files cannot be read, or if all TFF input files do not use the same major code.
- When defining dynamic tracepoints, if the executable module to contain the tracepoints cannot be read.
- The TDF, or TFF files cannot be written to.
- An error in the header definition in the TSF.
- A missing ending quote in the TSF.

TRCUST always returns 0 so that, when invoking it from a makefile, processing of the rest of the makefile can continue if TRCUST aborts.

8.2 The Syntax for Processing a TSF File

The syntax for processing is as follows:

```
[d:][path]TRCUST [d:][path]tsf [[d:][path]tdf] [/M=mapfile] [/Wn]
```

where:

TRCUST
: is the name of the Trace Customizer program. A drive and path may optionally be specified to explicitly define the location of the Trace Customizer program, otherwise the program is searched for in the current directory, followed by looking along the path defined by the PATH environment variable.

[d:][path].tsf
: is the name of the trace source file. If no file extension is provided then an extension of TSF is assumed. If no path is provided the trace source file is searched for in the current directory, followed by using the current value of DPATH.

[d:][path].tdf (optional)
: is the name of the trace definition file to store the dynamic tracepoint definitions. If not specified, the TSF filename is used with an extension of TDF. If no file extension is provided then an extension of TDF is assumed.

/M=mapfile (optional)
: defines mapfile as the MAP file for this module. The name may be qualified by a drive/directory, otherwise it will be searched for in the current directory followed by the path specified by the DPATH environment variable. If specified as an option, the MAP file must exist and the filename extension must be MAP or TRCUST will abort processing. The mapfile will only be used if a symbol is not found in the symbolic debug information stored in the executable module.

/Wn (optional)
: is the level of error messages to be displayed. n is 0, 1, or 2. A message level of 2 is the default. The possible message levels are shown below along with the messages that each displays:

  0 fatal and severe messages
  1 fatal, severe, and error messages
  2 all (fatal, severe, error, and warning) messages

8.3 The Syntax for Combining .TFF Files

The syntax for combining is as follows:

```
[d:][path]TRCUST [d:][path]tffsource /C=[d:][path]tffdest [/Wn]
```

where:

TRCUST
: is the name of the Trace Customizer program. A drive and path may optionally be specified to explicitly define the location of the Trace Customizer program, otherwise the program is searched for

in the current directory, followed by looking along the path defined by the PATH environment variable.

[d:][path]tffsource
is the name of a file containing fully qualified pathnames of TFF files including extensions to combine. Each TFF file must use the same major code and each filename in the tffsource file is separated by white space.
This will combine all TFF files for the same major code into a single TFF file. If duplicate minor code format definitions are found, the first format definition for the minor code remains valid, the duplicates are discarded and a warning message is issued. If no path is provided the tffsource file is searched for in the current directory, followed by using the current DPATH.

[d:][path]tffdest
is the name of the trace format destination file to store the combined trace format definitions.

/Wn (optional)
is the level of error messages to be displayed. n is 0, 1, or 2. A message level of 2 is the default. The possible message levels are shown below along with the messages that each displays:

0 fatal and severe messages
1 fatal, severe, and error messages
2 all (fatal, severe, error, and warning) messages

Chapter 9. The Layout of a Trace Source File

The layout of a trace source file is:

```
                  Header

            Type List Definition
            Group List Definition
               (both optional)



           Tracepoint Definitions
```

This section details the statements that can appear within a trace source file.

```
[...]              denotes optional items.
[... | ... | ...]  denotes a list of optional items, zero or more of
                   which may be chosen.
{... | ... | ...}  denotes a list of items of which ONE must be chosen.
item...            denotes that item is repeated zero or more times.
statement,.....    denotes this example is incomplete.
nnn                is a number in the range 0-255 inclusive.
nnnnn              is a number in the range 0-65535 inclusive.
All numbers and values can be entered in decimal form or
in C hexadecimal form (0x....).
```

Comments may be freely inserted anywhere in the trace source file. A comment is identified by a ′;′ or by using C syntax comments anywhere in the file.
A C comment has start and end delimiters, namely /* and */.
C type comments may span lines, and may be nested.

9.1 The Trace Source File Header

Examples of TRACE statements in the header are given as follows:

The TSF header defines common information for the module to be traced.
The format is:

```
MODNAME = [d:][path]Name
MAJOR   = nnn

[MAXDATALENGTH = nnnn]
```

where:

d: is the drive containing the module. If not specified the current drive is used.

path is the path to the module. If not specified the current path is used.

`Name` is the name of the executable module to be traced. If an extension is not specified and the Name is not OS2KRNL, an extension of DLL is appended to Name.

`MAJOR=nnn` defines the major trace ID allocated to this module. It may be in the range 1 to 255 decimal or specified 0x1 to 0xFF hex. The default value is 1. The major trace ID is part of the data placed in the trace buffer when a tracepoint is executed. Only one major code is permitted per module.

Note: OS/2 only supports major codes 0x1 - 0x00FF.

`MAXDATALENGTH=nnnn` (optional) defines the maximum amount of data that a single tracepoint call will insert into the trace buffer. The length may be in the range 20 to 512 decimal or specified 0x14 to 0x200 hex. The default value is 512. This limit on the amount of data to trace is to avoid yielding the processor when doing dynamic tracing.

9.2 TYPELIST and GROUPLIST Statements

Both the TYPELIST and the GROUPLIST statements are optional.

If you want to be able to control sets of tracepoints by type, or by group, you must list the names you plan to use in a TYPELIST or GROUPLIST statement. Failure to have the name in the appropriate LIST will result in an error when processing the tracepoint, and that tracepoint not be defined.

```
TYPELIST
  NAME=Typename
    defines a 1-8 byte character string used to reference the TypeValue
    in the tracepoint definitions.  All TypeNames and GroupNames within
    a TSF must be unique.

  ID=TypeValue
    defines a bit value of the form 2**y where y in range 0 to 15,
    permitting a maximum of 16 types to be defined in a single TSF.
    This can be decimal or specified  0xnnnn for hex.

An example TYPELIST definition follows:

            TYPELIST NAME=PRE,ID=1,
               [NAME=SYS,ID=0x40,]
               [NAME=API,ID=128,]
               [NAME=POST,ID=0x8000,... ]


GROUPLIST

NAME=GroupName
    defines a 1-8 byte character string used to reference the
    GroupValue in the tracepoint definitions.  There are a maximum of
    48 GroupNames allowed in a TSF file.  All TypeNames and GroupNames
    within a TSF must be unique.

ID=GroupValue
    defines a word value in the range 1 to 65535 or 0x1 to 0xFFFF hex.

An example GROUPLIST definition follows:

           GROUPLIST NAME=MEM,ID=2,
               [NAME=FS,ID=0x5,]
               [NAME=MOU,ID=13,... ]
```

9.3 The Tracepoint Definition

The tracepoint address and the data to be traced are specified by the TRACE statement.

There are at most 65535 tracepoints permitted in a trace source file. Minor code zero is not allowed.

The format of the TRACE statement is:

```
TRACE   [MINOR=minorcode,]
        TP={@STATIC,|@filename,linenum,|.name[{+|-}offs][,RETEP]},
        [OPCODE=0xnn,]
        [TYPE=(typename[,typename...]),]
        [GROUP=groupnam,]
         DESC="Tracepoint description",
        [FMT="Formatting string",]...
        [LEN=(length_spec,flag),]
        [DATA_STMT,]...
```

The TRACE keyword delimits a tracepoint definition statement. The definition is considered complete when the next TRACE keyword is encountered or the end of file is reached. There is one TRACE statement for each tracepoint.

MINOR parameter is an optional keyword parameter. If it is specified in the first tracepoint definition, it must be specified in every tracepoint definition. If it is not specified in the first tracepoint definition, it cannot be specified in any of the subsequent tracepoint definitions. It should be coded as:

MINOR=nnnnn,

where:

nnnnn is a decimal number from 1 to 65535 or a hex number from 0x1 to 0xFFFF. This represents the minor code for the tracepoint, which must be unique for the major code specified for this module. When tracepoints with duplicate minor codes are encountered, the first is saved and the rest are discarded, and an error message is issued.

If minor codes are not specified in the TSF, TRCUST provides them sequentially, starting with 1, for each tracepoint definition processed.

9.3.1 TRCUST and Debugging Options

If the module has been compiled and linked with the debug options, then the trace customizer can look into the module to extract symbolic information. In this case symbolic addresses may be used.

Note: Symbolic names are case sensitive. Leading underscore characters must be used, if part of the symbol.

1. /Ti on the C/Set2 language compiler for C source files
2. /CO on the link command

Note: Not all source files must be C language, although only public labels from assembler routines will be found in the symbolic information. You may specify filename and line number, a local variable name or a global variable name when using C routines.

The trace customizer can also use the symbolic information in the MAP file produced by the linker. All public symbols will be listed with their offsets in the module being traced. This is not as complete a support as offered by the debug compile option for C language source files, but it does allow entry points, public labels and global data to be referenced symbolically within the TSF. Note that the use of a MAP file is NOT language dependent.

To trace only public procedures, you only need the MAP file that was generated by linking the module.

It is quite simple to counterfeit a MAP file which has an entry which describes an invented symbol at the address of interest, for example, a file which looks like this

```
0002:0008   RIGHT_HERE
```

will allow you to use symbol RIGHT_HERE to define a tracepoint.

To trace local variables in C language routines, compile the C programs with the debug option and assemble the ASM routines with public symbols. Link all the OBJs together with the debug option(/CO) and run TRCUST on the executable module. You can now strip the debug information from the executable file by either relinking the OBJs without the debug option or by using a tool to delete the debug information from the executable module file.

9.3.2 Specifying Where to Cause the Trace Event

TP is a required keyword that defines where a tracepoint is located. There are three ways to specify the tracepoint, as follows:

1. Static tracepoint

   TP=@STATIC,

2. Dynamic, using debugging information

   TP=@filename,linenum,

3. Dynamic, using names from a map file

   TP=.name[{+|-}offs][,RETEP],

9.3.3 The TP Parameter - Define Where a Tracepoint Occurs

The TP parameter is a required keyword parameter. If TP is specified more than once for a single tracepoint definition, the tracepoint is discarded.

Note: In no case can two tracepoints be applied at the same address.

TP=@STATIC, ...

The @STATIC defines this tracepoint entry to be used only for creating a trace format statement for the TFF file. No tracepoint definition is created for the TDF, and the only other TRACE parameters that will be used are DESC, MINOR and FMT. This is used to create trace formatting information for static tracepoints. If the TSF contains only @STATIC directives, no TDF files are created.

An example of defining a static tracepoint follows:

```
  TRACE   MINOR=0x70C2,TP=@STATIC, ...
```

TP=@filename,linenum,

The filename is an ASCII string specifying the name (including extension) of a source filename used in creating the module. The source filename is stored in the debug information contained in the executable module, so debug information must exist to use this parameter. The filename is not case sensitive.

Linenum is a decimal number specifying the line number in the given source file name to place the tracepoint.

Note: Debug information must exist to use this option. The statement at the given source linenumber may have been rearranged during compiler optimization, so the developer must use this with caution. If the lin enumber is not found in the debug information, the tracepoint is applied at the next line number defined in the debug information and a warning message is issued to the user.

The module must include information supplied by the debug compile option (see Symbolic Debug Support), meaning that the source language must have been C, otherwise an error message will be generated and this tracepoint discarded.

When the RETEP is used, the name must be a valid entry point to a procedure.

Note: For ASM functions to accomplish tracing, a label must be made public to have a tracepoint applied. Therefore, to accomplish POST tracing, a label must be made public at the return statement.

An example to apply a tracepoint to line 35 of file STUBFILE.C is:

```
TRACE MINOR=0X70C3,TP=@stubfile.c,35, ...
```

TP=.name[{+|-}offs][,RETEP],

Where name is a public label or an entry point name of a procedure to be traced. The ″.″ preceding name is required. Name must be found in the debug information in the module or name must be a public symbol as found in the MAP file. If debug information is used, the address of this tracepoint will be immediately following the prologue of the procedure. If MAP information is used, this address points to the opcode at the given label. If the procedure was compiled with debug support, Name is case sensitive. If not, C language functions will be case sensitive and begin with an underscore "_" character unless the function is declared with the Pascal calling convention, in which case the underscore is omitted and the name is capitalized.

Offs (optional) is a decimal (specified as nnnnnnnn) or hex (specified as 0xnnnnnnnn) offset from the entry point address.

RETEP (optional) specifies that the tracepoint will be inserted at the return address corresponding to this entry point. This is just before the procedure epilogue is executed and at this point the procedure's automatic data is still addressable from register (E)BP and the return code (if any) is set up in (E)AX.

An example to apply a tracepoint at map symbol MYENTRY is:

TRACE MINOR=0x70FF,TP=.MYENTRY,...

9.3.4 OPCODE, TYPE and GROUP Statements

These statements are all optional.

An example of the optional keyword parameter OPCODE follows:

OPCODE=0x9A, where:

9A is the expected one byte hex opcode to be found at the tracepoint address and TRCUST verifies the value with that in the module. The opcode of the instruction being traced must be the same as this value or an error message is issued and the tracepoint is rejected. This is used to verify the opcode expected at the address specified by the TP parameter. This is useful when using TP = @filename,linenum to ensure the requested instruction is traced.

Note: It is not possible to set dynamic tracepoints on the following machine instructions:

```
0xCC  INT 3      0xCD  INT n      0xCE  INTO
0x62  BOUND      0x9C  PUSHF
0x69  IMUL       0x6B  IMUL
0xF6  DIV, IDIV, MUL, IMUL, NEG, NOT, TEST (immediate)
0xF7  DIV, IDIV, MUL, IMUL, NEG, NOT, TEST (immediate)
```

TRCUST gives an error for these opcodes and the tracepoint is rejected.

9.3.5 TYPE and GROUP Statements

The TYPE parameter is an optional keyword parameter that defines the event types of this tracepoint. For more description and examples of event types see the online help for the trace command.

TYPE=(typename[,typename...]),

Where typename is an ASCII string specifying the type of this tracepoint. The typename symbol must have been previously defined by the TYPELIST statement. If an invalid typename is given, the tracepoint will be discarded and a message issued.

The final type value is obtained by logically combining each type name value using the OR operator. If TYPE is omitted, the trace statement will have a typevalue of 0.

The GROUP parameter is an optional keyword parameter that defines the group this tracepoint belongs to. For more description and examples of groups see the online help for the trace command.

```
GROUP=Groupnam,
```

Where Groupnam is an ASCII string specifying which group this tracepoint belongs. The groupname symbol must have been previously defined by the GROUPLIST statement. If an invalid groupname is given, the tracepoint will be discarded and a message issued.

If GROUP is omitted, the trace statement will have a groupvalue of 0.

9.3.6 The Description of the Tracepoint

The DESC parameter is used to produce a description for the tracepoint that is output as the first line of formatted data. It should include the entry point name of the procedure being traced and whether this is an entry or return point. The descriptive string is enclosed in double quotes as for a C language string. The DESC parameter is required if any FMT specifications are present, but may be a null string.

The recommended formats for such strings are as follows:

```
DESC="name Pre-Invocation",
DESC="name Post-Invocation",
```

Where name is the system component (in parentheses) followed by the entry point name of the procedure.

Pre-Invocation identifies this tracepoint as an entry point, that is, before the function has been executed.

Post-Invocation identifies this tracepoint as a return point from the function. The words Pre-Invocation and Post-Invocation are not mandatory, merely recommendations to be compatible with the base OS/2 tracepoints, when formatted. If a tracepoint is inserted in the middle of a procedure it will be appropriate to use different wording. The trace customizer does not check the wording.

An example of pre-invocation and post-invocation tracepoints follow:

```
TRACE  MINOR=0x0001,
       TP=.DosOpen,TYPE=(PRE),
       DESC="(OS) DosOpen    Pre-Invocation",.....

TRACE  MINOR=0x8001,
       TP=.DosOpen,RETEP,TYPE=(POST),
       DESC="(OS) DosOpen    Post-Invocation",.....
```

9.4 Using FORMAT Strings to Format the Trace Data

The optional FMT parameter is used to produce the formatting string for the trace data. The developer should use these to control formatting the output produced by the Trace Formatter. Each FMT keyword causes CR/LF to be appended to the format string. The formatting string is similar to a C library printf string. It consists of ASCII characters and formatting controls enclosed in double quotes as for a C language string. Each formatting primitive describes the format of the data in the trace buffer at the formatting position and must match the data stored in the trace buffer by the data statements described later. See Formatting Trace Data for a description of how the data is stored in the trace buffer and subsequently formatted.

The formatting controls are as follows:

%Innn Ignore nnn number of bytes in the trace buffer.

This tells the trace formatter to skip over the next nnn bytes in the current trace record. This could be used for example to skip over unimportant data, traced as a block, and only output the data of interest. When using this control, nnn represents an ASCII decimal number and must be followed by a space.

```
statement: FMT = ″ignore ten bytes %I10 here″,
           FMT = ″       and two more %I2 here″,

generates: ignore ten bytes here
                  and two more here
```

%P Process the data prefix bytes associated with the trace data.

This tells the trace formatter that the next bytes in the trace record are the prefix or header bytes for data logged by the dynamic tracing mechanism. This is required to precede any format control describing data logged from memory. Do not use this before data that was logged from a register and never use it with static tracepoints.

```
statement: FMT=″memory byte = %P%B″,
generates:  memory byte = C2
```

%R Repeat the following format control for the rest of the memory that was logged. This is used for formatting variable length records. Use this in place of the prefix parameter %P to log the rest of the record in the format specified following the repeat code.

```
statement: FMT = ″log a variable number of words from memory = %R%W″
generates: log a variable number of words from memory = 0001 0004
```

%B Output a byte of data.

```
statement: FMT = ″memory byte = %P%B″
generates: memory byte = 01
```

%W Output a word of data.

```
statement: FMT = ″register word = %W″
generates: register word = 0001

statement: FMT = ″memory word = %P%W″
generates: memory word = 0001
```

%D Output a double word of data.

```
statement: FMT = ″double word EAX = %D″
generates: double word EAX = 0000 4B2C

statement: FMT = ″double memory word = %P%D″
generates: double memory word = 0000 4B2C
```

%Q Output a quad word of data.

```
statement: FMT = ″quad word from regs EAX and EBX = %Q″
generates: quad word from regs EAX and EBX = 00004B2C 00000001
```

%F Output a Flat (0:32 bit) address.

```
statement: FMT = ″flat address EAX = %F″
generates: flat address EAX = 00004B2C
```

%A Output a segmented(16:16 bit) address.

```
statement: FMT = ″segmented address in SS:SP = %A″
generates: segmented address in SS:SP = 00B7:0001

statement: FMT = ″segmented address in memory = %P%A″
generates: segmented address in memory = 00B7:0001
```

%S Output an ASCIIZ string.

```
The prefix formatting control should always precede this for
dynamic tracepoints because the data was logged from memory.
Note:  If the tracepoint is static, then %P should not be used
because the string is terminated with a null byte.

        statement: FMT = ″string = %P%S″
        generates: string = c:\os2\os2.ini
```

%X Output the major event code.

```
statement: FMT = ″major code = %X″
generates: major code = 00C2
```

%Y Output the minor event code.

```
statement: FMT = ″minor code = %Y″
generates: minor code = 0081
```

%U Format the remainder of the trace record as a sequence of bytes.

```
This will output the remaining of the traced data, including any
prefix bytes.

        statement: FMT = ″garbage = %U″
        generates: garbage = 00 00 00 03 c2 c1 c4 ff 04 00 09 c0 18
```

Note: To avoid conflicts with source file control information, all formatting specifications can be in upper or lower case. Also, prefix format specifications may be combined with data format specifications. For example, the following three statements create the same format controls in the TFF:

```
FMT = ″%P%W here″
FMT = ″%p%w here″
FMT = ″ %P %W here″
```

9.4.1 Specifying the Data to Trace

There are three types of data that may be traced as part of the optional DATA_STMT section of the TRACE statement, Registers, Memory, and ASCIIZ strings. The keywords for tracing the three types of data are REGS, (MEM32 and MEM),and (ASCIIZ32 and ASCIIZ).

9.4.1.1 Registers

The REGS keyword identifies which registers are to be recorded in the trace buffer.

```
REGS=(register[,register]...),
```

Where register is one of the following or the symbolic name of a C language variable declared with the register storage-class specifier as, .symbolic_name

```
SS,CS,DS,ES,FS,GS

 AX, BX, CX, DX, SI, DI, FLAGS, IP, SP, BP

EAX,EBX,ECX,EDX,ESI,EDI,EFLAGS,EIP,ESP,EBP
```

The same register may appear multiple times in the register list. It will be traced as many times as it appears. Extended registers (E) are 32 bits and logged as two words. All other registers are 16 bits and logged as one word.

```
Example of the REGS statement follows:
     /* Given the following declaration in a C language source file: */
     register int ret_code;
     /* To log registers AX, CX and the register variable ret_code: */
     TRACE MINOR=.....
           REGS=(AX,CX,.ret_code),
           FMT="AX=%W  CX=%W  ret_code=%W"
```

Note: When formatting the data logged from a register variable, remember that there are no memory prefix bytes put into the log buffer.

9.4.1.2 Memory

The MEM32 keyword is used to record sections of memory in the trace buffer by specifying 32-bit flat addresses. The MEM keyword is also used to record sections of memory in the trace buffer, but by specifying a segment:offset address form.

The ASCIIZ32 keyword is used to record an ASCIIZ string in the trace buffer. This is a special form of the MEM32 keyword. The ASCIIZ keyword is also used to record an ASCIIZ string in the trace buffer. This is a special form of the MEM keyword.

More than one keyword is permitted in a tracepoint definition. The order of the statements defines the order in which the data is inserted into the trace buffer. The combined amount of data to be traced for a single tracepoint cannot exceed MAXDATALENGTH. If TRCUST determines that the maximum data size might be exceeded, a warning message is issued but the tracepoint definition will remain valid.

9.4.2 Gathering Data from Memory: Address Specifications

All statements that specify memory data must have an address specification one of the following forms:

1. Symbolic name form can be used for MEM32, MEM, ASCIIZ32, and ASCIIZ.

   The symbolic name form is coded as follows:

   .name[{+|-}nnnnnnnn]...[{+|-}(iiiiiiii)],

   Where name is a symbolic name of a memory location. The "." is required before the name. The debug information in the module is checked for the name and if not found and a MAP was given, the MAP is checked. An error message is output by the Trace Customizer if the symbol is not found and the trace definition is ignored.

nnnnnnnn is an optional displacement from the symbolic address. If hex the syntax is 0xnnnnnnnn.

iiiiiiii is an optional displacement from the indirect address. If hex the syntax is 0xiiiiiiii. This specifies a displacement from the final address when using INDIRECT, IF(Indirect Flat) or IS(Indirect Segmented) addressing. This form of address is calculated at run time.

2. Flat register form can be used only for MEM32 and ASCIIZ32.

   Flat register form coded as:

   Fbreg[{+|-}ireg]...[{+|-}nnnnnnnn]...[{+|-}(iiiiiiii)],

   Wwhere breg is a flat model(0:32 bit) base register and is one of:

   EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI

   ireg (optional) is an extended data, base or index register. More than one ireg may be used to define a displacement from the flat register value to the memory location.

   It may be one of: EAX,EBX,ECX,EDX,EBP,ESI,EDI

   nnnnnnnn and iiiiiiii are the same as above.

3. Segment register form can be used only for MEM and ASCIIZ .

   The segment register form of an address is coded as:

   Rsreg[{+|-}dreg]...[{+|-}nnnnn]...[{+|-}(iiiii)],

   Where sreg is a segment register and is one of:

   CS,DS,SS,ES,FS,GS

   dreg is an optional data, base or index register. More than one dreg may be used to define a displacement from the segment register value to the memory location.

   It is one of: BP,SP,SI,DI,AX,BX,CX,DX

   nnnnn and iiii are as above, but limited to 16 bits (0 to 65535).

9.4.3 Gathering Data from Memory: Length Specifications

Each memory specfication must have a length specification. In the cases of MEM and MEM32, it specifies the length. In the cases of ASCIIZ and ASCIIZ32, it specifies the maximum length.

The length may be specified as a number, for fixed-length data, or by use of the LEN parameter for variable-length memory data. In the case of variable-length data, the length must be specified by a LEN statement which immediately precedes the data statement itself. The LEN statement gives the location of a one word field containing the number of bytes to log in the following data statement.

Length is the number of bytes at the memory location to be saved in the trace buffer. If length is too big, a warning message will be given, and length will be set to MAXDATALENGTH. If length is 0 an error message will be given, and this tracepoint will be ignored.

LEN=(length_spec,flag),

Where length_spec is an address specification that points to the one word length field of the next memory specification. This format can be symbolic_name+nnnnnnnn where symbolic_name is a symbolic memory location and nnnnnnnn is the offset from that symbolic address. The length_spec can also be Flat Register Form or Segment Register Form.

Flag is a mandatory parameter that identifies the level of indirection to be used on the length_spec.

It is either D[IRECT]

DIRECT implies that the length_spec specifies a memory location that contains the length of the variable length record.

Or I[NDIRECT][*[{+|-}iiiiiiii]]...

INDIRECT means that the length_spec contains an address and is dereferenced to obtain the memory location. The optional asterisks denote the level of indirection, one for each level. The indirect offsets iiiiiii are added to or subtracted from the value found at the given level of indirection.

```
TRACE ...,
/* Symbol vrecord is a record whose first field is a one    */
/* WORD value that is the total length of the entire record */
      LEN=(vrecord,DIRECT),
      MEM=(.vrecord,DIRECT,LEN),

/* Symbol vrec_ptr is a pointer to a variable length record.*/
/* The second field (2 bytes from beginning of record) is   */
/* total length of the variable length record.              */
      LEN=(vrec_ptr,INDIRECT*+2),
      MEM=(.vrec_ptr,INDIRECT,LEN),

/* Registers DS:DI are a pointer to a structure.  The */
/* third field in the structure (6 bytes from beginning) is */
/* a pointer to a variable record.  The fourth field in the */
/* variable length record(8 bytes from beginning) is the    */
/* total length of this variable length record.             */
      LEN=(RDS+DI,INDIRECT*+6*+8),
      MEM=(RDS+DI,INDIRECT*+6*,LEN)
```

9.4.4 Specifying Data from Memory

The MEM32 keyword is used to log memory using 32-bit flat addressing and is coded as follows:

MEM32=(address_spec,flag,{length|LEN}),

Where address_spec is a flat memory address as described above

flag is a mandatory parameter that identifies the level of indirection to be used on the address. It is the same as for LEN, described above, or

```
  IS(Indirect Segmented) means that the address contains a segmented
address that is dereferenced to obtain the memory location.
```

Only far pointers may be dereferenced when using segmented addressing. Either length or LEN must be specified, but not both. 'LEN' specifies that this is a variable length record to log and the length was specified by the preceding LEN statement. If there was no preceding LEN statement, this tracepoint is rejected.

The following are example LEN statements followed by the memory statement whose length they describe.

The MEM keyword is used to log memory in a function compiled using 16-bit segment:offset addressing and is coded as follows:

MEM=(address_spec,flag,{length|LEN}),

where: address_spec is a segmented memory address specification as described above.

flag is a mandatory parameter that identifies the level of indirection to be used on the address. It is the same as LEN, above, or IF (Indirect Flat) means that the address contains a flat address that is dereferenced to obtain the memory location.

LEN is described above.

The ASCIIZ32 keyword is used to log a string using 32-bit flat addressing and is coded as follows:

ASCIIZ32=(address_spec,flag,maxlength),

Where address_spec, flag and maxlength are as described above, under MEM32.

The ASCIIZ keyword is used to log a string in a function compiled using 16-bit segment:offset addressing and is coded as follows:

ASCIIZ=(address_spec,flag,maxlength),

Where address_spec, flag and maxlength are as described above, under MEM.

When using dynamic tracing, the OS/2 kernel does not place the terminating null byte into the trace buffer; therefore the prefix byte must be used by the Trace Formatter to obtain the length of the string.

9.5 Examples

This section gives a brief description of the formatting process as an aid to generating correct formatting strings.

Each trace record stored in the RAS buffer consists of a header followed by a number of variable length trace data records. The header identifies the major and minor code, time stamp, process ID, etc., and the total length of the trace data for that trace record.

Each MEM32, MEM, ASCIIZ32, or ASCIIZ data statement, coded in the trace source file for a tracepoint, produces an associated data record to be stored in the trace buffer. The data records consist of a 3-byte prefix followed by the trace data. This prefix consists of a status byte followed by the length of the data for that statement. The status byte indicates whether valid data has been traced.

Dynamic trace can only trace data that is resident in memory at the time that the tracepoint is executed. Data may not be able to be traced for two reasons: it resides in a page that is currently paged out or the address specified is invalid. This latter case usually occurs due to tracing indirectly via invalid pointer variables. In either of these two cases dynamic trace sets the status byte accordingly and stores the pointer in the place of the wanted data. No more data is attempted to be traced for this invocation of the tracepoint, but tracing will resume the next time this tracepoint is encountered.

Since the position of these prefix bytes, within a trace record, is dependent on the data being traced and the number of MEM32, MEM, ASCIIZ32, or ASCIIZ statements, the trace formatter must be told when to expect the prefix in the trace record. This is the purpose of the %P formatting control. It must be coded in the formatting string at every place a data record is expected. Note: With ASCIIZ and ASCIIZ32 commands, the prefix must be used to obtain the length of the string since the string is not null terminated.

This section gives sample TSF files. The first is for a module written in a mix of C and MASM and compiled with 16:16 segmented addressing. The second was compiled with 0:32 flat addressing. The third module consists of routines, some which were compiled using 16-bit segmented addressing and some that were compiled using 32-bit flat addressing. The fourth is for monitoring function references in a module.

9.5.1 Example 1

A 16-bit example.

```
;  Trace source file for the xxx DLL.  16-bit compilation.
;  We will want to trace up to 200 bytes in any one trace call.
 MODNAME=\c\src\xxx.dll, MAJOR=0xC5, MAXDATALEN=200,
 TYPELIST NAME=API,ID=08, NAME=SYS,ID=04, NAME=PRE,ID=02, NAME=POST,ID=64
 GROUPLIST NAME=MEM,ID=1, NAME=FS,ID=3

/* The following tracepoint does not need debug info, only a MAP file is
necessary with label xxalloc public in it.  The program must be compiled
in 16-bit mode because segmented addressing is used (ASCIIZ instead of
ASCIIZ32).  This logs the word registers AX and BX and the string pointed
at by DS:DI for a max of 20 bytes. */
      TRACE    MINOR=25, TP=.xxalloc,
               OPCODE=0x8B,    /* the opcode is optional */
               TYPE=(API,PRE), GROUP=MEM,
               DESC="(OS) xxalloc Pre-Invocation",
               FMT ="                                  AX = %W ",
               FMT ="                                  upper BX = %B",
               FMT ="                                  lower BX = %B",
               FMT ="                                  param = %P%S",
               REGS=(AX,BX),
               ASCIIZ=(RDS+DI,DIRECT,20)

/* This defines a tracepoint at Foo label.  The ten words to log are
found indirectly through SS:SP.  Note that each word needs a
format control but since only one memory access was done, one prefix
control is needed. */
       TRACE   MINOR=0xB0, TP=.Foo, TYPE=(SYS), GROUP=FS,
```

```
            DESC="(OS) Foo Pre-Invocation",
            FMT="                    First Five words = %P%W%W%W%W%W",
            FMT="                    Three words ignored %I6",
            FMT="                    Last Two Words = %W%W",
            MEM=(RSS+SP,INDIRECT,20)

/* This defines a tracepoint at Goo label.  DS:DI points to a structure
whose second field is a pointer to an ASCIIZ string. The offset from the
first field in the structure is 4 bytes. Max string size is 40 bytes. */
      TRACE  MINOR=0xB1, TP=.Goo, TYPE=(SYS), GROUP=FS,
            DESC="(OS) Goo Pre-Invocation",
            FMT="          Second field in struct points to %P%S",
            ASCIIZ=(RDS+DI+4,INDIRECT,40)

/* This defines a tracepoint at Hoo label.  DS:DI points to memory that
contains a pointer to a structure. We want to log the third field in the
structure (offset 6 from begin of structure). */
      TRACE  MINOR=0xB2, TP=.Hoo, TYPE=(SYS), GROUP=FS,
            DESC="(OS) Hoo Pre-Invocation",
            FMT="          Third field in struct is doubleword = %P%D",
            MEM=(RDS+DI,INDIRECT*+6,4)
/* This defines a tracepoint at Zoo label.  DS:DI points to memory
that contains a pointer to end of a structure. We want to log the last
field in the structure (offset -2 from end of structure). */
      TRACE  MINOR=0xB3, TP=.Zoo, TYPE=(SYS), GROUP=FS,
            DESC="(OS) Zoo Pre-Invocation",
            FMT="          Last field in struct is word = %P%W",
            MEM=(RDS+DI,INDIRECT*-2,2)

/* This defines a tracepoint at procedure CheckIt. This is a C routine
compiled with debug information.  The data to log is an ASCIIZ string
called NameIt.  */
      TRACE  MINOR=0xB3, TP=.CheckIt, TYPE=(PRE), GROUP=FS,
            DESC="(OS) CheckIt Pre-Invocation",
            FMT="              NameIt = %P%S",
            ASCIIZ=(.NameIt,DIRECT,64)

/* This defines a tracepoint at the return point of the procedure CheckIt,
a C routine compiled with debug. Status_Rec is a record variable. We want
to log the age field(four bytes from the begin of Status_Rec), the name
(six bytes from Status_Rec that points to an ASCIIZ string), the age of
the next Status_Rec (a pointer to the next Status_Rec is ten bytes from
the begin of Status_Rec, the age is four bytes from the begin of the
next Status_Rec). */
      TRACE  MINOR=0x80B3, TP=.CheckIt,RETEP, TYPE=(POST), GROUP=FS,
            DESC="(OS) CheckIt Post-Invocation",
            FMT="              Status_Rec.age = %P%W",
            FMT="              Status_Rec.name = %P%S",
            FMT="              Status_Rec.next->age = %P%W",
            MEM=(.Status_Rec+4,DIRECT,2),
            ASCIIZ=(.Status_Rec+6,INDIRECT,64),
            MEM=(.Status_Rec+10,INDIRECT*+4,2)

/* This defines a tracepoint at line 58 in the source file check.c
Debug info is needed to use this type of tracepoint. v_ptr is a pointer to
a variable sized record.  The length is 4 bytes past the beginning of the
record.  Log that record. */
      TRACE  MINOR=0x71B4, TP=@check.c,58, TYPE=(SYS), GROUP=FS,
            DESC="(OS) CheckIt   before allocation",
```

```
              FMT="             Variant Record = %P%W%D%U",
              LEN=(v_ptr,INDIRECT*+4),
              MEM=(.v_ptr,INDIRECT,LEN)

/* This does not define a tracepoint, it only defines a trace
formatting string for minor code 181(B5 hex). */
       TRACE   MINOR=0xB5, TP=@STATIC,
               DESC="(OS) StaticProcedure Pre-Invocation",
               FMT="                DI = %W FLAGS = %W"

/* This defines a tracepoint at routine LookUp, but no data is to be
logged, only the DESC will show up in the Trace log when the tracepoint
is formatted. */
       TRACE   MINOR=0xB6, TP=.LookUp, TYPE=(SYS), GROUP=FS,
               DESC="(APP) LookUp  Pre-Invocation",
```

9.5.2 Example 2

A 32-bit example

```
;  Trace source file for the NEW DLL. 32-bit compilation.
;  We will want to trace up to 200 bytes in any one trace call.
 MODNAME=NEWCALLS.DLL MAJOR=241 MAXDATALEN=200
 TYPELIST NAME=API,ID=08, NAME=SYS,ID=04, NAME=PRE,ID=02, NAME=POST,ID=64
 GROUPLIST NAME=MEM,ID=1, NAME=FS,ID=3

/* The following tracepoint does not need debug info, only a MAP file
is necessary with label NewAllocSeg public in it.  The program must be
compiled in 32-bit mode because flat addressing is used (ASCIIZ32
instead of ASCIIZ). This logs lower word of EAX, the double word of
EBX and the string at the address specified by ESP with offset ESI. */
      TRACE    MINOR=45, TP=.NewAllocSeg, TYPE=(API,PRE), GROUP=MEM,
               DESC="(NEW) NewAllocSeg Pre-Invocation",
               FMT ="                          AX = %W ",
               FMT ="                          EBX = %F",
               FMT ="                          param = %P%S",
               REGS=(AX,EBX),
               ASCIIZ32=(FESP+ESI,DIRECT,20)

/* This defines a tracepoint at Foo label.  The ten words to log are found
indirectly by using EBP with offset EDI. Note that each value logged needs
a format control. */
    TRACE   MINOR=0xD0, TP=.Foo, TYPE=(SYS), GROUP=FS,
            DESC="(NEW) Foo Pre-Invocation",
            FMT="                    First Five words = %P%W%W%W%W%W",
            FMT="                    Three words ignored %I6",
            FMT="                    Last Two Words = %W%W",
            MEM32=(FEBP+EDI,INDIRECT,20)

/* This defines a tracepoint at Goo label. EAX + EDI points to a structure
whose second field is a pointer to an ASCIIZ string.  The offset from the
first field in the structure is 4 bytes. Max string size is 40 bytes. */
       TRACE   MINOR=0xD1, TP=.Goo,
               TYPE=(SYS),
               GROUP=FS,
               DESC="(NEW) Goo Pre-Invocation",
               FMT="          Second field in struct points to %P%S",
               ASCIIZ32=(FEAX+EDI+4,INDIRECT,40)
```

```
/* This defines a tracepoint at Hoo label.  EBP + EDI points to memory
that contains a pointer to a structure. We want to log the third field
in the structure(offset 6 from begin of structure). */
    TRACE   MINOR=0xD2, TP=.Hoo, TYPE=(SYS), GROUP=FS,
            DESC="(NEW) Hoo Pre-Invocation",
            FMT="          Third field in struct is doubleword = %P%D",
            MEM32=(FEBP+EDI,INDIRECT*+6,4)

/* This defines a tracepoint at Zoo label.  EAX + EDI points to memory that
contains a pointer to end of a structure. We want to log the last field in
the structure(offset -2 from end of structure). */
       TRACE   MINOR=0xD3, TP=.Zoo, TYPE=(SYS), GROUP=FS,
               DESC="(OS) Zoo Pre-Invocation",
               FMT="          Last field in struct is word = %P%W",
               MEM=(FEAX+EDI,INDIRECT*-2,2)

/* This defines a tracepoint at procedure CheckIT.  This is a C routine
compiled with debug data. The data is an ASCIIZ string called NameIt. */
       TRACE   MINOR=0xD3, TP=.CheckIt, TYPE=(PRE), GROUP=FS,
               DESC="(NEW) CheckIt Pre-Invocation",
               FMT="              NameIt = %P%S",
               ASCIIZ32=(.NameIt,DIRECT,64)

/* This defines a tracepoint at the return point of the procedure CheckIt,
a C routine compiled with debug. Status_Rec is a record variable. We want
to log the age field(four bytes from the begin of Status_Rec) the name
(six bytes from Status_Rec that points to an ASCIIZ string) and the age
of the next Status_Rec (a pointer to the next Status_Rec is ten bytes
from the begin of Status_Rec, the age is four bytes from the begin of
the next Status_Rec). */
       TRACE   MINOR=0x80D3, TP=.CheckIt,RETEP, TYPE=(POST), GROUP=FS,
               DESC="(NEW) CheckIt Post-Invocation",
               FMT="              Status_Rec.age = %P%W",
               FMT="              Status_Rec.name = %P%S",
               FMT="              Status_Rec.next->age = %P%W",
               MEM32=(.Status_Rec+4,DIRECT,2),
               ASCIIZ32=(.Status_Rec+6,INDIRECT,64),
               MEM32=(.Status_Rec+10,INDIRECT*+4,2)

/* This does not define a tracepoint, it only defines a
trace formatting string for minor code 223(DF hex). */
       TRACE   MINOR=0xDF, TP=@STATIC,
               DESC="(NEW) StaticProcedure Pre-Invocation",
               FMT="                 DI = %W FLAGS = %W"

 /* This defines a tracepoint at routine LookUp, but no data is to be
 logged, only the DESC will show up in the Trace log when the tracepoint
 is formatted. LookUp is a C language routine not compiled with debug
 and not declared with Pascal calling conventions; the underscore is
 needed for this label. */
       TRACE   MINOR=0xE0, TP=._LookUp, TYPE=(SYS), GROUP=FS,
               DESC="(NEW) LookUp  Pre-Invocation"
```

9.5.3 Example 3

```
; Trace source file for the MIXED DLL. Parts were compiled with 16-bit
; compiler, some with 32-bit compiler.  The developer must know how the
; parameters are to be addressed, whether segmented or flat addresses.
;  We will want to trace up to 200 bytes in any one trace call.
 MODNAME=MIXCALLS.DLL MAJOR=250 MAXDATALEN=200
 TYPELIST NAME=API,ID=08, NAME=SYS,ID=04, NAME=PRE,ID=02, NAME=POST,ID=64
 GROUPLIST NAME=MEM,ID=1, NAME=FS,ID=3

/* The following tracepoint is for the routine MixStub.  This was compiled
using segmented addressing and one of the parameters to it is a pointer
to a control block called mix_ctrl.  This pointer, found at SS:SP,
is a flat address because the routine that sent it was compiled with the
flat addressing specification. This logs the mix_ctrl block for 6 bytes. */
      TRACE    MINOR=95, TP=.MixStub, TYPE=(API,PRE), GROUP=MEM,
               DESC="(OS) MixStub     Pre-Invocation",
               FMT ="                            mix_ctrl = %P%W %W %W",
               MEM=(RSS+SP,IF,6)    /* is an indirect flat address */

/* The following is for the routine FlatStub.  This was compiled using
32-bit flat addresses.  A parameter to flatstub is a pointer called
p_seg_info. This pointer is a segmented address because the routine
calling flatstub was compiled using 16-bit segmented addressing.
Value p_seg_info is a 16-bit segmented address.
Log where p_seg_info points for 2 bytes. */

       TRACE   MINOR=0xf0, TP=.FlatStub,
               TYPE=(SYS),
               GROUP=FS,
               DESC="(OS) FlatStub Pre-Invocation",
               FMT="                     seg_info = %P%W",
               MEM32=(.p_seg_info,IS,2)
```

Chapter 10. Steps to Diagnose a Hang

Problems which are called hangs fall into several categories.

The term hang has come into use because there is frequently no way for a user of OS/2 to determine whether the problem is a loop or a wait. The term hang is used in a generic way to mean ′the system does not respond as I expect′, or ′I am unable to interact with the system′. The problem may be a loop, or it may be a wait.

Diagnosing any hang will likely be much quicker if the system trace was used to collect appropriate data related to the symptoms.

10.1 Steps to Diagnose a Wait

Waits are recognized by the fact that no thread is ready. If the scope of the problem is a single application, we need only find out which thread is expected to run, and then analyze why it will not. If we cannot find out which thread we expect to run, we will need to do the analysis for each thread in the process, which will take somewhat more effort. Frequently, the application can be removed by using the window list to end it. If this has been attempted, and has not worked, one must find out why thread 1 will not execute.

If the scope of the problem is the user interface, one needs to examine it, as discussed above. The Workplace shell is discussed elsewhere; remember that from the kernel′s viewpoint, it is just another application. This used to be a much more common symptom than in relatively current releases. It was typically noticed on a LAN server, when requesters received normal service responses, but the system administrator could not use the keyboard or mouse.

Frequently, if you haven′t a well defined place to start, it works reasonably often to look at the blocking data for all threads, and to choose a resource which is needed by many threads. Pragmatically, if that resource could be made available, many threads would unblock. Therefore, choose one of the more popular BlockID′s, and proceed to find out what thread owns it, why that thread will not run, and so on. You may need to do this for only one or two resources before you discover the key thread, and can focus your efforts on it.

If the scope of the problem is that OS/2 refuses to run any thread, the problem must be extremely basic, for example, the drive containing SWAPPER.DAT is no longer available due to a hardware problem, or the system has actually terminated, but has been unable to display that fact.

10.2 Steps to Diagnose a Loop

Loops are also relatively easy to recognize. When one inspects the collective status of all threads in the system, one thread will be in ′run′ status, (if an SMP, one on each processor) and it is likely that many more threads are ready. If the priority of the thread is normal, an application may loop for a long time without the user being aware of the loop, although system performance may suffer somewhat.

To analyze a loop, follow one iteration of it. This is much easier to do with an interactive debugger than it is in a dump.

If the priority of the running thread is in the time-critical class, the dispatcher is designed to prevent OS/2 from dispatching other threads. The looping thread is the cause of the problem, unless the loop is the correct response to another problem. In this case, contact the developer to find out why the thread must be such a high priority, and while you are talking, ask what could cause it to enter a non-ending loop. To diagnose a loop, use an interactive debugger to step through the loop, and try to understand what each conditional jump is really trying to accomplish. You can use an interactive debugger to lower the priority of an offending thread, and observe the results. Recognize that this is quite legitimate, but that the application′s integrity may actually depend on the behavior you have just altered.

Chapter 11. Serialization and Priorities in OS/2

This section describes the various ways to serialize access to resources, which is often required in a pre-emptive multitasking environment.

11.1 Brute Force Serialization

There are several serialization methods which attempt unilateral control over the dispatcher. Each has it own advantages and disadvantages. They will each be explained here before going on to semaphores, which are much more granular, and therefore less intrusive than these serialization methods.

11.1.1 Uniprocessor Method - Disable Interrupts

There is a way for privileged code to guarantee serialization in a single-processor environment, namely to disable interrupts during the actual inspection and update of the protected resource, and then to enable interrupts promptly. The overhead of this method is practically nil, but it is potentially dangerous because it disables pre-emption, which reduces the responsiveness of the system to the user. It also represents a barrier to running successfully on a multiprocessor, because all other processors are unaffected by this, and it therefore requires the developer to re-examine parts of a program which are no longer serialized, but may well be thought to be properly serialized.

11.1.2 Multiprocessor Methods - Spin Locks

In a multiprocessor environment, there are additional problems, namely how to control the additional processors, which may be executing exactly the same instruction at the same cycle. The solution to successfully serializing access to critical structures is solved by using a special instruction prefix, LOCK, which guarantees that all accesses to memory for the following instruction occur as a unit, with no intervening cycles by other processors, DMA devices, or bus masters.

Instructions such as:

Increment(INC), Decrement(DEC), Add(ADD,ADC), Subtract(SUB,SBB),

Logical operations, (AND,OR,XOR,NOT,NEG),

Exchange(XCHG), Exchange&Add(XADD), Compare&Exchange(CMPXCHG)

can be used to claim a resource, add a node to a linked list, and perform other atomic events which normally require serialization, like selecting a ready thread to run. Each processor will use the appropriate method to attempt its task, and if the condition code does not indicate success, it will simply retry the operation until it does complete. This is called a spin loop, and this method of serialization is called a spin lock. It is used when it is expected to be able to access the lock in less time than it will take to save the current context and find another thread to run.

One should not expect to discover the presence of spin locks in a non-multiprocessor environment, because they should always be available, and the spinning should never occur.

11.1.3 DosEnterCriticalSection and DosExitCriticalSection

These API′s will serialize all threads in a process. They have no effect on threads in another process. At the time control returns from DosEnterCriticalSection, no other thread in the process is allowed to execute. Looking at the threads′ status, one will see ′crt′ for all threads in the process which did *not* issue the DosEnterCriticalSection. The only thread which is not marked ′crt′ is the one which has serialized the process. Only when that thread issues the DosExitCriticalSection are the other threads released, and again allowed to compete for use of the processor. This is really too much serialization for most situations, because it temporarily disables multithreading in the process, regardless of the other threads′ design, or current actual processing.

11.1.4 DosSuspendThread and DosResumeThread

Some applications are designed such that there is a limited number of threads which will access some shared resource, and others never will.

To access a resource in a protected way, one can simply suspend the other threads which represent a possibility of simultaneous update, and leave the remaining threads alone. This is therefore less intrusive than the critical section API′s, but still may affect threads which do not represent a threat at this instant, due to other processing, timing, and so on.

DosSuspendThread API will cause a specific thread to no longer compete for the processor, until DosResumeThread is issued. A thread in this situation will have the status of ′frz′. It may not be possible, without an appropriate trace, to find out which thread suspended another.

11.2 Semaphores

The least intrusive way to guarantee serial access to a shared resource is to associate a semaphore with it, and to acquire ownership of the semaphore before accessing the resource. The application threads will be suspended only when there is actual contention for the resource.

This does require all of the programmers involved to be careful to request the semaphore before accessing the resource, and to remember to free it when done. The classic solution to this is to build a low-level function which includes the serialization.

Semaphores are of three categories:

1. Kernel Semaphores, or KSEMS.

   KSEMS will be discussed later, because we will focus first on items available to the application programmer.

2. 16-bit Semaphores.

   There are two basic kinds of 16-bit semaphores, and an add-on structure which makes a third type by aggregation.

   They are the System Semaphore, the RamSem, and the FastSafe RamSem, which is an accounting structure prefixed onto a RamSem.

3. 32-bit Semaphores.

There are two types of 32-bit semaphores, Mutual Exclusion, or MutEx and Event Semaphores. It is also possible to wait on a list of EventSems or MutexSems, but all semaphores in a list must be of the same type.

11.2.1 16-Bit Semaphores

There are three types of 16-bit semaphores, namely system, RAM, and fast-safe RAM semaphores. There are compromises involved in using each.

System Semaphores

These are the most robust of the three, and have the most overhead.

One thread must create the semaphore, with DosCreateSem, which has a name in a format similar to a file name, but in root directory ′SEM′. Other threads must open it with DosOpenSem to get its handle.

Use is to issue DosSemRequest, use the resource, and then to issue DosSemClear so that other threads can access the resource. All threads should issue DosCloseSem before ending.

If a thread ends while owning a system semaphore, the first requestor is given a return code that indicates the situation, so that it is warned of a possibly incomplete update, and may take whatever action is necessary to recover, or terminate.

To find out which thread owns a system semaphore, display a word at the address provided in the blocking data. The address will be a logical address using a GDT selector, generally 400:xxxx. The 12 low order bits are the slot number of the thread which owns the semaphore. If unowned, the value is zero.

RAM Semaphores (RamSems)

At the opposite end of the scale is the extremely fast RamSem.

Most of the speed comes from the following facts:

API′s use the address of the RamSem as the handle.

OS/2 assumes a RamSem is local to a process.

OS/2 does absolutely no accounting for a RamSem.

OS/2 can not provide any recovery for a RamSem.

A RamSem is owned by a user thread if the first byte is hex ′FF′, otherwise it is not owned by a user thread. Unless you have a trace, there is no way to determine which thread owns a RamSem.

Fast-Safe RAM Semaphores (FSRamSems)

The FSRamSem is a compromise between the two earlier types.

The FSRamSem is nothing more than a structure which includes a RamSem. The fields of the structure record the process ID (Pid) and thread ID (Tid) of the thread which owns the semaphore, or zero if unowned. They also include a use count, which is incremented if the owning thread again requests the semaphore. This allows recursive functions to serialize without being blocked, waiting for a resource the thread already owns.

The DosFSRamSemRequest API is used to request the semaphore. It returns when the resource is owned by the thread.

The DosFSRamSemClear API is used to release the semaphore. If the use count is not zero after being decremented, the semaphore is NOT released.

There must be as many ′Clear′ as ′Request′ API calls to actually release the semaphore, and allow other threads to compete for it.

11.2.2 32-Bit Semaphores

There are two classes of 32-bit semaphores, private and shared. There are three types of semaphore in each class, Event, MUTual EXclusion, and multiple wait semaphores.

MUTEX semaphores correspond to one of the most common uses of the 16-bit semaphores, namely to allow competing threads to mutually exclude others from accessing a shared resource.

A MUTEX semaphore includes the slot number of its owner, if owned, or zero if unowned.

An EVENT semaphore contains a post count which is incremented each time it is POSTED, and decremented each time a WAIT for it is completed successfully. This type provides a way to insure that some processing occurs exactly once for each POST.

A multiple wait semaphore is nothing more than a list of semaphores, of the same type. A thread may wait on either ′ANY′ or ′ALL′ of the semaphores in the list.

All Semaphores must first be created with DosCreate???Sem, where ′???′ is the semaphore type. Other processes must open them with DosOpen???Sem to have access to them. Private semaphores have a null pointer to their name, and thus no name. Public ones have a name in the same format as that used for the 16-bit semaphores. DosClose???Sem is used when a thread is through using it.

DosRequestMutexSem and DosReleaseMutexSem are used to access the mutual exclusion semaphores.

DosPostEventSem and DosWaitEventSem are used to access an event semaphore. DosResetEventSem will allow immediate access, and return the post count, which is cleared by this API.

DosQuery???Sem will allow the retrieval of information about each type of semaphore.

DosAddMuxWaitSem and DosDeleteMuxWaitSem are used to add and delete semaphores from a multiple wait semaphore list, respectively.

11.3 Dispatching Priorities

This section describes how the priority of a thread is set, and defines what the classes mean for debugging.

11.4 The Dispatcher, Priorities and Dispatching Classes

The dispatcher′s task is to give control to the proper thread. The definition of proper thread can be difficult to state. My approach to this problem is to state the obvious cases, and then to focus on what is left. In a sense, this discussion parallels the logic in the dispatcher.

1. Idle Class.

   No other class will be pre-empted in order to run an idle class thread. The notion of starved, and the MAXWAIT parameter do NOT apply to Idle Class threads. OS/2 by design will not execute a ready Idle Class thread as long as threads in other classes are ready.

2. Regular Class.

   Most threads are expected to be in this class. All dispatching options and parameters apply to scheduling this thread.

3. Time-Critical Class.

   As long as any thread in this class is ready, OS/2 will give control to it. By design, this may prevent threads in other classes from running. You cannot use priority as a serialization method.

   For example, a page fault will result in temporarily blocking this priority thread.

4. Fixed-High, or Server Class.

   The threads in this class are at a somewhat higher priority than those in the regular class which do not have the focus, but below time-critical.

Note: The numbers above are what the programmer specifies in DosSetPriority, or the 16-bit API DosSetPrty, and are what is returned by DosGetPriority. OS/2 processes these class numbers to create an internal dispatching priority. There are 32 priority levels in each class, which range from 00 to 1F. The priority levels, or deltas, stay the same as the programmer specified initially, if PRIORITY=ABSOLUTE is specified.

The internal priorities have a range from 01 to 08, with 01 usually used for idle-class threads, and 08 usually used for time-critical threads. If PRIORITY=DYNAMIC was specified or defaulted, there are priority boosts given for the following reasons:

- Being the foreground process and for owning the keyboard.
- Yielding the processor before the end of the time slice.
- I/O completion.
- Being starved, that is, ready status and not dispatched for MAXWAIT seconds.

Dispatching is the process of finding the correct ready thread, and then giving control to it. Within each class, the priority delta is used to choose which thread should have control. When several ready threads have the same priority, control is given in turn to each of them, based on the TIMESLICE parameter. The minimum value of this parameter is the duration of the priority boosts which may be applied. The maximum value is the longest a thread can execute before being pre-empted for other threads which have the same internal dispatching priority.

As long as a group of threads at some priority use all the processor, control is not given to lower priority threads. What happens is that the other waiting threads become starved after MAXWAIT seconds have elapsed, and their priority increases until they receive at least a minimum timeslice.

Idle-class threads are never starved, and so will not receive this boost.

11.4.1 How to Display Dispatching Priority

Use the &PERIOD.p command on the slot of interest, and find the pTCB, which is the linear address of the Thread Control Block.

For slot F, below, we see the following for &PERIOD.p output:

```
                      *
Slot Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
000c 0005 0000 0005 0001 blk 0200 7cf7f000 7d1858a4 7d16a0d8 1eb8 00 pgma
0008 0005 0000 0005 0002 blk 081f 7cf77000 7d1858a4 7d169a28 1ea8 00 pgma
000e 0005 0000 0005 0003 blk 021f 7cf83000 7d1858a4 7d16a430 1ea8 00 pgma
000f 0005 0000 0005 0004 blk 061f 7cf85000 7d1858a4 7d16a5dc 1ea8 00 pgma
0010 0005 0000 0005 0005 blk 0200 7cf87000 7d1858a4 7d16a788      00 pgma
000d 0006 0000 0006 0001 blk 0200 7cf81000 7d1860d0 7d16a284 1eb8 00 pgmb
000a 0006 0000 0006 0002 blk 021f 7cf7b000 7d1860d0 7d169d80 1eb8 00 pgmb
0013 0006 0000 0006 0003 blk 0800 7cf8d000 7d1860d0 7d16ac8c 1eb8 00 pgmb

DB  %7D16A5DC+164  L 6           would show ( for slot f )
                                 SEE CAUTION, BELOW
%7D16A740   02  1F  xx  xx  1F  06
    class  /   /            ------  the word value 061F
      level  /              actually used by the dispatcher
```

CAUTION: the offset used is correct for Warp CONNECT, but the addresses are what were used in OS/2 2.11, so this is a somewhat mixed example. Any offset in any control block may change any time a fix or new version is installed. Please refer to the Thread Control Block in the System Diagnostic Reference for offsets relating to other versions.

The first byte is the programmer′s priority class, ranging from 1 to 4. The second byte is the level within the class, ranging from 00 to 1F. The third and fourth bytes are not useful. The fifth and sixth bytes are OS/2′s computed dispatching priority. This field is a word, so the high order part is byte 6. 081F is the highest possible value and 0100 is the lowest possible value.

On a uniprocessor, using DosSetPriority to make yourself time-critical at the highest delta value would give you an extremely good chance of not being pre-empted, and was occasionally misused for serialization. This will never work on a multiprocessor, and is risky even on a uniprocessor, because a page fault will cause you to lose control while the page is processed, just as doing I/O to a file will cause a thread to block if access to the actual device is required.

11.4.2 The Status of a Thread

A thread can be in one of several states. The following list is an attempt to list all the possible states, and to briefly discuss each.

run This thread is currently executing.

rdy This thread would like to run, but higher priority thread(s) are executing, which prevents this thread from running.

blk This thread is blocked. Use the **.pb** command to find out what resource (BlockID) it is blocked on (waiting for).

crt This thread cannot be run because another thread in this process is currently in a critical section. You can identify that thread because it will be the only thread not marked 'crt'.

frz This thread is frozen, that is, some other thread has called the API DosSuspendThread and passed the ID of this thread. This thread cannot execute until some thread issues DosResumeThread to inform OS/2 that this thread is once again eligible to be dispatched. There is no way to discover what thread suspended it.

Chapter 12. A Form to Use for Unwinding Stacks

Frame at	Caller's Frame	Return Offset	Return Selector	Parameters			

Chapter 13. Advanced Guide to Hang Analysis

What is a *hang?*

It's an external symptom or a user perception that little or no work is being done. It could be a case of extremely poor performance. The hang symptom categorizes itself into three distinct cases:

Wait

Threads and processes are not being dispatched by the operating system. Thread status gives the initial clue. Use of the `.P` command determines status.

Blocking

Threads may wait voluntarily for a resource or an event, in which case they will probably be blocked. They might poll for the resource, in which cases they will cycle through blocking, being ready and running until the resource becomes available.

A notoriously problematic case of blocking is the deadlock. This is where two threads are each own exclusively a resource and block waiting for ownership of the other's resource.

Suspension

Another thread may have deliberately debarred a thread from being scheduled, in which case we will see the waiting thread in a *crt* or *frz* state.

Preemption

Another thread monopolizes the system. Typically the hanging threads will be ready for dispatch (*rdy*), but will never or rarely receive a minimum time-slice. We look for running and ready threads with an excessively high priority. The `.P` will give the calculated priority of each thread.

Disabled wait

Looks rather like a hardware freeze. Last instruction executed was a *HLT* having sometime previously disabled interrupts using *CLI*. This usually happens only when ring 0 code detects an terminal condition. One would hope that some form of diagnostic information had been displayed prior to this particularly if NMIs have been disabled also! If NMIs have not been disabled then an artificially generated channel check may be used to cause an NMI, which would allow one to break into the kernel debugger. However the KDB to allows only one NMI channel check per boot. If NMIs are disabled then hardware analysis techniques may be the only recourse.

Loop

A thread is running more or less permanently. It's state will mostly be *rdy* or *run*. Similar analysis techniques to trap apply here. We examine the registers of the running thread by using `.R`. From the we can determine in which module the thread is running by looking at the owner of the executing code segment. If necessary we unwind the stack and determine the caller etc.. Analysis is very similar to trap analysis.

Hardware Freeze
The processor fetch-execute cycle has been suspended. Not even an NMI interrupt will resume instruction fetch. This is almost certainly a hardware problem. Timing/clocking problems caused by incompatible cards, overloaded busses, incorrect bus terminations, faulty processor or controller chips. Use hardware techniques such an ICE machine or Logic Analyzer.

The following Theory Topics are now covered in detail:

- See 13.1, "The Wait Condition - Diagnostic Techniques."

  This is further subdivided into two topics of discussion:

  13.1.1, "Memory Management and Ownership Topics."

  13.1.2, "Thread Scheduling and Dispatching Topics" on page 175.

- 13.2, "Program Design Issues and Weaknesses" on page 211.

The final section of this Guide is a collection of Chapter 14, "Worked Examples" on page 213 that explore memory management, the file system, Presentation Manager and Ring 0 loops from a dump.

13.1 The Wait Condition - Diagnostic Techniques

In most problem analyses the question of memory ownership or use will arise. For example:

To which module does this instruction belong?

Which process executed this module?

Who allocated this storage?

Who passed these parameters?

What control block does this address point to?

In fact the frequency with which this question is asked makes it a fundamental aspect of analytical technique.

For hang analysis this is no less true.

In the case of loops, analysis proceeds in a similar manner to that of traps.

In the case of waits, a key piece of information is the BlockID. In many cases this is an address of a system control block that relates closely to the reason for waiting. Discovery of the owner of storage pointed to by a BlockID is therefore of prime interest.

We start by reviewing memory management in OS/2 and in particular memory ownership.

13.1.1 Memory Management and Ownership Topics

Memory allocation in a demand-paging virtual storage operating system such as OS/2 embodies the allocation of a number of system resources with certain attributes applied:

Resources

Virtual address space

Real address space

Auxiliary address space (SWAPPER)

Attributes

Exclusion (privacy)

Inclusion (sharing)

Owner (Where is was allocated from or who it was allocated to)

Requestor (who made the request on behalf of the owner)

Permissions (Read-only, Read/Write, Executable)

Use of the resources and the attributes applied is tracked by the system in its VM control blocks. The most important of these are the following&cvolon.

VMAH The Virtual Memory Arena Header Record

VMOB The Virtual Memory Object Record

VMAR The Virtual Memory Arena Record

VMCO The Virtual Memory Context Record

PF The Page Frame Structure

VP The Virtual Page Structure

13.1.1.1 Virtual Address Space Arenas and Regions

OS/2 partitions the 4GB virtual address space into three types of arena:

System

Shared

Private

The system arena is common to all processes. It starts at the 512MB boundary and occupies the address space up to 4GB. Only system code (and device drivers) can access data in the system arena directly. User code must use APIs invoked by the call gate mechanism to access system arena code and data. Nearly all system arena data is global, that is, managed by a common set of page tables, whatever the current thread/process context. The exception to this is in the memory area mapped by selector 30. Page table entries are adjusted as part of context switching so that selector 30 addresses the current PTDA, TCB and TSD.

The shared arena address range is common to all processes, but it comprises data that is both global and instance. Instance data occurs where a separate set of page table entries are used per context to map the same linear address range.

Instance data is used when the same type of data needs to be allocated as multiple private copies to each process. An example of this would be a logical screen buffer. The shared arena starts initially at the 304MB boundary and ends at 512MB. User programs may access the shared arena. DLL code and data is located in the shared arena. DLL code segments are always global, but DLL data segments may be instance or global and are usually a mixture of both.

The shared arena is further subdivided into a number of regions:

Region	*Description*
Protected	This region is reserved for protected data segments of protected DLLs. In General 16- and 32-bit applications do not have addressability above the 448MB boundary. Potentially 32-bit applications are able to modify all read/write global data, whether intended by the owning DLL or not. The protected region is provided so that protected DLLs can isolate their data from general access. Only protected DLLs have addressability to the protected region by being assigned data selector 63. 32-bit DLLs become protected through use of the protect option at compile time. 16-bit DLLs may also use the protected region, if explicitly coded to do so and listed in CONFIG.SYS using: `PROTECT16=dll1,dll2,....` The protected region may be subsumed into the based region (see below) by coding in CONFIG.SYS the NOPROTECT option on the MEMMAN statement. The default is MEMMAN=PROTECT
Based	The based region is reserved for non-protected DLLs that have a preferential base address assigned by the linkage editor by using the BASE option. The purpose of the based region is to improve performance of module loading, by avoiding the need for the system loader to do fix-up processing. **Note:** Under OS/2 2.x, MEMMAN=NOPROTECT would cause the based and packed regions to move up 64M bytes - effectively giving another 64M bytes for general purpose use in the shared arena.
Packed	The packed region is reserved for 16-bit DLL code segments. Within the packed region the compatibility region mapping algorithm does not apply. Code segments are packed contiguously to make best use of physical pages. Potentially, tiny DLL code segments can deplete physical storage very rapidly if not packed. However, when packing is used there is no general algorithm that will convert 16-bit addresses into 32-bit addresses within the Packed Region. The system has to scan the LDT, over the packed region, to make this conversion. Packing may be disabled by specifying the NOPACK option of the MEMMAN statement in CONFIG.SYS. The default is PACK. When packing is disabled up to 32M bytes is made available to the Global Shared Region.

Notes

Under OS/2 2.x only MEMMAN=NOPACK would tend to reduce the Swapper Size where a great many 16-bit code segments are in use. This is because code segments outside the Packed Region are normally discardable (they are not swapped). Within the packed region they are swappable since a 4K page may contain code from a number of different modules.

Under OS/2 2.x MEMMAN=NOPACK would provide up to 32M bytes extra virtual address space for general purpose use in the Shared Arena.

Packing does not affect the availability of LDT selectors for allocations in the Packed Region, just the base linear address boundaries on which they are deployed.

Packing should not be confused with either the LINK386 PACK option or the PACK.EXE utility.

Global Shared

This region has a lower boundary at 320MB and includes the packed, based and protected regions. This is reserved for Global Read-Only allocations only. Since no read/write data is allocated in the global shared region some page table economies are possible. Also process context switching performance is improved.

Notes

The Global Shared Region is not configurable.

The Global Shared Region is only implemented in OS/2 WARP version 3.

Under OS/2 2.x read/write segments would be allocated in the based region.

Read/Write Basing

The read/write basing region is the preferred region for locating read/write DLL data segments where a base address has been assigned to a DLL module by the linkage editor. The purpose of this region is to keep read/write segments out of the Global Shared Region and thus retain its performance advantages. It also places an upper bound on the location of dynamic shared allocations, namely the Minimum Read/Write Basing Region address.

Notes

The Read/Write Basing Region is not defined in OS/2 versions prior to version 3.

Based DLLs under OS/2 2.x, by preference, have their segments loaded sequentially starting with segment 1 at the base address. With the implementation of the Global Shared Region only Read-Only segments can be loaded sequentially from the base address.

Expansion

The shared arena is an *expand-down* arena, that is, allocation searches for free regions from the high addresses to low. The shared arena therefore expands from the minimum read/write basing region address towards the highest upper bound of all the private arenas. This area is the expansion region for both the shared arena and all the private arenas.

The shared arena will not contract to an address higher than the minimum read/write basing address.

Note:

The expansion region for OS/2 2.x is from the lower boundary of the packed region, if present. If not, then from the lower boundary of the protected region, if present. If both the protected and packed regions are removed (using MEMMAN=NOPACK,NOPROTECT) then expansion occurs from the top of the shared arena.

Each private arena occupies the lowest range of virtual address space from 0 - 64MB expanding up to a maximum of 304MB, the minimum read/write basing address. None of the private arenas will be allowed to expand beyond the lowest allocation in the shared arena, that is private and shared arenas may not overlap.

In general each process uses a separate set of page table entries to map each page of its private arena. Thus the data in the private arena is private to each process. Code (.EXE files) however is treated differently. Since code is read only an economy is made whenever more than one process runs the same .EXE. Where this happens the same page table entries are used among the processes sharing the common .EXE file. User programs may only access the private arena of the process they are running in (a special exception to this is possible through the DosDebug API by defining memory aliases).

Virtual memory arenas and regions may be presented pictorially as in the following diagram.

Notes

Some regions of the 4GB address space are reserved. This is done for a variety of reasons which include:

- Hardware and BIOS restrictions.
- Enforced segregation between Arenas.
- Guaranteed reserved address ranges.

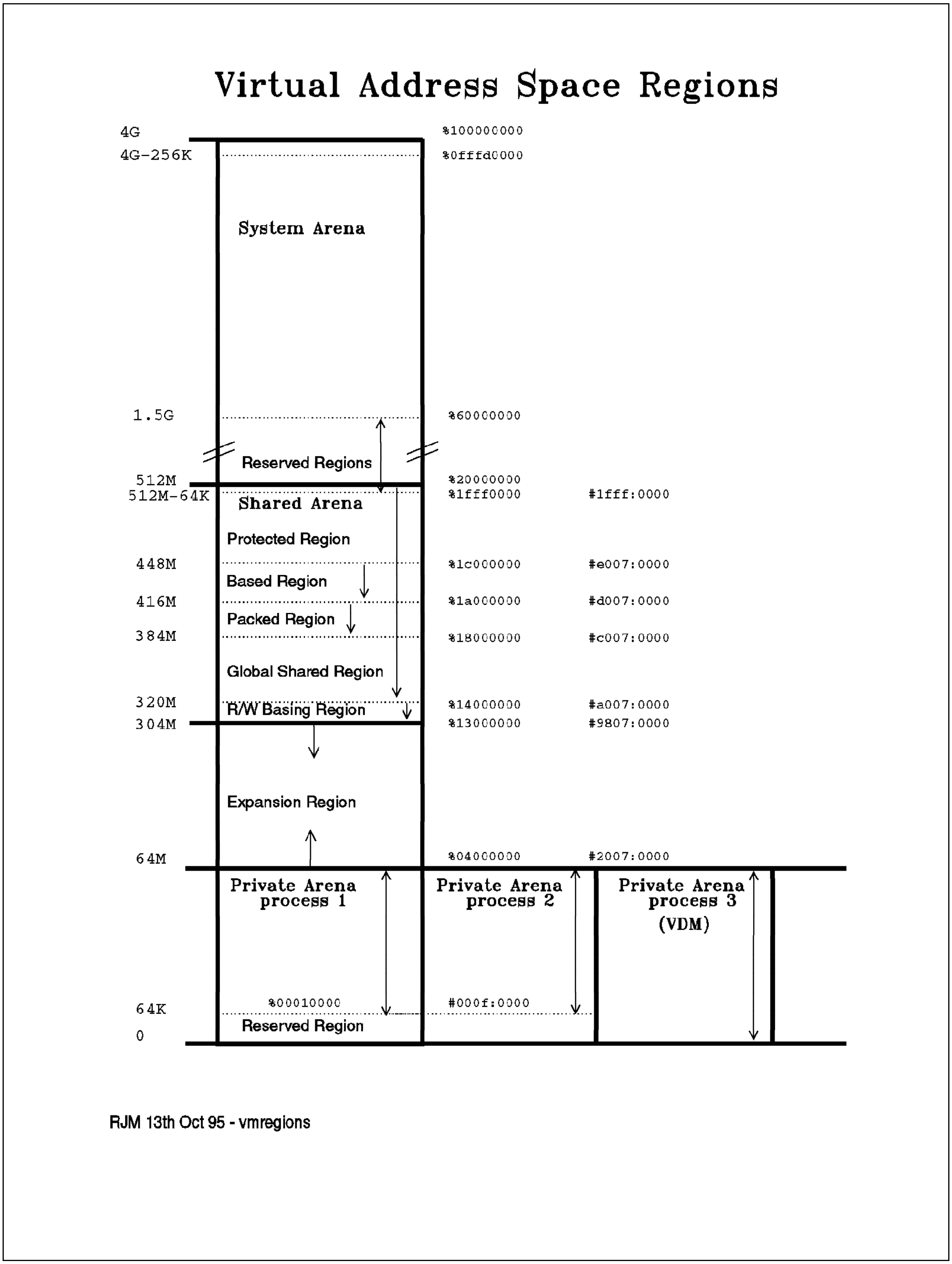


Figure 2. Virtual Address Space Regions

13.1.1.2 Virtual Address Space Management

Each of the three types of arena discussed in the previous section is managed by:

- An Arena Header Record (VMAH)
- A Sentinel Arena Record (VMAR)

The VMAHs are maintained in a double-linked list. They contain information about the extent to which an arena has been used. Of particular interest are the following fields:

+0x0
Pointer to the next VMAH.

+0x4
Pointer to the previous VMAH.

+0x8
Pointer to the Sentinel Arena Record for this arena.

+0xc
Pointer to the VMAR adjacent to the 1st free area.

In the case of expand down arenas (the shared arena), this is the VMAR for the region of memory allocated above the first free area below the minimum read/write basing region.

In the case of expand-up arenas (system and private) this is the VMAR for the region of memory allocated just below the lowest free area.

+0x20
Current minimum linear address allocated.

+0x24
Current Maximum linear address allocated.

VMAHs are located:

- At _vmahSys for the System Arena
- At _vmahShr for the System Arena
- Imbedded at +0x40 in each PTDA for Private Arenas

Arena Records (VMARs) are used to describe virtual storage reservations. These are described in more detail in 13.1.1.3, "Virtual Memory Arena Records" on page 150.

A special form of the VMAR is the Sentinel Arena Record. This serves two purposes:

- To track the theoretical size limits of an arena.
- To act as the head to a double-linked list of regular VMARs, each of which describes a specific allocation.

The sentinel VMAR for the Shared Arena is called the Boundary Sentinel, since it determines where the (dynamic) boundary between shared an private arenas lies. The boundary is adjusted to reflect the current highest private arena address.

The manner in which VMARs and VMAHs are organized to manage the three types of arena is shown in the following diagram:

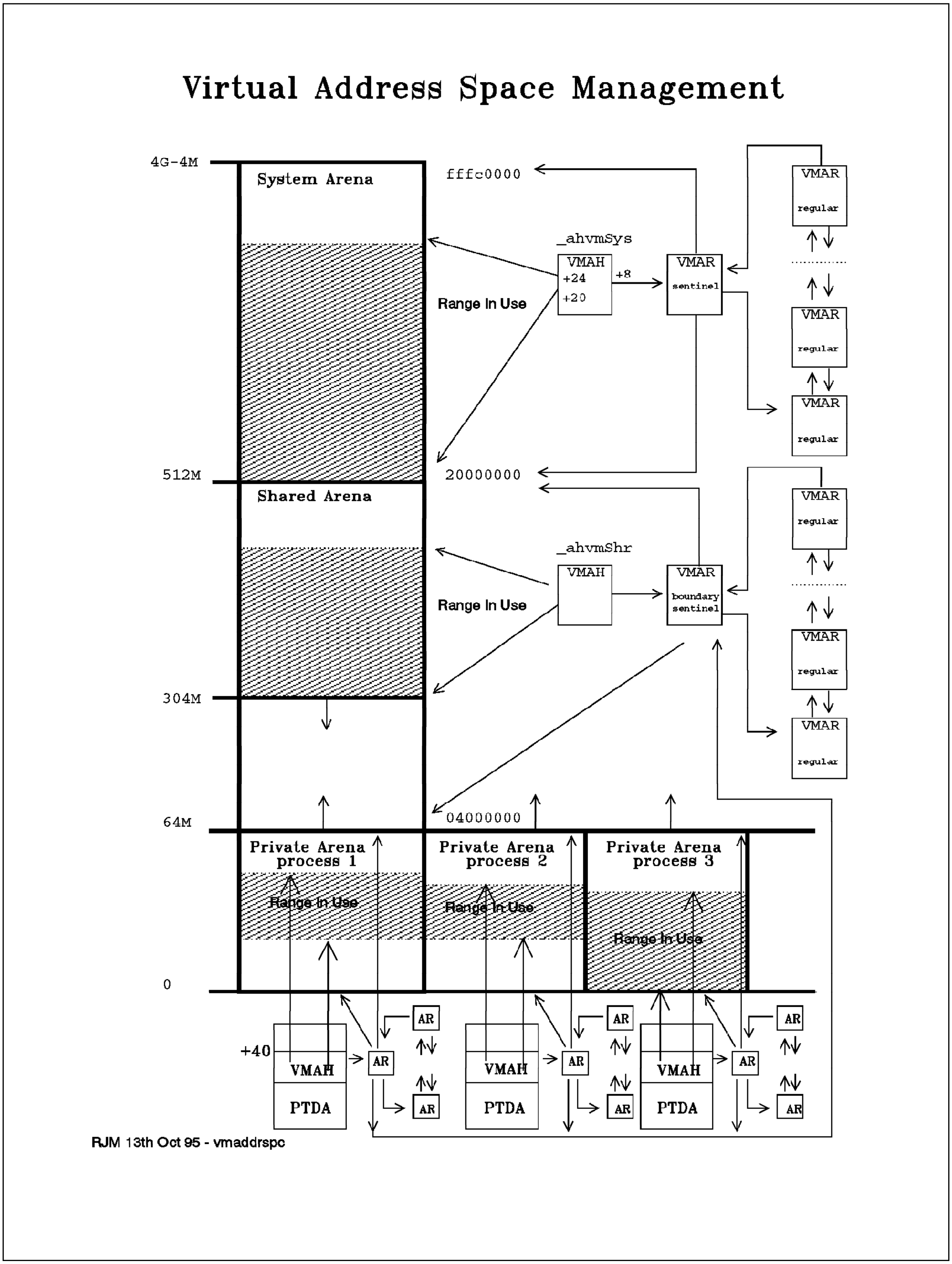


Figure 3. Virtual Address Space Management

13.1.1.3 Virtual Memory Arena Records

VMARs are 24-byte records that describe virtual address space allocation, or reservation. They are located in a table in system memory. The principle use of the VMAR is to track the allocation of virtual memory, which may or may not be backed in RAM or on the SWAP file.

Arena records appear in a number of guises depending on the area storage they describe and whether the storage is shared, instance or private data. They are formatted by the KDB and DF `.MA` command. The `.MA` command takes either the handle or address or the VMAR as a parameter, or if no parameters are specified then the entire VMAR table is formatted.

```
# .ma
 har    par      cpg        va    flg next prev link hash hob   hal
 0001 %feeef020 00000100 %ff050000 001 001f 0002 0000 0000 0001 0000      =0000
 0002 %feeef036 00000161 %feeef000 001 0001 0003 0000 0003 0002 0000      =0000
 0003 %feeef04c 00001000 %fdeef000 001 0002 0021 0000 0000 0003 0000      =0000
 0004 %feeef062 00000000 %60000000 003 05d3 0015 0000 0000 ffc0 0000 max=%fffc0000
 0005 %feeef078 0000cc40 %04000000 007 0617 0072 0000 0000 fff0 0000 max=%1fff0000
 0006 %feeef08e 00000003 %fff1b000 009 000f 03ad 0000 0000 0007 0000   sel=0100
 0007 %feeef0a4 0000000c %ffe22000 009 0008 0019 0000 0000 0008 0000   sel=0400
 0008 %feeef0ba 0000000d %ffe2e000 009 0009 0007 0000 0000 0009 0000   sel=1000
 0009 %feeef0d0 00000010 %ffe3b000 009 01e2 0008 0000 0000 000a 0000   sel=0120
 000a %feeef0e6 000001c1 %ac624000 121 002a 003a 0000 0000 000b 0000      =0000
 000b %feeef0fc 00000006 %ffe4d000 009 000c 01df 0000 0000 000c 0000   sel=0130
 000c %feeef112 00000003 %ffe53000 009 0094 000b 0000 0000 000d 0000   sel=0138
 000d %feeef128 00000010 %11450000 379 0394 02d6 0000 0000 000e 0000  hco=00174
 000e %feeef13e 00000001 %fff10000 001 03d3 0020 0000 0000 000f 0000      =0000
 .
 .
 00b6 %feeeffae 00000080 %00110000 169 03df 0076 0000 0000 041f 0000 hptda=03c9
```

The fields of principle interest are:

har
The arena record handle. This is a unique identifier assigned to each VMAR.

cpg
The number of pages (4K bytes) allocated or reserved.

va The address of the first page in the reservation.

hob
The handle of the VMOB that occupies the virtual address range covered by va and cpg.

The right-hand column gives descriptive information about the use of the address range in a VMAR. Of particular interest are:

sel=ssss
Indicates system storage mapped by a GDT selector.

hco=hhhh
Indicated shared global storage. The **hco** is the handle of the VMCO at the head of the list representing accessors to an allocation in the Shared Arena.

hptda=pppp
Indicated private memory allocated in the private arena of a process whose PTDA handle is pppp.

13.1.1.4 Virtual Memory Object Records

VMOBs are 16-byte records allocated contiguously in a table in system memory. Each table entry is numbered from 1 and is referred to as a memory object handle, or more simple as a hob.

VMOBs are use to store information about the allocation request. Of particular interest are:

- The Requestor
- The Owner
- The Permissions

The VMOB also has links to other related control blocks. Of these the important ones are:

The associated VMAR.

The associated VMCOs.

The associated VMOBs.

VMOB is formatted by using the KDB or DF `.MO` command. The `.MO` command takes either the handle or address or the VMOB as a parameter, or if no parameters are specified then the entire VMOB table is formatted.

```
##.mo
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0001  0001 fec8  0000 fff1 0000  0000 00  00 00 00 vmob
 0002  0002 fec8  0000 ffe3 0000  0000 00  00 00 00 vmar
 0003  0003 fec8  0000 ffec 0000  0000 00  01 00 00 vmkrhrw
 0004  %fff13238  8000 ffe1 0000  0000 00  00 00 00 vmah
 0005  %fff13190  8000 ffe1 0000  0000 00  00 00 00 vmah
 0006  %fff0a891  8000 ffa6 0000  0000 00  00 00 00 mte       doscalls.dll
 0007  0006 0000  0000 ff6d 0000  0000 00  00 00 00 doshlp
 0008  0007 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0009  0008 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000a  0009 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000b  000a 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000c  000b 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000d  000c 0000  0325 ffba 0000  0000 00  00 00 00 lock
 000e  000d 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000f  000e 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0010  008f 0000  402c 00ae 0115  0000 00  00 00 00 priv 0002 c:pmshell.exe
 0011  0010 0000  0000 ff37 0000  0000 00  00 00 00 romdata
 0012  0011 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
```

One VMOB is formatted per line with the hob in the left-hand column.

The owner is shown under the **own** column and is given as a hob that is associated with and uniquely identifies where the allocation is made from. For example:

Memory dynamically allocated within a Private arena uses the handle of the PTDA (hptda) as the owner.

The PTDA has a number of characteristics that make it an ideal choice for an owner:

Each process has a unique PTDA

The PTDA is the central control block from which all information about a process is obtained.

Each PTDA is allocated from a unique memory object so has a unique hob, which defined to be the hptda.

For storage allocated for a load module segment the module MTE handle (hmte) is used.

The MTE has a number of characteristics that make is an ideal choice for an owner:

Each loaded module is represented in the system by an unique MTE.

Each MTE is allocated from a unique memory object so has a unique associated hob, which is defined to be the hmte.

Memory allocated in the shared arena which is not specific to a particular process uses the following conventions for owner:

- For DLL instance and global data the owner is the hmte of the owning DLL.
- For giveable shared storage, the owner is (0xfff5).
- For gettable shared storage, the owner is (0xfff6).
- For giveable and gettable shared storage, the owner is (0xfff7).
- For named shared storage, the owner is (0xff82).

See DosAllocSeg, DosGiveSeg, DosGetSeg and DosAllocSharedMem APIs in the Control Program Programming References for OS/2 1.x, 2.x and 3.x.

Memory allocated or suballocated from the system arena uses an artificial system owner id (ffxx) that doesn′t actually correspond to a VMOB but is a conventional handle used to indicate the type of system object which has been allocated. An example this is hob 1 which is the table of VMOBs.

The requestor′s id is shown in the hmte column. This field is either:

The hmte of the module making the request.

An associated system object

zero where there is no associated requestor.

To the right of each line appears a textual interpretation of the own and hmte fields.

The handle of the associated VMAR is shown in the **har** column.

Associated VMOB′s that share the same virtual address (that is, instance data) are linked from the hobnxt field.

Not every VMOB is linked to an associated VMAR, as seen in hobs 4 and 5 in the example. These are known as pseudo-objects. They are used for some small system control blocks that are allocated, as required, from system storage but are too small to warrant the overhead of the minimum allocation of 1 page, which an arena records implies. PTDAs and MTEs are the most frequently encountered pseudo-objects. The va field replaces the har and hobnxt and points directly at the object itself.

13.1.1.5 The Virtual Memory Context Record

VMCOs are small control blocks that serve as extensions to the VMAR for shared arena, shared data. Whenever a process is given access to a shared global data object (most frequently DLL code and global data) then a VMCO is used to record the handle of the process (hptda) of the accessing process. Each process that shares a global data object will have a VMCO chained in a single-linked list from the object's VMAR.

VMCOs may be formatted using the KDB and DF `.MC` command, however they are usually displayed with their corresponding VMOB and VMAR by using either the `.MOC` or `.MAC` commands.

```
#.mac 297

*har     par      cpg        va     flg next prev link hash hob   hal
 0297 %feef2904 00000660 %11fb0000 369 0312 0295 0000 009e 02a1 0000  hco=0057f
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 02a1  0297 0000  4a2d fff5 0302  0000 00  00 00 00 give
 hco=057f pco=ffe70b96  hconext=00241 hptda=06d1 f=1e  pid=0059
 hco=0241 pco=ffe6fb60  hconext=004ce hptda=04b2 f=1e  pid=0043 c:pmspool.exe
 hco=04ce pco=ffe70821  hconext=0034c hptda=05c3 f=1e  pid=0016 c:pawn.exe
 hco=034c pco=ffe70097  hconext=001e4 hptda=04ca f=1e  pid=000f c:pmshell.exe
 hco=04ca pco=ffe7080d  hconext=00348 hptda=05c3 f=16  pid=0016 c:pawn.exe
 hco=0348 pco=ffe70083  hconext=0017a hptda=04ca f=16  pid=000f c:pmshell.exe
 hco=017a pco=ffe6f77d  hconext=00177 hptda=03d9 f=16  pid=000b c:spdaemon.exe
 hco=0177 pco=ffe6f76e  hconext=00148 hptda=03ec f=16  pid=000c
 hco=0148 pco=ffe6f683  hconext=000b2 hptda=03c9 f=16  pid=000a c:ddaemon.exe
 hco=00b2 pco=ffe6f395  hconext=00083 hptda=0359 f=16  pid=0009 c:harderr.exe
 hco=0083 pco=ffe6f2aa  hconext=00081 hptda=02e1 f=16  pid=0007 c:landll.exe
 hco=0081 pco=ffe6f2a0  hconext=0007d hptda=02ad f=16  pid=0006 c:lanmsgex.exe
 hco=007d pco=ffe6f28c  hconext=00037 hptda=027a f=16  pid=0008 c:pmshell.exe
 hco=0037 pco=ffe6f12e  hconext=00000 hptda=02ac f=16  pid=0004 c:gambit.exe
```

13.1.1.6 Private Arena Private Data

Private data, that is data in a private arena not accessible from any other context, is managed by VMARs and VMOBs as depicted by the following diagram.

Control blocks and data that directly represent the allocation are shown shaded.

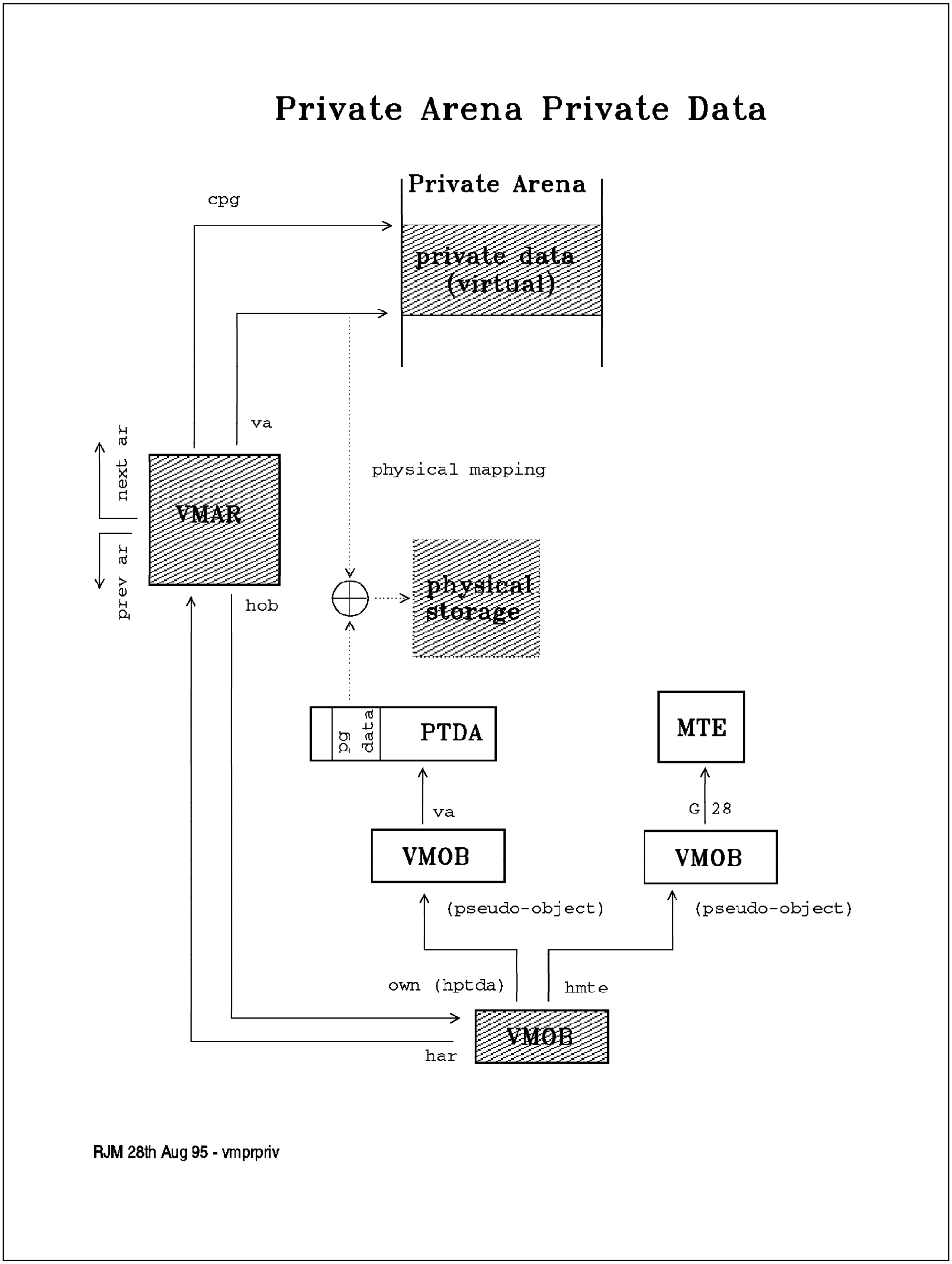


Figure 4. Private Arena Private Data

13.1.1.7 Private Arena Shared Data

This is the case where .EXE program Read/Only segments are shared across multiple private arenas.

The following diagram depicts this situation.

Note that only one VMOB is used, but there are multiple VMARs, one for each process accessing the allocation. Each VMAR is linked using the link field.

Control blocks and data that directly represent the allocation are shown shaded.

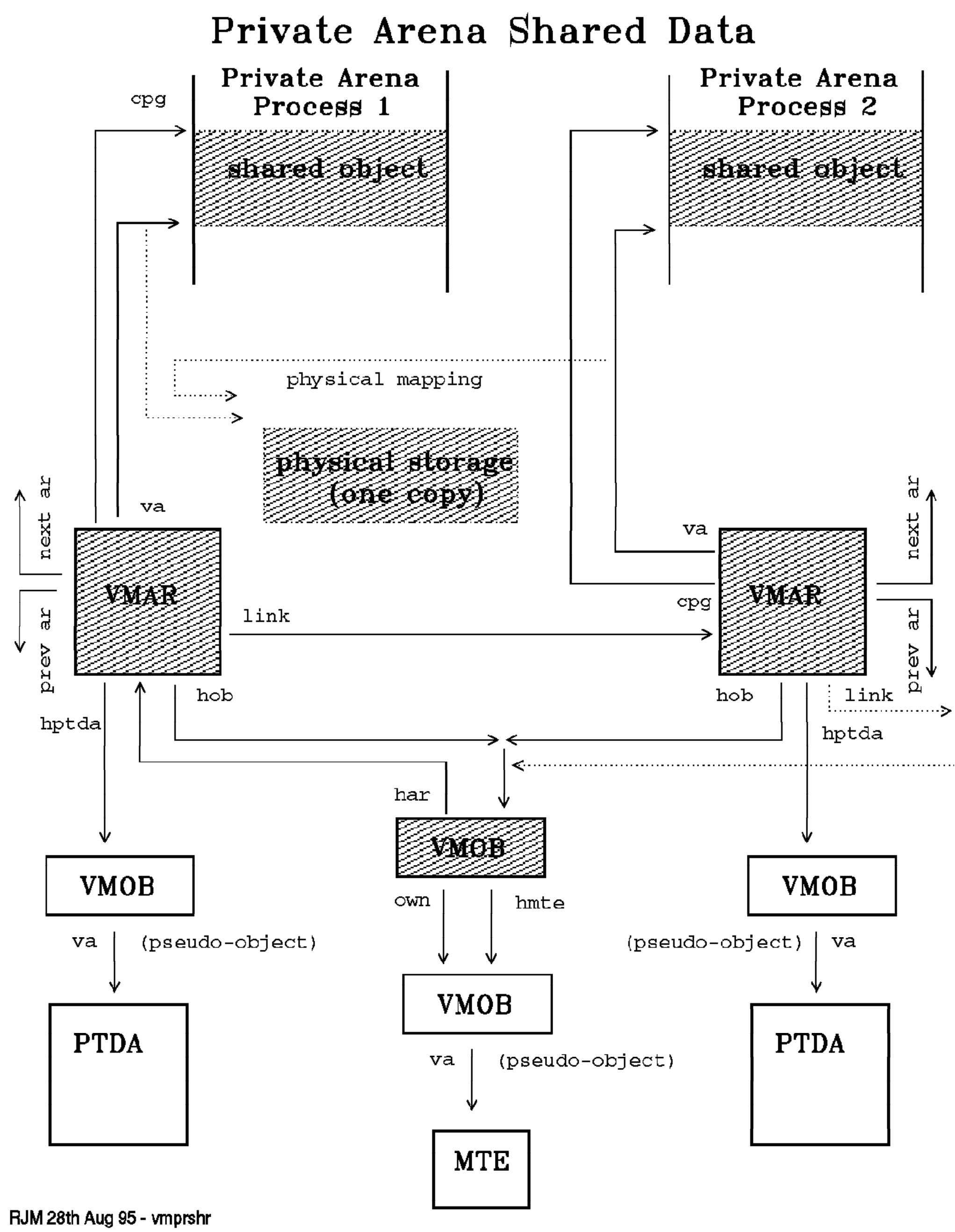


Figure 5. Private Arena Shared Data

13.1.1.8 Shared Arena Global Data

DLL global data and named shared, *giveable* and *gettable* allocations are potentially *shareable* among multiple processes. With these types of allocations data and address range is common to all who access it. Those that are given access are recorded by the VMCO chain.

With this type of allocation, there is only one VMAR and VMOB.

Note that the own field of the VMOC, which is interpreted on the right hand side of the `.MO` display, may be one of five possibilities:

hmte Data is global data or code segment of a DLL.

Give Data is allocated with the give attribute.

Get Data is allocated with the get attribute.

GiveGet Data is allocated with both the give and get attributes.

Mshare Data is named shared storage.

The following diagram depicts this situation. Control blocks and data that directly represent the allocation are shown shaded.

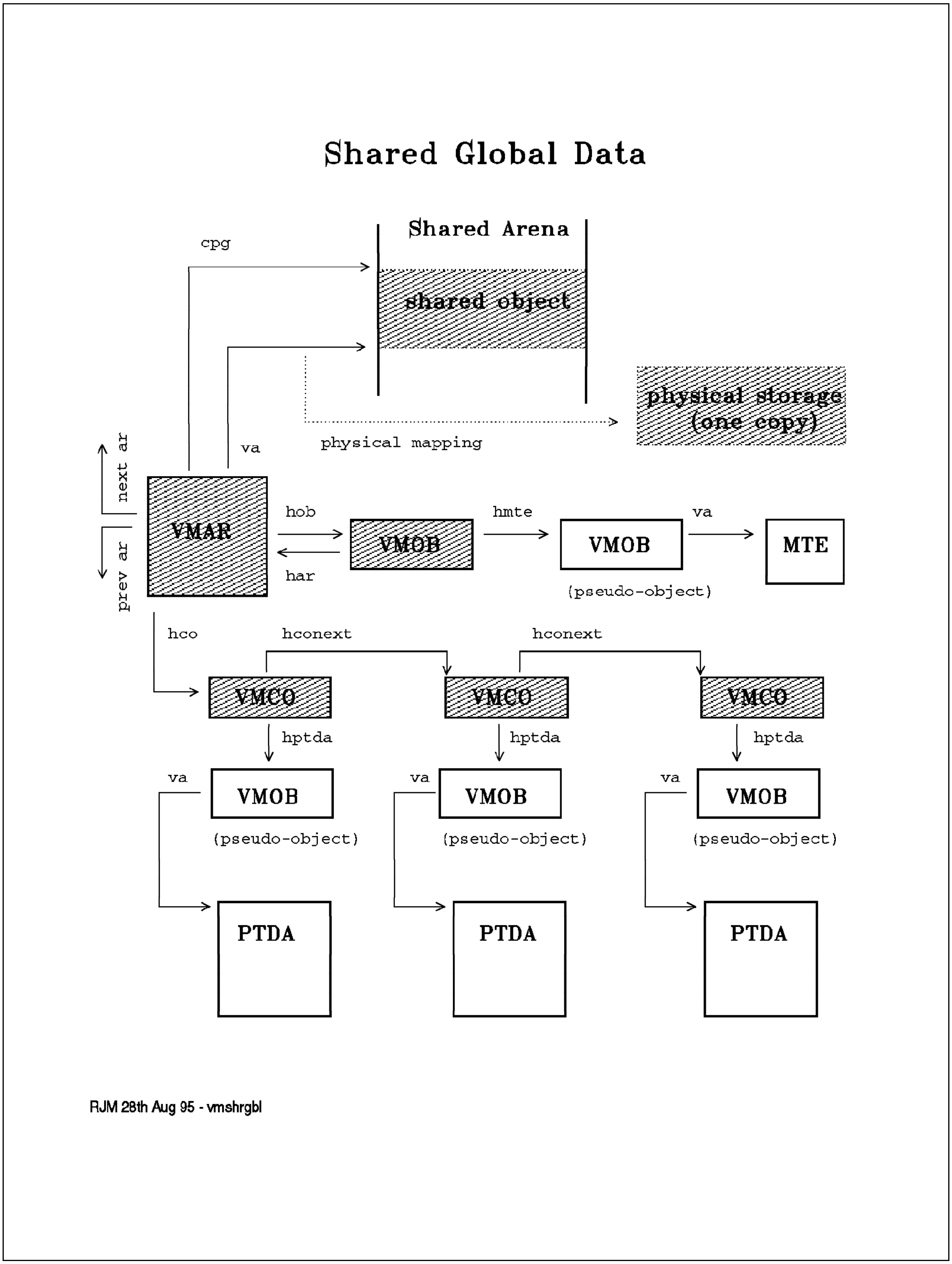


Figure 6. Shared Global Data

13.1.1.9 Shared Arena Instance Data

A DLL Instance data allocation shares only its address range among its accessors. The data is mapped to a different set of physical pages for each process.

This type of allocation is represented by a single VMAR with a chained list of VMOBs, one for each accessor. This is the only case where VMOBs are linked by the hobnxt field.

The following diagram depicts this situation. Control blocks and data that directly represent the allocation are shown shaded.

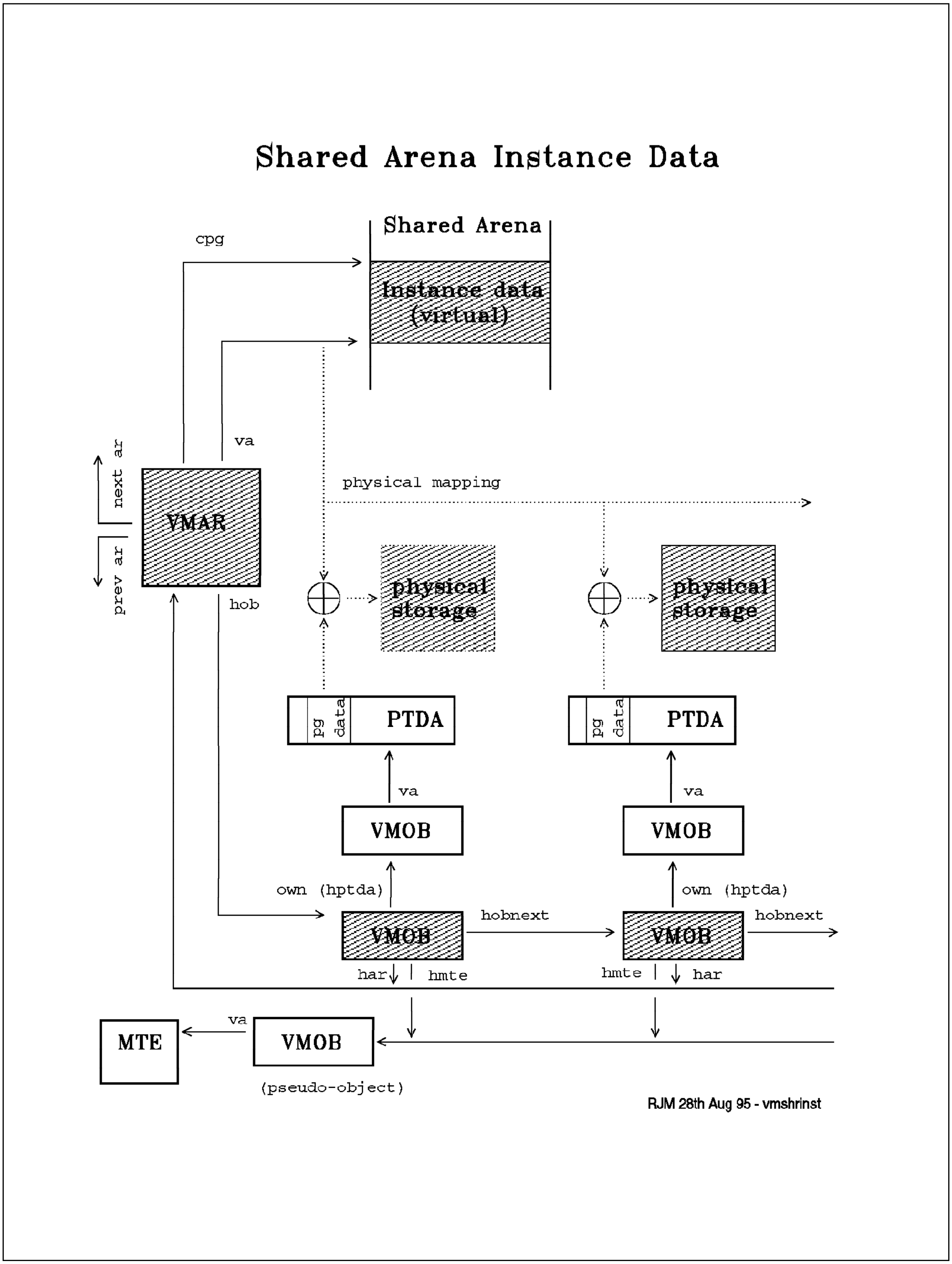


Figure 7. Shared Arena Instance Data

13.1.1.10 The Page Frame Structure

Occasionally we need to enquire into the ownership and disposition of real storage. The PF is used to track the use of all frames of real storage, whether allocated, idle (pending freeing) or free.

The PF is formatted using the **.**MP KDB and DF command. **.**MP will optionally take the frame number (real address MOD 4K) as a parameter.

```
# .mp
ffe1b000 InUse: pVP=ff406000 RefCnt=0001 Flg=0 ll=00 sl=00 Blk=00000 Frame=00000
ffe1b00c InUse: pVP=ff406050 RefCnt=0001 Flg=0 ll=00 sl=00 Blk=00000 Frame=00001
ffe1b018 InUse: pVP=ff40605a RefCnt=0001 Flg=0 ll=00 sl=00 Blk=00000 Frame=00002
ffe1b024 InUse: pVP=ff406064 RefCnt=0002 Flg=0 ll=00 sl=00 Blk=00000 Frame=00003
ffe1b030 InUse: pVP=ff40606e RefCnt=0001 Flg=0 ll=00 sl=00 Blk=00000 Frame=00004
```

Of particular interest are:

Frame=fffff
The real storage frame number represented by this PF.

pVP
The address of the related Virtual Page Structure (non-free PFs only). See next section.

ll The long term lock count.

This is non-zero when an otherwise non-resident page is long-term locked, that is prohibited from being discarded or swapped, and expected to be so for a relatively long time.

sl The short term lock count.

This is non-zero when an otherwise non-resident page is short-term locked, that is prohibited from being discarded or swapped, but temporarily so (much less than ten seconds).

13.1.1.11 The Virtual Page Structure

VPs track the disposition of every page of virtual storage of every object. They enable the system to locate the data for the page, whether it is in RAM or on the swap file.

VPs are formatted using the **.**MV KDB and DF command, which takes as a parameter the address of the VP, which may be obtained from the PF structure.

```
.mv %ff4060f0 l5
VPI=0018 pVP=ff4060f0 Res Frame=0011 Flg=410 HobPg=0001 Hob=0009 Ref=001
VPI=0019 pVP=ff4060fa Res Frame=0012 Flg=410 HobPg=0002 Hob=0009 Ref=001
VPI=001a pVP=ff406104 Res Frame=0013 Flg=410 HobPg=0003 Hob=0009 Ref=002
VPI=001b pVP=ff40610e Swp Block=08cc Flg=0a0 HobPg=027b Hob=0026 Ref=001
VPI=001c pVP=ff406118 Swp Block=0001 Flg=000 HobPg=0000 Hob=00e7 Ref=001
```

Of particular interest are:

Hob=nnnn
The hob of the object to which this page belongs.

HobPg=nnnn
The page number within the object.

Frame=ffff
The real storage frame number that backs this virtual page.

Block=bbbb
The Swap file 4K block that contains the data for this virtual page.

13.1.1.12 Page Management

The relationship between PF structures, page table entries, swap file blocks and memory objects is shown in the following diagrams.

The relationship between PF structures, VP structures, page table entries, swap file blocks and memory objects is shown in the following two diagrams

The first of these depicts the situation where storage is backed or committed by physical memory.

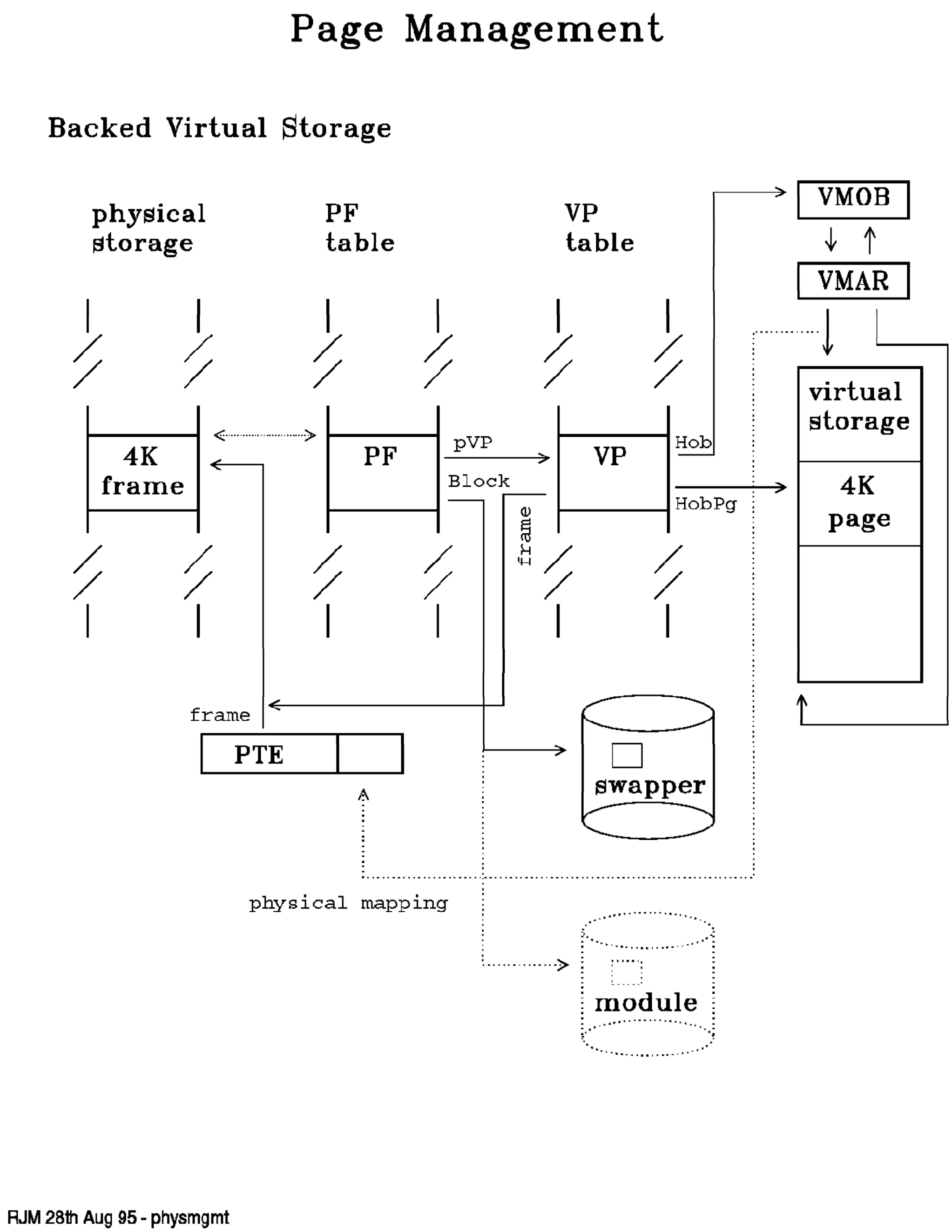


Figure 8. Page Management

The next diagram shows how this situation changes when storage is paged out.

Note that the page table entry is used to record the swapper block number when the page is not present.

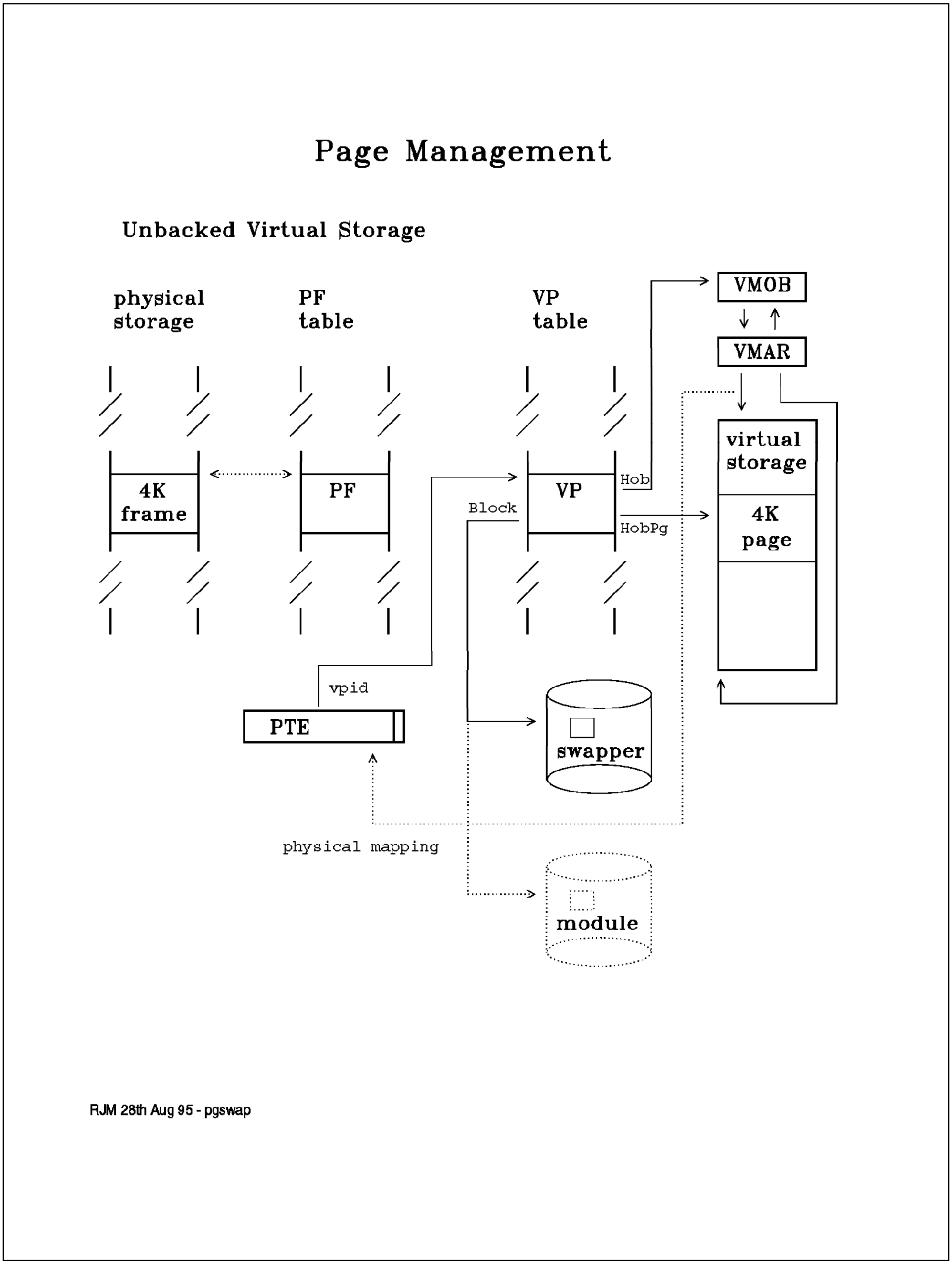


Figure 9. Page Management, Storage Paged Out

13.1.1.13 Aliasing

The situation described thus far can be further complicated by a technique known as aliasing. This is where one or more pages of an object may be mapped by page table manipulation to one or more pages of another object. In effect, this is partial object sharing.

This can occur between processes or within a process and is usually done for the following reasons:

- A device driver needs to create an I/O buffer to receive data at interrupt time and therefore in any context. The application that called the device driver also needs to have access to the results. This is commonly solved by the device driver making a UVIRT allocation in the system arena which aliases an application data buffer.
- A debugging application needs to access or even modify data and code of the debugee. This is achieved through CS and DS selector aliasing.
- A Dos Virtual Machine needs to simulate the A20 line wrap-around. Storage addressed above the A20 line aliases to addresses module 2**20.
- A Dos Virtual Machine′s Private Arena address space is aliased in the system arena so that it may be accessed by Virtual Device Drivers in a context other than that of the VDM. The VDM handle (HVMD) is the alias address, which the VDD may add to any Private Arena Address to obtain a context independent access to a location in a given VDM.

These situations require the introduction of another memory management control block: the alias record (VMAL). Each VMAL has a unique handle or hal, which is the table entry from which the VMAL is allocated.

Where aliasing occurs, the handle to the alias record (hal) is saved in the VMAR of the aliasing address range.

In the case of memory aliasing the VMAL contains the handle to the PTDA of the aliasing process.

In the case of CS/DS aliasing within a process the VMAL contains the CS selector.

The link field of the VMAR is used to link together aliasing and aliased address ranges.

Alias records may be formatted using the .ML command as shown in the following example:

```
##.ml
 hal=0001 pal=%fc5de020  har=00af  hptda=00ae  pgoff=00000  f=001
 hal=0002 pal=%fc5de028  har=00b0  hptda=00ae  pgoff=00000  f=001
 hal=0003 pal=%fc5de030  har=007a  hptda=00ae  pgoff=00000  f=001
 hal=0004 pal=%fc5de038  har=0160  cs=00e6 ds=d446 cref=001 f=13
 hal=0005 pal=%fc5de040  har=017f  hptda=00ae  pgoff=00000  f=001
 hal=0006 pal=%fc5de048  har=0197  hptda=00ae  pgoff=00000  f=021
 hal=0007 pal=%fc5de050  har=0198  hptda=00ae  pgoff=00000  f=021
 hal=0008 pal=%fc5de058  har=0199  hptda=00ae  pgoff=00000  f=021
 hal=0009 pal=%fc5de060  har=01c8  hptda=00ae  pgoff=00000  f=001
 hal=000a pal=%fc5de068  har=01db  cs=0056 ds=d446 cref=001 f=13
 hal=000b pal=%fc5de070  har=020b  cs=0056 ds=d446 cref=001 f=13
 hal=000c pal=%fc5de078  har=0242  cs=0056 ds=d446 cref=001 f=13
##
```

CS aliasing is depicted in the following diagram.

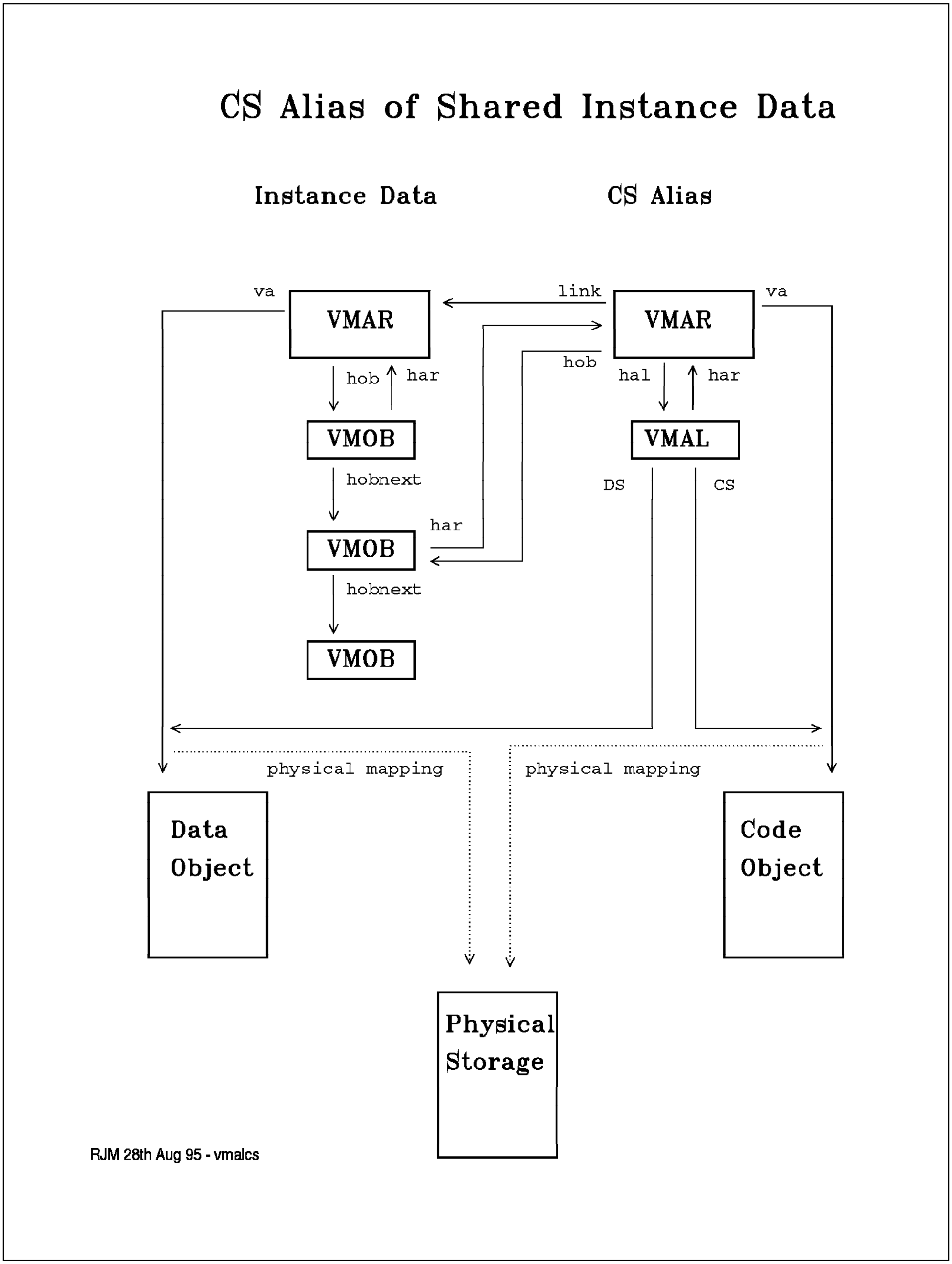


Figure 10. CS Alias of Shared Instance Data

The following diagram depicts multiple process memory aliasing.

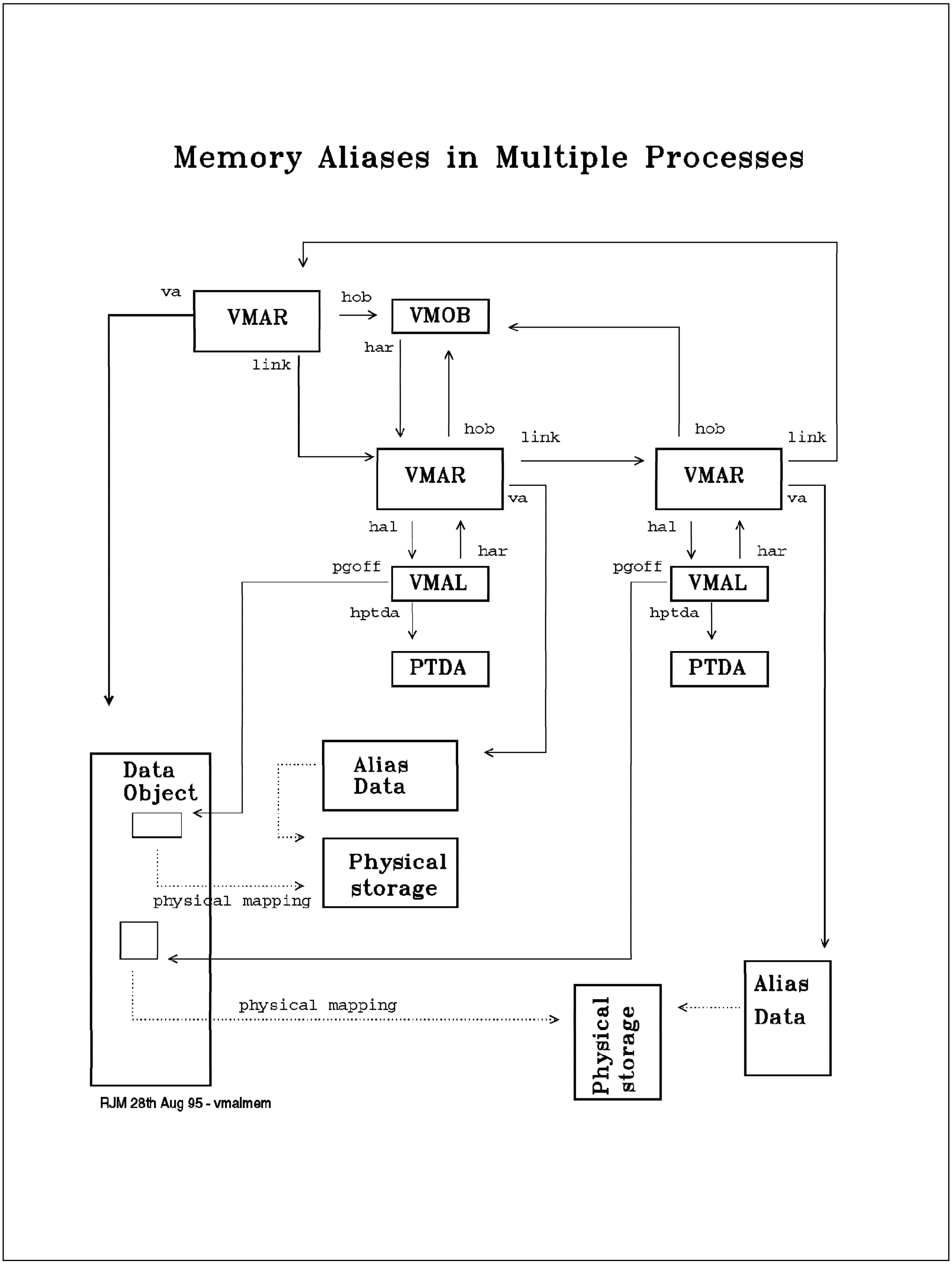


Figure 11. Multiple Alias in Multiple Processes

13.1.1.14 Who Owns Virtual Memory?

Given a virtual address, the procedure for determining who owns and is using this memory essentially amounts to the following steps:

1. If the question of ownership relates to a known process′s private storage then determine its hptda.
2. Locate the arena record(s) that encompass the address.
3. If more than one then select the one that relates to the process of interest (if known) by matching the hptda.
4. Locate the object records that are chained to the arena record.
5. If more than one then select the process of interest by matching the hptda.
6. Note the own and hmte values and their interpretation on the right-hand side of the VMOB display.
7. If necessary format the own and hmte VMOBs.
8. If either is an MTE then use `.LM` or `.LMO`, with the hob as parameter, to format the MTE.
9. If the memory is shared (hco=nnnnn appears in the arena record display) then format the chain of VMCOs and select the one that matches hptda from step 1.

Fortunately this task is reduced in complexity because of the **M** or match option that exists with both the `.MO` and `.MA` commands.

`.MOM addr` will display the VMOB of a pseudo-object that matches the *addr* if it exists. PTDAs are pseudo-objects and their addresses are listed by the `.P` command.

`.MAM addr` will search for all arena records whose address range encompasses *addr*. Under the kernel Debugger this search is restricted to the current context unless the *A* option (all contexts) is also specified. Under the dump formatter *A* in always in effect.

The *C* option further reduces the effort. This is the chain option and is applicable to `.MO`, `.MA`, `.MC` and `.ML` commands. Chaining formats all VMOBs, VMARs, VMCO and VMALs that are chained from each VMAR associated with the VM control block being formatted.

`.MAMC` (or `.MAMAC` under the DF) are the default options if just `.M` is specified. Furthermore the matching address defaults to the current CS:EIP.

The following sections illustrate memory ownership in:

Shared Arena Global Data

Shared Arena Instance Data

Private Arena Shared and Private Data

Physical Storage.

Further examples in memory management exploration, including looking at aliasing may be found in 14.1.2, “Exploring Memory Management” on page 245.

Shared Global Data: **Who owns %12123456**?

```
#.m %12123456

*har     par      cpg        va    flg next prev link hash hob   hal
 0297 %feef2904 00000660 %11fb0000 369 0312 0295 0000 009e 02a1 0000  hco=0057f
 hob    har hobnxt flgs own  hmte  sown,cnt lt st xf
 02a1  0297 0000  4a2d fff5 0302  0000 00  00 00 00 give
 hco=057f pco=ffe70b96  hconext=00241 hptda=06d1 f=1e  pid=0059
 hco=0241 pco=ffe6fb60  hconext=004ce hptda=04b2 f=1e  pid=0043 c:pmspool.exe
 hco=04ce pco=ffe70821  hconext=0034c hptda=05c3 f=1e  pid=0016 c:pawn.exe
 hco=034c pco=ffe70097  hconext=001e4 hptda=04ca f=1e  pid=000f c:pmshell.exe
 hco=04ca pco=ffe7080d  hconext=00348 hptda=05c3 f=16  pid=0016 c:pawn.exe
 hco=0348 pco=ffe70083  hconext=0017a hptda=04ca f=16  pid=000f c:pmshell.exe
 hco=017a pco=ffe6f77d  hconext=00177 hptda=03d9 f=16  pid=000b c:spdaemon.exe
 hco=0177 pco=ffe6f76e  hconext=00148 hptda=03ec f=16  pid=000c
 hco=0148 pco=ffe6f683  hconext=000b2 hptda=03c9 f=16  pid=000a c:ddaemon.exe
 hco=00b2 pco=ffe6f395  hconext=00083 hptda=0359 f=16  pid=0009 c:harderr.exe
 hco=0083 pco=ffe6f2aa  hconext=00081 hptda=02e1 f=16  pid=0007 c:landll.exe
 hco=0081 pco=ffe6f2a0  hconext=0007d hptda=02ad f=16  pid=0006 c:lanmsgex.exe
 hco=007d pco=ffe6f28c  hconext=00037 hptda=027a f=16  pid=0008 c:pmshell.exe
 hco=0037 pco=ffe6f12e  hconext=00000 hptda=02ac f=16  pid=0004 c:gambit.exe

# .mo 302
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0302  %fdf40844  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmmerge.dll
#
```

This is shared arena global data because of the presence of the hco chain. The storage was dynamically allocated by PMMERGE.DLL as giveable storage. It is currently being referenced by 14 processes.

Shared Instance Data **Who allocated %13fa1234**?

```
# .m %13fa1234
*har     par      cpg        va    flg next prev link hash hob   hal
 009c %feeefd72 00000010 %13fa0000 179 009b 009d 0000 0000 063f 0000      =0000
 hob    har hobnxt flgs own  hmte  sown,cnt lt st xf
 063f  009c 0497  002c 06d1 00a3  0000 00  00 00 00 priv 0059
 0497  009c 05c4  002c 04b2 00a3  0000 00  00 00 00 priv 0043 c:pmspool.exe
 05c4  009c 04ce  002c 05c3 00a3  0000 00  00 00 00 priv 0016 c:pawn.exe
 04ce  009c 03f0  002c 04ca 00a3  0000 00  00 00 00 priv 000f c:pmshell.exe
 03f0  009c 03dd  002c 03ec 00a3  0000 00  00 00 00 priv 000c
 03dd  009c 03cd  002c 03d9 00a3  0000 00  00 00 00 priv 000b c:spdaemon.exe
 03cd  009c 035d  002c 03c9 00a3  0000 00  00 00 00 priv 000a c:ddaemon.exe
 035d  009c 02f4  002c 0359 00a3  0000 00  00 00 00 priv 0009 c:harderr.exe
 02f4  009c 02e5  002c 027a 00a3  0000 00  00 00 00 priv 0008 c:pmshell.exe
 02e5  009c 02ae  002c 02e1 00a3  0000 00  00 00 00 priv 0007 c:landll.exe
 02ae  009c 02a8  002c 02ad 00a3  0000 00  00 00 00 priv 0006 c:lanmsgex.exe
 02a8  009c 0000  002c 02ac 00a3  0000 00  00 00 00 priv 0004 c:gambit.exe

# .mo a3
 hob       va     flgs own  hmte  sown,cnt lt st xf
 00a3  %fdef5f70  8000 ffa6 0006  0000 00  00 00 00 mte       c:doscall1.dll

# .lmo a3
hmte=00a3 pmte=%fdef5f70 mflags=0498b594 c:\os2\dll\doscall1.dll
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 00000360 1bf80000 80009025 00000001 00000001 00ac dfc6 r-x shr alias iopl
0002 0000aa34 1bf90000 80002025 00000002 0000000b 00ab dfcf r-x shr big
0003 0000d499 1bfa0000 8000d025 0000000d 0000000e 00aa dfd6 r-x shr alias conf iopl
```

```
0004 0000d4f0 1bfb0000 8000d025 0000001b 0000000e 00a9 dfde r-x shr alias conf iopl
0005 00001140 13f90000 80001023 00000029 00000002 00a8 9fcf rw- shr alias
0006 00001af0 13fa0000 80001003 0000002b 00000002 0000 9fd7 rw- alias
0007 00000e44 13fb0000 80001023 0000002d 00000001 00a6 9fdf rw- shr alias
0008 00000550 13fc0000 80001003 0000002e 00000001 0000 9fe7 rw- alias
0009 00001000 13fd0000 80001023 0000002f 00000000 00a4 9fef rw- shr alias
000a 00001000 13fe0000 80001023 0000002f 00000000 00a2 9ff7 rw- shr alias
#
```

%13fa1234 is the shared arena (there is no hptda associated with the arena record).

This is shared instance data because of the chain of related object records.

The storage was allocated by hmte=a3, but their are multiple owners.

mte a3 is DOSCALL1.DLL

%13fa1234 is within object 6 of the module. It is DLL RW instance data.

Private Data **Who owns #17:0 in thread slots 8 and 9?**

```
>> First find the hptda's for each of the slots of interest since we are
>> looking at private arena storage

# .p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008  0008 0001 0008 0007 blk 0200 abd2f000 abe497f0 abe28bf0      01 PMSHL32
# .mom %abe497f0
 hob      va     flgs own  hmte  sown,cnt lt st xf
 027a  %abe497f0  8000 ffcb ff79  0000 00  00 00 00 ptda 0008 c:pmshell.exe

# .p 9
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0009  0004 0001 0003 0001 blk 081f abd30000 abe48614 abe28de8      00 GAMBIT
# .mom %abe48614
 hob      va     flgs own  hmte  sown,cnt lt st xf
 02ac  %abe48614  8000 ffcb 02a8  0000 00  00 00 00 ptda 0004 c:gambit.exe


>> Next list all the owners of 17:0

# .m #17:0
*har     par      cpg        va     flg next prev link hash hob   hal
 026d %feef2568 00000010 %00020000 1d9 029a 026c 0000 0000 029d 0000 hptda=02ad
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 029d  026d 0000   0838 029e 029e  0000 00  00 00 00 shared    c:lanmsgex.exe

*har     par      cpg        va     flg next prev link hash hob   hal
 0277 %feef2644 00000010 %00020000 1d9 0276 0272 0000 0000 02b0 0000 hptda=02ac
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 02b0  0277 0000   0838 02b1 02b1  0000 00  00 00 00 shared    c:gambit.exe

*har     par      cpg        va     flg next prev link hash hob   hal
 02a0 %feef29ca 00000010 %00020000 179 02a4 029f 0000 0000 02e8 0000 hptda=02e1
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 02e8  02a0 0000   002c 02e1 02e7  0000 00  00 00 00 priv 0007 c:landll.exe

*har     par      cpg        va     flg next prev link hash hob   hal
```

```
 02aa %feef2aa6 00000010 %00020000 179 02ab 02a9 0000 0000 02f8 0000 hptda=027a
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 02f8  02aa 0000  002c 027a 02f7  0000 00  00 00 00 priv 0008 c:pmshell.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 02fc %feef31b2 00000010 %00020000 1d9 02fd 02fb 0000 0000 0360 0000 hptda=0359
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0360  02fc 0000  0838 035f 035f  0000 00  00 00 00 shared    c:harderr.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0360 %feef3a4a 00000010 %00020000 1d9 0361 035f 0000 0000 03d0 0000 hptda=03c9
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03d0  0360 0000  0838 03cf 03cf  0000 00  00 00 00 shared    c:ddaemon.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 036b %feef3b3c 00000010 %00020000 1d9 036c 036a 0000 0000 03e0 0000 hptda=03d9
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03e0  036b 0000  0838 03df 03df  0000 00  00 00 00 shared    c:spdaemon.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0378 %feef3c5a 00000010 %00020000 1d9 0379 0377 0000 0000 03f3 0000 hptda=03ec
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03f3  0378 0000  0838 03f2 03f2  0000 00  00 00 00 shared

*har     par      cpg        va    flg next prev link hash hob   hal
 040e %feef493e 00000010 %00020000 179 045c 040f 0000 0000 04c6 0000 hptda=04b2
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 04c6  040e 0000  002c 04b2 0522  0000 00  00 00 00 priv 0043 c:pmspool.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0427 %feef4b64 00000010 %00020000 179 0428 0426 0000 0000 04cf 0000 hptda=04ca
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 04cf  0427 0000  002c 04ca 02f7  0000 00  00 00 00 priv 000f c:pmshell.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 04e8 %feef5bfa 00000010 %00020000 179 04e6 04e5 0000 0000 05d4 0000 hptda=05c3
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 05d4  04e8 0000  002c 05c3 05cf  0000 00  00 00 00 priv 0016 c:pawn.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0502 %feef5e36 00000010 %00020000 1d9 059f 0598 0000 0000 0507 0000 hptda=06d1
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0507  0502 0000  0838 05b3 05b3  0000 00  00 00 00 shared

*har     par      cpg        va    flg next prev link hash hob   hal
 0507 %feef5ea4 00000010 %00100000 1e1 056c 05cb 05d4 0000 0678 0018 hptda=04af
 hal=0018 pal=%fddae0d8  har=0507  hptda=04af  pgoff=00000  f=081
 har     par      cpg        va    flg next prev link hash hob   hal
 05d4 %feef7042 00000040 %00000000 1e1 05bf 0461 0000 0000 0678 0000 hptda=04af
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0678  0507 0000  103c 04af 0000  0000 00  00 00 00 priv 005b *vdm

>> Slot 8:

# .mo 2b1
 hob       va     flgs own  hmte  sown,cnt lt st xf
 02b1  %feeeef38  8000 ffa6 02a7  0000 00  00 00 00 mte       c:gambit.exe
```

```
# .lmo 2b1
hmte=02b1 pmte=%feeeef38 mflags=00003140 c:\dcaf13\gambit.exe
seg  sect psiz vsiz hob  sel  flags
0001 0002 1fe0 1fe0 02b2 000f 2d20 code shr rel
0002 0013 002a 002c 02b0 0017 2d20 code shr rel
0003 0014 19ae 19ae 0000 001f 0d01 data rel
0004 0022 0002 0002 02a9 0027 2c20 code shr
0005 0000 0000 3400 0000 002f 0c01 data


#

>> Slot 9


# .mo 2f7
 hob      va     flgs own  hmte  sown,cnt lt st xf
 02f7  %fdf40a18  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmshell.exe
#
```

This is private arena data of some sort, whose address range in present in 13 processes.

The hptda for Pid 4 (slot 9 is 2ac)

The second major entry from `.M` output (har=277, hptda=2ac) is for GAMBIT.EXE in Pid 4.

The owner and hmte are the same (2b1). This indicates a code segment within the module GAMBIT.EXE.

`.LMO 2b1` show this to be in segment 2 of GAMBIT.EXE

The storage in Pid 8 (slot 8) is shown in the 4th entry, har=2aa.

Here own=27a and hmte=2f7.

The owner is shown to the right of the VMOB as being Pid 8. We can check this by displaying hob 27a. This turns out to be a ptda for Pid 8, as we saw when we used .mom against the PTDA address.

`.LMO 2f7` shows this to be the MTE for PMSHELL.EXE. We concluded that pmshell has allocated private memory in Pid 8 at this address.

Physical Memory: **Who owns physical address %%90123**?

```
# .mp 90
ffe1b6c0 InUse: pVP=ff408576 RefCnt=0001 Flg=1 ll=00 sl=00 Blk=00272 Frame=00090

# .mv %ff408576
VPI=03bf pVP=ff408576 Swp Frame=0090 Flg=030 HobPg=0011 Hob=0282 Ref=001

# .moc 282

*har     par      cpg        va    flg next prev link hash hob   hal
 0263 %feef248c 00000060 %a8650000 001 0262 0267 0000 0001 0282 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
```

```
 0282  0263 0000  0000 ffd4 0000  0000 00  00 00 00 vddheap

# dp %a8650000+11000 l1
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%a8661000* 0055d  frame=00090  0    0  c  A        s  r  P  pageable
%a8661000  00090  frame=00090  0    0  c  A        s  r  P  pageable
#
```

Physical address %%90123 is in frame 90.

This is currently assigned to VP at %ff408576 and is on the swap file at block 272.

The VP tells us the hob and page within the hob.

.MOC will format the VMOB and associated VMAR.

We can check that this is correct from the page table entries for the 17th (0x11th) page of the object's virtual address.

13.1.2 Thread Scheduling and Dispatching Topics

Part two of our discussion on the wait condition centres on the scheduler and dispatcher and the mechanisms that govern when threads will or will not run.

This is considered from the perspective the system, which leads us to divide the discussion into two cases:

- 13.1.2.1, "Blocking - Voluntary Suspension."
- 13.1.2.2, "Involuntary Suspension" on page 202.

13.1.2.1 Blocking - Voluntary Suspension

We now turn our attention to *blocking*, which is the mechanism that threads use to give up processor time voluntarily to wait for an event to occur or a resource to become available.

The term voluntary is chosen from the perspective of the scheduler and not necessarily from the application's perspective. In this context voluntary suspension refers to an action taken by a thread to give up its time-slice. This will include direct actions such as waiting on semaphores as well as calling APIs, which for internal reasons need to wait for a resource or an event.

PROCBLOCK and its counterpart PROCRUN are the two kernel routines at the heart of the block/run mechanism. These are callable directly by kernel component and also by device drivers and file system drivers through a small interface layer. Application code only gets to call PROCBLOCK and PROCRUN indirectly through system APIs and in particular through the semaphore APIs.

The block/run mechanism is designed with the following criteria:

- A thread should be able to block without the waking thread having to know whether anyone, or who, had blocked on a resource
- Multiple threads should be able to wake when an event or resource becomes available.

This is achieved by having an abstract token, known as the *BlockID*, associated with the resource or event. The BlockID is passed to PROCBLOCK when a thread blocks. Similarly when another thread wishes to wake threads waiting for a resource or event the BlockID that represents the resource or event is passed to PROCRUN.

In addition to the BlockID, callers of PROCRUN receive a flag that indicates whether all or just the highest priority thread waiting on the BlockID should wake.

This mechanism has shortcomings unless certain constraints are applied:

BlockIDs need to be subject to a convention that gives uniqueness otherwise it is possible that threads will incorrectly block and run. A solution is to use the address of a control block memory object that relates uniquely to the resource or event.

If addresses are to be used for BlockIDs then they must point to global data for reasons of uniqueness. Furthermore, if they are to be reference by disabled code then the storage needs to be in resident memory. This more or less implies that addresses must be taken from within the System Arena.

If BlockIDs are in use that do not represent addresses then they must not conflict with any potential addresses used as BlockIDs.

Even if addresses are use there is no accounting information that says who owns the related resource.

A workable scheme is implemented by limiting the direct use of PROCBLOCK and PROCRUN to system code, device drivers and file system drivers, all of which have access to the System Arena.

Apart from three special conventions the system and most device drivers use addresses as BlockIDs. There are three system defined conventional BlockIDs are:

fffe....
Results from a RAMSEM wait.

fffd....
Results from a MUXWAIT.

ffca....
Results from a child wait.

x....... (x=a - f)
Linear address of the memory object of control block that relates to the resource.

....
Probably selector:offset address of the memory object or control block that relates to the resource.

This scheme could be subverted by device drivers, but in general they will choose to block on addresses of resources they own, which are usually allocated out of the system arena and addressed using a GDT select:offset.

Accountability remains an exposure. For BlockIDs that are addresses the owner of the memory that the BlockID points to gives a big clue. For conventional BlockIDs we have to do more work. These are discussed in detail later. We will first we look at an example of a BlockID that is an address.

Basic Technique:

The technique for analyzing blocked threads is two-pronged:

1. We can look at the wait from the application perspective by examining the current user registers and by trying to identify the API issued. This is usually relatively easy but often gives no clue as to the underlying wait since any single API may block on many occasions for many reasons.

2. Examine the problem from the internal or kernel perspective to determine what an API might be waiting for. This process starts with finding the associated BlockID.

When a thread blocks its BlockID is stored in the TCB TCBSleepId field. Conveniently, this is formatted by using the .PB KDB and DF command.

Note: .PB under DF lists non-blocked threads. BlockIDs are irrelevant for such threads.

.PB also attempts to interpret the BlockID. The full details of these are given in the Kernel Debugger and Dump Formatter Command Reference (Volume 2). In addition to classifying the BlockID, .PB examines TCB_SemInfo and TCB_SemDebugAddr.

For many semaphore originated BlockIDs TCB_SemInfo is used to store the address or handle of the user's semaphore that lead to the thread blocking. The .PB command will attempt to locate a near symbol to the semaphore address and display it.

Under the kernel debugger, TCB_SemDebugAddr is used to store the address of the caller to the Semaphore API when the thread blocked. If this field is not 0xffffffff .PB attempts to locate a near symbol to the caller and display it.

Once we have the BlockID, TCB_SemInfo and TCB_SemDebugAddr we are able to begin searching for information associated with reason for blocking.

The next step is to decide whether the BlockID is one of the three special categories or to be treated as an address.

Blocking on the Address of a Resource: The initial analysis of BlockIDs that are linear addresses uses the .M command to determine ownership.

If we have appropriate symbols loaded, the LN command against the BlockID can also be very informative.

As mentioned in the previous section, for addresses to be effective BlockIDs they must be unique and so are generally allocated from the system arena. Most allocations from the system arena are labelled with a system object Id. If the .MO command is used against a system object Id it will display a meaningful mnemonic for the Owner Id. In many cases the mnemonic is for a system control block or buffer. BlockIDs that address the beginning of a control block tend to be used for serializing updates to the control block. There may be processes that a control block is associated with. These are often serialized by using the address of a field within the control, that is specifically associated with the process.

A complete list of system object Ids may be found in the under the Kernel Debugger Command Reference under the .MO command description.

We now look at some examples:

File System - Device Driver

```
# .pb41
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0041  blk 04085ca7 DEMO1
# ln 408:5ca7
No Symbols Found
# .m 408:5ca7

*har     par      cpg        va    flg next prev link hash hob   hal
 0079 %fef1fa70 00000010 %7bf27000 129 0078 0077 0000 0000 007b 0000   sel=0408
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 007b  0079 0000  0324 ffa1 0000  0000 00  00 00 00 sft

# .d sft 408:5ca7
      sf_ref_count: 0000                       sfi_mode: 0092
        sf_usercnt: 0000                       sfi_hVPB: 0000
          reserved: 00                        sfi_ctime: 0000
       sf_flags(2): 02c0:0000                 sfi_cdate: 0000
         sf_devptr: #0928:001c                sfi_atime: 0000
            sf_FSC: #00c8:ff40                sfi_adate: 0000
          sf_chain: #0000:0000                sfi_mtime: 3ce1
            sf_MFT: ffffffff                  sfi_mdate: 1eb0
sfdFAT_firFILEclus: 0000                       sfi_size: 00000000
    sfdFAT_cluspos: 0000                   sfi_position: 000013c0
    sfdFAT_lstclus: 0000                        sfi_UID: 0000
     sfdFAT_dirsec: 00000000                    sfi_Pid: 0012
     sfdFAT_dirpos: 00                          sfi_PDB: 0000
       sfdFAT_name: DEMODEV2                     sfi_selfsfn: 00b5
   sfdFAT_EAHandle: 0000                     sfi_tstamp: 00
          sf_plock: 0000                    sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000

# .p41
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0041  0012 000f 0012 0001 blk 0300 7bd19000 7bdfc218 7bddfc68 0ebc 13 DEMO1

# .s41
Current slot number: 0041

# .r
eax=00000000 ebx=00000002 ecx=00000000 edx=4d3409ea esi=d02f0021 edi=000009ea
eip=00000134 esp=0000a424 ebp=0000a43e iopl=2 -- -- -- nv up ei pl nz ac po nc
cs=000f ss=005f ds=005f es=004f fs=150b gs=0000  cr2=00000000  cr3=001ac000
000f:00000134 8946fe         mov     word ptr [bp-02],ax           ss:a43c=0d16

# u cs:ip-20
000f:00000114 681f00         push    001f
000f:00000117 682d00         push    002d
000f:0000011a 0e             push    cs
000f:0000011b e87c19         call    1a9a
000f:0000011e 83c40a         add     sp,+0a
000f:00000121 8e06ee09       mov     es,word ptr [09ee]
000f:00000125 8b5ef8         mov     bx,word ptr [bp-08]
000f:00000128 d1e3           shl     bx,1
000f:0000012a 26ffb764d2     push    word ptr es:[bx+d264]
000f:0000012f 9a0000ab1d     call    1dab:0000
```

```
000f:00000134 8946fe         mov     word ptr [bp-02],ax
000f:00000137 8e06ee09       mov     es,word ptr [09ee]

# dg 1dab
1dab  CallG32 Sel:Off=0148:00004414     DPL=3 P  DWC=1

# ln 148:4414
0148:00004414 OS2KRNL DOSCLOSE
```

Slot 41 is waiting on BlockID 04085ca7. This is too low to be a linear address. We assume selector:offset.

`.M 408:5ca7` reveals the owner to be *sft*. This is a System File Table structure.

The `.D` command will format STFs, so we do so using the BlockID as the SFT address.

This SFT represents a device driver called DEMODEV2. We can tell because there is no MFT pointer in the SFT and the flags indicate a device.

From the application side we unassemble back from the CS:IP.

The application has just issued a call-gate instruction.

Examination of the call-gate GDT descriptor show we were calling DOSCLOSE in the kernel.

We are waiting for the close to complete, possibly the device driver has not returned completion status to the last I/O request.

Named Pipes

```
#.s 18
Current slot number: 0018
# .pb#
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0018# blk 06700012 EPWPSI
#

#.m 670:12

*har     par      cpg        va    flg next prev link hash hob   hal
 00a8 %fef1fe7a 00000010 %7b563000 129 00a7 00a9 0000 0000 00b4 0000   sel=0670
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00b4  00a8 0000  0124 ff31 0000  0000 00  00 00 00 npipenp

#.p#
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0018# 000c 0000 000c 0001 blk 0200 7bc70000 7bd79964 7bd5b5f0 0ec8 00 EPWPSI

# .r
eax=00000000 ebx=00005552 ecx=00050000 edx=0000f020 esi=00000446 edi=00000302
eip=000014bb esp=000063da ebp=000063de iopl=2 -- -- -- nv up ei pl nz na po nc
cs=d01f ss=002f ds=beb7 es=0077 fs=150b gs=0000  cr2=00000000  cr3=001ac000
d01f:000014bb c9             leave
# u.cs:ip-10
Expression error
```

```
# u cs:ip-10
d01f:000014ab c9            leave
d01f:000014ac ca0a00        retf    000a
DOSCALL1 DOSCONNECTNMPIPE:
d01f:000014af c8000000      enter   0000,00
d01f:000014b3 ff7606        push    word ptr [bp+06]
d01f:000014b6 9a0000131c    call    1c13:0000
d01f:000014bb c9            leave
d01f:000014bc ca0200        retf    0002
DOSCALL1 DOSDISCONNECTNMPIPE:
d01f:000014bf c8000000      enter   0000,00
d01f:000014c3 ff7606        push    word ptr [bp+06]
d01f:000014c6 9a00001b1c    call    1c1b:0000
d01f:000014cb c9            leave
d01f:000014cc ca0200        retf    0002

# dg 1c13
1c13  CallG32 Sel:Off=0148:0000540c     DPL=3 P  DWC=1
# ln 148:540c
0148:0000540c OS2KRNL DOSCONNECTNMPIPE
#
```

In this example the BlockID is **06700012**. This is unlikely to be a linear address. We assume that it is *670:12*.

`.M 670:12` shows the owner to be *npipenm*. This is a named pipe name segment. Could the process be waiting for a pipe connection?

Looking at the application side we see that the last ring 3 instruction to be executed was a call-gate, which turns out to be DOSCONNECTNMPIPE in the Kernel.

These last two examples were reasonably revealing. More often than not we use .M against a BlockID (and other system data) and we get one of:

vmkshrw

vmkshro

vmkrhrw

vmkrhro

These are, so called public kernel heaps. Fortunately each allocated heap block is imbedded in a structure that reveals the owner of the block. This is discussed next.

Kernel Public Heaps: The kernel has seven heaps for general use by itself, device drivers and file system drivers. They have the following object id mnemonics:

vmkshro Swappable read-only heap

vmkshrw Swappable read/write heap

vmkrhro Resident read-only heap

vmkrhrw Resident read/write heap

krhro1m Resident read-only < 1Mb heap

krhrw1m Resident read/write < 1Mb heap

kbdsym Resident kernel debugger symbol heap

Not all heaps are always built. Note in particular:

The *hdbsym* heap is not present under the RETAIL kernel.

The *vmkshrw* is used for the *krhro1m*, *krhrw1m*, *vmkshrw* and *vmkshro* heaps under the RETAIL kernel.

The *vmkrhrw* heap is used for both *vmkrhrw* and *vmkrhro* under the RETAIL kernel.

Notice that each of the heaps is either resident or swappable.

Each heap is partitioned into blocks.

Swappable heap blocks have an 8 byte prefix followed by the block data.

Resident heap blocks have two forms:

Regular: for smaller allocation. These have a 4 byte prefix.

Attributed or extended: These use a 4 byte prefix and an 8 byte suffix.

Swappable Heap Blocks: Kernel swappable heap blocks for allocated blocks have the following layout:

```
<size><owner><selector><data>

Field          Bits           Description

size in bytes  63-32          Size of the block including the
                              header in bytes ORed with signature
                              0x52000000.

owner          31-16          Owner of heap block.  This is
                              either a system owner (value between
                              0xff2d and 0xfff8, or a
                              memory handle/pseudo handle such as
                              an MTE pseudo-handle.

selector       15-0           GDT selector mapping block's data
                              else null.
```

Finding the owner of a Swappable Head Selector

```
# .m 8f0:0

*har     par      cpg        va    flg next prev link hash hob   hal
 0021 %fef1f2e0 00001400 %fca5f000 121 0020 0022 0000 0020 0022 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0022  0021 0000  0225 ffef 0000  0000 00  04 00 00 vmkshrw

# dl 8f0
GDT
08f0  Code    Bas=fca95000 Lim=00008ed3 DPL=0 P  RE    A


# dd %fca95000-10
%fca94ff0  00000000 00000000 52008ee0 08f0ff49
```

```
%fca95000  08e8b81e 32b8d88e 16ca1f00 06c89000
%fca95010  1e560000 8e08e8b8 a23e83d8 06740009
%fca95020  eb63a5e8 c02b9003 0bfe4689 e90374c0
%fca95030  468b017e 10568b0e 52000805 6aff6a50
%fca95040  13969aff 5f3d1000 c4e77400 83260e5e
%fca95050  74000e7f 0142e903 0c47ff26 261276c4
%fca95060  2616448b 8918548b 5689fa46 0e468bfc

# .mo ff49
ff49 fsd2
# .lml
hmte=0982 pmte=%fe0e1a14 mflags=0408b186 e:\ibmlan\netlib\spl1a.dll
hmte=097e pmte=%fe0e1a54 mflags=0408b186 e:\ibmlan\netlib\lrhm1.dll
hmte=0979 pmte=%fe0e1bac mflags=0408b186 e:\ibmlan\netlib\lrns1.dll
hmte=096b pmte=%fe0e1d60 mflags=0408b186 e:\ibmlan\netlib\netibm.dll
hmte=0164 pmte=%fe02cc40 mflags=0498b1c8 e:\os2\dll\sysmono.fon
.
.
.
.
hmte=0181 pmte=%fe02ccb0 mflags=4498b1d5 e:\os2\dll\pmatm.dll
hmte=031b pmte=%fe02af18 mflags=0428a1c9 e:\ibmlan\netprog\netwksta.200
hmte=0306 pmte=%fe059f90 mflags=0428a1c9 e:\netware\nwifs.ifs
hmte=0160 pmte=%fe01ff4c mflags=0428a1c9 d:\dataex2\iwsfsd2.ifs
hmte=0117 pmte=%fdf5df60 mflags=0428a1c9 e:\os2\cdfs.ifs
hmte=00d2 pmte=%fdf53990 mflags=0428a1c9 e:\os2\hpfs.ifs

# .lmo 117
hmte=0117 pmte=%fdf5df60 mflags=0428a1c9 e:\os2\cdfs.ifs
seg  sect psiz vsiz hob  sel  flags
0001 0002 8ed3 8ed4 0000 08f0 8d60 code shr prel rel
0002 004a 0964 0ad0 0000 08e8 8c41 data prel
#
```

We use .M command to find that the owner of 8f0:0 is *vmkshrw*.

So, we look at the descriptor for 8f0 to find it′s base address. Note that the selectors assigned to kernel heap blocks address the data portion only.

We dump out 0x10 bytes before the selector base to show the block header to be 0x52008ee0 0x08f0ff49. This tells us the length of the block including header is 8ee0. (Data sizes are rounded up to the next quad-word). The user of the block is ff49.

Note:

The following short cut could have been used:

```
dd %(8f0:0)-10
```

.MO ff49 shows *fsd2*. This is the second file system driver to initialise.

.LML will list DLLs, Fonts and FSDs, newest first. Counting back from the end we see FSD1 is HPFS and FSD2 is CDFS.

.LMO 117 confirms that 8f0:0 does blong to CDFS.IFS.

Resident Heap Blocks: Kernel resident heap blocks are of two types, regular and attributed.

Regular blocks are the simplest and most common type. They have the form:

<simple header><data>

<simple header> is a dword (32-bits) having the following layout

<owner><prev block free flag><size in dwords><yielded flag><type flag>

Field	Bits	Description
owner	31-16	Owner of heap block. This is either a system owner (value between 0xff2d and 0xfff8, or a memory handle/pseudo handle such as an MTE pseudo-handle.
previous block free flag	15	1 if previous block is free, else 0
size in dwords	14-2	Size of the block including the header in dwords.
yielded flag	1	1 if a free block search yielded the CPU while looking at this block, else 0
type flag	0	0 (indicates Regular Block)

Extended blocks contain a two-part header and have the following form:

<size header><data><header extension>

<size header> is a dword (32-bits) having the following layout

<extra flags><size in dwords><yielded flag><type flag>.

Field	Bits	Description
extra flags	31-24	Additional flags. Bit 31 - set if block is free Bit 30 - set if prev block is free Bits 29-24 - reserved and 0
size in dwords	23-2	Size of the block including the header in dwords.
yielded flag	1	1 if a free block search yielded the CPU while looking at this block, else 0
type flag	0	1 (indicates Extended Block)

<data> is the data area available for use by the client and is always dword granular and dword aligned.

<header extension> is a dword-granular structure containing the following information

<owner><selector><hmte><pad>

```
Field          Bits           Description

owner          63-48          Owner of heap block.  This is
                              either a system owner (value between
                              0xff2d and 0xfff8, or a
                              memory handle/pseudo handle such as
                              an MTE pseudo-handle.
selector       47-32          GDT selector mapping block's data
                              else null.

hmte           31-16          hmte associated with this heap block?

pad            15-0           padding for double word alignment
```

When a block is free, its data portion contains additional information. The first two dwords contain forward and backward pointers to the next and previous blocks on the free list. The last dword contains a copy of the previous block pointer. Note that extended free blocks do not have an owner field, so bit 31 of their header is set indicating that they are free.

The **hmte** field of the header extension is no longer used for any specific purpose.

Now for an example of a regular heap block.

```
# .s 47
Current slot number: 0047

# .pb #
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0047# blk fe04c8e8 PMSHL32

# .m %fe04c8e8

*har     par      cpg        va    flg next prev link hash hob   hal
 0003 %fef1f04c 00001000 %fdf1f000 001 0002 0020 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 ff08  0000 ffec 0000  0000 00  06 00 00 vmkrhrw
# dd %fe04c8e8-10
%fe04c8d8  00031c3f 00000000 00000000 ffc20010
%fe04c8e8  00000010 00000000 fe040001 ffc20010
%fe04c8f8  00000010 00000000 fe040001 ffe900a8
%fe04c908  fe0c767c fe0c0ee0 00000000 00000000
%fe04c918  00000000 00000000 00000000 00000000
%fe04c928  00000000 00000000 00000000 00000000
%fe04c938  00000000 00000000 00000000 00000000
%fe04c948  00000000 00000000 00000000 00000000

# .mo ffc2
ffc2 semstruc

# .d sem32 %fe04c8e8

       Type: Private Event
       Flags: Reset
       pMuxQ: 00000000
  Post Count: 0000
  Open Count: 0001
 Create Addr: 0010fe04
```

```
# .p#
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0047# 000d 000a 000d 0004 blk 0200 7bd1f000 7bdfa188 7bde06b8      11 PMSHL32
```

In this example we are interested in slot 47. Its BlockID is owned by *vmkrhrw*.

We dump the heap block from 0x10 bytes before the start.

Note that the low order bit of the header is 0, therefore a regular block.

Since the two low order bit are flags and the size is in double words we conveniently ignore these to obtain the size in bytes, which happens to be 0x10.

The block is owned by ffc2, which the .MO command tells us is *semstruc*.

This is very good news because all the semstruc owner relates to the 32-bit semaphore APIs. The .D command formats these for us.

Finally note that if we attempt to look at this from the application perspective we see from .P that the TSD is swapped out (Disp is blank). This means that the user registers for slot 47 can't be loaded. Furthermore attempts to look at the registers are unpredictable as DF and KDB will have not changed the values since they last loaded registers.

This is a case where BlockID analysis will give us a clue even if the application data is unavailable.

Lastly we look at an extended heap block.

```
# .s 4b
Current slot number: 004b

# .pb #
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 004b# blk 21a0ade0 WKSTAHLP

# .m 21a0:ade0

*har     par      cpg        va     flg next prev link hash hob   hal
 0003 %fef1f04c 00001000 %fdf1f000 001 0002 0020 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 ff08  0000 ffec 0000  0000 00  06 00 00 vmkrhrw

# dg 21a0
21a0  Code    Bas=fe070000 Lim=0000bd5b DPL=0 P  RE    A

# dd %fe070000-10
%fe06fff0  00000000 00000000 fe06fd72 4000bd6d
%fe070000  10d0006b 9090cbcb 9090cb90 000af390
%fe070010  3c600104 6aec8b55 8d026a01 16ebd866
%fe070020  6aec8b55 8d026a02 0aebd866 6aec8b55
%fe070030  8d026a00 46c6d866 561e00ee 21906857
%fe070040  1e7ec41f f714568b 750001c2 568b520e
%fe070050  27e2830e 29558826 2605eb5a 002945c6
%fe070060  4000c2f7 c0330574 f70d39e9 748000c2
```

```
# dd %fe070000-4+bd6c-10
%fe07bd58  fee2e9de 00000000 ff4c21a0 fdf1ff32
%fe07bd68  ffe9008c fe06fd70 fe02aa70 00000000
%fe07bd78  00000000 00000000 00000000 00000000
%fe07bd88  00000000 00000000 00000000 00000000
%fe07bd98  00000000 00000000 00000000 00000000
%fe07bda8  fe07bd6a ffe90048 00000201 00000000
%fe07bdb8  00000000 00000000 00000000 02010000
%fe07bdc8  00000000 00000000 00000000 000000ed
#

# .mo ff4c
ff4c fsd5

# .lml
hmte=0982 pmte=%fe0e1a14 mflags=0408b186 e:\ibmlan\netlib\spl1a.dll
hmte=097e pmte=%fe0e1a54 mflags=0408b186 e:\ibmlan\netlib\lrhm1.dll
hmte=0979 pmte=%fe0e1bac mflags=0408b186 e:\ibmlan\netlib\lrns1.dll
hmte=096b pmte=%fe0e1d60 mflags=0408b186 e:\ibmlan\netlib\netibm.dll
hmte=0164 pmte=%fe02cc40 mflags=0498b1c8 e:\os2\dll\sysmono.fon
.
.
.
.
hmte=0181 pmte=%fe02ccb0 mflags=4498b1d5 e:\os2\dll\pmatm.dll
hmte=031b pmte=%fe02af18 mflags=0428a1c9 e:\ibmlan\netprog\netwksta.200
hmte=0306 pmte=%fe059f90 mflags=0428a1c9 e:\netware\nwifs.ifs
hmte=0160 pmte=%fe01ff4c mflags=0428a1c9 d:\dataex2\iwsfsd2.ifs
hmte=0117 pmte=%fdf5df60 mflags=0428a1c9 e:\os2\cdfs.ifs
hmte=00d2 pmte=%fdf53990 mflags=0428a1c9 e:\os2\hpfs.ifs


# .lmo 31b
hmte=031b pmte=%fe02af18 mflags=0428a1c9 e:\ibmlan\netprog\netwksta.200
seg  sect psiz vsiz hob  sel  flags
0001 0003 2a8a ffdc 0000 2190 8d41 data prel rel
0002 0019 1d93 1d94 0000 2198 8d60 code shr prel rel
0003 0028 bd5c bd5c 0000 21a0 8d60 code shr prel rel
0004 0088 fd62 fd62 0000 21a8 8d60 code shr prel rel
0005 0108 606a 606a 0000 21b0 8d60 code shr prel rel
0006 0139 3492 3492 0000 21b8 8d60 code shr prel rel
0007 0154 002e 002f 0000 21c0 8c41 data prel
0008 0155 04bb 04bb 0000 21c8 8d60 code shr prel rel
#
```

What does BlockID 0x21a0ade0 represent?

We assume selector:offset and discover the owner is *vmkshrw*.

We dump the descriptor for selector 21a0 to find its base address.

Next we dump 0x10 bytes before the descriptor base to see the heap block header.

In this example the low order bit of the header is 1 so we have to look at the header extension for the owner information.

Adding the length to the base and backing off 0x10 bytes again we uncover the block header extension.

Note: The following short cut could have been used:

```
dd %(21a0:0)-4+bd6c-10
```

In this case the owner is ff4c or fsd5. This is the 5th FSD to initialise.

We list the FSDs by using .LML and pick the 5th from the bottom. This turns out to be NETWKSTA.200.

Blocking on a ChildWait: When a process calls *DosWaitChild* and blocks waiting for a child process to terminate, the BlockID is of the form:

```
ffcapppp        where pppp is the process id of the waiting
                thread.
```

The BlockID doesn't help us pin-point the processes being waited for.

All the child process have to be examined. The process status byte at offset 0xa into the local information segment has either of the following bits set if the parent cares about termination of the child:

0x10 The parent cares

0x20 The parent did an exec-and-wait

The local information segment is embedded in the PTDA at the following offsets:

0x7ee Retail 2.11

0x7f6 Debug 2.11

0x5be Retail 3.0

05c6 Debug 3.0

Blocking on a RAMSEM: Potentially this is the most problematical type of wait to deal with. The BlockID is conventional and of the form:

```
fffexxxx        where xxxx is taken from the low order word
                of the user's RAMSEM.
```

There is no accountability associated with this type of semaphore. It is the responsibility of the user to manage their own accounting information. Accordingly most applications tend to imbed RAMSEMs into larger structures, which contain information such as use counts, owner identification and timeouts.

Two structures in particular are in common use:

The Fast Safe RAMSEM.

The PM Fast Safe RAMSEM.

The first step with RAMSEM BlockIDs is to locate the user's RAMSEM address.

Next check ownership just in case this gives a clue to the associated process.

Ownership is indicated by a non-zero value in byte 0 of the RAMSEM. Very occasionally a RAMSEM is owned by the system. When this happens happens the ownership flags takes the value of the owning process.

We hope that the RAMSEM is embedded in either a Fast Safe RAMSEM or PM Fast Safe RAMSEM.

Both of these structures have a length prefix. The PM version is 0x12 and the non-PM version 0x0e.

Display storage before the RAMSEM and examine offset -0x12. Is this word 0x0012? If not then this is not a PM FSRS. Try -0xe. Does that contain 0x000e? If not then we will have to resort to more speculative analysis.

If either of these lengths correspond, look at the next two words, these contain the owning Pid and Tid. See whether this process and thread exists and what it is doing.

Note: Tid is sometimes 0 when there is only one thread in a process.

If this technique fails us then check the owner of the semaphore address, which is saved in **TCB_SemInfo** and displayed by the `.PB` command. The owner of the semaphore, if it has not died, has to one of its accessors. If the RAMSEM is located in a Private Arena, then the owner is limited to one of the threads of the process that has blocked. If it is in shared storage, then the owned will be a thread in one of the processes on the VMCO chain. If we are lucky, the number of possibilities will be small, though this is not guaranteed.

The following example illustrates this technique:

```
>>> Slot 31 is blocked. Why?

.pb 31
 Slot  Sta BlockID  Name     Type        Addr    Symbol
 0031  blk fffe01ba aires    RamSem    e69f:000a

>>> Bad news! a RamSem. First check to see if its imbedded in a
>>> FastSafeRamSem. We look at the RamSem address, back a few bytes

##.s 31

##dw e69f:000a-10
Past end of segment: e69f:fffffffa

>>> It can't be a PM FSRamSem

##dw e69f:000a-a
e69f:00000000  000e 0019 0000 0000 0000 01ff 01ba 0000
e69f:00000010  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000020  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000030  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000040  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000050  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000060  0000 0000 0000 0000 0000 0000 0000 0000
e69f:00000070  0000 0000 0000 0000 0000 0000 0000 0000
```

```
>>> But it does look like a normal Fast Safe RamSem
>>> Pid 19, Tid=0 (this is OK if just one thread in process 19).

##.p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe3a000 ffe3c7d4 ffe3c620 1e7c 00 *ager
 0002  0001 0000 0000 0002 blk 0200 7b92a000 ffe3c7d4 7bb28020 1f3c 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 7b92c000 ffe3c7d4 7bb281d4 1f50 00 *ctxh
 0004  0001 0000 0000 0004 blk 081f 7b92e000 ffe3c7d4 7bb28388 1f48 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 7b930000 ffe3c7d4 7bb2853c 1f20 00 *lazyw
 0006  0001 0000 0000 0006 blk 0800 7b932000 ffe3c7d4 7bb286f0 1f3c 00 *asyncr
*0008# 0006 0001 0006 0001 blk 0500 7b936000 7bb460d0 7bb28a58 1eb8 01 pmshell
 000d  0006 0001 0006 0002 blk 0800 7b940000 7bb460d0 7bb292dc 1ed4 01 pmshell
 000e  0006 0001 0006 0003 blk 0800 7b942000 7bb460d0 7bb29490      01 pmshell
 000f  0006 0001 0006 0004 blk 0800 7b944000 7bb460d0 7bb29644      01 pmshell
 0010  0006 0001 0006 0005 blk 0800 7b946000 7bb460d0 7bb297f8      01 pmshell
 0007  0006 0001 0006 0006 blk 0200 7b934000 7bb460d0 7bb288a4 1ecc 01 pmshell
 0013  0006 0001 0006 0007 blk 0200 7b94c000 7bb460d0 7bb29d14 1ecc 01 pmshell
 0015  0006 0001 0006 0008 blk 0200 7b950000 7bb460d0 7bb2a07c      01 pmshell
 0016  0006 0001 0006 0009 blk 0200 7b952000 7bb460d0 7bb2a230      01 pmshell
 0017  0006 0001 0006 000a blk 0800 7b954000 7bb460d0 7bb2a3e4      01 pmshell
 0018  0006 0001 0006 000b blk 0800 7b956000 7bb460d0 7bb2a598      01 pmshell
 0019  0006 0001 0006 000c blk 0800 7b958000 7bb460d0 7bb2a74c 1eb8 01 pmshell
 001a  0006 0001 0006 000d blk 0804 7b95a000 7bb460d0 7bb2a900 1ea8 01 pmshell
 001b  0006 0001 0006 000e blk 0804 7b95c000 7bb460d0 7bb2aab4 1eb0 01 pmshell
 001c  0006 0001 0006 000f blk 0500 7b95e000 7bb460d0 7bb2ac68 1ea8 01 pmshell
 001d  0006 0001 0006 0010 blk 0800 7b960000 7bb460d0 7bb2ae1c 1bb0 01 pmshell
 001e  0006 0001 0006 0011 blk 0800 7b962000 7bb460d0 7bb2afd0 1b8c 01 pmshell
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001f  0006 0001 0006 0012 blk 0200 7b964000 7bb460d0 7bb2b184 1eb8 01 pmshell
 0009  0007 0006 0007 0001 blk 0800 7b938000 7bb44020 7bb28c0c      00 harderr
 0011  0007 0006 0007 0002 blk 0800 7b948000 7bb44020 7bb299ac      00 harderr
 0012  0007 0006 0007 0003 blk 0800 7b94a000 7bb44020 7bb29b60      00 harderr
 000a  0003 0000 0003 0001 blk 0200 7b93a000 7bb4484c 7bb28dc0      00 lanmsgex
 000b  0004 0000 0004 0001 blk 080b 7b93c000 7bb45078 7bb28f74 1cf0 00 landll
 000c  0005 0000 0005 0001 blk 0200 7b93e000 7bb458a4 7bb29128      00 lsdaemon
 0014  0008 0000 0008 0001 blk 0200 7b94e000 7bb468fc 7bb29ec8      01 stoplan
 0020  0009 0006 0009 0001 blk 0200 7b966000 7bb47128 7bb2b338      10 cmd
 0021  000a 0006 000a 0001 blk 0500 7b968000 7bb47954 7bb2b4ec 1eb8 11 pmshell
 0023  000a 0006 000a 0002 blk 0200 7b96c000 7bb47954 7bb2b854 1ecc 11 pmshell
 0024  000a 0006 000a 0003 blk 0200 7b96e000 7bb47954 7bb2ba08 1eb8 11 pmshell
 0025  000a 0006 000a 0004 blk 0200 7b970000 7bb47954 7bb2bbbc      11 pmshell
 0026  000a 0006 000a 0005 blk 0200 7b972000 7bb47954 7bb2bd70 1ecc 11 pmshell
 0027  000a 0006 000a 0006 blk 0200 7b974000 7bb47954 7bb2bf24      11 pmshell
 0028  000a 0006 000a 0007 blk 0200 7b976000 7bb47954 7bb2c0d8      11 pmshell
 0029  000a 0006 000a 0008 blk 0200 7b978000 7bb47954 7bb2c28c      11 pmshell
 002a  000a 0006 000a 0009 blk 0200 7b97a000 7bb47954 7bb2c440      11 pmshell
 002c  000a 0006 000a 000b blk 0200 7b97e000 7bb47954 7bb2c7a8 1eac 11 pmshell
 002d  000a 0006 000a 000c blk 0200 7b980000 7bb47954 7bb2c95c 1eb8 11 pmshell
 002b  000d 0006 000d 0001 blk 0200 7b97c000 7bb48180 7bb2c5f4 1eb8 12 mrfilepm
 0022  000d 0006 000d 0002 blk 0200 7b96a000 7bb48180 7bb2b6a0 1ecc 12 mrfilepm
 002e  000f 000e 000f 0001 blk 0200 7b982000 7bb491d8 7bb2cb10      13 fvp
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 002f  000e 0006 000e 0001 blk 0200 7b984000 7bb489ac 7bb2ccc4      13 cmd
 0030  0010 0006 0010 0001 blk 0200 7b986000 7bb49a04 7bb2ce78 1ed4 14 cmd
 0031  0018 0010 0018 0001 blk 0200 7b988000 7bb4a230 7bb2d02c 1f00 14 aries
 0032  0017 0006 0017 0001 blk 0400 7b98a000 7bb4aa5c 7bb2d1e0 1ed4 15 cmd
 0033  0019 0017 0019 0001 blk 0300 7b98c000 7bb4b288 7bb2d394 1f00 15 orian
```

```
>>> Pid 19 is single threaded and is blocked. See what its Block-Id is.


##.pb 33
 Slot  Sta BlockID  Name     Type       Addr    Symbol
 0033  blk fffe01bb orian    RamSem    e66f:0000

>>> Once again a RamSem. This time there's no point in looking back
>>> a few bytes to see if it's imbedded in a FastSafeRamSem because
>>> the RamSem is allocated at the beginning of segment e66f.

>>> Our only hope is to see who else has access to this semaphore.

##.m e66f:0000

*har     par      cpg        va     flg next prev link hash hob   hal
 0420 %fed03aca 00000010 %1ccd0000 369 03f1 0075 0000 0000 051b 0000  hco=007ff
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 051b  0420 0000  4a2c ff82 04f1  0000 00  00 00 00 mshare
 hco=007ff pco=fe85f816  hconext=007b6 hptda=04d1 f=16  pid=0019 a:orian.exe
 hco=007b6 pco=fe85f6a9  hconext=00000 hptda=04fd f=17  pid=0018 a:aries.exe

>>> The RamSem is allocated in Named Shared Storage (mshare is the
>>> owner). The only two processes able to access this are Pids 19 and
>>> 18. Pid 19 is this thread, which we know doesn't own this RamSem
>>> since it's waiting for it. This leaves 18.

>>> We can't be certain from the evidence presented so far but we can
>>> say:
>>> Either the RamSem is owned by 18, or it was owned by another
>>> thread that has since terminated. If it is owned by 18 then we
>>> have a deadlock between 18 and 19:
>>> orian.exe owns the FSRamSem and is waiting for the RamSem.
>>> aries.exe owns the RamSem and is waiting for the FSRamSem.
```

Fortunately simple RAMSEMs are becoming something of the past. And now that PM is 32-bit we will not see many Fast Safe RAMSEM either. We will look in detail later on at the semaphore structure that has replaced the FSRSEM in PM: the PMSEM and GRESEM.

The MUX Wait: The last category of BlockIDs to consider is the MUXWAIT. This has a BlockID of the form:

```
fffdssss       where ssss is the slot id of the waiting thread.
```

A MUXWAIT is a multiplex semaphore wait. The semaphore comprising the MUX list may be:

RAMSEMs

SYSSEMs

32-bit Event and Mutex SEMs

We will consider each of these in turn.

The first step is to format the *muxtable*. This comprises 9-byte entries. +0x2 is the slot number of the waiter. +0x5 indicates the type of semaphore. +5 is the semaphore handle, which is interpreted according to type as follows:

0x00 Entry unused

0x01 handle is offset of SYSSEM from selector 400

0x02 Entry is a hob:offset of RAMSEM

0x03 Entry is a physical address of a RAMSEM

0x04 Entry points to a 32-bit Event SEM.

The following shows an example formatted *muxtable.* There are up to 255 entries, but in practice the entries in use are grouped at the beginning of the table.

```
# db muxtable+(9*0) l9
0400:000048be c7 48 14 00 02 1a 07 be-00                GH.....>.
# db muxtable+(9*1) l9
0400:000048c7 d0 48 15 00 02 5c 07 be-00                PH...\.>.
# db muxtable+(9*2) l9
0400:000048d0 ff ff 15 00 02 78 07 be-00                .....x.>.
# db muxtable+(9*3) l9
0400:000048d9 e2 48 1f 00 02 58 07 fa-03                bH...X.z.
# db muxtable+(9*4) l9
0400:000048e2 fd 48 1f 00 02 5c 07 fa-03                }H...\.z.
# db muxtable+(9*5) l9
0400:000048eb c3 49 1f 00 02 50 07 fa-03                CI...P.z.
# db muxtable+(9*6) l9
0400:000048f4 57 49 58 00 02 61 01 64-07                WIX..a.d.
# db muxtable+(9*7) l9
0400:000048fd 06 49 1f 00 02 60 07 fa-03                .I...`.z.
# db muxtable+(9*8) l9
0400:00004906 0f 49 1f 00 02 64 07 fa-03                .I...d.z.
# db muxtable+(9*9) l9
0400:0000490f 18 49 1f 00 02 68 07 fa-03                .I...h.z.
# db muxtable+(9*a) l9
0400:00004918 a8 49 1f 00 02 6c 07 fa-03                (I...l.z.
# db muxtable+(9*b) l9
0400:00004921 2a 49 58 00 01 f0 53 00-00                *IX..pS..
# db muxtable+(9*c) l9
0400:0000492a cc 49 30 00 03 02 a7 f1-00                LI0...'q.
# db muxtable+(9*d) l9
0400:00004933 3c 49 1b 00 01 9c 53 00-00                <I....S..
# db muxtable+(9*e) l9
0400:0000493c be 48 1b 00 03 da a6 f1-00                >H...Z&q.
# db muxtable+(9*f) l9
0400:00004945 4e 49 63 00 01 fc 53 00-00                NIc..|S..
# db muxtable+(9*10) l9
0400:0000494e eb 48 63 00 01 32 54 00-00                kHc..2T..
# db muxtable+(9*11) l9
0400:00004957 60 49 58 00 01 e4 53 00-00                `IX..dS..
# db muxtable+(9*12) l9
0400:00004960 ba 49 58 00 01 ea 53 00-00                :IX..jS..
# db muxtable+(9*13) l9
0400:00004969 72 49 57 00 01 ba 53 00-00                rIW..:S..
# db muxtable+(9*14) l9
0400:00004972 7b 49 57 00 01 c0 53 00-00                {IW..@S..
```

```
# db muxtable+(9*15) l9
0400:0000497b 84 49 57 00 01 c6 53 00-00                      .IW..FS..
# db muxtable+(9*16) l9
0400:00004984 8d 49 57 00 01 cc 53 00-00                      .IW..LS..
# db muxtable+(9*17) l9
0400:0000498d 96 49 57 00 01 d2 53 00-00                      .IW..RS..
# db muxtable+(9*18) l9
0400:00004996 9f 49 57 00 01 d8 53 00-00                      .IW..XS..
# db muxtable+(9*19) l9
0400:0000499f f4 48 57 00 01 de 53 00-00                      tHW..^S..
# db muxtable+(9*1a) l9
0400:000049a8 b1 49 21 00 02 a8 07 fa-03                      1I!..(.z.
# db muxtable+(9*1b) l9
0400:000049b1 33 49 21 00 03 ee a6 f1-00                      3I!..n&q.
# db muxtable+(9*1c) l9
0400:000049ba 45 49 58 00 01 f0 53 00-00                      EIX..pS..
# db muxtable+(9*1d) l9
0400:000049c3 d9 48 1f 00 02 54 07 fa-03                      YH...T.z.
# db muxtable+(9*1e) l9
0400:000049cc d5 49 00 00 00 00 00 00-00                      UI.......
# db muxtable+(9*1f) l9
0400:000049d5 de 49 00 00 00 00 00 00-00                      ^I.......
# db muxtable+(9*20) l9
0400:000049de e7 49 00 00 00 00 00 00-00                      gI.......
# db muxtable+(9*21) l9
0400:000049e7 f0 49 00 00 00 00 00 00-00                      pI.......
# dp %%f1a6da l2
```

In this example there are only semaphore types 0, 1, 2 and 3. We will illustrate unravelling each of these in turn. For type 4 see the later section on 32-Bit semaphores.

The SYSSEM: The SYSSEM BlockID points to a SYSSEM table entry.

Note: In a MUXWAIT only the offset is recorded in the MUX table entry. This should be used with selector 400.

In a single SYSSEM, the BlockID is the selector:offset to the SYSSEM Table Entry. The .PB command will display the SYSSEM name.

```
The example below is from a MUXWAIT which includes a SYSSEM
>> The MUXTABLE entry for slot 58. SYSSEM offset = 53f0
# db muxtable+(9*b) l9
0400:00004921 2a 49 58 00 01 f0 53 00-00                      *IX..pS..

# .p 58
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0058  0014 0000 0014 0004 blk 021f 7bd30000 7bdfd260 7bde23f0 0eac 10 WKSTA

# .pb 58
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0058  blk fffd0058 WKSTA    MuxWait


>> The SYSSEM Data Table Entry
>> slot = 0058
>> flag = 02
>>           01= waiting
>>           02= mux waiting
```

```
>>            04= owner (Pid/Tid) died
>>            08= exclusive syssem
>>            10= name entry needs removing
>>            20= Tid owner died
>>            40= exit list thread owns this sem

>> reference count = 01
>> request count (by this owner) = 0
>> padding=00

# db 400:53f0 l6
0400:000053f0 58 00 02 01 00 00                               X.....


>> SYSSEM names are stored in a Record Management Package (RMP)
>> whose selector is the high word of:
# dd syssemrmphdl l1
0400:0000595a  04d00004

>> The RMP has a 0x14 byte header followed by variable length entries.
>> Each entry is prefixed with a word length followed by the entry data.
>> The entry data is the word offset of the corresponding SYSSEM Data Table
>> followed by offset 2 - n of the semaphore name. (the offset overlays
>> the first two bytes of the name which are always 'SE').

>>Scan the table looking for entry with offset 53f0...
# db 4d0:0
04d0:00000000 00 06 d0 02 0d 01 5b 03-01 00 00 00 00 04 00 00 ..P...[.........
04d0:00000010 36 ff 00 00 10 00 5a 53-45 4d 5c 56 49 4f 50 4f 6.....ZSEM\VIOPO
04d0:00000020 50 55 50 00 10 00 60 53-45 4d 5c 56 49 4f 50 52 PUP...`SEM\VIOPR
04d0:00000030 54 53 43 00 12 00 66 53-45 4d 5c 44 41 54 41 45 TSC...fSEM\DATAE
04d0:00000040 58 2e 45 52 52 00 12 00-6c 53 45 4d 5c 44 41 54 X.ERR...lSEM\DAT
04d0:00000050 41 45 58 2e 4c 4f 47 00-14 00 72 53 45 4d 5c 49 AEX.LOG...rSEM\I
04d0:00000060 50 43 51 55 45 55 45 2e-53 45 4d 00 12 00 78 53 PCQUEUE.SEM...xS
04d0:00000070 45 4d 5c 50 4d 44 52 41-47 2e 53 45 4d 00 14 00 EM\PMDRAG.SEM...
# d
04d0:00000080 7e 53 45 4d 5c 4c 4b 4e-45 44 30 30 30 2e 53 45 ~SEM\LKNED000.SE
04d0:00000090 4d 00 14 00 84 53 45 4d-5c 4c 4b 4e 45 44 30 30 M....SEM\LKNED00
04d0:000000a0 31 2e 53 45 4d 00 14 00-8a 53 45 4d 5c 4c 4b 4e 1.SEM....SEM\LKN
04d0:000000b0 45 44 30 30 32 2e 53 45-4d 00 14 00 90 53 45 4d ED002.SEM....SEM
04d0:000000c0 5c 4c 4b 4e 45 44 30 30-33 2e 53 45 4d 00 13 00 \LKNED003.SEM...
04d0:000000d0 96 53 45 4d 5c 53 4d 47-43 4f 4e 54 2e 53 45 4d .SEM\SMGCONT.SEM
04d0:000000e0 00 13 00 9c 53 45 4d 5c-50 4d 48 44 45 52 52 2e ....SEM\PMHDERR.
04d0:000000f0 53 45 4d 00 19 00 a2 53-45 4d 5c 4e 50 49 50 45 SEM..."SEM\NPIPE
# d
04d0:00000100 53 5c 52 49 50 56 41 4e-2e 57 4e 4b 00 19 80 00 S\RIPVAN.WNK....
04d0:00000110 00 79 02 5c 49 42 4d 4c-41 4e 5c 53 49 4e 47 4c .y.\IBMLAN\SINGL
04d0:00000120 45 2e 52 43 46 00 19 00-ae 53 45 4d 5c 54 49 4d E.RCF....SEM\TIM
04d0:00000130 45 58 45 43 5c 49 53 5c-4c 4f 41 44 45 44 00 17 EXEC\IS\LOADED..
04d0:00000140 00 b4 53 45 4d 5c 4d 41-47 4e 55 4d 5c 4d 41 49 .4SEM\MAGNUM\MAI
04d0:00000150 4e 2e 53 45 4d 00 16 00-ba 53 45 4d 5c 4e 45 54 N.SEM...:SEM\NET
04d0:00000160 5c 42 52 4f 57 53 4e 43-42 2e 30 00 16 00 c0 53 \BROWSNCB.0...@S
04d0:00000170 45 4d 5c 4e 45 54 5c 42-52 4f 57 53 4e 43 42 2e EM\NET\BROWSNCB.
# d
04d0:00000180 31 00 16 00 c6 53 45 4d-5c 4e 45 54 5c 42 52 4f 1...FSEM\NET\BRO
04d0:00000190 57 53 4e 43 42 2e 32 00-16 00 cc 53 45 4d 5c 4e WSNCB.2...LSEM\N
04d0:000001a0 45 54 5c 42 52 4f 57 53-4e 43 42 2e 33 00 16 00 ET\BROWSNCB.3...
04d0:000001b0 d2 53 45 4d 5c 4e 45 54-5c 42 52 4f 57 53 4e 43 RSEM\NET\BROWSNC
04d0:000001c0 42 2e 34 00 16 00 d8 53-45 4d 5c 4e 45 54 5c 42 B.4...XSEM\NET\B
```

```
04d0:000001d0 52 4f 57 53 4e 43 42 2e-35 00 16 00 de 53 45 4d ROWSNCB.5...^SEM
04d0:000001e0 5c 4e 45 54 5c 42 52 4f-57 53 4e 43 42 2e 36 00 \NET\BROWSNCB.6.
04d0:000001f0 18 00 e4 53 45 4d 5c 4e-45 54 5c 48 4f 53 54 41 ..dSEM\NET\HOSTA
# d
04d0:00000200 4e 4e 43 2e 53 45 4d 00-1d 00 ea 53 45 4d 5c 4e NNC.SEM...jSEM\N
04d0:00000210 45 54 5c 57 4b 53 54 41-5c 49 4e 54 45 52 47 54 ET\WKSTA\INTERGT
04d0:00000220 2e 53 45 4d 00 1d 00 f0-53 45 4d 5c 4e 45 54 5c .SEM...pSEM\NET\
04d0:00000230 57 4b 53 54 41 5c 52 45-4c 4f 47 4f 4e 2e 53 45 WKSTA\RELOGON.SE
04d0:00000240 4d 00 0f 00 f6 53 45 4d-5c 4d 53 52 56 57 55 30 M...vSEM\MSRVWU0
04d0:00000250 00 14 00 a8 53 45 4d 5c-4c 4b 4e 45 44 30 30 34 ...(SEM\LKNED004
04d0:00000260 2e 53 45 4d 00 14 00 02-54 45 4d 5c 4c 4b 4e 45 .SEM....TEM\LKNE
04d0:00000270 44 30 30 35 2e 53 45 4d-00 12 80 0d 01 5b 03 4e D005.SEM.....[.N

>> We find the entry at 4d0:227
>> The semaphore name is SEM\NET\WKSTA\RELOGON.SEM
```

The MUX RAMSEM: In a MUX wait the RAMSEM id is recorded as a hob:offset.

In this example we look at the RAMSEM being waited on by slot 1f.

```
>> The MUX table entry:
>> slot = 1f, type = 2, hob= 03fa, offset=0758
0400:000048d9 e2 48 1f 00 02 58 07 fa-03                     bH...X.z.
# db muxtable+(9*4) l9

>> Use .MOC to find the linear address
# .moc 3fa
*har     par      cpg        va     flg next prev link hash hob   hal
 039b %fef23f5c 00000010 %1a350000 379 0413 039c 0000 0000 03fa 0000  hco=00c45
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03fa  039b 0000  082c 03fb 03fb  0000 00  00 00 00 shared    e:pmshapi.dll
 hco=0c45 pco=ffe77d74  hconext=00ee0 hptda=0873 f=16  pid=00e0 e:cmd.exe

>> This is owned by PMSHAPI. LN gives a lable.
# ln %1a350758
%1a350750 PMSHAPI ASEMRS + 8
#
```

MUX Physical RAMSEM: In this example the MUX wait entry is for a physical address of a RAMSEM. A physical address would be used where the RAMSEM is in instance data - that makes it unique among RAMSEMs providing the RAMSEM is not swappable.

In this example the waiting slot is 1b

```
>> Mux table entry for slot 1b, type=3 (physical RAMSEM)
>> The physical address of the RAMSEM is %%00f1a6da
>> We need to determine the owner of this address.

# db muxtable+(9*e) l9
0400:0000493c be 48 1b 00 03 da a6 f1-00                     >H...Z&q.

>> Display the page frame structure for frame 00f1a:
# .mp f1a
ffe24538 InUse: pVP=ff4076ce RefCnt=0003 Flg=0 ll=01 sl=00 Blk=0006a Frame=00f1a

>> Now display the virtual page structure to see who has backed this
>> frame:
```

```
# .mv %ff4076ce
VPI=057b pVP=ff4076ce SOW Frame=0f1a Flg=9d0 HobPg=0000 Hob=03df Ref=011

>> Now we have the hob and page offset into the hob.  Display the
>> linear address of the Hob using .MOC, add the page offset and
>> the byte index from the physical address to obtain the
>> virtual address of the RAMSEM

# .moc 3df

*har     par      cpg        va    flg next prev link hash hob   hal
 0382 %fef23d36 00000010 %1a260000 379 0381 0383 0000 0000 03df 0000  hco=00f37
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03df  0382 0000  082c 03da 03da  0000 00  01 00 00 shared    e:pmwin.dll
 hco=0f37 pco=ffe78c2e  hconext=00e8b hptda=0873 f=16  pid=00e0 e:cmd.exe

>> RAMSEM is at %1a260000+00000000+6da
>> RAMSEM is owned by pmwin.dll

# ln %1a2606da
No Symbols Found

>> LN doesn't work so thunk to a selector:offset and try again
>> cheat by looking up the selector assigned to pmwin in its
>> segment table:
# .lmo 3da
hmte=03da pmte=%fdf21c14 mflags=0498b194 e:\os2\dll\pmwin.dll
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 0000f6f8 1a1b0000 80005025 00000001 00000010 03e9 d0df r-x shr alias conf
0002 0000c24e 1a1c0000 80005025 00000011 0000000d 03e8 d0e7 r-x shr alias conf
0003 00008c84 1a1d0000 80005025 0000001e 00000009 03e7 d0ef r-x shr alias conf
0004 0000b6e2 1a1e0000 80005025 00000027 0000000c 03e6 d0f7 r-x shr alias conf
0005 0000eb10 1a1f0000 80005025 00000033 0000000f 03e5 d0ff r-x shr alias conf
0006 00006292 1a200000 8000d025 00000042 00000007 03e4 d106 r-x shr alias conf iopl
0007 00003738 1a210000 8000d025 00000049 00000004 03e3 d10e r-x shr alias conf iopl
0008 000010c5 1a220000 80009025 0000004d 00000002 03e2 d116 r-x shr alias iopl
0009 000124d4 1a230000 80003025 0000004f 00000013 03e1 d11f r-x shr alias big
000a 000070ca 1a250000 80001025 00000062 00000008 03e0 d12f r-x shr alias
000b 00000ada 1a260000 80001063 0000006a 00000001 03df d137 rw- shr prel alias
000c 00001478 1a270000 80003063 0000006b 00000002 03de d13f rw- shr prel alias big
000d 000023f8 1a280000 80001063 0000006d 00000003 03dd d147 rw- shr prel alias
000e 00006444 1a290000 80001063 00000070 00000002 03dc d14f rw- shr prel alias
000f 00000142 1a2a0000 80001063 00000072 00000001 03db d157 rw- shr prel alias
0010 00000018 1a2b0000 80002063 00000073 00000001 03d9 d15f rw- shr prel big
0011 000003b8 16100000 80002079 00000074 00000001 051e b087 r-- rsrc disc shr prel big
0012 00000dcc 161c0000 80002069 00000075 00000001 0509 b0e7 r-- rsrc shr prel big
0013 0000ffbc 16210000 80002029 00000076 00000010 0504 b10f r-- rsrc shr big
0014 000002f0 00000000 00002039 00000086 00000001 0000 0000 r-- rsrc disc shr big
0015 00003524 16120000 80002029 00000087 00000004 051b b097 r-- rsrc shr big

# ln d137:6dad137:000006da PMWIN MSGQUEUESEM1
```

Structured Semaphores: We have discussed the following types of semaphore:

RAMSEM

SYSSEM

FSRAMSEM

PMFSRAMSEM

There are three others that occur with regularity in the system:

KSEM

32-bit SEM

GRESEM/PMSEM

The Kernel Semaphore (KSEM): The kernel semaphore is a RAMSEM and EVENT SEM with accountability in-built.

Many system control block have imbedded KSEMs. Included among these are the PTDA and MFT.

Some KSEMs are allocated out of the kernel heaps and have the owner mnemonic *KSEM*.

When a thread blocks on a KSEM the address of the KSEM is used as the BlockID.

Under the debug (ALLSTRICT) kernel the KSEM contains an additional signature ′KSEM′. Always check a BlockID address to see if the ′KSEM′ signature is present.

`.D KSEM` will format the KSEM.

In this example we look at Slot 6c to find out why it will not run.

```
# .pb 6c
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 006c  blk 7bdfc910 DEMO1

# .m 7bdfc910
*har     par      cpg        va    flg next prev link hash hob   hal
 0087 %fef1fba4 00000082 %7bdf5000 121 0085 0088 0000 0000 0089 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0089  0087 0000  0325 ffcb 0000  0000 00  00 00 00 ptda

>> This thread is blocked on an address in (its) PTDA. All PTDA
>> semaphores are KSEMs.

# .d KSEM %7bdfc910
Signature     : KSEM                        Nest: 0001
Type          : MUTEX
Flags         : 00
Owner         : 0041              PendingWriters: 0001

>> So the owner is Slot 41. Lets look at him to see what he′s up to.

# .pb 41
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0041  blk 04085ca7 DEMO1
# .m #408:5ca7
*har     par      cpg        va    flg next prev link hash hob   hal
 0079 %fef1fa70 00000010 %7bf27000 129 0078 0077 0000 0000 007b 0000   sel=0408
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 007b  0079 0000  0324 ffa1 0000  0000 00  00 00 00 sft
#
```

```
>> Slot 41 is blocked waiting for some file system activity to complete.
>> We looked at this slot some time ago and found out that it was
>> waiting to close a device driver.
```

The 32-Bit Semaphore Event and Mutex Semaphores: BlockIDs for 32-bit sems point to kernel heap allocated structure with object mnemonic *semstruc*.

.PB under the KDB usually identifies these as *SEM32*, but DF doe not.

The &peirod.D SEM32 command will format a 32-bit semaphore structure.

There are several structures that relate to 32-bit semaphores. Each of these is allocated from the kernel heaps and is assigned the following meaningful owner id mnemonics:

semmuxq (0xffbe) Semaphore Mux Queue. This records instances of single event or mutex semaphores being also waited on in a mux wait.

semopenq (0xffbf) Semaphore Open Queue. This tracks all processes that have opened a 32-bit semaphore.

semrec (0xffc0) SemRecord. This is a system copy of the user's SemRecord structure, which was created when a Mux wait was declared. It correlates user semaphore Ids with semaphore handles.

semstr (0xffc1) The semaphore name string.

semstruc (0xffc2) The main 32-bit structure. The address of this forms the BlockID when a thread waits on a 32-bit semaphore.

Of the associated structures listed above the Open Queue and Mux Queue may be formatted using:

:.D OPENQ

.D MUXQ

In this example we look at the BlockID slot 42 is waiting on.

```
# .p 42 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0042  000d 000a 000d 0002 blk 0200 7bd1a000 7bdfa188 7bddfe20 0ed4 11 PMSHL32

# .pb42
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0042  blk fe0bf91c PMSHL32

>> check owner of BlockID
# .m %fe0bf91c

*har     par      cpg        va    flg next prev link hash hob   hal
 0003 %fef1f04c 00001000 %fdf1f000 001 0002 0020 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 ff08  0000 ffec 0000  0000 00  06 00 00 vmkrhrw

>> kernel swappable heap.  Check current user of heap block.

# dd  %fe0bf91c-10
%fe0bf90c  ffbf000c 00010008 fe0bff20 ffc20014
```

```
%fe0bf91c  00000011 00000000 fef1ef94 fcae5a28
%fe0bf92c  ffbf0010 00010006 fe08f24c fe0bf92e
%fe0bf93c  ffbf000c 00010006 fe08f7c0 ffa4000c
%fe0bf94c  fe0567d0 00010494 ffbf000c 0001000a
%fe0bf95c  00000000 ffa4000c fe0bf970 000104ec
%fe0bf96c  ffa4000c fef1ef4c 000200ba ffbf000c
%fe0bf97c  00010005 fe0bf758 ffbf000c 00010005

# .mo ffc2
ffc2 semstruc

>> This is a 32-bit Semaphore

# .d sem32 %fe0bf91c
        Type: Shared Event
       Flags: Reset
       pMuxQ: 00000000
  Post Count: 0000
      pOpenQ: fef1ef94
       pName: fcae5a28
 Create Addr: ffbf0010

# .d openq %fef1ef94

  Pid   Open Count
  ----------------
  000d     0001
  000a     0001

# da %fcae5a28
%fcae5a28 WORKPLAC\LAZYWRIT.SEM

>> For interest look for the owner of the OPENQ:

# .m %fef1ef94
*har     par      cpg        va    flg next prev link hash hob   hal
 0003 %fef1f04c 00001000 %fdf1f000 001 0002 0020 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 ff08  0000 ffec 0000  0000 00  06 00 00 vmkrhrw

# dd %fef1ef94 -10
%fef1ef84  00000000 00000000 fef10001 ffbf000c
%fef1ef94  0001000d fe0bf958 ffc20018 00000009
%fef1efa4  00000000 00000000 fef1ef58 fcb18478
%fef1efb4  ffbf000c 0001000d 00000000 ffc20018
%fef1efc4  00000009 00000000 00000000 fef1efb8
%fef1efd4  fcb18458 ff910024 00000007 00000000
%fef1efe4  00000000 00000000 00000000 00000000
%fef1eff4  00000000 00000000 ffea0004 fef2a128
# .mo ffbfffbf semopenq

>> For interest look for the owner of the pName:

# .m  %fcae5a28

*har     par      cpg        va    flg next prev link hash hob   hal
 0021 %fef1f2e0 00001400 %fca5f000 121 0020 0022 0000 0020 0022 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
```

```
 0022  0021 0000  0225 ffef 0000  0000 00  04 00 00 vmkshrw
# dd  %fcae5a28-10
%fcae5a18  003f1d1a 0007004a 52000020 0000ffc1
%fcae5a28  4b524f57 43414c50 5a414c5c 49525759
%fcae5a38  45532e54 5412004d 520000c8 0000ff60
%fcae5a48  005d004a 1b131b12 1b151b14 1b171b16
%fcae5a58  1b191b18 1b1b1b1a 1b1d1b1c 1b1f1b1e
%fcae5a68  1b211b20 1b231b22 1b251b24 1b271b26
%fcae5a78  1b291b28 1b2b1b2a 1b2d1b2c 1b2f1b2e
%fcae5a88  1b311b30 1b331b32 1b351b34 1b371b36

# .mo ffc1
ffc1 semstr
```

PMSEM/GRESEM: 32-bit PM (WARP) and Graphics Engine use a composite semaphore structure to serialize their resources.

This semaphore has the structure:

+0x0 7 byte Signature. 'PMSEM' for PMWIN and 'GRESEM' for PMGRE.

+0x7 386 semaphore byte (PM uses the *bts* instruction on this under 386 processors otherwise it uses the 486 *cmpxchg* on the Pid/Tid).

+0x8 Owner Pid (word).

+0xa Owner Tid (word).

+0xc Owner nested use count (long).

+0x10 Number of waiters.

+0x14 Number of times sem used (zero unless Debug version of PM).

+0x18 Handle for event semaphore.

+0x1c Address of caller (zero unless Debug version of PM).

PM uses a technique of polling this semaphore by waiting on the imbedded event semaphore handle for a limited time.

This technique has the advantage of speed combined with accountability but a thread waiting for a PMSEM or GRESEM may appear blocked, ready or running depending on the polling cycle. However it will be executing in a routine with a name such as *PMRequestMutexSem*. If the PMMERGE symbols are loaded this is readily detected.

The PM and GRE SEMs are contiguous and located at label **pmSemaphores**.

The handle (linear address) of the PM/GRE Semaphore is passed on entry to PMREquestMutexSem and tends to be retained in the EDX register.

The following semaphores are defined by PM:

0 PMSEM ATOM

1 PMSEM USER

2 PMSEM VISLOCK

3 PMSEM DEBUG

4 PMSEM HOOK

5 PMSEM HEAP

6 PMSEM DLL

7 PMSEM THUNK

8 PMSEM XLCE

9 PMSEM UPDATE

10 PMSEM CLIP

11 PMSEM INPUT

12 PMSEM DESKTOP

13 PMSEM HANDLE

14 PMSEM ALARM

15 PMSEM STRRES

16 PMSEM TIMER

17 PMSEM CONTROLS

18 GRESEM GreInit

19 GRESEM AutoHeap

20 GRESEM PDEV

21 GRESEM LDEV

22 GRESEM CodePage

23 GRESEM HFont

24 GRESEM FontCntxt

25 GRESEM FntDrvr

26 GRESEM ShMalloc

27 GRESEM GlobalData

28 GRESEM DbcsEnv

29 GRESEM SrvLock

30 GRESEM SelLock

31 GRESEM ProcLock

32 GRESEM DriverSem

33 GRESEM semIfiCache

34 GRESEM semFontTable

In this example one of the shell threads seems to be getting very little CPU, though is frequently ready:

```
# .p 3a
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*003a# 000d 0005 000d 000a rdy 0200 abd61000 abe4b5b4 abe2ee60 0ee4 11 PMSHL32

# .r
eax=13e30025 ebx=00000000 ecx=000a000d edx=13e7b4d4 esi=ffffffff edi=0068e55c
eip=1bd0d7ea esp=00637f44 ebp=00637f60 iopl=2 -- -- -- nv up ei pl nz na po nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001ad000
005b:1bd0d7ea ff4a10         dec     dword ptr [edx+10]    ds:13e7b4e4=00000006
```

```
# ln
%1bd0d770 PMMERGE PMREQUESTMUTEXSEM + 7a

# db %edx
%13e7b4d4 50 4d 53 45 4d 00 00 00-10 00 01 00 02 00 00 00 PMSEM...........
%13e7b4e4 06 00 00 00 00 00 00 00-05 00 01 80 00 00 00 00 ................
%13e7b4f4 50 4d 53 45 4d 00 00 00-00 00 00 00 00 00 00 00 PMSEM...........
%13e7b504 00 00 00 00 00 00 00 00-06 00 01 80 00 00 00 00 ................
%13e7b514 50 4d 53 45 4d 00 00 00-00 00 00 00 00 00 00 00 PMSEM...........
%13e7b524 00 00 00 00 00 00 00 00-07 00 01 80 00 00 00 00 ................
%13e7b534 50 4d 53 45 4d 00 00 00-00 00 00 00 00 00 00 00 PMSEM...........
%13e7b544 00 00 00 00 00 00 00 00-08 00 01 80 00 00 00 00 ................

>> PMSEM owner is Pid 10 Tid 1 and it has been requested twice by
Tid/Pid 10/1.  There are 6 waiting threads.

# .p 42
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0042  0010 0005 0010 0001 blk 0500 abd69000 abe4c19c abe2fe20 0ed8 13 MRFILEPM

# .pb 42
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0042  blk fdf8841c MRFILEPM

>> The owner is blocked.

# .m %fdf8841c

*har     par      cpg        va     flg next prev link hash hob   hal
 0003 %feeef04c 00001000 %fdeef000 001 0002 0021 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 ff05  0000 ffec 0000  0000 00  02 00 00 vmkrhrw

# dd %fdf8841c-10 l4
%fdf8840c  000101c1 00000000 fdf88406 ffc20018

# .mo ffc2
ffc2 semstruc

# .d sem32 %fdf8841c
        Type: Shared Event
       Flags: Reset
       pMuxQ: 00000000
  Post Count: 0000
      pOpenQ: fde305f8
       pName: NULL (anonymous)
 Create Addr: abe2fe20

# .d openq  %fde305f8

  Pid   Open Count
  ----------------
  0010     0001

#
# ln pmsemaphores
9f3f:0000b4b4 PMMERGE PMSEMAPHORES
# dl 9f3f
9f3f  Data    Bas=13e70000 Lim=0000ffff DPL=3 P  RW    A
```

```
#
>> The sem we were waiting on was at %13e7b4d4 so must be the
>> USER SEM.
```

13.1.2.2 Involuntary Suspension

In this section we discuss the mechanisms involved when a thread involuntarily gives up CPU processing time. That is, another thread independently causes a thread not to receive or to give up its time-slice.

The mechanisms available that cause suspension are:

Preemption
: Another thread of a high priority becomes ready.

  The suspended thread becomes ready and the pre-empting thread runs.

Critical Section
: Another thread in the same process enters critical section.

  The critical section thread runs and none of the other threads will run except if a signal fires. If another ready thread in the same process is selected by the dispatcher for running it is helped on a temporary queue with its status set to *crt*.

  Note: The critical section thread has *run* status.

DosSuspendThread
: Another thread in the same process has issued DosSuspendThread.

  The suspending thread runs and the suspended thread enters *frz* state.

Freeze Process
: Either a Session Manager switch is in progress, a new process has been created suspended, a Virtual Device Driver has called the VDHFreezeVDM helper routine or the DosDebug DBG_C_Freeze command has been executed against a debugee process.

  The frozen process has a state of *frz* in all its threads.

Voluntary suspension is indicated by the *blk* state.

When a thread is suspended involuntarily it will normally be in one of the following states:

rdy Ready and waiting to run.

crt Ready but prohibited from dispatch by a critical section thread.

frz Frozen or Suspended by freeze-process or DosSuspendThread.

The remaining six thread states related to transient system processing on behalf of a thread. These are the following:

dly Delayed wake-up. Multiple threads have been woken from a blocked state because they were all waiting on the same BlockID and a multiple wake-up was specified to ProcRun. Each delayed thread is queued pending scheduling where priority recalculation and the thread's ring 0 stack is checked for presence in memory. If all is well then the thread is placed on the ready queue pending dispatch. If not, then the thread is placed on the TSD Daemon′s queue for paging in the thread′s TSD (ring 0 stack).

tsd The thread is on the TSD Daemon′s queue waiting for ring 0 stack page-in. The Daemon runs as an internal thread, which is labelled *tsd by the `.P` command. This thread is responsible for calling the page manager to page in a thread′s TSD. Because a paging operation involves I/O and is therefore relatively slow, this operation is performed under the control of a separate thread. This allows other threads to be processed while the paging operation takes place.

gsk Get Stack request in progress. The TSD Daemon is waiting for the Page Manager to signal completion of the paging I/O operation. Effectively a thread in this state is blocked waiting for completion of a TSD paging I/O request.

bst Boosted Ready State. When the TSD page-in completes successfully, the thread is placed on the dispatchers ready queue with a priority boost. This condition is indicated by the boosted ready state. Strictly speaking this is not an independent state since no operation is required to take the thread from *bst* to *rdy*.

bad TSD page-in request has failed. This is a serious and terminal condition, which is not expected to occur. It is possible that an I/O error has occurred during the TSD page-in request.

--- The null state occurs very fleetingly during thread creation and termination. It signifies that the thread′s environment is incomplete.

The complete set of scheduler states for a finite state machine, which is illustrated in the following diagram.

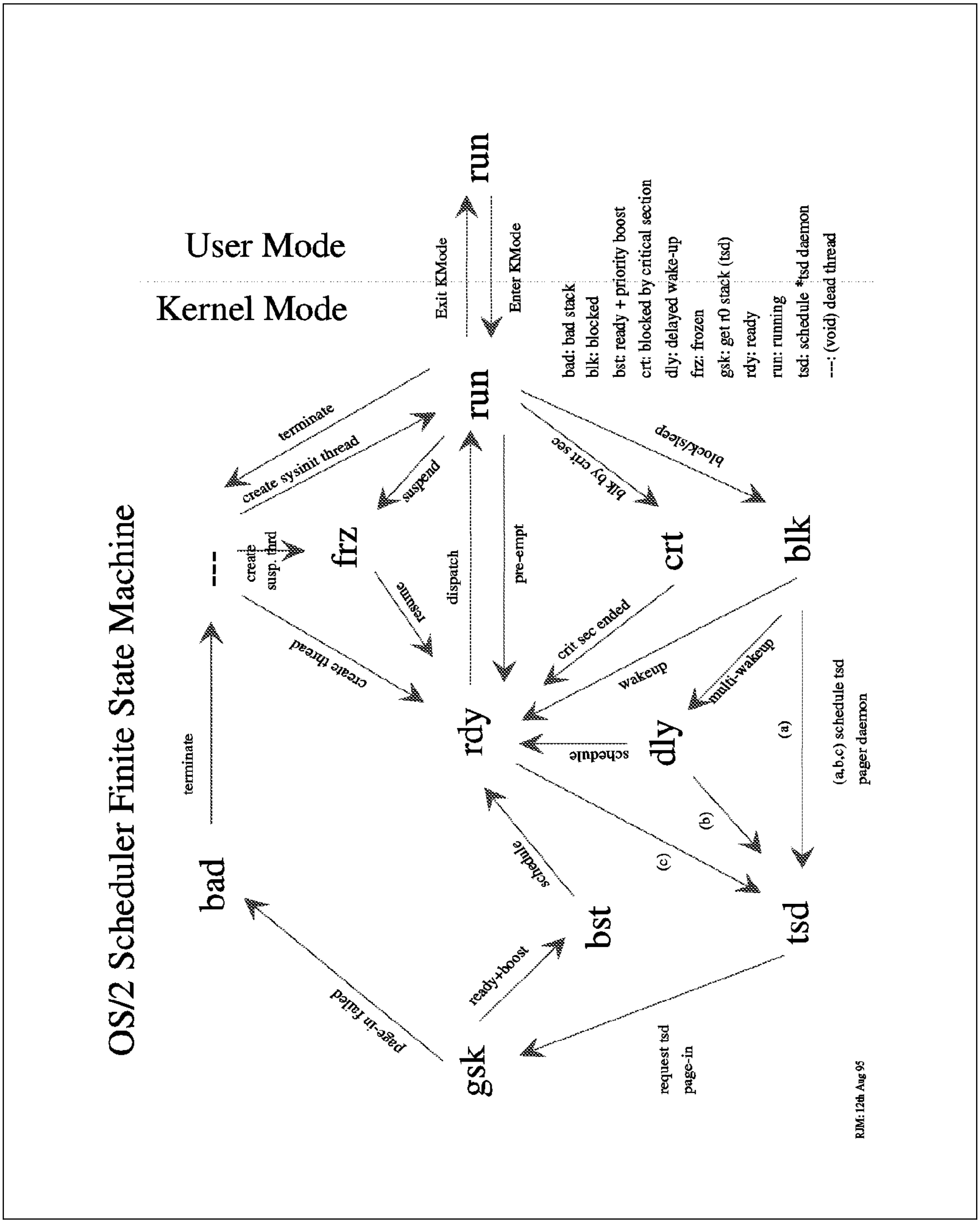


Figure 12. Scheduler States for a Finite State Machine

Preemption and Priority Calculation: A thread is preempted when higher priority work becomes ready to process. Under normal circumstances the preempting thread will run then give up its time-slice and eventually the original thread will be re-scheduled.

It is possible for a thread not to be re-scheduled if a higher priority thread will not give up the processor. However, the OS/2 scheduler applies dynamic boost to priorities according to resource requirements and makes priority comparisons based on a calculated priority. The elements involved in the priority calculation are the following:

TCBPriClass
: The thread's priority class. There are four classes, which in order of priority are:

3 Time-critical

4 Foreground Server (or fixed high)

2 Regular

1 Idle

TCBPriLevel
: The priority delta which may range from 0x00 to 0x1f.

TCBPriClassMod
: The priority boosts which may be any of the combined values:

0x04 Keyboard Boost

0x08 CPU Starvation Boost

0x10 Device I/O Boost

0x20 Foreground Boost

0x40 Window Boost

0x80 VDM Simulated Interrupt

TCBPriorityMin
: The minimum allowed priority. Normally 0 but set when priority inversion becomes a possibility. This is discussed later.

Priority is calculated by forming an index by ORing TCBPriClass and TCBPriClassMod and reading a constant value from the priority table. The low byte of this is then further ORed with the TCBPriLevel.

The following diagram shows the priority table.

```
     Table Index = (TCBPriClass | TCBPriClassMod)

     Starved         08 ------------------------------------------------+
     Device I/O      10 -----------------------------------------------+|
     Foreground      20 ----------------------------------------------+||
     Window          40 ---------------------------------------------+|||
     VDM Interrupt   80 --------------------------------------------+||||
                                                                    |||||
+-- TCBPriClass                                                     |||||
|                                                                   |||||
|    ============================================================   |||||
|    Not Keyboard                ||  Keyboard                       |||||
|                                ||
+-> Server  Idle    Regular TC  ||  Server  Idle    Regular TC      IWFDS


     0x300,  0x100,  0x200,  0x800,  0x300,  0x100,  0x200,  0x800,// -----
     0x62f,  0x100,  0x61f,  0x800,  0x62f,  0x100,  0x61f,  0x800,// ----S
     0x72f,  0x100,  0x71f,  0x800,  0x72f,  0x100,  0x71f,  0x800,// ---D-
     0x72f,  0x100,  0x71f,  0x800,  0x72f,  0x100,  0x71f,  0x800,// ---DS
     0x300,  0x100,  0x300,  0x800,  0x300,  0x100,  0x400,  0x800,// --F--
     0x62f,  0x100,  0x61f,  0x800,  0x62f,  0x100,  0x61f,  0x800,// --F-S
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// --FD-
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// --FDS
     0x500,  0x100,  0x500,  0x800,  0x500,  0x100,  0x500,  0x800,// -W---
     0x62f,  0x100,  0x61f,  0x800,  0x62f,  0x100,  0x61f,  0x800,// -W--S
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// -W-D-
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// -W-DS
     0x500,  0x100,  0x500,  0x800,  0x500,  0x100,  0x500,  0x800,// -WF--
     0x62f,  0x100,  0x61f,  0x800,  0x62f,  0x100,  0x61f,  0x800,// -WF-S
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// -WFD-
     0x74f,  0x100,  0x73f,  0x800,  0x74f,  0x100,  0x73f,  0x800,// -WFDS
```

Notes:

VDM Simulated interrupts always result in a value of 0x800

Foreground server class is not affected by the keyboard boost.

Time-critical class is not affected by any boosts.

Idle class is not affected by any boosts.

By examining the priority table it is clear that idle class will always be preempted by any other class.

Time-critical class can never be preempted by any other class.

Time-critical threads can only be preempted by other time-critical threads with a higher delta.

Server and regular class threads may preempt each other depending on priority boosts and delta.

The key to looking at preemption problems is to look for other CPU bound threads of a higher priority. In particular time-critical threads.

The .P command displays the current calculated priority for each thread.

Critical Sections: When a thread enters critical section it effectively suspends all other threads in its process. There is an exception to this. If a signal is sent to the process and a signal handler is registered, then thread 1 will be dispatched to run the signal handler regardless of critical section.

The critical section thread may voluntarily block.

Other threads may attempt to become ready. If this happens the dispatcher will temporarily suspend them in *crt* state.

The appearance of the *crt* state certainly guarantees that another thread in the same process is in critical section. However, the converse in not true: the absence of *crt* does not preclude another thread from being in a critical section.

If a thread running in critical section blocks on a resource owned by any other thread in the same process then a deadlock will result. Because of this it is unwise to call any system API when in critical section.

Thread running in critical section have their TCB address stored in their process's PTDA at ptda_pTCBCritSec.

The following example illustrates locating the critical section thread in a process.

```
# .p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe4b000 ffe4c7dc ffe4c624 0e84 00 *ager
 0002  0001 0000 0000 0002 blk 0200 7a49e000 ffe4c7dc 7b49c020 0f44 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 7a49f000 ffe4c7dc 7b49c1d8 0f54 00 *ctxh
 0004  0001 0000 0000 0004 blk 0800 7a4a0000 ffe4c7dc 7b49c390 0f24 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 7a4a1000 ffe4c7dc 7b49c548 0f40 00 *lazyw
 000a  0004 0000 0004 0001 blk 0200 7a4a6000 7b655068 7b49cde0      00 LANMSGEX
*000c# 0006 0000 0006 0001 blk 0804 7a4a8000 7b6560b0 7b49d150 0c94 00 CNTRL
 000d  0006 0000 0006 0002 blk 0804 7a4a9000 7b6560b0 7b49d308      00 CNTRL
 000e  0006 0000 0006 0003 blk 0804 7a4aa000 7b6560b0 7b49d4c0 0f04 00 CNTRL
 000f  0006 0000 0006 0004 blk 0804 7a4ab000 7b6560b0 7b49d678 0f04 00 CNTRL
 0010  0006 0000 0006 0005 blk 0804 7a4ac000 7b6560b0 7b49d830      00 CNTRL
 0011  0006 0000 0006 0006 blk 0804 7a4ad000 7b6560b0 7b49d9e8 0cc4 00 CNTRL
 0012  0006 0000 0006 0007 blk 0804 7a4ae000 7b6560b0 7b49dba0 0f04 00 CNTRL
 0013  0006 0000 0006 0008 blk 0804 7a4af000 7b6560b0 7b49dd58 0cb0 00 CNTRL
 0007  0007 0001 0007 0001 blk 0500 7a4a3000 7b6568d4 7b49c8b8 0ebc 01 PMSHL32
 000b  0007 0001 0007 0002 blk 0800 7a4a7000 7b6568d4 7b49cf98      01 PMSHL32
 0009  0007 0001 0007 0003 blk 0800 7a4a5000 7b6568d4 7b49cc28      01 PMSHL32
 0014  0007 0001 0007 0004 blk 0800 7a4b0000 7b6568d4 7b49df10      01 PMSHL32
 0015  0007 0001 0007 0005 blk 0800 7a4b1000 7b6568d4 7b49e0c8      01 PMSHL32
 0006  0007 0001 0007 0006 blk 0200 7a4a2000 7b6568d4 7b49c700      01 PMSHL32
 0018  0007 0001 0007 0007 blk 0200 7a4b4000 7b6568d4 7b49e5f0 0ed4 01 PMSHL32
 0019  0007 0001 0007 0008 blk 0200 7a4b5000 7b6568d4 7b49e7a8      01 PMSHL32
 001a  0007 0001 0007 0009 blk 0200 7a4b6000 7b6568d4 7b49e960      01 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001b  0007 0001 0007 000a blk 0800 7a4b7000 7b6568d4 7b49eb18      01 PMSHL32
 001c  0007 0001 0007 000b blk 0800 7a4b8000 7b6568d4 7b49ecd0      01 PMSHL32
 001d  0007 0001 0007 000c blk 0800 7a4b9000 7b6568d4 7b49ee88      01 PMSHL32
 001e  0007 0001 0007 000d blk 0804 7a4ba000 7b6568d4 7b49f040      01 PMSHL32
 001f  0007 0001 0007 000e blk 0804 7a4bb000 7b6568d4 7b49f1f8      01 PMSHL32
 0020  0007 0001 0007 000f blk 0500 7a4bc000 7b6568d4 7b49f3b0      01 PMSHL32
 0021  0007 0001 0007 0010 blk 0200 7a4bd000 7b6568d4 7b49f568 0ebc 01 PMSHL32
 002f  0012 0007 0012 0001 blk 0200 7a4cb000 7b658140 7b4a0d78      15 CMD
```

```
 002e  0011 0007 0011 0001 blk 0200 7a4ca000 7b65791c 7b4a0bc0      14 CMD
 0026  000b 0007 000b 0001 blk 0400 7a4c2000 7b6570f8 7b49fe00 0ebc 12 CMD
 0023  000a 0007 000a 0001 blk 0500 7a4bf000 7b654844 7b49f8d8 0ebc 11 PMSHL32
 0024  000a 0007 000a 0002 blk 0200 7a4c0000 7b654844 7b49fa90      11 PMSHL32
 0025  000a 0007 000a 0003 blk 0200 7a4c1000 7b654844 7b49fc48 0ebc 11 PMSHL32
 0022  000a 0007 000a 0004 blk 0200 7a4be000 7b654844 7b49f720      11 PMSHL32
 0027  000a 0007 000a 0005 blk 0200 7a4c3000 7b654844 7b49ffb8 0ed4 11 PMSHL32
 0028  000a 0007 000a 0006 blk 0200 7a4c4000 7b654844 7b4a0170      11 PMSHL32
 0029  000a 0007 000a 0007 blk 0200 7a4c5000 7b654844 7b4a0328      11 PMSHL32
 002a  000a 0007 000a 0008 blk 0200 7a4c6000 7b654844 7b4a04e0      11 PMSHL32
 002c  000a 0007 000a 000a blk 0200 7a4c8000 7b654844 7b4a0850 0eb0 11 PMSHL32
 002d  000a 0007 000a 000b blk 0200 7a4c9000 7b654844 7b4a0a08 0ebc 11 PMSHL32
 0008  0008 0007 0008 0001 blk 0800 7a4a4000 7b654020 7b49ca70      00 HARDERR
 0016  0008 0007 0008 0002 blk 0800 7a4b2000 7b654020 7b49e280      00 HARDERR
 0017  0008 0007 0008 0003 blk 0800 7a4b3000 7b654020 7b49e438      00 HARDERR
 002b  0013 0011 0013 0001 blk 0200 7a4c7000 7b65588c 7b4a0698      14 DEMORUN
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0030  0014 0013 0014 0001 blk 0200 7a4cc000 7b658964 7b4a0f30      14 CMD
 0031  0015 0014 0015 0001 crt 0809 7a4cd000 7b659188 7b4a10e8 0f24 14 DEMOCRT
 0032  0015 0014 0015 0002 crt 080b 7a4ce000 7b659188 7b4a12a0 0f0c 14 DEMOCRT
 0033  0015 0014 0015 0003 blk 080b 7a4cf000 7b659188 7b4a1458 0cc4 14 DEMOCRT
 0034  0015 0014 0015 0004 crt 011e 7a4d0000 7b659188 7b4a1610 0f20 14 DEMOCRT
 0035  0015 0014 0015 0005 crt 080f 7a4d1000 7b659188 7b4a17c8 0f0c 14 DEMOCRT
 0036  0015 0014 0015 0006 blk 080a 7a4d2000 7b659188 7b4a1980 0c04 14 DEMOCRT
 003c  0015 0014 0015 0007 blk 080a 7a4d8000 7b659188 7b4a23d0 0eb8 14 DEMOCRT
 0038  0015 0014 0015 0008 blk 080a 7a4d4000 7b659188 7b4a1cf0      14 DEMOCRT
 0039  0015 0014 0015 0009 blk 080a 7a4d5000 7b659188 7b4a1ea8      14 DEMOCRT
 003a  0015 0014 0015 000a crt 080c 7a4d6000 7b659188 7b4a2060 0f0c 14 DEMOCRT
 003b  0015 0014 0015 000b blk 080c 7a4d7000 7b659188 7b4a2218 0c80 14 DEMOCRT

# dd %7b659188+ptda_ptcbcritsec-ptda_start l1
%7b6596c0  7b4a23d0

# dw %7b4a23d0 l2
%7b4a23d0  0007 003c

# .pb 3c
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 003c  blk fffe0027 DEMOCRT  RamSem    00bf:0024
```

In this example Pid 15 is stuck, threads are either blocked or suspended by critical section.

From the PTDA we find the critical section TCB address. From this we can either scan the .P command output listing for the TCB address or look at the second word, which contains the slot number for the thread.

The critical section thread has blocked on a RAMSEM whose address is *00bf:0024*. Since the selector is less than *2007* this has to be in its private arena. This is significant; only another thread in the same process could possibly post this semaphore.

Suspension and Freezing: Suspension is achieved by any thread in a process calling DosSuspendThread. There is no accounting information associated with this API. One must examine all threads in the process to see if they are functioning correctly.

Freezing occurs for a number of reasons:

A new process has been created with the thread initially suspended.

The Session Manager (Shell Process 1) has used DosSystemService to freeze all threads of a process while a screen group switch occurs.

A Virtual device driver has called VDHFreezeVDM.

A Debug thread has called DosDebug using the DBG_C_Freeze command.

Again, there is no accounting information kept for this state.

If a single thread exists in the frozen process, check the parent process's threads for correct functioning.

If all threads are frozen check the Shell process 1 for correct processing.

13.1.2.3 Priority Inversion

Priority inversion is a hybrid situation that involves both the involuntary and voluntary suspension of two threads.

Consider the following:

A high priority thread is blocked on a resource.

A low priority thread owns the resource on which the high priority thread is blocked.

An independent thread of intermediate priority is running.

```
┌──────────────────────────────────────────┐
│                                          │
│   ┌─────────┐                            │
│   │Thread 1 │                            │
│   │ High    │                            │
│   │ blk     │                            │
│   └─────────┘                            │
│                                          │
│       ║                                  │
│       ║                                  │
│       ║                                  │
│ waits ║                                  │
│  on   ║                   ┌────────┐     │
│ SEM   ║                   │Thread 3│     │
│ owned ║ < - - - - - - - - │ Medium │     │
│  by   ║                   │  run   │     │
│ thd 2 ║                   └────────┘     │
│       ║                                  │
│       V                                  │
│                                          │
│    ┌────────┐                            │
│    │Thread 2│                            │
│    │  Low   │                            │
│    │  rdy   │                            │
│    └────────┘                            │
│                                          │
└──────────────────────────────────────────┘
```

Thread 1 will not run until thread 2 gets a time-slice that allows it to run and release the semaphore thread 1 is waiting for.

Since thread 3 is a higher priority than thread 2 and is CPU bound, thread 2 never runs, nor does thread 1.

Thread 1′s priority has effectively been reduced to that of thread 2′s by a lower priority thread, thread 3. Thread 1 is said to have its priority inverted with respect to thread 3.

The Kernel implements an automatic inversion protection mechanism whenever a process blocks using a KSEM. This mechanism is implemented by the following three routines:

TKEnterInversion
Called to protect against priority inversion. For example, when a mutex KSEM is obtained increments TCBcBoostLock.

TKExitInversion
When an inversion protected KSEM is released TCBcboostlock is decremented. When TCBcBoostlock is zero and TCBPriorityMin is not zero then is set to zero and priority recalculated

TKDeclareInversion
Used to set the minimum priority of a thread to be waited on. If the owner′s TCBPriorityMin < waiter′s and owner is in ready state then TCBPriority then set the owner′s TCBPriorityMin to our priority +1.

For this mechanism to work, it must be possible to determine ownership from the semaphore so that TKDeclareInversion can determine which thread′s priority to alter. It is also necessary to be able to determine whether raising the priority of thread will lead to other synchronization problems or deadlocks through race conditions. Since the kernel is a special case and preemption cannot occur while running in kernel mode, the kernel limits inversion protection to the KSEM only.

13.2 Program Design Issues and Weaknesses

The following hit-list identifies potential weaknesses in program design that can lead to hang symptoms or serialization problems:

1. Manipulation of thread priorities for the purpose of serialization or sequencing execution is haphazard at best. At worst the performance of the entire system can be jeopardized.

   The following guidelines should be applied when considering priority manipulation:

   - Use priority delta to tell the system the relative importance of an application′s threads.
   - Avoid priority class manipulation. Priority class tends to specify the relative importance of a thread with respect to all other threads in the system.
   - Avoid the use of time-critical priority. By setting this class, a thread is assuming the position of utmost importance in the system. This may not be a valid assumption for some system configurations and some users.
   - If priority class manipulation is desirable under some circumstances, then it should be parameterized so that it can be controlled as an option by the user.

2. If a window of exposure exists it will be exposed.
3. Any common resource that is ever modified must have an associated lock or serialization mechanism.
4. Locks (serialization techniques such as semaphores) that are concurrently held and waited on must be obtained in an established order.
5. Simplistic approach (one lock) forces work to be channelled through a single-queue. Therefore design locks at the lowest level of contention.
6. Distinguish process/data/repository serialization otherwise an inconsistent system of locks may result:

   - Process locks are required where a only single instance of a process is allowed to operate. For example:

     Finite State Machine state transitions;

     Some FSM state users;

     Any non-reentrant process.

   - Distinguishing Repository locks allows the repository is updated:

     Disk/directory reorganization while file data is in use.

     Physical page assignments are allowed to change while data is in use - swapping.

7. Data optimization: Artificial association of unrelated data items imposes serialization constraints that will have two possible effects:

   - This necessitates unrelated processes to serialize.
   - Serialization may lead to unavoidable deadlocks.

8. Code optimization: imposes process lock constraints in a similar way that data optimization does.

9. O-O tends to hide the data repository and structure. May even hide the process. Therefore designers need to consider whether locks are managed internally, within the object or explicitly. It may not be possible to handle the locks internally, because the context in which an instance method is being use (that is, the process) is not discernible from within the object.

Chapter 14. Worked Examples

The following collection of worked examples illustrate how to use the debugging tools, in particular the Dump Formatter and Kernel Debugger to obtain information from a system under diagnosis.

The following topics are included:

14.1.1 How to Find File System Information

This section gives a basic overview of the file system control blocks, and shows how to answer the following questions:

1. What file system objects are open in a given process?
2. What file system objects are open in a VDM?
3. What file corresponds to a given handle?
4. What processes have opened a given file system object?

If the reader is unfamiliar with this subject then the sections that follow should be read in order:

These topics now follow:

Finding files from handles

Finding files from handles in a VDM

Finding handles from file names

The Record Lock Record

Note: The examples included in this section are worked on an OS/2 2.11 system. For OS/2 WARP the same techniques work, however the *SFTs*, whilst they may be located from the *SAS* in the same manner, they are allocated in segments that are mapped by different selectors. The effect of this is that short-cut techniques used to locate WARP *SFTs* may need to be re-worked.

14.1.1.1 Finding Files from Handles

Open file system objects (files, named pipes, devices etc.) are represented by the SFT control block. The *SFT* contains three sections:

- Kernel data
- File system independent data
- File system dependent data

The kernel data section contains information to link the *SFT* to other system control blocks and to make the SFT usable by kernel APIs. Of principle interest in this section are flags, handle and pointer to the MFT and a chain pointer to other *SFTs* that represent other open instances of the same object. The kernel data is split into two discontiguous sections at each end of the *SFT*.

The file system independent data section contains information common to all FSDs needed to drive the file system. Of principle interest are the file attributes, open mode flags, opening process id and handle to the associated VPB.

The file system dependent data section is, as the name suggests a work area private to the *FSD* that manages the file system object.

Note: The `.D SFT` command formats the *SFT* always as if it is a FAT file. The information displayed in the file system dependent section may be misleading for non-FAT objects. The names of the fields formatted by `.D SFT` command are prefixed by *sfdFAT_* for the file system dependent data so make it clear which information to treat with circumspection. The kernel and file system independent data name are prefixed with *sf_* and *sfi_* respectively.

When a file system object is opened, DosOpen returns a handle that represents the open object for all subsequent file system manipulation by the process until the object is closed. This handle is unique only within process and is referred to as the JFN. In protect mode processes the *JFN* is a 16-bit entity. In VDMs, however, to be consistent with DOS the *JFN* is an 8-bit entity, which may be correlated to the *real JFN* through a table in the PDB. This is illustrated later.

Each open file system object is also known by a system-wide unique handle, the SFN. Once the *SFN* is known then the corresponding *SFT* may be located and thence all file system information relating to the object.

Each process is assigned by default a table of 20 words, which is indexed by the *JFN*. Each word of the *JFN_table* contains the corresponding *SFN* for the open file. The default *JFN_table* is imbedded within the PTDA. Prefixing the *JFN_table* is a double-word pointer (*JFN_ptable*) that points to this table. If the table is expanded (using *DosSetMaxFH*) then *JFN_ptable* is updated to point to the current *JFN_table*.

The key to finding information about open object in a given process is to locate *JFN_ptable* and *JFN_table*. Since both of these fields are part of the *PTDA* they may be refered to by name as symbols for the current (system) context only. For other contexts we may still use the *PTDA* symbols but in a relative fashion. The *PTDA* symbols are defined for the current process, which means that to use them successfully for another process, one must relocate them to the PTDA one wishes to reference. This is easily done by subtracting the label *PTDA_START* from the desired symbol, then adding the address of the *PTDA* one wishes to see. For example, to see the *jfn_table* field, enter

```
dw <ptda address>+jfn_table-ptda_start L2.
```

The relationships between the *JFN_table*, *PTDA* and the *SFT* is illustrated in the following diagram:

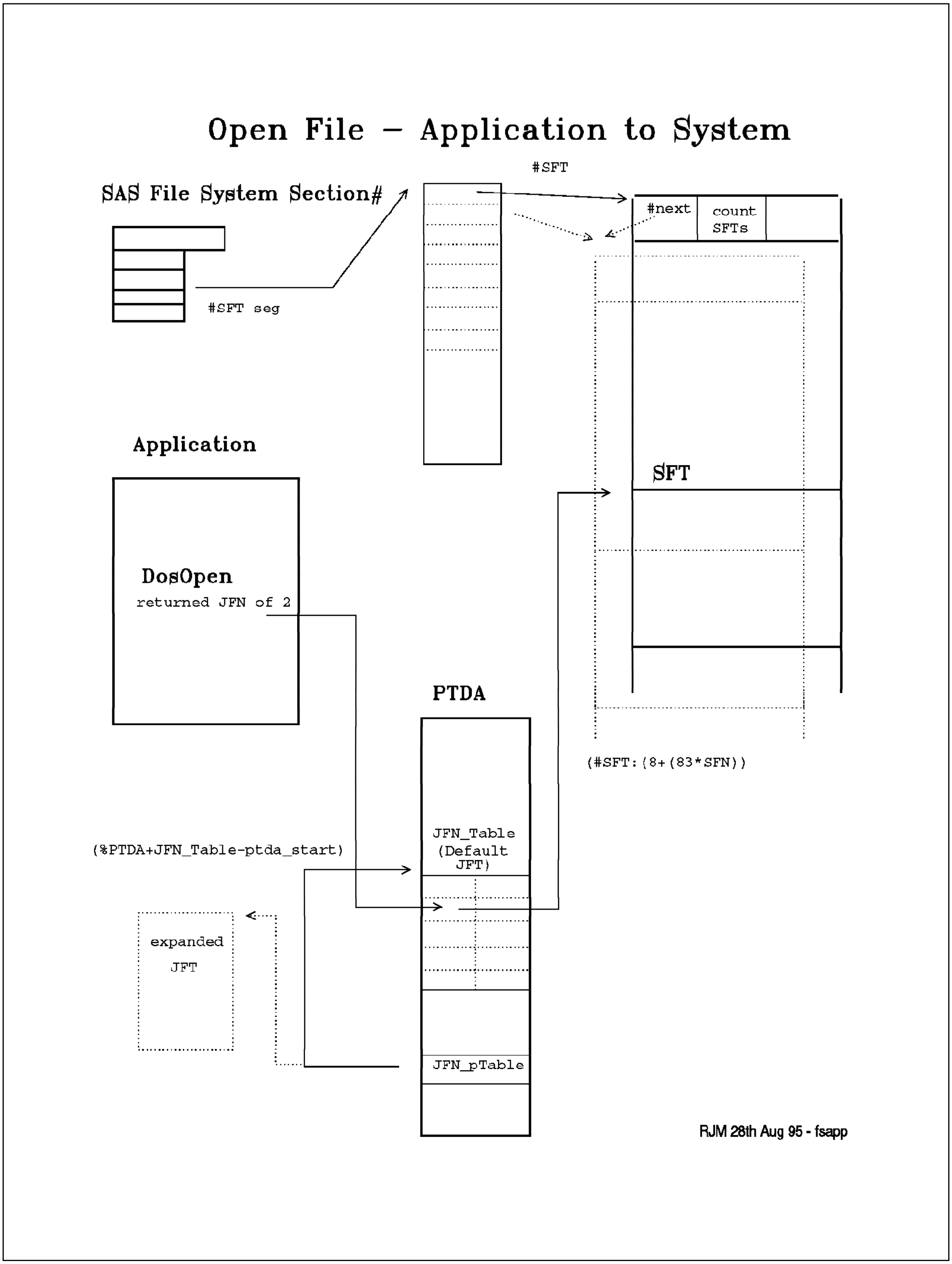


Figure 13. Open File, Application to System

In the examples that follow, we explore the relationships between each of the major file-system control blocks. These relationships are illustrated in the following diagrams.

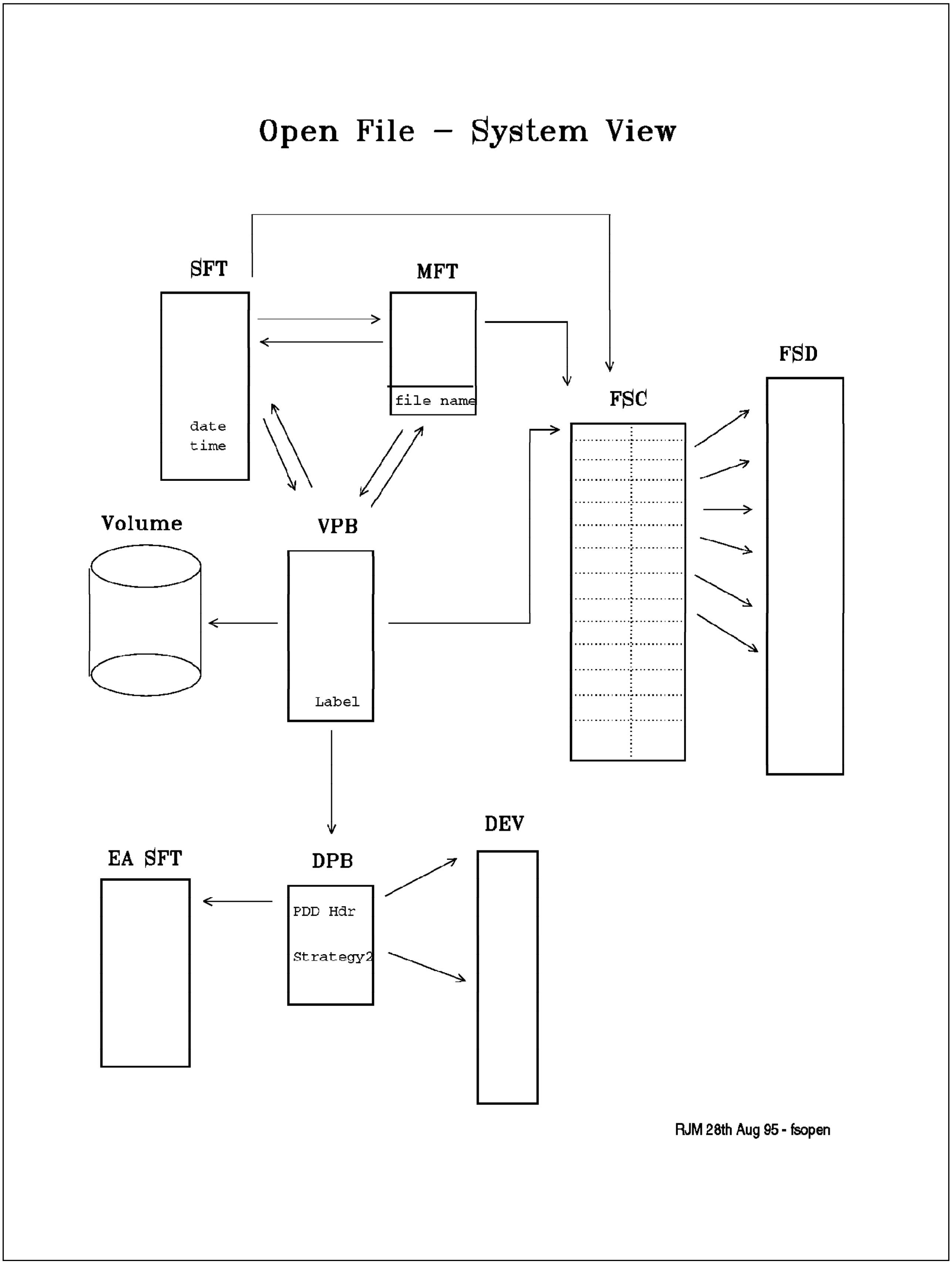


Figure 14. Open File, System View

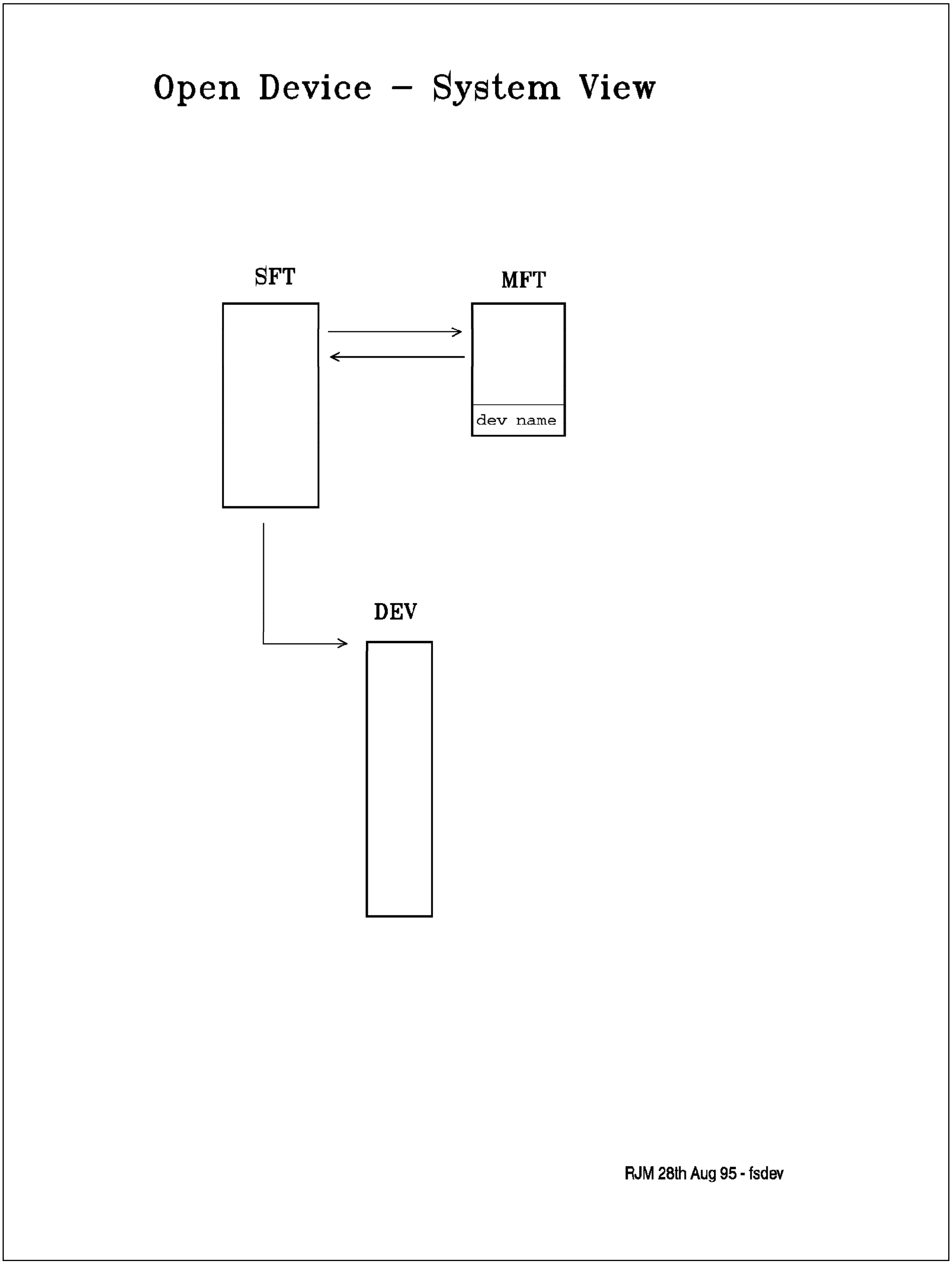


Figure 15. Open Device, System View

14.1.1.2 Finding Files from Handles - An Example

In the following example we choose to discover all the open file system objects in process 19, which happens to be running the IPFC compiler.

```
>>> List all the thread slots in the system to find IPFC
# .p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe3a000 ffe3c7d4 ffe3c61c 1e7c 00 *ager
 0002  0001 0000 0000 0002 blk 0200 7b7aa000 ffe3c7d4 7b9a8020 1f3c 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 7b7ac000 ffe3c7d4 7b9a81d8 1f50 00 *ctxh
 0004  0001 0000 0000 0004 blk 081f 7b7ae000 ffe3c7d4 7b9a8390 1f48 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 7b7b0000 ffe3c7d4 7b9a8548 1f20 00 *lazyw
 0006  0001 0000 0000 0006 blk 0800 7b7b2000 ffe3c7d4 7b9a8700 1f3c 00 *asyncr
 0009  0002 0000 0002 0001 blk 021f 7b7b8000 7b9c4020 7b9a8c28      00 LOGDAEM
 0008  0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
 000b  0003 0001 0003 0002 blk 0800 7b7bc000 7b9c484c 7b9a8f98      01 PMSHL32
 000c  0003 0001 0003 0003 blk 0800 7b7be000 7b9c484c 7b9a9150      01 PMSHL32
 000d  0003 0001 0003 0004 blk 0800 7b7c0000 7b9c484c 7b9a9308      01 PMSHL32
 000e  0003 0001 0003 0005 blk 0800 7b7c2000 7b9c484c 7b9a94c0      01 PMSHL32
 0007  0003 0001 0003 0006 blk 0200 7b7b4000 7b9c484c 7b9a88b8 1ecc 01 PMSHL32
 0011  0003 0001 0003 0007 blk 0200 7b7c8000 7b9c484c 7b9a99e8 1ecc 01 PMSHL32
 0012  0003 0001 0003 0008 blk 0200 7b7ca000 7b9c484c 7b9a9ba0      01 PMSHL32
 0013  0003 0001 0003 0009 blk 0200 7b7cc000 7b9c484c 7b9a9d58      01 PMSHL32
 0014  0003 0001 0003 000a blk 0800 7b7ce000 7b9c484c 7b9a9f10      01 PMSHL32
 0015  0003 0001 0003 000b blk 0800 7b7d0000 7b9c484c 7b9aa0c8      01 PMSHL32
 0016  0003 0001 0003 000c blk 0800 7b7d2000 7b9c484c 7b9aa280      01 PMSHL32
 0017  0003 0001 0003 000d blk 0804 7b7d4000 7b9c484c 7b9aa438 1ea8 01 PMSHL32
 0018  0003 0001 0003 000e rdy 0804 7b7d6000 7b9c484c 7b9aa5f0      01 PMSHL32
 0019  0003 0001 0003 000f blk 0500 7b7d8000 7b9c484c 7b9aa7a8      01 PMSHL32
 001a  0003 0001 0003 0010 rdy 0801 7b7da000 7b9c484c 7b9aa960 1bac 01 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001b  0003 0001 0003 0011 blk 0800 7b7dc000 7b9c484c 7b9aab18      01 PMSHL32
*001c# 0003 0001 0003 0012 run 0800 7b7de000 7b9c484c 7b9aacd0 1b8c 01 PMSHL32
 001d  0003 0001 0003 0013 blk 0200 7b7e0000 7b9c484c 7b9aae88      01 PMSHL32
 0023  0018 0003 0018 0001 rdy 061f 7b7ec000 7b9c7128 7b9ab8d8 1eb8 13 EPM
 0038  0018 0003 0018 0002 blk 0200 7b816000 7b9c7128 7b9adcf0 1ecc 13 EPM
 0037  0013 0003 0013 0001 blk 0200 7b814000 7b9c9a04 7b9adb38      19 IBMAVSD
 0033  0012 0003 0012 0001 blk 0200 7b80c000 7b9c89ac 7b9ad458 1eb8 17 PMDRAW
 0035  0012 0003 0012 0002 blk 0200 7b810000 7b9c89ac 7b9ad7c8 1eb8 17 PMDRAW
 0036  0012 0003 0012 0003 blk 0200 7b812000 7b9c89ac 7b9ad980      17 PMDRAW
 0034  0010 0003 0010 0001 blk 0400 7b80e000 7b9c91d8 7b9ad610 1ed4 12 CMD
 002e  000d 0003 000d 0001 blk 0200 7b802000 7b9c8180 7b9acbc0 1eb8 16 PULSE
 0030  000d 0003 000d 0002 rdy 0100 7b806000 7b9c8180 7b9acf30 1f28 16 PULSE
 002f  000d 0003 000d 0003 rdy 081f 7b804000 7b9c8180 7b9acd78 1f00 16 PULSE
 002d  000c 0003 000c 0001 blk 0200 7b800000 7b9c7954 7b9aca08 1eb8 15 DINFO
 0032  000c 0003 000c 0002 rdy 061f 7b80a000 7b9c7954 7b9ad2a0 1f00 15 DINFO
 002c  000b 0003 000b 0001 blk 0200 7b7fe000 7b9c58a4 7b9ac850 1eb8 14 MRFILE32
 0031  000b 0003 000b 0002 blk 0200 7b808000 7b9c58a4 7b9ad0e8 1ecc 14 MRFILE32
 0029  000a 0003 000a 0001 rdy 061f 7b7f8000 7b9c68fc 7b9ac328 1eb8 10 PMDIARY
 001f  0006 0003 0006 0001 rdy 062f 7b7e4000 7b9c60d0 7b9ab1f8 1eb8 11 PMSHL32
 0021  0006 0003 0006 0002 blk 0200 7b7e8000 7b9c60d0 7b9ab568      11 PMSHL32
 0022  0006 0003 0006 0003 blk 0200 7b7ea000 7b9c60d0 7b9ab720 1eb8 11 PMSHL32
 0020  0006 0003 0006 0004 blk 0200 7b7e6000 7b9c60d0 7b9ab3b0      11 PMSHL32
 001e  0006 0003 0006 0005 blk 0200 7b7e2000 7b9c60d0 7b9ab040 1ecc 11 PMSHL32
 0024  0006 0003 0006 0006 blk 0200 7b7ee000 7b9c60d0 7b9aba90      11 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0025  0006 0003 0006 0007 blk 0200 7b7f0000 7b9c60d0 7b9abc48      11 PMSHL32
```

```
 0026  0006 0003 0006 0008 blk 0200 7b7f2000 7b9c60d0 7b9abe00      11 PMSHL32
 0027  0006 0003 0006 0009 blk 0200 7b7f4000 7b9c60d0 7b9abfb8      11 PMSHL32
 0028  0006 0003 0006 000a blk 0200 7b7f6000 7b9c60d0 7b9ac170      11 PMSHL32
 002a  0006 0003 0006 000c blk 021f 7b7fa000 7b9c60d0 7b9ac4e0 1eac 11 PMSHL32
 002b  0006 0003 0006 000d blk 0200 7b7fc000 7b9c60d0 7b9ac698 1eb8 11 PMSHL32
 000a  0004 0003 0004 0001 blk 0800 7b7ba000 7b9c5078 7b9a8de0      00 HARDERR
 000f  0004 0003 0004 0002 blk 0800 7b7c4000 7b9c5078 7b9a9678      00 HARDERR
 0010  0004 0003 0004 0003 blk 0800 7b7c6000 7b9c5078 7b9a9830      00 HARDERR
 0039  0019 0010 0019 0001 rdy 061f 7b818000 7b9ca230 7b9adea8 1f0c 12 IPFC

>>> IPFC is running in slot 39, but this is not the current system
>>> slot so we have to refer to PTDA symbols relative to pPTDA
>>> First establish whether JFN_table has been expanded?

# dw %7b9ca230 + jfn_ptable-ptda_start l2
%7b9caa18  fd8a 0030

>>> No it hasn't - it's still based on selector 30 and therefore
>>> still imbedded in the PTDA at label JFN_table.
>>> Note: we can't display it as 30:fd8a since selector 30
>>> aliases the current system PTDA, hence:

# dw %7b9ca230 + jfn_table-ptda_start l14
%7b9ca7e6  0027 0027 0027 0074 002a 0072 0077 0068
%7b9ca7f6  0015 0041 0069 007f ffff ffff ffff ffff
%7b9ca806  ffff ffff ffff ffff

>>> These are the SFNs that correspond to JFNs 0000 through 0014.
>>> In fact the highest JFN currently open in this process is 000b
>>> which corresponds to SFN 007f

>>> Next we locate the STF.  From the SAS we look for the SFT selector:


# .a
--- SAS Base Section ---
                      SAS signature: SAS
           offset to tables section: 0016
       FLAT selector for kernel data: 0168
     offset to configuration section: 001E
     offset to device driver section: 0020
    offset to Virtual Memory section: 002C
          offset to Tasking section: 005C
              offset to RAS section: 006E
       offset to File System section: 0074
          offset to infoseg section: 0080
--- SAS Protected Modes Tables Section ---
                   selector for GDT: 0008
                   selector for LDT: 0000
                   selector for IDT: 0018
               selector for GDTPOOL: 0100
--- SAS Device Driver Section ---
     offset for the first bimodal dd: 0CB9
   offset for the first real mode dd: 0000
       sel for Drive Parameter Block: 04C8
        sel for ABIOS prot. mode CDA: 0000
         seg for ABIOS real mode CDA: 0000
                   selector for FSC: 00C8
--- SAS Task Section ---
```

```
          selector for current PTDA: 0030
  FLAT offset for process tree head: FFF10910
 FLAT address for TCB address array: FFF06BB6
      offset for current TCB number: FFDFFB5E
             offset for ThreadCount: FFDFFB62
--- SAS File System Section ---
                handle to MFT PTree: FE72CFBC
     selector for System File Table: 00C0
     sel. for Volume Parameter Bloc: 0788
   sel. for Current Directory Struc: 07B8
        selector for buffer segment: 00A8
--- SAS Information Segment Section ---
       selector for global info seg: 0428
   address of curtask local infoseg: 03C80000
      address of DOS task's infoseg: FFFFFFFF
         selector for Codepage Data: 07CB
--- SAS RAS Section ---
selector for System Trace Data Area: 04B0
 segment for System Trace Data Area: 04B0
        offset for trace event mask: 0B28
--- SAS Configuration Section ---
    offset for Device Config. Table: 0D50
--- SAS Virtual Memory Mgt. Section ---
       Flat offset of arena records: FFF13304
      Flat offset of object records: FFF1331C
     Flat offset of context records: FFF1330C
  Flat offset of kernel mte records: FFF0A891
     Flat offset of linked mte list: FFF07934
    Flat offset of page frame table: FFF11A70
    Flat offset of page range table: FFF111EC
    Flat offset of swap frame array: FFF03BAC
           Flat offset of Idle Head: FFF10090
           Flat offset of Free Head: FFF10080
          Flat offset of Heap Array: FFF11B78
     Flat offset of all mte records: FFF12E04

>>> We see this is assigned to selector c0.
>>> This is not quite the SFT but a table of selectors that point to
>>> each extent of the SFT. Each extent holds up to 500 STF entries.
>>> All the SFN's we're interested in are less than 500 so occupy the
>>> first extent. Note: we could have obtained the SFT selector from:

# ln GDT_SFT
138:000000c0 os2krnl DOSGDTDATA:GDT_SFT
#

>>> List the table of extents:

# dw c0:0
00c0:00000000  0438 0000 0000 0000 0000 0000 0000 0000
00c0:00000010  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000020  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000030  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000040  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000050  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000060  0000 0000 0000 0000 0000 0000 0000 0000
00c0:00000070  0000 0000 0000 0000 0000 0000 0000 0000

>>> Now list the first extent:
```

```
# dw 438:0
0438:00000000  0000 0000 0000 0000 0001 0000 0000 0001
0438:00000010  0000 0000 0800 c800 0000 0000 0000 0000
0438:00000020  7000 860e c6fe 6821 0008 0000 0000 0000
0438:00000030  0000 0000 0000 0000 0000 0000 0000 0000
0438:00000040  0000 0000 0000 0000 0000 8a30 007a 0000
0438:00000050  0000 0000 0000 a000 0000 1200 0000 0000
0438:00000060  0000 0000 0000 0000 e700 0110 6000 00ee
0438:00000070  0000 0000 0000 0000 0000 0000 0000 0000

>>> There is an 8 byte header to each extent. It followed by one or
>>> more 131 (hex 83) byte SFT entries. The first word of the header
>>> contains the selector for the next extent.  In this case there
>>> isn't one.

>>> To locate the SFT entry corresponding to SFN we use the formula
>>> 438:(8+(83*SFN))
>>> We can dump this out directly or by using the .D SFT command:
>>> Start by examining SFN 0077 (JFN 0006 for slot 39)

# .d sft 438:(8+(83*77))
      sf_ref_count: 0001                          sfi_mode: 20a2
        sf_usercnt: 0000                          sfi_hVPB: 0012
          reserved: 00                           sfi_ctime: 0000
       sf_flags(2): 0000:0000                    sfi_cdate: 0000
         sf_devptr: #0000:0000                   sfi_atime: 0000
            sf_FSC: #00c8:0008                   sfi_adate: 0000
          sf_chain: #0000:0000                   sfi_mtime: 6000
            sf_MFT: fe87ebf0                     sfi_mdate: 1b0b
sfdFAT_firFILEclus: 57e4                          sfi_size: 00000000
    sfdFAT_cluspos: 0ff8                      sfi_position: 00000000
    sfdFAT_lstclus: 0038                           sfi_UID: 0000
     sfdFAT_dirsec: 00002cad                       sfi_Pid: 0019
     sfdFAT_dirpos: 09                             sfi_PDB: 0000
       sfdFAT_name: FCLDLGP DLL                        sfi_selfsfn: 0077
   sfdFAT_EAHandle: 0000                        sfi_tstamp: 00
          sf_plock: 0000                       sfi_DOSattr: 20
      sf_NmPipeSfn: 0000
       sf_codepage: 0000

>>> Fully qualified file system names are maintained in the Master
>>> File Table entries. Lets check out the MFT for this SFT, which
>>> is pointed to by sf_MFT.
>>> Under the Kernel debugger we could use .D MFT to format an MFT
>>> entry. Under the dump formatter .D MFT does not work correctly:


# db %fe87ebf0
%fe87ebf0 4b 53 45 4d 01 02 00 00-00 00 00 00 00 00 00 00 KSEM............
%fe87ec00 00 00 ed 3c 38 04 00 00-00 00 4b 53 45 4d 01 02 ..m<8.....KSEM..
%fe87ec10 00 00 00 00 00 00 00 00-00 00 00 00 1c 0e 00 00 ................
%fe87ec20 6d 46 12 00 43 3a 5c 24-30 30 30 30 24 00 87 fe mF..C:\$0000$..~
%fe87ec30 14 00 9e ff 29 00 0b 00-dc eb 87 fe 08 43 83 fe ....)...\k.~.C.~
%fe87ec40 f0 eb 87 fe 48 00 9e ff-4b 53 45 4d 01 02 00 00 pk.~H...KSEM....
%fe87ec50 00 00 00 00 00 00 00 00-00 00 5e 3a 38 04 00 00 ..........^:8...
%fe87ec60 00 00 4b 53 45 4d 01 02-00 00 00 00 00 00 00 00 ..KSEM..........
```

```
>>> The file name is at MFT+34 in the ALLSTRICT kernel and 2a in the
>>> RETAIL kernel. There are two imbedded KSEMs, which only only
>>> contain the signature KSEM in the ALLSTRICT kernel, also the MFT
>>> contains the signature mF at +30 in the ALLSTRICT kernel.
>>> The first KSEM used for serialising read/single write access to
>>> the file. The second KSEM is used for updating the cluster map.

>>> These KSEMs can be formatted using .d KSEM

# .d ksem %fe87ebf0
Signature     : KSEM                    Nest: 0000
Type          : SHARE                Readers: 0000
Flags         : 01            PendingReaders: 0000
Owner         : 0000          PendingWriters: 0000
# .d ksem %fe87ebf0+1a
Signature     : KSEM                    Nest: 0000
Type          : SHARE                Readers: 0000
Flags         : 01            PendingReaders: 0000
Owner         : 0000          PendingWriters: 0000
#
>>> In this case they are unowned.

>>> The file name in the MFT does not agree with the sfdFAT_name.
>>> We suspect that this is not a FAT file. This can be verified by
>>> examining the file system control block entry for the FSD that's
>>> managing this file. The FSC entry address appears in the SFT at
>>> sf_FSC. In this case it is 00c8:0008.

# dw c8:8
00c8:00000008  0b68 0840 0b6c 0840 0000 0828 01fc 0828
00c8:00000018  0010 0828 05b4 0820 0570 0828 0580 0828
00c8:00000028  0634 0828 0640 0828 0e3c 0828 1120 0828
00c8:00000038  0834 0828 090c 0828 09f8 0820 1130 0828
00c8:00000048  1f24 0828 1f6e 0828 2122 0828 16e4 0828
00c8:00000058  1b10 0828 1b38 0828 1bec 0828 1dc8 0828
00c8:00000068  0c60 0820 0d70 0820 1f14 0828 215c 0828
00c8:00000078  22a0 0828 2294 0828 111c 0820 25fc 0828

>>> Each FSC enrty is a table of far16 pointers. The first points the
>>> The FSD attributes, the second to the name and the remainder are
>>> standard FSD entry points. (See OEMI IFS Documentation).
>>> the name of this FSD is....

# da 840:b6c
0840:00000b6c HPFS

>>> The word prefixing the file name in the MFT is the handle to the
>>> Volume Parameter Block (hVPB). This also appears in the SFT under
>>> sfi_hVPB. In this instance the hVPB is 0012.
>>> To format the VPB we need to obtain the selector for the VPB
>>> segment. N.B. this is not stored in the SAS under Volume Parameter
>>> Block. We have to locate this using:

# ln GDT_VPB
138:00000098 os2krnl DOSGDTDATA:GDT_VPB
#

>>> The hVPB is an offset into the VPB segment. Format a VPB
>>> using .D VPB
```

```
# .d vpb 98:12
        vpb_flink: 0000                vpdFAT_cluster_mask: 02
        vpb_blink: 008d               vpdFAT_cluster_shift: 00
    vpb_ref_count: 0057                   vpdFAT_first_FAT: 0000
 vpb_search_count: 0004                   vpdFAT_FAT_count: 00
 vpb_first_access: 00                  vpdFAT_root_entries: 0030
    vpb_signature: 444a                vpdFAT_first_sector: 06001100
     vpb_flags(2): 02:00                vpdFAT_max_cluster: 7d5c
          vpb_FSC: #00c8:0008             vpdFAT_FAT_size: b213
           vpi_ID: 25be2014             vpdFAT_dir_sector: fc04b800
         vpi_pDPB: #04c8:0038               vpdFAT_media: 0a
     vpi_cbSector: 0200                  vpdFAT_next_free: 00b2
       vpi_totsec: 00049020               vpdFAT_free_cnt: 04b8
       vpi_trksec: 0023              vpdFAT_FATentrysize: b2
        vpi_nhead: 000c                   vpdFAT_IDsector: 00000000
         vpi_pDCS: #0000:0000               vpdFAT_access: 0000
         vpi_pVCS: #0000:0000              vpdFAT_accwait: 0000
        vpi_drive: 02                        vpdFAT_pEASFT: #0000:0000
         vpi_unit: 02
         vpi_text: UNLABELED
        vpi_flags: 0003
#
>>> Two important pieces of information in the VPB: vpi_drive and
>>> vpi_text. The drive number is the logical drive, numbering from
>>> 0, Thus 02 is drive C:
>>> vpi_text is the volume label. in this case UNLABELED.
>>> The VPB contains a signature which when dumped as bytes appears as
>>> JD. Each VPB is 7b bytes, the first starts at +12. Each VPB can
>>> be dumped using the formula: 98:(12+(7b*entry))


# db 98:12+(7b*0) l7b
0098:00000012 00 00 8d 00 57 00 04 00-00 4a 44 02 00 08 00 c8 ....W....JD....H
0098:00000022 00 02 00 00 00 00 30 00-00 11 00 06 5c 7d 13 b2 ......0.....\}.2
0098:00000032 00 b8 04 fc 0a b2 00 b8-04 b2 00 00 00 00 0a 11 .8.|.2.8.2......
0098:00000042 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
0098:00000052 00 00 00 00 00 00 00 00-00 00 00 9a 7b 4c 5c 00 ............{L\.
0098:00000062 00 14 20 be 25 38 00 c8-04 00 02 20 90 04 00 23 .. >%8.H... ...#
0098:00000072 00 0c 00 55 4e 4c 41 42-45 4c 45 44 00 00 00 00 ...UNLABELED....
0098:00000082 00 00 00 00 00 00 00 02-02 03 00                ...........

>>> The word at +2 is a chain pointer offset to the next VPB. In this
>>> case  008d (=7b+12)

>>> We can also obtain a link to disk device driver information from
>>> the VPB via vpi_pDPB (the disk parameter block). Under the
>>> kernel debugger this may be formatted using .D DPB, but gives
>>> erroneous results under DF.

##.d dpb 4c8:38
        dpb_drive: 02
         dpb_unit: 02
  dpb_driver_addr: #0738:0000
     dpb_next_dpb: #04c8:0054
     dpb_cbSector: 0200
    dpb_first_FAT: 0001
  dpb_toggle_time: 00000000
         dpb_hVPB: 0012
```

```
        dpb_media: f8
        dpb_flags: 20
   dpb_drive_lock: 0000
    dpb_strategy2: #0740:135e

>>> From here we can locate the device driver header, but note: that
>>> the strategy2 routine address is located from the DPB.

##.d dev 738:0
          DevNext: 0588:0000
          DevAttr: 2880
         DevStrat: 0d7e
           DevInt: 0000
         NumUnits: 08
        DevProtCS: 0740
        DevProtDS: 0738
        DevRealCS: 0000
        DevRealDS: 0000

>>> Returning to the JFN_table for slot 36. We now examine JFN 0004
>>> which correlates with SFN 002a
>>> Dump the SFT as before:


# .d sft 438:(8+(83*2a))
      sf_ref_count: 000c                        sfi_mode: 0042
        sf_usercnt: 0000                        sfi_hVPB: 0000
          reserved: 00                         sfi_ctime: 0000
       sf_flags(2): 00c1:0000                  sfi_cdate: 0000
         sf_devptr: #04f8:0000                 sfi_atime: 0000
            sf_FSC: #00c8:ff40                 sfi_adate: 0000
          sf_chain: #0438:0f62                 sfi_mtime: 3c83
            sf_MFT: fe73ecfc                   sfi_mdate: 1d62
sfdFAT_firFILEclus: 0000                        sfi_size: 00000000
    sfdFAT_cluspos: 0000                    sfi_position: 00000000
    sfdFAT_lstclus: 0000                         sfi_UID: 0000
     sfdFAT_dirsec: 00000000                     sfi_Pid: 0003
     sfdFAT_dirpos: 00                           sfi_PDB: 0000
       sfdFAT_name: KBD$                      sfi_selfsfn: 002a
   sfdFAT_EAHandle: 0000                      sfi_tstamp: 00
          sf_plock: 0000                     sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000


>>> The first flag word is 00c1 = 0000 0000 1100 0001
>>>                                         .      ..
>>>                                         .      ..
>>>                                         Device ..
>>>                                                ..
>>>                                                .console input dev
>>>                                                .
>>>                                                not console output dev
>>> Note: the hVPB is 0000 but sf_devptr is not and points to the
>>> device driver header for KDB$ thus:

##.d dev 4f8:0
          DevNext: 04e8:0000
          DevAttr: c981
```

```
         DevStrat: 0000
           DevInt: 2a29
          DevName: KBD$
        DevProtCS: 0500
        DevProtDS: 04f8
        DevRealCS: 0000
        DevRealDS: 0000

>>> The MFT entry for this device is:
# db %fe73ecfc
%fe73ecfc 4b 53 45 4d 01 02 00 00-00 00 00 00 00 00 00 00 KSEM............
%fe73ed0c 00 00 33 1d 38 04 00 00-00 00 4b 53 45 4d 01 02 ..3.8.....KSEM..
%fe73ed1c 00 00 00 00 00 00 00 00-00 00 00 00 2a 00 00 00 ............*...
%fe73ed2c 6d 46 00 00 5c 44 45 56-5c 4b 42 44 24 00 73 fe mF..\DEV\KBD$.s~
%fe73ed3c 30 00 a6 ff 02 00 f9 00-48 2f 1c fd 60 ed 73 fe 0.&...y.H/.}`ms~
%fe73ed4c 94 ed 73 fe 88 b1 98 04-01 00 00 00 1f 00 00 00 .ms~.1..........
%fe73ed5c a4 dc 72 fe 08 42 4d 53-43 41 4c 4c 53 ec 73 fe $\r~.BMSCALLSls~
%fe73ed6c 18 00 9e ff 3c 00 0b 00-d4 ef 73 fe cc 14 86 fe ....<...Tos~L..~
#
>>> Note: the name of the KBD$ device driver known to the file system is
>>> \DEV\KDB$


>>> Finally, using the .D MFT command (under the KDB) this MFT formats as:


##.d mft %fe73ecfc
     mft_ksem:
Signature     : KSEM                      Nest: 0000
Type          : SHARE                  Readers: 0000
Flags         : 01              PendingReaders: 0000
Owner         : 0000            PendingWriters: 0000
     mft_lptr: 0000                  mft_sptr: 0438:1586
    mft_pCMap: 00000000              mft_serl: 002c       mft_signature: 466d
 mft_CMapKSem:
     mft_hvpb: 0000               mft_opflags: 0000           mft_flags: 0000
     mft_name: \DEV\KBD$

>>> Note: mft_sptr points to the associated SFT
```

Finding Files from Handles in a VDM: The situation in a VDM is slightly more complex, since it required the *JFN* to be compatible with DOS and therefore an 8-bit entity. Furthermore, the *JFN_table* in DOS is traditionally imbedded or chained from the DOS PDB (or PSP). For this reason a second level of indirection is employed.

The *JFN* returned from a VDM open indexes the byte array of virtual system file number (*VSFNs*). The *VSFN* ranges from 0 - 255. The high 47 (from 0xd0 though 0xfe) are used as real mode device handles. 0xff indicates an unused handle. When a VDM is created the initial *PDB* contains the default array of 20 handles at label *PDB_JFN_table* (**PDB** + 0x18). This current array's far segment address is at *PDB_JFN_pointer* (**PDB** + 0x34) and the size of the array is a word at *PDB_JFN_Length* (**PDB** + 0x32). The *PDB* lies on a paragraph boundary (16-byte boundary) and its segment address is saved in the *PTDA* at &currentpdb. Once again the usual precaution applies when referencing *PTDA* fields: their symbols are publicly defined for the current system context only. Therefore, to reference a *CurrentPDB* out-of-context must be done relative to the *PTDA* address for that context.

These points are illustrated in the following example:

```
##.p 46
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0046  001d 0007 001d 0001 blk 0200 7b732000 7b8c9a04 7b8af720 1f08 17 *vdm
##.s 46

>>> Slot 46 is a VDM but not the current context, so we locate the PDB
>>> relative to the PTDA (otherwise we could have just used
>>> dw currentpdb l1)

##dw %7b8c9a04+currentpdb-ptda_start l1
0030:0000fd16  0e01

>>> This is a segment address so use the & operator to display the
>>> PDB.

##db &e01:0
&0e01:00000000 cd 20 00 a0 00 9a f0 fe-1d f0 f5 01 99 0d 08 02 M . ..p˜.pu.....
&0e01:00000010 28 08 65 07 28 08 99 0d-d1 d1 d1 d0 d2 ff ff ff (.e.(...QQQPR...
&0e01:00000020 ff ff ff ff ff ff ff ff-ff ff ff ff d7 00 e4 03 ............W.d.
&0e01:00000030 11 0e 30 00 00 00 40 09-ff ff ff ff 00 00 00 00 ..0...@.........
&0e01:00000040 14 0b 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
&0e01:00000050 cd 21 cb 72 6e 6c 20 2d-20 4e 6f 74 00 20 20 20 M!Krnl - Not.
&0e01:00000060 20 20 20 20 20 20 20 20-00 00 00 00 00 20 20 20         .....
&0e01:00000070 20 20 20 20 20 20 20 20-00 00 00 00 e7 08 50 03         ....g.P.

>>> Word at PDB+0x32 is 0030. This is the number of file handles
>>> supported in this VDM. The default is 0014. So the PDB_JFN_table
>>> has been expanded.
>>> Far pointer at PDB+34 is &0940:0000. This is the current
>>> PDB_JFN_table address. (By default this would have pointed to
>>> PDB+0x18, but the table has been expanded.)

>>> Now dump the current PDB_JFN_table.


##db &940:0
&0940:00000000 d1 d1 d1 d0 d2 00 01 02-03 04 05 06 07 08 09 ff QQQPR...........
&0940:00000010 0b ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff ................
&0940:00000020 ff ff ff ff ff ff ff ff-ff ff ff ff ff ff ff ff ................
&0940:00000030 4d 00 00 08 00 00 00 00-00 00 00 00 00 00 00 00 M...............
&0940:00000040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
&0940:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
&0940:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
&0940:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

>>> JFNs 0 - 4 correspond to VSFNs d1, d1, d1, d0 and d2. Each of these
>>> is greater than 0xcf and therefore a real mode device handle
>>> and not managed by the protect mode file system.
>>> JFN 5 is the first open file in this VDM. It has VSFN 00. This
>>> may be used as an index into the protect mode JFN_table to
>>> find the corresponding SFT, MFT and file name. The technique from
>>> this point is the same as in the preceding section.
```

```
>>> Dump the JFN_table from the PTDA.

##dw %7b8c9a04 jfn_ptable-ptda_start l2
0030:0000ffbc  0000 1ea8

>>> We are no longer based on selector 30 so the JFN table has been
>>> expanded. Now dump the current table...

##dw #1ea8:0
1ea8:00000000  006a 0069 0075 008c 008b 005e 0089 0088
1ea8:00000010  008a 0097 ffff 008d ffff ffff ffff ffff
1ea8:00000020  ffff ffff ffff ffff ffff ffff ffff ffff
1ea8:00000030  ffff ffff ffff ffff ffff ffff ffff ffff
1ea8:00000040  ffff ffff ffff ffff ffff ffff ffff ffff
1ea8:00000050  ffff ffff ffff ffff ffff ffff ffff ffff
1ea8:00000060
Past end of segment: 1ea8:00000060

>>> VSFN 00 corresponds to SFN 006a. Now dump the SFT...

##.d sft 438:(8+(83*6a))
      sf_ref_count: 0001                       sfi_mode: 00a0
        sf_usercnt: 0000                       sfi_hVPB: 0012
          reserved: 00                        sfi_ctime: 0000
       sf_flags(2): 0000:0000                 sfi_cdate: 0000
         sf_devptr: #0000:0000                sfi_atime: 0000
            sf_FSC: #00c8:0008                sfi_adate: 0000
          sf_chain: #0000:0000                sfi_mtime: 0000
            sf_MFT: fe7c8b54                  sfi_mdate: 0000
sfdFAT_firFILEclus: 4d58                       sfi_size: 00000594
    sfTrap 13 (0DH) - General Protection Fault 0000 - In Debugger
eax=90000000 ebx=ffdd55ba ecx=00000000 edx=013c0000 esi=00006373 edi=ffff2940
eip=000054ae esp=00003b0a ebp=00003b14 iopl=0 rf -- -- nv up di pl zr na pe nc
cs=0120 ss=0128 ds=0128 es=0128 fs=0168 gs=0000  cr2=16ea2000  cr3=001d9000
0120:000054ae ff56fe         call    word ptr [bp-02]              ss:3b12=b23b

>>> Oops! the debugger had a problem.  Not to worry, he told us
>>> the MFT address, so dump that ... (this also appears at SFT+0x19)

>>> the SFT again ...
##dw  438:(8+(83*6a))
0438:00003646  0001 0000 0000 0000 0000 0000 0800 c800
0438:00003656  0000 0000 0000 0000 5400 7c8b 58fe 484d
0438:00003666  001f 0000 0000 0000 0000 0000 0000 0000
0438:00003676  0000 0000 0000 0000 0000 0000 0000 0000
0438:00003686  0000 7830 007a 0000 0000 0000 0000 a000
0438:00003696  0000 1200 0000 0000 0000 0000 0000 0000
0438:000036a6  9400 0005 9400 0005 0000 1d00 0100 6a0e
0438:000036b6  0000 0000 0000 0000 0020 0000 0000 0000

>>> MFT
##db  %fe7c8b54
%fe7c8b54 4b 53 45 4d 01 02 00 00-00 00 00 00 00 00 00 00 KSEM............
%fe7c8b64 00 00 46 36 38 04 00 00-00 00 4b 53 45 4d 01 02 ..F68.....KSEM..
%fe7c8b74 00 00 00 00 00 00 00 00-00 00 00 00 ba 0f 00 00 ............:...
%fe7c8b84 6d 46 12 00 43 3a 5c 4f-53 32 5c 4d 44 4f 53 5c mF..C:\OS2\MDOS\
%fe7c8b94 57 49 4e 4f 53 32 5c 53-59 53 54 45 4d 5c 4b 42 WINOS2\SYSTEM\KB
%fe7c8ba4 44 55 4b 2e 44 4c 4c 00-52 8b 7c fe 3c 00 7d ff DUK.DLL.R.|˜<.}.
```

```
%fe7c8bb4 00 00 00 00 00 00 00 00-f0 8b 7c fe 20 f7 8a 7b ........p.|˜ w.{
%fe7c8bc4 f0 f1 f9 78 00 04 00 00-48 4f 4f 4b 60 bf f9 ff pqyx....HOOK`?y.

>>> The file name is: C:\OS2\MDOS\WINOS2\SYSTEM\KBDUK.DLL
```

Finding Handles from File Names: File system names are recorded in the MFT control block. Each *MFT* has a handle, which is its linear address and a key which is the concatenation of the *hVPB* with the file name considered as a string of bytes. The *MFT* keys are managed in a Patricia Tree structure similar to that described by 'Knuth' in *The Art of computer Programming, Volume 3, Sorting and Searching Algorithms* However, note that the implementation of the *PTree* in OS/2 is slightly modified from the 'Knuth' treatment.

The SAS gives us the address of the header node for the *MFT PTree*. The header node points to the first *PTree* entry. Each entry comprises a bit index, a key length, a left pointer, a right pointer and an *MFT* handle. The bit-index is used to specify the bit in the key to be tested. If the bit 0 then the left pointer is taken, otherwise the right pointer is taken. When the selected pointer points back to the same *PTree* entry then the search stops and the required *MFT* is found from the *MFT* handle. The bit index count the bits of each byte of the key from left to right.

This technique is now illustrated in the following example:

```
>>> Who's got C:\OS2\HELP\HMHELP.HLP open?

>>> First look through VPBs to find hVPB for C:
>>> Find the VPB segment and chain through them starting with the
>>> first at offset 0x12.

# ln gdt_vpb
0138:00000098 OS2KRNL GDT_VPB
# .d vpb 98:12
        vpb_flink: 0000                vpdFAT_cluster_mask: 02
        vpb_blink: 008d               vpdFAT_cluster_shift: 00
    vpb_ref_count: 0057                   vpdFAT_first_FAT: 0000
 vpb_search_count: 0004                   vpdFAT_FAT_count: 00
 vpb_first_access: 00                  vpdFAT_root_entries: 0030
    vpb_signature: 444a                vpdFAT_first_sector: 06001100
     vpb_flags(2): 02:00                 vpdFAT_max_cluster: 7d5c
          vpb_FSC: #00c8:0008               vpdFAT_FAT_size: b213
           vpi_ID: 25be2014               vpdFAT_dir_sector: fc04b800
         vpi_pDPB: #04c8:0038                  vpdFAT_media: 0a
     vpi_cbSector: 0200                    vpdFAT_next_free: 00b2
       vpi_totsec: 00049020                 vpdFAT_free_cnt: 04b8
       vpi_trksec: 0023                vpdFAT_FATentrysize: b2
        vpi_nhead: 000c                     vpdFAT_IDsector: 00000000
         vpi_pDCS: #0000:0000                 vpdFAT_access: 0000
         vpi_pVCS: #0000:0000                vpdFAT_accwait: 0000
        vpi_drive: 02                         vpdFAT_pEASFT: #0000:0000
         vpi_unit: 02
         vpi_text: UNLABELED
         vpi_flags: 0003
```

```
>>> We get lucky the first time. This VPD is for drive 2, that is C:
>>> So the hVPB=0012 (i.e the offset into the VPD segment).

>>> We now need to form the MFT key for the file name we wish to
>>> look up. So convert the file name to ASCII and concatenate to the
>>> hVPB (as a byte pair, that is, reversed)
>>>       C  :  \  O  S  2  \  H  E  L  P  \  H  M  H  E  L  P  .  H  L  P
>>> 12 00 43 3a 5c 4f 53 32 5c 48 45 4c 50 5c 48 4d 48 45 4c 50 2e 48-4c 50

>>> Locate the MFT PTree head from the SAS - in a dump use .A
>>> otherwise unravel the SAS from selector 70

# .a
--- SAS Base Section ---
                      SAS signature: SAS
           offset to tables section: 0016
      FLAT selector for kernel data: 0168
    offset to configuration section: 001E
    offset to device driver section: 0020
   offset to Virtual Memory section: 002C
          offset to Tasking section: 005C
              offset to RAS section: 006E
      offset to File System section: 0074
          offset to infoseg section: 0080
--- SAS Protected Modes Tables Section ---
                   selector for GDT: 0008
                   selector for LDT: 0000
                   selector for IDT: 0018
               selector for GDTPOOL: 0100
--- SAS Device Driver Section ---
    offset for the first bimodal dd: 0CB9
  offset for the first real mode dd: 0000
      sel for Drive Parameter Block: 04C8
       sel for ABIOS prot. mode CDA: 0000
        seg for ABIOS real mode CDA: 0000
                   selector for FSC: 00C8
--- SAS Task Section ---
          selector for current PTDA: 0030
  FLAT offset for process tree head: FFF10910
 FLAT address for TCB address array: FFF06BB6
      offset for current TCB number: FFDFFB5E
             offset for ThreadCount: FFDFFB62
--- SAS File System Section ---
                handle to MFT PTree: FE72CFBC
     selector for System File Table: 00C0
     sel. for Volume Parameter Bloc: 0788
   sel. for Current Directory Struc: 07B8
        selector for buffer segment: 00A8
--- SAS Information Segment Section ---
       selector for global info seg: 0428
   address of curtask local infoseg: 03C80000
      address of DOS task's infoseg: FFFFFFFF
         selector for Codepage Data: 07CB
--- SAS RAS Section ---
selector for System Trace Data Area: 04B0
 segment for System Trace Data Area: 04B0
        offset for trace event mask: 0B28
--- SAS Configuration Section ---
```

```
    offset for Device Config. Table: 0D50
--- SAS Virtual Memory Mgt. Section ---
       Flat offset of arena records: FFF13304
      Flat offset of object records: FFF1331C
     Flat offset of context records: FFF1330C
  Flat offset of kernel mte records: FFF0A891
     Flat offset of linked mte list: FFF07934
    Flat offset of page frame table: FFF11A70
    Flat offset of page range table: FFF111EC
    Flat offset of swap frame array: FFF03BAC
           Flat offset of Idle Head: FFF10090
           Flat offset of Free Head: FFF10080
          Flat offset of Heap Array: FFF11B78
     Flat offset of all mte records: FFF12E04

>>> MFT Ptree is at %fe72cfbc
>>> each entry including the header has the following format:
>>> +0 W Bit index
>>> +2 W key length
>>> +4 D left link
>>> +8 D right link
>>> +c D MFT handle (MFT pointer)

>>> dump the header and the first entry pointed to by the left link
# dd %FE72CFBC l4
%fe72cfbc  ffffffff fe867f10 fe72cfbc 00000000
# dd %FE867f10 l4
%fe867f10  00100000 fe861454 fe861470 fe721a04
>>>        ----.... -------- -------- --------
>>>         Kl  BI    left     right    MFT
>>>
>>> Note the word reversal of the Bit index and the Key length because
>>> we dumped double-words.
>>> BI tells us to test bit 0 of the key (numbering from the left
>>> starting with 0). 12 00 .. .. .. = 0001 0010 0000 0000 .... ....
>>> Bit zero is 0 so take the left link.

# dd %FE861454 l4
%fe861454  00100001 fe73d194 fe845370 fe72196c

>>> Now test bit 1. This is still zero. Again take the left link.

# dd %FE73d194 l4
%fe73d194  00190003 fe72cf3c fe87ec34 fe72dea4

>>> Now test bit 3. This is 1 so take the right link.

# dd %FE87ec34 l4
%fe87ec34  000b0029 fe87ebdc fe834308 fe87ebf0

>>> Now test bit 29.  ..... 4f .... = 0100 1111
>>> This is 1 so take the right link.

# dd %FE834308 l4
%fe834308  0019002b fe869f30 fe834254 fe834274

>>> Test bit 2b. This is 0. Turn left.
```

```
# dd %FE869f30 l4
%fe869f30  0017002c fe885ac4 fe87ec90 fe869ee0

>>> Test bit 2c. This is 1. Turn right.

# dd %FE87ec90 l4
%fe87ec90  000f002d fe87ec90 fe834ac8 fe87ec48

>>> Test bit 2d. This is 1. Turn right.

# dd %FE834ac8 l4
%fe834ac8  001a0044 fe845ef8 fe724fe4 fe845c0c

>>> Test bit 44. .....5c.... = 0101 1100.
>>> This is 1. Turn right.

# dd %FE724fe4 l4
%fe724fe4  001b004b fe801414 fe862de8 fe722fac

>>> Test bit 4b. .....48..... = 0100 1000
>>> This is 0. Turn left.

# dd %FE801414 l4
%fe801414  0017004c fe801d90 fe7cef84 fe7dffb0

>>> Test bit 4c. This is 1. Turn right.

# dd %FE7cef84 l4
%fe7cef84  00180073 fe7cef84 fe801414 fe7cef30

>>> Test bit 73. .....48.... = 0100 1000
>>> This is zero and the left link points to the same node.
>>> Therefore the search ends and we should have found the MFT
>>> for our file name. Dump the MFT to check .....


# db %FE7cef30 l50
%fe7cef30 4b 53 45 4d 01 02 00 00-00 00 00 00 00 00 00 00 KSEM............
%fe7cef40 00 00 28 31 38 04 00 00-00 00 4b 53 45 4d 01 02 ..(18.....KSEM..
%fe7cef50 00 00 00 00 00 00 00 00-00 00 00 00 69 03 00 00 ............i...
%fe7cef60 6d 46 12 00 43 3a 5c 4f-53 32 5c 48 45 4c 50 5c mF..C:\OS2\HELP\
%fe7cef70 48 4d 48 45 4c 50 2e 48-4c 50 00 00 16 e6 7c fe HMHELP.HLP...f|˜

>>> The MFT + 0x22 points is the SFT segment's offset. So dump the
>>> SFT ....

# ln gdt_sft
0138:000000c0 OS2KRNL GDT_SFT
dw c0:0l1
#00c0:00000000 0438

# .d sft 438:3128
      sf_ref_count: 0001                    sfi_mode: 00a0
        sf_usercnt: 0000                    sfi_hVPB: 0012
          reserved: 00                     sfi_ctime: 0000
       sf_flags(2): 0000:0000              sfi_cdate: 0000
         sf_devptr: #0000:0000             sfi_atime: 0000
            sf_FSC: #00c8:0008             sfi_adate: 0000
          sf_chain: #0438:33b7             sfi_mtime: 0000
```

```
          sf_MFT: fe7cef30                    sfi_mdate: 0000
sfdFAT_firFILEclus: 3344                       sfi_size: 00007058
    sfdFAT_cluspos: 0f10                   sfi_position: 00000a90
    sfdFAT_lstclus: 0000                        sfi_UID: 0000
     sfdFAT_dirsec: 00000000                    sfi_Pid: 0012
     sfdFAT_dirpos: 00                          sfi_PDB: 0000
       sfdFAT_name:                       sfi_selfsfn: 0060
   sfdFAT_EAHandle: 0000                     sfi_tstamp: 00
          sf_plock: 0000                    sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000

>>> sfi_Pid tells us Pid 12 has opened this file. But the
>>> sf_chain is not zero, so other processes have also opened
>>> this file. Follow the sf_chain .....

# .d sft 438:33b7
      sf_ref_count: 0001                       sfi_mode: 00a0
        sf_usercnt: 0000                       sfi_hVPB: 0012
          reserved: 00                        sfi_ctime: 0000
       sf_flags(2): 0000:0000                 sfi_cdate: 0000
         sf_devptr: #0000:0000                sfi_atime: 0000
            sf_FSC: #00c8:0008                sfi_adate: 0000
          sf_chain: #0438:2d10                sfi_mtime: 0000
            sf_MFT: fe7cef30                  sfi_mdate: 0000
sfdFAT_firFILEclus: 284a                       sfi_size: 00007058
    sfdFAT_cluspos: 0f10                   sfi_position: 00000a90
    sfdFAT_lstclus: 0000                        sfi_UID: 0000
     sfdFAT_dirsec: 00000000                    sfi_Pid: 000c
     sfdFAT_dirpos: 00                          sfi_PDB: 0000
       sfdFAT_name:                       sfi_selfsfn: 0065
   sfdFAT_EAHandle: 0000                     sfi_tstamp: 00
          sf_plock: 0000                    sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000


# .d sft 438:2d10
      sf_ref_count: 0001                       sfi_mode: 00a0
        sf_usercnt: 0000                       sfi_hVPB: 0012
          reserved: 00                        sfi_ctime: 0000
       sf_flags(2): 0000:0000                 sfi_cdate: 0000
         sf_devptr: #0000:0000                sfi_atime: 0000
            sf_FSC: #00c8:0008                sfi_adate: 0000
          sf_chain: #0438:2e16                sfi_mtime: 0000
            sf_MFT: fe7cef30                  sfi_mdate: 0000
sfdFAT_firFILEclus: 1986                       sfi_size: 00007058
    sfdFAT_cluspos: 0f10                   sfi_position: 00000a90
    sfdFAT_lstclus: 0000                        sfi_UID: 0000
     sfdFAT_dirsec: 00000000                    sfi_Pid: 000b
     sfdFAT_dirpos: 00                          sfi_PDB: 0000
       sfdFAT_name:                       sfi_selfsfn: 0058
   sfdFAT_EAHandle: 0000                     sfi_tstamp: 00
          sf_plock: 0000                    sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000


# .d sft 438:2e16
```

```
      sf_ref_count: 0001                      sfi_mode: 00a0
        sf_usercnt: 0000                      sfi_hVPB: 0012
          reserved: 00                       sfi_ctime: 0000
       sf_flags(2): 0000:0000                sfi_cdate: 0000
         sf_devptr: #0000:0000               sfi_atime: 0000
            sf_FSC: #00c8:0008               sfi_adate: 0000
          sf_chain: #0000:0000               sfi_mtime: 3ca2
            sf_MFT: fe7cef30                 sfi_mdate: 1d62
sfdFAT_firFILEclus: 050c                      sfi_size: 00007058
    sfdFAT_cluspos: 0f10                  sfi_position: 00000a90
    sfdFAT_lstclus: 0000                       sfi_UID: 0000
     sfdFAT_dirsec: 00000000                   sfi_Pid: 000d
     sfdFAT_dirpos: 00                         sfi_PDB: 0000
       sfdFAT_name: SINGLEQ$                    sfi_selfsfn: 005a
   sfdFAT_EAHandle: 0000                    sfi_tstamp: 00
          sf_plock: 0000                   sfi_DOSattr: 00
      sf_NmPipeSfn: 0000
       sf_codepage: 0000

>>> In all, PIDs 0012, 000c, 000b and 000d have opened
>>> C:\OS2\HELP\HMHELP.HLP
```

14.1.1.3 The Record Lock Record

In this example we investigate the RLR and how it records a locked range within a file. We will also see how the BlockID of a thread waiting for access to a locked file range directly leads to discovery of the RLR.

We introduce the *RLR* by showing its relationship to other file-system control blocks in the following diagram. This depicts the following situation:

- Three processes have opened the same file, in the order process 1, 2, then 3.
- The *MFT* heads the chain of *SFTs*, each representing an open instance of the same file. The *MFT* points to the most recent *SFT* open instance.
- Process 1 and process 2 have each locked a range within the same file. *RLRs* 1 and 2 correspond to process 1 and 2.
- The *MFT* heads the chain of *RLRs* starting with the most recent. The pointer from the *MFT* (*lptr*) is the offset within the *RLR* segment.

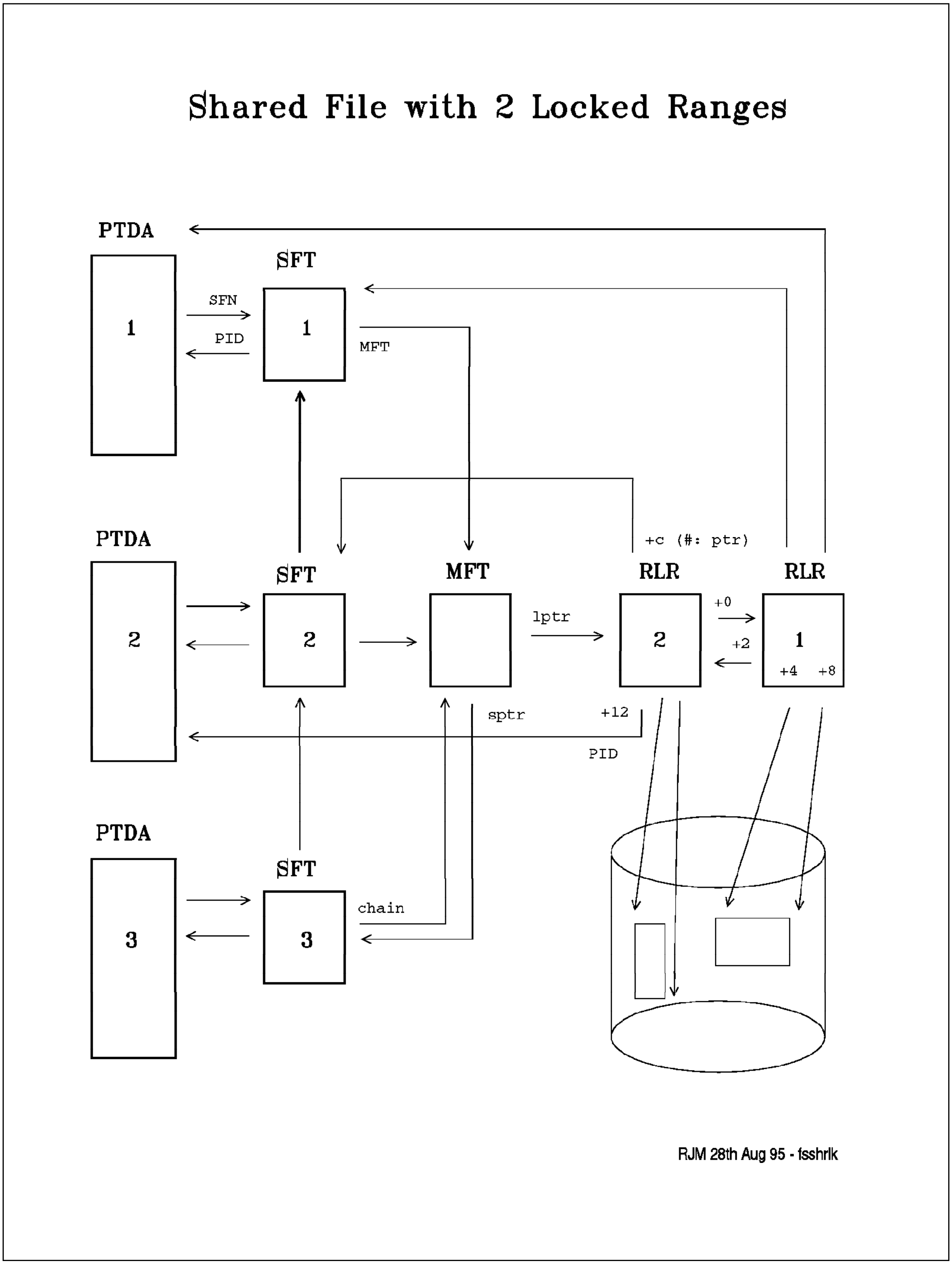


Figure 16. Shared File with Two Locked Ranges

```
>>> Problem: Program PAIN running in Slot 4d is hung with a blank
>>> screen. Everything else in the system seems OK. Mouse moves, we
>>> can change focus and so on..

>>> Lets take a look at slot 4d.

# .p 4d
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 004d# 001c 001b 001c 0001 blk 0300 ab80f000 ab99e220 ab980620 1e60 1d PAIN

>>> Blocked!

>>> We can approach this two ways:

>>> 1) Take a look at what the application did

>>> 2) Take a look at the BlockID and try see how far the system got


>>> Looking at the application we examine its registers and determine
>>> what API it called to cause it to block.

# .s 4d
Current slot number: 004d

# .r
eax=00023305 ebx=00000000 ecx=00000006 edx=0002004f esi=00000000 edi=00000000
eip=00010181 esp=0002337c ebp=000233fc iopl=2 -- -- -- nv up ei pl zr na pe nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001d9000
005b:00010181 83c414         add     esp,+14

# u %eip-10

%00010171 e8508d45e0         call    %e0468ec6
%00010176 50                 push    eax
%00010177 ff75f8             push    dword ptr [ebp-08]
%0001017a b005               mov     al,05
%0001017c e8c762f81b         call    %1bf96448
%00010181 83c414             add     esp,+14
%00010184 0bc0               or      eax,eax
%00010186 7404               jz      %0001018c
%00010188 ebde               jmp     %00010168
%0001018a 8bc0               mov     eax,eax
%0001018c b858000200         mov     eax,00020058
%00010191 e8fa000000         call    %00010290
# ln  %1bf96448

%1bf96448 DOSCALL1 DOS32SETFILELOCKS

>>> The last call was to DosSetFileLocks and we haven't returned. If
>>> we want any more information we have to analyze the BlockID.


# .pb#

 Slot  Sta BlockID  Name     Type        Addr        Symbol
```

```
 004d# blk 00b80029 PAIN
# .m 0b8:29

*har     par      cpg        va    flg next prev link hash hob   hal
 00dd %feaf0308 00000010 %aa7a3000 129 00dc 00de 0000 00cb 00ea 0000   sel=00b8
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00ea  00dd 0000  0124 ff47 0000  0000 00  00 00 00 fsreclok

>>> Seems to be blocked on a File System Record Lock (RLR).
>>> This implies that someone else has already locked a
>>> conflicting record range.

>>> We need to dump the RLR and locate the System File Table Entry
>>> associated with it. The BlockID is the address of the RLR.


# dw 0b8:29

00b8:00000029  0000 0000 0020 0000 002f 0000 526b 00d0
00b8:00000039  0000 0018 0000 0002 0000 2000 0000 2f00
00b8:00000049  0000 2000 9806 29ab 1c00 0300 0000 006e
00b8:00000059  0000 0000 0000 0000 0000 0000 0000 0000
00b8:00000069  0000 0000 8500 0000 0000 0000 0000 0000
00b8:00000079  0000 0000 0000 0000 0000 0000 009c 0000
00b8:00000089  0000 0000 0000 0000 0000 0000 0000 0000
00b8:00000099  0000 b300 0000 0000 0000 0000 0000 0000


>>> RLR+c is a far16 pointer to the associated System File Table entry
>>> (SFT).

>>> RLR+4 and RLR+8 are the range of bytes locked. Offset +20 - +2f
>>> has been locked.

>>> We now format the SFT for the process that locked this range:


# .d sft d0:526b

      sf_ref_count: 0001                      sfi_mode: 0042
        sf_usercnt: 0000                      sfi_hVPB: 04e0
          reserved: 00                       sfi_ctime: 0000
       sf_flags(2): 0040:0000                sfi_cdate: 0000
         sf_devptr: #0000:04e0               sfi_atime: 0000
            sf_FSC: #0000:ff40               sfi_adate: 0000
          sf_chain: #0000:0000               sfi_mtime: 5df6
            sf_MFT: fe87ca0c                 sfi_mdate: 1f3a
sfdFAT_firFILEclus: 0197                      sfi_size: 00000061
    sfdFAT_cluspos: 0000                  sfi_position: 00000030
    sfdFAT_lstclus: 0197                       sfi_UID: 0000
     sfdFAT_dirsec: 0000009f                   sfi_Pid: 0018
     sfdFAT_dirpos: 0a                         sfi_PDB: 0000
       sfdFAT_name: VIN                            sfi_selfsfn: 00a1
   sfdFAT_EAHandle: 0000                     sfi_tstamp: 00
          sf_plock: 0000                    sfi_DOSattr: 20
      sf_NmPipeSfn: 0000
       sf_codepage: 0000
```

```
>>> The SFT contains a pointer to the MFT, which contains the fully
>>> qualified file name. If the file is FAT then the short name in the
>>> SFT is also meaningful.

# .d mft % fe87ca0c

     mft_ksem:
Signature     : KSEM                        Nest: 0000
Type          : SHARE                    Readers: 0000
Flags         : 01                PendingReaders: 0000
Owner         : 0000              PendingWriters: 0000
     mft_lptr: 0029                     mft_sptr: 00d0:5600
    mft_pCMap: 00000000                 mft_serl: 128f
 mft_CMapKSem:
     mft_hvpb: 466d                  mft_opflags: 0000          mft_flags: 0000
     mft_name: A:\LAB19\VIN

>>> So the locked File is a:\lab19\vin


>>> The SFT also contains the Pid of the process that opened, and in
>>> this case locked, this file - Pid 18


# .p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe3a000 ffe3ca00 ffe3c800 1e7c 00 *ager
 0002  0001 0000 0000 0002 blk 0200 ab779000 ffe3ca00 ab977020 1f3c 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 ab77b000 ffe3ca00 ab977220 1f50 00 *ctxh
 0004  0001 0000 0000 0004 blk 081f ab77d000 ffe3ca00 ab977420 1f48 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 ab77f000 ffe3ca00 ab977620 1f20 00 *lazyw
 0006  0001 0000 0000 0006 blk 0800 ab781000 ffe3ca00 ab977820 1f3c 00 *asyncr
 0009  0002 0000 0002 0001 rdy 0804 ab787000 ab997020 ab977e20 1c88 00 CNTRL
 0008  0002 0000 0002 0002 blk 0804 ab785000 ab997020 ab977c20      00 CNTRL
 000b  0002 0000 0002 0003 blk 0804 ab78b000 ab997020 ab978220      00 CNTRL
 000c  0002 0000 0002 0004 rdy 0804 ab78d000 ab997020 ab978420 1c9c 00 CNTRL
 000a  0003 0000 0003 0001 blk 0800 ab789000 ab997620 ab978020      00 DOSCTL
 000d  0004 0001 0004 0001 rdy 0500 ab78f000 ab997c20 ab978620 1ed0 01 PMSHL32
 000f  0004 0001 0004 0002 blk 0800 ab793000 ab997c20 ab978a20 1ed4 01 PMSHL32
 0010  0004 0001 0004 0003 blk 0800 ab795000 ab997c20 ab978c20      01 PMSHL32
 0011  0004 0001 0004 0004 blk 0800 ab797000 ab997c20 ab978e20      01 PMSHL32
 0012  0004 0001 0004 0005 blk 0800 ab799000 ab997c20 ab979020      01 PMSHL32
 0015  0004 0001 0004 0006 blk 0200 ab79f000 ab997c20 ab979620 1edc 01 PMSHL32
 0016  0004 0001 0004 0007 blk 0200 ab7a1000 ab997c20 ab979820 1edc 01 PMSHL32
 0017  0004 0001 0004 0008 blk 0200 ab7a3000 ab997c20 ab979a20      01 PMSHL32
 0007  0004 0001 0004 0009 blk 0500 ab783000 ab997c20 ab977a20      01 PMSHL32
 0018  0004 0001 0004 000a blk 0800 ab7a5000 ab997c20 ab979c20      01 PMSHL32
 0019  0004 0001 0004 000b blk 0800 ab7a7000 ab997c20 ab979e20 1eb8 01 PMSHL32
 001a  0004 0001 0004 000c blk 0800 ab7a9000 ab997c20 ab97a020 1eb8 01 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001b  0004 0001 0004 000d blk 0804 ab7ab000 ab997c20 ab97a220 1ea8 01 PMSHL32
 001c  0004 0001 0004 000e blk 0804 ab7ad000 ab997c20 ab97a420 1eb0 01 PMSHL32
 001d  0004 0001 0004 000f blk 0500 ab7af000 ab997c20 ab97a620 1ea8 01 PMSHL32
 001e  0004 0001 0004 0010 blk 0801 ab7b1000 ab997c20 ab97a820 1bac 01 PMSHL32
 001f  0004 0001 0004 0011 blk 0801 ab7b3000 ab997c20 ab97aa20      01 PMSHL32
 0020  0004 0001 0004 0012 blk 0801 ab7b5000 ab997c20 ab97ac20      01 PMSHL32
 0021  0004 0001 0004 0013 blk 0800 ab7b7000 ab997c20 ab97ae20      01 PMSHL32
 0022  0004 0001 0004 0014 blk 0800 ab7b9000 ab997c20 ab97b020 1b80 01 PMSHL32
 0024  0004 0001 0004 0015 blk 0200 ab7bd000 ab997c20 ab97b420 1ed0 01 PMSHL32
```

```
 0030  0004 0001 0004 0016 blk 0800 ab7d5000 ab997c20 ab97cc20 1eac 01 PMSHL32
 004c  001b 0004 001b 0001 blk 0400 ab80d000 ab99dc20 ab980420 1ed4 1d CMD
 004b  001a 0004 001a 0001 blk 0200 ab80b000 ab99d620 ab980220 1eb8 1c CMD
 004a  0019 0004 0019 0001 blk 0200 ab809000 ab99d020 ab980020 1eb8 14 CMD
 0048  0017 0004 0017 0001 blk 0200 ab805000 ab99a620 ab97fc20 1ed4 04 CMD
 0047  0014 0004 0014 0001 blk 0200 ab803000 ab99c420 ab97fa20 1eb8 11 CMD
 003d  0012 0004 0012 0001 blk 0200 ab7ef000 ab99be20 ab97e620 1ed0 1a IBMAVSD
 0046  0011 0004 0011 0001 blk 0200 ab801000 ab99b820 ab97f820 1ed0 19 FPWMON
 003a  0010 0004 0010 0001 blk 0200 ab7e9000 ab99b220 ab97e020 1ed0 18 PMFAX
 0041  0010 0004 0010 0002 blk 0800 ab7f7000 ab99b220 ab97ee20 1edc 18 PMFAX
 0043  0010 0004 0010 0003 blk 0500 ab7fb000 ab99b220 ab97f220      18 PMFAX
 0045  0010 0004 0010 0005 blk 0500 ab7ff000 ab99b220 ab97f620 1d24 18 PMFAX
 0039  000f 0004 000f 0001 blk 0200 ab7e7000 ab99ac20 ab97de20 1ed0 17 FPWPIMX
 0040  000f 0004 000f 0002 blk 0200 ab7f5000 ab99ac20 ab97ec20 1ed0 17 FPWPIMX
 0042  000f 0004 000f 0003 blk 0200 ab7f9000 ab99ac20 ab97f020 1ed0 17 FPWPIMX
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0038  000e 0004 000e 0001 blk 0200 ab7e5000 ab99a020 ab97dc20 1ed0 16 DINFO
 003f  000e 0004 000e 0002 blk 0500 ab7f3000 ab99a020 ab97ea20 1f00 16 DINFO
 0037  000d 0004 000d 0001 blk 0200 ab7e3000 ab999a20 ab97da20 1ed0 15 MRFILE32
 003e  000d 0004 000d 0002 blk 0200 ab7f1000 ab999a20 ab97e820      15 MRFILE32
 0033  000c 0004 000c 0001 blk 0200 ab7db000 ab998e20 ab97d220 1ed0 13 PULSE
*003b  000c 0004 000c 0002 run 0100 ab7eb000 ab998e20 ab97e220 1f28 13 PULSE
 003c  000c 0004 000c 0003 blk 081f ab7ed000 ab998e20 ab97e420 1f00 13 PULSE
 0026  0008 0004 0008 0001 blk 0500 ab7c1000 ab999420 ab97b820 1ed0 12 PMSHL32
 002d  0008 0004 0008 0002 blk 0200 ab7cf000 ab999420 ab97c620 1edc 12 PMSHL32
 002e  0008 0004 0008 0003 blk 0200 ab7d1000 ab999420 ab97c820      12 PMSHL32
 002f  0008 0004 0008 0004 blk 0200 ab7d3000 ab999420 ab97ca20 1ed0 12 PMSHL32
 0028  0008 0004 0008 0005 blk 0200 ab7c5000 ab999420 ab97bc20      12 PMSHL32
 0025  0008 0004 0008 0006 blk 0200 ab7bf000 ab999420 ab97b620 1edc 12 PMSHL32
 002c  0008 0004 0008 0007 blk 0200 ab7cd000 ab999420 ab97c420 1ed0 12 PMSHL32
 0031  0008 0004 0008 0008 blk 0500 ab7d7000 ab999420 ab97ce20 1edc 12 PMSHL32
 0032  0008 0004 0008 0009 blk 0200 ab7d9000 ab999420 ab97d020 1edc 12 PMSHL32
 0034  0008 0004 0008 000b blk 0500 ab7dd000 ab999420 ab97d420      12 PMSHL32
 0035  0008 0004 0008 000c blk 0200 ab7df000 ab999420 ab97d620 1eac 12 PMSHL32
 0036  0008 0004 0008 000d blk 0500 ab7e1000 ab999420 ab97d820 1eb8 12 PMSHL32
 0044  0008 0004 0008 000e blk 0200 ab7fd000 ab999420 ab97f420 1ed0 12 PMSHL32
 0023  0006 0004 0006 0001 blk 0200 ab7bb000 ab998820 ab97b220      10 PMSPOOL
 0027  0006 0004 0006 0002 blk 0500 ab7c3000 ab998820 ab97ba20      10 PMSPOOL
 0029  0006 0004 0006 0003 blk 0200 ab7c7000 ab998820 ab97be20      10 PMSPOOL
 002a  0006 0004 0006 0004 blk 0500 ab7c9000 ab998820 ab97c020      10 PMSPOOL
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 002b  0006 0004 0006 0005 blk 0500 ab7cb000 ab998820 ab97c220      10 PMSPOOL
 000e  0005 0004 0005 0001 blk 0800 ab791000 ab998220 ab978820      00 HARDERR
 0013  0005 0004 0005 0002 blk 0800 ab79b000 ab998220 ab979220      00 HARDERR
 0014  0005 0004 0005 0003 blk 0800 ab79d000 ab998220 ab979420      00 HARDERR
 0049  0018 0017 0018 0001 blk 0200 ab807000 ab99ca20 ab97fe20 1cd4 04 FROMAGE
 004d# 001c 001b 001c 0001 blk 0300 ab80f000 ab99e220 ab980620 1e60 1d PAIN

>>> Pid 18 is evidently FROMAGE.

>>> FROMAGE has the VIN and PAIN wants it!

>>> We had better find out why FROMAGE has blocked.

# .s 49
Current slot number: 0049

# .r
eax=00000001 ebx=00020366 ecx=1bf90000 edx=00020004 esi=13fa0000 edi=13fa1052
```

```
eip=0000a0c3 esp=0000324a ebp=00023270 iopl=2 -- -- -- nv up ei ng nz ac pe cy
cs=dfdf ss=0017 ds=9fe7 es=9fe7 fs=150b gs=0000  cr2=00000000  cr3=001d9000

Invalid linear address: dfdf:0000a0c3
# ln

dfdf:0000a053 DOSCALL1 GetCharIn + 70

>>> Looking at this from the application aspect may be difficult since
>>> some of the code has been paged out (The invalid linear address
>>> message).

>>> The near symbol gives a clue. We could try unwinding the stack and
>>> hope that the stack is still paged in.

# dw %ebp
%00023270  b17e dfdf 9fe7 0000 0000 a97a dfdf 0000
%00023280  0000 0366 9fe7 a8ec a48d 1052 a399 32a6
%00023290  203c 0053 9fdf 1052 1052 32c8 32a6 0360
%000232a0  0007 4fa8 32b4 b1f0 dfdf 9fd7 0000 0005
%000232b0  bb2b dfd7 0000 0000 1044 9fd7 1052 9fd7
%000232c0  32c8 0002 0053 1bf9 32f0 0002 1cba 1bf9
%000232d0  0000 0000 0000 0000 1044 13fa 1052 13fa
%000232e0  0000 0000 0360 0002 0360 0002 0042 0008
# d

%000232f0  3334 0002 78cd 0001 0000 0000 0000 0009
%00023300  1000 0000 3330 0002 0000 0000 0360 0002
%00023310  0360 0002 0000 0002 0000 0000 0000 0000
%00023320  0000 0000 0000 0000 0000 0000 0000 0000
%00023330  7000 ab80 3388 0002 2abc 0001 0360 0002
%00023340  0000 0009 1000 0000 33f0 0002 0029 0000
%00023350  0020 0000 0001 0000 0000 0000 0000 0000
%00023360  0000 0000 3388 0002 02c5 0001 339c 0002

>>> This is going to be haphazard. Evidently the code we are currently
>>> executing is not using EBP as a stack frame pointer. All we can do
>>> is scan through the stack looking for a likely stack frame or a
>>> return address to user code.

>>> %232f0 looks like a candidate. Let's unassemble the return address
>>> to see if it makes sense.

# u %178cd-10

%000178bd 03ca               add     ecx,edx
%000178bf 51                 push    ecx
%000178c0 8b5d08             mov     ebx,dword ptr [ebp+08]
%000178c3 ff7320             push    dword ptr [ebx+20]
%000178c6 b004               mov     al,04
%000178c8 e83ba3f71b         call    %1bf91c08
%000178cd 83c410             add     esp,+10
%000178d0 8bc8               mov     ecx,eax
%000178d2 0bc0               or      eax,eax
%000178d4 741b               jz      %000178f1
%000178d6 824b0802           or      byte ptr [ebx+08],02
%000178da c705e81302003c000000 mov    dword ptr [000213e8],0000003c
# ln  %1bf91c08
```

```
%1bf91c08 DOSCALL1 DOS32READ

>>> So far so good. We need to see if this is consistent with the
>>> BlockID, which is the most up-to-date status indicator for this
>>> process.


# .pb#
 Slot  Sta BlockID  Name     Type        Addr        Symbol
 0049# blk 05100604 FROMAGE
# .m  0510:0604

*har     par      cpg        va    flg next prev link hash hob   hal
 0003 %feaef04c 00000400 %fe6ef000 001 0002 0023 0000 0000 0003 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0003  0003 fec5  0000 ffec 0000  0000 00  01 00 00 vmkrhrw

>>> This BlockID points to data within the kernel resident read/write
>>> heap. Heap blocks have headers that tell us more about the user of
>>> the data.  The data portion of a help block is usually mapped to a
>>> GDT selector.  In this example, selector 510.  510:0 should
>>> be the address of the beginning of the data and therefore point
>>> just after the end of the header.  We look at the data before
>>> 510:0 to see the heap block header.

# dg 510
0510  Data    Bas=fe6f3000 Lim=00000b2a DPL=0 P  RW    A
# dd % fe6f3000-10

%fe6f2ff0  ffa4000c feaeef28 0002036f 00000b39
%fe6f3000  05000000 0000c981 424b2d29 20202444
%fe6f3010  05182020 00000510 00020000 00000000
%fe6f3020  a8030ec8 424b0170 19002444 00000000
%fe6f3030  00000000 00000000 00000000 00000000
%fe6f3040  00000000 00061400 001b0014 0a000003
%fe6f3050  00028101 81010500 0a000001 0003c747
%fe6f3060  00000000 00000000 00000000 00000000


>>> For resident heaps the header is a double-word.  This one begins
>>> at %fe6f2ffc.  The low 2 bits are flags, the remainder is the
>>> length of the heap block in double-words.

>>> If the flag bit 0 is 1 then this is an extended heap. Which it is.
>>> We need to look at the header extension at the end of the block.

>>> The length of the block in bytes is b38 (by mentally AND-ing b39
>>> and 0xffc)


# dd % fe6f3000-4+b38-10

%fe6f3b24  ffff0000 0000ffff ff530510 ff77bd64
%fe6f3b34  ffc2001c 00000008 00000000 00000000
%fe6f3b44  ab240001 4d5000a6 00005854 ff9e0014
%fe6f3b54  001b0083 fe6f3b80 fe701da0 fe8777b0
%fe6f3b64  ffa4000c fe6f3b74 000102e9 ffa4000c
%fe6f3b74  fe7ea73c 0001071f ff9e0014 001b005d
```

```
%fe6f3b84  fe83baa8 fe876eac fe877654 ffc2001c
%fe6f3b94  00000008 00000000 00000000 da680001

>>> The header extension is in the last 2 double-words of the heap
>>> block.  The owner Id and the selector are in the first of these (at
>>> %fe6f3b2c).


# .mo ff53

ff53 dd4

>>> This tells us selector 510 was allocated by, or is part of the 4th
>>> device driver to initialize. Listing the physical device driver
>>> MTEs will find this. The are listed last initialized first.

>>> Note: frequently we find that dd16 is the owner. This refers to
>>> all device drivers from the 16th and subsequent. The first 15
>>> device drivers to initialize are assigned unique owner ids from
>>> dd1 to dd15, where the numbers are in decimal.

# .lmp
hmte=0249 pmte=%fe848df4 mflags=0008f1c9 h:\faxpro\fmd.sys
hmte=0242 pmte=%fe856f04 mflags=0008f1c9 c:\os2tools\theseus2.sys
hmte=0240 pmte=%fe848e7c mflags=0008f1c9 c:\os2\vdisk.sys
hmte=0235 pmte=%fe848f58 mflags=0008f1c9 h:\os2\apps\sysios2.sys
hmte=0234 pmte=%fe848f08 mflags=0008f1c9 h:\tcpip\bin\ifndisnl.sys
hmte=011e pmte=%fe83ff84 mflags=0008f1c9 h:\tcpip\bin\inet.sys
hmte=012b pmte=%fe83996c mflags=0008f1c9 h:\mmos2\r0stub.sys
hmte=012a pmte=%fe839da0 mflags=0000f1c1 h:\mmos2\ssmdd.sys
hmte=0123 pmte=%fe83df4c mflags=8008f1c9 c:\os2\com.sys
hmte=0121 pmte=%fe839e64 mflags=8008f1c9 h:\os2\boot\mouse.sys
hmte=0120 pmte=%fe839ef4 mflags=8008f1c9 h:\os2\boot\pointdd.sys
hmte=011d pmte=%fe83afdc mflags=8008f1c9 h:\os2\boot\os2cdrom.dmd
hmte=0111 pmte=%fe83af88 mflags=8008f1ca h:\os2\boot\pmdd.sys
hmte=0087 pmte=%fe839fdc mflags=0008f1c9 h:\os2\boot\dos.sys
hmte=0089 pmte=%fe839f80 mflags=8008f1c9 h:\os2\boot\testcfg.sys
hmte=0103 pmte=%fe71095c mflags=0008f1c9 i:\brew\os20memu.sys
hmte=00e2 pmte=%fe6fefc4 mflags=8008e1c9 h:\os2scsi.dmd
hmte=00e1 pmte=%fe6fdf64 mflags=8008e1c9 h:\os2dasd.dmd
hmte=00de pmte=%fe6f6fb0 mflags=0008e1c9 h:\xdfloppy.flt
hmte=00a9 pmte=%fe6f5fb0 mflags=8008e1c9 h:\fd16-700.add
hmte=00a7 pmte=%fe6f4f3c mflags=8008e1c9 h:\ibm1s506.add
hmte=00a5 pmte=%fe6f3db4 mflags=8008e1c9 h:\ibm1flpy.add
hmte=009c pmte=%feaeeea0 mflags=8008e1c9 h:\print01.sys
hmte=009b pmte=%fe6f1fb8 mflags=8008e1c9 h:\ibmkbd.sys
hmte=009a pmte=%feaeeed8 mflags=8008e1c9 h:\kbdbase.sys
hmte=0099 pmte=%feaeef34 mflags=0008e1c9 h:\screen01.sys
hmte=0098 pmte=%feaeefc0 mflags=8008e1c9 h:\clock01.sys
hmte=0096 pmte=%fe6f1fdc mflags=0008e1c9 h:\resource.sys

>>> Counting backwards, the 4th device driver is KBDBASE.SYS.

>>> We dump its object table.


# .lmo 9a
hmte=009a pmte=%feaeeed8 mflags=8008e1c9 h:\kbdbase.sys
seg  sect psiz vsiz hob  sel  flags
```

```
0001 0001 158c 170e 0000 0510 8c41 data prel
0002 000c 3270 3270 0000 0518 8d60 code shr prel rel
0003 0026 1987 1988 0000 0520 8d60 code shr prel rel
0004 0033 0743 0744 0000 0528 8d60 code shr prel rel

>>> Selector 510 is indeed the first data selector of KBDBASE.SYS and
>>> will contain the device driver header at offset +0

# .d dev 510:0
          DevNext: 0500:0000
          DevAttr: c981
         DevStrat: 0000
           DevInt: 2d29
          DevName: KBD$
        DevProtCS: 0518
        DevProtDS: 0510
        DevRealCS: 0000
        DevRealDS: 0000


>>> We conclude that FROMAGE is waiting for the device driver to
>>> respond to the DosRead - That is, a keyboard interrupt
```

14.1.2 Exploring Memory Management

This section gives a basic overview of memory management, and shows how to answer the following questions:

1. 14.1.2.1, “Who Owns Virtual Memory and Who Allocated it?”
2. 14.1.2.2, “How to Correlate Named Memory with its Address” on page 262
3. 14.1.2.3, “How Memory Aliasing Works” on page 267

If the reader is unfamiliar with this subject then these sections should be read in order.

14.1.2.1 Who Owns Virtual Memory and Who Allocated it?

In this section we take a look at the primary system structures used in memory management and how they are located using the Dump Formatter and Kernel Debugger. These structures are:

- The memory arena record (VMAR)
- The memory arena header record (VMAH)
- The memory object record (VMOB)
- The memory context record (VMCO)

The examples worked in this section illustrate:

- How to find all memory allocations made by a given process and what executable made the allocation.
- How to determine ownership of non-system memory.
- The use of memory objects, pseudo-objects and system objects.

Memory allocations have many attributes, included among which are:

- Data or content
- Location or address
- Size
- Ownership
- Requestor

The composite set of attributes associated with a memory allocation is referred to as a memory object. OS/2′s virtual memory manager tracks memory objects using arena, object and context records.

We start by looking at the arena record, which is used to record virtual address assignments to memory objects.

The entire system address space of 4GB is partitioned into three types of memory arena:

System Arena
: This is the range of virtual addresses where system information and ring 0 code executes. Typically device drivers, file system drivers and the OS/2 kernel executes and uses data assigned to the system arena. There is just

one instance of the system arena. It is assigned the virtual address range from 512MB to 4GB.

Shared Arena

This is the range of virtual addresses assigned to shared objects. Shared data objects come in the following two varieties:

Global data Such objects exist as unique entities. Their address range and data content are common to all accessing processes. This is achieved by using common page tables in all processes.

Instance data Such objects share the same address range, but exist as distinct data instances in each accessing process. Page table entries for instance data are specific to each process.

Code objects from DLL modules are also consigned to the shared arena.

In general processes are not given automatic access to instance or global data. Access is granted either implicitly by the system loader because of calls to other DLLs or explicitly by use of the DosGiveXxxx and DosGetXxxx set of APIs.

There is just one shared arena, which initially reserves virtual memory addresses from 304MB to 512MB. This may be expanded by lowering the lower boundary. The current address range assigned to the shared arena is managed by a special arena record called the boundary sentinel arena record.

Private Arena

This is the range of virtual addresses used to map objects that are unique to each process. A private arena therefore exists for each process. In general the page tables of each private arena will map to unique real storage frames. An exception to this is with code objects. Since code segments are always read-only then if more than one process is running the same executable module their page tables will map to a common set of real storage frames for the code segments of the executable module.

Private arenas are assigned an initial address range from 64K to 64MB. This may be expanded upwards as more memory is allocated. The current size of a private arena is tracked by a special arena record called the sentinel arena record.

The private arena upper boundary and shared arena lower boundary may grow towards each other but not overlap.

These worked examples now follow:

Exploring arena records

Exploring object records

Finding who owns memory

Exploring Arena Records: The following example illustrates the use of arena records:

```
>>> We start by asking the question: what ranges of addresses are
>>> currently allocated in the private arena of the process that's
>>> running the IPFC complier.

>>> List all processes to find the one of interest

# .p
 Slot  Pid  PPid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe3a000 ffe3c7d4 ffe3c61c 1e7c 00 *ager
 0002  0001 0000 0000 0002 blk 0200 7b7aa000 ffe3c7d4 7b9a8020 1f3c 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 7b7ac000 ffe3c7d4 7b9a81d8 1f50 00 *ctxh
 0004  0001 0000 0000 0004 blk 081f 7b7ae000 ffe3c7d4 7b9a8390 1f48 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 7b7b0000 ffe3c7d4 7b9a8548 1f20 00 *lazyw
 0006  0001 0000 0000 0006 blk 0800 7b7b2000 ffe3c7d4 7b9a8700 1f3c 00 *asyncr
 0009  0002 0000 0002 0001 blk 021f 7b7b8000 7b9c4020 7b9a8c28      00 LOGDAEM
 0008  0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
 000b  0003 0001 0003 0002 blk 0800 7b7bc000 7b9c484c 7b9a8f98      01 PMSHL32
 000c  0003 0001 0003 0003 blk 0800 7b7be000 7b9c484c 7b9a9150      01 PMSHL32
 000d  0003 0001 0003 0004 blk 0800 7b7c0000 7b9c484c 7b9a9308      01 PMSHL32
 000e  0003 0001 0003 0005 blk 0800 7b7c2000 7b9c484c 7b9a94c0      01 PMSHL32
 0007  0003 0001 0003 0006 blk 0200 7b7b4000 7b9c484c 7b9a88b8 1ecc 01 PMSHL32
 0011  0003 0001 0003 0007 blk 0200 7b7c8000 7b9c484c 7b9a99e8 1ecc 01 PMSHL32
 0012  0003 0001 0003 0008 blk 0200 7b7ca000 7b9c484c 7b9a9ba0      01 PMSHL32
 0013  0003 0001 0003 0009 blk 0200 7b7cc000 7b9c484c 7b9a9d58      01 PMSHL32
 0014  0003 0001 0003 000a blk 0800 7b7ce000 7b9c484c 7b9a9f10      01 PMSHL32
 0015  0003 0001 0003 000b blk 0800 7b7d0000 7b9c484c 7b9aa0c8      01 PMSHL32
 0016  0003 0001 0003 000c blk 0800 7b7d2000 7b9c484c 7b9aa280      01 PMSHL32
 0017  0003 0001 0003 000d blk 0804 7b7d4000 7b9c484c 7b9aa438 1ea8 01 PMSHL32
 0018  0003 0001 0003 000e rdy 0804 7b7d6000 7b9c484c 7b9aa5f0      01 PMSHL32
 0019  0003 0001 0003 000f blk 0500 7b7d8000 7b9c484c 7b9aa7a8      01 PMSHL32
 001a  0003 0001 0003 0010 rdy 0801 7b7da000 7b9c484c 7b9aa960 1bac 01 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001b  0003 0001 0003 0011 blk 0800 7b7dc000 7b9c484c 7b9aab18      01 PMSHL32
*001c# 0003 0001 0003 0012 run 0800 7b7de000 7b9c484c 7b9aacd0 1b8c 01 PMSHL32
 001d  0003 0001 0003 0013 blk 0200 7b7e0000 7b9c484c 7b9aae88      01 PMSHL32
 0023  0018 0003 0018 0001 rdy 061f 7b7ec000 7b9c7128 7b9ab8d8 1eb8 13 EPM
 0038  0018 0003 0018 0002 blk 0200 7b816000 7b9c7128 7b9adcf0 1ecc 13 EPM
 0037  0013 0003 0013 0001 blk 0200 7b814000 7b9c9a04 7b9adb38      19 IBMAVSD
 0033  0012 0003 0012 0001 blk 0200 7b80c000 7b9c89ac 7b9ad458 1eb8 17 PMDRAW
 0035  0012 0003 0012 0002 blk 0200 7b810000 7b9c89ac 7b9ad7c8 1eb8 17 PMDRAW
 0036  0012 0003 0012 0003 blk 0200 7b812000 7b9c89ac 7b9ad980      17 PMDRAW
 0034  0010 0003 0010 0001 blk 0400 7b80e000 7b9c91d8 7b9ad610 1ed4 12 CMD
 002e  000d 0003 000d 0001 blk 0200 7b802000 7b9c8180 7b9acbc0 1eb8 16 PULSE
 0030  000d 0003 000d 0002 rdy 0100 7b806000 7b9c8180 7b9acf30 1f28 16 PULSE
 002f  000d 0003 000d 0003 rdy 081f 7b804000 7b9c8180 7b9acd78 1f00 16 PULSE
 002d  000c 0003 000c 0001 blk 0200 7b800000 7b9c7954 7b9aca08 1eb8 15 DINFO
 0032  000c 0003 000c 0002 rdy 061f 7b80a000 7b9c7954 7b9ad2a0 1f00 15 DINFO
 002c  000b 0003 000b 0001 blk 0200 7b7fe000 7b9c58a4 7b9ac850 1eb8 14 MRFILE32
 0031  000b 0003 000b 0002 blk 0200 7b808000 7b9c58a4 7b9ad0e8 1ecc 14 MRFILE32
 0029  000a 0003 000a 0001 rdy 061f 7b7f8000 7b9c68fc 7b9ac328 1eb8 10 PMDIARY
 001f  0006 0003 0006 0001 rdy 062f 7b7e4000 7b9c60d0 7b9ab1f8 1eb8 11 PMSHL32
 0021  0006 0003 0006 0002 blk 0200 7b7e8000 7b9c60d0 7b9ab568      11 PMSHL32
 0022  0006 0003 0006 0003 blk 0200 7b7ea000 7b9c60d0 7b9ab720 1eb8 11 PMSHL32
 0020  0006 0003 0006 0004 blk 0200 7b7e6000 7b9c60d0 7b9ab3b0      11 PMSHL32
 001e  0006 0003 0006 0005 blk 0200 7b7e2000 7b9c60d0 7b9ab040 1ecc 11 PMSHL32
 0024  0006 0003 0006 0006 blk 0200 7b7ee000 7b9c60d0 7b9aba90      11 PMSHL32
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0025  0006 0003 0006 0007 blk 0200 7b7f0000 7b9c60d0 7b9abc48      11 PMSHL32
 0026  0006 0003 0006 0008 blk 0200 7b7f2000 7b9c60d0 7b9abe00      11 PMSHL32
```

```
 0027  0006 0003 0006 0009 blk 0200 7b7f4000 7b9c60d0 7b9abfb8      11 PMSHL32
 0028  0006 0003 0006 000a blk 0200 7b7f6000 7b9c60d0 7b9ac170      11 PMSHL32
 002a  0006 0003 0006 000c blk 021f 7b7fa000 7b9c60d0 7b9ac4e0 1eac 11 PMSHL32
 002b  0006 0003 0006 000d blk 0200 7b7fc000 7b9c60d0 7b9ac698 1eb8 11 PMSHL32
 000a  0004 0003 0004 0001 blk 0800 7b7ba000 7b9c5078 7b9a8de0      00 HARDERR
 000f  0004 0003 0004 0002 blk 0800 7b7c4000 7b9c5078 7b9a9678      00 HARDERR
 0010  0004 0003 0004 0003 blk 0800 7b7c6000 7b9c5078 7b9a9830      00 HARDERR
 0039  0019 0010 0019 0001 rdy 061f 7b818000 7b9ca230 7b9adea8 1f0c 12 IPFC

>>> From the name printed in the right hand column we see that slot 39
>>> is the one of interest.


>>> Imbedded in each PTDA at offset +0x40 is the VMAH that heads the
>>> private arena.  From the VMAH we can obtain the pointer to the
>>> sentinel area record.

>>> Dump out the VMAH for slot 39 using the pPTDA address from the
>>> .p command output...

# dd %7b9ca230+40 L10
%7b9ca270  7b9c7168 fff13190 feb24cae feb261bc
%7b9ca280  fe79ba54 fe87e9a0 fff03e30 00000002
%7b9ca290  00010000 00370000 00000000 00000000
%7b9ca2a0  00000003 00000000 00000041 000005db

>>> The third double word (feb24cae) is the address of the sentinel
>>> record.  To format this using the .MA command we need to determine
>>> the handle for this record.  Arena records are organized in a
>>> table of 0x16 byte length entries.  Their handles are their
>>> corresponding table entry number.  The address of the first
>>> arena record is located at symbol _parvmone...

# dd _parvmone l1
%fff13304  feb1f020

>>> Arena record 1 is located at %feb1f020.  We wish to determine the
>>> handle for the sentinel, whose address is %feb24cae.  We use the
>>> hex calculator facility of the Dump Formatter/Kernel Debugger thus.

# ? (%feb24cae-%feb1f020)/16 +1
%00000436

>>> The handle we require is 436.  We can now format the sentinel
>>> for slot 39 ....

# .ma  436
 har     par      cpg        va    flg next prev link hash hob   hal
 0436 %feb24cae 00000000 %00010000 003 050a 0526 0005 0000 4000 0000 max=%04000000

>>> Note the max=%04000000 to the right indicating the current private
>>> arena maximum address is 64M - 1 and incidentally distinguishing
>>> this as a sentinel or boundary sentinel arena record.
>>> Note also that this is merely a boundary marker and not an indication
>>> of which addresses within the private arena have been allocated.

>>> Regular arena record are chained to the sentinel in a circular
>>> double linked list using the 'next' and 'prev' pointers.
>>> We can format the entire chain using .MAL (or .MAR) but we have to
```

```
>>> break in using Ctrl-C to stop the chain endlessly traversing the
>>> circular chain.

# .ma 50a
 har     par      cpg        va    flg next prev link hash hob   hal
 050a %feb25ee6 00000030 %00010000 1d9 0509 0436 0000 0000 05e2 0000 hptda=050c
# .ma 509
 har     par      cpg        va    flg next prev link hash hob   hal
 0509 %feb25ed0 00000010 %00040000 179 050b 050a 0000 0000 05dd 0000 hptda=050c
# .ma 50b
 har     par      cpg        va    flg next prev link hash hob   hal
 050b %feb25efc 00000010 %00050000 169 0507 0509 0000 0000 05e9 0000 hptda=050c
# .mal 507
 har     par      cpg        va    flg next prev link hash hob   hal
 0507 %feb25ea4 00000010 %00060000 169 0506 050b 0000 0000 05ea 0000 hptda=050c
 0506 %feb25e8e 00000010 %00070000 169 050f 0507 0000 0000 05eb 0000 hptda=050c
 050f %feb25f54 00000010 %00080000 169 050c 0506 0000 0000 05ec 0000 hptda=050c
 050c %feb25f12 00000010 %00090000 169 0511 050f 0000 0000 05ed 0000 hptda=050c
 0511 %feb25f80 00000010 %000a0000 169 050e 050c 0000 0000 05f0 0000 hptda=050c
 050e %feb25f3e 00000010 %000b0000 1c9 050d 0511 01c7 0000 05ee 0016 hptda=050c
 050d %feb25f28 00000010 %000c0000 169 0512 050e 0000 0000 05f1 0000 hptda=050c
 0512 %feb25f96 00000010 %000d0000 169 0513 050d 0000 0000 05f2 0000 hptda=050c
 0513 %feb25fac 00000010 %000e0000 169 0514 0512 0000 0000 05f3 0000 hptda=050c
 0514 %feb25fc2 00000010 %000f0000 169 0515 0513 0000 0000 05f4 0000 hptda=050c
 0515 %feb25fd8 00000010 %00100000 169 0516 0514 0000 0000 05f5 0000 hptda=050c
 0516 %feb25fee 00000010 %00110000 169 0517 0515 0000 0000 05f6 0000 hptda=050c
 0517 %feb26004 00000010 %00120000 169 0519 0516 0000 0000 05f7 0000 hptda=050c
 0519 %feb26030 00000010 %00130000 169 0518 0517 0000 0000 05f9 0000 hptda=050c
 0518 %feb2601a 00000010 %00140000 169 051a 0519 0000 0000 05f8 0000 hptda=050c
 051a %feb26046 00000010 %00150000 169 051b 0518 0000 0000 05fa 0000 hptda=050c
 051b %feb2605c 00000010 %00160000 169 051c 051a 0000 0000 05fb 0000 hptda=050c
 051c %feb26072 00000010 %00170000 169 051d 051b 0000 0000 05fc 0000 hptda=050c
 051d %feb26088 00000010 %00180000 169 051e 051c 0000 0000 05fd 0000 hptda=050c
 051e %feb2609e 00000010 %00190000 169 0521 051d 0000 0000 05fe 0000 hptda=050c
 0521 %feb260e0 00000010 %001a0000 169 0520 051e 0000 0000 0601 0000 hptda=050c
 0520 %feb260ca 00000010 %001b0000 169 051f 0521 0000 0000 0600 0000 hptda=050c
 051f %feb260b4 000000f0 %001c0000 169 0523 0520 0000 0000 05ff 0000 hptda=050c
 har     par      cpg        va    flg next prev link hash hob   hal
 0523 %feb2610c 00000010 %002b0000 169 0527 051f 0000 0000 0603 0000 hptda=050c
 0527 %feb26164 00000010 %002c0000 169 0522 0523 0000 0000 0607 0000 hptda=050c
 0522 %feb260f6 00000020 %002d0000 169 0525 0527 0000 0000 0602 0000 hptda=050c
 0525 %feb26138 00000010 %002f0000 169 0524 0522 0000 0000 0605 0000 hptda=050c
 0524 %feb26122 00000010 %00300000 169 052a 0525 0000 0000 0604 0000 hptda=050c
 052a %feb261a6 00000010 %00310000 169 052d 0524 0000 0000 060a 0000 hptda=050c
 052d %feb261e8 00000010 %00320000 169 0529 052a 0000 0000 060d 0000 hptda=050c
 0529 %feb26190 00000010 %00330000 169 052b 052d 0000 0000 0609 0000 hptda=050c
 052b %feb261bc 00000020 %00340000 169 0526 0529 0000 0000 060b 0000 hptda=050c
 0526 %feb2614e 00000010 %00360000 169 0436 052b 0000 0000 0606 0000 hptda=050c
 0436 %feb24cae 00000000 %00010000 003 050a 0526 0005 0000 4000 0000 max=%04000000
 050a %feb25ee6 00000030 %00010000 1d9 0509 0436 0000 0000 05e2 0000 hptda=050c
 0509 %feb25ed0 00000010 %00040000 179 050b 050a 0000 0000 05dd 0000 hptda=050c

#

>>> Each regular private arena record is distinguished by the appearance
>>> hptda=nnn to the right of each line.  This is the handle of the PTDA
>>> of the process to which the arena record belongs.  Each of the hptda
>>> values is 50c indicating each of regular arena records above belongs
>>> to the same process.  More on the hptda later.
```

```
>>> Each regular arena represents the address range reserved for a
>>> memory object.  cpg is the size reservation in pages, but note
>>> that this is only an address space reservation, not necessarily what
>>> is currently committed.  Most objects reserve 0x10
>>> pages or 64K, which corresponds to the maximum 16-bit segment size.

>>> va shows the start address of each memory object.
>>> By examining va and cpg we can see that the minimum and maximum
>>> addresses allocated in the private arena of slot 39 is %10000 and
>>> %36ffff (=%360000 + 0x10 pages -1).  We can also see that this
>>> allocation is contiguous and therefore the total allocated private
>>> arena virtual address space is 0x360000 bytes or 3.375M

>>> The VMAH records the minimum and maximum +1 allocated addresses
>>> at +0x20 and +0x24, but the allocation might be sparse so the VMAH
>>> does not indicate directly the total memory in use.

>>> We now move onto the shared arena.

>>> The link field of each sentinel points to the boundary sentinel


# .ma  436
 har     par      cpg        va    flg next prev link hash hob   hal
 0436 %feb24cae 00000000 %00010000 003 050a 0526 0005 0000 4000 0000 max=%04000000
# .ma 5
 har     par      cpg        va    flg next prev link hash hob   hal
 0005 %feb1f078 00011a20 %04000000 007 0508 0075 0000 0000 fff0 0000 max=%1fff0000

>>> Once again each regular arena record in the shared arena is linked
>>> in a circular double linked list. This time we enter the chain from
>>> the boundary sentinel next and prev fields.

# .mal 508
 har     par      cpg        va    flg next prev link hash hob   hal
 0508 %feb25eba 00000010 %15a20000 369 0437 0005 0000 0000 05df 0000  hco=008a8
 0437 %feb24cc4 00000010 %15a40000 369 0438 0508 0000 0000 050d 0000  hco=00248
 0438 %feb24cda 00000010 %15a50000 369 0444 0437 0000 0000 050e 0000  hco=0076e
 0444 %feb24de2 00000020 %15a60000 3d9 0441 0438 0000 0000 0518 0000  hco=007aa
 0441 %feb24da0 00000010 %15a80000 3d9 043b 0444 0000 0000 051a 0000  hco=007a9
 043b %feb24d1c 00000010 %15a90000 3d9 043a 0441 0000 0000 0517 0000  hco=002b7
 043a %feb24d06 00000010 %15aa0000 3d9 0443 043b 0000 0000 0511 0000  hco=007a8
 0443 %feb24dcc 00000010 %15ab0000 179 0439 043a 0000 0000 0519 0000      =0000
 0439 %feb24cf0 00000010 %15ac0000 369 0433 0443 0000 0000 050f 0000  hco=00763
 0433 %feb24c6c 00000010 %15ad0000 369 0432 0439 0000 0000 0509 0000  hco=00777
 0432 %feb24c56 00000010 %15ae0000 369 041e 0433 0000 0000 0508 0000  hco=00776
 041e %feb24a9e 00000030 %15af0000 369 041c 0432 0000 0000 04f4 0000  hco=007d8
 041c %feb24a72 00000010 %15b20000 369 03ee 041e 0000 0000 04d1 0000  hco=0075c
 03ee %feb2467e 00000010 %15b30000 349 03eb 041c 0000 0000 04c1 0000  hco=001f6

  .
  .
  .
  .
  .
  .

 0169 %feb20f10 00000010 %1acb0000 179 0168 016a 0000 0000 05e6 0000      =0000
```

```
 0168 %feb20efa 00000020 %1acc0000 379 0077 0169 0000 0000 01af 0000  hco=007c0
 0077 %feb1fa44 00000010 %1bfe0000 349 0075 0168 0000 0000 0077 0000  hco=007a7
 0075 %feb1fa18 00000010 %1bff0000 349 0005 0077 0000 0000 0075 0000  hco=007b8
 0005 %feb1f078 00011a20 %04000000 007 0508 0075 0000 0000 fff0 0000 max=%1fff0000
 0508 %feb25eba 00000010 %15a20000 369 0437 0005 0000 0000 05df 0000  hco=008a8
 0437 %feb24cc4 00000010 %15a40000 369 0438 0508 0000 0000 050d 0000  hco=00248

>>> There are two types of regular arena record that appear in the
>>> Shared arena.  These are distinguished by the right-hand column:
>>>   hco=nnnnn
>>>      =0000
>>> The first type is global shared data.  The hco is the context record
>>> handle, which will be discussed later.
>>> The second type represents instance data.  Both of these will be
>>> looked at in more detail in the next section.

>>> Finally we look at the system arena.  The sentinel for the system
>>> arena is har=4.  Once again each regular arena record is linked
>>> in a circular double-linked list.

# .ma 4
 har     par      cpg        va    flg next prev link hash hob   hal
 0004 %feb1f062 00000000 %60000000 003 0504 0016 0000 0000 ffc0 0000 max=%fffc0000
# .mal 504
 har     par      cpg        va    flg next prev link hash hob   hal
 0504 %feb25e62 00000010 %79eb7000 121 03d2 0004 0000 0081 05e1 0000      =0000
 03d2 %feb24416 00000010 %79ec7000 121 0363 0504 0000 0080 049e 0000      =0000
 0363 %feb23a8c 00000010 %79ed7000 121 0374 03d2 0000 007f 0434 0000      =0000
 0374 %feb23c02 00000010 %79ee7000 121 02e4 0363 0000 0095 0422 0000      =0000
 02e4 %feb22fa2 00000010 %79ef7000 121 02db 0374 0000 00df 0382 0000      =0000
 02db %feb22edc 00000010 %79f07000 121 02cc 02e4 0000 007c 036e 0000      =0000
 02cc %feb22d92 00000010 %79f17000 121 0405 02db 0000 007b 0350 0000      =0000

 .
 .
 .
 .

 0012 %feb1f196 00000016 %ffefe000 001 0013 0011 0000 0000 0013 0000      =0000
 0013 %feb1f1ac 00000010 %fff14000 009 0014 0012 0000 0000 0014 0000   sel=0150
 0014 %feb1f1c2 0000000a %fff24000 009 0015 0013 0000 0000 0015 0000   sel=0158
 0015 %feb1f1d8 00000010 %fff2e000 009 0016 0014 0000 0000 0016 0000   sel=0160
 0016 %feb1f1ee 00000082 %fff3e000 001 0004 0015 0000 0000 0017 0000      =0000
 0004 %feb1f062 00000000 %60000000 003 0504 0016 0000 0000 ffc0 0000 max=%fffc0000
 0504 %feb25e62 00000010 %79eb7000 121 03d2 0004 0000 0081 05e1 0000      =0000
 03d2 %feb24416 00000010 %79ec7000 121 0363 0504 0000 0080 049e 0000      =0000


>>> Two types of regular record appear, distinguished by the right-hand
>>> column:
>>>     =0000
>>>  sel=nnnn
>>> The first of these indicates heap data.  The second GDT selector
>>> assigned data.  Device driver and IFS code and data objects will
>>> appear among these.
```

Exploring Object Records: We now explore the memory object record (VMOB) and the .MO command.

```
>>> There are two types of object managed by object records:
>>>     pseudo-objects
>>>     non-pseudo-object

>>> Non-pseudo-objects have an associated arena record.  These are by
>>> far the most common type of memory object.  They include code and
>>> data segments of application and system code.  However, there is
>>> a draw-back in that arena records deal with page size quantities.
>>> For certain system control blocks it is useful to have distinct
>>> objects associated with each instance of them.  But these objects
>>> are generally very much smaller than a page.  To overcome these
>>> difficulties such objects are sub-allocated from the system
>>> heap and given an object type of pseudo-object.  They have
>>> no associated arena record.

>>> We list a few pseudo-objects.  Note the 'p' parameter of .mo to
>>> do this.


# .mop
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0004  %fff13238  8000 ffe1 0000  0000 00  00 00 00 vmah
 0005  %fff13190  8000 ffe1 0000  0000 00  00 00 00 vmah
 0006  %fff0a891  8000 ffa6 0000  0000 00  00 00 00 mte       doscalls.dll
 0072  %ffe3c7d4  8000 ffcb 0000  0000 00  00 00 00 ptda 0001 *sysinit
 007a  %fff0b3fa  8000 ffa6 0000  0000 00  00 00 00 mte       mvdm.dll
 007b  %fff0b26b  8000 ffa6 0000  0000 00  00 00 00 mte       fshelper.dll
 007d  %fe720f60  8000 ffa6 0000  0000 00  00 00 00 mte       a:mini_fsd.fsd
 0086  %fe861ee0  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmdd.sys
 0087  %fe861f30  8000 ffa6 0000  0000 00  00 00 00 mte       c:dos.sys
 0088  %fe861f58  8000 ffa6 0000  0000 00  00 00 00 mte       c:testcfg.sys
 008a  %fe860f9c  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmshapim.dll
 0091  %7b9c484c  8000 ffcb ff79  0000 00  00 00 00 ptda 0003 c:pmshell.exe
 0096  %fe721fb8  8000 ffa6 0000  0000 00  00 00 00 mte       c:clock01.sys
 0097  %fe721f1c  8000 ffa6 0000  0000 00  00 00 00 mte       c:screen01.sys
 0098  %fe721eb0  8000 ffa6 0000  0000 00  00 00 00 mte       c:kbd01.sys
 0099  %fe7246bc  8000 ffa6 0000  0000 00  00 00 00 mte       c:print01.sys
 009f  %fe724f84  8000 ffa6 0000  0000 00  00 00 00 mte       c:ibm1flpy.add
 00a1  %fe725f88  8000 ffa6 0000  0000 00  00 00 00 mte       c:ibm1s506.add

#

>>> Pseudo-objects apply to four types of control block but in general
>>> we will only be concerned with the PTDA and the MTE.
>>> The pseudo-object record is distinguished by the presence of the
>>> 'va' field.  The control block name is shown in column 11.
>>>  |                                                    |
>>>  .......                                              |
>>>        |                                              |
>>>        V                                              V
# .mo 91
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0091  %7b9c484c  8000 ffcb ff79  0000 00  00 00 00 ptda 0003 c:pmshell.exe
```

```
>>> The 'va' field gives the address of the object itself.  In this
>>> it's a PTDA address.  We can find the thread slots which correspond
>>> to this PTDA either by using .P and looking for a match in the
>>> pPTDA field or directly:

# dw %7b9c484c+Pid-ptda_start l1
%7b9c5042  0003
>>> This is the Pid.  Note .mo 91 extracts this for us - the Pid appears
>>> after 'ptda'

# dd %7b9c484c+ptda_pTCBHead-ptda_start l1
%7b9c486c  7b9a8a70
>>> This is the head of the TCB tree for Pid 3.

# dw 7b9a8a70 l2
%7b9a8a70  0001 0008
>>> Words 0 and 1 of the TCB contain the thread ordinal and its slot
>>> number. This is Tid 1 in slot 8.

# .p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008  0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32

>>> The PTDA for slot 8 has Pid 3, is at %7b9c484c which is hob 91.
>>> The handle of the PTDA (hptda) is defined to be the hob of the
>>> object it occupies. Thus this ptda is also identified by hptda=91

>>> The other most frequently encountered pseudo-object is the mte.

 hob       va     flgs own  hmte  sown,cnt lt st xf
 0193  %fe722dec  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmshell.exe

>>> The MTE represents a loaded module. In this case the MTE control
>>> block is located at %fe722dec and is assigned the mte handle of its
>>> hob.  In this case the MTE at %fe722dec is also referred to as
>>> hmte=193.  The .LM command will respond to either hmte or MTE address
>>> and format the MTE for us ...
# .lm 193
hmte=0193 pmte=%fe722dec mflags=84903150 c:\os2\pmshell.exe

>>> .MO extracts the module name from the MTE and displays this to the
>>> right of 'mte'

>>> In each object record is the 'own' and 'hmte' fields.  These are used
>>> to attribute ownership and associate a module or part of the system
>>> that was involved with the allocation request.  In many cases
>>> these fields contain hobs of related objects.  In some cases
>>> attribution needs to be made to a system resource.  For this a
>>> number of generic system object Ids have been defined.  They all
>>> are greater then 0xff00.  .MO will translate system object Ids into
>>> a more meaningful text string.  For example: in hob 193 the
>>> owner id ffa6
# .mo ffa6
ffa6 ldrmte

>>> Similarly in hob 91 the owner is ffcb
# .mo ffcb
ffcb ptda
```

```
>>> Both of these give an indication of the type of system object.
>>> In the first case a load MTE, in the second a PTDA.  The 'own' and
>>> 'hmte' interpretation is used to form the description that appears
>>> to the right of each .MO line.


>>> We now turn our attention to non-pseudo objects or normal
>>> memory objects:


.mon
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0001  0001 fec8  0000 fff1 0000  0000 00  00 00 00 vmob
 0002  0002 fec8  0000 ffe3 0000  0000 00  00 00 00 vmar
 0003  0003 fec8  0000 ffec 0000  0000 00  01 00 00 vmkrhrw
 0007  0006 0000  0000 ff6d 0000  0000 00  00 00 00 doshlp
 0008  0007 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0009  0008 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000a  0009 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000b  000a 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000c  000b 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000d  000c 0000  0325 ffba 0000  0000 00  00 00 00 lock
 000e  000d 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 000f  000e 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0010  0087 0000  402c 0091 019f  0000 00  00 00 00 priv 0003 c:pmshell.exe
 0011  0010 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0012  0011 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 0013  0012 0000  0000 ffaa 0006  0000 00  00 00 00 os2krnl
 .
 .
 .
 .
 .
 009d  0096 0000  0225 ff8c 0000  0000 00  00 00 00 perfview
 009e  0097 0000  0524 ff88 ff54  0000 00  00 00 00 ptogdt    dd5
 00a0  0098 0000  0524 ff88 ff56  0000 00  00 00 00 ptogdt    dd7
 00a3  0099 0000  0524 ff88 ff56  0000 00  00 00 00 ptogdt    dd7
 00a4  009a 0000  0524 ff88 ff56  0000 00  00 00 00 ptogdt    dd7
 00a5  009b 0000  0524 ff88 ff56  0000 00  00 00 00 ptogdt    dd7
 00a6  009c 0000  0524 ff88 ff56  0000 00  00 00 00 ptogdt    dd7
 .
 .
 .

 00e1  00d5 0000  0324 ff93 0000  0000 00  00 00 00 fsbuf
 00e2  00d6 0000  482c fff7 019f  0000 00  00 00 04 giveget
 00e3  00d7 0000  0124 ff8f 0000  0000 00  00 00 00 resource
 00e4  01c8 0000  0824 0131 0131  0000 00  00 00 00 shared    c:display.dll
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00e6  00d9 0000  082c 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00e7  00da 0000  0838 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00e8  00db 0000  0838 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00e9  00dc 0000  0838 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00ea  00dd 0000  0838 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00eb  00de 0000  0830 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 00ec  00f1 0000  422c 0091 01b0  0000 00  00 00 00 priv 0003 c:pmshell.exe
 00ed  010d 0000  402c 0091 01b0  0000 00  00 00 00 priv 0003 c:pmshell.exe
 00ee  00e1 0000  082c 00f1 00f1  0000 00  00 00 00 shared    c:os2char.dll
 00f2  01c9 0000  0824 0131 0131  0000 00  00 00 00 shared    c:display.dll
```

```
 00f3  00e3 0000  0838 00f1 00f1  0000 00  00 00 00 shared    c:os2char.dll
 00f4  00e4 0000  482c fff7 00f1  0000 00  00 00 00 giveget
 00f6  00e5 0000  082c 00fb 00fb  0000 00  00 00 00 shared    c:sesmgr.dll
 00fc  00e6 0000  082c 00fb 00fb  0000 00  00 00 00 shared    c:sesmgr.dll
 00fd  00e7 0000  0838 00fb 00fb  0000 00  00 00 00 shared    c:sesmgr.dll
 00fe  00e8 0000  0838 00fb 00fb  0000 00  00 00 00 shared    c:sesmgr.dll
 00ff  00e9 0000  082c 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0101  00ea 0000  082c 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0102  00eb 0000  082c 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0103  00ec 0000  0838 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0104  00ed 0000  0838 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0105  00ee 0000  0838 0100 0100  0000 00  00 00 00 shared    c:quecalls.dll
 0106  01cd 0000  1024 0091 0131  0000 00  00 00 00 priv 0003 c:pmshell.exe
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0107  00f0 0000  0124 ffc4 0000  0000 00  00 00 00 smdfh
 0108  010f 0000  4a2c fff5 01b0  0000 00  00 00 00 give
 0109  0085 0000  0524 ff88 ff5b  0000 00  00 00 00 ptogdt    dd12
 010b  0084 0000  0524 ff88 ff5b  0000 00  00 00 00 ptogdt    dd12
 010c  0083 0000  0524 ff88 ff5b  0000 00  00 00 00 ptogdt    dd12
 010d  00f2 0000  0524 ff88 ff5b  0000 00  00 00 00 ptogdt    dd12
 .
 .
 .

#

>>> Many of these objects have a system ID owners but those of current
>>> interest are objects allocated within the shared and private arenas
>>> by application programs.

>>> Private arena private data:
>>> --------------------------

>>> We start by examining hob 10 in more detail.

>>> We list the object and its associated arena record using the 'c'
>>> parameter of .MO
# .moc 10

*har     par      cpg        va    flg next prev link hash hob   hal
 0087 %feb1fba4 00000010 %00070000 169 01a0 019a 0000 0000 0010 0000 hptda=0091
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0010  0087 0000  402c 0091 019f  0000 00  00 00 00 priv 0003 c:pmshell.exe

>>> We can tell from its location (%70000) that this is a private
>>> arena address in process hptda=91.  The 'own' field of the object
>>> record is also hob 91 which again implies an object owned by the
>>> process.


# .mo 91
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0091  %7b9c484c  8000 ffcb ff79  0000 00  00 00 00 ptda 0003 c:pmshell.exe

>>> This tells us the owner is a PTDA (that is, a process private arena)
>>> and the Pid is 3, which is executing PMSHELL.EXE
>>> Note: the Pid and executable have been extracted from hob 91 and
>>> displayed in the description area of hob 10.
```

```
>>> Now look at the hmte for hob 10.

# .mo 19f
 hob       va     flgs own  hmte  sown,cnt lt st xf
 019f  %fe8629e0  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmwin.dll

>>> This is the MTE for pmwin.dll.

>>> The 'own' and 'hmte' of hob 10 tell us that hob 10 was allocated in
>>> the private arena of process Pid 3 by pmwin.dll as a result of a
>>> direct or indirect call to pmwin from pmshell.

>>> The flags in hob 10 can give us more information on the
>>> characteristics of hob 10
>>> 4    0    2    c
    0100 0000 0010 1100
     '          '  ''
     '          '  ''. writeable
     '          '  '...user storage
     '          '......executable
     '.................API located

>>> The combination writeable + executable should be interpreted as
>>> R/W storage rather than executable storage.  Looking at the page
>>> table entry for %7000 in slot 8 (Pid 3) will confirm this:
# .s8
Current slot number: 0008
# .p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008  0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
# dp %70000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00070000* 012f3  frame=0120d  0    0  D  A        U  W  P  pageable
%00070000  0120d  frame=0120d  0    0  D  A        U  W  P  pageable
#
>>> Private arena shared data:
>>> -------------------------

# .moc 192
*har     par      cpg        va    flg next prev link hash hob   hal
 0249 %feb22250 00000010 %00010000 1c9 024a 0247 014e 0000 0192 0000 hptda=02a6
 014e %feb20cbe 00000010 %00010000 1d9 014f 008e 0000 0000 0192 0000 hptda=0091
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0192  0249 0000  0838 0193 0193  0000 00  00 00 00 shared    c:pmshell.exe

>>> Object 192 has two private arena records pointing to it.  One
>>> associated with hptda=2a6 and the other with hptda=91.  We established
>>> earlier that hptda=91 is Pid 3 and was running pmshell.exe

#.mo 2a6
 hob       va     flgs own  hmte  sown,cnt lt st xf
 02a6  %7b9c60d0  8000 ffcb 0000  0000 00  00 00 00 ptda 0006 c:pmshell.exe

>>> So hptda=2a6 refers to Pid 6, which is also running pmshell.exe
>>> Note the use of the 'link' field in har=249 to point to har=14e.  The two
>>> arena records are chained in this way to link all arena records that
>>> share a private data object.  The object record points to the head of
>>> the chain.
```

```
>>> The 'own' and 'hmte' fields both point to object 193.  This tells us
>>> that object 192 is not only allocated by the module whose
>>> hmte is 193, but is also part of this load module.
>>> This may be verified as follows:

#.mo 193
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0193  %fe722dec  8000 ffa6 0000  0000 00  00 00 00 mte       c:pmshell.exe

# .lmo 193
hmte=0193 pmte=%fe722dec mflags=84903150 c:\os2\pmshell.exe
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 00000600 00010000 80002025 00000001 00000001 0192 000f r-x shr big
0002 0000005c 00020000 80002003 00000002 00000001 0000 0017 rw- big
0003 0000fa20 00030000 80002003 00000003 00000001 0000 001f rw- big

>>> We actually discover this is object 1 of the pmshell.exe load
>>> module.

>>> Examining the flags from hob 192 we see:
>>> 0    8    3    8
>>> 0000 1000 0011 1000
>>>      '       '' '
>>>      '       '' '.... User storage
>>>      '       ''...... Readable
>>>      '       '....... Executable
>>>      '.............. Shared
>>>
>>> This is information summarized in the description field of hob 192.

>>> Finally we take a look at the page table entries for %10000 in Pid 3
>>> and 6. We should see the same real storage frame being accessed by
>>> both processes:
# .s8
Current slot number: 0008
# .p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008  0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
# dp %10000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00010000* 012f3  frame=011cb  0    0  c  A        U  r  P  pageable
%00010000  011cb  frame=011cb  0    0  c  A        U  r  P  pageable
# .s 1f
Current slot number: 001f
# .p 1f
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001f# 0006 0003 0006 0001 rdy 062f 7b7e4000 7b9c60d0 7b9ab1f8 1eb8 11 PMSHL32
# dp %10000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00010000* 0093d  frame=011cb  0    0  c  A        U  r  P  pageable
%00010000  011cb  frame=011cb  0    0  c  A        U  r  P  pageable
#

>>> In both processes %00010000 translates to %%011cb000



>>> Shared Arena, Global Data:
>>> -------------------------
```

```
# .moc e6

*har     par      cpg        va    flg next prev link hash hob   hal
 00d9 %feb202b0 00000010 %1a060000 379 00d8 00da 0000 0000 00e6 0000  hco=0075d
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00e6  00d9 0000  082c 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 hco=075d pco=fe6804ec  hconext=0084c hptda=050c f=16  pid=0019 e:ipfc.exe
 hco=084c pco=fe680997  hconext=006de hptda=04c6 f=16  pid=0018 c:epm.exe
 hco=06de pco=fe680271  hconext=00660 hptda=049c f=16  pid=0013 d:ibmavsd.exe
 hco=0660 pco=fe67fffb  hconext=00497 hptda=0410 f=16  pid=0012 c:pmdraw.exe
 hco=0497 pco=fe67f70e  hconext=0036b hptda=0420 f=16  pid=0010 c:cmd.exe
 hco=036b pco=fe67f132  hconext=00327 hptda=0380 f=16  pid=000d c:pulse.exe
 hco=0327 pco=fe67efde  hconext=001a0 hptda=036c f=16  pid=000c c:dinfo.exe
 hco=01a0 pco=fe67e83b  hconext=002c6 hptda=034e f=16  pid=000b c:mrfile32.exe
 hco=02c6 pco=fe67edf9  hconext=0014c hptda=0317 f=16  pid=000a c:pmdiary.exe
 hco=014c pco=fe67e697  hconext=000a2 hptda=02a6 f=16  pid=0006 c:pmshell.exe
 hco=00a2 pco=fe67e345  hconext=00033 hptda=0205 f=16  pid=0004 c:harderr.exe
 hco=0033 pco=fe67e11a  hconext=00029 hptda=0091 f=16  pid=0003 c:pmshell.exe
 hco=0029 pco=fe67e0e8  hconext=00000 hptda=0169 f=16  pid=0002 c:logdaem.exe


>>> We can tell immediately that this is shared arena global data from the
>>> presence of hco= in the arena record.  The hco is the handle to the
>>> context record.  These record the hptda of the process that is accessing
>>> shared global data.  Each of the VMCOs, that's sharing the same object
>>> is chained in a single linked list from the arena record.
>>> The description to the right of each VMCO is derived from the hptda object.

>>> Note: the .MC will format a VMCO.  Under the Dump Formatter the VMCO chain is
>>> not run to completion so we must run the chain manually by using
>>> hconext= field as the VMCO chain pointer.

>>> The 'own' and 'hmte' fields being equal indicate that the object is
>>> part of DOSCALL1.DLL.  We can check out which object in DOSCALL1 using
>>> .lmo

#.lmo e5
hmte=00e5 pmte=%fe72dfac mflags=8498b594 c:\os2\dll\doscall1.dll
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 00001354 1a010000 80009025 00000001 00000002 00eb d00e r-x shr alias iopl
0002 0000ced0 1a020000 80002025 00000003 0000000d 00ea d017 r-x shr big
0003 00001928 1a030000 80001025 00000010 00000002 00e9 d01f r-x shr alias
0004 000002ce 1a040000 80001025 00000012 00000001 00e8 d027 r-x shr alias
0005 000054f8 1a050000 8000d025 00000013 00000006 00e7 d02e r-x shr alias conf iopl
0006 00000280 1a060000 80001023 00000019 00000001 00e6 d037 rw- shr alias
0007 00001b40 1a070000 80001003 0000001a 00000002 0000 d03f rw- alias

>>> hob e6 is load module object 6 of doscall1.dll.  Furthermore it is a
>>> read/write object.  We can illustrate this by looking at the page table
>>> entries for two of the processes that are accessing hob e6.

# .s8
Current slot number: 0008
# .p8 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008# 0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
# dp %1a060000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a060000* 012fe  frame=00e1e  0    0  D  u        U  W  P  pageable
```

```
%1a060000  00e1e  frame=00e1e  0    0  D  u        U  W  P  pageable
# .s1f
Current slot number: 001f
# .p1f
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001f# 0006 0003 0006 0001 rdy 062f 7b7e4000 7b9c60d0 7b9ab1f8 1eb8 11 PMSHL32
# dp %1a060000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a060000* 0093e  frame=00e1e  0    0  c  A        U  W  P  pageable
%1a060000  00e1e  frame=00e1e  0    0  c  A        U  W  P  pageable
#
>>> As expected the same page frame (00e1e) is being referenced.

>>> Note also: frame e1e of slot 8 is dirty (Dc=D) and unaccessed (Au=u)
>>> while in slot 1f it is clean and accessed.  This tends to suggest
>>> that frame e1e and therefore page %1a060000 was most recently
>>> updated by slot 8 and read by slot 1f before the update took place.

>>> Shared Arena, Instance Data:
>>> ---------------------------

#.moc 5ef
*har     par      cpg        va    flg next prev link hash hob   hal
 01c6 %feb2170e 00000010 %1a890000 139 01c5 01c7 0000 0000 05ef 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 05ef  01c6 04e8  0024 050c 0131  0000 00  00 00 00 priv 0019 e:ipfc.exe
 04e8  01c6 02ec  0024 04c6 0131  0000 00  00 00 00 priv 0018 c:epm.exe
 02ec  01c6 0457  0024 049c 0131  0000 00  00 00 00 priv 0013 d:ibmavsd.exe
 0457  01c6 0432  0024 0410 0131  0000 00  00 00 00 priv 0012 c:pmdraw.exe
 0432  01c6 03d6  0024 0420 0131  0000 00  00 00 00 priv 0010 c:cmd.exe
 03d6  01c6 0390  0024 0380 0131  0000 00  00 00 00 priv 000d c:pulse.exe
 0390  01c6 03c8  0024 036c 0131  0000 00  00 00 00 priv 000c c:dinfo.exe
 03c8  01c6 03a7  0024 034e 0131  0000 00  00 00 00 priv 000b c:mrfile32.exe
 03a7  01c6 02c7  0024 0317 0131  0000 00  00 00 00 priv 000a c:pmdiary.exe
 02c7  01c6 0112  0024 02a6 0131  0000 00  00 00 00 priv 0006 c:pmshell.exe
 0112  01c6 0000  0024 0091 0131  0000 00  00 00 00 priv 0003 c:pmshell.exe

>>> Object 5ef is an example of shared arena instance data.  Each instance
>>> of the object has its own object record (VMOB), but they all share the
>>> same arena record.  Each of these VMOBs is chained from the 'hob'
>>> field of the arena record via their 'hobnxt' field.
>>> The VMOBs appear as private arena objects, but the arena record
>>> does not point to a specific hptda, which distinguishes this case as
>>> shared instance data.

>>> As would be expected with shared instance data the owners would differ
>>> for each instance object.  They are in-fact the hptda's for each owner.
>>> The hmte's we would expect to be common to all the VMOBs.

#.mo 131
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0131  %fe860978  8000 ffa6 0127  0000 00  00 00 00 mte       c:display.dll

>>> So, %1a890000 has been allocated by display.dll

>>> Again we can illustrate that we are really looking at instance data by
>>> examining the page tables of two examples:
```

```
.s8
Current slot number: 0008
# .p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0008# 0003 0001 0003 0001 rdy 061f 7b7b6000 7b9c484c 7b9a8a70 1eb8 01 PMSHL32
# dp %1a890000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a890000* 01291  frame=01066  0    0  D  u        s  W  P  pageable
%1a890000  01066  frame=01066  0    0  D  u        s  W  P  pageable
# .s1f
Current slot number: 001f
# .p1f
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001f# 0006 0003 0006 0001 rdy 062f 7b7e4000 7b9c60d0 7b9ab1f8 1eb8 11 PMSHL32
# dp %1a890000
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a890000* 0093c  frame=008e4  0    0  D  u        s  W  P  pageable
%1a890000  008e4  frame=008e4  0    0  D  u        s  W  P  pageable
#
>>> In Pid 3, %1a890000 translates to physical address %01066000, but
>>> in Pid 6, %1a890000 translates to physical address %008e4000.
```

Finding Who Owns Memory: Having examined various types of arena, object and context record we now turn our attention to a more commonly asked question: "Who owns a particular location of memory?". To answer this we need to explore the match parameter of the .MA command.

The .MAM command search for arena records that encompass a given address:

```
# .mam %123456
 har     par      cpg        va    flg next prev link hash hob   hal
 008c %feb1fc12 00000080 %00110000 169 00f1 0073 0000 0000 008f 0000 hptda=0091
 023b %feb2211c 00001000 %000c0000 1e9 0238 0266 0000 0000 029f 0000 hptda=02a6
 02fc %feb231b2 00000010 %00120000 169 02fd 02a2 0000 0000 0318 0000 hptda=036c
 0306 %feb2328e 00000010 %00120000 169 0309 0321 0000 0000 03ba 0000 hptda=0380
 0312 %feb23396 00000010 %00120000 169 0313 0311 0000 0000 03cd 0000 hptda=034e
 032f %feb23614 00000080 %00110000 169 034b 02fa 0000 0000 03e4 0000 hptda=0317
 0393 %feb23eac 00000010 %00120000 169 0394 038d 0000 0000 0452 0000 hptda=0410
 0412 %feb24996 00000010 %00120000 169 0414 0411 0000 0000 04ef 0000 hptda=04c6
 0517 %feb26004 00000010 %00120000 169 0519 0516 0000 0000 05f7 0000 hptda=050c


>>> Dump Formatter hard-wires the 'A' parameter so .mam=/.mama (or .maam)
>>> Kernel Debugger needs 'A' explicitly if all contexts are to be searched.
>>> This only affects results from searching private arena addresses.


>>> We can also add the 'C' parameter to chain through the related VMOBs
>>> and VMCOs at the same time.

# .mamc %123456

*har     par      cpg        va    flg next prev link hash hob   hal
 008c %feb1fc12 00000080 %00110000 169 00f1 0073 0000 0000 008f 0000 hptda=0091
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 008f  008c 0000  422c 0091 01c0  0000 00  00 00 00 priv 0003 c:pmshell.exe
```

```
*har     par      cpg        va    flg next prev link hash hob   hal
 023b %feb2211c 00001000 %000c0000 1e9 0238 0266 0000 0000 029f 0000 hptda=02a6
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 029f  023b 0000  423c 02a6 01d7  0000 00  00 00 00 priv 0006 c:pmshell.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 02fc %feb231b2 00000010 %00120000 169 02fd 02a2 0000 0000 0318 0000 hptda=036c
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0318  02fc 0000  422c 036c 0371  0000 00  00 00 00 priv 000c c:dinfo.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0306 %feb2328e 00000010 %00120000 169 0309 0321 0000 0000 03ba 0000 hptda=0380
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03ba  0306 0000  422c 0380 035c  0000 00  00 00 00 priv 000d c:pulse.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0312 %feb23396 00000010 %00120000 169 0313 0311 0000 0000 03cd 0000 hptda=034e
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03cd  0312 0000  422c 034e 0354  0000 00  00 00 00 priv 000b c:mrfile32.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 032f %feb23614 00000080 %00110000 169 034b 02fa 0000 0000 03e4 0000 hptda=0317
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 03e4  032f 0000  422c 0317 01c0  0000 00  00 00 00 priv 000a c:pmdiary.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0393 %feb23eac 00000010 %00120000 169 0394 038d 0000 0000 0452 0000 hptda=0410
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0452  0393 0000  402c 0410 ff3e  0000 00  00 00 00 priv 0012 c:pmdraw.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0412 %feb24996 00000010 %00120000 169 0414 0411 0000 0000 04ef 0000 hptda=04c6
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 04ef  0412 0000  422c 04c6 04f3  0000 00  00 00 00 priv 0018 c:epm.exe

*har     par      cpg        va    flg next prev link hash hob   hal
 0517 %feb26004 00000010 %00120000 169 0519 0516 0000 0000 05f7 0000 hptda=050c
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 05f7  0517 0000  422c 050c 05de  0000 00  00 00 00 priv 0019 e:ipfc.exe

>>> .MAMC is such a frequently used command that it is made the default
>>> specification for .M
>>> Further more, .M will take the default CS:EIP as the match
>>> address if no address is given.

>>> Suppose we wish to find out what code is being currently executed in
>>> in slot 39...

# .s 39
Current slot number: 0039
# .p #
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0039  0019 0010 0019 0001 rdy 061f 7b818000 7b9ca230 7b9adea8 1f0c 12 IPFC
# .r
eax=00000000 ebx=00307d90 ecx=00320000 edx=00000000 esi=00001000 edi=00001000
eip=1a022240 esp=0004d098 ebp=0004d0b4 iopl=2 -- -- -- nv up ei pl nz na pe nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001d6000
005b:1a022240 83c418         add     esp,+18
```

```
# ln
No Symbols Found
# .m

*har     par      cpg        va    flg next prev link hash hob   hal
 00dd %feb20308 00000010 %1a020000 3d9 00dc 00de 0000 0000 00ea 0000  hco=007ba
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00ea  00dd 0000  0838 00e5 00e5  0000 00  00 00 00 shared    c:doscall1.dll
 hco=07ba pco=fe6806bd  hconext=00822 hptda=050c f=1c  pid=0019 e:ipfc.exe

>>> The current cs:eip for slot 39 is executing in doscall1.dll and
>>> has been called either directly or indirectly by ipfc.exe
```

Finally in this section we answer, ″What is the hptda given the PTDA address?″

This required the use of the match parameter with .MO.

.MOM is more restrictive and .MAM. It will only return a result if the supplied address is a precise match for the beginning of a pseudo-object. Since the PTDA is a pseudo-object we can use its address with .MOM:

```
# .p 2a
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 002a  0006 0003 0006 000c blk 021f 7b7fa000 7b9c60d0 7b9ac4e0 1eac 11 PMSHL32
# .mom %7b9c60d0
 hob       va     flgs own  hmte  sown,cnt lt st xf
 02a6  %7b9c60d0  8000 ffcb 0000  0000 00  00 00 00 ptda 0006 c:pmshell.exe

>>> The hptda for Pid 6 is therefore 2a6.
```

14.1.2.2 How to Correlate Named Memory with its Address

Here′s how to locate named shared memory, answer who′s sharing it and whether a particular address in the shared arena is named.

Named memory is managed using a RMP (Record Management Package). This is a generalized kernel facility for managing global data of variable length. The *RMP* facility provides allocation, deletion, add and find services. Each *RMP* user references his *RMP* structure using a handle. The *RMP* itself is limited to a maximum of 64K.

```
>>> First locate the handle of named shared memory′s RMP:

##dw sharermpstruc l2
0400:00004506  0004 0178
                 || |
                 || ....... Selector for RMP segment
                 .......... Flags xxxxxxxx
```

```
                                  .......1 = Segment busy
                                  ......1. = Somebody's waiting
                                  .....1.. = Segment allocated

>>> The RMP handle is used as the BlockID (by ProcBlock) for serializing
>>> RMP manipulations.
>>>
>>> Now display the named memory RMP segment:
##db 178:0
0178:00000000 00 06 a2 03 5e 02 5e 02-03 00 00 00 00 00 00 00 .."·^·^.........
0178:00000010 83 ff 00 00 11 00 00 00-9b 06 01 00 44 4f 53 5c ............DOS\
0178:00000020 43 44 49 42 00 13 00 88-01 47 bd 04 00 50 4d 44 CDIB.....G=..PMD
0178:00000030 52 41 47 2e 4d 45 4d 00-08 00 9f 00 88 01 01 00 RAG.MEM.........
0178:00000040 14 00 91 01 d7 bc 02 00-53 4d 47 5c 53 47 54 49 ....W<..SMG\SGTI
0178:00000050 54 4c 45 00 08 00 9f 00-91 01 01 00 12 00 93 01 TLE.............
0178:00000060 b7 bc 02 00 42 56 53 5c-42 56 53 30 30 00 08 00 7<..BVS\BVS00...
0178:00000070 9f 00 93 01 01 00 12 00-a8 01 8f bc 01 00 42 56 ........(..<..BV
##d
0178:00000080 53 5c 42 56 53 30 31 00-08 00 9f 00 a8 01 01 00 S\BVS01.....(...
0178:00000090 12 00 ab 01 67 bc 01 00-42 56 53 5c 42 56 53 30 ..+.g<..BVS\BVS0
0178:000000a0 33 00 08 00 9f 00 ab 01-01 00 18 00 c4 01 3f bc 3.....+.....D.?<
0178:000000b0 02 00 53 4d 47 5c 50 4d-48 44 45 52 52 2e 44 41 ..SMG\PMHDERR.DA
0178:000000c0 54 00 08 00 af 01 c4 01-01 00 08 00 af 01 91 01 T.../.D...../...
0178:000000d0 01 00 08 00 af 01 93 01-01 00 08 00 9f 00 c4 01 ..../.........D.
0178:000000e0 01 00 12 00 43 02 2f a0-02 00 42 56 53 5c 42 56 ....C./ ..BVS\BV
0178:000000f0 53 31 30 00 08 00 9f 00-43 02 01 00 08 00 4c 02 S10.....C.....L.
##d
0178:00000100 43 02 01 00 08 00 58 02-88 01 01 00 17 00 9c 02 C.....X.........
0178:00000110 cf 9f 01 00 50 4d 57 50-5c 43 4c 41 53 53 2e 54 O...PMWP\CLASS.T
0178:00000120 42 4c 00 08 00 58 02 9c-02 01 00 08 00 fa 02 88 BL...X.......z..
0178:00000130 01 01 00 18 00 13 04 e7-9e 01 00 45 50 4d 5c 45 .......g...EPM\E
0178:00000140 54 4b 45 36 30 30 2e 45-50 4d 00 08 00 fa 02 13 TKE600.EPM...z..
0178:00000150 04 01 00 10 00 15 04 ff-9d 01 00 45 50 4d 47 4e ...........EPMGN
0178:00000160 4c 53 00 08 00 fa 02 15-04 01 00 18 00 03 04 e7 LS...z.........g
0178:00000170 9d 01 00 31 35 35 30 32-33 33 33 5c 45 50 4d 2e ...15502333\EPM.
##d
0178:00000180 45 58 00 08 00 fa 02 03-04 01 00 08 00 56 03 88 EX...z.......V..
0178:00000190 01 01 00 1b 00 f6 02 bf-9d 01 00 31 35 33 39 34 .....v.?...15394
0178:000001a0 39 31 34 5c 45 33 45 4d-55 4c 2e 45 58 00 08 00 914\E3EMUL.EX...
0178:000001b0 fa 02 f6 02 01 00 12 00-90 03 8f 9d 03 00 42 56 z.v...........BV
0178:000001c0 53 5c 42 56 53 31 34 00-08 00 9f 00 90 03 01 00 S\BVS14.........
0178:000001d0 08 00 32 04 90 03 01 00-12 00 43 04 6f 9d 03 00 ..2.......C.o...
0178:000001e0 42 56 53 5c 42 56 53 31-35 00 08 00 9f 00 43 04 BVS\BVS15.....C.
0178:000001f0 01 00 08 00 48 04 43 04-01 00 08 00 57 04 90 03 ....H.C.....W...
##d
0178:00000200 01 00 08 00 62 04 43 04-01 00 12 00 87 04 4f 9d ....b.C.......O.
0178:00000210 03 00 42 56 53 5c 42 56-53 31 36 00 08 00 9f 00 ..BVS\BVS16.....
0178:00000220 87 04 01 00 08 00 8b 04-87 04 01 00 08 00 99 04 ................
0178:00000230 87 04 01 00 12 00 0a 03-ef 9c 03 00 42 56 53 5c ........o...BVS\
0178:00000240 42 56 53 31 38 00 08 00-9f 00 0a 03 01 00 08 00 BVS18...........
0178:00000250 d6 04 0a 03 01 00 08 00-bb 04 0a 03 01 00 a2 83 V.......;.....".
0178:00000260 00 00 5e 02 00 00 9a 83-00 00 66 02 00 00 00 00 ..^.......f.....
0178:00000270 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................

>>> The first 20 bytes form the RMP header, the remainder is a series of
>>> variable length records.
>>> Examining the header first we have:
>>> +00 00 06       = total size of segment (0600)
>>> +02 a2 03       = amount of free space (03a2)
```

```
>>> +04 5e 02       = link to first free block (025e)
>>> +06 5e 02       = start of last free block (025e)
>>> +08 03 00 00 00 = heap handle (0003 is kernel heap handle from which
>>>                   RMP is alloc'd)
>>> +0c 00 00 00 00 = PG alloc/realloc flags
>>> +10 83 ff       = hobowner (handle of user of this RMP is ff83)
>>> +12 00 00       = hobmte (hmte of user of this RMP. It's the kernel so 0000)

>>> Check out the owner of this RMP

##.mo ff83
ff83 mshrmp

>>> 'mshrmp' is named shared memory management

>>> Records follow the header. They are prefixed by a word length that includes
>>> 2 bytes for the length field itself. If the record is free then the high
>>> order bit of the length is set. The data within the record is private to
>>> the owner.

>>> The first record in this RMP is:

                            ....... length 0011
                            || ||
>>> 0178:00000010 .. .. .. .. 11 00 00 00-9b 06 01 00 44 4f 53 5c
>>> 0178:00000020 43 44 49 42 00 .. .. ..

>>> The second record in this RMP is:

>>> 0178:00000020 .. .. .. .. .. 13 00 88-01 47 bd 04 00 50 4d 44
>>> 0178:00000030 52 41 47 2e 4d 45 4d 00-.. .. .. .. .. .. .. ..

>>> Named shared memory management uses two forms of RMP record:
>>>   Global - to keep the name, handle, selector and total reference count
>>>   Local  - One for each process sharing the named memory. Contains
>>>            hptda, hob and ref count for within the given process.
>>>
>>> Breaking down record 2 we have:

>>> +00 0013        length of record
>>> +02 0188        hob of shared object
>>> +04 bd47        selector of shared object
>>> +06 0004        reference count
>>> +08 PMDRAG.MEM  name (with \SHAREMEM\ prefix omitted) and terminated with
>>>                 a null byte (that is, it's an ASCIIZ string)

>>> Check out hob 188
##.moc 188

*har     par      cpg        va    flg next prev link hash hob   hal
 0140 %fecffb8a 00000010 %17a80000 369 000f 0141 0000 0000 0188 0000  hco=00291
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0188  0140 0000  482c ff82 017d  0000 00  00 00 00 mshare
 hco=00291 pco=fe85dcf0  hconext=003c8 hptda=02fa f=16  pid=0013 c:epm.exe
 hco=003c8 pco=fe85e303  hconext=00246 hptda=0356 f=16  pid=0009 c:mrfile32.exe
 hco=00246 pco=fe85db79  hconext=00070 hptda=0258 f=16  pid=0005 c:pmshell.exe
 hco=00070 pco=fe85d24b  hconext=00000 hptda=009f f=17  pid=0002 c:pmshell.exe
```

```
>>> We see 4 owners in accordance with the reference count
>>> note: the owner of the object is 'mshare'

>>> Check out the selector in record 2:

##dl bd47
bd47  Data    Bas=17a80000 Lim=00000067 DPL=3 P  RW    A

>>> In this case it's within the compatibility region so could have used the
>>> CRMA to get %17a80000 directly

>>> The processes sharing the named storage may be obtained directly from local
>>> records in the RMP. A local record is of the following form:
>>>
>>> +00 word  - length of record (always 0008)
>>> +02 word  - hptda of user
>>> +04 word  - handle of shared memory object
>>> +06 word  - reference count for this ptda.

>>> Scanning through the RMP (for object 0188) we find the following local
>>> records:

>>> 0178:00000030 .. .. .. .. .. .. .. ..-08 00 9f 00 88 01 01 00

>>> 0178:00000100 .. .. .. .. 08 00 58 02-88 01 01 00 .. .. .. ..

>>> 0178:00000120 .. .. .. .. .. .. .. ..-.. .. .. 08 00 fa 02 88
>>> 0178:00000130 01 01 00 .. .. .. .. ..-.. .. .. .. .. .. .. ..

>>> 0178:00000180 .. .. .. .. .. .. .. ..-.. .. .. 08 00 56 03 88
>>> 0178:00000190 01 01 00 .. .. .. .. ..-.. .. .. .. .. .. .. ..

>>> This confirms what was shown in the .mo 188 display, but we have in addition
>>> the reference count for each process.


>>> Finally, we can cut the cake a different way by asking what is all the
>>> named storage being referenced by a particular process. For example
>>> EPM.
>>> Start by finding its slot nos.

.p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 .
 .
 .
 0027  0013 0002 0013 0001 blk 0200 7b974000 7bb460d0 7bb2bf24 1eb8 12 epm
 0020  0013 0002 0013 0002 blk 0200 7b966000 7bb460d0 7bb2b338 1ecc 12 epm
 .
 .

>>> EPM's pPTDA is %7bb460d0. Now find the hptda of this PTDA

##.mom %7bb460d0
 hob       va     flgs own  hmte  sown,cnt lt st xf
 02fa  %7bb460d0  8000 ffcb 035b  0000 00  00 00 00 ptda 0013 c:epm.exe
```

```
>>> Answer: 2fa.
>>> Now look through the RMP for local records that begin:
>>> 08 00 fa 02

>>> 0178:00000120 .. .. .. .. .. .. .. ..-.. .. .. 08 00 fa 02 88
>>> 0178:00000130 01 01 00 .. .. .. .. ..-.. .. .. .. .. .. .. ..

>>> 0178:00000140 .. .. .. .. .. .. .. ..-.. .. .. 08 00 fa 02 13
>>> 0178:00000150 04 01 00 .. .. .. .. ..-.. .. .. .. .. .. .. ..

>>> 0178:00000160 .. .. .. 08 00 fa 02 15-04 01 00 .. .. .. .. ..

>>> 0178:00000180 .. .. .. 08 00 fa 02 03-04 01 00 .. .. .. .. ..

>>> 0178:000001a0 .. .. .. .. .. .. .. ..-.. .. .. .. .. .. 08 00
>>> 0178:000001b0 fa 02 f6 02 01 00 .. ..-.. .. .. .. .. .. .. ..

>>> So, 5 named objects, with hobs=0188, 0413, 0415, 0403 and 02f6

>>> Scanning the the RMP for Global records for these objects will reveal
>>> their names: PMDRAG.MEM, EPM\ETKE600.EPM, EPMGNLS, 15502333\EPM.EXE,
>>> 15394914\E3EMUL.EX.

>>> issuing .mo against each of the hobs will reveal whether these are shared
>>> and with whom:

.moc 188

*har     par      cpg        va    flg next prev link hash hob   hal
 0140 %fecffb8a 00000010 %17a80000 369 000f 0141 0000 0000 0188 0000  hco=00291
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0188  0140 0000  482c ff82 017d  0000 00  00 00 00 mshare
 hco=00291 pco=fe85dcf0  hconext=003c8 hptda=02fa f=16  pid=0013 c:epm.exe
 hco=003c8 pco=fe85e303  hconext=00246 hptda=0356 f=16  pid=0009 c:mrfile32.exe
 hco=00246 pco=fe85db79  hconext=00070 hptda=0258 f=16  pid=0005 c:pmshell.exe
 hco=00070 pco=fe85d24b  hconext=00000 hptda=009f f=17  pid=0002 c:pmshell.exe
##.moc 413

*har     par      cpg        va    flg next prev link hash hob   hal
 0367 %fed02ae4 00000010 %13dc0000 369 025e 0372 0000 0000 0413 0000  hco=00293
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0413  0367 0000  4a2c ff82 030c  0000 00  00 00 00 mshare
 hco=00293 pco=fe85dcfa  hconext=00000 hptda=02fa f=17  pid=0013 c:epm.exe
##.moc 415

*har     par      cpg        va    flg next prev link hash hob   hal
 0307 %fed022a4 00000010 %13bf0000 369 037e 02c2 0000 0000 0415 0000  hco=003cc
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0415  0307 0000  4a2c ff82 0407  0000 00  00 00 00 mshare
 hco=003cc pco=fe85e317  hconext=00000 hptda=02fa f=17  pid=0013 c:epm.exe
##.moc 403

*har     par      cpg        va    flg next prev link hash hob   hal
 02c2 %fed01cb6 00000030 %13bc0000 369 0307 0262 0000 0000 0403 0000  hco=00309
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0403  02c2 0000  4a2c ff82 0407  0000 00  00 00 00 mshare
 hco=00309 pco=fe85df48  hconext=00000 hptda=02fa f=17  pid=0013 c:epm.exe
##.moc 2f6
```

```
*har     par      cpg        va     flg next prev link hash hob   hal
 0273 %fed015ec 00000010 %13b70000 369 027c 02ed 0000 0000 02f6 0000  hco=00308
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 02f6  0273 0000  4a2c ff82 0407  0000 00  00 00 00 mshare
 hco=00308 pco=fe85df43  hconext=00000 hptda=02fa f=17  pid=0013 c:epm.exe
##
```

```
>>> We see that except for hob=188 all the others are for the private use of
>>> EPM.
```

14.1.2.3 How Memory Aliasing Works

Aliasing is a facility in virtual memory management whereby one or more pages of a memory object may be referenced from an alternative virtual address, possibly from a different process or arena and possibly with different read/write/execute characteristics. It is used extensively by device drivers debugging applications and VDMs.

This example shows how aliasing is represented in the system for a debugging application and how shared storage becomes privatized. There are many ways of creating aliases. The application in this example is IPMD, which uses DosDebug function MapWRAlias to alias the debugee's storage and DosCreateCSAlias to map a code selector to one of his own data segments.

We introduce the memory alias record (VMAL) and the `.ML` command.

```
>>> For reference list the thread slots in the system...

.p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 .
 .
 001c  0004 0002 0004 0001 blk 0200 7b95e000 7bb45078 7bb2ac68      10 cmd
 .
 .
 0020  0008 0002 0008 0002 blk 0200 7b966000 7bb460d0 7bb2b338      12 mrfile32
 002b# 000b 0002 000b 0001 blk 0200 7b97c000 7bb468fc 7bb2c5f4 1eb8 14 ipmd
 002a  000b 0002 000b 0002 blk 0200 7b97a000 7bb468fc 7bb2c440 1eb8 14 ipmd
 002c  000d 0002 000d 0001 blk 0200 7b97e000 7bb47954 7bb2c7a8 1eb8 16 cmd
 002d  000c 0002 000c 0001 blk 0200 7b980000 7bb47128 7bb2c95c 1e98 15 dpmlines
 002e  000e 0002 000e 0001 blk 0300 7b982000 7bb48180 7bb2cb10 1eb8 17 epm
 002f  000e 0002 000e 0002 blk 0300 7b984000 7bb48180 7bb2ccc4 1ecc 17 epm


>>> Now list all the busy alias records:

##.ml
 hal=0001 pal=%ffe61020  har=00b8  hptda=009f  pgoff=00000  f=001
 hal=0002 pal=%ffe61028  har=00b9  hptda=009f  pgoff=00000  f=001
 hal=0003 pal=%ffe61030  har=001b  hptda=009f  pgoff=00000  f=001
 hal=0004 pal=%ffe61038  har=0183  cs=00e6 ds=d446 cref=001 f=13
 hal=0005 pal=%ffe61040  har=0199  hptda=009f  pgoff=00006  f=001
 hal=0006 pal=%ffe61048  har=01b8  hptda=009f  pgoff=00000  f=021
 hal=0007 pal=%ffe61050  har=01b9  hptda=009f  pgoff=00000  f=021
```

```
 hal=0008 pal=%ffe61058  har=01ba  hptda=009f  pgoff=00000  f=021
 hal=0009 pal=%ffe61060  har=01e7  hptda=009f  pgoff=00000  f=001
 hal=000a pal=%ffe61068  har=0208  cs=0056 ds=d446 cref=001 f=13
 hal=000b pal=%ffe61070  har=020b  cs=0056 ds=d446 cref=001 f=13
 hal=000c pal=%ffe61078  har=026f  cs=007e ds=d446 cref=001 f=13
 hal=000d pal=%ffe61080  har=02bf  cs=00ae ds=d446 cref=001 f=13
 hal=000e pal=%ffe61088  har=02df  cs=01ae ds=0077 cref=001 f=13
 hal=000f pal=%ffe61090  har=0305  hptda=0389  pgoff=00000  f=049
 hal=0010 pal=%ffe61098  har=0306  hptda=0389  pgoff=00000  f=049
 hal=0011 pal=%ffe610a0  har=030e  cs=0056 ds=d446 cref=001 f=13
 hal=0012 pal=%ffe610a8  har=0323  cs=0056 ds=d446 cref=001 f=13
 hal=0013 pal=%ffe610b0  har=032e  cs=007e ds=d446 cref=001 f=13

>>> hal f & 10 have f=049 = 0000 0100 1001
>>>                                  |   |  |... Busy
>>>                                  |   |
>>>                                  |   ....... Debug
>>>                                  |.......... Privatized
>>>
>>> har=305 is an alias for a linear address in hptda=389
>>> similarly har=306 is an alias for a linear address in hptda=389
>>> Now look closer at hal=f.

##.mlc f
>>> chaining doesn't always work so..

##.mac 305

*har     par      cpg        va    flg next prev link hash hob   hal
 0305 %fed02278 00000010 %00520000 169 0307 0304 00b5 0000 00c1 000f hptda=031a
 hal=000f pal=%ffe61090  har=0305  hptda=0389  pgoff=00000  f=049
 har     par      cpg        va    flg next prev link hash hob   hal
 00b5 %fecfef98 00000010 %1a030000 3d9 00b4 00b6 0000 0000 00c1 0000  hco=00502
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00c1  0305 0000  1838 00bd 00bd  0000 00  00 00 00 shared    c:doscall1.dll
 hco=00502 pco=fe85e925  hconext=00473 hptda=03cb f=1c  pid=000e c:epm.exe
 hco=00473 pco=fe85e65a  hconext=00455 hptda=03b9 f=1c  pid=000d c:cmd.exe
 hco=00455 pco=fe85e5c4  hconext=0045a hptda=0389 f=9c  pid=000c d:dpmlines.exe
 hco=0045a pco=fe85e5dd  hconext=00283 hptda=031a f=1c  pid=000b d:ipmd.exe
 hco=00283 pco=fe85dcaa  hconext=0014a hptda=02d6 f=1c  pid=0008 c:mrfile32.exe
 hco=0014a pco=fe85d68d  hconext=00133 hptda=0257 f=1c  pid=0005 c:pmshell.exe
 hco=00133 pco=fe85d61a  hconext=00092 hptda=0248 f=1c  pid=0004 c:cmd.exe
 hco=00092 pco=fe85d2f5  hconext=00020 hptda=01ae f=1c  pid=0003 c:harderr.exe
 hco=00020 pco=fe85d0bb  hconext=00000 hptda=009f f=1c  pid=0002 c:pmshell.exe


>>> Ignoring hco=455 for the moment. har=b5 represents linear address range
>>> %1a030000 in the shared arena. This is in fact a shared object (hob=1c)
>>> which is being accessed by 9 different processes. The hco chain lists those
>>> processes that access this object. hob=1c is one of the objects in doscall1.dll

>>> Let's verify that %1a030000 is indeed that same data in each of the contexts.
>>> hco=133 is for Pid=4, slot=1c, cmd.exe(1)
##.s 1c
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 00236  frame=00236  2    0  D  A        U  W  P  resident
%1a030000  00a22  frame=00a22  0    0  c  u        U  r  P  pageable
%1a031000  00a1c  vp id=00320  0    0  c  u        U  r  n  pageable
```

```
>>> %1a030000 equates to real address %%a22000
>>> hco=473 is for Pid=d, slot=2c, cmd.exe (2)
##.s 2c
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 00c3c  frame=00c3c  2    0  D  A        U  W  P  resident
%1a030000  00a22  frame=00a22  0    0  c  u        U  r  P  pageable
%1a031000  00a1c  frame=00a1c  0    0  c  u        U  r  P  pageable

>>> %1a030000 equates to real address %%a22000
>>> hco=502 is for Pid=e, slot=2e, epm.exe
##.s 2e
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 00418  frame=00418  2    0  D  A        U  W  P  resident
%1a030000  00a22  frame=00a22  0    0  c  u        U  r  P  pageable
%1a031000  00a1c  frame=00a1c  0    0  c  u        U  r  P  pageable

>>> In each of these cases looked so far, linear address %1a030000 is
>>> mapped to the same real address %%a22000.
>>> Now examine the hco flags:  9c = 1001 1100
>>>                                  |  | ||.... User
>>>                                  |  | |..... Executable
>>>                                  |  |....... Read
>>>                                  |.......... Privatized

>>> now turn our attention to har=305, hal=f, address %520000 and hco=455.

>>> hco=455 has the additional Privatized flag set.
>>> Now look at the PTE for this storage in slot=2d, Pid=c, dpmlines.exe
##.s 2d
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 005ab  frame=005ab  2    0  D  A        U  W  P  resident
%1a030000  00bda  frame=00bda  0    0  c  u        U  r  P  pageable
%1a031000         vp id=01616  0    0  c  u        U  r  n  pageable

>>> This is no-longer the same storage as in the other contexts. After DPMLINES
>>> was loaded, IPMD created an alias to object 1c reference by DMPLINES.
>>> The loader/memory had to make a private copy to protect the integrity
>>> of other contexts who were sharing the same object. Having privatized
>>> this object for the one context, the loader will not share it with
>>> other contexts.
>>> If we hadn't started with the alias record we could have done a .ml now
>>> and looked for the records which referenced hptda=389. As it happens we
>>> know already that har=305 is an alias of har=b5. We can check this out
>>> by looking at the page tables for %520000 in hptda=031a (pid=b, slot=2c)

##.s 2c
##dp %520000
##.i %520000
##dp %520000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00520000* 00390  frame=00390  2    0  D  A        U  W  P  resident
%00520000  00bda  frame=00bda  0    0  c  u        U  W  P  pageable
%00521000         vp id=01616  0    0  c  u        U  W  n  pageable
```

```
>>> %520000 is %%bda000 which is the same real address as %1a030000 in slot=
>>> 2d. We had to page in %520000 so we should check %1a030000 in slot=2d
>>> again, in case it was discarded.

##.s 2d
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 005ab  frame=005ab  2    0  D  A        U  W  P  resident
%1a030000  00bda  frame=00bda  0    0  c  u        U  r  P  pageable
%1a031000         vp id=01616  0    0  c  u        U  r  n  pageable

> ... and it is the same.
> While we are at it, lets check %1a030000 in slot=2c (IPMD)

##.s 2c
##dp %1a030000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%1a030000* 00611  frame=00611  2    0  D  A        U  W  P  resident
%1a030000  00a22  frame=00a22  0    0  c  u        U  r  P  pageable
%1a031000  00a1c  vp id=00320  0    0  c  u        U  r  n  pageable

>>> ... and yes as expected IPMD is referencing the shared %1a030000, in
>>> fact he is referencing both copies.

>>> Now let's look at some of the other aliases set up by IPMD
##.mlc 10

*har     par      cpg        va    flg next prev link hash hob   hal
 0306 %fed0228e 00000010 %00540000 169 0308 0307 02ee 0000 0386 0010 hptda=031a
 hal=0010 pal=%ffe61098  har=0306  hptda=0389  pgoff=00000  f=049
 har     par      cpg        va    flg next prev link hash hob   hal
 02ee %fed0207e 00000010 %00010000 1d9 02ed 02f0 0000 0000 0386 ffff hptda=0389

>>> Note: hal=ffff for har=2ee. This is a special hal to indicate a
>>> privatized arena - there isn't a context record to put the privatized
>>> flag in as this was private arena, shared data.

>>> Check out the hptda Pids as we have forgotten who 31a and 389 are..
##.mo 31a
 hob       va     flgs own  hmte  sown,cnt lt st xf
 031a  %7bb468fc  8000 ffcb 02db  0000 00  00 00 00 ptda 000b d:ipmd.exe
##.mo 389
 hob       va     flgs own  hmte  sown,cnt lt st xf
 0389  %7bb47128  8000 ffcb 0000  0000 00  00 00 00 ptda 000c d:dpmlines.exe

>>> Now check the page tables in each processes to prove we are looking
>>> at the same thing...
##.ss 2b
##dp %540000 l1
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00540000* 00390  frame=00390  2    0  D  A        U  W  P  resident
%00540000         vp id=015f0  0    0  c  u        U  W  n  pageable
##.i %540000
##dp %540000 l1
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00540000* 00390  frame=00390  2    0  D  A        U  W  P  resident
%00540000  005d2  frame=005d2  0    0  c  u        U  W  P  pageable
##.ss 2d
##dp %10000 l2
```

```
##.i %10000
##dp %10000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00010000* 002b3  frame=002b3  2    0  D  A        U  W  P  resident
%00010000  005d2  frame=005d2  0    0  c  u        U  r  P  pageable
%00011000         vp id=015f1  0    0  c  u        U  r  n  pageable
##.ss 2b
##dp %540000 l1
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00540000* 00390  frame=00390  2    0  D  A        U  W  P  resident
%00540000  005d2  frame=005d2  0    0  c  u        U  W  P  pageable

>>> Finally look alias record hal=e. This is a CS Alias of a data
>>> segment within in the same process. The hal flags indicate:

>>> 13=0001 0011
>>>       |    ||... Busy (in use)
>>>       |    |.... CS Alias
>>>       |........ DS selector valid




##.mlc e

*har     par      cpg        va    flg next prev link hash hob   hal
 02df %fed01f34 00000010 %00350000 1c9 02e2 02de 029d 0000 031b 000e hptda=031a
 hal=000e pal=%ffe61088  har=02df  cs=01ae ds=0077 cref=001 f=13
 har     par      cpg        va    flg next prev link hash hob   hal
 029d %fed01988 00000010 %000e0000 179 02b9 0293 0000 0000 031b 0000 hptda=031a
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 031b  02df 0000  102c 031a 031e  0000 00  00 00 00 priv 000b d:ipmd.exe

>>> Check out the page tables...
##dp %350000 l2
##.i %350000
##dp %350000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00350000* 00625  frame=00625  2    0  D  A        U  W  P  resident
%00350000  00362  frame=00362  0    0  c  u        U  r  P  pageable
%00351000         vp id=01403  0    0  c  u        U  r  n  pageable
##dp %e0000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%000e0000* 00625  frame=00625  2    0  D  A        U  W  P  resident
%000e0000  00362  vp id=01402  0    0  c  u        U  W  n  pageable
%000e1000         vp id=01403  0    0  c  u        U  W  n  pageable
##.i %e0000
##dp %e0000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%000e0000* 00625  frame=00625  2    0  D  A        U  W  P  resident
%000e0000  00362  frame=00362  0    0  c  u        U  W  P  pageable
%000e1000         vp id=01403  0    0  c  u        U  W  n  pageable
##dp %350000 l2
 linaddr   frame   pteframe  state res Dc Au CD WT Us rW Pn state
%00350000* 00625  frame=00625  2    0  D  A        U  W  P  resident
%00350000  00362  frame=00362  0    0  c  u        U  r  P  pageable
%00351000         vp id=01403  0    0  c  u        U  r  n  pageable
```

```
>>> Check out the segment descriptors ....
##dl 1ae
01ae  Code    Bas=00350000 Lim=0000f15f DPL=2 P  RE C
##dl 77
0077  Data    Bas=000e0000 Lim=0000f15f DPL=3 P  RW    A
##

>>> Because of the existence of CS/DS alias, IPMD can effectively
>>> read, write and execute the same storage.  This is how IPMD is able
>>> to implement break points, by copying code, patching in INT 3
>>> instructions and executing the copied code; all from ring 3 privilege
>>> without compromising other processes or the system.
```

14.1.3 Exploring 32-bit Presentation Manager Under WARP

In this section we look specifically at the messaging function within Presentation Manager (PM).

Sending and receiving messages lies at the heart of how PM applications communicate with each other and the system. Messages may be sent synchronously and posted asynchronously. The mismanagement of messages by an application leads frequently to the ′bad application′ pop-up dialog. In extreme cases deadlocks result.

This topic applies to OS/2 V3, which introduced the 32-bit version of Presentation Manager. The previous 16-bit environment has analogous concepts which are briefly explored through a final worked example.

Prerequisites to any PM debugging requires the following:

- Availability of the symbol file (*PMMERGE.SYM*) for *PMMERGE.DLL*. This should be installed in the same directory as *PMMERGE.DLL* when using the Kernel Debugger or for dump analysis, in the same directory as the Dump Formatter.
- Availability of the PM programming reference from the Programmer′s ToolKit.
- Also of use, is ready access to the C header files from the Programmer′s Toolkit.

We start by giving an overview of the PM messaging environment in which an application′s PM thread operates.

14.1.3.1 The PM Messaging Environment

First consider the non-PM application programming model as shown in Figure 17 on page 274.

This diagram illustrates the following points:

- Non-PM application threads run in a relatively unconstrained environment (compare this with the following situation).
- The operating system provides a *black-box* set of services and interfaces.
- The hardware is not directly accessible by the application.

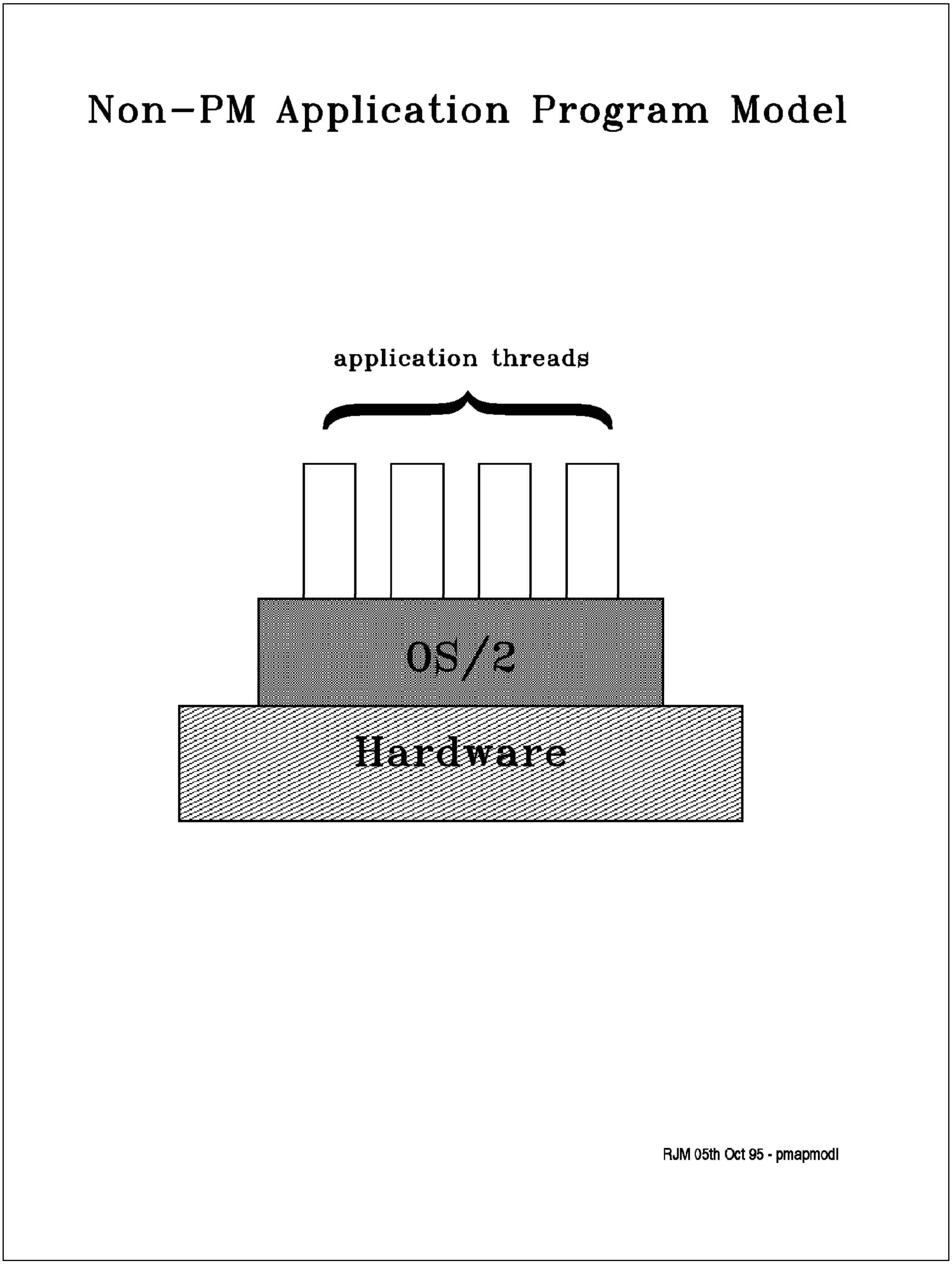


Figure 17. Non-PM Application Program Model

For PM message threads, the environment is radically different. The key difference is that application code that runs on a PM message thread is effectively a subroutine of the *WinGetMsg* API even though *WinGetMsg* is called by the application. The terminology often used to describe this reversal is *Program Inversion*. WinGetMsg is said to be *inverted* with respect to the application′s message thread.

We see this illustrated in Figure 18 on page 276.

Also illustrated by this diagram are the following points:

- PM message threads act in a cooperative way. They wait for messages and *pass them on* to the appropriate application if not for themselves.
- PM message threads should spend most of their elapsed time waiting for notification of messages, because of their cooperative nature.
- Application code that runs on the message thread should be limited to very short duration processing. We often speak of the *tenth-of-a-second rule*, which is intended to imply the transient nature of application code processing rather than a precise measure.
- If a message thread communicates with another thread or the operating system, then this should be done either asynchronously or so as not to violate the tenth-of-a-second rule.

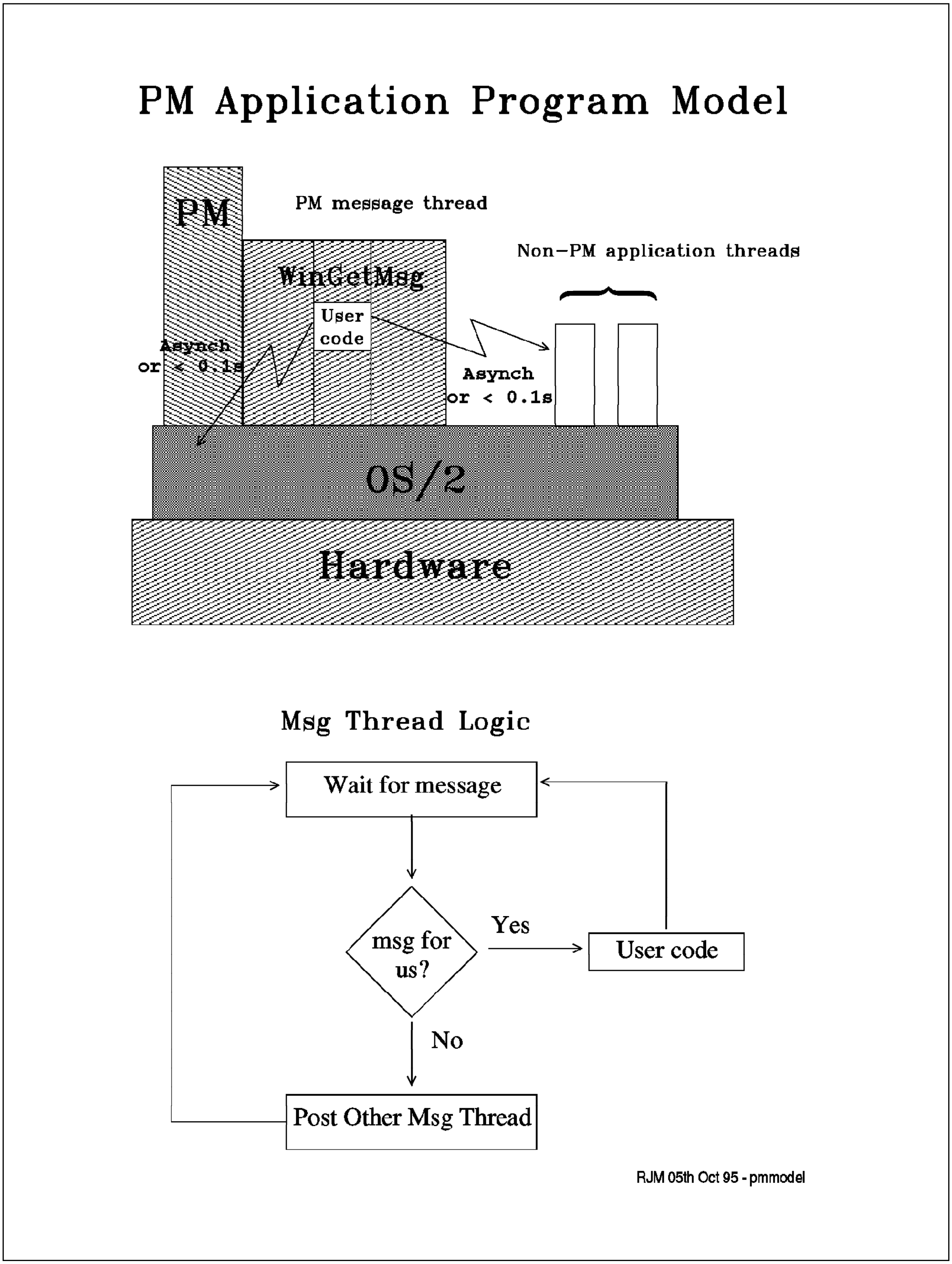


Figure 18. PM Application Program Model

14.1.3.2 PM Message Queues

PM messages are generated either as the result of user interaction with the system or by the use of various PM APIs. Both PM and non-PM applications may generate PM messages.

Messages may flow:

Synchronously, that is, require processing by the recipient before the sender can continue, or

Asynchronously, that is, where no response is required.

They may flow between threads (inter-thread messages) or from a thread to itself (intra-thread messages).

These characteristics require a message queuing mechanism to be implemented so that message order may be preserved.

Note: A message's meaning may often depend on the outcome of a preceding message. For example, consider the action of the F4 key after the Alt key has been pressed.

Each PM message thread has two *queues* or more strictly speaking a message queue and a message list. There may be only one instance of these two structures per thread.

- The message queue is a circular array, the size of which is specified or defaulted by the application when it creates the queue using *WinCreateMsgQueue*.

  This queue is used for the receipt of asynchronous messages generated by use of the *WinPostMsg* API.

- The message list has no depth and is created implicitly by *WinCreateMsgQueue*.

  This is used for the receipt of synchronous messages sent using *WinSendMsg*.

WinCreateMsgQueue also creates a message event semaphore that is posted whenever a message, synchronous or asynchronous, is posted; or a message response is generated. This is the semaphore on which *WinGetMsg* waits for message notification.

There is a system queue, which is also a circular array. Messages are enqueued on the system queue by the PMDD.SYS device driver as the result of external events deriving directly from the following:

Mouse activity

Keyboard activity

Use of a light pen

Timer ticks

PM maintains knowledge of who the current, mouse, keyboard, pen, and event receivers are. When an external event causes a message to be queued on the system queue, PM posts the message event semaphore of the current receiver of that particular event.

An application may define Window Procedures - entry points within the message thread. These are associated with a PM Window and a message queue. They receive control when a message is dispatched, that is dequeued from the message queue or message list. More than one window procedure may be serviced by the same message queue/list. Which one should be dispatched is determined from the *HWND*, which is one of the parameters associated with a message.

When *WinGetMsg* receives a message event notification, it first checks for the presence of received synchronous messages, if there are any dispatches them directly. Next it looks for an application generated (posted) message and finally for a system queue message.

The application thread explicitly dispatches asynchronous messages using *WinDispatchMsg*.

The System Queue entries are *SQMSG* structures.

The Application Queue entries are *QMSG* structures.

The Application Send Message List comprises a chain of *SMS* structures.

This scenario is illustrated in the following diagram:

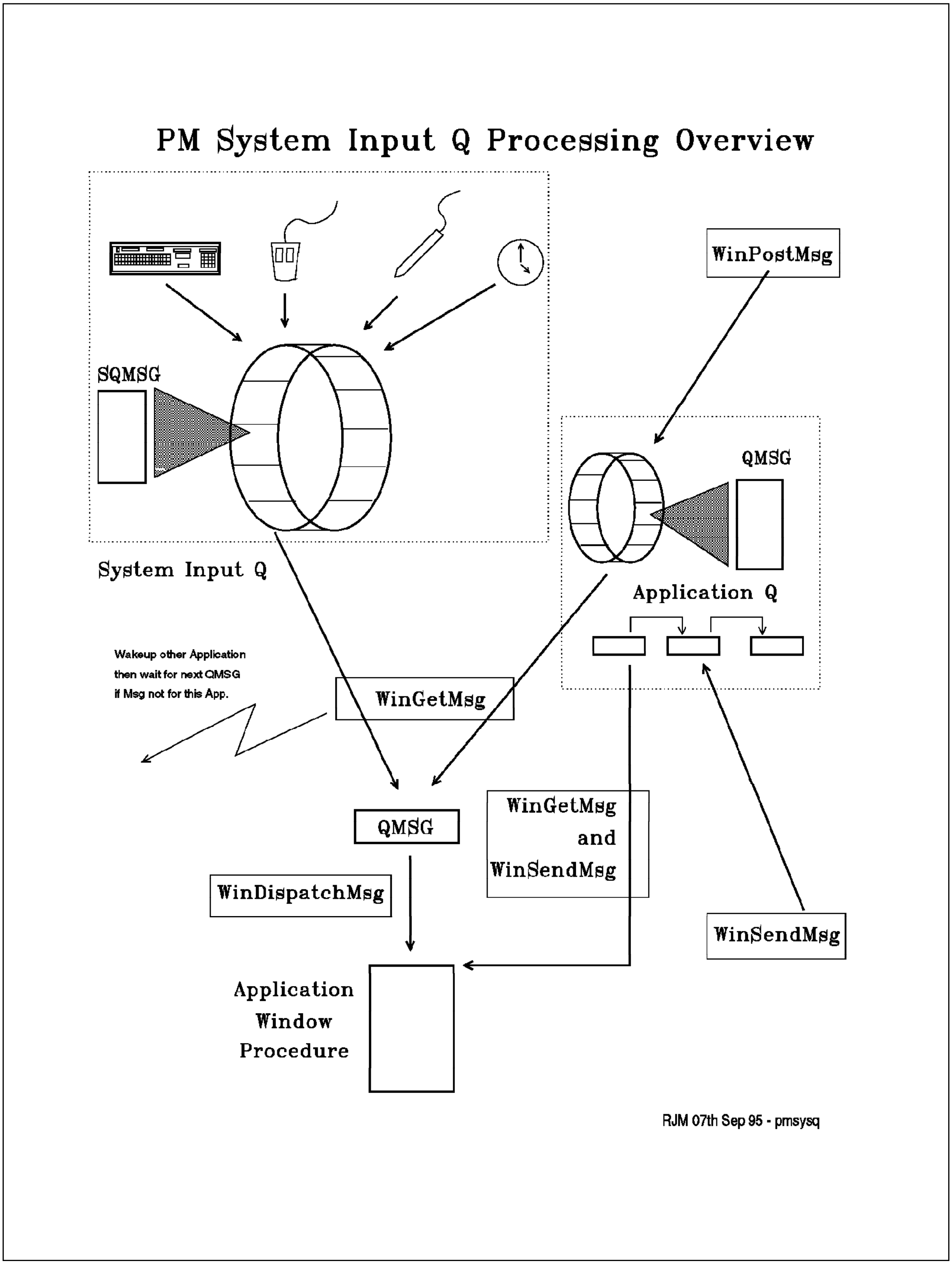


Figure 19. PM System Input Queue Processing Overview

14.1.3.3 An Application Thread′s Messaging Structures

The previous section introduced the notion of a message queue and list, of where there is one pair per PM message thread. We now look at these in more detail, with the associated PM system structures that comprise the application′s messaging environment.

The Message Queue Header (MQ)

This structure acts an anchor for all the message processing structures of an application message thread. It is created by WinCreateMsgQueue and the returned *HMQ* (message queue handle) is the address of this structure. PM often refers to the address of a *MQ* as its *PMQ*.

The principle fields of interest are:

Offset *Description*

+0x14 The current read position of the message queue.

Since the message queue is a circular array, four pointers have to be maintained: the current read position, current write position, top and bottom of array.

Each queue entry is a *QMSG* structure.

Entries are added to the queue in increasing address order, until the maximum (bottom) is reached then entries are added from the top.

Removal of entries only involves updating the current read pointer. Thus, a small trace of past message activity may be seen by scanning backwards from the current read pointer to the current write pointer.

+0x18 The current write pointer.

+0x24 The Pid of the message thread to which this *MQ* belongs.

+0x28 The Tid of the message thread to which this *MQ* belongs.

+0x44 The most recent *SMS* on which a response is awaited.

The presence of a non-zero value in this field implies that the message thread is currently blocked in WinSendMsg waiting for a response.

If the message thread recurses, for example through the receipt of a synchronous message, then a subsequent WinSendMsg will cause this field to be updated. The previous contents are saved on the stack.

A non-zero value in this field is of prime interest when diagnosing hangs. It immediately focuses our attention on the recipient of this message.

+0x48 The current *SMS* received.

This field is non-zero when an *SMS* is removed from the receive list for processing by its associated window procedure.

When this field is non-zero, it implies that the thread′s window procedure has been dispatched to process a received message.

+09c The Received message list.

SMSs are chained from this location pending dispatch.

Upon dispatch the oldest message is removed from the list and pointed to from offset +0x48 of the *MQ*.

+0xa4 The thread slot number of the message thread to which this *MQ* belongs.

This is very useful for correlating *MQs* to threads.

The Send Message Structure (SMS)

The *SMS* is created for synchronous messages and linked to the receive list (**MQ+0x9c**) when WinSendMsg is called.

The principle fields of interest are:

Offset	*Description*
+0c	Pointer to the next (more recent) *SMS* in the receive list.
+14	Pointer to the *MQ* to which this *SMS* has been sent.
+18	Pointer to the *MQ* of the thread from which this *SMS* was sent.
+24	Pointer to the *WND* the represents the Window to which this message has been sent.
+28	The message Id.
+2c	Message parameter 1.
+30	Message parameter 2.

Offsets +0x14 and +0x18 are of prime interest in diagnosing hangs. They enable us to locate the recipient of a message, of which a response is pending and therefore the thread which is causing our thread to remain blocked.

The Queue Message Structure (QMSG)

This is the structure used by applications when calling WinDispatchMsg.

The *QMSG* is also the form of an entry on the application′s message queue.

The principle fields of interest are:

Offset	**Description**
+0	The window handle (HWND). This is an index (ignoring the high-order bit) in to the handle table. From the handle table we can obtain the equivalent PWND or pointer to the WND.
+4	The message Id.
+8	Message parameter 1.
+c	Message parameter 2.

The Handle Table

This is a global table that is used to correlate window handles (HWNDs) with pointers to WNDs (PWNDs). The table comprises a 0x20 byte header with 8-byte entries. The first word of each entry is a PWND and the second a

Boolean flag, which if non-zero, indicates that an HWND/PWND combination is non-deleteable.

The handle table may be located from the address at symbol:

```
pHandleTable
```

The Window Structure (WND)

This structure represents a window. It is created by WinCreateWindow.

The WND has two main functions:

>> It acts as the link between the *MQ* and the thread's window procedure

>> It establishes the *WND* hierarchy.

The principle fields of interest are:

Offset	Description
+0	Next sibling WND.
+4	Parent WND.
+8	First Child WND.
+28	The pointer to the MQ that will queue messages sent and posted to this window.
+2c	The window's HWND.
+30	A Boolean, which if non-zero, indicates that the window procedure address is a 16-bit far pointer.
+34	The address of the window procedure.

This scenario is illustrated in the following diagram:

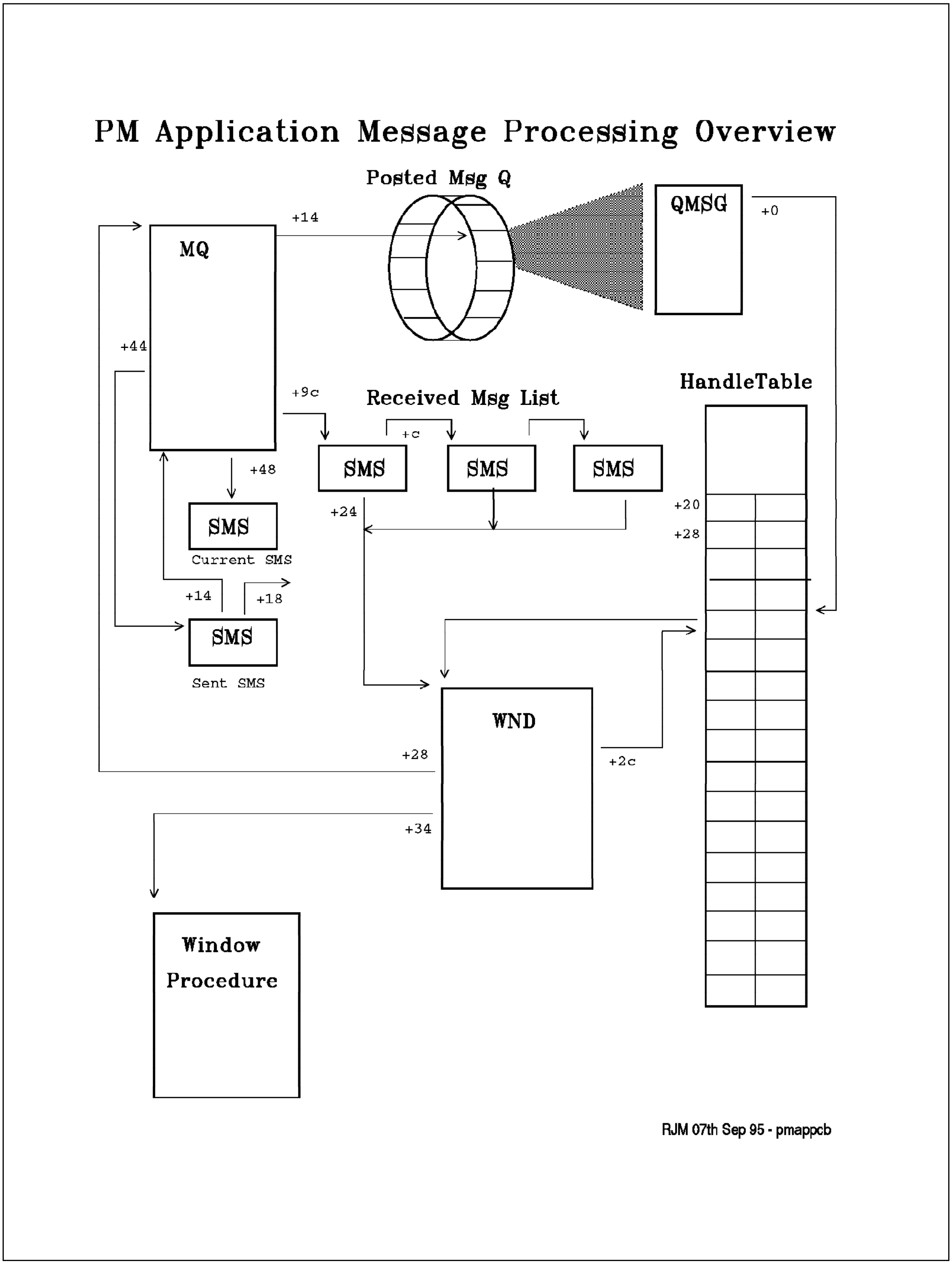


Figure 20. PM Application Message Processing Overview

14.1.3.4 PM Message Processing Logic

The following sections provide a summary of the essential internal logic of PM′s message handling. This is provided to give the reader sufficient understanding that will enable most application problems, especially those that cause hangs, to be identified. In most cases hangs in the PM environment are caused by a misuse or misunderstanding of the message thread model, especially the way in which message threads act in a cooperative manner.

An outline is given for each of the following:

"WinGetMsg Logic."

"WinSendMsg Logic" on page 287.

"Waiting for Message Activity" on page 289.

"WinSetFocus Logic" on page 289.

Note:

In each of these outlines, the added complication of calling hooks has been omitted.

WinGetMsg Logic: WinGetMsg operates essentially as a loop that waits for message activity and returns messages to the user. Conceptually the application′s code acts as an inverted subroutine of WinGetMsg. This was illustrated in The PM Messaging Environment.

These are the essential steps in WinGetMsg processing:

- When WinGetMsg wakes, it first unlocks the System Queue if owned or if it is the *active thread*.

  The MQ of the current system queue owner is pointed to by *pmqsyslock*. This is set to zero if this points to the MQ of the current thread.

  The active thread is the thread that has the right to unlock to system queue if locked by another thread. Normally this is the thread that manages the MQ of the window in focus. Normally the thread that has locked the system queue is the active thread.

- The received list is checked.

  If SMSs are queued then each is removed successively and the corresponding window procedure is called.

  Received messages, that is messages sent via WinSendMsg to this thread, are not returned by WinGetMsg. The window procedure is called directly.

- The application′s queue is checked for posted messages.

  If one is found it is dequeued and returned to the application.

- If no posted message is found then WinGetMsg tries to process the system queue.

- We attempt to lock the system queue if free.

  If *pmqsyslock* is zero it is set to the current thread′s **MQ** address.

- If the lock was successful then we peek the next system queue message.

  If the lock was unsuccessful we return to the beginning of the loop and wait on the message queue semaphore.

- If the next system message is for this thread then it is dequeued and returned to the application with the system queue lock still held.

  The possibility of holding the system queue lock while running in user code is vital to note. While this happens only active thread can dequeue a system queue message. The reason for holding the system queue lock is for performance. It is likely that one system message will be followed by a sequence for the same thread. If the lock was released, unnecessary processing on other message queue threads would take place. More recently queued messages could not be processed out of turn anyway, since the interpretation of a system message depends upon the outcome of the preceding system message.

- If the next system message is for another application then the system queue is unlocked.

- The other application is made the current input receiver.

  PM distinguishes between current mouse, keyboard and event receiver. WinGetMsg makes the other application the current mouse, keyboard or event receiver depending upon the message category.

- Finally the other application′s message queue semaphore is posted. WinGetMsg returns to the beginning of its message loop by waiting on its own message semaphore for more message activity.

This processing is illustrated in the following diagram:

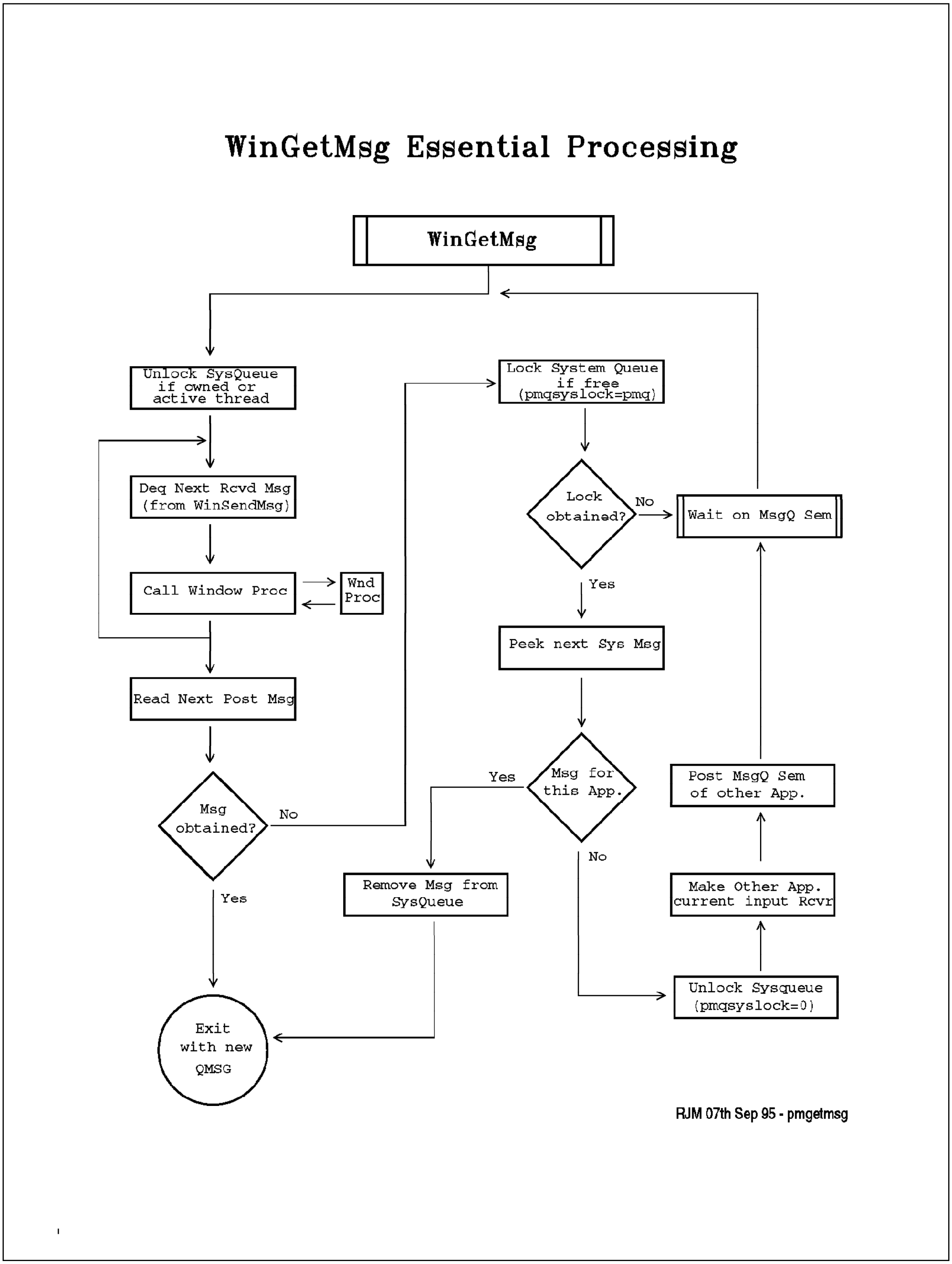


Figure 21. WinGetMsg Essential Processing

WinSendMsg Logic

These are the essential steps in WinSendMsg processing:

- We check to see if the message is being sent to the same thread.

  WinSendMsg to the same thread is known as an *Intra-thread* send.

  WinSendMsg to another thread is known as an *Inter-thread* send.

- If intra-thread then the message is dispatched immediately, from within WinSendMsg.

  This behavior implies that a window procedure may recurse many times, even if waiting for a response to a WinSendMsg.

- If inter-thread then the message is enqueued to the receive list of the recipient′s MQ.

- Active thread status is transferred to the receiving thread (if currently owned).

  This allows the receiver to unlock the system queue if it had been locked by the current thread and the current thread is the active thread.

- The receiver′s message queue semaphore is posted.

- WinSendMsg waits on the current thread′s message queue for a response to the sent message.

This processing is illustrated in the following diagram:

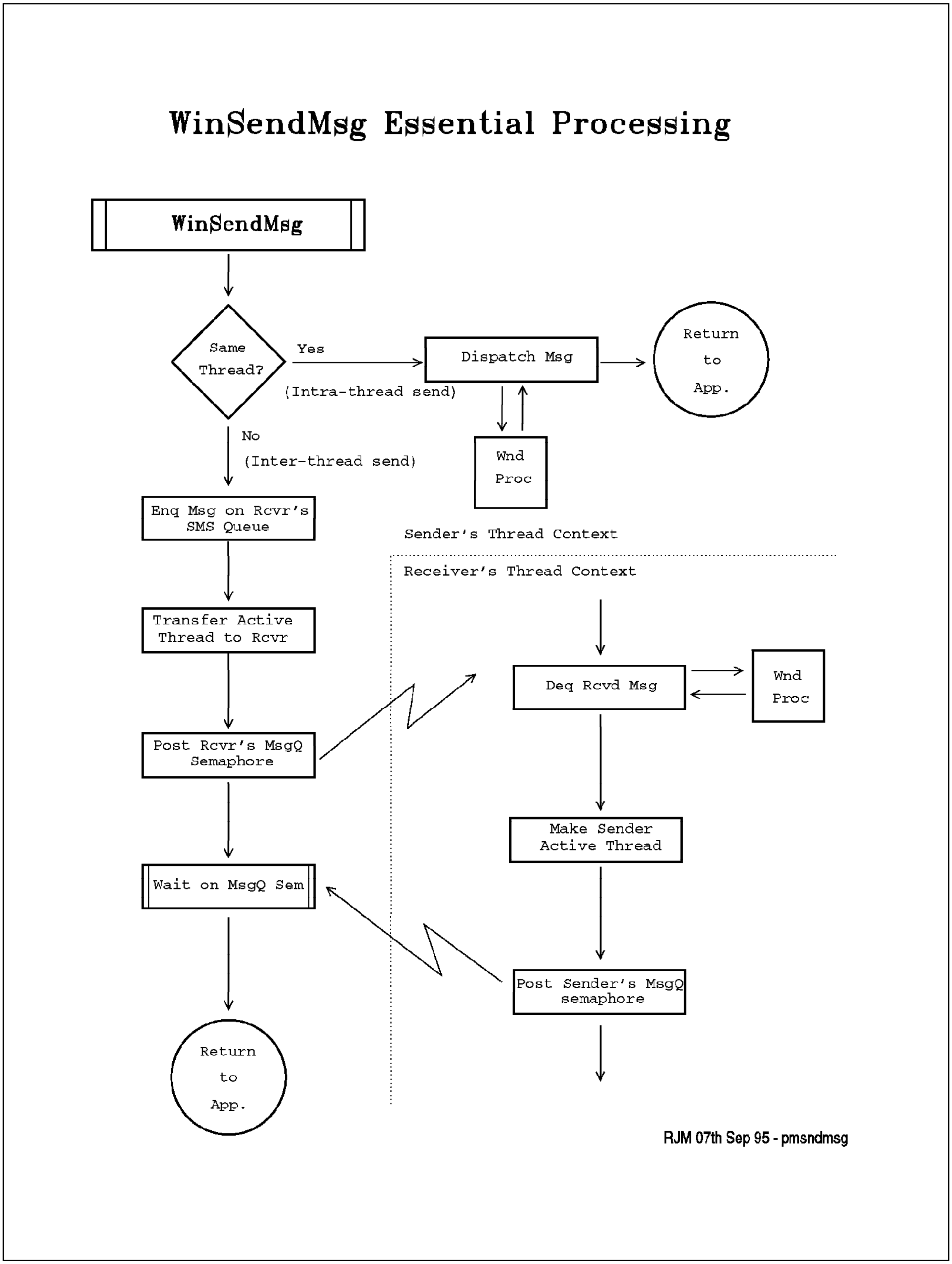


Figure 22. WinSendMsg Essential Processing

Waiting for Message Activity: In order to process synchronous messages as swiftly as possible, PM always checks the receive list of the current thread for pending SMSs before waiting on an internal PM semaphore.

If SMSs are found queued, they are successively dispatched.

Only when the receive list has been processed does PM finally wait on a semaphore.

This applies particularly to the message processing semaphore, but also equally to semaphores that serialize access to resources such as the .INI files.

WinSetFocus Logic: WinSetFocus has a subtle bearing on message processing since it selects a new active thread and new current input receivers for system messages.

Note:

Focus may be changed by a third party.

These are the essential steps in WinSetFocus processing:

- WM_FOCUSCHANGE is sent to the message thread losing the focus.
- The current window in focus is changed.

  pwndfocus points to the WND of the focus owner.
- Unlock the system queue if locked.

  The MQ of the current system queue owner is pointed to by *pmqsyslock*. This is set to zero if it points to the MQ of the current focus owner.
- The target window's message thread is marked as the new focus owner.

  pmqfocus is set to the address of the new focus owner's MQ.
- The target thread's message queue is made the current mouse and keyboard input receiver.

  pmqMouseWake and *pmqKeyWake* are set to the address of the new focus owner's MQ.
- The new focus owner's message thread is marked as the new active thread.
- A priority boost is applied to the new focus owner's message thread.
- WM_FOCUSCHANGE is sent to the message thread gaining the focus.

This processing is illustrated in the following diagram:

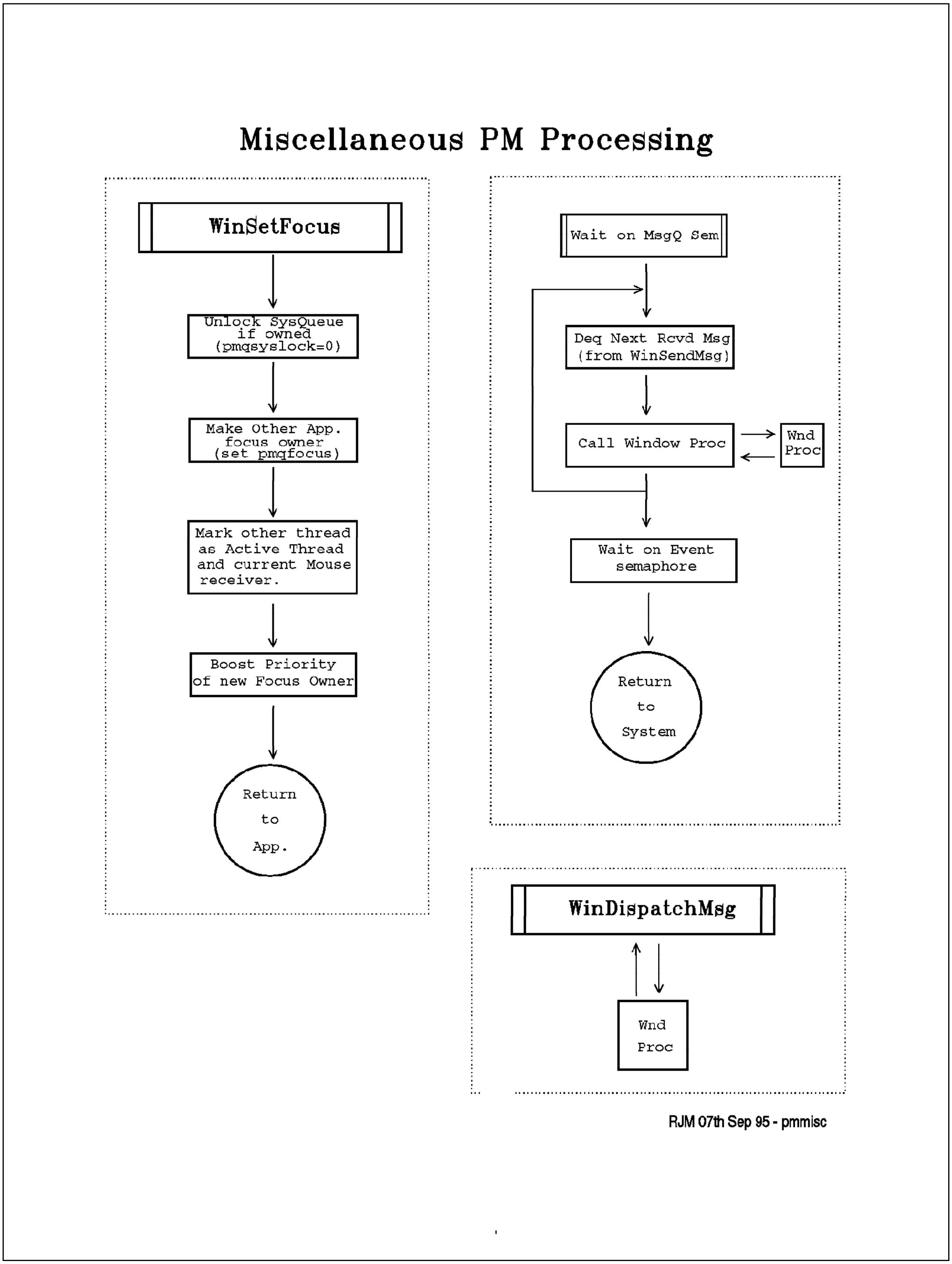


Figure 23. Miscellaneous PM Processing

Application Not Responding to Messages Logic: The application not responding to messages dialog, or *BadApp* dialog as it is sometimes referred to, appears after Ctrl-Esc has been hit and the system has not been able to display the task list.

The essential logic for this processing is as follows:

- Ctrl-Esc is hit and 5 second timer is started.

- If the task list appears before the time-out then all is well, if not we check for who might be holding things up.

- We try for 5 seconds to obtain the User PM Semaphore. If unsuccessful then the Pid, Tid and Session Id of the owner is saved in the QHPSTRUCT.

- If *pmqsyslock* is owned then the Pid, Tid and Session Id of the owner is saved in the QHPSTRUCT.

- If *pmqfocus* is owned then the Pid, Tid and Session Id of the owner is saved in the QHPSTRUCT.

- We now enter a second wait of a further 3 seconds, after which, we build a second QHPSTRUCT.

- If the Tid and Pid are different and we have not yet had an acknowledgement from the task list then we wait a further 12 seconds, on the assumption the processing is slow, but not hung.

- If the Tid and Pid are the same or we have expired on our third time-out then we set *fBadAppDialog* true and reset the cause for the hang:

  - If *User_Sem* held then release *User_Sem*
  - if system queue locked then reset *pmqsyslock*
  - if focus owner hung, then reset *pmqfocus*

  Report the hanging application in the *BadApp* dialog.

This processing is illustrated in the following two diagrams:

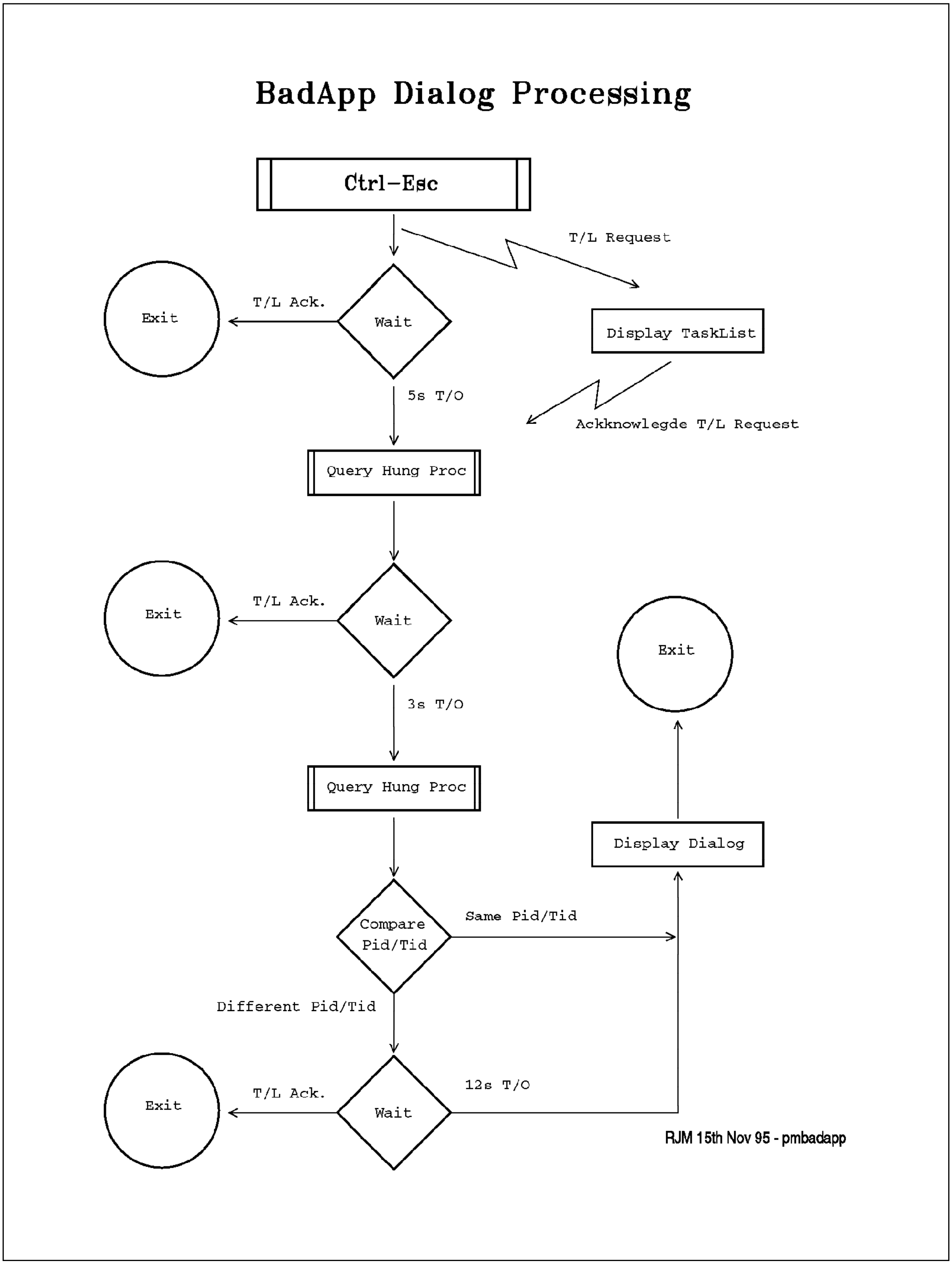


Figure 24. BadApp Dialog Processing

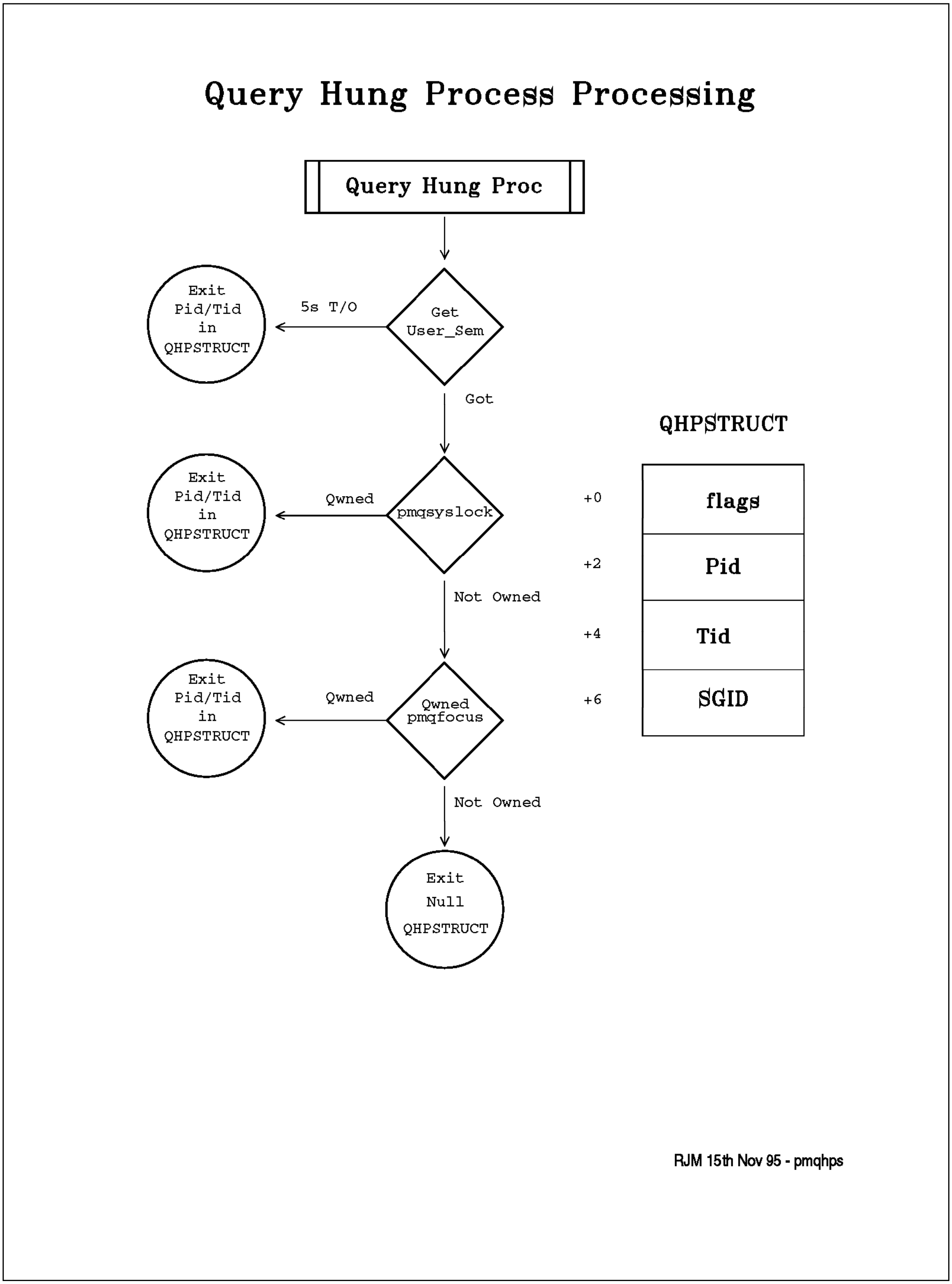


Figure 25. Query Hung Process Processing

The flags in the QHPSTRUCT indicate the detected reason for hanging. These may be a combination of:

Name	Bit Mask	Description
QHP_SYSQUEUELOCK	0x0001	System Queue Locked
QHP_SENDMSGLOCK	0x0002	Waiting for a response to WinSendMsg
QHP_CLIPBRDLOCK	0x0004	
QHP_WINDOWLOCKED	0x0008	
QHP_VISRGNLOCKED	0x0010	
QHP_LOCKWINDOWUPDATE	0x0020	
QHP_FSRSUSERHANG	0x4000	Waiting for the User_Sem
QHP_INPUTPROCESSED	0x8000	

14.1.3.5 Useful Symbols for PM Structures

The following list is a small selection of global symbols from PMMERGE.SYM that will be of help in locating the structures associated with message handling:

pmsemaphores
This is the label for the table of PMSEM and GRESEM semaphore structures. Offset +0x20 is the location of the user semaphore. If this is owned and not released within the time-out period after Ctrl-Esc has been hit, then the ′Application Not Responding to Messages′ reports the semaphore owner as the culprit.

pmqSyslock
The PMQ of the thread that has locked the system queue.

This is the first place to look when investigating a hang in a PM application. The system uses this to name a *bad* application when the ′Application Not Responding to Messages′ dialog appears.

pmqFocus
The PMQ of the window that has the focus.

If *pmqSyslock* is zero, the system uses this as a second choice for the *bad* application when the ′Application Not Responding to Messages′ dialog appears.

pmqKeyWake
The PMQ of the current keyboard event receiver.

pmqMouseWake
The PMQ of the current mouse event receiver.

pmqEventWake
The PMQ of the current miscellaneous event receiver.

pwndFocus
The PWND of the window currently in focus.

pHandleTable
The address of the handle table.

pSysqueue
The MQ of the system input queue.

The system queue is headed by a partial MQ since it does not require the fields to support a receive list.

pmqShell
The PMQ of the 1st thread of the 1st Shell Process.

This thread is responsible for starting and re-starting the Workplace Shell.

pmqShell2
The PMQ of the 1st thread of the 2nd Shell or Workplace Shell Process.

This thread is the main thread of the desktop PM application.

paAABRegs
The address of the application anchor block registers (AAB).

AAB registers are allocated for each PM application message thread. This is located in thread local memory, which implies that it is correct only for the current thread context. The thread local memory area is saved in the TCB and restored when the thread is made current. Since the TCB may be located in System storage under any context then a thread′s PMQ may be found in any context from its TCB.

pwndObject
The PWND of the Object Window.

This is the parent or owner of all non-display windows.

pwndDesktop
The PWND of the Desktop Window.

This is the parent of owner of all displayable windows.

SleepPMQ+nnn
When a thread is blocked at approximately offset +0x155 into *SleepPMQ* then it is waiting on the message queue semaphore for new messages or responses to outstanding sent messages. Offset +0x30 from the current stack pointer usually contains the PMQ for the current thread.

pmqList
: All MQs are chained on a single linked master list from offset +0x0 of the MQ. The current head of the master list is pointed to by *pmqList*.

psmsList
: All SMSs are chained on a single linked master list from offset +0x0 of the SMS. The current head of the master list is pointed to by *psmsList*.

qhpsBadApp
: This is the label of the QHPSTRUCT saved the first time a time-out occurs after Ctrl-Esc has been hit and the ′Application Not Responding to Messages′ dialog appears.

fBadAppDialog
: A Boolean that indicates when the ′Application Not Responding to Messages′ dialog has been displayed.

14.1.3.6 Useful PM Structures

The following diagrams illustrate the main PM structures for the messaging function. These are laid out as if viewed under the Kernel Debugger or Dump Formatter by displaying them using the DD command.

"PM Message Queue Header" on page 297.

"PM Window Structure (WND)" on page 298.

"PM Message Structures (SMS, QMSG, SQMSG)" on page 299.

"PM Application Anchor Block Registers" on page 300.

"Stack Layout at Useful Entry Points" on page 301.

PM Message Queue Header

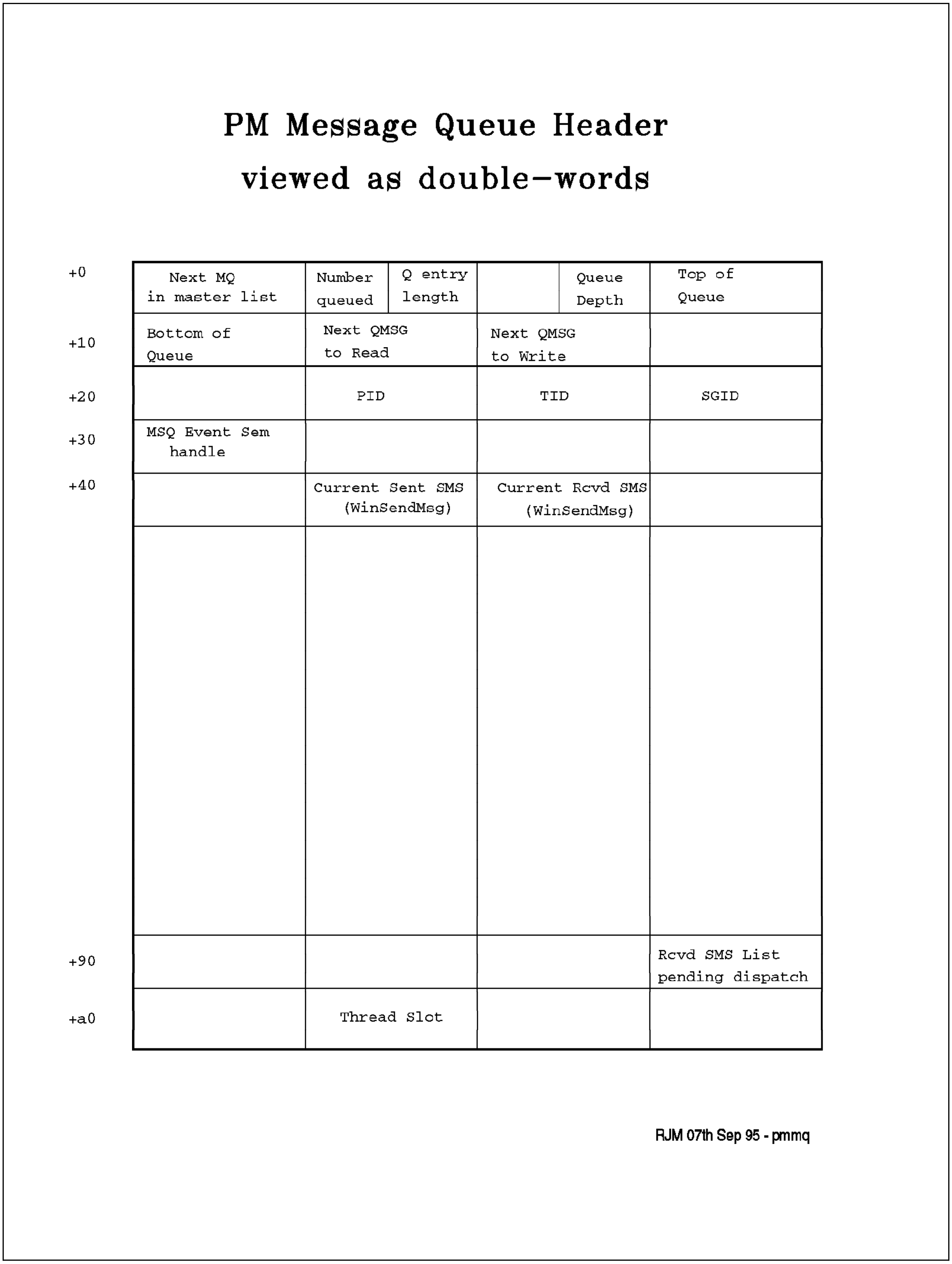


Figure 26. PM Message Queue Header Viewed as Doublewords

PM Window Structure (WND)

viewed as double-words

+0	Next Sibling WND	Parent WND	Child WND	Owner WND
+10				User specified id
+20			PMQ	HWND
+30	16-bit BOOL	Window Proc address		
+40				
+50				
+60	Window words start here			

RJM 07th Sep 95 - pmwnd

Figure 27. PM Window Structure (WND) Viewed as Double-words

PM Send Message Structure (SMS) viewed as double-words

+0	Next SMS on master list	Send List Head		Next SMS in Receive List
+10	time	Sender PMQ	Receiver PMQ	Result
+20		PWND	Msg Id	MP1
+30	MP2			

PM Queue Message Structure (QMSG) viewed as double-words

+0	HWND	Posted Msg Id	MP1	MP2
+10	time	X Co-ord	Y Co-ord	

PM System Queue Msg Structure (SQMSG) viewed as double-words

+0	Msg Id	MP1	MP2	time
+10				

RJM 07th Sep 95 - pmsms

Figure 28. PM Send, Queue and Queue Message Structures

PM Application Anchor Block Registers

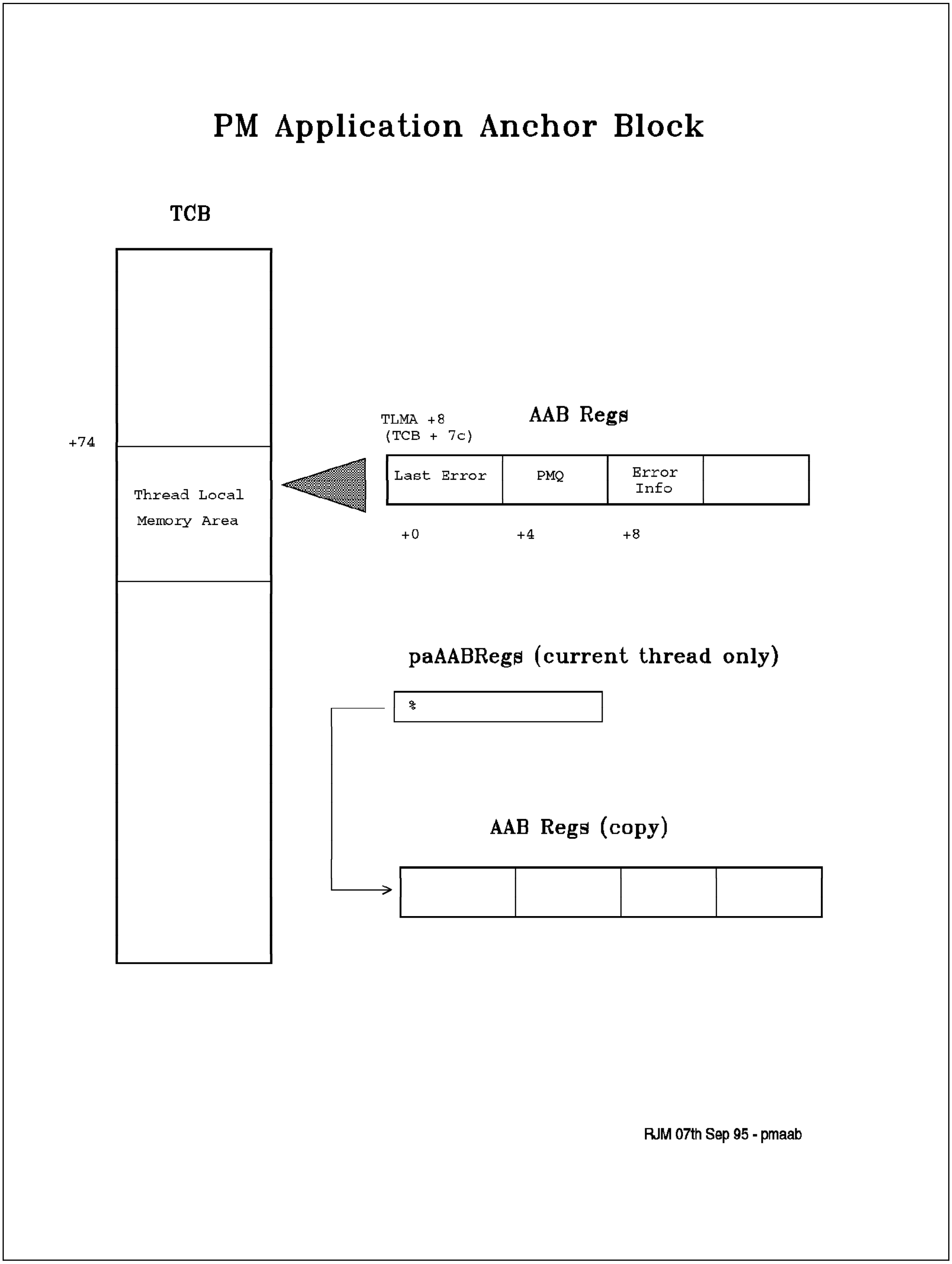


Figure 29. PM Application Anchor Block

Stack Frames for Common Entry Points viewed as double-words

Win32PostMsg Entry Point
Win32SendMsg Entry Point
Window Procedure Entry Point

+0	Next Frame pointer	Return Address following call to entry point	HWND	Msg ID
+10	MP1	MP2		

Win32DispatchMsg Entry Point

+0	Next Frame pointer	Return Address following call to entry point	HAB	Pointer to QMSG
+10				

RJM 14th Nov 95 - pmstack

Figure 30. Stack Frames for Common Entry Points Viewed as Doublewords

14.1.3.7 PM Worked Examples under WARP

We give two examples of diagnosing common application problems:

“Example 1 - A Trap in PMMERGE.DLL” - A trap in PMMERGE.DLL caused by an application fault.

“Example 2 - A Hang in a PM Application” on page 305 - A hang in the WorkPlace, again caused by an application fault.

Further techniques are illustrated in:

“How to Find the MQ of any Thread” on page 308.

“How to find the MQ of a BadApp Application” on page 309.

“Finding Application and System Queue Elements” on page 311.

Further examples, with annotated solutions, may be found on the accompanying CD-ROM in the TURKEY lab exercise.

Example 1 - A Trap in PMMERGE.DLL:

Steps for analyzing traps in PM DLLs:

- Intercept the trap at the point of failure.
- Unwind the stack to the application call.
- Validate the parameters to the API call.
- If necessary, determine how the user routine was invoked by examining the MQ and looking for dispatched messages or by unwinding the stack further.

This example is of a trap in PMMERGE.DLL, but caused by an application fault.

Because we have a trap E, we set the fatal vector under the Kernel Debugger (or use TRAPDUMP=ON in CONFIG.SYS to take a dump) then re-create the problem.

```
##vsf *
##g
Trap 14 (0EH) - Page Fault 0006, Not Present, Write Access, User Mode
eax=00000007 ebx=00273fcc ecx=00000001 edx=00000007 esi=12d3e089 edi=00000000
eip=1bd3d261 esp=00273ebc ebp=00273f1c iopl=2 rf -- -- nv up ei pl nz ac po cy
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001d4000
005b:1bd3d261 f3a5          repe movsd es:00000000=invalid ds:12d3e089=69727453
##ln
005b:00000000 turkey:FLAT:__$dummy$ + 1bd3d261
```

We have trapped in a DLL, probably PMMERGE.DLL, certainly not in the user’s .EXE code. We unwind the stack to find the return address in the user’s .EXE

```
##dd %ebp
%00273f1c  00273f5c 1bd3d05b 00000000 00000005
%00273f2c  00080001 00000000 00080013 00010423
%00273f3c  00190008 00000000 00000001 00000014
%00273f4c  00000000 80000144 00000001 00190008
%00273f5c  00273fa8 1bd03893 80000144 00000071
%00273f6c  00280018 00000000 000103ec 00000000
%00273f7c  00000000 00190008 00273fcc 00000000
%00273f8c  00000000 ffffffff 80000144 12d31630
```

```
##d
%00273f9c  00000000 12d31494 00190008 00273ff4
%00273fac  00010932 00190008 00273fcc 0000000c
%00273fbc  00000004 00000004 00000007 80000144
%00273fcc  80000144 00000071 00280018 00000000
%00273fdc  0092d87d 0000027a 000000f0 00000000
%00273fec  12d31630 00190008 00000000 1bfbbf68
%00273ffc  00022bf4
Invalid linear address: %00274000
```

Not all the stack was paged in to physical memory, but never mind. Enough is there to allow us to find the return address to the user's application code.

Following the base pointer (EBP):

```
%00273f1c  00273f5c 1bd3d05b

%00273f5c  00273fa8 1bd03893 80000144 00000071

%00273f9c                             00273ff4
%00273fac  00010932 00190008 00273fcc 0000000c
```

The return address is %10932. We inspect the code just before this address.

```
##u %10932-20
%00010912 25837ddc2a        and     eax,2adc7d83
%00010917 0f8507000000      jnz     %00010924
%0001091d e916000000        jmp     %00010938
%00010922 8bc0              mov     eax,eax
%00010924 8d45d8            lea     eax,[ebp-28]
%00010927 50                push    eax
%00010928 ff75fc            push    dword ptr [ebp-04]
%0001092b b002              mov     al,02
%0001092d e8862ecf1b        call    %1bd037b8
%00010932 83c408            add     esp,+08
%00010935 ebc1              jmp     %000108f8
%00010937 fc                cld
##ln %1bd037b8
%00000000 turkey:FLAT:__$dummy$ + 1bd037b8
```

This call was to a routine at %1bd037b8. LN doesn't give us a useful symbol (our .EXE is being selected instead of the correct .DLL symbol). We find out who owns this address from the memory management control blocks.

```
##.m %1bd037b8

*har     par      cpg        va     flg next prev link hash hob   hal
 01c0 %feaf168a 00000001 %ffe3c000 101 000d 01e2 01bd 0000 013e 000c      =0000
 hal=000c pal=%ffe5c078  har=01c0  hptda=010c  pgoff=000d7  f=021
 har     par      cpg        va     flg next prev link hash hob   hal
 01bd %feaf1648 00000001 %ffede000 101 01a2 01be 0106 0000 013e 0009      =0000
 hal=0009 pal=%ffe5c060  har=01bd  hptda=010c  pgoff=00000  f=021
 har     par      cpg        va     flg next prev link hash hob   hal
 0106 %feaf068e 000000f0 %1bd00000 3d9 0105 0107 0000 0000 013e 0000  hco=00307
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 013e  01c0 0000   1838 0139 0139  0000 00  02 00 00 shared    c:pmmerge.dll
```

```
hco=00307 pco=fe64ef3e  hconext=00296 hptda=032f f=1c  pid=0019 a:turkey.exe
hco=00296 pco=fe64ed09  hconext=001f9 hptda=0301 f=1c  pid=0009 d:cmd.exe
hco=001f9 pco=fe64e9f8  hconext=00188 hptda=02c9 f=1c  pid=0007 d:cmd.exe
hco=00188 pco=fe64e7c3  hconext=00107 hptda=0291 f=1c  pid=0005 c:2000.exe
hco=00107 pco=fe64e53e  hconext=0008e hptda=023b f=1c  pid=0004 c:pmshell.exe
hco=0008e pco=fe64e2e1  hconext=0002a hptda=01b5 f=1c  pid=0003 c:harderr.exe
hco=0002a pco=fe64e0ed  hconext=00000 hptda=010c f=1d  pid=0002 c:pmshell.exe
```

We can see that our call was to an entry point in PMMERGE.DLL. We need to activate PMMERGE's symbols.

```
##wa pmmerge
##ln %1bd037b8
%1bd037b8 pmmerge:PM32BIT:WIN32DISPATCHMSG
##ln
005b:1bd3d064 pmmerge:PM32BIT:LoadStrMsg + 1fd
005b:1bd3d2a8 WIN32POSTQUEUEMSG - 47
```

So we called WinDispatchMsg and some time later we probably called LoadStrMsg, which is where we trapped. First we need to check the parameters to WinDispatchMsg. These are:

HAB 00190008

PQMSG 00273fcc

The QMSG at %273fcc is also in the stack we dumped:

```
%00273fcc  80000144 00000071 00280018 00000000
%00273fdc  0092d87d 0000027a 000000f0 00000000
```

The first parameter is the HWND. We convert this to a PWND, dump the WND and look for the window procedure entry point.

```
##dd phandletable l1
9f3f:0000ab78  12d50000

##dd %12d50000+20+(8*144) l2
%12d50a40  12d31494 00000000

##dd %12d31494
%12d31494  12d31838 12d3c974 00000000 12d3c974
%12d314a4  00c80262 0104029e 80000000 00000008
%12d314b4  12d314f0 00000004 12d31630 80000144
%12d314c4  00000000 000103ec 00000000 00000000
%12d314d4  12d3147c 00000000 00000000 00000000
%12d314e4  00000000 2050534d 00000034 12d31894
%12d314f4  00004b4e 00000000 0000fc4e 00000019
%12d31504  00000000 000103ec 00000000 00000000
```

Note: We could have used the following more complex single command construct to achieve the same result:

```
##dd %(dw(%(dw(phandletable))+20+(8*144)))
%12d31494  12d31838 12d3c974 00000000 12d3c974
%12d314a4  00c80262 0104029e 80000000 00000008
%12d314b4  12d314f0 00000004 12d31630 80000144
%12d314c4  00000000 000103ec 00000000 00000000
%12d314d4  12d3147c 00000000 00000000 00000000
%12d314e4  00000000 2050534d 00000034 12d31894
%12d314f4  00004b4e 00000000 0000fc4e 00000019
%12d31504  00000000 000103ec 00000000 00000000
```

The window procedure entry point is at offset +0x34.

We now unassemble this:

```
##u %103ec
%000103ec 55              push    ebp
%000103ed 8bec            mov     ebp,esp
%000103ef 83ec08          sub     esp,+08
%000103f2 8b4508          mov     eax,dword ptr [ebp+08]
%000103f5 a32c0d0200      mov     dword ptr [00020d2c],eax
%000103fa 8b450c          mov     eax,dword ptr [ebp+0c]
%000103fd e93a000000      jmp     %0001043c
%00010402 8bc0            mov     eax,eax
%00010404 ff7508          push    dword ptr [ebp+08]
%00010407 b001            mov     al,01
%00010409 e8322dcf1b      call    WIN32QUERYANCHORBLOCK (%1bd03140)
%0001040e 8945fc          mov     dword ptr [ebp-04],eax
##u
%00010411 6a00            push    +00
%00010413 6a14            push    +14
%00010415 6a01            push    +01
%00010417 6a00            push    +00
%00010419 ff75fc          push    dword ptr [ebp-04]
%0001041c b005            mov     al,05
%0001041e e81dccd21b      call    WIN32LOADSTRING (%1bd3d040)
%00010423 83c418          add     esp,+18
%00010426 eb1e            jmp     %00010446
%00010428 8b4508          mov     eax,dword ptr [ebp+08]
%0001042b e8d0fbffff      call    main (%00010000)
%00010430 eb14            jmp     %00010446
```

We notice that we trapped in an internal routine called LoadStrMsg and that we have called WinLoadString in the window procedure. Could these be related?

We see from the PM Programming Reference that WinLoadString has five parameters. The right most is a pointer to a buffer and we see that the window procedure has pushed 0 on the stack this will surely cause WinLoadString to trap at some point. How do we make this supposition less circumstantial and more concrete?

Clearly, for EBP to take us back to a call to WinDispatchMsg, without finding a stack frame from the window procedure implies that PMMERGE is using optimised code when the trap occurred. That is, the conventional use of EBP is not in place - and this does occur in many internal routines in PMMERGE, for performance reasons. If we scan back through the stack we notice the address %10423 occurring shortly before (in time) the call to LoadStrMsg. This address is the return address from the WinLoadString call in the window procedure. It would seem therefore that we have called that API with the bad parameter as suspected.

Example 2 - A Hang in a PM Application:

Steps for analyzing hangs in PM applications:

- Determine whether there is a general hang in the PM environment, or a just in one application. If the latter then proceed with normal hang analysis.

- Check whether the *User_Sem* is owned. If it is then this may be an indication of a problem. Determine the owner and their thread status.
- Check *pmqsyslock* to see if the system queue is locked. If it is, then determine the owner of the lock and their thread status.
- Check *pmqfocus* if neither of the preceding checks reveals anything informative. Determine the thread in focus and its status.
- If *pmqfocus* is a shell thread, check *fBadAppDialog*. If it is non-zero then analyze the QHPSTRUCT at label *qhpsbadapp*.
- If none of the preceding steps yields any results then check the shell processes. In particular *pmqshell* and *pmqshell2*. Most of the time these threads should be waiting for a message to arrive. Any other state should be transient.

This example is of a hang in the WorkPlace caused by a PM application fault.

First we check out whether the *User_Sem* is held, whether the system queue is locked and if necessary who has the focus.

```
##db pmsemaphores+20 l20
9f3f:0000b4d4 50 4d 53 45 4d 00 00 00-00 00 00 00 00 00 00 00 PMSEM...........
9f3f:0000b4e4 00 00 00 00 00 00 00 00-03 00 01 80 00 00 00 00 ................
##dd pmqsyslock l1
9f3f:0000ed14  12d3128c
##dd %12d3128c
%12d3128c  12d31630 00000020 0000000a 12d31334
%12d3129c  12d31474 12d31334 12d31334 0000a400
%12d312ac  00000000 0000001a 00000009 00000012
%12d312bc  80030059 0097ec67 00000048 00000089
%12d312cc  00000001 12d3ff5c 00000000 00000010
%12d312dc  00000000 00000000 00000000 00000000
%12d312ec  00000000 00000000 00000000 00000000
%12d312fc  00000000 00000000 8000016e 00000000
##d
%12d3130c  0fe90000 00005453 00000325 00000000
%12d3131c  12d31228 0bff0002 00000000 00000000
%12d3132c  00000001 0000002e 00000000 00000000
%12d3133c  00000000 00000000 00000000 00000000
%12d3134c  00000000 00000000 00000000 00000000
%12d3135c  00000000 00000000 00000000 00000000
%12d3136c  00000000 00000000 00000000 00000000
%12d3137c  00000000 00000000 00000000 00000000
##.p2e
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 002e  001a 0002 001a 0009 blk 0500 ab911000 ab9c9408 ab9bc6c0 1ed0 12 turkey
##.s 2e
##.r
eax=80030059 ebx=00008000 ecx=00090000 edx=00000004 esi=ffffffff edi=12d3128c
eip=1bd0c8e1 esp=00293ea8 ebp=00090000 iopl=2 -- -- -- nv up ei pl zr na pe nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=12d3028c  cr3=001d4000
005b:1bd0c8e1 83c40c         add     esp,+0c
##ln
005b:1bd0c78c pmmerge:PM32BIT:SleepPmq + 155
005b:1bd0c940 CalcWakeBits - 5f
```

No one Owns the *User_Sem* since words at offsets +0x8 and +0xa are both zero.

We see that the system queue is held by slot 2e, who happens to be blocked in PMMERGE which is waiting for message activity. We also notice that at MQ+44 there is a non-zero value, which indicates that this thread has called WinSendMsg and is waiting for a response.

We investigate the WinSendMsg by examining the SMS pointed to by MQ+44

```
##dd %12d3ff5c
%12d3ff5c  00000000 12d3ff5c 00000000 00000000
%12d3ff6c  0097ec67 12d3128c 12d32cb4 00000000
%12d3ff7c  00000000 12d33168 00000071 00250016
%12d3ff8c  00000000 12d3ff5c 12d3ff5c 00000000
%12d3ff9c  00000000 0092e954 12d3910c 12d3ca34
%12d3ffac  00000000 00000002 12d34fac 00000407
%12d3ffbc  00000000 00000000 12d3ff90 12d3ff5c
%12d3ffcc  00000000 00000000 0092834a 12d3910c
```

The target MQ for the sent message is at offset +18, i.e. %12d32cb4

We find out who this is (the slot number is at MQ+a4).

```
##dd %12d32cb4
%12d32cb4  12d34940 00000020 0000000a 12d32d5c
%12d32cc4  12d32e9c 12d32d5c 12d32d5c 04002fff
%12d32cd4  04000400 0000001a 00000001 00000012
%12d32ce4  80030051 0093c378 00000000 00000000
%12d32cf4  00000000 00000000 00000000 00000010
%12d32d04  00000000 00000000 00000000 00000000
%12d32d14  00000000 00000000 00000000 00000000
%12d32d24  00000000 00000000 8000006c 00000000
##d
%12d32d34  0fe90000 00005453 00000325 00000000
%12d32d44  12d33304 0bff0000 00000000 12d3ff5c
%12d32d54  00000001 00000028 00000000 00000000
%12d32d64  00000000 00000000 00000000 00000000
%12d32d74  00000000 00000000 00000000 00000000
%12d32d84  00000000 00000000 00000000 00000000
%12d32d94  00000000 00000000 00000000 00000000
%12d32da4  00000000 00000000 00000000 00000000
##.p 28
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0028  001a 0002 001a 0001 crt 0500 ab905000 ab9c9408 ab9bbaf0 1f10 12 turkey
```

Offset +a4 gives us the slot number which turns out to be another thread of the turkey application. The status of this thread is *crt*. This indicates that some other thread in the same process has entered critical section, furthermore slot 28 would be ready to run had it not been for the critical section thread. Clearly this is why our application has hung the PM messaging function. The real culprit is the user of critical section, who is it?

The PTDA contains the address of the TCB in critical section. The TCB offset +0 contains the thread id followed by the slot number.

```
##dd %ab9c9408+ptda_ptcbcritsec-ptda_start l1
%ab9c96e8  ab9bc6c0
##dd %ab9bc6c0 l1
%ab9bc6c0  002e0009
##.p 2e
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
```

```
 002e# 001a 0002 001a 0009 blk 0500 ab911000 ab9c9408 ab9bc6c0 1ed0 12 turkey
```

Our application has perpetrated one, if not two, faults:

- First, we are using DosEnterCriticalSection in a PM application. This is a very heavy-handed way of serializing and likely to impact PM message processing, particularly if one of the other threads in the application holds the system queue lock.
- Secondly and more seriously, the thread that has entered critical section has subsequently called an API. The consequences of this are unpredictable and can lead to a hang as illustrated. Furthermore, this would apply whether or not the application was running in a PM environment.

How to Find the MQ of any Thread:

This example illustrates a basic technique for finding the MQ for a specific thread.

```
We find the MQ for thread slot 8:
##.p8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*0008# 0004 0001 0004 0001 blk 0500 ab596000 ab9c7020 ab988bf0 1ed0 01 pmshell
##dd %ab988bf0 +74
%ab988c64  00000000 00070000 00041304 12d2ca34
%ab988c74  12d2ded8 00000000 00000000 00000000
%ab988c84  00000000 00000000 00000000 00000000
%ab988c94  00000000 00000000 00000000 00000000
%ab988ca4  00000000 00000000 00000000 00000000
%ab988cb4  00000000 00000000 00000000 00000000
%ab988cc4  00000000 00000000 00000000 00000000
%ab988cd4  00000000 00000000 00000000 00000000
```

The TCB address is found from the `.P` output.

Offset +0x74 into the TCB is the saved thread local memory area.

Offset +0x0c into the TLMA are the AAB registers.

The first is the last PM error to occur on this thread. In this case severity 4 error code **1304**.

The next doubleword is the PMQ.

We can verify this by displaying it and checking the offset +0a4 is the same thread slot number.

Note: After fix-pack 7 the TCB in WARP is extended by 4 bytes. The TLMA begins at **TCB**+0x78.

```
##dd %12d2ca34
%12d2ca34  00000000 00000020 00000064 12d2cadc
%12d2ca44  12d2d75c 12d2cd3c 12d2cd3c 80002fff
%12d2ca54  80008000 00000004 00000001 00000001
%12d2ca64  80030038 0032bd01 000000ce 00000050
%12d2ca74  00000001 00000000 00000000 00000010
%12d2ca84  12d2c974 00000000 00000000 00000000
%12d2ca94  00000000 00000000 00000000 00000000
%12d2caa4  00000000 00000000 80000006 00000006
##d
```

```
%12d2cab4  00000000 00005453 0000024f 00000000
%12d2cac4  12d2c910 0bff0c02 00000000 00000000
%12d2cad4  00000001 00000008 80000007 00002f43
%12d2cae4  00000004 00000128 0000dc92 00010000
%12d2caf4  00000000 00000000 80000007 00002f43
%12d2cb04  00010001 00000128 0001168e 00010000
%12d2cb14  00000000 00000000 80000007 00002f43
%12d2cb24  00000004 00000128 0001168e 00010000
```

How to find the MQ of a BadApp Application:

This example illustrates how to find the MQ of the application that causes the *BadApp* dialog to appear.

As discussed in "Application Not Responding to Messages Logic" on page 291 **pmqsyslock**, **pmqfocus** and the **User_Sem** PM semaphore owner will be reset when the **BadApp** dialog is displayed.

To find the MQ of the bad application under these circumstances we look at the Query Hung Process Structure (QHPSTRCUT).

```
##db fbadappdialog l1
9f3f:0000035c 01
##dd pmqsyslock l1
9f3f:0000ed14  00000000
##dd pmqfocus l1
9f3f:0000e0fc  12d2b0f0
##dd %12d2b0f0
%12d2b0f0  12d2b344 00000020 0000000a 12d2b198
%12d2b100  12d2b2d8 12d2b198 12d2b198 00002fff
%12d2b110  00010001 00000004 0000000f 00000001
%12d2b120  8003004a 0032e98f 00000021 00000157
%12d2b130  00000001 00000000 00000000 00000000
%12d2b140  00000000 00000000 00000000 00000000
%12d2b150  00000000 00000000 00000000 00000000
%12d2b160  00000000 00000000 80000021 00000000
##d
%12d2b170  10ff0000 00005453 0000024f 00000000
%12d2b180  12d2b08c 0bff0002 00000000 00000000
%12d2b190  00000000 00000018 00000000 00000000
%12d2b1a0  00000000 00000000 00000000 00000000
%12d2b1b0  00000000 00000000 00000000 00000000
%12d2b1c0  00000000 00000000 00000000 00000000
%12d2b1d0  00000000 00000000 00000000 00000000
%12d2b1e0  00000000 00000000 00000000 00000000
##.p 18
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0018  0004 0001 0004 000f blk 0500 ab5b6000 ab9c7020 ab98ab70 1ed0 01 pmshell
```

The **fbadappdialog** is non-zero, which indicates that the **BadApp** dialog has been displayed.

The **pmqsyslock** is not owned.

The **pmqfocus** points to a shell thread, in fact the **BadApp** dialog thread.

So we look at *qhpsbadapp*

```
##dw qhpsbadapp l4
9f3f:0000e490  0002 000e 0008 0016

##.p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0001  0001 0000 0000 0001 blk 0100 ffe38000 ffe3aa04 ffe3a80c 1eb4 00 *ager
 0002  0001 0000 0000 0002 blk 0200 ab58a000 ffe3aa04 ab988020 1f3c 00 *tsd
 0003  0001 0000 0000 0003 blk 0200 ab58c000 ffe3aa04 ab988218 1f50 00 *ctxh
 0004  0001 0000 0000 0004 blk 081f ab58e000 ffe3aa04 ab988410 1f48 00 *kdb
 0005  0001 0000 0000 0005 blk 0800 ab590000 ffe3aa04 ab988608 1f20 00 *lazyw
 0006  0001 0000 0000 0006 blk 0800 ab592000 ffe3aa04 ab988800 1f3c 00 *asyncr
*0008# 0004 0001 0004 0001 blk 0500 ab596000 ab9c7020 ab988bf0 1ed0 01 pmshell
 000a  0004 0001 0004 0002 blk 0800 ab59a000 ab9c7020 ab988fe0 1ed4 01 pmshell
 000b  0004 0001 0004 0003 blk 0800 ab59c000 ab9c7020 ab9891d8 1ea8 01 pmshell
 000c  0004 0001 0004 0004 blk 0800 ab59e000 ab9c7020 ab9893d0 1ea8 01 pmshell
 000d  0004 0001 0004 0005 blk 0800 ab5a0000 ab9c7020 ab9895c8 1eb0 01 pmshell
 0010  0004 0001 0004 0006 blk 0200 ab5a6000 ab9c7020 ab989bb0 1edc 01 pmshell
 0011  0004 0001 0004 0007 blk 0200 ab5a8000 ab9c7020 ab989da8 1edc 01 pmshell
 0012  0004 0001 0004 0008 blk 0200 ab5aa000 ab9c7020 ab989fa0 1eb8 01 pmshell
 0007  0004 0001 0004 0009 blk 0200 ab594000 ab9c7020 ab9889f8 1ea8 01 pmshell
 0013  0004 0001 0004 000a blk 0800 ab5ac000 ab9c7020 ab98a198 1eb8 01 pmshell
 0014  0004 0001 0004 000b blk 0800 ab5ae000 ab9c7020 ab98a390 1eb8 01 pmshell
 0015  0004 0001 0004 000c blk 0800 ab5b0000 ab9c7020 ab98a588 1eb8 01 pmshell
 0016  0004 0001 0004 000d blk 0804 ab5b2000 ab9c7020 ab98a780 1ea8 01 pmshell
 0017  0004 0001 0004 000e blk 0804 ab5b4000 ab9c7020 ab98a978 1eb0 01 pmshell
 0018  0004 0001 0004 000f blk 0500 ab5b6000 ab9c7020 ab98ab70 1ea8 01 pmshell
 001a  0004 0001 0004 0010 blk 0200 ab5ba000 ab9c7020 ab98af60 1ed0 01 pmshell
 0009  0005 0004 0005 0001 blk 0800 ab598000 ab9c761c ab988de8 1eb4 00 harderr
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 000e  0005 0004 0005 0002 blk 0800 ab5a2000 ab9c761c ab9897c0 1ebc 00 harderr
 000f  0005 0004 0005 0003 blk 0800 ab5a4000 ab9c761c ab9899b8 1eb8 00 harderr
 0019  0007 0004 0007 0001 blk 0500 ab5b8000 ab9c8214 ab98ad68 1ed0 11 pmshell
 001c  0007 0004 0007 0002 blk 0800 ab5be000 ab9c8214 ab98b350 1edc 11 pmshell
 001d  0007 0004 0007 0003 blk 080a ab5c0000 ab9c8214 ab98b548 1edc 11 pmshell
 001e  0007 0004 0007 0004 blk 0800 ab5c2000 ab9c8214 ab98b740 1ed0 11 pmshell
 0020  0007 0004 0007 0005 blk 0800 ab5c6000 ab9c8214 ab98bb30 1edc 11 pmshell
 0021  0007 0004 0007 0006 blk 0200 ab5c8000 ab9c8214 ab98bd28 1edc 11 pmshell
 0025  0007 0004 0007 0007 blk 0200 ab5d0000 ab9c8214 ab98c508 1ed0 11 pmshell
 0026  0007 0004 0007 0008 blk 0200 ab5d2000 ab9c8214 ab98c700 1edc 11 pmshell
 0027  0007 0004 0007 0009 blk 0200 ab5d4000 ab9c8214 ab98c8f8 1edc 11 pmshell
 0029  0007 0004 0007 000b blk 0300 ab5d8000 ab9c8214 ab98cce8 1ed0 11 pmshell
 002a  0007 0004 0007 000c blk 021f ab5da000 ab9c8214 ab98cee0 1eac 11 pmshell
 002b  0007 0004 0007 000d blk 0200 ab5dc000 ab9c8214 ab98d0d8 1eb8 11 pmshell
 002f  0007 0004 0007 000e blk 0800 ab5e4000 ab9c8214 ab98d8b8 1ed0 11 pmshell
 001b  0006 0004 0006 0001 blk 0200 ab5bc000 ab9c7c18 ab98b158 1f00 10 pmspool
 001f  0006 0004 0006 0002 blk 0200 ab5c4000 ab9c7c18 ab98b938 1edc 10 pmspool
 0022  0006 0004 0006 0003 blk 0200 ab5ca000 ab9c7c18 ab98bf20 1e6c 10 pmspool
 0023  0006 0004 0006 0004 blk 0200 ab5cc000 ab9c7c18 ab98c118 1edc 10 pmspool
 0024  0006 0004 0006 0005 blk 0200 ab5ce000 ab9c7c18 ab98c310 1edc 10 pmspool
 0028  0008 0004 0008 0001 blk 0200 ab5d6000 ab9c8810 ab98caf0 1ed0 12 cometrun
 002c  0008 0004 0008 0002 blk 0801 ab5de000 ab9c8810 ab98d2d0 1edc 12 cometrun
 002d  0009 0004 0009 0001 blk 0200 ab5e0000 ab9c8e0c ab98d4c8 1ed0 13 fpwmon
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 002e  000b 0004 000b 0001 blk 0400 ab5e2000 ab9c9408 ab98d6c0 1eb8 15 cmd
 0030  000e 0004 000e 0001 blk 0200 ab5e6000 ab9c9a04 ab98dab0 1ed0 16 turkey
 0034  000e 0004 000e 0002 blk 0200 ab5ee000 ab9c9a04 ab98e290 1ed0 16 turkey
 0038  000e 0004 000e 0003 blk 0200 ab5f6000 ab9c9a04 ab98ea70 1ed0 16 turkey
 0037  000e 0004 000e 0004 blk 0200 ab5f4000 ab9c9a04 ab98e878 1ed0 16 turkey
 0031  000e 0004 000e 0005 blk 0200 ab5e8000 ab9c9a04 ab98dca8 1ed0 16 turkey
```

```
0032  000e 0004 000e 0006 blk 0200 ab5ea000 ab9c9a04 ab98dea0 1ed0 16 turkey
0033  000e 0004 000e 0007 blk 0200 ab5ec000 ab9c9a04 ab98e098 1ed0 16 turkey
0035  000e 0004 000e 0008 crt 0500 ab5f0000 ab9c9a04 ab98e488 1f10 16 turkey
0036  000e 0004 000e 0009 blk 0500 ab5f2000 ab9c9a04 ab98e680 1ed0 16 turkey
0039  000e 0004 000e 000a blk 0200 ab5f8000 ab9c9a04 ab98ec68 1ed0 16 turkey
003a  000e 0004 000e 000b blk 0200 ab5fa000 ab9c9a04 ab98ee60 1ed0 16 turkey
003b  000e 0004 000e 000c blk 0200 ab5fc000 ab9c9a04 ab98f058 1ed0 16 turkey
003c  000e 0004 000e 000d blk 0200 ab5fe000 ab9c9a04 ab98f250 1ed0 16 turkey
003d  000e 0004 000e 000e blk 0200 ab600000 ab9c9a04 ab98f448 1ed0 16 turkey
003e  000e 0004 000e 000f blk 0200 ab602000 ab9c9a04 ab98f640 1ed0 16 turkey
```

The QHPSTRUCT shows Tid 8, Pid e, flags 2 and SGID 16

.P shows this to be slot 35.

If we use the technique described in “How to Find the MQ of any Thread” on page 308 we will find the MQ for the bad application.

Finding Application and System Queue Elements:

This example shows how to find the queue element on both the system queue and an application queue.

A similar technique applies to both types of queue. The system queue header is located from the address at *psysqueue*. Location of application queue headers has been discussed in “How to Find the MQ of any Thread” on page 308.

The queue header contains the current read and write pointers, the queue element length and number of elements queued.

We illustrate this with the system queue in the following example:

```
##dd psysqueue l1
deff:00000000  1bdf0ac0
##dd %1bdf0ac0
%1bdf0ac0  00000000 0030001e 00000078 1bdf0ae4
%1bdf0ad0  1bdf18f4 1bdf1840 1bdf0fd0 00060000
%1bdf0ae0  00070007 00000072 00510196 000002fe
%1bdf0af0  00342420 1c0a9c00 01040040 00335362
%1bdf0b00  00700040 015c0000 000000c1 26cf0000
%1bdf0b10  1c000034 00401c0a 53c00104 00000033
%1bdf0b20  00000071 00c1015c 000082fe 003426cf
%1bdf0b30  1c0a9c00 01040040 003353ff 00700040
##dw %1bdf1840
%1bdf1840  0070 0000 0134 0050 0000 0000 1616 0034
%1bdf1850  8e00 0e7f 0040 0104 50f1 0033 0040 0071
%1bdf1860  0000 0134 0050 82fe 0000 172f 0034 1c00
%1bdf1870  1c0a 0040 0104 51cc 0033 0000 0072 0000
%1bdf1880  0134 0050 02fe 0000 180a 0034 9c00 1c0a
%1bdf1890  0040 0104 522a 0033 0040 0070 0000 019f
%1bdf18a0  0052 0000 0000 22c8 0034 1c00 1c0a 0040
%1bdf18b0  0104 5268 0033 0000 0071 0000 019f 0052
##
```

MQ+0x4 tells us 0x30 elements are queued, of length 0x1e bytes each.

MQ+0x14 is the current read pointer.

Displaying the queue from the current read pointer we can read off the first few message IDs since they are located at +0x0 of each entry: 70, 71, 72 and so on.

In an application queue the element length is 0x20.

14.1.3.8 PM Worked Examples under OS/2 2.x

Dealing with PM application problems under OS/2 2.x is similar to WARP. The principle difference being that the messaging and windowing function in PM is provided by the 16-bit DLL, PMWIN.DLL.

Most of the message structures are analogous to those of PMMERGE.DLL, their layouts are similar.

Under PMWIN.DLL most pointers are either offsets from a predefined segments or selectors. Thus where there are double-word pointers in PMMERGE.DLL structures, there are word length fields in PMWIN.DLL.

The following three symbol files are required for debugging PM applications problems under OS/2 2.x:

- PMWIN.SYM
- PMGRE.SYM
- PMSHAPI.SYM

A selection of useful symbols in the OS/2 2.x PM environment, with their equivalent OS/2 3.0 is listed below:

OS/2 3.0	OS/2 2.x
pmqlist	**smqlist**
pmqsyslock	**smqsyslock**
pmqfocus	**smqfocus**
pmqshell	**smqshell**
pmqshell2	**smqshell2**
pwndfocus	**pwndfocus**
fBadAppDialog	**fBadAppDialog**
qhpsBadApp	**qhpsBadApp**

Another significant difference between the two environments is in the calling conventions:

- PMMERGE APIs use the 32-bit C calling convention.
- PMWIN APIs use the 16-bit Pascal calling convention.

In effect this means that parameters on stacks and in some control blocks, are stored in reverse order.

There are four symbols that do not have equivalents in PMMERGE, these are:

winsel The selector for the AAB regs segment for a process.

selsms The selector for the SMS segment.

vphheapwnd The table of WND heap pointers.

fsrsuser The PM FastSafe RAMSEM, which is equivalent to the User_Sem PM Semaphore of PMMERGE.

We now run through a brief sequence of examples that illustrate:

"Finding an MQ and AAB Registers."

"Finding an SMS from an MQ" on page 314.

"Finding a WND from an HWND" on page 315.

"Finding a BadApp Process and MQ" on page 317.

"Finding the System Queue" on page 317.

Finding an MQ and AAB Registers:

```
##.s 8
##.p 8
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*0008# 0006 0001 0006 0001 blk 0500 7b936000 7bb460d0 7bb28a58 1eb8 01 pmshell
##dw winsel l1
fd17:00000032  003f
##dd 3f:0
003f:00000000  00000000 00000000 00000000 00000000
003f:00000010  00081037 0000ebe7 ff3f0000 00000000
003f:00000020  00000000 00000000 00000000 00000000
003f:00000030  00000000 00000000 00000000 00000000
003f:00000040  00000000 00000000 00000000 00000000
003f:00000050  00000000 00000000 00000000 00000000
003f:00000060  00000000 00000000 00000000 00000000
003f:00000070  00000000 00000000 00000000 00000000
##d
003f:00000080  00000000 00000000 00000000 00000000
003f:00000090  00000000 0000e92f 00000000 00000000
003f:000000a0  00000000 0000f827 00000000 00000000
003f:000000b0  00000000 0000e957 00000000 00000000
003f:000000c0  00000000 0000e94f 00000000 00000000
003f:000000d0  00081001 0000f81f e8ff0000 00000000
003f:000000e0  00000000 00000000 00000000 00000000
003f:000000f0  00000000 0000e947 00000000 00000000
##dd ebe7:0
ebe7:00000000  001a0000 00640000 0aaa0082 06e806e8
ebe7:00000010  80002fff 80018001 00010006 06d60001
ebe7:00000020  bf5108b3 02252549 016a0000 00010000
ebe7:00000030  00000000 00000004 00000000 00000000
ebe7:00000040  00000000 00000000 00000000 00000000
ebe7:00000050  00000000 00000000 09d70000 000023cc
ebe7:00000060  00000010 54530000 0000022a 018c0000
ebe7:00000070  1802ec6f 00000bff 00010000 00080000
##dw ebe7:0
ebe7:00000000  0000 001a 0000 0064 0082 0aaa 06e8 06e8
ebe7:00000010  2fff 8000 8001 8001 0006 0001 0001 06d6
ebe7:00000020  08b3 bf51 2549 0225 0000 016a 0000 0001
ebe7:00000030  0000 0000 0004 0000 0000 0000 0000 0000
ebe7:00000040  0000 0000 0000 0000 0000 0000 0000 0000
ebe7:00000050  0000 0000 0000 0000 0000 09d7 23cc 0000
ebe7:00000060  0010 0000 0000 5453 022a 0000 0000 018c
ebe7:00000070  ec6f 1802 0bff 0000 0000 0001 0000 0008
```

WinSel gives the AAB segment selector.

Note:

> WinSel is allocated in instance data, so must be viewed from a thread slot of the process in question.

Each entry is 0x10 bytes, one for each thread of the process.

The first entry is reserved.

The first doubleword of each entry is the past PM error for that thread and the second doubleword contains the selector for the MQ of that thread.

The key fields of interest in the MQ are:

Offset	Description
+0x0	chain pointer
+0x2	Queue element length
+0x4	number of elements queued
+0x6	Queue depth
+0x8	Top of queue
+0xa	Bottom of queue
+0xc	Current read pointer
+0xe	Current write pointer
+18	Pid
+1a	Tid
+1c	SGID
+30	SMS on which we are waiting a response
+32	SMS currently dispatched to our window procedure
+78	SMS at head of received list
+7e	thread slot id

Finding an SMS from an MQ

```
##dw smqsyslock l1
fd9f:000003d4  e55f
##dw e55f:0
e55f:00000000  e567 001a 0000 000a 0082 0186 0082 0082
e55f:00000010  a400 0006 0006 0006 003d 0009 001e 072a
e55f:00000020  08b3 db25 2549 00f5 0000 005d 0000 0001
e55f:00000030  0094 0000 0010 0000 0000 0000 0000 0000
e55f:00000040  0000 0000 0000 0000 0000 0000 0000 0000
e55f:00000050  0000 0000 0000 0000 0000 0000 4d24 0000
e55f:00000060  0000 0000 ec37 5453 061c 0000 0000 6c4c
e55f:00000070  ec6f 0002 0bff 0000 0000 0001 0000 0048
##.s 48
##.p 48
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0048# 003d 0006 003d 0009 blk 0500 7b9b6000 7bb51cc4 7bb2f758 1eb8 1e turkey
##dw selsms l1
```

```
fd9f:00001c2a  ec5f
##dw ec5f:94
ec5f:00000094  0000 0094 0000 0000 db25 2549 e55f e557
ec5f:000000a4  0000 0000 0020 0000 0000 0023 002b 0071
ec5f:000000b4  6d2c ec6f 0001 0024 0000 0000 3ae8 253a
ec5f:000000c4  e8df ebe7 0000 0000 0002 1b70 0410 0005
ec5f:000000d4  0000 0051 156c ec6f 0001 0038 01a8 0008
ec5f:000000e4  003c 003c 0082 0172 0000 0000 712c 1ec0
ec5f:000000f4  0172 0082 003c 003c 0172 0082 01ae 00be
ec5f:00000104  0000 0000 0000 0000 0002 0045 0003 0000

##dw e557:0
e557:00000000  e55f 001a 0000 000a 0082 0186 0082 0082
e557:00000010  2fff 0400 0400 0400 003d 000a 001e 072e
e557:00000020  08b3 98f9 2529 0000 0000 0000 0000 0000
e557:00000030  0000 0000 0010 0000 0000 0000 0000 0000
e557:00000040  0000 0000 0000 0000 0000 0000 0000 0000
e557:00000050  0000 0000 0000 0000 0000 0000 4dc0 0000
e557:00000060  0000 0000 ec37 5453 061c 0000 0000 6ce0
e557:00000070  ec6f 0002 0bff 0000 0094 0001 0000 0049
```

The thread with the system queue locked is waiting for a response to WinSendMsg. MQ+0x30 has the sent SMS offset.

The SMS selector is found from *selsms*.

The key fields in the SMS are:

Offset	Description
+0x0	Chain pointer offset
+0xc	Sending MQ selector
+0xe	Receiving MQ selector
+0x1a	Message Parameter 2
+0x1c	Message Parameter 1
+0x1e	Message Id
+0x20	Offset to WND
+0x22	Selector to WND

In this example the message has been sent to slot 49.

We see that the message has yet to be dispatched since it is still queued on the receive list (MQ+0x78).

Finding a WND from an HWND

```
##dw hwnddesktop l2
fd9f:0000053a  013c 0020

##dw vphheapwnd l2
fd9f:00001610  00ae ec6f
```

```
##dw ec6f:ae
ec6f:000000ae  0000 ec6f 0000 0000 0000 0000 0000 0000
ec6f:000000be  0000 0000 0000 0000 0000 0000 0000 0000
ec6f:000000ce  0000 0000 0000 0000 0000 0000 0000 0000
ec6f:000000de  0000 0000 0000 0000 0000 0000 0000 0000
ec6f:000000ee  0000 0000 0044 0000 0000 0000 0000 7098
ec6f:000000fe  ec6f 0000 0000 0000 0000 0000 0000 0000
ec6f:0000010e  0000 0000 0000 1ffc 0000 ebe7 0010 165e
ec6f:0000011e  fd2f 0065 1fad 0000 0000 0000 0000 0000


##dw ec6f:13c
ec6f:0000013c  0000 0000 0000 0000 1250 ec6f 0000 0000
ec6f:0000014c  0000 0000 0400 0300 0004 0000 a000 0000
ec6f:0000015c  1fd8 0000 ebe7 0020 3a24 fd4f 0065 1fad
ec6f:0000016c  0000 0000 0000 0000 0000 0000 0000 0000
ec6f:0000017c  0000 0000 0000 0000 0000 0000 0000 0048
ec6f:0000018c  0354 ec6f 00f4 ec6f 0000 0000 0000 0000
ec6f:0000019c  0000 0000 0000 0000 0000 0000 8000 0000
ec6f:000001ac  1e4c 0000 ebe7 0030 1d42 fd2f 0065 1fad
```

In this example we find the WND for the desktop from the HWND which is stored at *hwnddesktop*.

The HWND comprises an offset then concatenated with an identifier, the low order nibble of which is a heap index. Thus, for the desktop:

```
##dw hwnddesktop l2
fd9f:0000053a  013c 0020
               .    . .
               .    . .
               .    . .
       offset...    . .
       id..........  .
       index...........
```

vphheapwnd points to a table of heaps. Each entry is a far pointer and there are at most 16. The index nibble of the HWND is used to select the heap pointer. In this example there is just one entry: *ec6f:0000*

We use the offset from the HWND with the heap selector to get the PWND. In this case *ec6f:13c*.

The key fields of interest in the WND are:

Offset	Description
+0x0	Next Sibling WND far pointer
+0x4	Parent WND far pointer
+0x8	Child WND far pointer
+0xc	Owner WND far pointer
+0x24	MQ selector that services this window
+0x26	ID and Index portion of the HWND for this WND.
+0x28	16-bit far pointer to the Window Procedure.
+0x2c	32-bit pointer to the Window Procedure.

Finding a BadApp Process and MQ

```
##db fbadappdialog l1
%1f8d07ae 01
##dw qhpsbadapp
%1f8d16b8  0002 003d 000a 001e 000c 0000 0000 0000
%1f8d16c8  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d16d8  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d16e8  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d16f8  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d1708  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d1718  0000 0000 0000 0000 0000 0000 0000 0000
%1f8d1728  0000 0000 0000 0000 0000 0000 0000 0000
```

The fields of the QHPSTRUCT are in the same order to the PMMERGE version:

Offset	Description
+0x0	Flags
+0x2	Pid
+0x4	Tid
+0x6	SGID

Finding the System Queue

```
##dw mqhsysqueue
fd87:00000000  0000 001c 001a 0078 0ad6 17f6 0ee2 11ba
fd87:00000010  0000 0006 0007 0007 0000 0001 0001 ebe7
fd87:00000020  06d6 0008 0ba5 0000 0000 e55f 072a 0048
fd87:00000030  000f 000b 0002 0000 0300 0000 0000 0000
fd87:00000040  0000 fffe ee6f 2549 0000 0000 0000 0588
fd87:00000050  0339 032c 00ac 00bc 00cc 00dc 00ec 00fc
fd87:00000060  010c 012c 011c 013c 014c 015c 0000 0000
fd87:00000070  0000 0000 0000 0000 0000 0000 0000 0000
##dw ee2
fd87:00000ee2  0072 00f5 005d 02fe 0000 dbc2 2549 8000
fd87:00000ef2  0000 8000 0000 9150 08a8 08a8 0070 0032
fd87:00000f02  00ec 0000 0000 dfaa 2549 0000 c3b7 fff6
fd87:00000f12  2002 0000 0012 08a8 0071 0032 00ec 83fe
fd87:00000f22  0000 e007 2549 fff6 2002 0000 0012 08a8
fd87:00000f32  e007 08a8 0072 0032 00ec 03fe 0000 e065
fd87:00000f42  2549 2006 0000 0016 0000 2006 08a8 08a8
fd87:00000f52  0071 0032 00ec 82fe 0000 e0c3 2549 0000
```

mhsysqueue points directly at the system queue.

The queue pointers are offsets from the same segment as the MQ.

The current read pointer at +0x0e, is 0x0ee2.

The queue entry length at +0x02, is 0x001c.

Displaying the queue from the read pointer shows the first few elements queued are for message IDs, 72, 70, 71, 72 and so on.

Each entry on the system queue is a SQMSG. The key fields are:

Offset	Description
+0x0	Message ID
+0x2	Message Parameter 1
+0x6	Message Parameter 2

For an application queue, the entries are QMSG structures. the key fields of these are:

Offset	Description
+0x0	HWND
+0x4	Message Id
+0x8	Message Parameter 1
+0xc	Message Parameter 2

14.1.4 Dump Analysis of Loops in Ring 0 Code

Ring zero loops can sometimes be successfully analyzed from a dump. The trick is knowing how to locate the register set at the time the dump was taken.

The Dump Formatter only implements the `.R` command, which obtains the registers from a stack frame on the thread′s ring 0 stack. Under the kernel Debugger there is no problem: the `R` command will display the current system registers.

Note:

> If a thread never runs in User Mode, such as the internal Pid 0 threads then a stack frame is never built and `.R` will be unsuccessful in formatting the registers.

Fortunately there is a way of obtaining the current registers:

- When a dump is initiated using Ctrl-Alt-NumLock-NumLock a keyboard interrupt is initiated by the processor hardware.
- Via the IDT control passes to the interrupt router who is responsible for switching to the interrupt stack before passing control to the appropriate interrupt handler.
- The interrupt router checks to see if the system is already running from the interrupt stack.
  - If it isn′t then an interrupt stack frame is built on the current stack and the stack frame pointer is saved in *fpoldstack.* Then the SS selector is switched to the interrupt stack selector (E8).
  - If it is then a nested interrupt has occurred and the interrupt stack frame is built on the interrupt stack itself.

It is from *fpoldstack* that we are able to obtain the registers before any interrupt occurred. The following debug log illustrates this and many of the techniques previously discussed.

14.1.4.1 Ring 0 Loop Dump Analysis Example

This example finds a loop in a file system driver from a system dump. For reference, we note the format of the interrupt stack frame as pointed by *fpoldstack* as follows:

+0x0	Current interrupt level when prior to interrupt.
+0x4	GS
+0x8	FS
+0xc	ES
+0x10	DS
+0x14	EDI
+0x18	ESI
+0x1c	EBP
+0x20	padesp
+0x24	EBX
+0x28	EDX
+0x2c	ECX
+0x30	EAX
+0x34	pad
+0x3c	EIP
+0x40	CS
+0x44	EFLAG
+0x48	ESP
+0x4c	SS
+0x4c	SS

```
>>Who's the current thread?
# .p*
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*00a3# 006c 000a 006c 0001 run 0200 7b720000 7bb025c0 7ba8f9e0 0894 25 FRNOLBMG

>>Probably a loop of some kind, could be a hot I/O or even dispatcher
bug (unlikely).

>> Where are we?
# .r
eax=001fe624 ebx=00002022 ecx=00000029 edx=00000007 esi=00000000 edi=0003e77c
eip=00000179 esp=0003e624 ebp=0003e68c iopl=2 -- -- -- nv up ei pl nz na po nc
cs=d02f ss=001f ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001bb000
d02f:00000179 66ea4102021a5b00 jmp     005b:1a020241
# .m

*har     par      cpg        va    flg next prev link hash hob   hal
 00b1 %fee13f40 00000010 %1a050000 3d9 00b0 00b2 0000 0000 00bd 0000  hco=02c2c
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 00bd  00b1 0000  0838 00bb 00bb  0000 00  00 00 00 shared    c:doscall1.dll
 hco=2c2c pco=ffe71cf7  hconext=02c20 hptda=18c9 f=1c  pid=00bc c:cmd.exe

>> We are in DOSCALL1. Let's see what function was called.
```

```
# dw ss:bp
001f:0000e68c  e6bc 0003 ae60 1a02 e77c 0003 e728 0003
001f:0000e69c  e70c 0003 0000 0000 0000 0000 0010 0000
001f:0000e6ac  2022 0000 0000 0000 28a4 111b e697 0003
001f:0000e6bc  e740 0003 4a6d 1113 e77c 0003 e728 0003
001f:0000e6cc  e70c 0003 0000 0000 0000 0000 0010 0000
001f:0000e6dc  2022 0000 0000 0000 0010 0000 ab18 1111
001f:0000e6ec  e77c 0003 0002 0000 0000 0000 0000 0000
001f:0000e6fc  0001 0000 0007 0000 0000 0000 0180 0000


# ln %11134a6d
No Symbols Found

# .m  %11134a6d

*har     par      cpg        va    flg next prev link hash hob   hal
 0e82 %fee26f36 00000030 %11110000 3d9 09af 09f1 0000 0000 15b1 0000  hco=02c17
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 15b1  0e82 0000  0838 08fe 08fe  0000 00  00 00 00 shared    c:frnococl.dll
 hco=2c17 pco=ffe71c8e  hconext=02de0 hptda=1938 f=1c  pid=00bb c:frnosa.exe


# .lmo 8fe
hmte=08fe pmte=%fdef0c68 mflags=4498b186 c:\frnv1r0\dll\frnococl.dll
obj   vsize    vbase    flags   ipagemap cpagemap hob  sel
0001 00026ee8 11110000 80002025 00000001 00000027 15b1 888f r-x shr big
0002 0000001f 11180000 80001025 00000028 00000001 0caf 88c7 r-x shr alias
0003 00000030 11190000 80002025 00000029 00000001 15a4 88cf r-x shr big
0004 00000008 111a0000 80001003 0000002a 00000000 0000 88d7 rw- alias
0005 00004e74 111b0000 80002003 0000002a 00000005 0000 88df rw- big


# u  %11134a6d-10%11134a5d 085657             or      byte ptr [esi+57],dl
%11134a60 6a00              push    +00
%11134a62 51                push    ecx
%11134a63 8b4dac            mov     ecx,dword ptr [ebp-54]
%11134a66 52                push    edx
%11134a67 51                push    ecx
%11134a68 e8af63ef08        call    %1a02ae1c
%11134a6d 8bc8              mov     ecx,eax
%11134a6f 8b45ac            mov     eax,dword ptr [ebp-54]
%11134a72 83c420            add     esp,+20         ;' '
%11134a75 894ddc            mov     dword ptr [ebp-24],ecx
%11134a78 83f904            cmp     ecx,+04

# ln %1a02ae1c
%1a02ae1c DOSCALL1 DOS32OPEN

>> So we're in DOSOPEN

>> We've almost certainly call-gated into the kernel.
>> Check this out ...

# u cs:eip -10
d02f:00000169 268805        mov     byte ptr es:[di],al
d02f:0000016c 8bc3          mov     ax,bx
```

```
d02f:0000016e 8cc2           mov     dx,es
d02f:00000170 5f             pop     di
d02f:00000171 c9             leave
d02f:00000172 c3             ret
d02f:00000173 90             nop
d02f:00000174 9a00000a1b     call    1b0a:0000
d02f:00000179 66ea4102021a5b00 jmp     005b:1a020241
d02f:00000181 9a00001a1b     call    1b1a:0000
d02f:00000186 66ea3b05021a5b00 jmp     005b:1a02053b
d02f:0000018e 9a0000631b     call    1b63:0000
# dg 1b0a
1b0a  CallG32 Sel:Off=0148:0000529f     DPL=3 P  DWC=8
# ln 148:529f
0148:0000529f OS2KRNL DOSOPEN2
#

>> Yes, that's where we are.
>> Now let's see if we can find out where in R0 DOSOPEN has got...

# dw interruptlevel l1
0400:00006382  0000
# dd currintlevel l1
0148:0000529f OS2KRNL DOSOPEN2
#
>> So, no nested interrupts, but we are handling one
>> (interruptlevel=0000).
>> The current interrupt came from IRQ 1 (currintlevel=1)
>> So a keyboard interrupt, not surprising because the customer was
>> asked to take a dump using ctrl-alt-numlock-numlock, furthermore
>> he obeyed!

>> Lets look at the interrupt stack saved by the interrupt router
>> (prior to switching stacks).

# dd fpoldstack l2
%fff27310  00004b0c 00000030

>> So, the Interrupt Stack Frame maps the frame at 30:4b0c

# dw 30:4b0c
0030:00004b0c  ffff ffff 0000 0000 03b8 0000 2410 0000
0030:00004b1c  23f8 0000 4c66 7b61 0000 7b61 4b5e 0003
0030:00004b2c  4b40 0000 0000 0000 24d0 0000 ffff 0000
0030:00004b3c  0000 0000 0000 0000 0000 0000 be93 0000
0030:00004b4c  23d0 0000 2292 0000 23f8 0000 006c 0000
0030:00004b5c  00a3 4b8c b541 23d0 23f8 0000 4c66 0000
0030:00004b6c  4baa 0029 2b00 0000 0004 0001 2101 2d76
0030:00004b7c  0001 0094 0022 003f 0000 0000 20d4 4be8

>> bp is at +1c, sp at +20,  cs at +40, eip at +3c
>> We took the dump at when cs:eip=23d0:be93
>> who is this?

# .m 23d0:0

*har     par      cpg        va    flg next prev link hash hob   hal
 0021 %fee132e0 00001400 %fc953000 121 0020 0022 0000 0020 0022 0000      =0000
 hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
 0022  0021 0000  0225 ffef 0000  0000 00  04 00 00 vmkshrw
```

```
>> 23d0 is allocated out of the Kernel Swappable Read/Write heap.
>> Lets see who owns this heap block. Need to look at the VMKSHB
>> shared heap block header...

# dg 23d0
23d0  Code    Bas=fca15000 Lim=0000ff5f DPL=0 P  RE    A
# dw %face15000-10
%fca14ff0  0000 0000 0000 0000 ff68 5200 ff4d 23d0
%fca15000
Invalid linear address: %fca15000

>> VMKSHB is an 8 byte prefix of the form:
>> ulong size || 0x520000
>> ushort hob
>> ushort sel

>> check the owner hob

# .mo ff4d
ff4d fsd6
#

>> This was allocated by/for the 6th loaded FSD.
>> N.B fsd8 is used for the 8th and subsequent FSDs
>> in the same way dd16 is used for the 16th and subsequent
>> device driver.

>> Who is fsd6?
>> There are two ways to home in on this.

>> First method.
>> List all library modules (includes DLLs IFSs FONs etc)

# .lml
hmte=18e2 pmte=%fe1a1000 mflags=0498b188 c:\frnv1r0\dll\frnolgar.dll
hmte=193e pmte=%fe1a13ac mflags=0498b188 c:\frnv1r0\dll\frnosars.dll
hmte=18fd pmte=%fe1a18d4 mflags=4498b186 c:\frnv1r0\dll\frnofo2.dll
hmte=1933 pmte=%fe1a194c mflags=4498b186 c:\frnv1r0\dll\frnofios.dll
hmte=18f2 pmte=%fdebd1c4 mflags=4498b186 c:\frnv1r0\dll\frnoutil.dll
hmte=0fc7 pmte=%fdedf550 mflags=4498b1c6 c:\frnv1r0\dll\frnollmn.dll
.
.
>> 200 hundred lines later
.
.
hmte=0b0a pmte=%fe0bb6f8 mflags=0408b1c8 c:\cmlib\dll\acshpres.dll
hmte=0af9 pmte=%fe0bb864 mflags=0408b1c8 c:\cmlib\redj.pml
hmte=0af6 pmte=%fe0bb8f0 mflags=0408b1c8 c:\cmlib\redj.pdl
hmte=08f7 pmte=%fe0bcf8c mflags=0408b1c8 c:\cmlib\redj2.pml
hmte=094d pmte=%fe11dc44 mflags=0408b1c8 c:\cmlib\redj2.pdl
hmte=0a8a pmte=%fe098ea4 mflags=0408b1c8 c:\cmlib\cm20sys.pml
hmte=0a87 pmte=%fe098f5c mflags=0408b1c8 c:\cmlib\cm20sys.pdl
hmte=04c7 pmte=%fde5ff60 mflags=0498b1c8 c:\os2\dll\times.fon
hmte=04c5 pmte=%fde5fa7c mflags=0498b1c8 c:\os2\dll\helv.fon
hmte=04c3 pmte=%fde5fb44 mflags=0498b1c8 c:\os2\dll\courier.fon
hmte=04c1 pmte=%fde5fbb4 mflags=0498b1c8 c:\os2\dll\sysmono.fon
hmte=04b9 pmte=%fde5fc8c mflags=4498b1c5 c:\os2\dll\pmatm.dll
hmte=0324 pmte=%fe134f54 mflags=0428a1c9 d:\ibm3995\demoifs.ifs
```

```
hmte=02ff pmte=%fe0fff90 mflags=0428a1c9 c:\netw\nwifs.ifs
hmte=01b9 pmte=%fe0dbcb4 mflags=0428a1c9 c:\showcase\sdcfs.ifs
hmte=0109 pmte=%fe0befa0 mflags=0428a1c9 c:\os2\cdfs.ifs
hmte=00e0 pmte=%fde46d3c mflags=0428a1c9 c:\os2\hpfs.ifs
hmte=0076 pmte=%fde14f68 mflags=0428a1c9 a:\mini_fsd.fsd

>> FSDs were installed in order, 76, e0, 109, 1b9, 2ff and 324.

>> fsd6 is therefore hmte=324. Lets check for certain:

# .lmo 324
hmte=0324 pmte=%fe134f54 mflags=0428a1c9 d:\ibm3995\demoifs.ifs
seg  sect psiz vsiz hob  sel  flags
0001 0003 d7ba d7ba 0000 2398 8d60 code shr prel rel
0002 0070 2325 2325 0000 23a0 8d60 code shr prel rel
0003 0083 ec66 ec66 0000 23a8 8d60 code shr prel rel
0004 00fa f7a6 f7a6 0000 23b0 8d60 code shr prel rel
0005 0176 b1e4 b1e4 0000 23b8 8d60 code shr prel rel
0006 01d0 f3e6 f3e6 0000 23c0 8d60 code shr prel rel
0007 024b eeda eeda 0000 23c8 8d60 code shr prel rel
0008 02c3 ff60 ff60 0000 23d0 8d60 code shr prel rel
0009 0343 fe82 fe82 0000 23d8 8d60 code shr prel rel
000a 03c3 4b4e 4b4e 0000 23e0 8d60 code shr prel rel
000b 03e9 002a 002a 0000 23e8 8c60 code shr prel
000c 03ea 4298 4298 0000 23f0 8d60 code shr prel rel
000d 040c 9dfd 9dfe 0000 23f8 8d41 data prel rel
000e 0000 0000 1964 0000 2400 8c41 data prel
000f 0000 0000 fe88 0000 2408 8c41 data prel
0010 0000 0000 d75e 0000 2410 8c41 data prel

>> and yes we find selector 23d0 in object 8 of demoifs.ifs

>> Second method
>> We could approach this from the FSC control block, which is similar
>> in purpose to the DD header chain.
>> The FSC is a table of FSD entry point tables. We might spot the
>> selector in question being referenced in the FSC. If not, we can
>> unwind the R0 stack until we do find a reference.

>> First the FSC. Dump the SAS for the FSC selector

# .a
--- SAS Base Section ---
                      SAS signature: SAS
           offset to tables section: 0016
       FLAT selector for kernel data: 0158
     offset to configuration section: 001E
     offset to device driver section: 0020
    offset to Virtual Memory section: 002C
          offset to Tasking section: 005C
              offset to RAS section: 006E
      offset to File System section: 0074
          offset to infoseg section: 0080
--- SAS Protected Modes Tables Section ---
                   selector for GDT: 0008
                   selector for LDT: 0000
                   selector for IDT: 0018
               selector for GDTPOOL: 0100
--- SAS Device Driver Section ---
```

```
    offset for the first bimodal dd: 0CB9
  offset for the first real mode dd: 0000
      sel for Drive Parameter Block: 0520
       sel for ABIOS prot. mode CDA: 0468
        seg for ABIOS real mode CDA: 6800
                   selector for FSC: 00C8
--- SAS Task Section ---
          selector for current PTDA: 0030
  FLAT offset for process tree head: FFF29714
 FLAT address for TCB address array: FFF26BDA
      offset for current TCB number: FFE23A0E
             offset for ThreadCount: FFE23A12
--- SAS File System Section ---
                handle to MFT PTree: FDE55FB4
     selector for System File Table: 00C0
     sel. for Volume Parameter Bloc: 0678
   sel. for Current Directory Struc: 06A8
        selector for buffer segment: 00A8
--- SAS Information Segment Section ---
       selector for global info seg: 0428
   address of curtask local infoseg: 03B80000
      address of DOS task's infoseg: FFFFFFFF
         selector for Codepage Data: 06BB
--- SAS RAS Section ---
selector for System Trace Data Area: 0508
 segment for System Trace Data Area: 0508
        offset for trace event mask: 09D6
--- SAS Configuration Section ---
    offset for Device Config. Table: 0D40
--- SAS Virtual Memory Mgt. Section ---
       Flat offset of arena records: FFF2C314
      Flat offset of object records: FFF2C32C
     Flat offset of context records: FFF2C31C
  Flat offset of kernel mte records: FFF27E68
     Flat offset of linked mte list: FFF273B8
    Flat offset of page frame table: FFF2A768
    Flat offset of page range table: FFF29CC0
    Flat offset of swap frame array: FFF260B0
           Flat offset of Idle Head: FFF294D4
           Flat offset of Free Head: FFF294C4
          Flat offset of Heap Array: FFF2A770
     Flat offset of all mte records: FFF2BE24
#

>> FSC selector is c8. Now dump the FCS segment.

# dw c8:0
00c8:00000000  03c8 0000 0000 fde1 0b68 0738 0b6c 0738
00c8:00000010  0000 0720 01fc 0720 0010 0720 05b4 0718
00c8:00000020  0570 0720 0580 0720 0634 0720 0640 0720
00c8:00000030  0e3c 0720 1120 0720 0834 0720 090c 0720
00c8:00000040  09f8 0718 1130 0720 1f24 0720 1f6e 0720
00c8:00000050  2122 0720 16e4 0720 1b10 0720 1b38 0720
00c8:00000060  1bec 0720 1dc8 0720 0c60 0718 0d70 0718
00c8:00000070  1f14 0720 215c 0720 22a0 0720 2294 0720
# d
00c8:00000080  111c 0718 25fc 0720 26b0 0720 117c 0718
00c8:00000090  0fdc 0718 0000 0718 0000 0000 0000 0000
00c8:000000a0  0000 0000 03b0 0720 062c 0718 137c 0720
```

```
00c8:000000b0  0bb4 0718 26bc 0720 0000 0000 0000 0000
00c8:000000c0  0000 0000 0000 0000 0000 0a58 0004 0a58
00c8:000000d0  0000 0a50 01b6 0a50 000e 0a50 04c4 0a50
00c8:000000e0  060e 0a50 061c 0a50 065c 0a50 0666 0a50
00c8:000000f0  1198 0a50 1224 0a50 086e 0a50 0e0e 0a50
# d
00c8:00000100  09e6 0a50 125a 0a50 299a 0a50 29a8 0a50
00c8:00000110  29b6 0a50 263e 0a50 278c 0a50 27aa 0a50
00c8:00000120  2842 0a50 288a 0a50 28fc 0a50 298c 0a50
00c8:00000130  297e 0a50 29c4 0a50 2cf6 0a50 2cb8 0a50
00c8:00000140  2f0c 0a50 34b2 0a50 34f2 0a50 350c 0a50
00c8:00000150  5029 1000 9cab 0140 1232 0a50 1240 0a50
00c8:00000160  0000 0000 0602 0a50 9d8c 0140 1568 0a50
00c8:00000170  124e 0a50 3500 0a50 0000 0000 0000 0000
# d
00c8:00000180  0000 0000 0000 0000 0090 1028 008a 1028
00c8:00000190  00be 1018 03e6 1018 05da 1018 0766 1018
00c8:000001a0  09ee 1018 0c38 1018 0db6 1018 0dc2 1018
00c8:000001b0  0fa4 1018 1488 1018 1496 1018 158a 1018
00c8:000001c0  1998 1018 1cae 1018 1f92 1018 1fa0 1018
00c8:000001d0  1fae 1018 2998 1018 2bf0 1018 2c9c 1018
00c8:000001e0  2caa 1018 2f90 1018 2f9e 1018 31c4 1018
00c8:000001f0  332c 1018 333a 1018 3950 1018 3e9c 1018
# d
00c8:00000200  3fa2 1018 419a 1018 4318 1018 4332 1018
00c8:00000210  5029 1000 9cab 0140 0000 0000 0000 0000
00c8:00000220  0000 0000 08aa 1018 9d8c 0140 2114 1018
00c8:00000230  1fbc 1018 4326 1018 0000 0000 0000 0000
00c8:00000240  0000 0000 0000 0000 0098 22a8 0090 22a8
00c8:00000250  01b0 22b0 1500 22b0 7470 22b0 1650 22b0
00c8:00000260  18e0 22b0 51b0 22b0 25d0 22b0 344d 22b0
00c8:00000270  3ca7 22b0 469a 22b0 28ac 22b0 279b 22b0
# d
00c8:00000280  4316 22b0 28c5 22b0 4343 22b0 4347 22b0
00c8:00000290  434d 22b0 3b30 22b0 43c0 22b0 4353 22b0
00c8:000002a0  195d 22b0 4310 22b0 6e50 22b0 4c10 22b0
00c8:000002b0  4d20 22b0 4ec6 22b0 5bc2 22b0 4359 22b0
00c8:000002c0  6d27 22b0 1aab 22b0 43a1 22b0 8740 22b0
00c8:000002d0  5029 1000 9cab 0140 4a70 22b0 4a7f 22b0
00c8:000002e0  84c0 22b0 17bb 22b0 9d8c 0140 37f0 22b0
00c8:000002f0  25f0 22b0 7570 22b0 0000 0000 0000 0000
# d
00c8:00000300  0000 0000 0000 0000 0c96 23f8 8880 23f8
00c8:00000310  00de 23a0 0152 23a0 01bd 23a0 0228 23a0
00c8:00000320  02f2 23a0 0369 23a0 03d1 23a0 042d 23a0
00c8:00000330  049e 23a0 050f 23a0 0580 23a0 05df 23a0
00c8:00000340  066b 23a0 06eb 23a0 075f 23a0 07b5 23a0
00c8:00000350  083e 23a0 0982 23a0 09e7 23a0 0a4c 23a0
00c8:00000360  0acf 23a0 0b40 23a0 0bab 23a0 0c1f 23a0
00c8:00000370  0c87 23a0 0cfb 23a0 0d8d 23a0 0e04 23a0
# d
00c8:00000380  0e5d 23a0 0ecb 23a0 0f33 23a0 0f92 23a0
00c8:00000390  5029 1000 9cab 0140 0000 0000 0000 0000
00c8:000003a0  0000 0000 028d 23a0 9d8c 0140 0905 23a0
00c8:000003b0  08ac 23a0 1000 23a0 0000 0000 0000 0000
00c8:000003c0  0000 0000 0000 0000
Past end of segment: 00c8:000003c8
#
>> The FSC starts with an 8 byte header. Word 1 is the length.
```

```
>> Each entry is for each FSD starting with fsd2 (fsd1 is OS2BOOT
>> and not used once the kernel is loaded). Each FSD entry comprises
>> a table of far16 pointers. The first two are a) pointer to FSD
>> attributes and b) FSD name. The remaining are the function entry
>> points (See IFS OEM reference). There are 46 of these. In other
>> words the first fsd entry is at c8:8 and ever subsequent entry is
>> every 12 lines of display. fsd 6 entry starts at c3:308

>> what's fsd6 called?

# db 23f8:8880 l8
23f8:00008880 4f 50 54 4c 49 42 00 00                         OPTLIB..

>> The evidently is the optical library fsd.
>> We didn't find the current cs:eip in the fsd function table so
>> we unwind the r0 stack ....


# dw 30:4b8c l8
0030:00004b8c  4be8 7426 23a8 4bb2 0030 4bb0 0030 4bd8
# dw 30:4be8 l8
0030:00004be8  4c0a 738a 23a8 0000 1004 0000 2f48 4c2c
# dw 30:4c0a l8
0030:00004c0a  4c3a bb28 23a8 d7ce 0001 4c2c 0030 0000
# dw 30:4c3a l8
0030:00004c3a  4c6c 1b8f 23b0 0000 0000 0000 d7ce 0001
# dw 30:4c6c l8
0030:00004c6c  4c7e 1e87 23b0 0300 24f8 4ca8 0030 23f8
# dw 30:4c7e l8
0030:00004c7e  4cca 08ed 23b0 0300 24f8 4ca8 0030 4cb0
# dw 30:4cca l8
0030:00004cca  4cf6 011e 23b0 0100 24f0 0073 2520 0000
# dw 30:4cf6 l8
0030:00004cf6  4d36 7314 23b0 04d3 2408 0073 2520 0000
# dw 30:4d36 l8
0030:00004d36  4d70 5be8 23b0 006a 2520 0000 04d3 2408
# dw 30:4d70 l8
0030:00004d70  4dbc 04e3 23a8 0000 04d3 2408 0020 2022
# dw 30:4dbc l8
0030:00004dbc  4e46 2b1f 2398 4eb6 0030 0000 0003 0020
# dw 30:4e4d l8
0030:00004e4d  304e 0000 0300 2000 bc00 304e 1000 2200
# dw 30:304e l8
Invalid linear address: 0030:0000304e

>> The problem here is that the kernel is not using ebp
>> before calling the fsd. So dump the R0 stack from
>> the last recognisable fsd selector. Look for the
>> first selector that matches one used in fsd6's
>> function table.

dw 30:4dbc
0030:00004dbc  4e46 2b1f 2398 4eb6 0030 0000 0003 0020
0030:00004dcc  4ebc 0030 0010 2022 0000 41a5 2cf0 4df2
0030:00004ddc  0030 ffff 04d0 2408 4edb 0030 0000 2f40
0030:00004dec  23f8 8bfc 0308 2022 0000 1004 8ce8 1c63
0030:00004dfc  8ce8 1c63 8ce8 1c63 0000 0000 0000 0000
0030:00004e0c  0000 006c 0000 0274 0000 0000 0000 2100
0030:00004e1c  0001 4e3e 0006 0004 0000 2f40 039c 2408
```

```
0030:00004e2c  0345 0098 006c 0003 0001 0000 04d0 2408
# d
0030:00004e3c  02f4 0305 0098 2f40 1004 4e84 0d6e 23a0
0030:00004e4c  4eb6 0030 0000 0003 0020 4ebc 0030 0010
0030:00004e5c  2022 0000 41a5 2cf0 41d7 2cf0 ffff 4fee
0030:00004e6c  0030 4edb 0030 4ee3 0030 0308 8bfc 510d
0030:00004e7c  0000 9410 00c8 0030 4ec0 9a09 0140 4eb6
0030:00004e8c  0030 0000 0003 0020 4ebc 0030 0010 2022
0030:00004e9c  0000 41a5 2cf0 41d7 2cf0 ffff 4fee 0030
0030:00004eac  4edb 0030 4ee3 0030 0000 0000 04d0 0007

>> at 30:4e48 we have 23a0:d6e. Looking at the function
>> table we see entry point 26 at 23a0:cfb is the closest.
>> fsd entry point 26 is FS_OPENCREATE. This seems to be
>> consistent with what ring 3 was doing.
>> Finally for future reference the FSD entry structure is
>> as follows:

>>+0   FS_ATTRIBUTE;  /* -> FSD attribute. (in FSD memory) */
>>+4   FS_NAME;       /* -> FSD name.      (in FSD memory) */
>>+8   FS_ATTACH;    /* DosQFsAttach, DosFsAttach */
>>+c   FS_CHDIR;  /* DosChdir */
>>+10  FS_CHGFILEPTR; /* DosChgFilePtr */
>>+14  FS_CLOSE;  /* DosClose */
>>+18  FS_COPY;  /* DosCopy */
>>+1c  FS_DELETE;  /* DosDelete */
>>+20  FS_EXIT;  /* DosExit */
>>+24  FS_FILEATTRIBUTE; /* DosFileInfo, DosSetFileMode */
>>+28  FS_FILEINFO;  /* DosQFileInfo, DosSetFileInfo */
>>+2c  FS_FILEIO;  /* DosFileIO */
>>+30  FS_FINDCLOSE;  /* DosFindClose */
>>+34  FS_FINDFIRST;  /* DosFindFirst */
>>+38  FS_FINDFROMNAME; /* DosFindFromName-Private to server */
>>+3c  FS_FINDNEXT;  /* DosFindNext */
>>+40  FS_FINDNOTIFYCLOSE; /* DosFindNotifyClose */
>>+44  FS_FINDNOTIFYFIRST; /* DosFindNotifyFirst */
>>+48  FS_FINDNOTIFYNEXT; /* DosFindNotifyNext */
>>+4c  FS_FSINFO;  /* DosQFsInfo, DosSetFsInfo */
>>+50  FS_INIT;  /* -- No corresponding API */
>>+54  FS_IOCTL;  /* DosDevIoctl */
>>+58  FS_MKDIR;  /* DosMkdir */
>>+5c  FS_MOUNT;  /* -- No corresponding API */
>>+60  FS_MOVE;  /* DosMove */
>>+64  FS_NEWSIZE;  /* DosNewsize */
>>+68  FS_NMPIPE;  /* All named pipe related API's */
>>+6c  FS_OPENCREATE; /* DosOpen */
>>+70  FS_PATHINFO;  /* DosQPathInfo, DosSetPathInfo */
>>+74  FS_PROCESSNAME; /* -- No corresponding API */
>>+78  FS_READ;  /* DosRead, DosReadAsync */
>>+7c  FS_RMDIR;  /* DosRmdir */
>>+80  FS_SETSWAP;  /* -- No correcponding API */
>>+84  FS_WRITE;  /* DosWrite, DosWriteAsync */
>>+88  FS_OPENPAGEFILE; /* init time only */
>>+8c  FS_ALLOCATEPAGESPACE; /* size swap file */
>>+90  FS_CANCELLOCKREQUEST; /* DosCancelLockRequest */
>>+94  FS_FILELOCKS;  /* DosSetFileLocks */
>>+98  FS_VERIFYUNCNAME; /* Used to save function addresses */
>>+9c  FS_COMMIT;  /* DosBufReset, DosClose */
>>+a0  FS_DOPAGEIO;  /* perform paging */
```

```
>>+a4  FS_FSCTL;  /* DosFsCtl */
>>+a8  FS_FLUSHBUF;  /* DosBufReset */
>>+ac  FS_SHUTDOWN;  /* DosShutdown */
>>+b0  FS_SDCHGFILEPTR; /* Used to save function addresses */
>>+b4  FS_SDFSINFO;  /* at shutdown time.  These functions */
>>+b8  FS_SDREAD;  /* will only be called by shutdown */
>>+bc  FS_SDWRITE;  /* filters. */
>>
>> * Bit masks for FS_ATTRIBUTE (remember FS_ATTRIBUTE points to the
>>attribute
>> * word rather than containing it directly.)
>>
>> FS_ATTR_REMOTE  0x0001 /* 0 = local FSD, 1 = remote FSD     */
>> FS_ATTR_UNC  0x0002 /* 0 = normal, 1 = this is UNC FSD   */
>> FS_ATTR_LOCKINFO 0x0004 /* 0 = no notice, 1=notify filelocks */
>> FS_ATTR_LVL7  0x0008 /* 0 = no level 7 requests, 1 = yes  */
>> FS_ATTR_PIPESVR  0x0010 /* 0 = don't FSD on PIPE req,1 = yes */
>>
>> /* bit masks for FS_ATTRIBUTE (High Word) */
>> FS_ATTR_VERNO  0x7000 /* bits 28-30 version no */
>> FS_ATTR_EA  0x8000 /* bit 31 -> 1 = extended attribute */
>>
>> /* equates for commit type */
>> FS_COMMIT_ALL  2 /* all handles commit */
>> FS_COMMIT_ONE  1 /* one handle commit */
>>
>> /* equates for close type */
>> FS_CL_ORDINARY  0 /* ordinary close */
>> FS_CL_FORPROC  1 /* final close for process */
>> FS_CL_FORSYS  2 /* final close for system */
----------------------------------------------------------------------
```

Appendix A. Minimal Command Reference

This reference provides some minimal guidance for using the dump formatter, and the debug kernel, which share a common command set.

A.1 To Display Descriptors

The following commands can be used to display descriptor information. Use them when you want to find out what type of storage is described, what linear address contains the data at a logical address, the limit (or size) of a piece of memory, or what privilege level contains it.

- The operands for these commands follow:

<none> Display the whole table.

<selector> Display a single descriptor from the appropriate table.

<selector1> <selector2> Display descriptor information from ″selector1″ through ″selector2″.

<selector> L <number> Display up to ″number″ descriptors, starting with ″selector″. Invalid selectors are not normally displayed, but are counted.

- DI - Display descriptors from the IDT

In the IDT, the interrupt number is used, rather than a selector.

- DG - Display descriptors from the GDT
- DL - Display descriptors from the LDT
- Example 1: DL 7 2F

This will show you the first 6 descriptors in the LDT.

- Example 2: DG 7 2F

This will show you the first 6 descriptors in the LDT, after indicating that the LDT is the correct table.

- Example 3: DL 7 L6

This will show you the first 6 descriptors in the LDT.

- Example4: DGA 0 18

This will show you the first 4 descriptors in the GDT, whether they are valid (useable) or not.

A.2 To Display Page Table Entries

This explains how to display entries from the page directory and page tables. Note that the direct hardware approach using ″dd %%cr3″ does not work as expected for the lowest 4 megabytes. This is because the dump program uses the first entry in the page directory. The content that was there during execution may be found using the ″.N″ command, labelled ″savepage″.

- DP <address>

The first line of output pertains to the entry in the page directory.

The remaining lines pertain to the successive pages. The column heading ″frame″ will tell you what physical address has been assigned. A blank, or missing entry indicates that the page in question is not in real storage. The column heading pteframe will give you the same information, as the previous column, if the address is paged in. If it is paged out, it will have the virtual page id, which is generally where it exists in the file SWAPPER.DAT, if it has been paged out.

If you use a logical address (#sel:offset), limit checking is performed for you; if you use a linear address, the entries for the next available address range are displayed without indication that what you asked for does not exist. Read the output.

A.3 To Display Storage Itself

This explains how to display data storage in several formats.

- The operands for these commands are as follow:
- <address> displays hex 80 bytes beginning at ″address″, if possible.
- <address1> <address2> displays data from ″address1″ through ″address2″, if possible. If ″address1″ is a logical address, ″address2″ must be an offset only (near address).
- <address> L <number> display ″number″ items beginning at ″address″.
- DA - display in ASCII only, no hex. Display is to the null (hex ′00′) at the end of string.
- DB - display hex bytes, and ASCII on the right, as well.
- DW - display data as words. The bytes will be exchanged and formatted in pairs, so you see what would be used for word accesses to storage in a more ′natural′ manner.
- DD - display data as doublewords. The bytes will be reversed and displayed in groups of 4, so you see what would be used for doubleword accesses in a more ′natural′ manner.
- D - display further in the same format.
- U - display storage as instructions, creating the assembler mnemonics and operands from the raw hex data. The raw data is also shown.

If ″U″ is entered repetitively without operands, it will continue to unassemble forward from the end of the last output. Either use an operand, or use ″.R″ to refresh the registers. U updates EIP internally, which may confuse you or the dump formatter, depending on your perspective.

A.4 Miscellaneous Commands

There are several other useful commands, listed here in no particular order:

- Q - quit the dump formatter
- ? - evaluate an expression

any expression entered after the question mark will be evaluated, and the result will be displayed in hex, decimal, octal, binary, character and Boolean formats.

?′ - actually evaluates the following string. Useful for commenting the log file.

- ? - internal help

The question mark by itself will provide a brief list of possible internal commands, with which operands are acceptable.

- .? - external help

A period followed by a question mark will give a brief list of external commands, followed by their operands.

Note: The distinction between internal and external commands is that the internal commands function without any knowledge of OS/2 control blocks, or structures, providing direct access to hardware information.

The external commands need to know how the content of OS/2 structures and how to find them in order to function.

- .R - display the ring 3 registers for the current thread
- .S nn - change to slot nn.

This is how you look at another thread. Be sure to issue .R after using .S because the .R command sets the symbolic registers.

- .P* - display information about the current thread.

One of the columns is headed ″Pid″, this is the process ID.

One of the columns is headed ″ORD″, this is the thread number.

- .I (dump formatter only) will give you basic information about the current process.
- .M <address> - identify the owner of a storage address.

This frequently results in 20-40 lines of output. You must look through it until you find the group you want. Generally, the easy way is to match the program short name from .P* to a set of lines.

- .LMO <handle> - display the load module objects for the module that is identified by the ″handle″. The handle may be found by inspecting the correct group of .M output for the column headed ″hmte″, which will be in the last line of each output group.

A.5 Controlling Execution with the Debug Kernel

This explains how to display get control at the debug terminal at a particular place or situation.

- VL

List the vectors (interrupts) monitored

- VC n, or *

Clear one or move vectors

- VSF n or *

Get control when an interrupt in ring 3 will be fatal to a thread

- VTF n or *

Get control when a trap occurs in Ring 0

- BP <address>[,″commands″]

Install a breakpoint instruction at <address>, when execution arrives there, execute ″command″.

- BR x,<address>[,″commands″]

This sets a breakpint using the hardware debug registers

BR E,... triggers when Executed

BR R,... triggers on data read or write

BR W,... triggers on data write only

E, R, or W may be followed by 1, 2, or 4; size of area monitored. The area must be on the appropriate boundary if 2 or 4 is used.

- .REBOOT

Cause the machine under test to reboot, if possible.

A.6 Device Driver Mini-Reference

This explains the format of the DD header, the required segments of a devuce driver, and most of the Device Help (DevHlp) codes.

Physical Device Driver Header

Offset	Size	Content
0	4	16:16 address of next DD header
4	2	Device Attribute bit
		8000 0=block, 1=character;
		4000 1=Inter DeviceDriverCommunication ok
		2000 0=IBM driver, 0=OEM driver; 1000=sharing handled by DD
		0800 Block: removable media Character: must Open & Close
		0380 =1 OS/2 driver; =2 IOCtl2 & Shutdown; =3 Capabilities bits
		0008 if clock device 0004 if NUL device
		0002 if STDOUT 0001 if STDIN
6	2	offset to strategy routine
8	2	offset to IDC routine
A	8	device name, if character; number of units if block
12	8	Reserved
1a	4	Capabilities bit strip

Request Packet Header

Offset	Size	Content
0	1	Length of packet
1	1	Block device unit code
2	1	Command code
3	2	Packet Status
5	4	Reserved
9	4	Queue linkage
D	?	Command-specific data

Status Code: 8000 (bit)=error, 0200 (bit)=busy, 0100 (bit)=Done
low byte is error code:
00=write protect, 01=unknown unit, 02=device not ready,
03=unknown command, 04=CRC error, 05=Bad Drive request structure length,
06=seek error, 07=unknown media, 08=sector not found, 09=out of paper,
0A=write fault, 0B=read fault, 0C=general failure, 0D=Change disk,
10=uncertain media, 11=character I/O interrupted, 12=monitor not supported,
13=invalid parameter, 14=device already in use, 15=initialization failed

Command	Function	Command	Function
00	Init	10	Generic IOCTL
01	Media Check	11	Reset Media
02	Build BPB	12	Get logical drive map
03	reserved	13	Set logical drive map
04	Read (input)	14	Deinstall
05	Peek Nowait	15	Reserved
06	Input Status	16	Partitionable hard drives
07	Input Flush	17	Get hard drive unit map
08	Write (output)	18	Reserved
09	Write + Verify	19	Reserved
0A	Output Status	1A	Reserved
0B	Output Flush	1B	Reserved
0C	Reserved	1C	Shutdown
0D	Open Device	1D	Get Driver Capabilities
0E	Close Device	1E	Reserved
0F	Removable Media	1F	Initialization Complete

A.7 Device Help function Numbers

This is a simple table of DevHlp functions. The function is put into DL prior to calling DevHlp.

```
Code Service            Code Service            Code Service
00   SchedClockAddr     10   QueueFlush         20   Register
01   DevDone            11   QueueWrite         21   DeRegister
02   Yield              12   QueueRead          22   MonWrite
03   TCYield            13   Lock               23   MonFlush
04   ProcBlock          14   Unlock             24   GetDosVar
05   ProcRun            15   PhysToVirt         25   SendEvent
06   SemRequest         16   VirtToPhys         26   not used
07   SemClear           17   PhysToUVirt        27   VerifyAccess
08   SemHandle          18   AllocPhys          28   reserved
09   PushReqPacket      19   FreePhys           29   reserved
0A   PullReqPacket      1A   not used           2A   AttachDD
0B   PullParticular     1B   SetIRQ             2B   InternalError
0C   SortReqPacket      1C   UnSetIRQ           2C   reserved
0D   AllocReqPacket     1D   SetTimer           2D   AllocGDTSelector
0E   FreeReqPacket      1E   UnSetTimer         2E   PhysToGDTSelector
0F   QueueInit          1F   Monitor Create     2F   not used

Code Service            Code Service            Code Service
30   not used           50   RegisterPDD        60   PageListToGDTSelector
31   EOI                51   RegisterBeep       61   RegisterTmrDD
32   UnPhysToVirt       52   Beep               62   reserved
33   TickCount          53   FreeGDTSelector    63   AllocCtxHook
34   GetLIDEntry        54   PhysToGDTSel       64   FreeCtxHook
35   FreeLIDEntry       55   VMLock             65   ArmCtxHook
36   ABIOSCall          56   VMUnlock           66   VMSetMem
37   ABIOSCommonEntry   57   VMAlloc            67   OpenEventSem
38   GetDeviceBlock     58   VMFree             68   CloseEventSem
39   reserved           59   VMProcessToGlobal  69   PostEventSem
3A   RegisterStackUsage 5A   VMGlobalToProcess  6A   ResetEventSem
3B   reserved           5B   VirtToLin          6B   reserved
3C   VideoPause         5C   LinToGDTSelector   6C   DynamicAPI
3D   SaveMessage        5D   GetDescInfo        6D   reserved
3E   reserved           5E   LinToPageList      6E   reserved
3F   reserved           5F   PageListToLin      6F   reserved
```

A.8 Partial Content of the System Anchor Segment (SAS)

This is an extract of the SAS content. The full content is beyond the scope of this document. Word fields are offsets within the SAS, unless otherwise noted.

Offset	Length	Contains
00	4	EYECATCHER ‘SAS’
+0A	2	Offset of DD section
DD section + 0	2	Offset of first DD header
DD section + A	2	Selector for FSC

A.9 Partial Content of the File System Control Block (FSC)

This is an extract of the FSC content. The full content is beyond the scope of this document.

Generally, selector C8 is the selector for the first FSC.

This is the way the data will appear if the DD command is used.

Each field is a 16 bit FAR ADDRESS of the named function unless specifically stated otherwise.

OFFSET	+00	+04	+08	+0C
+00	actual size of FSC		Address of FSD Attributes	Address of FSD Name
+10	FSAttach	ChDir	ChFilePtr	Close
20	Copy	Delete	Exit	SetFileMode
30	SetFileInfo	FileIO	FindClose	FindFirst
40	FindFromName	FindNext	FindNotifyClose	FindNotifyFirst
50	FindNotifyNext	QFSInfo	(Init)	DecIOCyl
60	MkDir	(no api)	Move	NewSize
70	Named Pipe API′s	Open	SetPathInfo	(no api)
80	Read	RmDir	(no api)	Write
90	(no api)	(no api)	CancelLockRequest	SetFileLocks
A0	VerUNCName	Commit	PageIO	FSCtl

Glossary

Application Anchor Block (AAB). A PM **Application Anchor Block** is allocated in the Thread Local Memory Area (TLMA) when a PM application thread creates a message queue. The AAB contains a pointer to the MQ which allows PM to find the MQ in any context. This is particularly useful to the debugger since it also allows the MQ of any PM thread in the system since the TLMA is saved in a thread's TCB.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

BlockIDs. BlockIDs are conventional tokens used to represent the reason for a thread that blocks. This occurs as the result of the kernel entering TKSleep (either directly or via ProcBlock). The address of the BlockID is passed to TKSleep and stored in TCBSleepID. A thread wakes when the kernel calls TKWakeup (or ProcRun) with a corresponding BlockID. All, zero or the highest priority thread blocked on the BlockID will be woken depending on parameter flags. This mechanism is used by most functions and APIs that cause thread execution to be suspended, either for an event or serialization.

Examples are:

DosSleep

DosSemWait

DosWaitChild

DosRead

DevHlp_ProcBlock

BIOS Parameter Block. A BIOS Parameter Block is used for low level Disk I/O calls to the BIOS.

For further information see:

The .D BPB Command in the Kernel Debugger and Dump Formatter Command Reference.

The BPB Structure in the System Reference.

Breakpoint. A breakpoint is a location in a program where execution is suspended and control is given to a debugging tool.

The INTEL architecture supports two implementations of breakpoints for debugging purposes:

The software generated breakpoint using the INT 3 instruction;

The hardware generated breakpoint using the Debugging Registers.

The use of software breakpoints require code modification, whereas the use of debugging registers does not. However, the number of predefined software breakpoints is potentially unlimited whereas there are only 4 breakpoints specifiable using Debugging Registers.

A further distinction between the two types is that software breakpoints only intercept the execution of a particular instruction path, whereas Debugging Registers may be used, in addition, to intercept data fetches and stores from a particular location in virtual memory.

The Kernel Debugger supports both implementations of breakpoints through the use of the:

The BR command, which uses Debugging Registers.

The BP command, which uses INT 3 instructions.

The Kernel Debugger limits the predefinition of BP breakpoints to 10, however the programmer may code as many additional INT 3 instructions into their program as desired.

The Kernel Debugger refers to breakpoints explicitly set by the BP and BR commands as ***sticky*** (implying a certain permanence about them). The G command may have one or more temporary breakpoints established when one or more stop addresses are specified. These are referred to as ***go*** breakpoints. Once the Kernel Debugger breaks in ***go*** breakpoints are removed. The internal operation of the Kernel Debugger may also necessitate the use of the occasional temporary breakpoints when instruction tracing (see the T and P commands). These are set implicitly and discarded without the user being aware of their existence. Go and temporary breakpoints are created using the INT 3 instruction. Go and sticky BP breakpoints count towards the Kernel Debugger imposed limit of 10, but temporary breakpoints only ever exist singly so do not.

Block Management Package (BMP). A Block Management Package (BMP) is a data structure used to manage a pool of fixed length blocks using a bit string. Each bit in the bit string corresponds to an entry. A set bit indicates whether the entry is in use.

Typically this is used for:

Kernel Heap allocation
Memory object allocation

cbargs. cbargs is the argument count associated with the hardware defined call gate mechanism. The count is the number of words or double-words (as defined by the gate descriptor) that are inserted into a ring 0 stack when ring 2 or ring 3 code executes a call gate instruction.

CBIOS. The Compatibility BIOS (CBIOS) is a layer of code in the OS2LDR that presents a hardware independent interface to the BIOS for the OS2KRNL. The interface to the OS2KRNL is provided through a set of entry points called Dos Helper Functions.

Codepage Data Information Block (CDIB). The Codepage Data Information Block (CDIB) contains country-specific constant information relating to screen, keyboard and printer devices. The CDIB is built from information derived directly from CONFIG.SYS statements.

The CDIB may be located from the SAS.

Client Register Information (CRI). The Client Register Information (CRI) is a table of Register Information Packets (RIPs) that describe the offset and length of each register that is stored in a ring 0 stack frame on entry to the kernel. This level of indirection allows kernel routines to access entry registers regardless of the stack frame type, of which there are a number, for example:

System Entry Frames from API calls

Trap Frames from traps and exceptions

Interrupt Frames from the interrupt manager

VDM Stack Frames

Kernel Stack Frames

Each TCB points to a CRI and the associated stack frame from TCB_pcriFrameType(TCB + 0x38) and TCB_pFrameBase(TCB + 0x3c) respectively.

Command Subtree Identifier. The Command Subtree Identifier is used to represent a part of a process (or command) tree headed by a particular parent process. The ID used is the Pid of the process that heads the subtree.

Normally a process has a CSID equal to it′s own Pid. However, when processes become orphaned they acquire the subtree Id of their original parent and become adopted by their grand-parents by acquiring their grand-parents′s Pid as their new parent Pid.

Common ABIOS Data Area (CDA). The CDA is the Common ABIOS Data Area.

Refer to the Kernel Debug and Dump formatter guide, external command .C - display Common ABIOS data area.

Compatibility Region Mapping Algorithm. The Compatibility Region Mapping Algorithm (also referred to as the thunking algorithm) is used by thunking code to convert 16:16 addresses to 0:32 addresses and vice versa.

This is achieved by ensuring LDT selectors have their limits set to 64K so that they tile the compatibility region (0M to 448M). This gives an easy conversion algorithm from the selector:offset address to the 32-bit linear address. In C language syntax this is expressed as follows:

```
linear_address=((selector>>3)<<16)+offset

selector:offset=((linear_address&&0xffff0000)>>13)||7:(linear_address&&0x0000ffff)
```

Context (or thread context). Context refers that the *view* of the system a given thread has. Thus switching contexts refers to the process of preparing the system for another thread to run.

Context switching involves a number of actions, which include:

- Updating GDT descriptor entries 28, 30, 38 and 150b, which point to

  the current process′ **LDT**,

  the current threads ring 0 stack,

  the current thread′s floating point emulator work area,

  the current thread′s TIB. (By default the **FS** selector is loaded with 150b).

  The **LDT** selector is only updated when the process changes with a context switch, that is, for a process context switch.

- Switching pages tables for a process context switch.
- Updating the **TR** register if the process switch involves a task switch (normally only VDMs).
- Updating the current TSS ring 0 and ring 2 stack pointers.
- Updating system copies of the Global and Information Segments.
- Copying the Local Information Segment from the incoming PTDA and the Thread Local Memory Area from the incoming TCB to the segment mapped by LDT selector **dfff**.

Besides addressing the current ring 0 stack, selector 30 also addresses the current thread′s scheduling control blocks. In particular: the PTDA, TCB and TSD. This is done by aliasing selected address ranges from selector 30 to those of the true PTDA, TCB and TSD in the system arena global memory for the current context. The system defines a dummy module containing a hard-coded PTDA. The symbols of this module have the same name as those of the fields in the PTDA. The system arranges for this to map the PTDA addressed by selector 30. This trick allows the system to refer to PTDA fields for the current context without regard for which process is current, simply by using the field names as public symbols. The user may use the same symbols for referencing the PTDA but these are only valid for the current system context. To access PTDA fields in other contexts the following technique can be used:

```
Note that the current PTDA is located at PTDA_Start

##.p *
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*0025  0004 0002 0004 0001 blk 0300 7b7c8000 7bbc4080 7bbe8a90 1fc4 16 someprog

The current thread slot is 25

We wish to know the thread that has entered critical section in process of
thread slot 40.  The address of the critical section TCB is
saved in ptda_pTCBCritSec and the thread ordinal and slot number
are the first two words of the TCB.

##.p 40
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0040  0012 0002 0012 0001 blk 0500 7b7d6000 7b9e4020 7b9c8a70 1eb8 10 userprog

##dw %(DW(%7b9e4020+ptda_ptcbcritsec-ptda_start)) l2
%7b9c8de0 0002 0041

Thread 2 of 12 or thread slot 41 is in critical section
##.p 41
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 0041  0012 0002 0012 0002 blk 0800 7b7da000 7b9e4020 7b9c8de0 1ed4 10 userprog
```

Refer to the Kernel Debugger and Dump Formatter .P and .S commands for more information.

Current Directory Structure (CDS). A Current Directory Structure (CDS) is used to store file system information about the current directory per drive of each process.

Each CDS is managed in an RMP segment. The PTDA for each process contains an imbedded array of 26 CDS handles, one for each drive. The CDS RMP segment may be located from the SAS.

See also related structures:

MFT

SFT

DPB

FSC

VPB

DEM. DEM is the DOS Emulation component of OS/2.

Driver Parameter Block (DPB). A Driver Parameter Block (DPB) contains vital information about the state and format of a disk drive. The DPBs are chained together and located in a single segment whose selector may be obtained from the SAS.

See also related structures:

CDS

MFT

SFT

FSC

VPB

DosHlp. DosHlp services comprise a set of hardware dependent service routines established during system initialization for use by the OS2KRNL and user programs via the OEMHLP$ device driver. Many of the DosHlp services deal with device dependent BIOS behaviour and therefore provide a device independent interface to the BIOS.

File Allocation Table (FAT). The File Allocation Table file system is the default filing system supported by OS/2. Support for FAT is always present, regardless of any installed file systems.

File System Control Block (FSC). A File System Control Block (FSC) represents an installed file system (IFS). The FSC contains a table of entry points implemented by the file system driver (FSD). All FSCs are located in a single segment whose selector may be obtained from the the SAS.

See also related structures:

CDS

MFT

SFT

DPB

VPB

File System Driver (FSD). A File System Driver (FSD) is a special load module that implements an installed file system (IFS). FSDs are loaded during system initialization when the .IFS statement of CONFIG.SYS is encountered.

Examples of FSDs are:

HPFS.IFS

HPFS386.IFS

CDROM.IFS

Gate. A gate descriptor is one that defines to the hardware a means of entering code that executes at a more privileged level of authority. Four types of gate are defined:

Call Gate The subject of a CALL instruction. Typically used to implement operating system and device driver application programming interfaces (APIs). Device drivers may create *Call gates* dynamically using the DosDynamicAPI facility.

Task Gate The subject of a call or exception where a (hardware assisted) task switch is required.

Interrupt Gate The subject of a hardware or software generated interrupt. Typically an Interrupt gate will switch execution to an interrupt handler when a device presents an interrupt.

Trap Gate The subject of a trap exception. Used to handle programming errors.

Global Descriptor Table (GDT). The Global Descriptor Table (GDT) is a hardware architected control block. The GDT is common to all protect mode processes. It contains descriptors for memory segments common to all protect mode processes.

Global Information Segment (GISEG). The Global Information Segment (GISEG) is a single instance control block that records the current session status, date and time, trace status and version of the system.

The system maintains two copies of the Global Information Segment to fence against system damage.

The selector for the GISEG may be located from the SAS. See the Dump Formatter and Kernel Debugger .A command.

The GISEG is also mapped locally per-process by the LDT descriptor 0xdff4.

hal. The memory alias record handle (hal) is an index into the table of memory alias records (VMALs) whose address is located at **_parVMAliases.**

har. The memory arena record handle (har) is an index into the table of memory arena records (VMARs) whose address is located at **_parvmOne.**

hco. A hco is a handle for a memory context record. This is an index into the table of memory arena records (VMCOs) whose address is located at **_pcovmOne.**

hmte. The MTE control block handle (hmte) is the hob of the memory object that contains the MTE.

MTEs are allocated as pseudo-objects, so do not have Arena Records associated with them.

hob. The memory object record handle (hob) is an index into the table of memory objects records (VMOBs) whose address is located at **_pobvmOne.**

hptda. The PTDA control block handle hptda is the hob of the memory object that contains the PTDA.

PTDAs are allocated as pseudo-objects, so do not have Arena Records associated with them.

HWND. An **hwnd** is the handle to a WND structure. This is returned to an application when it uses **WinCreateWindow** and is used for subsequent PM API calls that affect the window.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

Interrupt Descriptor Table (IDT). The Interrupt descriptor table (IDT) is a hardware architected structure that comprises a table of gate descriptors, one for each interrupt vector. The low numbered entries are defined by the hardware architecture and dedicated to exception management.

The Kernel Debugger′s V command may be used to intercept system exception handlers.

Interrupt Vector. An interrupt vector is presented to the processor when an interrupt is generated either externally by the Programmed Interrupt Controller or internally within the processor chip itself. It is used by the processor as an index into the IDT to determine which interrupt routine should be dispatched.

The processor reserves vectors 0 - 31 to correspond to hardware architected exceptions 0 through 31. Vectors 32 - 255 are reserved for I/O interrupts, which are presented to the processor by the Programmed Interrupt Controller when the one of its IRQ lines is triggered. The correspondence between vectors and IRQs is defined during system initialization as follows:

IRQs 0 - 7 vectors 0x50 - 0x57

IRQs 8 - 15 vectors 0x70 - 0x77

Thus a keyboard interrupt, which is assigned to IRQ 1 under the IBM PC architecture will be handled by the interrupted handler whose interrupt gate is assigned to IDT descriptor 0x51.

See the Dump Formatter and Kernel Debugger DI command for information on displaying IDT entries.

Internal Processing Errors (IPEs). Internal Processing Errors are unrecoverable error conditions detected by the system while running in ring 0. The may arise from inconsistencies detected by the OS/2 Kernel or from traps occurring in any ring 0 code (Kernel, Installable File System Drivers and Device Drivers).

When the system detects an IPE it enters a routine called panic where an error message is formatted and displayed and the system is halted.

Job File Number (JFN). A Job File Number (JFN) is a handle for open file system objects, unique within the process that opened the file system object. The JFN is returned by DosOpen. It is used and an index into the JFN Table to locate the corresponding SFT handle.

Job File Number Table (JFT). A Job File Number Table (JFT) entry is assigned to each open file system object within a process. The JFT provides a cross-reference to the handle for the corresponding SFT. The JFT is locatable from the PTDA field **JFN_pTable** (**PTDA** +0x5b8 (H/R: +0x5b0)) for each process.

The JFT is initially allocated within the PTDA at label **JFN_Table** (**PTDA** +0x35e) with 20 entries. If this is expanded by use of the DosSetMaxFH then JFN_pTable is updated to point to the new table.

See also related structures:

- CDS
- DPB
- MFT
- FSC
- VPB

Kernel Semaphores. Kernel Semaphores are a form of semaphore, similar to the application 32-bit semaphore, used by kernel routines for longer term blocking. Kernel Semaphores provide addition function over the simple blocking mechanism, which includes:

- Priority inversion protection.
- Ownership auditability.

For additional information see the following:

- The .D KSEM command.
- The .PB command.
- The KSEM structures in the System Reference.

Loader Cache. The Loader Cache is used for saving discardable pages of instance data segments from DLLs loaded from mountable media. The caches is allocated from the kernel heap and has a system object owner ID of cache.

Local Descriptor Table (LDT). The Local Descriptor Table (LDT) is a hardware architected table of memory descriptors.

Under OS/2 one LDT is allocated per process.

Local Information Segment. The Local Information Segment is a per-process control block that records the current status of the process. It is imbedded in the PTDA and is also mapped by the LDT descriptor 0xdfff.

Master File Table (MFT). A Master File Table (MFT) entry is used to associate path names with open files (SFTs) and lock records (RLRs). The MFTs are managed in a PTREE structure, which is locatable from the SAS.

See also related structures:

- CDS
- DPB
- SFT
- FSC
- VPB

Message Queue Header (MQ). A PM **Message Queue Header (MQ)** is used as an anchor for message processing for a given PM Application's message thread. The MQ is created when a threads calls **WinCreateMsgQueue**.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

Module Table Entry (MTE) (non-swappable). The (non-swappable) Module Table Entry (MTE) for a loaded module is use to record information about loaded modules. Since the

MTE is allocated in non-swappable only information that must be resident at all times is recorded here. Related information that may be paged out is recorded in its sister control, the Swappable Module Table Entry (SMTE).

The MTE contains the following information:

- pointers to related control blocks such as, SMTE, resource and fix-up tables.
- attributes of the load module.
- Use count for .EXE modules.

Each MTE is identified by a unique handle referred to as the hmte.

Object Table Entry (OTE). An Object Table Entry (OTE) describes the address, size and attributes an object within a loaded 32-bit load module.

The corresponding control block for a 16-bit load module is the STE.

Page Frame Structure (PF). A Page Frame Structure (PF) is used by page frame management to track the status of a physical storage frame. The Page Frame Structures are allocated in contiguous storage, anchored from the address specified in global variable:

_pft

Each PF corresponds one to one with a frame of physical storage and provides links to Virtual Page Structures VPs.

Zero or more PTEs may be pinned to a physical frame, this is reflected in a reference count maintained in the associated PF.

UVIRT mappings have their corresponding PFs reserved unless aliased by non-UVIRT storage.

Paragraph. A paragraph is a unit of memory allocation of 16 bytes. Paragraph aligned allocations lie on a 16-byte boundary.

Patricia Tree (PTREE). A Patricia Tree (PTREE) is a form of tree structure designed to offer a fast look-up facility for generically specified keys. In OS/2 a modified form of the PTREE is use to manage MFTs for fast path-name look-up.

Page Table Entry (PTE). A Page Table Entry (PTE) is a hardware architected structure that is used to map virtual addresses to physical storage addresses.

Per-Task Data Area (PTDA). The Per-Task Data Area (PTDA) is the anchor point for all process (task) related control information. One PTDA exists per process and from it is located the LDT, TCB chain, Page tables and Arena Headers for a process.

All active PTDAs are addressable, whatever the current process, from a global address in the system arena. However, for the current process an alias address is created using selector 30 and in addition the many of the PTDA field names are declared as public symbols. This allows the fields names in the PTDA for the current process to be referred to directly under the Kernel Debugger and Dump Formatter.

PTDA_Start is the symbol assigned to the beginning of the current PTDA. Using the ? command against this and other PTDA field names allows relative offsets for PTDA fields to be calculated and used in other contexts as offsets from the global PTDA address.

Physical Arena Information block (PAI). The Physical Arena Information block (PAI) describes ranges of physical memory to memory management.

Pageable physical memory is described by the PAI pointed to by the SAS.

Process Information Block (PIB). The Process Information Block (PIB) is a supplemental process related control block made accessible to ring 3 programs. It contains process status information obtained from the process′ PTDA.

The PIB may be located from ptda_avatib(PTDA + 0x28) using the Dump Formatter or Kernel Debugger.

A program gains access to the PIB along with the TIB by calling the DosGetInfoBlocks API.

Process Identifier (Pid). The Process Identifier (Pid) is a unique system wide value used to identify a given process.

Note: It is not the same as the hptda which also uniquely identifies a process.

The Pid is used as a handle in process related APIs such as DosKillProcess and DosWaitChild.

Program Data Block. The Program Data Block is the name given to the DOS PSP by the DEM component of OS/2.

Program Segment Prefix (PSP). The Program Segment Prefix (PSP) is a DOS control block that forms the header of a loaded program. Under OS/2 the DEM component refers to this as the PDB or Program Data Block.

Process. A process is a collection of threads that share a common address space.

Each process is primarily represented by a PTDA structure and is assigned a unique identifier, the Pid.

Processes are organized in hierarchical tree structures known as process or Command Subtrees.

Pseudo-Objects. Pseudo-Objects are small system objects that comprise control blocks and other system areas, which for reasons of virtual memory conservation are not represented by a corresponding Arena Records. They are allocated out of the kernel resident heaps and comprise the following types of object:

- MTE
- VMAH
- PTDA
- Loader Cache

Queue Message (QMSG). A PM **Queue Message (QMSG)** is used by **WinPostMsg** to enqueue an asynchronously sent message to a thread′s message queue. QMSGs are chained from the MQ of the receiver in a circular array.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

PWND. A **pwnd** is a 32-bit pointer to a WND structure.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

Record Lock Record (RLR). A Record Lock Record (RLR) describes a locked region of a file system record. RLRs are chained from the related MFT and point to the associated SFT. They record the owner of the record lock.

See also related structures:

- CDS
- DPB
- SFT
- FSC
- VPB

Record Management Package (RMP). A Record Management Package (RMP) is a data structure designed for tabulating variable length records. Typically OS/2 uses RMPs to manage:

- Named Storage names
- Open File names

Directory names
System Semaphore names

Reliability, Availability and Serviceability (RAS). RAS is an acronym that refers to diagnostic and service support within OS/2. Frequently it is used as a synonym for the adjective diagnostic.

Register Information Packet (RIP). A Register Information Packet (RIP) is an entity used to describe the size and offset of a register in a system stack frame. RIPs are located in a CRI.

Scheduler. The Scheduler component of OS/2 is responsible for managing threads on queues according to priority and status.

Screen Group. A Screen Group is a logical full screen buffer and keyboard. A number of processes may be assigned to run in a given screen group. The workplace shell is one such screen group. Each screen group is assigned an ID. The screen group assigned to a process is recorded in its Local Information Segment. The currently active screen group is recorded in the Global Information Segment.

Screen Groups are represented by SGCB structures.

Under version 2 of OS/2 the screen group concept has been extended to that of a session.

Screen Group Control Block (SGCB). The Screen Group Control Block (SGCB) is used by the session manager component of the system to represent a Screen Group. It contains status information for the screen group and acts as a cross reference between the Pid currently associated with a given screen group and vice versa.

Segment Table Entry (STE). A Segment Table Entry (STE) describes the address, size and attributes of a segment (object) within a loaded 16-bit load module.

The corresponding control block for a 32-bit load module is the OTE.

Session. Sessions are groups of related processes initiated using DosStartSession API. Each session is assigned a logical screen buffer or presentation space. Sessions are identified by a unique ID that corresponds with their Screen Group Id (though the range of numbers is extended to included PM sessions, which all share then same screen group).

The following session ID/Screen Group ID ranges are defined:

SG	Usage
0	Hard Error Popups
1	Shell Screen Group
2	Real Mode Screen Group
3	VioPopUp Screen Group
4	First Full Screen Application Session
15	Last Full Screen Application Session
16	First Windowable VIO-Session
31	Last Windowable VIO-Session
32	First PM session
255	Last PM session

System Anchor Segment (SAS). The System Anchor Segment (SAS) is a central system control block use to anchor control blocks for major system components such as:

File systems

Device Drivers

Scheduler

- Memory management

The SAS is built at the beginning of the segment addressable from selector 70 and 78.

Send Message Structure (SMS). A PM **Send Message Structure (SMS)** is used by **WinSendMessage** to enqueue a synchronously sent message. SMSs are chained from the MQ of both the sender and receiver.

See 14.1.3, “Exploring 32-bit Presentation Manager Under WARP” on page 273 for more information.

Swappable Module Table Entry (SMTE). The Swappable Module Table Entry (SMTE) contains characteristics of a loaded module that may be page out of memory. The SMTE is the sister control block to the MTE, which records those characteristics that must be resident at all times.

The SMTE principally contains:

- A pointer to OTE or STE.
- A pointer to the fully qualified module name.
- The entry point and initial stack pointers.

System File Table (SFT). A System File Table (SFT) entry is used to describe the attributes of each instance of an open file system object. SFTs are stored in a segment directly locatable from the SAS. SFTs are indirectly locatable from the JFN Table imbedded in the PTDA of each process that opens a file system object.

See also related structures:

- CDS
- DPB
- MFT
- FSC
- VPB

System Queue Message (SQMSG). A PM **System Queue Message (SQMSG)** is used by the **PMDD.SYS** device driver to enqueue messages, which represent system input activity, to the system input queue.

See 14.1.3, “Exploring 32-bit Presentation Manager Under WARP” on page 273 for more information.

System Trace Data Area (STDA). The System Trace Data Area (STDA) is a circular buffer used to record trace events. The STDA may be located from the SAS.

Symbol. A symbol is the name given to a program code or data location that has been made public by the programmer. Such symbolic definitions appear in the map file output from the linkage editor. They may be referenced in the Dump Formatter and Kernel Debugger using the L command when the map file is converted to a symbol file using the MAPSYM utility.

Symbol Absolute. An Absolute symbol is a symbolized constant value that has been made public by the programmer. Such symbolic definitions appear in the map file output from the linkage editor and may be referenced in the Dump Formatter and Kernel Debugger using the LA command when the map file is converted to a symbol file using the MAPSYM utility.

Symbol Group. A symbol group is the set of symbols that are defined within a program segment. Frequently a program segment is given its own selector at load time.

Symbol Map. A symbol map is created from symbolic name information generated by a program compiler and converted for used by the Dump Formatter and Kernel Debugger by the linkage editor and MAPSYM utilities. This allows program code and data locations to be referred to by name as well as by address.

System File Number (SFN). A System File Number (SFN) is the system-wide unique handle by which an open file system object is known. It is the offset into the SFT segment that locates the corresponding SFT entry.

Task. A task is a hardware architected thread of execution. The INTEL architecture allows for multiple independent tasks to coexist and provides the task gate mechanism as a means of switching between tasks. Tasks are represented to the hardware by the TSS.

The characteristics of a task are very similar to that of the OS/2 process. Protect-mode processes however, tend to run under a single task in OS/2 and implement switching through the more efficient software managed context switching mechanism.

Only VDMs and error recovery processes run as independent tasks.

See the *INTEL486 Programmer's Reference* for more information.

Task State Segment (TSS). The Task State Segment (TSS) is a hardware architected control block that is used for two purposes:

1. To implement the privileged level transition mechanism initiated with a Call Gate instruction.
2. To provide a register save area for hardware task switching initiated with a call to a Task Gate.

In general OS/2 does not use the hardware task switching mechanism, so TSSs are few. It does however use the TSS for implementing Application Programming Interfaces (APIs) in the system.

A TSS may be formatted using the Kernel Debugger and Dump Formatter DT command.

Translation Lookaside Buffer (TLB). The Translation Lookaside Buffer (TLB) is a hardware implemented buffer used for caching linear to physical address mappings.

The Intel486(TM) processor provides test registers for manipulating the TLB.

Thrashing. Thrashing refers to the state of a system where most of the CPU time is spent paging in and out memory from the swap file. This happens when real storage is heavily over committed and storage references encompass a wide range of virtual pages over a short processing time.

Such a condition can indicate a poorly tuned application where paging is caused by the process of accessing data the application needs. A typical scenario is where work data is chained in a single, very extended, queue and no mechanism exits to access the required data without scanning the entire chain. Use of hashing techniques greatly reduce this problem.

Thread. A thread is a independently scheduleable entity that competes for processor resource with other threads.

Each thread is represented by a TCB and TSD structure.

Threads are organized within processes and assigned a unique identifier within the owning process known as the Tid.

All threads within the system are assigned a system wide unique identifier known as the Thread Slot Number.

Thread Control Block (TCB). The Tread Control Block (TCB) contains per-thread control and status information that must be resident at all times. The swappable counterpart to the TCB is the TSD

Thread Identifier (Tid). The Thread Identifier (Tid) is a value, unique within the owning processes, used to identify the thread. It is not the same as the Thread Slot Number, which uniquely identifies a thread, system-wide.

The Tid is used in thread related APIs such as DosKillThread and DosSetPriority.

Thread Information Block (TIB). The Thread Information Block (TIB) is a supplemental thread related control block made accessible to ring 3 programs. It contains thread information obtained from the thread's TCB and acts as an anchor for exception-handlers registered for the thread.

The TIB may be located from TCBptib(TCB + 0x10) using the Dump Formatter or Kernel Debugger.

A program gains access to the TIB along with the PIB by calling the DosGetInfoBlocks API.

Thread Local Memory Area (TLMA). The **Thread Local Memory Area (TLMA)** a an area of private arena memory that is instanciated at a thread level. This is achieved by copying the contents of the TLMA to dfff:0024 when a thread switch occurs. The TLMA contents are saved in the TCB at **TCBTLMA**.

Storage is allocated from the TLMA by using the **DosAllocThreadLocalMemory** API.

Thread Slot Number. The Thread Slot Number is a system wide unique identifier assigned to each thread in the system.

Threads are located from the thread slot table whose linear address is at global symbol:

_papTCBSlots

Each slot is a double-word linear address of the corresponding thread's TCB. The first slot (slot=0) is reserved.

Under the Kernel Debugger and Dump Formatter the following symbols may be used to represent particular threads in many of the commands that accept a slot number as a parameter:

***** The current or last dispatched thread as recorded in word global variable **_TaskNumber**

The default thread slot used by the Dump Formatter and Kernel Debugger.

Thread Swappable Data (TSD). The Thread Swappable Data (TSD) control block contains per-thread status and control information that resides in swappable memory and therefore is not required for reference out of context of the related thread. The resident memory counterpart to the TSD is the TCB (Thread Control Block).

The vast majority of the TSD is used as the ring 0 stack when a thread makes a privilege level transition to ring 0 via a call gate descriptor. The base of the ring 0 stack will therefore include the ring 3 call gate stack frame on entry to ring 0 (which is usually kernel or device driver code).

In the debug kernel a dummy page prefixes the used part of the TSD in order to catch ring 0 stack faults.

Other information contained in the TSD includes GTD instance data for the corresponding thread's context. This comprises descriptors for:

28: The LDT descriptor.

30: Base selector for ring 0 process instance data, which includes the ring 0 stack, TCBs and PTDA.

38: Floating point emulator instance data

40: FS mapping to the TIB

When an an inter-process thread context switches, descriptors 30 - 40 are loaded into the GDT from the TSD. When an intra-process thread context switches, descriptors 28 - 40 are loaded into the GDT from the TSD.

Thunking. Thunking is the process of calling 16-bit code from 32-bit code and vice versa. Thunking consists of applying the CRMA to convert from one form of address to the other and making any stack parameter adjustments either by padding 16-bit operands to 32-bit with leading zeros (16- to 31-bit conversion) or truncating the padded 32-bit value to 16 bits (32- to 16-bit conversion).

Tracepoint. A **tracepoint** is designated location in system or application code where the System Trace Facility will gather data for logging by the STDA.

Tracepoints may be implemented statically by use of the **DosSysTrace** API or dynamically through use of the Dynamic Trace Customiser.

System defined tracepoints are documented in the System Tracepoints Reference.

UVIRT. The UVIRT attribute signifies virtual storage mapping to a pages of physical storage.

The full set of memory management structures associated with virtual storage allocation may not exist for UVIRT storage.

The UVIRT attribute may be associated with a number of structures, for example:

- PTE
- LDT and GDT descriptors
- VMAL

In general UVIRT allocations are ′convenience′ mappings memory to selectors. Typically they are created by device drivers using the DevHlp_PhysToUvirt facility.

Virtual DOS Machine (VDM). A Virtual DOS Machine (VDM) is a type of process that runs in an emulated DOS environment using the DOS Emulation (DEM) component of OS/2.

Virtual Page Structure (VP). A Virtual Page Structure (VP) is used by memory management to track the status of a virtual storage frame, whether backed by physical storage, cached by the loader or paged out to the swapper. The Virtual Page Structures are allocated in contiguous storage, anchored from the address specified in global variable:

_pgpVPBase

Virtual Memory Arena Header Record (VMAH). One Virtual Memory Arena Header Record (VMAH) is allocated per arena to record information about the address range of an arena. The VMAH points to its sentinel arena record (VMAR).

Each VMAH chained in a double linked list.

The system arena VMAH is located at global symbol:

_ahvmSys

The shared arena VMAH is located at global symbol: _ahvmShr

For each private arena the VMAH is imbedded in the PTDA at label &ptdaah..

Under OS/2 2.1 the system and shared arena VMAHs are assigned to objects 4 and 5 respectively.

Virtual Memory Alias Record (VMAL). The Virtual Memory Alias Record (VMAL) is used to represent aliased regions of virtual memory. These are either:

- regions of physical storage that may be addressed by more than one virtual or
- linear address that are not associated with a memory object, such as VDM UVIRT allocations.

When two memory objects are aliases of each other then they need not have coincident sizes or origins within the aliased arena record. Aliases are designed to provide alternative attributes for accessing the same piece of data within or across processes. Compare this with shared instance data within the shared arena, where multiple object records share a common arena record. In this case each object is associated with a unique process and is not considered an alias.

Each VMAL is identified by a unique handle referred to as the hal.

Virtual Memory Kernel Heap (VMKH). The Virtual Memory Kernel Heap (VMKH) structures are used to describe system heap memory. Many objects allocated out of the kernel heap are assigned a System Object identifier.

Virtual Memory Arena Record (VMAR). The Virtual Memory Arena Record (VMAR) is used to represent a contiguous region of virtual memory allocated in page quantities. Such storage may or may not be committed or resident.

Arena records are chained in a doubly linked lists, one for each arena type. That is, the chain chain exists separately for each private arena, the shared arena and system arena.

Special arena records, known as Sentinels head each chain. They describe the entire arena which they head.

All virtual memory is described by by at least one arena record.

Each VMAR is identified by a unique handle referred to as the har.

Arena also records point to the following related memory structures:

VMOB

VMAL

VMCO

Virtual Memory Context Record (VMCO). A Virtual Memory Context Record (VMCO) is used to record the association of shared arena, shared data objects with processes that are using.

Each VMCO is identified by a unique handle referred to as the hco.

Virtual Memory Object Record (VMOB). The Virtual Memory Object Record (VMOB) are used to represent memory objects, that is the instance data associated with a particular virtual address. VMOBs contain pointers to the the owning and requesting objects as well as the corresponding arena record (VMAH).

Each VMOB is identified by a unique handle referred to as the hob.

Volume Parameter Block (VPB). A Volume Parameter Block (VPB) is used to store volume information associated with a file system object. All VPBs are contained within a single segment locatable from the SAS. Most file system structures contain a VPB handle for an associated volume. The handle is used as an offset into the VPB segment.

See also related structures:

CDS

MFT

SFT

DBP

FSC

Window Structure (WND). A PM **Window Structure (WND)** is used by PM to represent a window. When an application uses **WinCreateWindow** the WND is created and the HWND is returned to the user. The WND contains information about a window′s hierarchy, and associated MQ.

See 14.1.3, "Exploring 32-bit Presentation Manager Under WARP" on page 273 for more information.

Zombie. The term **Zombie** is used to describe a.Z terminal condition of a thread or process. There is a strict operating system definition and two colloquial uses:

-
- The strict system definition refers to a process that has terminated but whose PTDA has been retained on the zombie queue (**_pPTDAFirstZombie**) because the process status byte (LISEG+0xa) indicates that its parent wishes to collect termination information through **DosWaitChild**. The dead child is retained on the zombie queue until either the parent dies or issues **DosWaitChild**.
- Zombie is also commonly used to refer to a terminating thread or process that has blocked after the application has returned to the operating system.

Usually this implies a problem freeing memory because one or more pages have been long-term locked by a device driver.

- The third use of zombie refers to any process that is anonymous. Internal thread, VDMs, and terminating threads can be anonymous.

List of Abbreviations

AAB	Application Anchor Block
BMP	Block Management Package
CDA	Common ABIOS Data Area
CDIB	Codepage Data Information Block
CDS	Current Directory Structure
CRI	Client Register Information
DEM	DOS Emulation
DPB	Driver Parameter Block
FAT	File Allocation Table
FSC	File System Control Block
FSD	File System Driver
GDT	Global Descriptor Table
GISEG	Global Information Segment
IBM	International Business Machines Corporation
IDT	Interrupt Descriptor Table
IPE	Internal Processing Errors
ITSO	International Technical Support Organization
JFN	Job File Number
JFT	Job File Number Table
LDT	Local Descriptor Table
MQ	Message Queue Header
MFT	Master File Table
MTE	Module Table Entry
OTE	Object Table Entry
PAI	Physical Arena Information block
PF	Page Frame Structure
PIB	Process Information Block
Pid	Process Identifier
PSP	Program Segment Prefix
PTDA	Per-Task Data Area
PTE	Page Table Entry
PTREE	Patricia Tree
QMSG	Queue Message
RAS	Reliability, Availability and Serviceability
RIP	Register Information Packet

RLR	Record Lock Record
RMP	Record Management Package
SAS	System Anchor Segment
SDTA	System Trace Data Area
SFN	System File Number
SFT	System File Table
SGCB	Screen Group Control Block
SMS	Send Message Structure
SMTE	Swappable Module Table Entry
SQMSG	System Queue Message
STE	Segment Table Entry
TCB	Thread Control Block
TIB	Thread Information Block
Tid	Thread Identifier
TLMA	Thread Local Memory Area
TLB	Translation Lookaside Buffer
TLMA	Thread Local Memory Area
TSD	Thread Swappable Data
TSS	Task State Segment
VDM	Virtual DOS Machine
VMAH	Virtual Memory Arena Header Record
VMAL	Virtual Memory Alias Record
VMAR	Virtual Memory Arena Record
VMCO	Virtual Memory Context Record
VMKH	Virtual Memory Kernel Heap
VMOB	Virtual Memory Object Record
VP	Virtual Page Structure
VPB	Volume Parameter Block
WND	Window Sturcture

Index

Special Characters

Numerics

A

B

C

D

E

F

G

M

N

O

P

T

U

V

W

Z

ITSO Technical Bulletin Evaluation RED000

International Technical Support Organization
The OS/2 Debugging Handbook - Volume I
Basic Skills and Diagnostic Techniques
February 1996

Publication No. SG24-4640-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction ____

Organization of the book	____	Grammar/punctuation/spelling	____
Accuracy of the information	____	Ease of reading and understanding	____
Relevance of the information	____	Ease of finding information	____
Completeness of the information	____	Level of technical detail	____
Value of illustrations	____	Print quality	____

Please answer the following questions:

a) If you are an employee of IBM or its subsidiaries:

Do you provide billable services for 20% or more of your time? Yes____ No____

Are you in a Services Organization? Yes____ No____

b) Are you working in the USA? Yes____ No____

c) Was the Bulletin published in time for your needs? Yes____ No____

d) Did this Bulletin meet your needs? Yes____ No____

If no, please explain:

____

____

What other topics would you like to see in this Bulletin?

____

____

What other Technical Bulletins would you like to see published?

____

Comments/Suggestions: **(THANK YOU FOR YOUR FEEDBACK!)**

Name ______________________________ Address ______________________________

Company or Organization ______________________________

Phone No. ______________________________

ITSO Technical Bulletin Evaluation **RED000**

SG24-4640-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization
Department 626C, Building 014-1
Internal Zip 5220
1000 Northwest 51st Street
Boca Raton, Florida
USA 33431-1328

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Printed in U.S.A.

SG24-4640-00