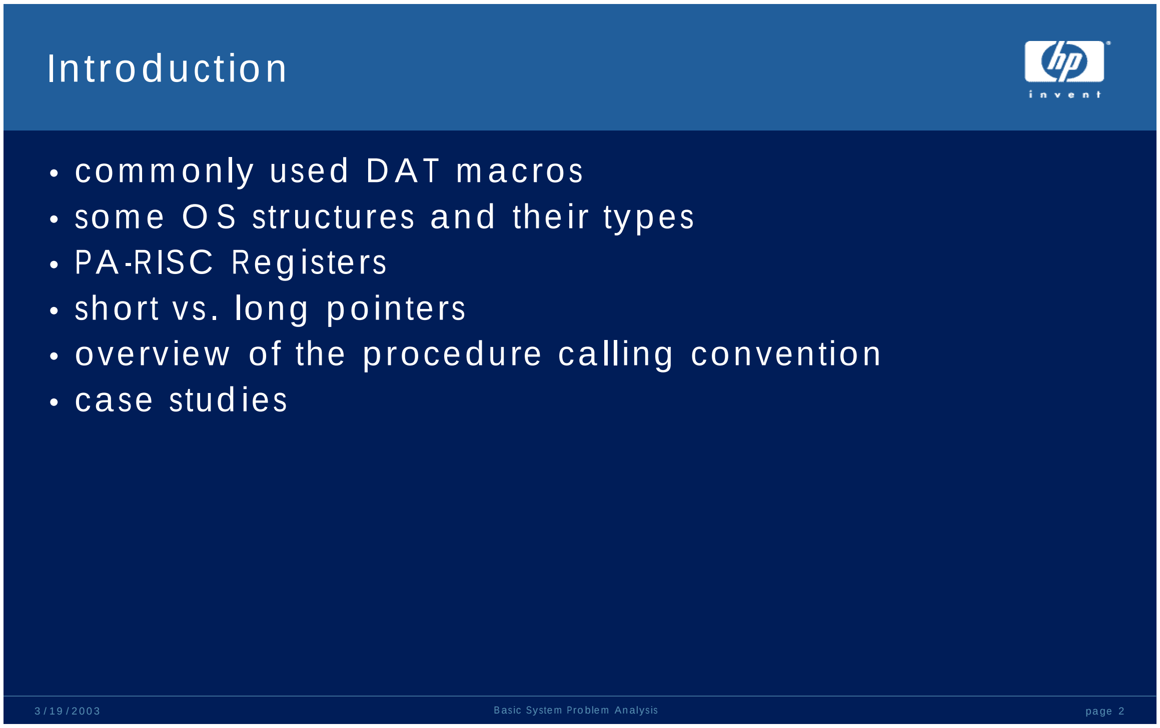
Basic System Problem Analysis

Bill Cadier
Hewlett-Packard Co.
TCSD MPE/iX Lab
25 April 2003

Table of Contents



Introduction

- commonly used DAT macros
- some OS structures and their types
- PA-RISC Registers
- short vs. long pointers
- overview of the procedure calling convention
- case studies

3/19/2003 Basic System Problem Analysis page 2

Notes:

Introduction

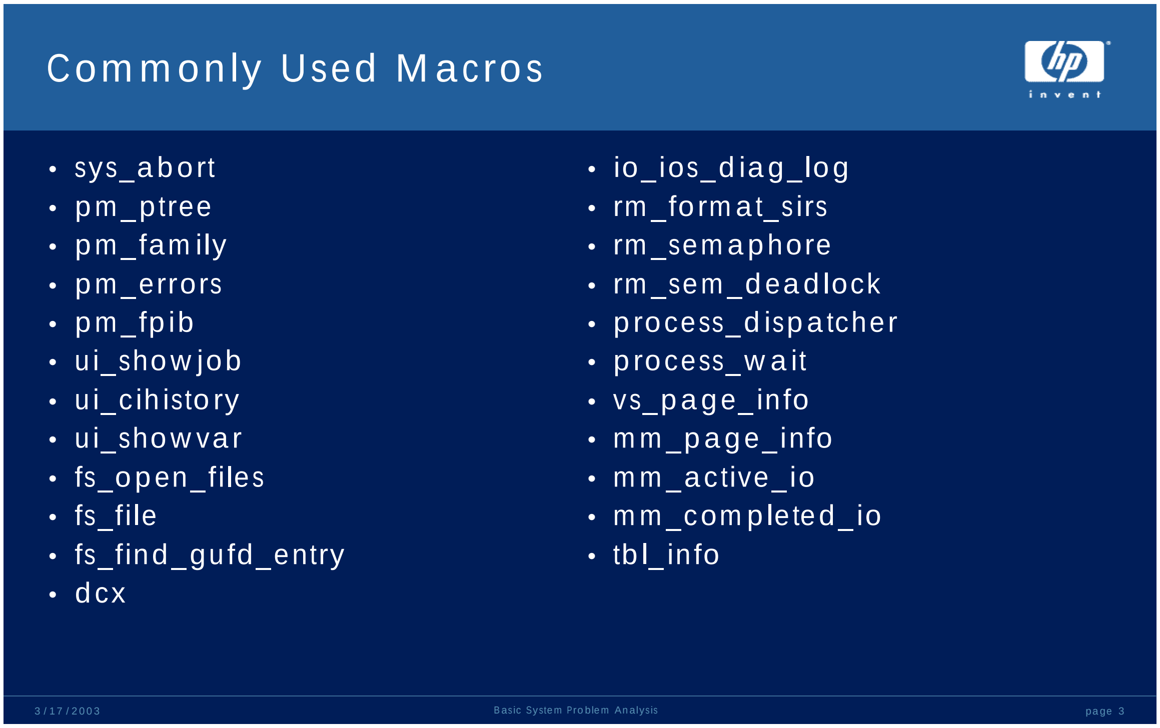
This paper is being presented at the West Coast HP3000 Solution Symposium in San Jose, 25 April 2003

The purpose of this paper is to try to provide basic information how to diagnose system aborts and hangs.

As the HP3000 winds down it will be advantageous for owners of this system to be able to perform as much trouble shooting as possible. The amount of trouble shooting will be limited because source code for the OS is not available outside HP.

It is assumed that readers have good familiarity with the tools DEBUG, DAT and SAT. The documentation for these tools may be found online at:

http://docs.hp.com/mpeix/onlinedocs/32650-90901/32650-90901.html



Commonly Used Macros

- sys_abort
- pm_ptree
- pm_family
- pm_errors
- pm_fpib
- ui_showjob
- ui_cihistory
- ui_showvar
- fs_open_files
- fs_file
- fs_find_gufd_entry
- dcx
- io_ios_diag_log
- rm_format_sirs
- rm_semaphore
- rm_sem_deadlock
- process_dispatcher
- process_wait
- vs_page_info
- mm_page_info
- mm_active_io
- mm_completed_io
- tbl_info

3/17/2003 Basic System Problem Analysis page 3

Notes:

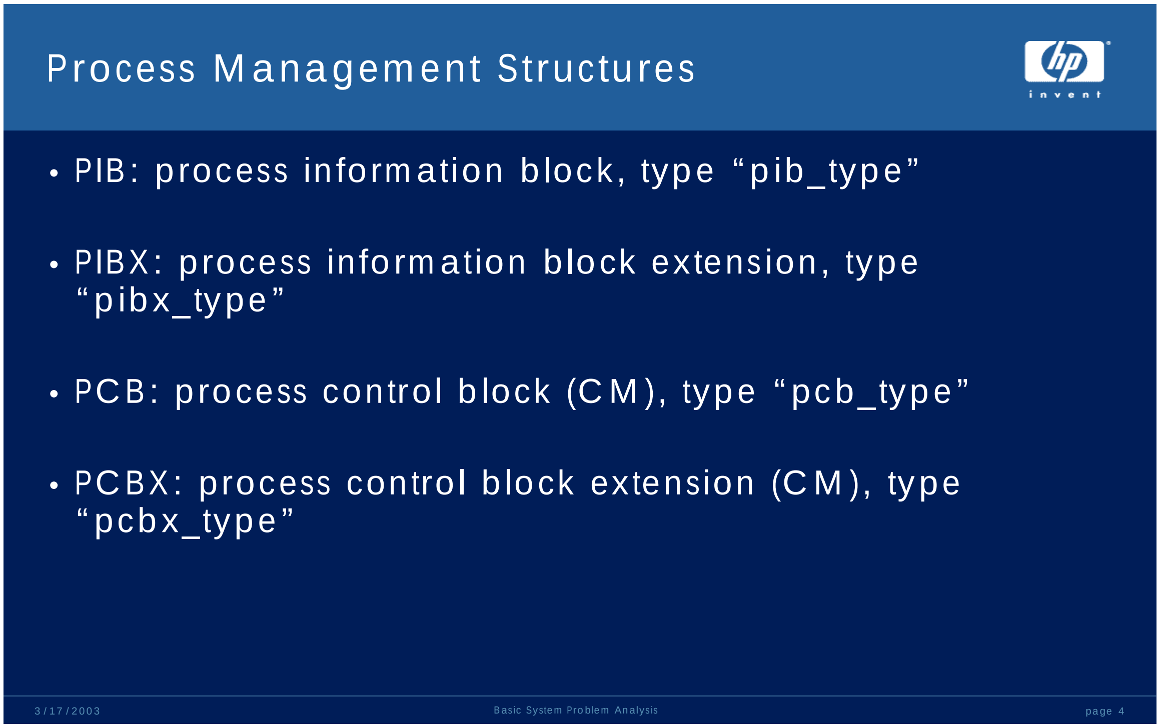
Commonly Used Macros

This is by no means a comprehensive list of macros available in the OS macro set but these are some of the more commonly used macros.

The MACLIST (MACL) command can be used to list all current macros once they have been restored. Many of the macros listed will be second level macros, those called by other macros and so would be of limited value. Use the HELP command to see the source for a given macro, i.e. HELP PM_FPIB.

Most macros are prefaced with a designator to indicate what area of the OS they are meant to be used for. Here’s a list of some of the designators:

```
pm     = process management
fs     = file system
mm     = memory management
vsm    = virtual space management
rm     = resource management (sirs and semaphores)
xm     = transaction management
ui     = user interface (CI commands)
io     = i/o subsystem
config = hardware configuration
```



Process Management Structures

- PIB: process information block, type “pib_type”
- PIBX: process information block extension, type “pibx_type”
- PCB: process control block (CM), type “pcb_type”
- PCBX: process control block extension (CM), type “pcbx_type”

3/17/2003 Basic System Problem Analysis page 4

Notes:

Process Management Structures

These are the fundamental process management structures and their types. DEBUG, DAT and SAT provide functions that return pointers to these structures. These functions are:

```
PIB   - returns a pointer to the PIB for a given pin
        Example: fv pib(5) ‘pib_type’

PIBX  - returns a pointer to the PIBX for a given pin
        Example: fv pibx(pin) ‘pibx_type’

PCB   - returns a pointer to the PCB for a given pin
        Example: fv pcb(200) ‘pcb_type’

PCBX  - returns a pointer to the PCBX for a given pin
        Example: fv pcbx(10) ‘pcbx_type’
```

The PIB contains information about a given process. The type for the PIB is divided into functional areas such as:

DISPATCH_INFO which contains linkages to the dispatcher run queues.

IO_AREA which contains information about outstanding non-memory management I/O requests for the process.

PIB_ERROR_STACK is the area that holds that status of errors or warnings. The values in this stack are of type “HPE_STATUS” and are pushed onto this stack by the procedure HPERRPUSH. The PM_ERRORS macro will dump this stack but often times it is useful to dump it raw, e.g. DV PIB(PIN)+350,20 so you can see all the errors, even those that are not current. The PM_ERRORS macro will only dump the active part of the error stack.

Decoding HPE_STATUS errors is accomplished using the ERRMSG function in DAT and DEBUG, for example given an HPE_STATUS of fffd008f decoding would be:

```
$1d5 ($21d) nmdat > wl errmsg(S16(fffd), 8f)
Intrinsic layer; an access violation occurred.
```

The “S16” function is used so that “fffd” is treated as a signed quantity rather than as the low 16 bits of a 32 bit quantity.

Process Management Structures: continued

Two other fields in the PIB worth noting are the PIB_TRAP_PC and PIB_TRAP_ISM. These two fields are used for certain types of process traps. The PC (program counter) of the trap and the interrupt stack marker (ISM) active at the time of the trap are loaded into these fields. If the system should fail as the result of a process trap it may be possible to use the command "INITNM" supplying the ISM pointer in PIB_TRAP_ISM to restore the stack as it was at the time of the trap. Unfortunately it is often the case that the old stack location has been overwritten by activity that transpired from the time of the trap to the time of the abort. It is always worth a shot to see if something meaningful can be retrieved. At the very least PIB_TRAP_PC can tell you what piece of code caused it, e.g. DCS [the value of PIB_TRAP_PC]

Useful process management macros are PM_PTREE which is a more full-featured version of the built-in DPTREE. Unlike DPTREE the PM_PTREE macro will display the job or session number. This can be used as input to some of the UI macros

PM_FAMILY provides similar output as PM_PTREE but for the whole process family. Note that you get a more complete list of the family tree using the JSMAIN pin. This will give you the JSMAIN, the CI under it and any descendents under that. The UI_SHOWJOB macro lists the JSMAIN pin for each job or session.

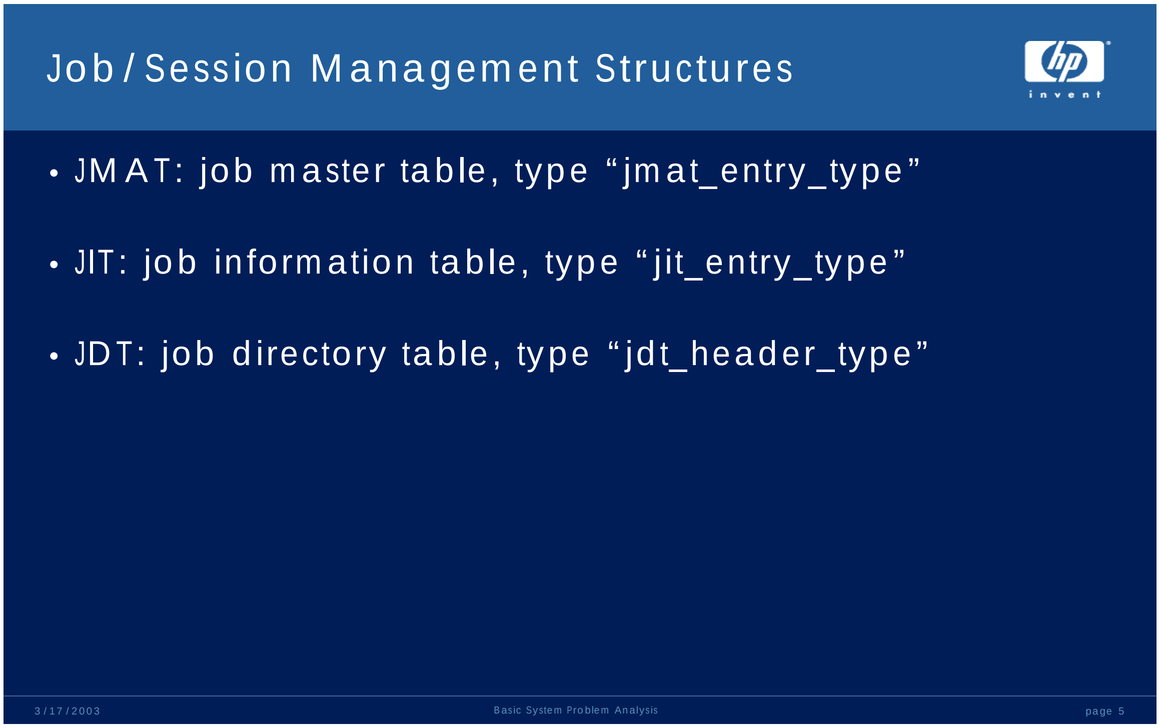
The PM_FPIB macro is an almost complete formatting of the PIB structure and can be useful in describing the overall state of the process. The input to this macro is, oddly enough a string so the macro would be called like this,

```
$1d6 ($21d) nmdat > pm_fpib('pin')
```

Or

```
$1d7 ($21d) nmdat > pm_fpib('21d')
```

This page intentionally left blank



Job / Session Management Structures

- JMAT: job master table, type “jmat_entry_type”
- JIT: job information table, type “jit_entry_type”
- JDT: job directory table, type “jdt_header_type”

3/17/2003 Basic System Problem Analysis page 5

Notes:

Job/Session Management Structures

The JMAT or job master table is what is displayed with the SHOWJOB CI command and there is an equivalent OS macro UI_SHOWJOB. Like its CI counterpart the macro displays all jobs and sessions or will display a specific job or session when a string with the “#Jnnn” or “#Snnn” value is supplied.

The JIT and JDT are compatibility mode data segments (DST) but all CM DST’s are objects and have NM virtual addresses. The DSTVA function translates a CM data segment number to its NM virtual address equivalent.

The JIT and JDT DST’s are kept in the CM stack in the “PXGLOBAL” area which is more easily remembered as being the first 12 (decimal) 16 bit words. So the quickest way to find the JIT and JDT are to dump the CM stack of the process you want them for.

```
$1de ($21d) nmdat > cm

%737 (%1035) cmdat > dd sdst.0,#12
DST %40346.0
%0      % 000450 000600 137677 005700 003461 000000 020400 000000
%10     % 040001 040000 040332 040330
```

Job/Session Management Structures: continued

The same thing can be accomplished using the native mode types:

```
$1e1 ($21d) nmdat > fv pcbx(pin) 'pcbx_type.pxglob,true'

CRUNCHED RECORD
      DL_MINUS_A      : 128
      DB_MINUS_A      : 180
      USER_ATT        : bfbf
      JMAT_INDEX      : bc0
      JPCNTINDEX      : 731
      JCUTINDEX       : 0
      STUNBIT         : FALSE
      RESTART         : FALSE
      JOBTYPE         : 2
      DUPLICATIVE     : FALSE
      INTERACTIVE     : FALSE
      ALLOWMASK       : FALSE
      JSMSTATE        : TRUE
      JSMCHANGE       : FALSE
      FILLER1         : 0
      STACKDUMP_FLAGS :
            STACKDUMP_INT : 0
      FILLER2         : 0
      NATIVE_LANG     : 0
      JOB_INPUT_LDN   : 4001
      JOB_OUTPUT_LDN  : 4000
      JDTDST          : 40da
      JITDST          : 40d8
END
```

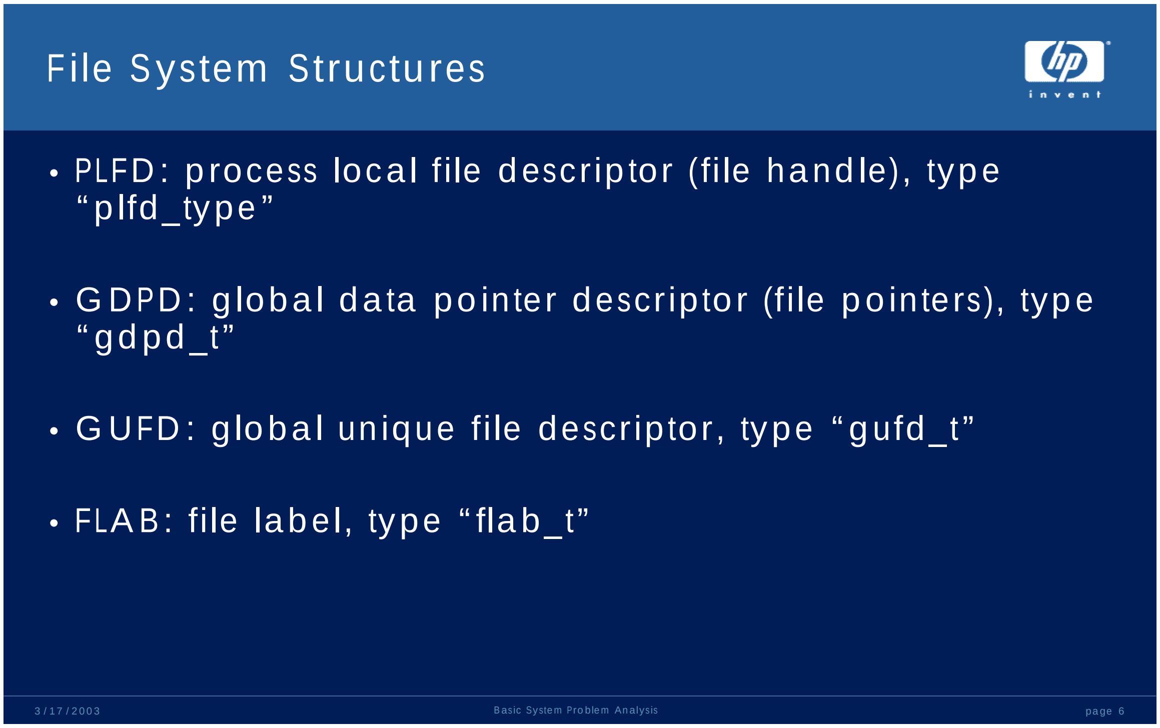
You want to do an FT on PCBX_TYPE to see where the ".PXGLOB" came from. Further you will see that the field PXGLOB is of type PXGLOB_TYPE. A format type on that shows that the less useful record variant appears first, a crunched array of 12 BIT16's. That will be the variant used unless another is explicitly specified. That's what you need to do in this case hence the ",TRUE" added to the format virtual command.

Now that we know the JIT and JDT DST numbers we can use the DSTVA function to translate that to a virtual address and finally format the type:

```
$1e3 ($21d) nmdat > fv dstva(40da.0) 'jdt_header_type'
```

See HELP DSTVA for additional details.

This page intentionally left blank



File System Structures

- PLFD: process local file descriptor (file handle), type “plfd_type”
- GDPD: global data pointer descriptor (file pointers), type “gdpd_t”
- GUFD: global unique file descriptor, type “gufd_t”
- FLAB: file label, type “flab_t”

3/17/2003 Basic System Problem Analysis page 6

Notes:

File System Structures

These structures are but the tip of the iceberg when it comes to the file system!

The PLFD is a file handle, whenever a process has a file or socket or pipe opened that entity will occupy a slot in the PLFD table.

The PLFD structure will contain pointers to the GDPD for the file and to the GUFD for the file, if there is one. The PLFD is also where we keep the “type manager control block” which is an area used by the type manager bound to the file at open-time. The type manager’s “PLABEL” (code address) is also kept in the PLFD. Note that this field is usually stored as a short pointer and as a result may be represented for example as “eaca68.0”. This is really “a.eaca68” and can be displayed via “dcs eaca68”.

The macro FS_PLFD can be used to return the PLFD pointer for a given file number associated with a particular pin, for example format the PLFD for file #11 ($b) for the current pin:

```
$1e5 ($21d) nmdat > fv fs_plfd(,b) 'plfd_t'
```

The FS_FILE macros is quite useful for formatting all of the more important areas of the PLFD structure. Like the FS_PLFD macro it takes both a PIN and file number as input.

The GDPD is where we keep the current pointers for a file and it is also where we keep the “storage management control block”. The tail end of the GDPD has an SM_CB which is used by storage management to know how to prefetch information from a disk file and where to write information back to the file. Software updates the SM_CB prior to initiating a read from disk or a write to disk.

Files that are not opened MULTI or GMULTI will have their own unique GDPD. Files opened MULTI or GMULTI will, of course, share one. The linkage will be through the NEXT_PLFD field in the PLFD.

The GUFD structure exists only for disk files, so it is normal to find files that do not have a GUFD. All disk files had better have one!

Technically the GUFD is not a file system structure, it is actually part of storage management. Additionally the GUFD is kept immediately adjacent to the “VSOD” structure in the VSM “VSOD/GUFD Table” which will be discussed a bit later.

File System Structures: continued

The GUFD structure is also retained in most cases when a process closes a file. In other words, if a process is the last accessor of a disk file and closes it we do not release the GUFD rather it is appended to a least recently used (LRU) list. If the file is re-opened chances are the GUFD will be on that list and we can simply pull it off the LRU and use it making the file open process quicker.

The GUFD structure contains the virtual address of the file. There's also the GDPD pointer which is the end of a linked list of GDPD's associated with the file.

If the file is attached to XM that will tracked in the GUFD.

Finally, the GUFD contains information taken from the file label, things such as the EOF offset and number of records, the number of readers and writers. The GUFD also contains the pointer to the file label. (Technically the file label is not a file system structure, it is part of label management.)

The file label is an address that ends in $20 and the reason for that is that the FLAB_T type is part of a slightly larger structure "T_FILE_LABEL_ENTRY". This larger structure contains components of what will become the UFID or Unique File Identifier of a file (type is "UFID_TYPE"). And it also contains an offset to the extent block for the file. Replacing the $20 from a file label pointer with $00 allows it to be formatted using the "T_FILE_LABEL_ENTRY" type. This type is a boolean variant and it has the less-than-useful variant first so proper formatting requires specifying the TRUE variant, for example:

```
$1f8 ($70) nmdat > fv 15f.fc600 't_file_label_entry,TRUE'
```

Extent blocks migrate away from the file label as the file grows and more extents are added. The most recent extent block is always kept adjacent to the file label. Extent blocks are formatted with the type "T_EXTENT_BLOCK_ENTRY" which also suffers from the less-than-useful-variant-first problem so formatting with this type also requires the use of the TRUE variant. Each extent block will contain a pointer to the next extent block, if there is one. And it's worth noting too that all references to disks are volume ID's and not LDEV's.

File System Structures: continued

Finding The GUFD of an opened or closed file

GUFD’s for opened files are kept on a HASH_LINK from Storage Management Globals (KSO #210, type “SM_GLOBAL_REC”) and GUFD’s for closed files are kept on a least recently used (LRU) list also from SM Globals. The top portion of the GUFD_T shows these links:

```
GUFD_T =
   RECORD
   HASH_LINK                         : GUFD_PTR_TYPE;
   LRU_LINK                          : GUFD_PTR_TYPE;
   PREV_LRU_LINK                     : GUFD_PTR_TYPE;
```

When a process opens a disk file a search of these lists will be made to see if the file is opened or if the file has been recently closed. Files do not remain on the LRU indefinitely, the list can be no more than 1500 entries long and, if we should run short of GUFD entries for files being opened, the oldest file on the LRU will be pulled off, mapped out and the GUFD given over to a new file open request.

With MPE/iX 6.5 onward we also try to hold larger files, those over 1GB in size on the LRU as long as possible because performance can suffer if a very large file is mapped out all at once. These files are rotated around the LRU up to 16 times and at each rotation a 16th of the file is mapped out, from the bottom up. We map out from the bottom up so that if the file is removed from the LRU because it has been re-opened chances are the top portion of the file object will be referenced first and that will minimize the need to page-fault the data in from disk.

The FS_FIND_GUFD_ENTRY macro can be used to locate a GUFD for an opened or recently closed file. The macro takes as input the interval timer for the file you want to locate. Since the interval timer is kept in the extended file label that is a good way to get this information. It is not ever going to change so if you were looking at a memory dump and for example wanted to locate the GUFD for XL.PUB.SYS you could use the interval timer from the live system (assuming, of course, you’re logged on the system whose memory dump you’re looking at!).

A LISTFILE, –3 will display provide the file label pointer, for example:

File System Structures: continued

```
:listfile XL.PUB.SYS,-3
********************
FILE: XL.PUB.SYS

FILE CODE : 1032                FOPTIONS: BINARY,FIXED,NOCCTL,STD
BLK FACTOR: 1                   CREATOR : MANAGER.SYS
REC SIZE: 256(BYTES)            LOCKWORD:
BLK SIZE: 256(BYTES)            SECURITY--READ    : ANY
EXT SIZE: 0(SECT)                         WRITE   : ANY
NUM REC: 77293                            APPEND  : ANY
NUM SEC: 77824                            LOCK    : ANY
NUM EXT: 41                               EXECUTE : ANY
MAX REC: 4096000                        **SECURITY IS ON
                                FLAGS   : 1 ACCESSOR,SHARED,1 R
NUM LABELS: 0                   CREATED : TUE, MAR  4, 2003, 10:43 AM
MAX LABELS: 0                   MODIFIED: TUE, MAR  4, 2003, 10:44 AM
DISC DEV #: 1                   ACCESSED: MON, MAR 17, 2003,  4:09 AM
SEC OFFSET: 0                   LABEL ADDR: $00000013.$00204020
VOLNAME   : MPEXL_SYSTEM_VOLUME_SET:MEMBER1
```

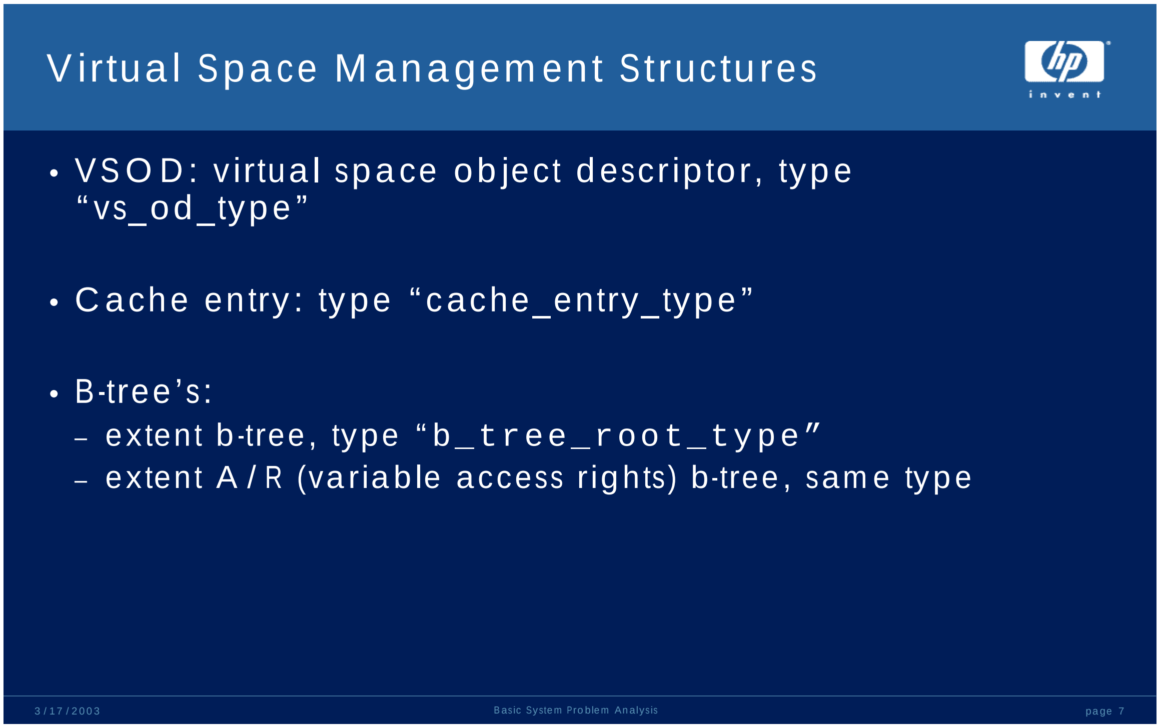
The file label address is 13.204020 and since the interval timer is found at offset $10 in the extended label structure replacing the $20 with a $10 points us right at it.

Now, using DAT/DEBUG indirection we can pass that value to the FS_FIND_GUFD macro:

```
$115 ($31) nmdebug > fs_find_gufd_entry([13.204010])
gufd_record pointer  : $ca016e60
File virtual address : $f8.0
End of file offset   : $12ded00
File name            : XL.PUB.SYS
```

And as this example shows, this macro works quite well in DEBUG too.

This page intentionally left blank



Virtual Space Management Structures

- VSOD: virtual space object descriptor, type “vs_od_type”
- Cache entry: type “cache_entry_type”
- B-tree’s:
  - extent b-tree, type “b_tree_root_type”
  - extent A/R (variable access rights) b-tree, same type

Notes:

Virtual Space Management Structures

There are two VSOD tables, one for files and one for everything else. Everything that has a virtual address has an entry in one of the two VSOD tables. These tables are:

VSOD/GUFD table, KSO #201
VSOD table, KSO #53

Both tables consist of entries whose type is "VS_OD_TYPE". The difference is that the VSOD/GUFD table, KSO #201 contains both VSOD entries and GUFD entries adjacent one another.

That means that if you know the address of a GUFD all you need to do is subtract the length of VS_OD_TYPE from it to get a pointer to the VSOD. Of course you can also use VAINFO to return that value by providing the virtual address of the file, for example:

```
$20b ($70) nmdat > fv ca12cda8 'gufd_t.file_vir_addr'

2e4.0

$20c ($70) nmdat > wl vainfo(2e4.0, 'vs_od_ptr')
$ca12cd48

$20d ($70) nmdat > wl ca12cda8-symlen('VS_OD_TYPE')
$ca12cd48
```

It makes sense to keep the VSOD and GUFD adjacent each other for file objects because when we need to map the object into memory we are going to need to know where the file is on disk.

Non-file objects such system objects, DST's and so forth reside in KSO #35 which is formatted using the same VS_OD_TYPE.

Both file, and non-file objects use identical methods of mapping the secondary storage image. This is done using several B-TREE's.

There is what is called an "AR B-TREE" or access rights B-TREE that keeps track of objects that have variable access rights. An object typically has a single access right (can you read it, write to it, what privilege level do you need etc.) . Program files, libraries and stacks are examples of objects that require variable access rights. A stack is actually a good example, the majority of the stack is accessible by a process running at normal user mode (ring 3). Portions of the stack, CM as well as NM are protected so you need to be at ring 2 (PM) or better to write to these areas.

The other B-TREE (non-AR) is for objects that have a single set of access rights.

Virtual Space Management Structures: continued

VSM uses VPN (virtual page number) Cache entries for portions of objects that are either in memory or on the way in to memory. These cache entries provide a fast means of linking the VSM structures such as the VSOD with the physical page addresses that the object occupies in real memory.

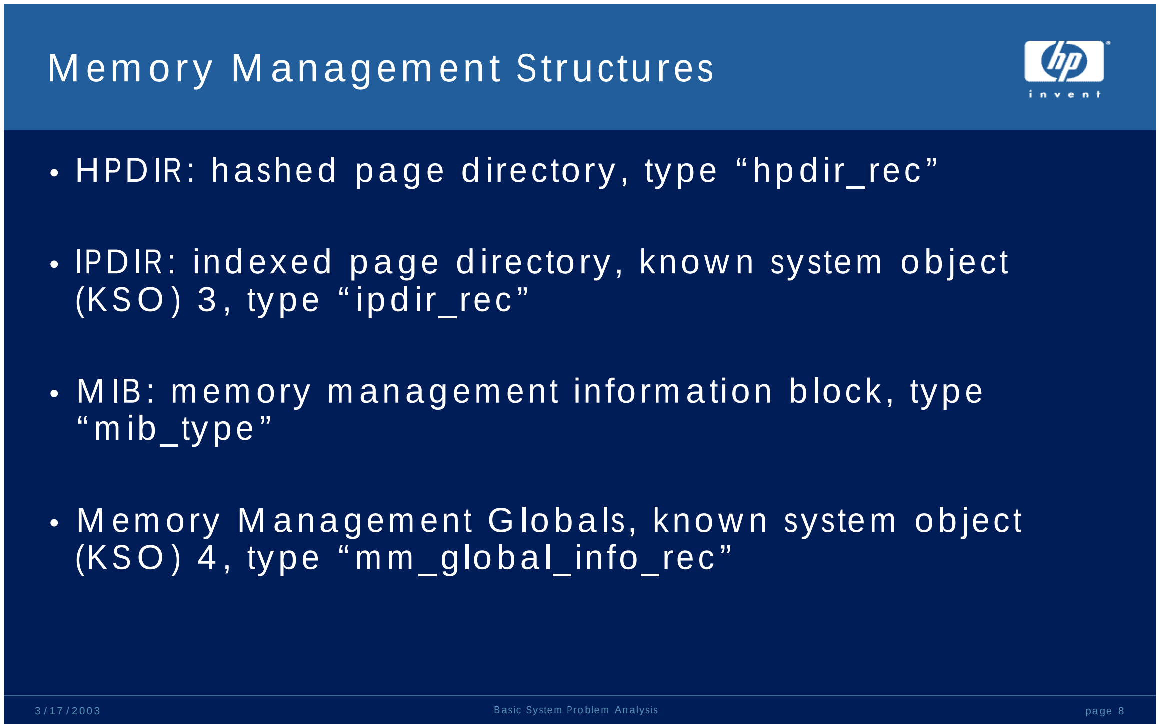
The VAINFO function is useful for finding out information about objects however certain information is unavailable in DEBUG. See HELP VAINFO for more details.

A couple useful macros to note are:

VS_PAGE_INFO which format virtual and secondary storage information for a given virtual address.

VS_ALLOCATION formats information about PID usage as well as SR6 and SR7 allocations.

This page intentionally left blank



Memory Management Structures

- HPDIR: hashed page directory, type “hpdir_rec”
- IPDIR: indexed page directory, known system object (KSO) 3, type “ipdir_rec”
- MIB: memory management information block, type “mib_type”
- Memory Management Globals, known system object (KSO) 4, type “mm_global_info_rec”

3/17/2003 Basic System Problem Analysis page 8

Notes:

Memory Management Structures

Memory management dove tails with Virtual Space management and keeps track of the real memory pages in use (among a lot of other things).

The hashed page directory or HPDIR is the structure used at the lowest levels of the OS to load the TLB (translation look-aside buffer) with a virtual-to-real translation.

When a virtual address is referenced, say a LDW (load word) instruction, the hardware will expect to find in the TLB a translation of that virtual address to a real address in memory. If there is no TLB translation found the hardware will go back to software (a page absence trap) and the first thing the software will do is try to locate an entry in this hashed page table. If an entry cannot be found in the hashed page table then we need to go back to VSM to find out if the virtual address is valid or not. If it is invalid the process or system will be aborted depending on where this trap occurred. If the address is valid then we need to go find the corresponding secondary storage address and start the process of swapping in a part of that object.

The IPDIR or indexed page directory, KSO 3 is a table consisting of a 64 byte entry for each physical 4K page of memory configured on the system. The header for this table is of type “LIST_IPDIR” and contains pointers to lists of pages.

These lists are

1. free list, free pages ready for use
2. present list, pages “owned” by a virtual object
3. ROC list, a list of recoverable overlay candidate pages
4. Critical list, pages held in reserve for critical operations (stack overflows etc.)
5. Unusable list, pages found to have had errors and deallocated by PDC

The meaning of “free” and “present” is obvious. A ROC page is a page that has not been touched recently or one whose owner, playing nice in the pool, has said can be taken if necessary. When a page is made ROC the contents of the page are written out to disk in anticipation that the page will be re-used but the I/O is done at a lower priority (sometimes called an “anticipatory write”). If the page winds up being made free so it can be reused the priority of any outstanding I/O is bumped up so it completes quickly.

The IPDIR also tracks when pages are “dirty”, that is they have been modified. If the page is part of a non-memory resident object it would need to be posted to disk in the event the page was to be reused for another purpose.

Finally MIB’s are Memory Manager I/O Blocks (not to be confused with MIB’s in SNMP). Whenever pages of memory are read in from disk or written out to disk a MIB is created to track that.

Memory Management Structures: continued

Some useful memory management macros are:

MM_ACTIVE_IO – lists all active I/O at the time of the dump, this macro probably won't work very well on a live system although you can try!

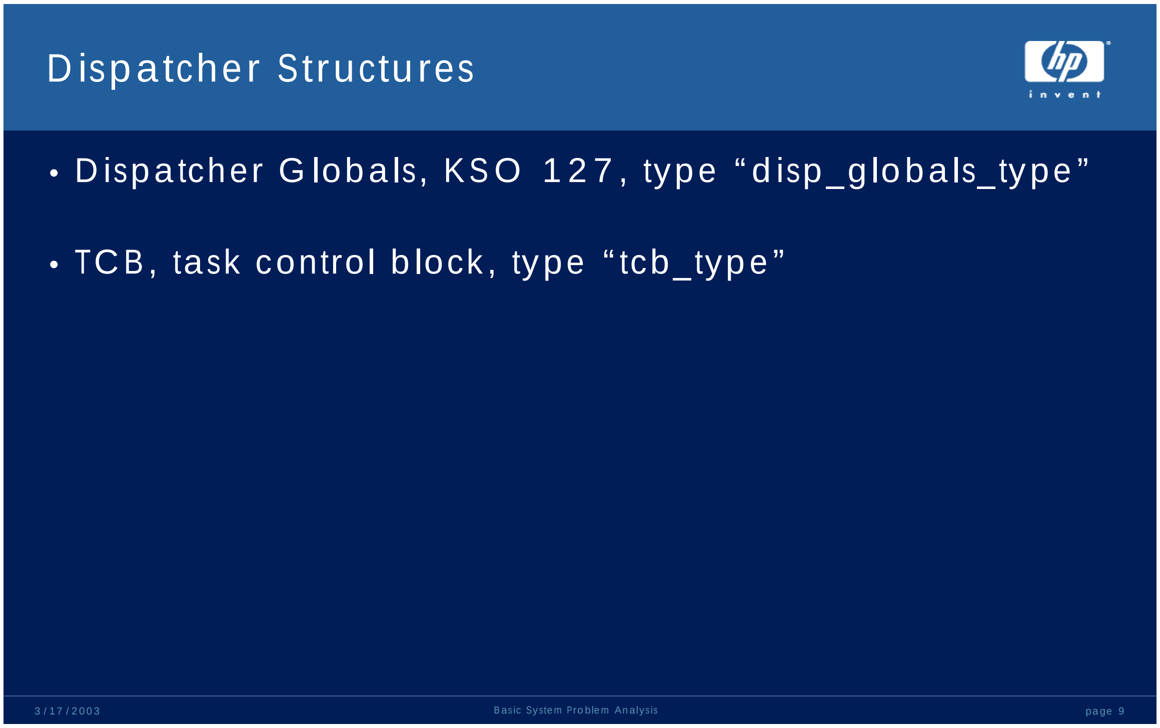
MM_COMPLETE_IO – lists all completed I/O. Often you may want to set a filter on a specific virtual address, for example ENV FILTER "a.c0000000" because the list can be quite lengthy.

MM_PAGE_INFO is like VS_PAGE_INFO and lists memory manager specific information about the virtual address.

MM_GLOBALS formats the data in the memory manager globals KSO 4.

NOTE: The memory management "hashed page directory" uses types prefixed with "HPDIR" but so do types that define objects in the hierarchical file system directory. If you see references to UFID's and names then you're looking at the directory structures!

This page intentionally left blank



Dispatcher Structures

- Dispatcher Globals, KSO 127, type “disp_globals_type”
- TCB, task control block, type “tcb_type”

3/17/2003 Basic System Problem Analysis page 9

Notes:

Dispatcher Structures

These are not the only dispatcher structures but they are worth mentioning in this context.

The TCB, or task control block rates a special mention because it is the structure use to save process state when a process looses the CPU.

It is kept in real memory but is “equivalently mapped” meaning that it is given an address in space 0 at the same offset it occupies in real memory.

The TCB function will return the real memory address of a TCB for a given PIN. You cannot format real memory using the FV command. Since the TCB is equivalently mapped into space 0 you can supply that creating a virtual address from the real address returned by the TCB function, for example:

```
$217 ($70) nmdat > wl TCB(pin)
$8c01d00

$218 ($70) nmdat > fv 0.TCB(pin) 'TCB_TYPE'

RECORD
   STACK_BASE            : 41854000
   STACK_LIMIT           : 418b4000
   STK_ADDR_NOT_IN_CACHE : 418aa000
```

Another useful thing to note about the TCB is that since it stores state information about a process it can use used in a dump to restore that state. That is, you can try to rewind a process to the state it was in the last time it lost the CPU. For example:

```
$219 ($70) nmdat > tr,d,i
       PC=a.0019fe78 system_abort
NM* 0) SP=418562e0 RP=a.00a51bc8 sm_quarantine_gufd+$1fc
NM  1) SP=418562e0 RP=a.00ee5a5c
tm_close_common.tm_unlink_plfd_and_gdpd+$184
NM  2) SP=418558e0 RP=a.00ee75cc tm_close_common+$1a98
NM  3) SP=41855860 RP=a.0158a8e4 tm_ord_fix_buf_disc+$1e4
NM  5) SP=418547e0 RP=a.01163d68 ?fclose_nm+$8

$21a ($70) nmdat > initnm tcb
$21b ($70) nmdat > tr,d,i
       PC=a.00394f78 $$lr_wa_14_long
NM* 0) SP=418565a0 RP=a.010722c0 sd_log_data+$504
NM  1) SP=418565a0 RP=a.01072ca8 sdump_data+$108
NM  2) SP=418564a0 RP=a.009d6140 sdump_dump_navigation_structure+$328c
NM  3) SP=418563e0 RP=a.00a51b98 sm_quarantine_gufd+$1cc
```

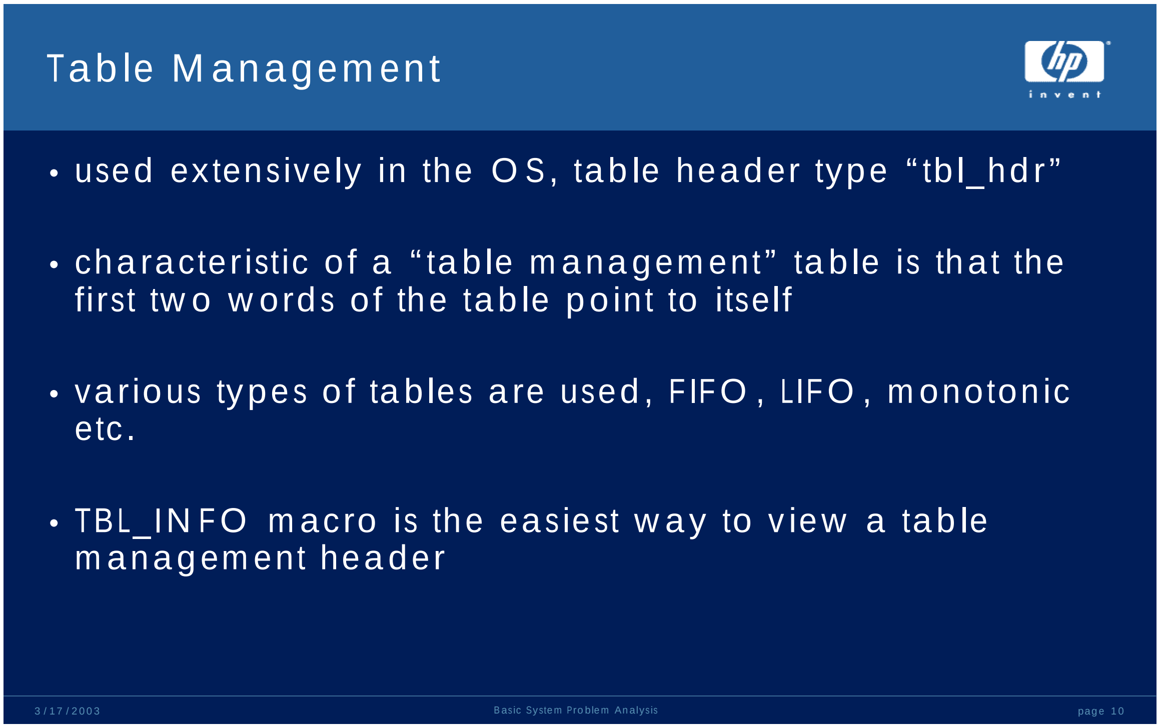


Table Management

- used extensively in the OS, table header type “tbl_hdr”
- characteristic of a “table management” table is that the first two words of the table point to itself
- various types of tables are used, FIFO, LIFO, monotonic etc.
- TBL_INFO macro is the easiest way to view a table management header

Notes:

Table Management

The use of table management in the MPE/iX OS is so pervasive that it warrants this mention.

Table management is essentially a centralized method for managing an object. A object is created and then transformed into a table. The table consists of a “header” and a “body”. The header is formatted using the type “TBL_HDR”. Each entry in the body portion will be whatever type the owner decides to use.

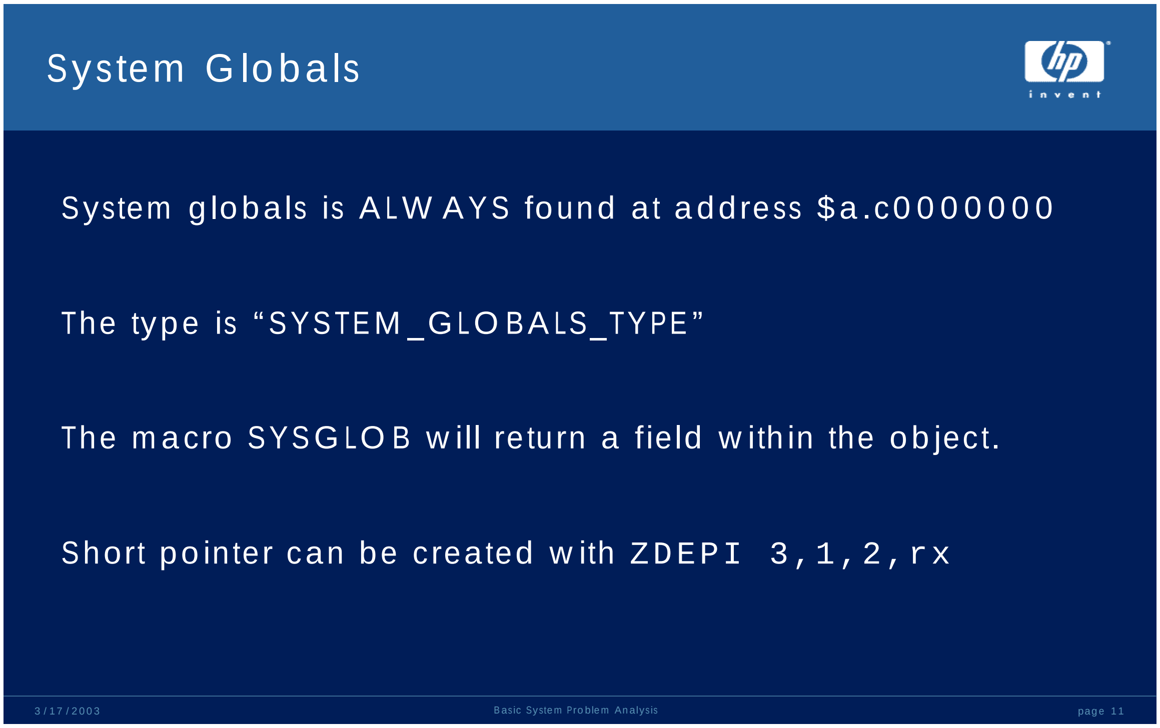
Tables can have various management types the most common being LIFO, last in first out, FIFO, first in first out and monotonic, meaning each entry in the table retains it position. The PLFD table is an example of a monotonic table. Each table entry or PLFD corresponds to a file number. So file number 10 needs to remain in the 10th position, it cannot be linked into a list after being closed because that reference would be lost.

A characteristic of a table-management table is that the first two words of the table header will be a pointer to itself. But it is important to stress that any object with this characteristic is not necessarily a table. An example of that would be the System Globals structure which is always found at address a.c0000000

```
$21c ($70) nmdat > dv a.c0000000,2
VIRT $a.c0000000 $ 0000000a c0000000
```

This is not a table header. The top portion of system globals is where we keep the KSO (known system object) pointers so KSO 0 is System Globals.

The TBL_INFO macro is quite useful in formatting table headers. It will decode the various table options and display information about the cache lists (LIFO or FIFO). One other thing it will do is walk down through the list of free entries which can take a while depending on how many there are. So if the macro appears to pause give it a few seconds before stopping it with a control-Y.



System Globals

System globals is ALWAYS found at address $a.c0000000

The type is “SYSTEM_GLOBALS_TYPE”

The macro SYSGLOB will return a field within the object.

Short pointer can be created with ZDEPI 3,1,2,rx

3/17/2003 Basic System Problem Analysis page 11

Notes:

System Globals

System globals is centralized table of information used by all parts of the OS. As was mentioned earlier, the KSO table is located at the top of System Globals.

At the end of the system globals structure is an array of 32 entries, one for each active processor (type “SPSD_ENTRY_TYPE”) that tracks information about the process active on each CPU.

The macro “SYSGLOB” is an easy way to get information out of the system globals structure but you need to know the name of the field you want to see. The macro also only handles individual fields within a structure and not structures themselves. So you could use SYSGLOB to display the highest PIN number used so far, whose field is PM_HIGHEST_PIN

```
$22c ($70) nmdat > wl SYSGLOB ("PM_HIGHEST_PIN")
$223
```

But you could not use it to display the “SG_SPSD_ENTRY” structure for CPU 3 by doing

```
$22d ($70) nmdat > wl SYSGLOB ("SG_SPSD_ARRAY[3]")
Error while retrieving the requested data from SYMVAL
```

Your best bet is to use FT to find the field or structure within system globals that you want to format and just use the FV command, for example:

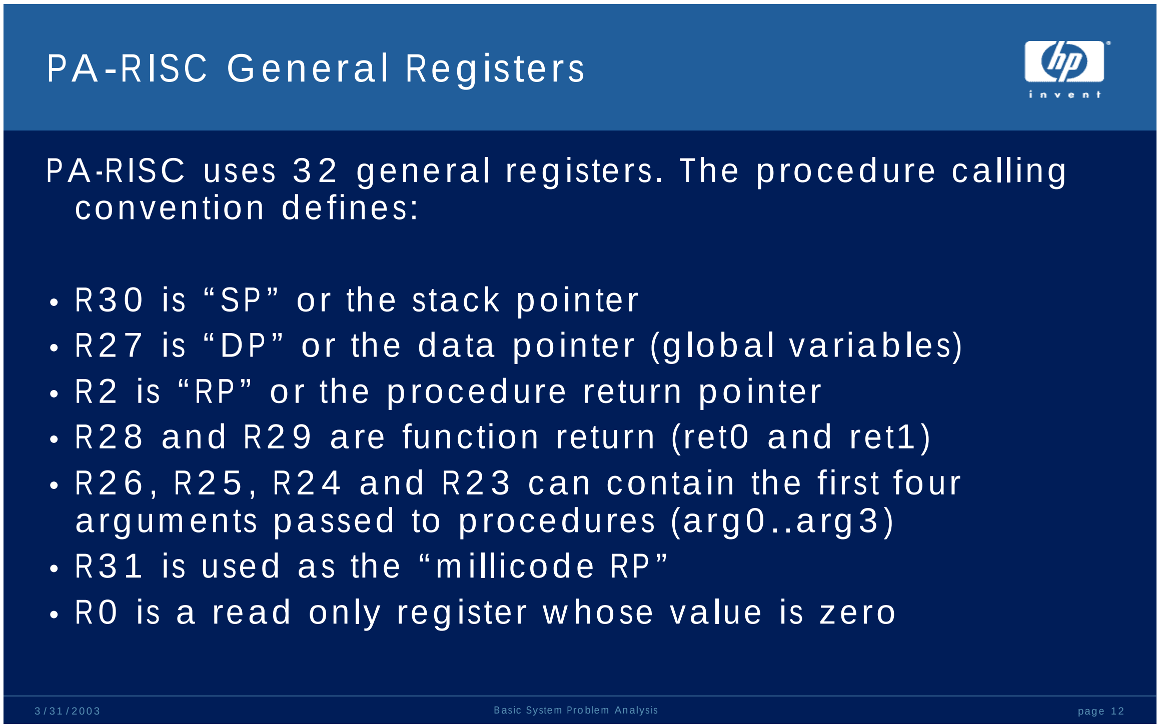
```
$22e ($70) nmdat > fv c0000000 'system_globals_type.sg_spsd_array[3]'
```

Finally, it is worth noting that the short pointer to system globals “C0000000” can be constructed with a single instruction. The ZDEPI or zero and deposit, immediate instruction

```
ZDEPI    3,1,2,rx
```

Where “rx” is R1 to R31. This says zero the target register and deposit the value 3 beginning in bit 1 for 2 bits to the left. Bits are numbered left to right, 0 to 31. So bit 1 would be the 2nd bit from the left and the quantity 3 in binary is “11”. The resulting value in binary is “11” followed by 30 zeros and when represented in hexadecimal you have “C0000000”.

When you see this sequence followed by the target register “rx” being used in a load or store with an offset you can match that to the offsets found by doing an FT “system_globals_type, m” remembering that the offsets are in hex from FT and decimal in the instruction. You can then see which field of system globals is being referenced.



PA-RISC General Registers

PA-RISC uses 32 general registers. The procedure calling convention defines:

- R30 is “SP” or the stack pointer
- R27 is “DP” or the data pointer (global variables)
- R2 is “RP” or the procedure return pointer
- R28 and R29 are function return (ret0 and ret1)
- R26, R25, R24 and R23 can contain the first four arguments passed to procedures (arg0..arg3)
- R31 is used as the “millicode RP”
- R0 is a read only register whose value is zero

Notes:

PA-RISC General Registers

The PARISC Instruction Set Reference Manual and the Procedure Calling Convention manual are pretty hard to come by. They are not at the docs.hp.com web site so it is worth spending a little time going over some of the basics of the hardware.

DEBUG, DAT and SAT use aliases for certain of the registers, SP, the stack pointer will always be R30. DP, the data pointer (global variables in a program context) will always be R27. RP or the procedure return pointer is R2.

The procedure calling convention specifies that the first four argument values being passed in a procedure call be placed in registers R26 to R23. The first parameter going into R26 and onward to R23. All additional parameters are placed into the stack frame that was created by the procedure making the call.

Parameters may require more than one register, a long pointer or LONGINT for example, will take two registers. If that occurs the registers must be aligned. This may result in one of the registers being skipped and left unused (more on this in a bit).

GR31 is called the “millicode RP” but it is also where the “BLE” instruction initially stores the current value of the PC register before making the branch. It moved to R2 immediately after that, in the “delay slot” of the branch.

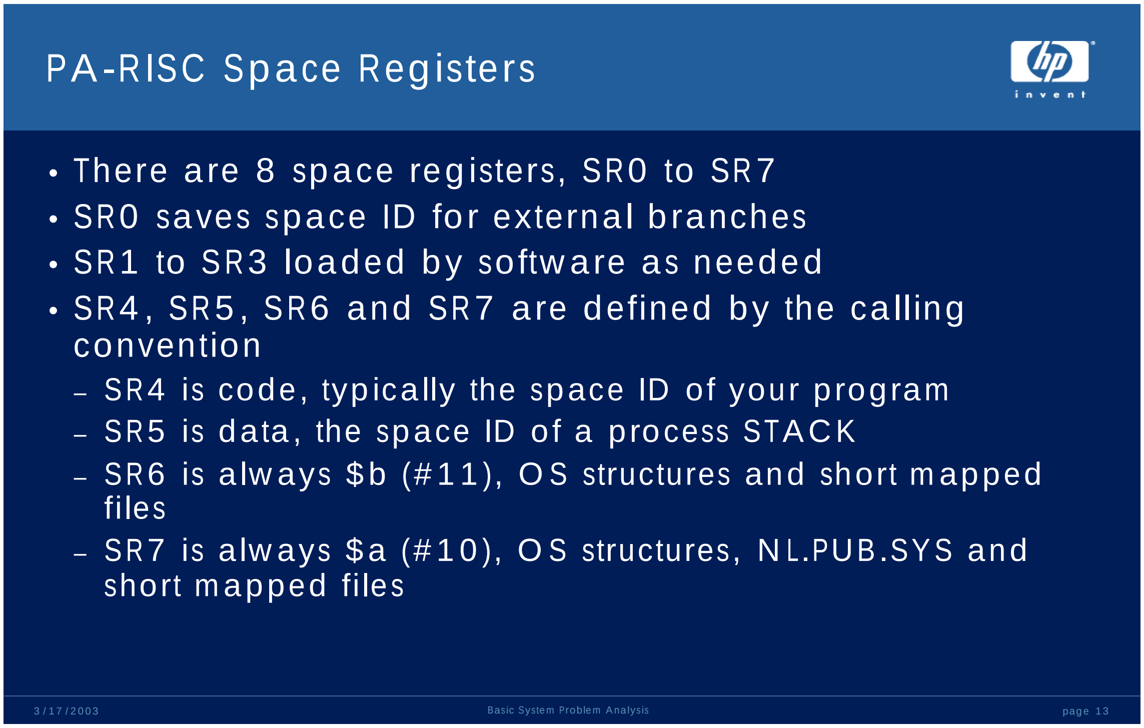
R0 is a scratch register that contains the value 0. It is cannot be written to but it is legal to use R0 as a target register when a value is not required. For example, the “NO OP” instruction (one that does nothing) is 08000240 OR r0, r0, r0. Logically OR R0, through R0 giving R0… nothing.

PA-RISC instructions are pipelined. Whenever a branch instruction is executed there is a delay in processing that branch while the target address is fetched. This delay affords the hardware the opportunity to execute an instruction in that delay slot. The delay can be nullified. Typically, the long branch code sequence moves R31, the target offset address of the BLE into R2 (architected “RP”) in this delay slot.

```
23ed0012  LDIL     $91a000,r31
e7e02640  BLE      800(sr4,r31)
081f0242  OR       r31,r0,r2
```

By the way, in DEBUG, DAT or SAT you can manually find out where this branch goes by doing:

```
$235 ($70) nmdat > dcs 91a000+#800
SYS $a.91a320
0091a320  tm_unlink_plfd         6bc23fd9  STW      r2,-20(sr0,r30)
```



PA-RISC Space Registers

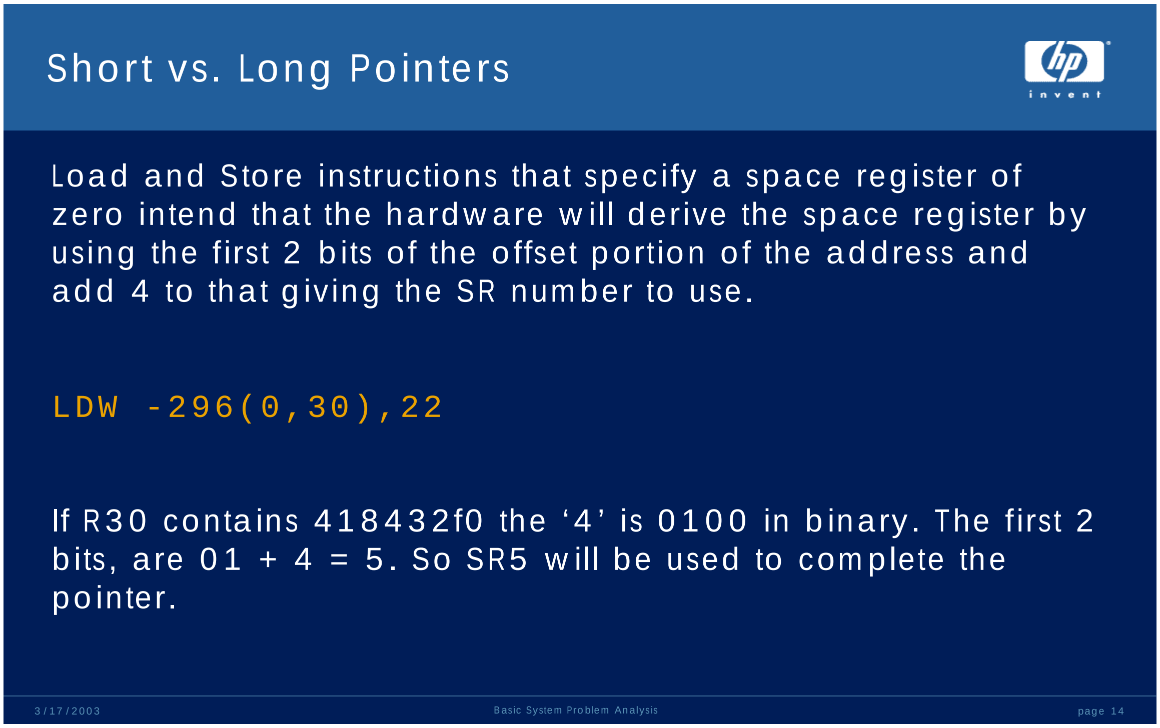
- There are 8 space registers, SR0 to SR7
- SR0 saves space ID for external branches
- SR1 to SR3 loaded by software as needed
- SR4, SR5, SR6 and SR7 are defined by the calling convention
  - SR4 is code, typically the space ID of your program
  - SR5 is data, the space ID of a process STACK
  - SR6 is always $b (#11), OS structures and short mapped files
  - SR7 is always $a (#10), OS structures, NL.PUB.SYS and short mapped files

3/17/2003 Basic System Problem Analysis page 13

Notes:

PA-RISC Space Registers

One point that the illustration did not mention is that SR5, 6 and 7 can only be written by code running at the highest privilege level which is 0 (user mode being 3).



Short vs. Long Pointers

Load and Store instructions that specify a space register of zero intend that the hardware will derive the space register by using the first 2 bits of the offset portion of the address and add 4 to that giving the SR number to use.

```
LDW -296(0,30),22
```

If R30 contains 418432f0 the ‘4’ is 0100 in binary. The first 2 bits, are 01 + 4 = 5. So SR5 will be used to complete the pointer.

Notes:

Short vs. Long Pointers

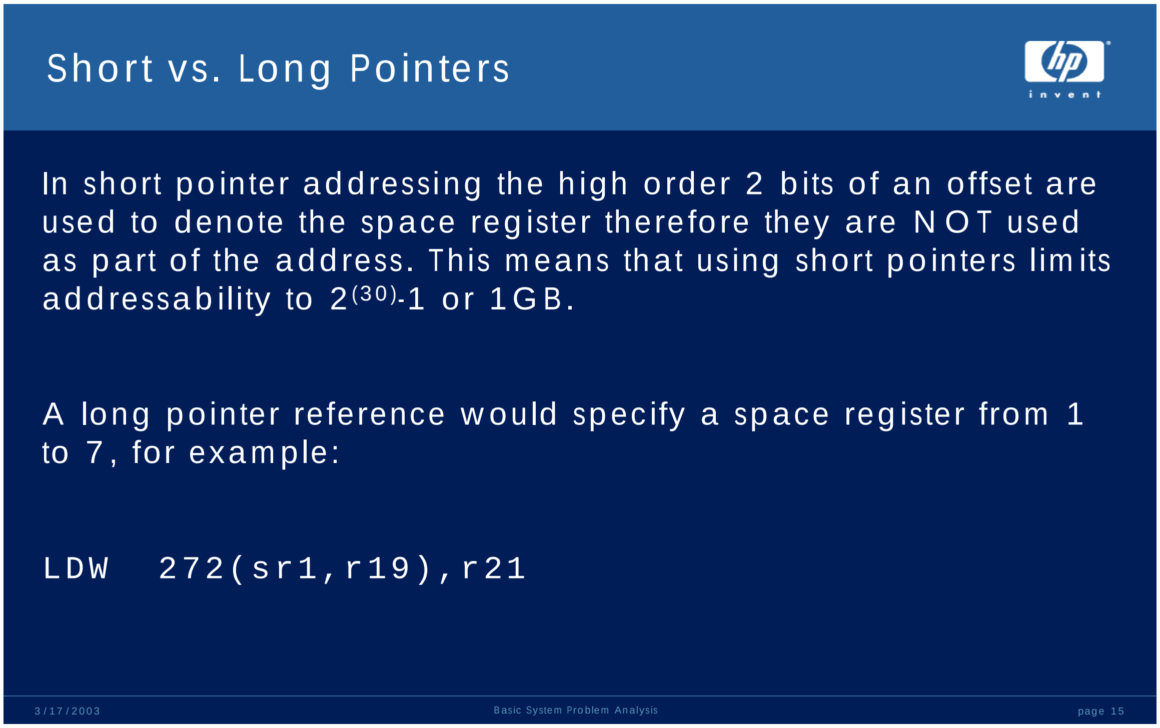
Here’s a table of where various addresses would be resolved using short pointer references:

Address Range	Space ID used
00000000 to 3fffffff	SR4
40000000 to 7fffffff	SR5
80000000 to bfffffff	SR6
c0000000 to ffffffff	SR7

These address ranges are also called “QUADS” as they represent ¼ of a 4GB space so each QUAD is 1GB of address space.

The OS uses SR6 and SR7 for resident and non-resident OS structures as well as NL.PUB.SYS. Whatever is left over can be allocated to files opened as short mapped for “share” access.

Files opened with exclusive access and short mapped will use SR5. Files opened this way will also NOT have their GUFD’s put on the storage management LRU list. The reason for this is simple; the GUFD contains the file’s virtual address. The file is mapped into a process SR5 space. If the file is closed and the GUFD saved that process might terminate invalidating the SR5. So we cannot retain the GUFD for a file opened with short mapped, exclusive access.



Short vs. Long Pointers

In short pointer addressing the high order 2 bits of an offset are used to denote the space register therefore they are NOT used as part of the address. This means that using short pointers limits addressability to $2^{(30)}$-1 or 1GB.

A long pointer reference would specify a space register from 1 to 7, for example:

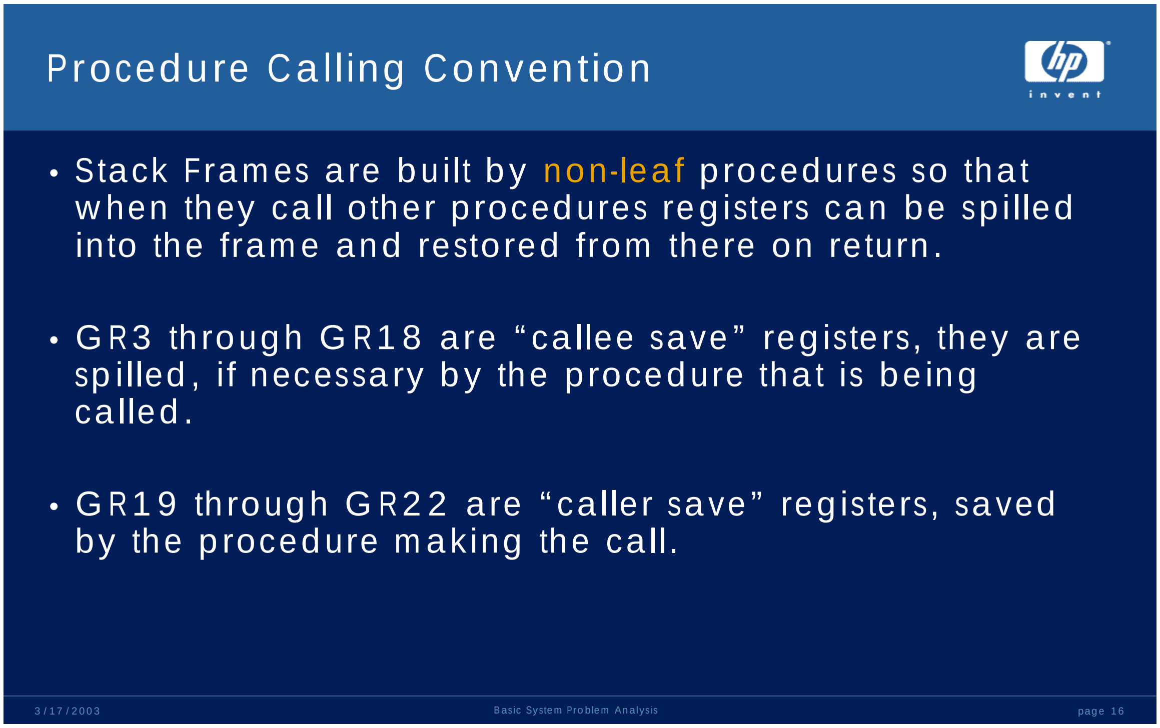
```
LDW   272(sr1,r19),r21
```

3/17/2003 Basic System Problem Analysis page 15

Notes:

Short vs. Long Pointers: continued

The slide says it all!



Procedure Calling Convention

- Stack Frames are built by non-leaf procedures so that when they call other procedures registers can be spilled into the frame and restored from there on return.
- GR3 through GR18 are “callee save” registers, they are spilled, if necessary by the procedure that is being called.
- GR19 through GR22 are “caller save” registers, saved by the procedure making the call.

3/17/2003 Basic System Problem Analysis page 16

Notes:

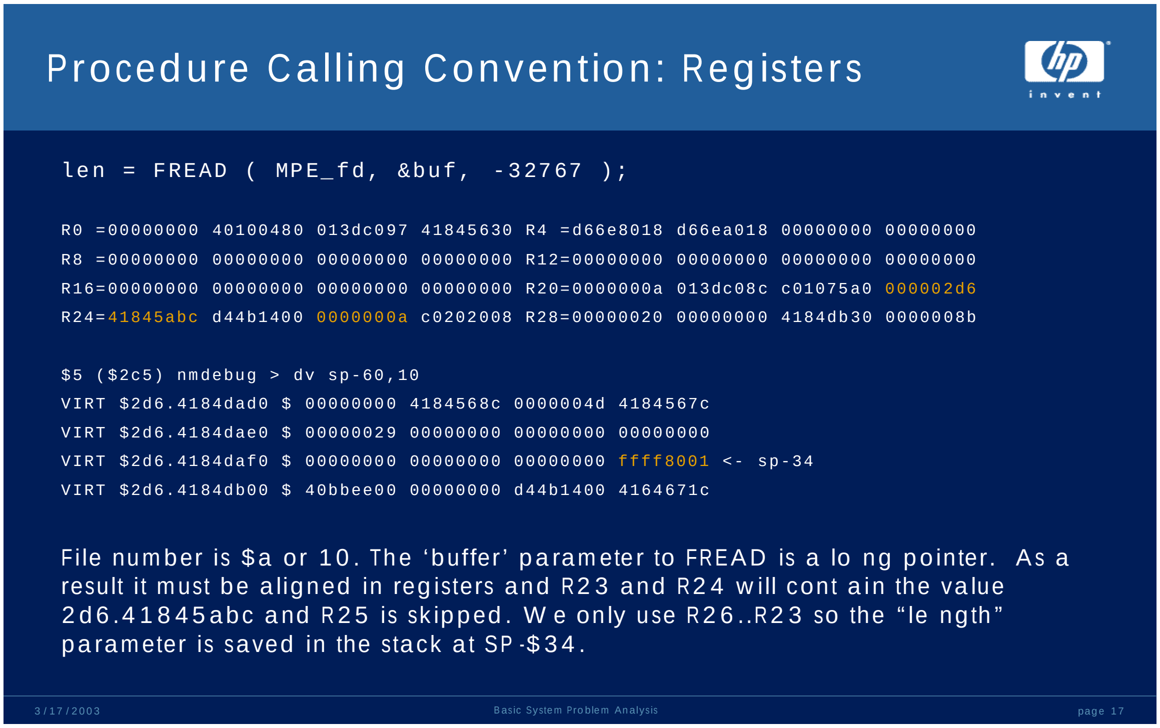
Procedure Calling Convention

There is considerably more to the procedure calling convention than is represented on the previous page but those are some of the more important points.

A stack frame only needs to be built if the current procedure will call other procedures. A leaf procedure would be one that makes no calls so there is no need for it to allocate space to spill registers.

It is worth noting that while the caller and callee are responsible for saving ranges of registers they are not obliged to save them all. For example, the “caller save” registers only need to be saved if they are active at the time of the call and need to be restored to their prior state on return from the call.

This fact can make it quite difficult, if not impossible to locate values that may have been available before a procedure call was made but which may have disappeared after the call because they are no longer needed.



Procedure Calling Convention: Registers

```
len = FREAD ( MPE_fd, &buf, -32767 );

R0 =00000000 40100480 013dc097 41845630 R4 =d66e8018 d66ea018 00000000 00000000
R8 =00000000 00000000 00000000 00000000 R12=00000000 00000000 00000000 00000000
R16=00000000 00000000 00000000 00000000 R20=0000000a 013dc08c c01075a0 000002d6
R24=41845abc d44b1400 0000000a c0202008 R28=00000020 00000000 4184db30 0000008b

$5 ($2c5) nmdebug > dv sp-60,10
VIRT $2d6.4184dad0 $ 00000000 4184568c 0000004d 4184567c
VIRT $2d6.4184dae0 $ 00000029 00000000 00000000 00000000
VIRT $2d6.4184daf0 $ 00000000 00000000 00000000 ffff8001 <- sp-34
VIRT $2d6.4184db00 $ 40bbee00 00000000 d44b1400 4164671c
```

File number is $a or 10. The ‘buffer’ parameter to FREAD is a long pointer. As a result it must be aligned in registers and R23 and R24 will contain the value 2d6.41845abc and R25 is skipped. We only use R26..R23 so the “length” parameter is saved in the stack at SP-$34.

Notes:

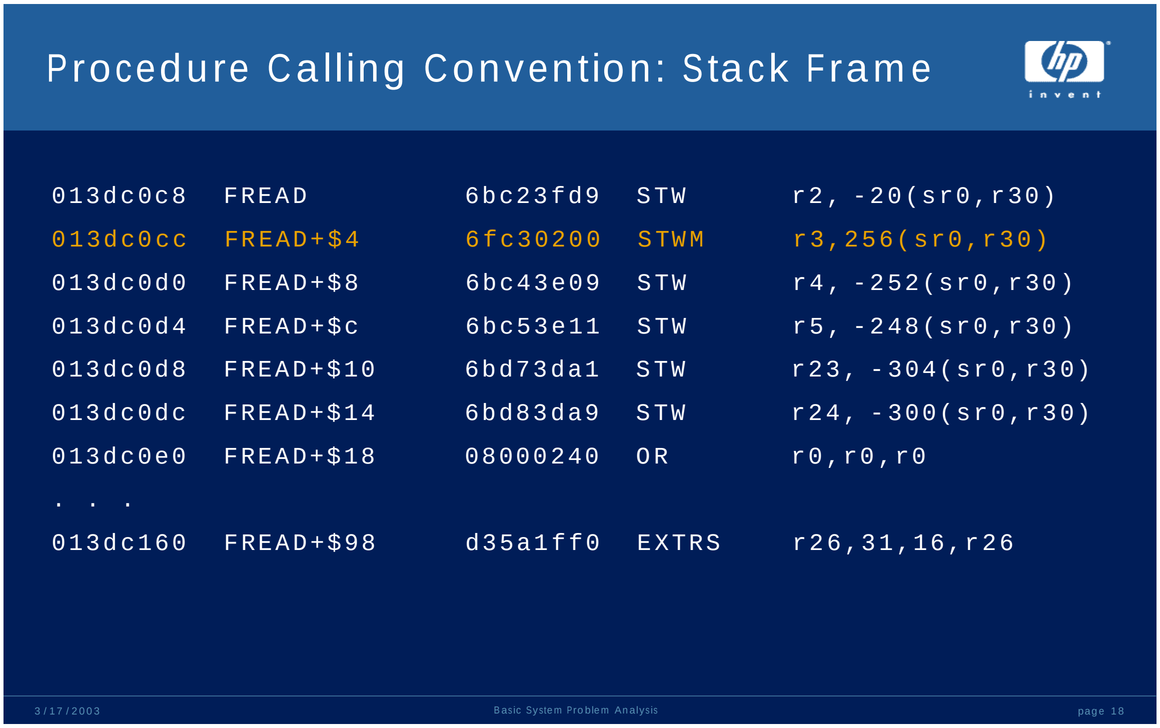
Procedure Calling Convention: Registers

The previous page illustrates how parameters for a call to FREAD would be passed. The convention says that R26 to R23 are used for the first 4 arguments.

R26 is referred to as “arg0” with R25 “arg1”, R24 “arg2” and R23 “arg3”.

The second parameter to FREAD is defined as a long pointer, which takes 64 bits (2 words). The procedure calling convention specifies that 64 bit quantities be passed with the high order word in an ODD argument register. In this example the space ID portion of the pointer is the high order word and it is loaded into R23 (arg3) so that the offset portion of the address can be loaded adjacent to it in R24 (arg2). This leaves R25 (arg1) unused and the value is whatever happened to be there the last time the register was used.

Since all four of the argument registers are used the “length” parameter must be saved into the stack. Here also, the first four positions are skipped and reserved for later use as the values in R26..R23 may eventually need to be spilled into the stack frame.



Procedure Calling Convention: Stack Frame

```
013dc0c8   FREAD           6bc23fd9   STW       r2, -20(sr0,r30)
013dc0cc   FREAD+$4        6fc30200   STWM      r3,256(sr0,r30)
013dc0d0   FREAD+$8        6bc43e09   STW       r4, -252(sr0,r30)
013dc0d4   FREAD+$c        6bc53e11   STW       r5, -248(sr0,r30)
013dc0d8   FREAD+$10       6bd73da1   STW       r23, -304(sr0,r30)
013dc0dc   FREAD+$14       6bd83da9   STW       r24, -300(sr0,r30)
013dc0e0   FREAD+$18       08000240   OR        r0,r0,r0
. . .
013dc160   FREAD+$98       d35a1ff0   EXTRS     r26,31,16,r26
```

3/17/2003 Basic System Problem Analysis page 18

Notes:

Procedure Calling Convention: Stack Frame

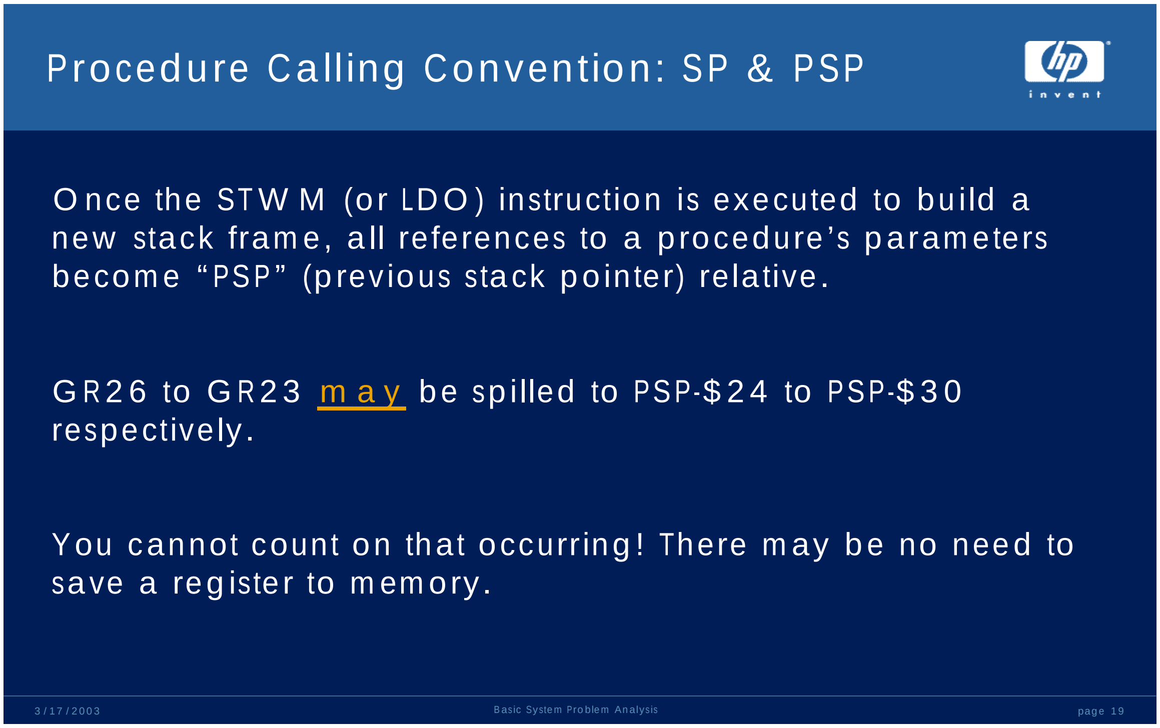
The illustration shows the first things that FREAD does when it is called. These steps are roughly the same for all OS procedures;

1. the current value of R2 (RP) is saved at SP-#20, that will be picked up at the end of the procedure to return to the caller. The caller will have had to be sure that R2 does contain a pointer back to it!
2. if necessary a stack frame is built. One way is using a "store word and modify" instruction (STWM) which in this particular form saves the register R3 on top of stack and then adds the offset value to R30 moving SP out that many words. Occasionally you may find the LDO (load offset) being used for this purpose.
3. the procedure then saves any of the "callee save" registers it needs to as well as any of the register-passed parameters that it needs to.

What you will notice in this case is that FREAD is not saving R26, which holds the file number.

It does not need to because all FREAD does is call "fread_nm" which also defines file number as the first parameter. This is impossible to know without the source code so you may be forced to make a few scientific guesses for other routines.

The key point being that you cannot expect that arguments passed in registers 26..23 will be saved off to the stack so that you can conveniently level down to them and find what you want.



Procedure Calling Convention: SP & PSP

Once the STWM (or LDO) instruction is executed to build a new stack frame, all references to a procedure's parameters become "PSP" (previous stack pointer) relative.

GR26 to GR23 may be spilled to PSP-$24 to PSP-$30 respectively.

You cannot count on that occurring! There may be no need to save a register to memory.

Notes:

Procedure Calling Convention: SP & PSP

SP is a real register, R30 by convention. PSP is not. It is the value of SP with the size of the current frame subtracted.

Let’s say you run a program and set a break point at FREAD. At the point before the stack frame is built you could count on the argument registers 26..23 being correct and that SP-negative addresses would give you any additional parameters that might be there.

Once the stack frame is built those SP-negative addresses become PSP-negative addresses.

And, as the procedure executes and calls other procedures you are less able to assume anything about where parameter values might be if they were not initially spilled to the stack. The only way to be sure is to read the instructions the procedure executed.

Here’s a trick for helping to find how registers get moved around.

Let’s say you have the following stack trace in a dump:

```
     PC=a.0019fe78 system_abort
* 0) SP=418562e0 RP=a.00a51bc8 sm_quarantine_gufd+$1fc
  1) SP=418562e0 RP=a.00ee5a5c
tm_close_common.tm_unlink_plfd_and_gdpd+$184
  2) SP=418558e0 RP=a.00ee75cc tm_close_common+$1a98
  3) SP=41855860 RP=a.0158a8e4 tm_ord_fix_buf_disc+$1e4
  4) SP=418548a0 RP=a.01164370 fclose_nm+$5d4
  5) SP=418547e0 RP=a.01163d68 ?fclose_nm+$8
       export stub: a.013d22a8 FCLOSE+$b8
  6) SP=41854560 RP=a.013d21bc ?FCLOSE+$8
       export stub: 298.00279b68 cr_fclose+$1c
  7) SP=418544a0 RP=298.00272350 COB_CLOSE+$17c
  8) SP=41854468 RP=298.0026e804 ?COB_CLOSE+$8
       export stub: 97c.0000e1d4
  9) SP=418543f0 RP=97c.00000000
     (end of NM stack)
```

You level down to look for the file number at FCLOSE (lev 6). You notice that PSP-$24 is zero, no file number there and R26 doesn’t look good either.

```
$23b ($70) nmdat > env filter 'r26,'
$23c ($70) nmdat r26,> dc pc-b8,bc/4
013d2288  FCLOSE+$98     d35a1ff0  EXTRS    r26,31,16,r26

$23d ($70) nmdat r26,> env filter ',r26'
$23e ($70) nmdat ,r26> dc pc-b8,bc/4
013d2288  FCLOSE+$98     d35a1ff0  EXTRS    r26,31,16,r26
```

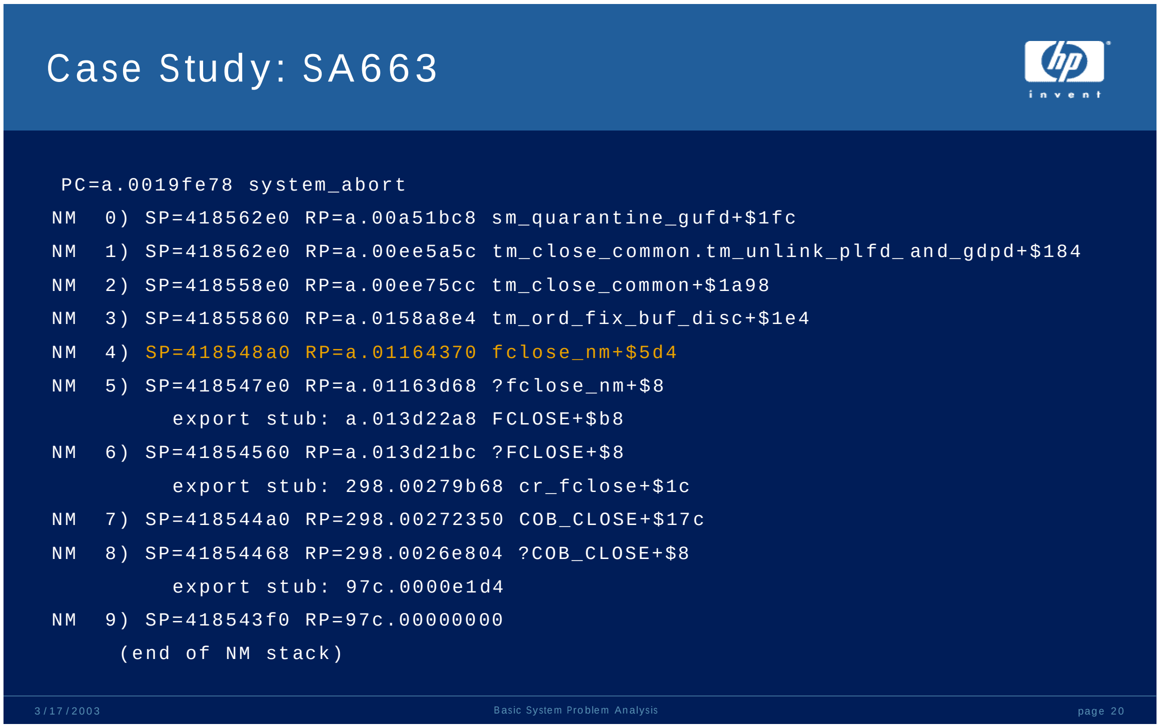
Procedure Calling Convention: SP & PSP

This, admittedly very simple example shows how to look for r26 appearing either as the source register or destination register to see whether it has been moved. In this example the only reference to R26 from the beginning of FCLOSE to the current offset is that one instruction. All that is doing is extracting the right 16 bits of the register because file number is defined as a 16 bit value.

This also assumes (which is not always a good thing) that FCLOSE has not hopped around and executed code past the current PC location which might have saved R26 someplace you could find it. You can determine that by reading each instruction from top to bottom and essentially “replaying” the procedure based on the data you find in registers and on the stack (assuming you have the time and inclination!).

Here we can see that FCLOSE did not save R26 to the stack. It had no need to. Perhaps fclose_nm did?

This page intentionally left blank



Case Study: SA663

```
 PC=a.0019fe78 system_abort
NM  0) SP=418562e0 RP=a.00a51bc8 sm_quarantine_gufd+$1fc
NM  1) SP=418562e0 RP=a.00ee5a5c tm_close_common.tm_unlink_plfd_and_gdpd+$184
NM  2) SP=418558e0 RP=a.00ee75cc tm_close_common+$1a98
NM  3) SP=41855860 RP=a.0158a8e4 tm_ord_fix_buf_disc+$1e4
NM  4) SP=418548a0 RP=a.01164370 fclose_nm+$5d4
NM  5) SP=418547e0 RP=a.01163d68 ?fclose_nm+$8
         export stub: a.013d22a8 FCLOSE+$b8
NM  6) SP=41854560 RP=a.013d21bc ?FCLOSE+$8
         export stub: 298.00279b68 cr_fclose+$1c
NM  7) SP=418544a0 RP=298.00272350 COB_CLOSE+$17c
NM  8) SP=41854468 RP=298.0026e804 ?COB_CLOSE+$8
         export stub: 97c.0000e1d4
NM  9) SP=418543f0 RP=97c.00000000
     (end of NM stack)
```

3/17/2003 Basic System Problem Analysis page 20

Notes:

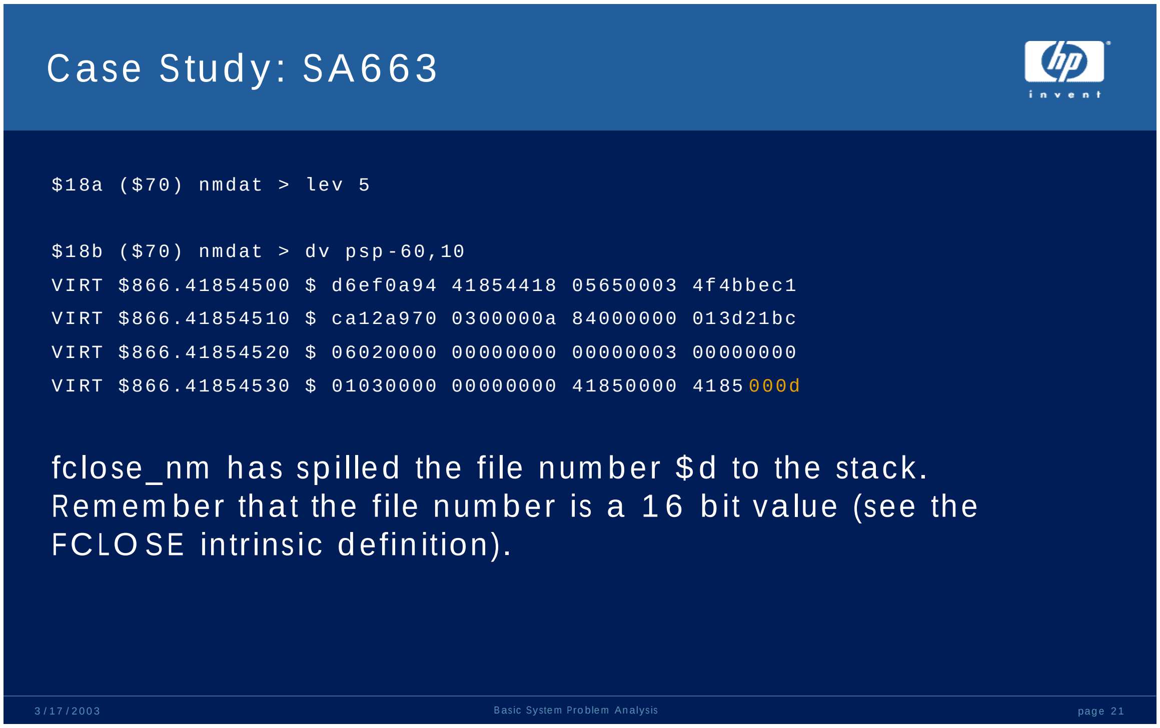
Case Study: SA663

A system abort 663 occurs when a problem is encountered in a file system structure but the Subsystem Dump facility has not been enabled by running SDUTIL. Had it been enabled the file system and storage management would have been able to quarantine the file preventing it from being accessed until it could be checked and, if necessary restored with a good copy.

Since the failure is the result of a problem with a file the first thing to do would be to find out what file that is.

We already know that FCLOSE would not have saved the file number in the stack so there is no point looking there. The routine fclose_nm may have.

Note! The “level” (lev) command is used to move to a particular stack frame in a trace. You always move to the “level” one past the code you want to look at. We want to look at “fclose_nm” so we must set the level to 5 not 4.



Case Study: SA663

```
$18a ($70) nmdat > lev 5

$18b ($70) nmdat > dv psp-60,10
VIRT $866.41854500 $ d6ef0a94 41854418 05650003 4f4bbec1
VIRT $866.41854510 $ ca12a970 0300000a 84000000 013d21bc
VIRT $866.41854520 $ 06020000 00000000 00000003 00000000
VIRT $866.41854530 $ 01030000 00000000 41850000 4185000d
```

fclose_nm has spilled the file number $d to the stack. Remember that the file number is a 16 bit value (see the FCLOSE intrinsic definition).

3/17/2003 Basic System Problem Analysis page 21

Notes:

Case Study: SA663 continued

Yup, it did save the file number in the stack. Well, to be honest we would have to assume that the $d is the file number just by looking at the value in PSP-$24. If we wanted to be absolutely certain it is (and absolute certainty is handy a lot of the time) then we would need to examine the code that fclose_nm executed to see if it did spill the file number parameter to the stack.

```
01163d9c  fclose_nm         6bc23fd9  STW      r2,-20(sr0,r30)
01163da0  fclose_nm+$4      6fc30500  STWM     r3,640(sr0,r30)
01163da4  fclose_nm+$8      6bc43b09  STW      r4,-636(sr0,r30)
01163da8  fclose_nm+$c      6bc53b11  STW      r5,-632(sr0,r30)
01163dac  fclose_nm+$10     6bc63b19  STW      r6,-628(sr0,r30)
01163db0  fclose_nm+$14     6bc73b21  STW      r7,-624(sr0,r30)
01163db4  fclose_nm+$18     6bc83b29  STW      r8,-620(sr0,r30)
01163db8  fclose_nm+$1c     6bc93b31  STW      r9,-616(sr0,r30)
01163dbc  fclose_nm+$20     67da3abd  STH      r26,-674(sr0,r30)
01163dc0  fclose_nm+$24     67d93ab5  STH      r25,-678(sr0,r30)
01163dc4  fclose_nm+$28     67d83aad  STH      r24,-682(sr0,r30)
01163dc8  fclose_nm+$2c     67d73aa5  STH      r23,-686(sr0,r30)
```

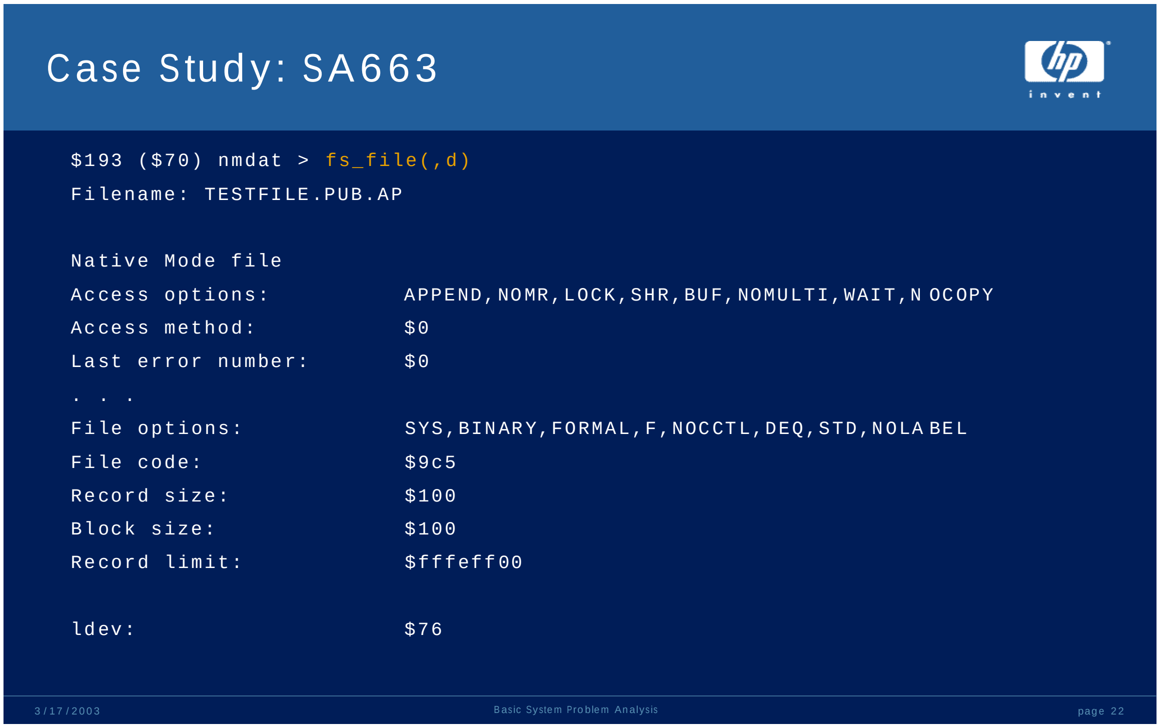
Yes, the file number really was saved to the stack frame. Note that a store half-word was used since the file number is a 16 bit value.

The value of R26 should be stored to PSP-$24 so if we want to check we can do the math:

```
$243 ($70) nmdat > =#674-#640
$22
```

The location it was saved to, SP-#674 less the size of the stack frame, #640 results in the value hexadecimal 22. Since it is a half word quantity and is aligned in the right 16 bits of the value it should really be saved at PSP-$22 and that’s exactly where it is.

```
VIRT $866.4185453c $ 4185000d
                           --  SP-$21
                         --    SP-$22
                       --      SP-$23
                     --        SP-$24
```



Case Study: SA663

```
$193 ($70) nmdat > fs_file(,d)
Filename: TESTFILE.PUB.AP

Native Mode file
Access options:          APPEND,NOMR,LOCK,SHR,BUF,NOMULTI,WAIT,NOCOPY
Access method:           $0
Last error number:       $0
. . .
File options:            SYS,BINARY,FORMAL,F,NOCCTL,DEQ,STD,NOLABEL
File code:               $9c5
Record size:             $100
Block size:              $100
Record limit:            $fffeff00

ldev:                    $76
```

3/17/2003 Basic System Problem Analysis page 22

Notes:

Case Study: SA663 continued

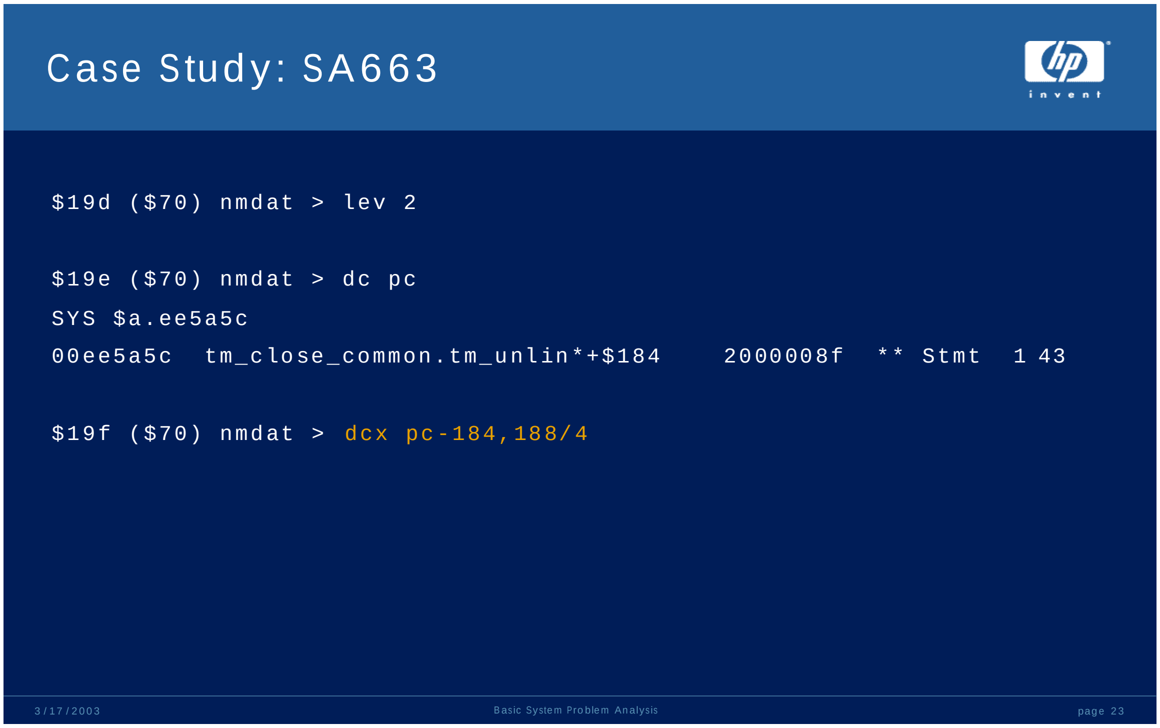
Now that we have the file number we can use the FS_FILE macro to display information about this file. The most important thing is the file name because if this file is damaged and could not be quarantined there is a good chance someone else may try to access the file which could cause another system abort.

Notice the record limit on the file. That’s pretty large. The proper way to convert that to decimal is this:

```
$245 ($70) nmdat > =U32(fffeff00),d
#4294901504

$246 ($70) nmdat > =fffeff00,d
#-65792
```

Unless otherwise directed numbers are treated as signed 32 bit values. So if you do not say that the value should be treated as an unsigned value all you get is a negative number back.

If you look at the type “GUFD_T” you will see that the field “FILE_SIZE” is defined as a BIT32 which is how an unsigned integer is defined.



Case Study: SA663

```
$19d ($70) nmdat > lev 2

$19e ($70) nmdat > dc pc
SYS $a.ee5a5c
00ee5a5c  tm_close_common.tm_unlin*+$184    2000008f  ** Stmt  1 43

$19f ($70) nmdat > dcx pc-184,188/4
```

3/17/2003 Basic System Problem Analysis page 23

Notes:

Case Study: SA663 continued

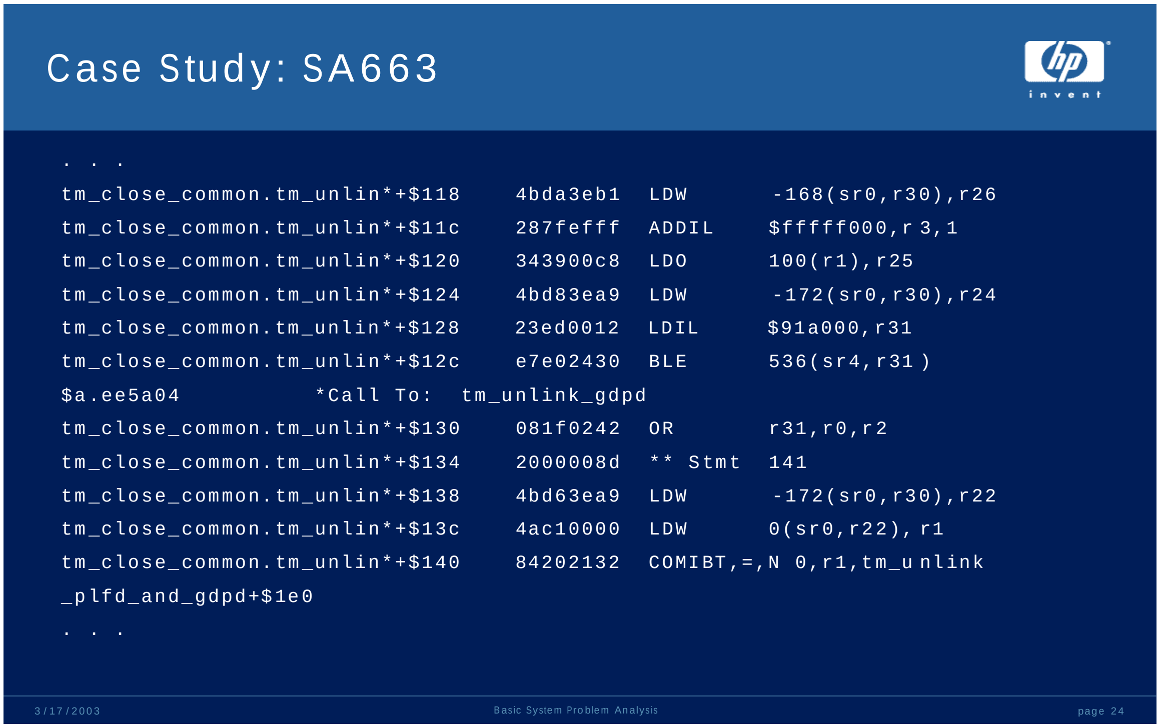
Recognizing that we do not have the source code and cannot just go look at what may have caused the problem we can try to find out some relevant information.

The 2nd level procedure tm_unlink_plfd_and_gdpd of tm_close_common made the call to sm_quarantine_gufd. The question is, why did it want to quarantine the GUFD?

We can dump out the code for that 2nd level routine from its beginning up to the location of the PC counter in that routine as shown in the illustration above. The DC command expects a “count” and so we provide a value 4 bytes larger than the current PC offset and then because that offset represents bytes and we want 32 bit words we divide by 4.

That formula will produce an inclusive list of all instructions executed from the beginning of the procedure through the instruction pointed to by PC.

Also note that while the DC command could have been used, the DCX macro was used instead. This macro translates the “long call” sequence of LDIL and BLE and displays the name of the procedure being called by that sequence.



Case Study: SA663

```
. . .
tm_close_common.tm_unlin*+$118    4bda3eb1  LDW       -168(sr0,r30),r26
tm_close_common.tm_unlin*+$11c    287fefff  ADDIL     $fffff000,r 3,1
tm_close_common.tm_unlin*+$120    343900c8  LDO       100(r1),r25
tm_close_common.tm_unlin*+$124    4bd83ea9  LDW       -172(sr0,r30),r24
tm_close_common.tm_unlin*+$128    23ed0012  LDIL      $91a000,r31
tm_close_common.tm_unlin*+$12c    e7e02430  BLE       536(sr4,r31 )
$a.ee5a04          *Call To:  tm_unlink_gdpd
tm_close_common.tm_unlin*+$130    081f0242  OR        r31,r0,r2
tm_close_common.tm_unlin*+$134    2000008d  ** Stmt  141
tm_close_common.tm_unlin*+$138    4bd63ea9  LDW       -172(sr0,r30),r22
tm_close_common.tm_unlin*+$13c    4ac10000  LDW       0(sr0,r22), r1
tm_close_common.tm_unlin*+$140    84202132  COMIBT,=,N 0,r1,tm_u nlink
_plfd_and_gdpd+$1e0
. . .
```

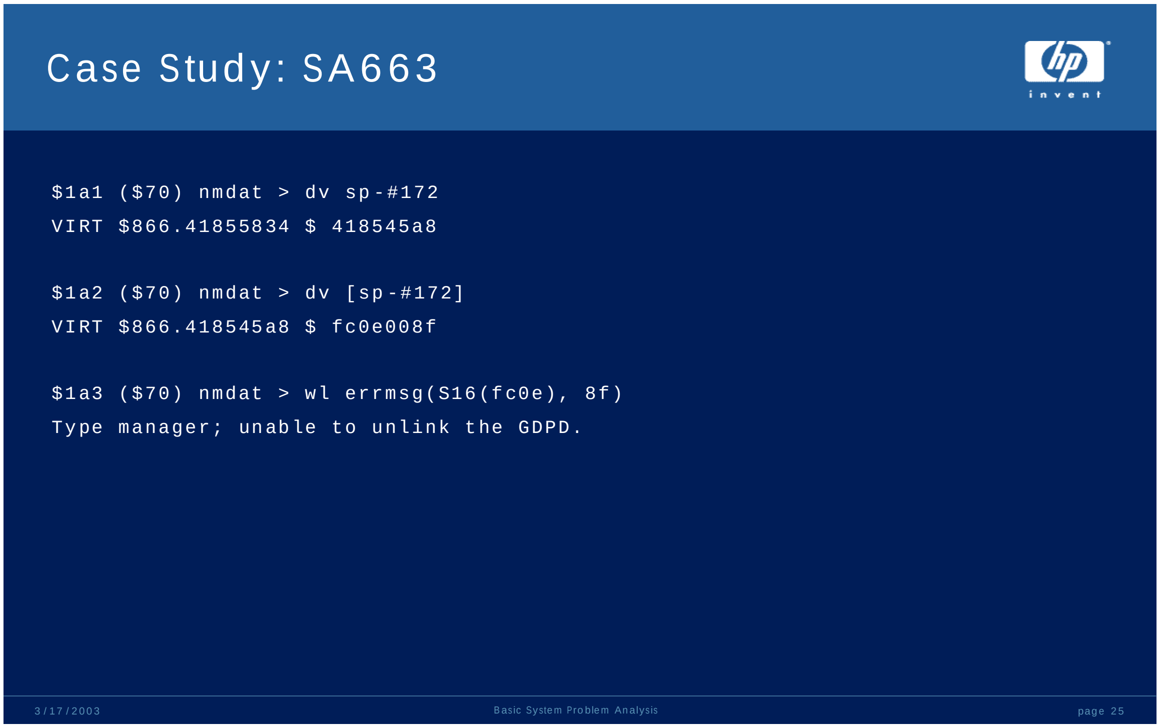
Notes:

Case Study: SA663 continued

Illustrated above is a small portion of the code that would be displayed by the DCX macro call.

From this we can see that the 2nd level procedure made a call to the procedure tm_unlink_gdpd. On return from that procedure a value a SP-#172 was loaded into R22. Then R22 was used to load a value into R1. The value in R1 was compared to zero (COMBIT is compare, immediate branch if true) and if the condition was met PC would have moved to a point BEYOND where we would have called sm_quarantine_gufd. So the next thing we would want to look at is that value since it is looks like we did not take that branch.

Note: Technically it would be incorrect to say that the code did not take that branch. In actual fact the branch may have been taken and another branch might have move PC back to the instruction just following the branch at offset $140. If after looking at the value it should have used you find that it should have taken the branch the next steps (and this is where it gets time consuming) would be to walk through the instructions in an attempt to replay the code. This becomes so time consuming that it really is not worth the investment in time.



Case Study: SA663

```
$1a1 ($70) nmdat > dv sp-#172
VIRT $866.41855834 $ 418545a8

$1a2 ($70) nmdat > dv [sp-#172]
VIRT $866.418545a8 $ fc0e008f

$1a3 ($70) nmdat > wl errmsg(S16(fc0e), 8f)
Type manager; unable to unlink the GDPD.
```

3/17/2003 Basic System Problem Analysis page 25

Notes:

Case Study: SA663 continued

The illustration shows how we would mimic the actions of the instructions to find out what this value was.

The command

```
$1a2 ($70) nmdat > dv [sp-#172]
```

Employs indirection ([and]) so that rather than loading the value at SP-#172 we are instead saying take the value at SP-#172 and show me what it points to.

The value that it points to looks like it could be an HPE_STATUS. This would make sense because the call to tm_unlink_gdpd passed in R24 the value at SP-#172 so it would be logical to assume that this is a status variable and on return from the procedure we are checking it to see if the call succeeded.

Note: Most HPE_STATUS values have a the following characteristics:

- The left 16 bit value will be negative “info” value, positive values are warnings but because they are positive they are harder to spot if you are guessing
- The right 16 bits will have a small positive “subsys” value. You can get a list of subsystem values with “syml subsys@,,const”. There will be a few non-subsystems scattered in there but most will be valid MPE/iX subsystem numbers.

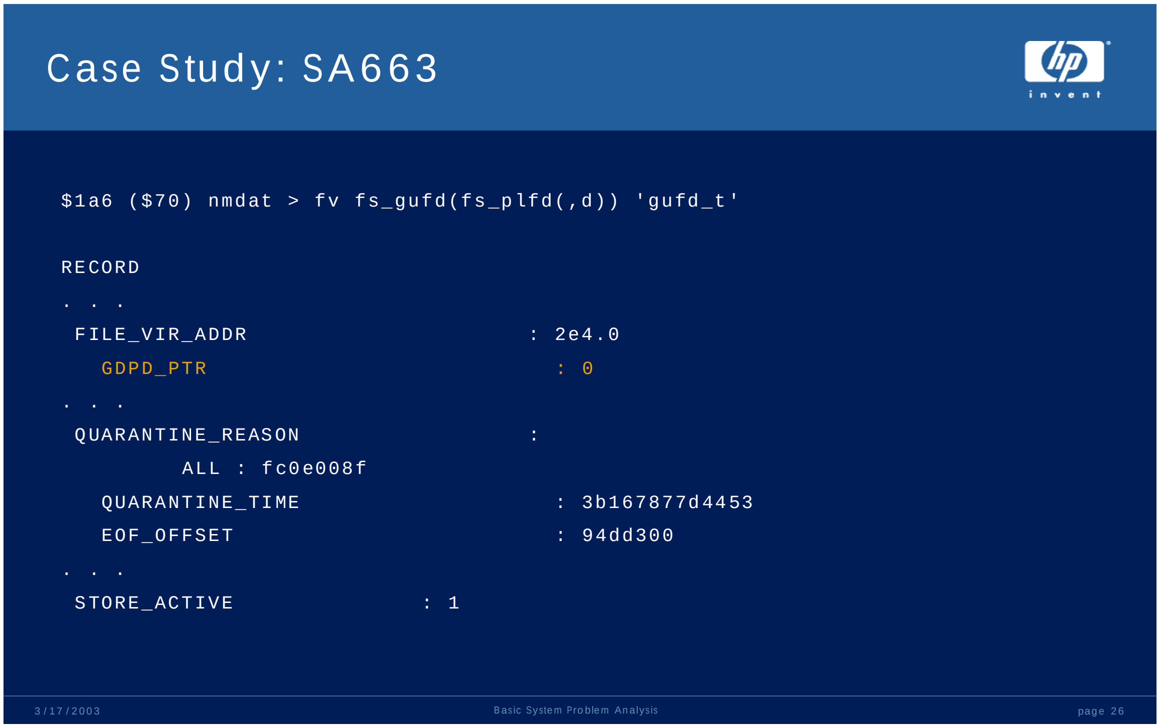
One of the tricks in successful dump reading is being able to identify a something just by they way it looks. For example:

```
ffef00a6
```

That looks like it could be an HPE_STATUS. It has a negative “info” field and the “a6” happens to be a valid subsystem number. Where as

```
00ab9a00
```

Does not look like it could be a valid HPE_STATUS value. The ‘9a00’ is way out of range of subsystem numbers.



Case Study: SA663

```
$1a6 ($70) nmdat > fv fs_gufd(fs_plfd(,d)) 'gufd_t'

RECORD
. . .
 FILE_VIR_ADDR                      : 2e4.0
   GDPD_PTR                           : 0
. . .
 QUARANTINE_REASON                  :
         ALL : fc0e008f
   QUARANTINE_TIME                    : 3b167877d4453
   EOF_OFFSET                         : 94dd300
. . .
 STORE_ACTIVE               : 1
```

3/17/2003 Basic System Problem Analysis page 26

Notes:

Case Study: SA663 continued

Finally we can format the GUFD for file $d and see that it agrees with what was found, the bad status was $fc0e008f.

There are some other interesting things to be seen; the GDPD_PTR is zero. This should be a pointer to the last GDPD in a linked list. Since the call to tm_unlink_gdpd failed we could assume (and it would be correct!) that the failure was due to the fact that this value is null. Something else seems to have either cleared the value or unlinked the GDPD erroneously. It is also possible that the file being closed was never actually linked into the list correctly in the first place.

That's a problem with reading memory dumps, it's relatively simple to find out what happened. Figuring out why or how it happened is far more difficult!

One final bit of potentially interesting information is the fact that the GUFD field STORE_ACTIVE is 1. That seems to imply that a STORE may be running. This is another of those "how can you tell without the source code" problems.

The most direct way of confirming the suspicion would be to find out if STORE is running. That can be done by setting a filter on the string "STORE" and using the PM_PTREE macro (with no input PIN) to scan all processes in the dump:

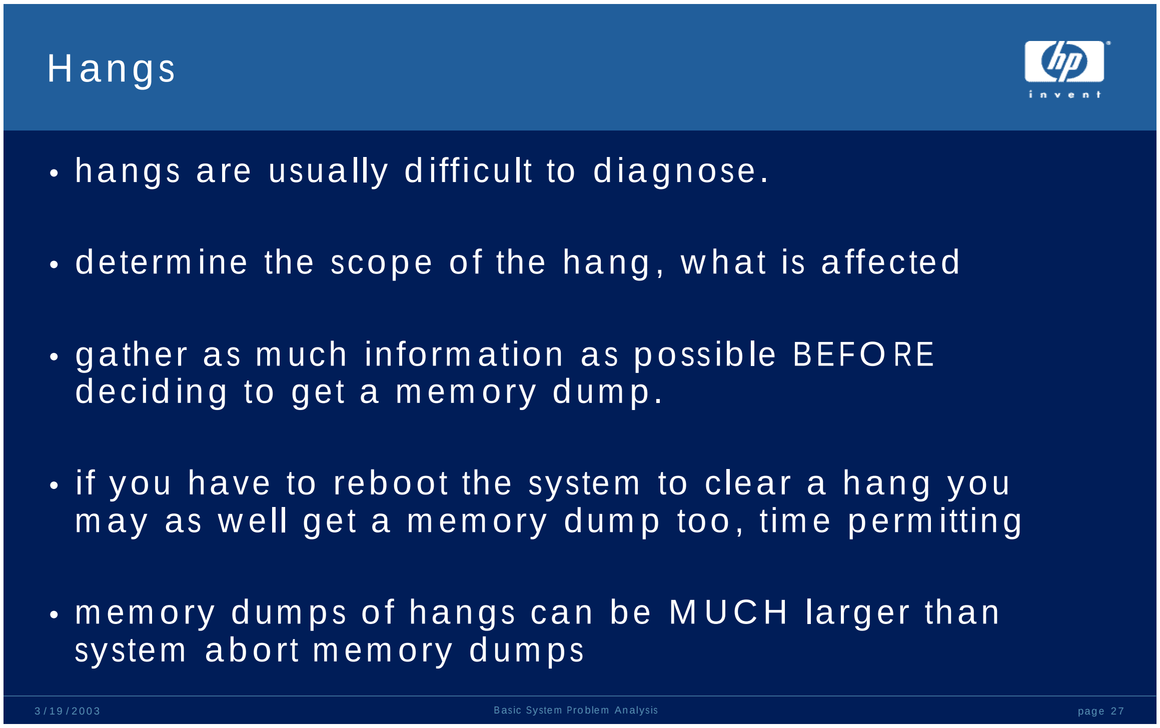
```
$251 ($70) nmdat > env filter 'STORE'
$252 ($70) nmdat STORE> pm_ptree
     $1d7 (STORE.PUB.SYS)  #J2697
      $21d (STORE.PUB.SYS)  #J2697
```

It's running alright. So does that mean STORE did this? No, obviously not but it would be a data point.

If you are reporting this to the Response Center you can supply this information in your initial contact with them. If there are internal reports that indicate there is problem the support engineer may be able to recommend a patch right away. You can also check the ITRC database to see if there are any documents reporting this too.

This information can also be recorded in a failure log for later reference.

Last but certainly not least, since the problem involves a file it would be wise to schedule a time to run FSCHECK. The problem itself is unlikely to be due to physical damage to the file but why take chances.



Hangs

- hangs are usually difficult to diagnose.
- determine the scope of the hang, what is affected
- gather as much information as possible BEFORE deciding to get a memory dump.
- if you have to reboot the system to clear a hang you may as well get a memory dump too, time permitting
- memory dumps of hangs can be MUCH larger than system abort memory dumps

3/19/2003 Basic System Problem Analysis page 27

Notes:

Hangs

Hangs do tend to be more difficult to diagnose than aborts. Often what is called a “hang” is really a performance slow-down. It can often be limited to a particular application or area of the OS.

If the sole function of a system is to run account’s payable and the accounts payable yet anyone trying to do so hangs then it is technically correct that the “system” is hung. But telling that to a support engineer might mislead them badly!

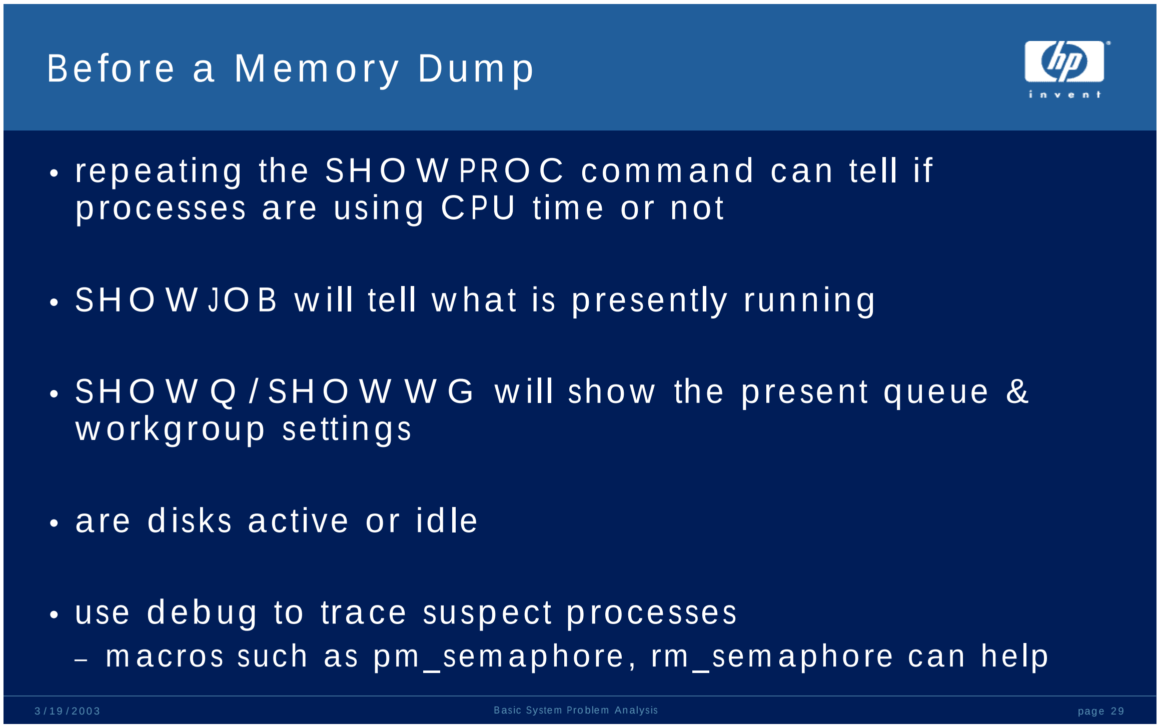
If is important to determine the scope of the problem. Are only certain users affected? Are certain programs or applications affected?

Is it possible to log on the system? And log off? Does a control-A produce an equal (=) sign? If so than the OS is able to respond.

If people are unable to connect to the system how are they connecting, DTC, TELNET, VT, FTP, via the web? Do some connection methods work where others don’t

If the problem is going to require a reboot to clear it then if you haven’t determined the cause and can invest the time you should get a memory dump. Even if all you do is hold on to it, it is better to have it than not have it if the problem appears again.

But remember, memory dumps of hangs can be a log larger than memory dumps of system aborts. They will take longer, especially if you are writing them to tape.



Before a Memory Dump

- repeating the SHOWPROC command can tell if processes are using CPU time or not
- SHOWJOB will tell what is presently running
- SHOWQ / SHOWWG will show the present queue & workgroup settings
- are disks active or idle
- use debug to trace suspect processes
  - macros such as pm_semaphore, rm_semaphore can help

3/19/2003 Basic System Problem Analysis page 29

Notes:

Before Memory Dump

Gather as much information as possible!

If you are able to log on or if a session logged on as MANAGER.SYS is already logged on you should try to gather as much information as possible.

SHOWPROC is extremely useful in cases where the system is not completely hung up but people are complaining of problems. For example,

SHOWPROC PIN=1;TREE;SYSTEM

Will display all processes on the system. Use this to locate processes that may be blocked. You would see either “BLKMM” or “BLKCB” for processes blocked on memory or on a control block.

If you find processes in this state repeat the SHOWPROC command on those specific PIN’s to see if their CPU time increases. It is completely normal for processes to be blocked in this way but if they remain blocked without accumulating CPU time then they may be part of the problem. They would be a good place to begin looking.

You can use DEBUG to trace their stacks too. A useful way to do this would be as follows, say we find PIN 9a blocked on memory and it appears to be using CPU time but anytime you catch it with SHOWPROC it is back in that BLKMM state:

```
$1a ($2d) nmdebug > pin 9a;tr,i,d
```

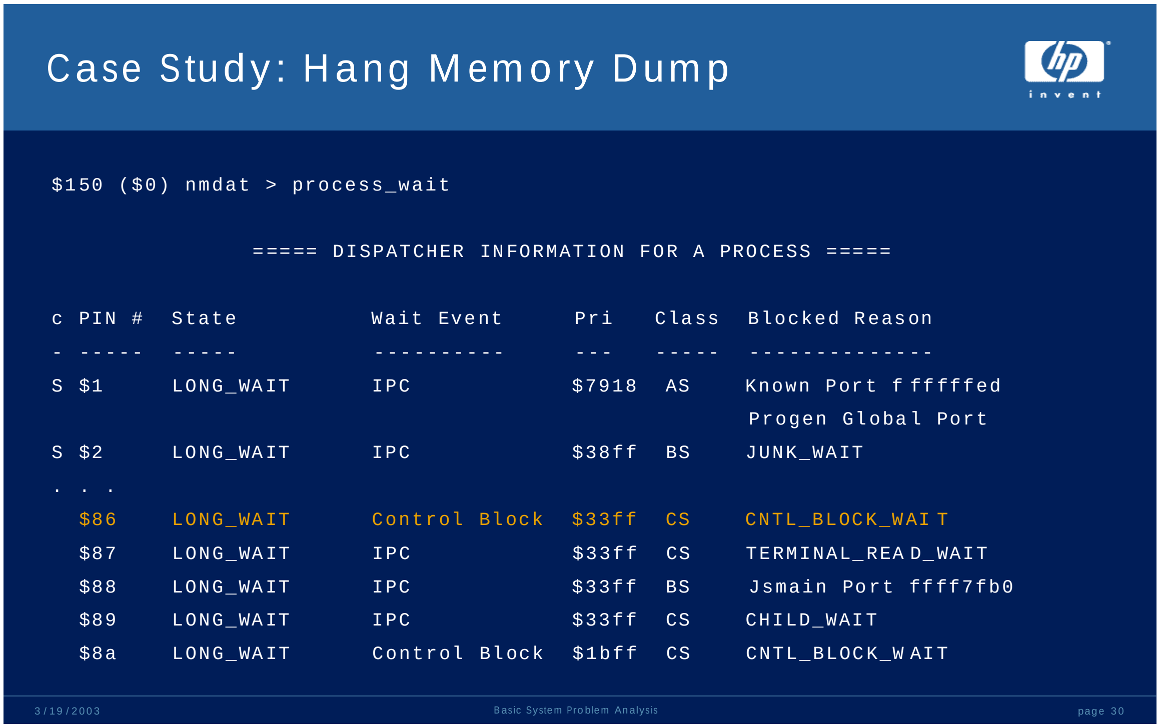
You will note that the pin command and the trace command are submitted together separated by a semicolon. That minimizes the time between the two commands so you area more likely to catch the process in a way that will allow a useful trace to be displayed. Remember, the process is running, it may be slow, but it’s running!

You could also use the “CRON” (carriage return repeats the last command) feature, as in:

```
$1b ($2d) nmdebug > set cron
```

Then all you would need to do is press return to capture a trace.

If the process is using NO CPU time then you would still want to grab a stack trace because you may find that the process is blocked on something which can be fixed without the need of getting a memory dump!



Case Study: Hang Memory Dump

```
$150 ($0) nmdat > process_wait

               ===== DISPATCHER INFORMATION FOR A PROCESS =====

c PIN #  State          Wait Event     Pri   Class  Blocked Reason
- -----  -----          ----------     ---   -----  --------------
S $1     LONG_WAIT      IPC            $7918  AS    Known Port ffffffed
                                                    Progen Global Port
S $2     LONG_WAIT      IPC            $38ff  BS    JUNK_WAIT
. . .
  $86    LONG_WAIT      Control Block  $33ff  CS    CNTL_BLOCK_WAIT
  $87    LONG_WAIT      IPC            $33ff  CS    TERMINAL_READ_WAIT
  $88    LONG_WAIT      IPC            $33ff  BS    Jsmain Port ffff7fb0
  $89    LONG_WAIT      IPC            $33ff  CS    CHILD_WAIT
  $8a    LONG_WAIT      Control Block  $1bff  CS    CNTL_BLOCK_WAIT
```

Notes:

Case Study: Hang Memory Dump

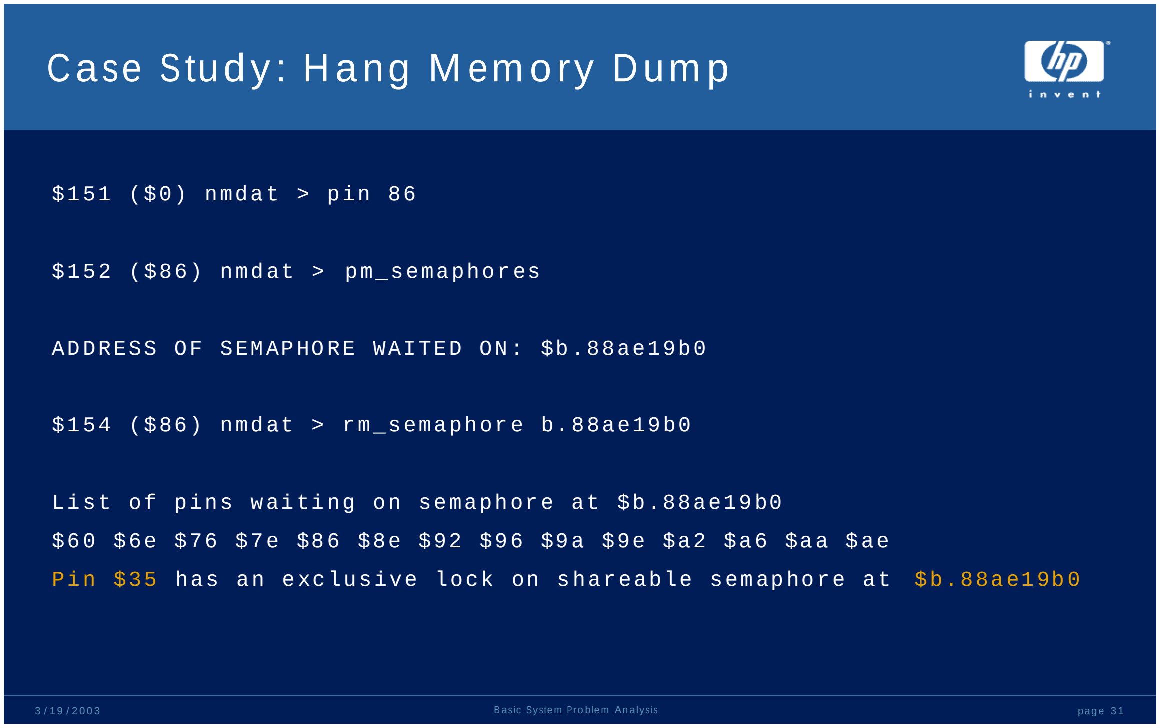
If you have a memory dump of a hang it is not so important to begin looking at stack traces as it is finding “interesting processes”. These would be processes blocked in ways that would not be normal.

Now, without having had the complete MPE/iX internals training and a few years of reading memory dumps, knowing what is “normal” is not quite that simple. For example, “JUNK_WAIT” doesn’t look all that normal but it is. Pin 2 is the CM loader process and that is how it normally waits. Likewise, pin 1 is PROGEN and it waits on a “port”.

Illustrated on the previous page is output from the **PROCESS_WAIT** macro which walks down the list of processes and reports what they are blocked on. This is probably the best macro to employ when beginning to look at a memory dump of a hang. This macro can be used in DEBUG as well, but when the system is running, even slowly you would need to be skeptical of any output because process states could change.

The “CNTRL_BLOCK_WAIT” is definitely an “interesting process” because this indicates that the process has blocked on a semaphore.

The full listing from PROCESS_WAIT actually showed a large number of processes in this state.



Case Study: Hang Memory Dump

```
$151 ($0) nmdat > pin 86

$152 ($86) nmdat > pm_semaphores

ADDRESS OF SEMAPHORE WAITED ON: $b.88ae19b0

$154 ($86) nmdat > rm_semaphore b.88ae19b0

List of pins waiting on semaphore at $b.88ae19b0
$60 $6e $76 $7e $86 $8e $92 $96 $9a $9e $a2 $a6 $aa $ae
Pin $35 has an exclusive lock on shareable semaphore at  $b.88ae19b0
```

3/19/2003 Basic System Problem Analysis page 31

Notes:

Case Study: Hang Memory Dump continued

What we do once we find an interesting process is to switch to that pin and have a look at the trace. That isn’t shown in the illustration on the prior page:

```
$14a ($0) nmdat > pin 35

$150 ($35) nmdat > tr,d,i
       PC=a.0017099c enable_int+$2c
NM* 0) SP=41853ef0 RP=a.00786004
notify_dispatcher.block_current_process+$338
NM  1) SP=41853ef0 RP=a.00787e44 notify_dispatcher+$268
NM  2) SP=41853e70 RP=a.001b6034 sem_block.wait_for_resource+$1bc
NM  3) SP=41853d70 RP=a.001b6428 sem_block+$358
NM  4) SP=41853cb0 RP=a.00757ce8 cb_shr_lock+$240
NM  5) SP=41853bb0 RP=a.00757a94 ?cb_shr_lock+$8
         export stub: fb.011380f8 lock'set'exclusive_345+$230
NM  6) SP=41853a70 RP=fb.0113bc78 nmdbunlock+$16f4
NM  7) SP=41853a30 RP=fb.0109ae70 dblock+$10c
NM  8) SP=418522b0 RP=fb.0109ad38 ?dblock+$8
         export stub: 48f.000060a0
NM  9) SP=418521b0 RP=48f.00000000
     (end of NM stack)
```

The call to “SEM_BLOCK” at level 3 (or 4 if we want to look at the parameters to it!) is what causes the process to block on a semaphore owned by some other process.

Illustrated is the use of the PM_SEMAPHORES macro which is actually an easier way of extracting the address of the semaphore. That address can then be passed to the RM_SEMAPHORE macro which will list the processes waiting on it as well as the owner PIN and format it for you (not shown in the illustration).

Case Study: Hang Memory Dump continued

```
$152 ($86) nmdat > pm_semaphores

ADDRESS OF SEMAPHORE WAITED ON: $b.88ae19b0

$154 ($86) nmdat > rm_semaphore b.88ae19b0
List of pins waiting on semaphore at $b.88ae19b0
$60 $6e $76 $7e $86 $8e $92 $96 $9a $9e $a2 $a6 $aa $ae

Pin $35 has an exclusive lock on shareable semaphore at
$b.88ae19b0

RECORD
   SEM_INFO_WORD :
      SEM_STATE : 2
      SEM_LOCK  : 1
      SEM_SPEC  : 4
      SEM_CLASS : 39
         SEM_OWNER : 35
      SEM_OWNER_COUNT : 1
      SEM_WAIT_COUNT  : e
      SEM_HEAD_WAITER : d3818280
      SEM_TAIL_WAITER : d382ba80
END
```

The “SEM_HEAD_WAITER” and “SEM_TAIL_WAITER” fields are actually PIB pointers formatted using the type PIB_TYPE.

The RM_SEMPHORE macro is a lot easier as it displays the list of pins that are currently blocked waiting for this semaphore.

If you had wanted to manually get the address of the semaphore then you would do it this way:

```
$166 ($35) nmdat > lev 4

$167 ($35) nmdat > dv psp-28,2
VIRT $480.41853b88 $ 0000000b 88ae18d0
```

The semaphore address is passed, as a long pointer to SEM_BLOCK so it will be found at PSP-28 and PSP-24.

This is another of those “how can I know this without the source code” problems. So that is why I am sharing it here J

Case Study: Hang Memory Dump continued

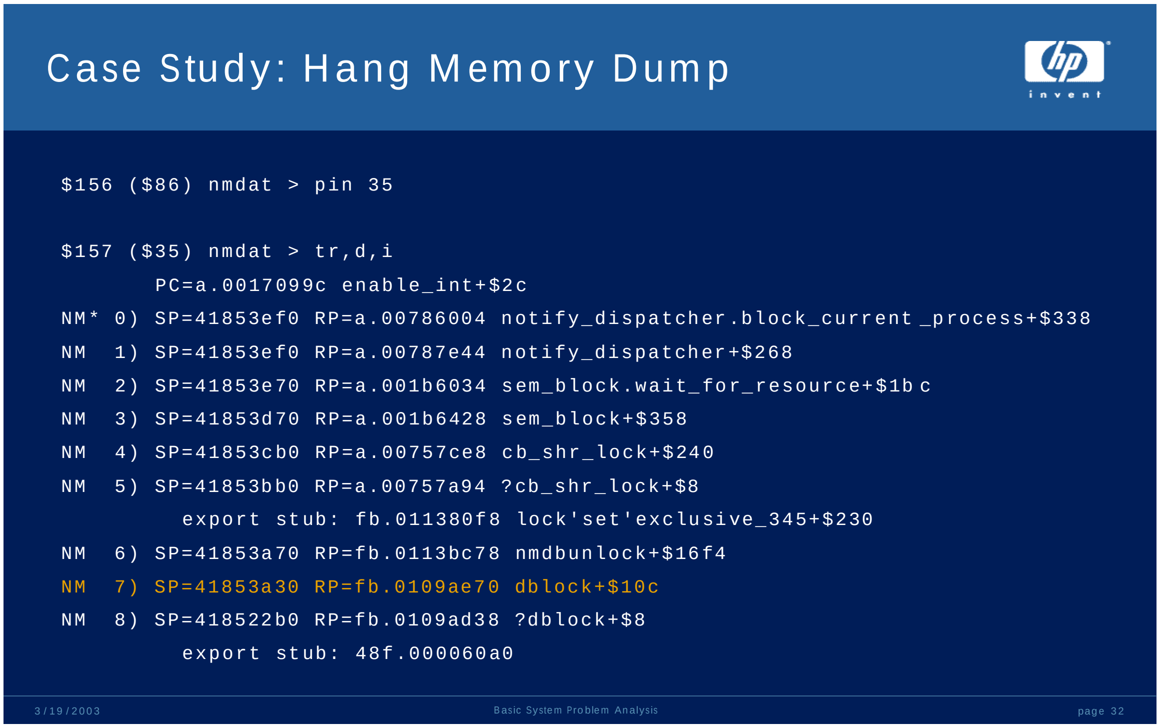
Now that you have the address of the semaphore you could, for example, use the function VAINFO to get the BASE_VA (or address) of that semaphore. It is very likely to be a part of some larger structure. VAINFO could also be used to tell you the OBJ_CLASS (object class) of the structure it is in. These would be useful data points.

```
$169 ($35) nmdat > wl vainfo(b.88ae18d0, 'BASE_VA')
$b.88ae0000

$16a ($35) nmdat > wl vainfo(b.88ae18d0, 'OBJECT_CLASS')
$16e
```

With the object class you can set a filter on the value, $16e and then use the SYMLIST or SYML command to see what that may be. Object class constants in the OS all begin with the string OBJCL

```
$16b ($35) nmdat > env filter '16e'

$16c ($35) nmdat 16e> syml objcl@,,const
OBJCL_TURBO_GLOBAL_CB            CONST   INTEGER      $16e
```

So the semaphore is in an object whose class is OBJCL_TURBO_GLOBAL_CB.



Case Study: Hang Memory Dump

```
$156 ($86) nmdat > pin 35

$157 ($35) nmdat > tr,d,i
       PC=a.0017099c enable_int+$2c
NM* 0) SP=41853ef0 RP=a.00786004 notify_dispatcher.block_current_process+$338
NM  1) SP=41853ef0 RP=a.00787e44 notify_dispatcher+$268
NM  2) SP=41853e70 RP=a.001b6034 sem_block.wait_for_resource+$1bc
NM  3) SP=41853d70 RP=a.001b6428 sem_block+$358
NM  4) SP=41853cb0 RP=a.00757ce8 cb_shr_lock+$240
NM  5) SP=41853bb0 RP=a.00757a94 ?cb_shr_lock+$8
         export stub: fb.011380f8 lock'set'exclusive_345+$230
NM  6) SP=41853a70 RP=fb.0113bc78 nmdbunlock+$16f4
NM  7) SP=41853a30 RP=fb.0109ae70 dblock+$10c
NM  8) SP=418522b0 RP=fb.0109ad38 ?dblock+$8
         export stub: 48f.000060a0
```

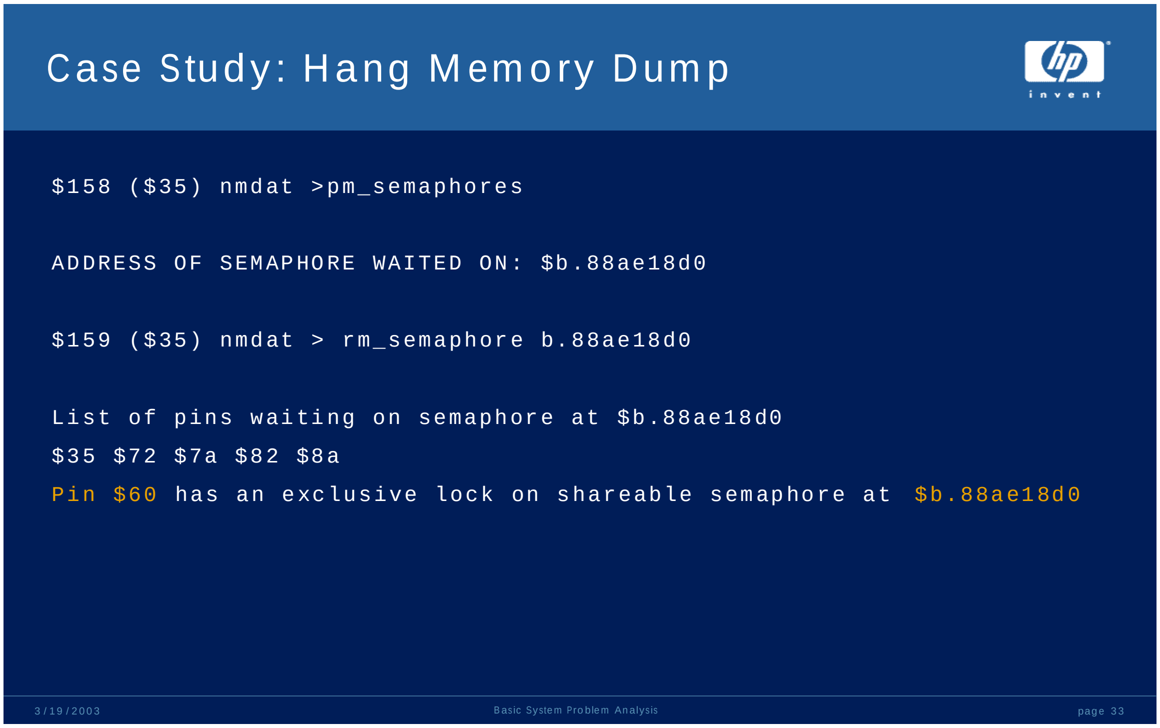
Notes:

Case Study: Hang Memory Dump continued

At this point we know that Pin 86 was blocked on a semaphore owned by Pin 35. We want to go look at what Pin 35 is doing and we find that this process has called DBLOCK and is also blocked. Note that it also has “SEM_BLOCK” in its stack trace. You only see this when a process blocks on a semaphore.

We need to see what semaphore this process is waiting on.

In terms of red flags, this is a great big banner sized flag, a process owning a semaphore is also blocked on one! Not a good sign at all.



Case Study: Hang Memory Dump

```
$158 ($35) nmdat >pm_semaphores

ADDRESS OF SEMAPHORE WAITED ON: $b.88ae18d0

$159 ($35) nmdat > rm_semaphore b.88ae18d0

List of pins waiting on semaphore at $b.88ae18d0
$35 $72 $7a $82 $8a
Pin $60 has an exclusive lock on shareable semaphore at  $b.88ae18d0
```

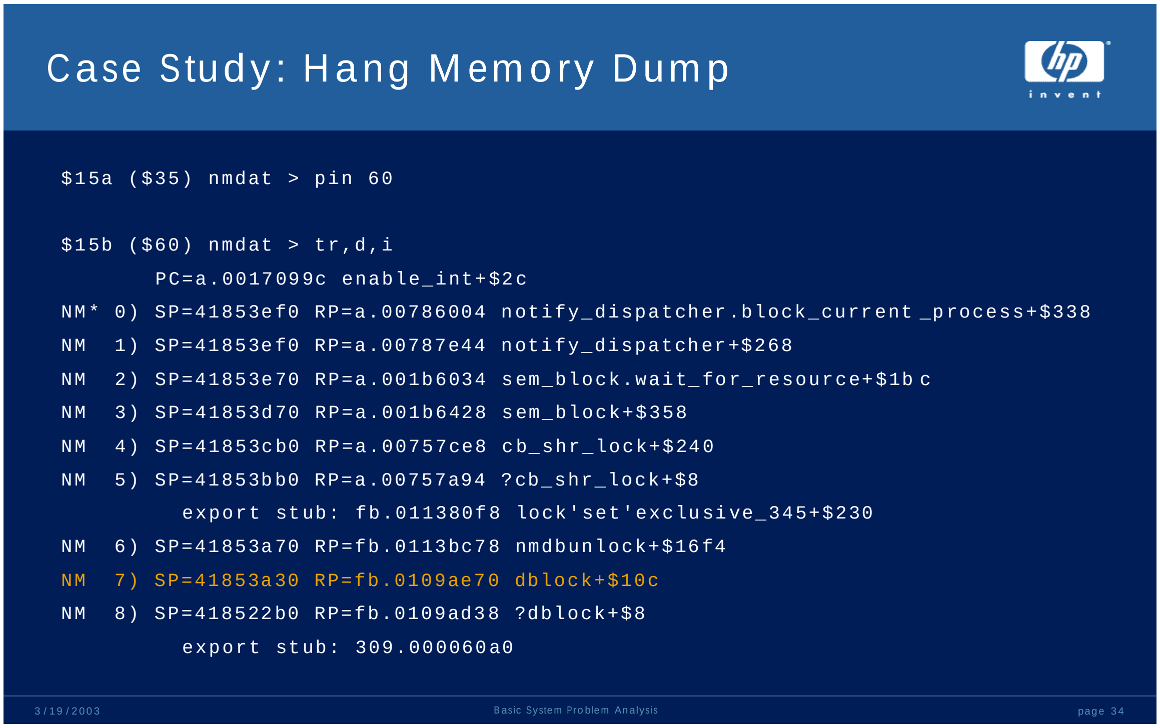
3/19/2003 Basic System Problem Analysis page 33

Notes:

Case Study: Hang Memory Dump

This is exactly what was done with Pin 86. Here we see that pin 60 owns the semaphore that pin 35 is blocked on.

So let’s go to pin 60…



Case Study: Hang Memory Dump

```
$15a ($35) nmdat > pin 60

$15b ($60) nmdat > tr,d,i
       PC=a.0017099c enable_int+$2c
NM* 0) SP=41853ef0 RP=a.00786004 notify_dispatcher.block_current_process+$338
NM  1) SP=41853ef0 RP=a.00787e44 notify_dispatcher+$268
NM  2) SP=41853e70 RP=a.001b6034 sem_block.wait_for_resource+$1bc
NM  3) SP=41853d70 RP=a.001b6428 sem_block+$358
NM  4) SP=41853cb0 RP=a.00757ce8 cb_shr_lock+$240
NM  5) SP=41853bb0 RP=a.00757a94 ?cb_shr_lock+$8
         export stub: fb.011380f8 lock'set'exclusive_345+$230
NM  6) SP=41853a70 RP=fb.0113bc78 nmdbunlock+$16f4
NM  7) SP=41853a30 RP=fb.0109ae70 dblock+$10c
NM  8) SP=418522b0 RP=fb.0109ad38 ?dblock+$8
         export stub: 309.000060a0
```

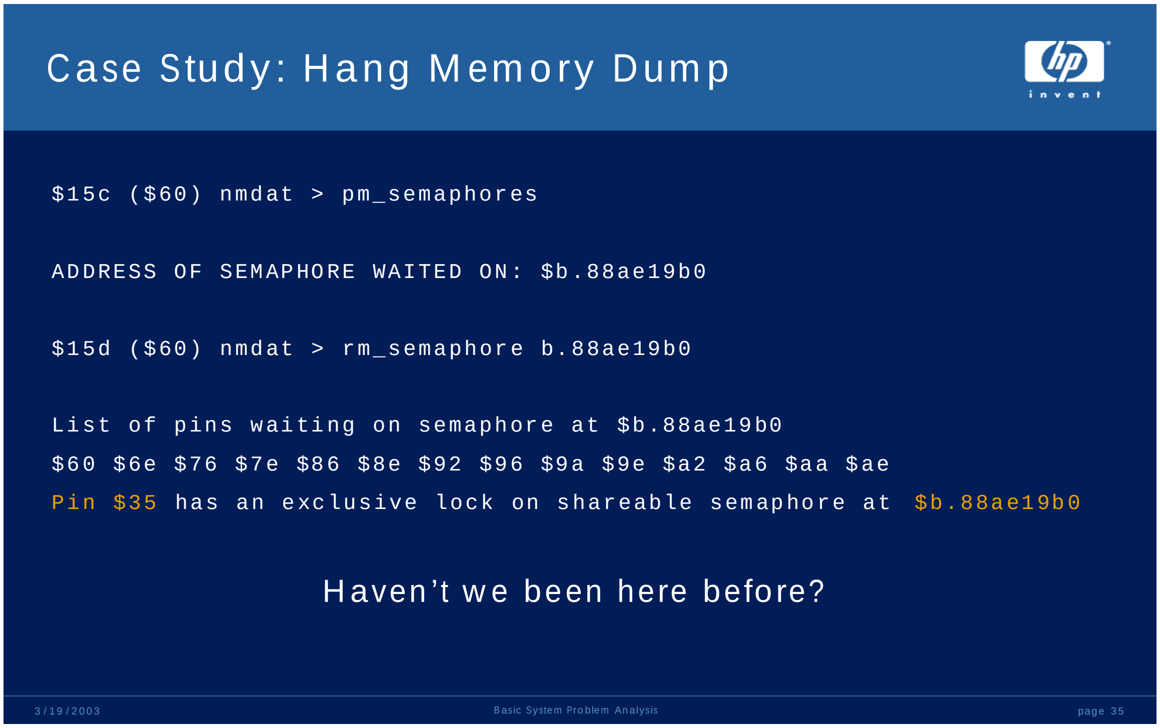
Notes:

Case Study: Hang Memory Dump continued

We seem to have a pattern developing here…

Pin 60 owns a semaphore but it also has both a DBLOCK call and a SEM_BLOCK call in its stack. So it is also waiting on a semaphore while holding one just like pin 35.

We will use the same two macros to look at what pin 60 is waiting for.



Case Study: Hang Memory Dump

```
$15c ($60) nmdat > pm_semaphores

ADDRESS OF SEMAPHORE WAITED ON: $b.88ae19b0

$15d ($60) nmdat > rm_semaphore b.88ae19b0

List of pins waiting on semaphore at $b.88ae19b0
$60 $6e $76 $7e $86 $8e $92 $96 $9a $9e $a2 $a6 $aa $ae
Pin $35 has an exclusive lock on shareable semaphore at $b.88ae19b0
```

Haven’t we been here before?

Notes:

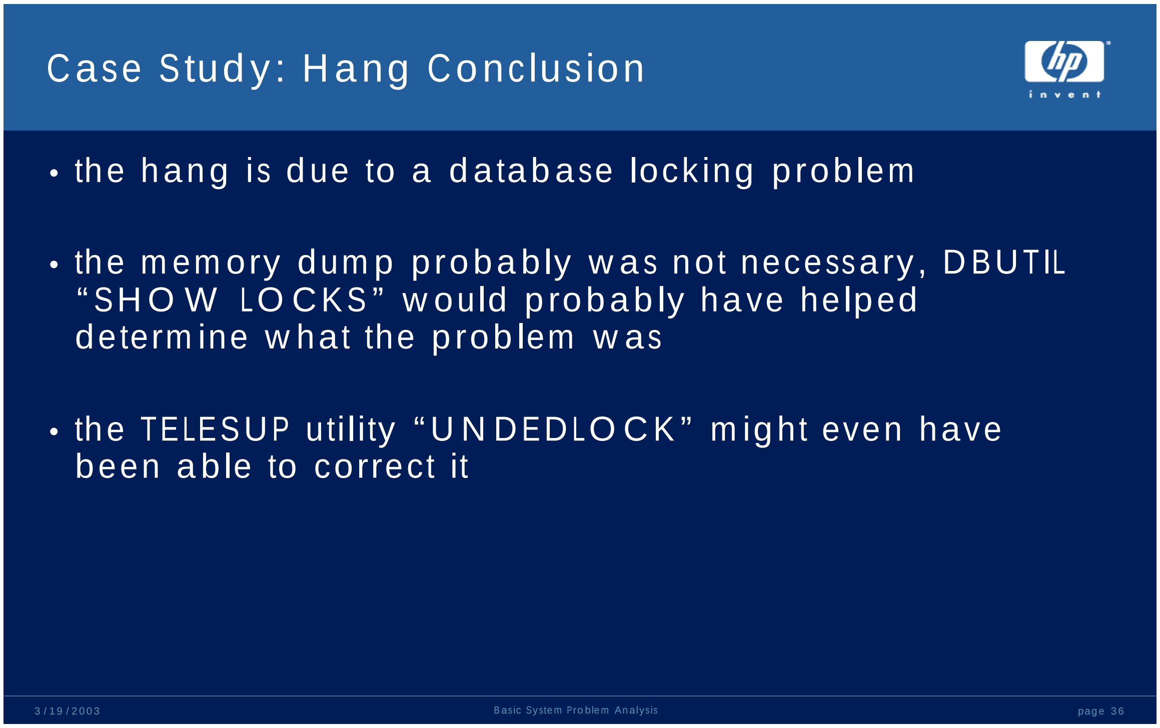
Case Study: Hang Memory Dump continued

Pin 60 is waiting on the semaphore that pin 35 holds. Pin 35 is waiting on the semaphore that pin 60 holds. The classic deadly embrace.

The macro RM_SEM_DEADLOCK would actually have been a much better choice here, as it would have detected this and displayed the two pins involved:

```
$161 ($0) nmdat > rm_sem_deadlock

**********************
* Deadlock detected. *
**********************

Suspected PINs:  $$35 $60  $$60 $35
```

The output “$$35” and “$$60” indicate the holders of semaphores and “$35” and “$60” are the pins waiting. The problem is obvious!



Case Study: Hang Conclusion

- the hang is due to a database locking problem
- the memory dump probably was not necessary, DBUTIL "SHOW LOCKS" would probably have helped determine what the problem was
- the TELESUP utility "UNDEDLOCK" might even have been able to correct it

3/19/2003 Basic System Problem Analysis page 36

Notes:

Case Study: Hang Conclusion

This is pretty obviously an application problem. Two programs are locking datasets in a database in the opposite order.

It is also very likely that some far less drastic measure could have been taken to diagnose this short of taking the system down and dumping it.

Unfortunately most system hang's are not as easy and obvious as this one was to diagnose. Any information that can be gather beforehand will help.