TECHNOTE 1094

Virtual Memory Application Compatibility

By Quinn "The Eskimo!"
Apple Developer Technical Support
devsupport@apple.com

Although Virtual Memory (VM) has been available on the Mac OS since the release of System 7, its use has become more widespread in recent years. This is because of a combination of factors, including

- the introduction of the Power Macintosh, on which VM radically reduces application memory usage
- the popularity of third-party products, such as RAM Doubler, which behave in a manner similar to that of standard virtual memory
- the release of System 7.5.5, which significantly improved the performance and reliability of the VM implementation
- the release of Mac OS 7.6, which file maps CFM-68K PEF containers to reduce memory usage just like a Power Macintosh.

Because VM is so widely used, it's critically important to make sure that your software is compatible with it. This Technote, which is divided into two parts, provides you with

1. an introduction to how VM works, both in theory and its specific implementation under System 7
2. a detailed explanation of the steps you need to follow in order to ensure that your software executes successfully under VM.

All Mac OS programmers should read this Note. For application developers, this Note supplants and extends the information presented in Technote ME 09, "Coping With VM and Memory Mappings". Device driver writers should continue to refer to Technote ME 09.

Note

This note uses the term **application** to denote both

- standard Mac OS applications
- other code resources that run in the application environment, e.g.`'CDEF'`s, `'WDEF'`s, control panels, and so on.

As far as VM is concerned, the important distinction is between application code and device drivers.

Contents

VM Compatibility Checklist

In general, programs should be able to ignore VM and operate successfully. Specific problems may occur under the following conditions:

- Does your program call SCSI or ATA Managers directly?
- Does it use hardware interrupt handlers?
- Does it use physical (as opposed to logical) addresses?
- Does it contain an AppleTalk socket listener?
- Does it call system services that are not interrupt-safe while paging is not safe?

If your program does none of the operations listed, it should already be VM-compatible. If you want to improve your code's performance under VM, you should read the section Ways to Improve Performance in this Note.

On the other hand, if your program does any of these operations, you'll want to read the next section, Programming Implications, which explains how to make your program VM-compatible. That section is also a good place to start if your program is incompatible with VM and all you want to do is fix it as quickly as possible.

For an overview of VM theory, terminology, and implementation under System 7, you'll want to go to the How VM Works section.

Programming Implications

If your program does any of the things listed above, you should read the next few sections for information on how to execute safely under VM.

Direct SCSI Manager or ATA Manager Calls

Some applications, such as scanning programs or hard disk formatters, need to call the SCSI or ATA Managers directly. These applications must take special care when VM is enabled. For example, if such an application was to grab exclusive access to the SCSI bus and then take a page fault, VM would be unable to swap in the new page, making the page fault fatal.

The solution is simple: if your program calls the SCSI or ATA Manager directly, you must make sure that any code executed or data accessed while paging is unsafe is held in memory.

If you're writing an application, meeting that requirement can be tricky because it's hard to hold just part of an application's code in memory. For this reason, your application might want to implement this functionality as a bundled device driver or other code resource. This separation makes it easy for the application to ensure that the code running while paging is unsafe is held.

Hardware Interrupt Handlers and Physical Addressing

Some applications need to drive hardware devices directly; for example, a data capture program that talks to a dedicated PCI card. Such applications typically install hardware interrupt handlers (either using `SIntInstall`, the native Interrupt Manager on PCI machines, or the low memory interrupt dispatch tables) and program a DMA controller using physical addresses. Because this sort of operation is typically the realm of device drivers, the steps you take to operate under VM are specified in device driver documentation. The following references are useful:

1. IM:Memory, Virtual Memory Manager chapter
2. Technote ME 09 - "Coping With VM and Memory Mappings"
3. Designing PCI Cards and Drivers for Power Macintosh Computers

The basic rules for device driver writers are:

1. If you install a hardware interrupt handler, you must ensure that all code and data it accesses is held resident in memory.
2. If you have a hardware device that uses physical addressing (typically a DMA peripheral), you must translate the logical addresses you are accessing into physical addresses before passing them to the hardware.

For further explanation, see the above references.

As an aside, in future versions of the Mac OS it is likely that these functions will be privileged and inaccessible from applications. If your application does this sort of thing, it is most probably a good idea to prepare for the future by factoring your code into an application and a bundled device driver.

AppleTalk Socket Listeners

AppleTalk socket listeners are also not VM-safe. An AppleTalk socket listener on a slow computer using LocalTalk networking has very difficult real-time goals. Deferring this while waiting for a page fault would cause serious packet loss. So, for performance reasons, VM does not defer socket listeners until paging is safe. Socket listeners must be written so that they don't cause a page fault.

The way to make your socket listener VM-safe is to ensure that all code that can be called by the socket listener, and the data it accesses, is either held in memory or deferred using `DeferUserFn`. This can be tricky to do from an application. Technote NW 13 "AppleTalk: The Rest of the Story" demonstrates a technique that you can use.

Calling Non-Interrupt Safe Routines

If your code calls routines that are not interrupt-safe while handling a non-deferred hardware interrupt (or, for that matter, any time paging is unsafe), you may encounter a problem running on Mac OS 7.6 and future releases of the Mac OS that use System 7-style VM.

In older Mac OS implementations, large parts of system software were in the system heap simply so that they could be shared between applications. These parts caused the system heap to grow in size. Because the system heap is always held, these parts were resident, even though they didn't need to be. This reduced the number of physical pages of RAM that VM had available to use as a cache for its various logical address ranges. This reduction made the system slower.

Under versions of the Mac OS that use System 7-style VM, these system parts are being moved out of the system heap and into file mapped CFM containers. This makes them available for paging, and increases the number of physical pages available to VM. In general, this helps system performance.

Such parts of the system are only made pageable if none of their routines can be called at interrupt time. Otherwise, a non-deferred hardware interrupt might call these routines and cause a fatal page fault.

However, some programs call non-interrupt safe routines when paging isn't safe. This works under earlier versions of Mac OS because the code for these routines was in the system heap, and hence resident. Such software has problems under newer versions of Mac OS when VM is turned on because these routines are no longer held. If your software was previously compatible with VM and broke under Mac OS 7.6, you should check to be sure that you are not calling non-interrupt safe routines when paging is unsafe.

Ways to Improve Performance

This section describes a number of things that you can do to make your application work better under VM.

Grouping Commonly Used Code and Data

Perhaps the best thing you can do to make your application work better under VM is to analyze its working set. The working set of a program is the set of memory pages that the program accesses most often. The smaller you make your working set, the better your program will run under VM. If the working set of all the active processes exceeds the amount of physical memory available for paging, the system begins to take an excessive number of page faults. This state is known as thrashing.

Under System 7 there are no good tools for analyzing your application's working set automatically. However, you can do some things to empirically adjust your working set. One good thing is to 'segment' your CFM-based applications sensibly.

Most development environments provide a mechanism for CFM-based applications to sort routines in their PEF container according to a "group" that is set using compiler directives. This is analogous to the classic runtime segmentation model. You can use these directives to group rarely used functions together and away from the commonly used functions. This helps keep rarely used code paged out, which reduces your working set.

Sensible Memory Management

Another good place to look when analyzing your working set is your memory management system. Some memory management systems are VM-friendly, and some are not. For example, if your memory management system looks at every block in the heap when a block is freed, it has pathologically bad VM performance.

On PowerPC machines, the system Memory Manager (also known as the Modern Memory Manager) has been optimized to be as VM-friendly as possible. It's important that you be compatible with the Modern Memory Manager, for this and other reasons.

On 680x0 machines, the Memory Manager has some behaviors that cause excessive page faults under VM. Unfortunately, there's not a lot you can do about the system Memory Manager on these machines other than avoid using it.

If you're using your own memory management scheme (for C++ objects, for example), you should look at its implementation to determine whether it's VM-friendly. A VM-friendly memory manager attempts to reduce the number of times it looks at bytes located in different pages, and thus minimizes the program's working set. You may be able to switch memory managers and get VM performance benefits.

Another way to avoid thrashing is to minimize your use of Process Manager temporary memory. While it may appear that this memory is free for application use, under VM this memory has often been paged out, and therefore is not 'real'. So if you look at temporary memory and see that there's a huge amount of free space, do not allocate and use it all. This will likely cause the system to thrash. Only allocate as much temporary memory as you actually need.

Judicious Use of `HoldMemory`

The `HoldMemory` call causes a certain range of logical memory to be marked as ineligible for paging. The call was designed to allow system and user programs to avoid fatal page faults by forcing critical chunks of code and data to be held in real memory.

However, sometimes the `HoldMemory` call is useful for applications to allow them to meet real-time goals. For example, it's possible for high quality sound playback to break up when playing on a machine running VM. One way to avoid this is to hold the buffer that contains the sound, to avoid it being paged out to disk.

Application programs should not make arbitrary decisions to hold large chunks of memory. You should only resort to holding memory if you have an identifiable problem under VM. Remember that the Virtual Memory Manager acts like a cache. Every time you hold a chunk of memory, that memory becomes unavailable for paging for the rest of the system, effectively decreasing the size of the cache. If you do this without due care and attention, you cause the system to thrash.

Emulated Instructions

When you execute a privileged 680x0 instruction, the Virtual Memory Manager emulates that instruction to prevent a privilege violation exception. This emulation takes time. You should avoid executing privileged 680x0 instructions where possible. This list of these instructions is given in the section Privileged Instruction Emulation.

One common use of emulated instructions is disabling interrupts to ensure atomicity. In a lot of cases, you can avoid this by using the processor's built-in synchronization primitives. This includes the `CAS` instruction on the 68020 and higher, and the `lwarx` and `stwcx` instructions on the PowerPC. Some system services, specifically Open Transport's `OTLIFO` lists and `OTGate`, are based on top of these instructions and provide for fast atomic operations without using any privileged instructions or Mixed Mode switches.

Note

The `CAS` instruction is not available on the 68000 processor. However, as VM does not work on computers with that processor, this should not be a problem: you can just use privileged instructions without any speed penalty.

Important

You should not use the `TAS` instruction as a replacement for `CAS` because it is not supported on certain Mac OS computers.

VM and Interrupt Latency

If interrupt latency is important to you, VM presents further challenges. Because VM defers user code until page faults are no longer fatal, it can increase the latency between when the interrupt is signalled and when the user code executes.

There are some things you can do to reduce specific interrupt latency problems when running under VM:

- The Time Manager has a special "back door" that allows you to install Time Manager tasks that bypass the safety net provided by VM. Technote 1063 "Inside Macintosh: Processes: Time Manager Addenda" documents this technique.
- Device Drivers that are marked as VM immune are ignored by the Virtual Memory Manager. As such, the Virtual Memory Manager does not defer the completion routines issued by these device drivers. If you have your own private device driver, and you can guarantee that all the driver's clients are prepared to be called at times when paging is not safe, you can set this bit in your device driver and decrease the interrupt latency to your clients. Technote NW 13 "AppleTalk: The Rest of the Story" describes this technique.

Both techniques employ the same basic idea, which is that you deliberately take some of your user code and turn it into non-user code to avoid VM's latency.

Some VM Weirdnesses

This section describes a few little things about VM -- its weirdnesses -- that you should know about.

System Heap is Held

Under Apple's current VM implementation, the entire system heap is automatically held resident in physical memory. This is done to provide compatibility with old device drivers that would not otherwise be compatible with VM. There may be future Apple VM implementations (or current non-Apple VM implementations) that do not hold the entire system heap. Thus, you should not rely on this safety net. If you want something to be held, you should explicitly hold it.

Holding Memory at Interrupt Time

The VM API routines for holding memory resident (`HoldMemory`, `LockMemory`, `LockMemoryForOutput`) are safe to call at interrupt time if one of the following conditions is met:

- paging is safe, or
- paging is unsafe but the range of pages passed to `HoldMemory` is already held, or
- paging is unsafe but the range of pages passed to `LockMemory` is already locked.

These routines can cause paging activity as VM swaps in the memory that is to be held. Obviously, this is bad when paging is unsafe. So you must either ensure that paging is safe or that the call won't cause any paging activity.

Holding Memory with Interrupts Disabled

On computers running system software prior to System 7.1.2, VM specifically prohibited all routines based on the `_MemoryDispatch` trap (the most important being `HoldMemory`) from being called at interrupt time. This restriction has now been lifted.

The typical reason for calling `HoldMemory` with interrupts disabled is that you are calling it at interrupt time. If so, you must make sure you follow the rules in the previous section.

Open and Close Asynchronous

Technote ME 09,"Coping With VM and Memory Mappings" says:

> As it turns out, when you make an asynchronous `_Open` or `_Close` call to a device driver, any completion routine you supply is never called. Since Virtual Memory patches `_Open` and `_Close`, and generates an entry for the completion routine in the user function queue, the implication is that the user functions are never executed and the queue may simply fill up.

This statement is erroneous in one respect. VM does not patch `_Open` or `_Close`, so calling these routines asynchronously will not mess up the VM patches. However, for other reasons, you should not open or close drivers using asynchronous calls. If you are acting on a device driver, you should always use the high-level glue routines `OpenDriver` and `CloseDriver`.

What Form of Address to Pass?

All of the VM API routines (`HoldMemory`, `LockMemory`, etc) work as expected in either 24-bit mode or 32-bit mode. In 24-bit mode, for instance, master pointer flags or other garbage in the high-order eight bits are ignored and assumed to be zero. When switching between 24-bit and 32-bit modes, remember to use `_StripAddress` as outlined in Technote ME 06 "_StripAddress: The Untold Story".

Note

Mac OS 7.6 and beyond do not support 24-bit mode. PowerPC-based Mac OS computers do not support 24-bit mode.

Allocating Memory Above BufPtr at Startup Time

Inside Macintosh, Volume IV describes, on page 257, a method of static allocation for drivers or other data structures that has been popular with a number of developers:

> Static allocation off the address contained in the global variable `BufPtr` is useful when a large amount of space is needed which will never be deallocated (once space is allocated, it may not be deallocated unless no one has allocated space below). An 'INIT' resource may obtain permanent space by moving `BufPtr` down, but no further than the location of the boot blocks (`MemTop`/2 + 1KB).

The main thing to remember about allocating memory above `BufPtr` is that memory allocated in this way is not held in physical memory by default. If you require that this memory be resident, you must be explicitly hold it yourself. This is in constrast with memory in the system heap, which is automatically held in the current VM implementation.

In addition, when allocating memory above `BufPtr`, always use the equation given above. The actual configuration of memory at boot time is much more complicated than the illustration in *Inside Macintosh IV* indicates, especially with System 7 and VM. The System 7 boot code passes a specially-conditioned version of `MemTop` to system extensions, which guarantees that the equation has valid results.

Due to the way memory is organized with VM in 24-bit addressing, you may not be able to achieve nearly as much memory above `BufPtr` as you would think possible for a given virtual memory size. This is due to the possibility of VM fragmentation. Without VM, the available space above `BufPtr` is generally somewhat less than half the amount of memory installed in the machine. With 24-bit VM, the available space may be significantly less, and is probably far less than one half of the virtual memory size. The conditioning of the `MemTop` variable takes this into account.

Compatibility with Accelerator Upgrades

The burden of compatibility has long been on the shoulders of accelerator manufacturers. VM may present some additional compatibility challenges for these manufacturers.

Virtual Memory requires services which are not present in the ROMs of 68000-based machines, so VM is not supported on the Macintosh SE, even one with a 68030 accelerator. The same is true of the Macintosh Plus, the Macintosh Classic, Macintosh Portable and PowerBook 100. Apple's VM only works on machines which Apple intended to include MMUs.

Virtual Memory depends on low-memory globals to indicate the presence of a memory management unit at a very early stage of the boot process. If the hardware features of an accelerator are significantly different from those of the stock Macintosh computer, the low-memory globals are not properly set by the boot code in ROM. The most likely problems are exhibited by 68000 Macintosh computers, 68020 Macintosh computers with 68030 accelerators, and Macintosh computers with 68040 accelerators.

It's not that VM does not work with any accelerators, but rather that System 7-style VM is not guaranteed to support third-party accelerators.

VM and Passwords

Some developers have expressed concern about the safety of passwords under VM. For secure environments, any possibility of a password being copied to disk is unacceptable. Under VM there is such a possibility. For example, imagine the following sequence:

1. program asks for password
2. program does authentication
3. program clears out password in memory

If VM pages out the password buffer during step 2 and the system crashes sometime after step 3 but before VM manages to page out the (now cleared) password buffer, a copy of the password will remain on disk in the backing store.

The solution is simple: the program should hold the password buffer before step 1 and unhold it after step 3. Holding the password buffer ensures that VM never pages it out while it contains the password.

Things to Do in MacsBug When You're Dead

When, after reading through this Note and checking that you follow all the rules given here, you still find that your software crashes when VM is enabled, there are things you can do to determine what went wrong.

The most common symptom of failure under VM is that you drop into MacsBug with a bus error because VM was unable to satisfy a page fault request. Remember that VM is hooked into the bus error handler and propagates any bus errors that it's unable to handle -- either because it's not in a section of memory that's under VM control, or because it's already handling a page fault, or for any other reason -- to the bus error handler pointed to by the low memory pseudo-vector.

There are a number of things you can do to recognize and analyze this situation listed in the following sections.

Recognizing a Fatal Page Fault

The most obvious symptom of a fatal page fault is that you end up in MacsBug with a bus error. Unfortunately bus errors have more than one cause. You can a bus error because of a fatal page fault, but the most common cause for bus errors is dereferencing a bogus pointer.

You can quickly see whether this bus error is a possible fatal page fault by checking whether paging is safe. If you look on the left side of the MacsBug display, you will see a two character code that describes the state of VM. If this code is "RM" (Real Memory), VM is not running. If this state is "VM", VM is running but paging is safe. If the state is "vM", VM is running and paging is not safe. If paging is safe, the bus error is not a fatal page fault, and you should look elsewhere for the cause.

Of course not all bus errors that happen when paging is unsafe are fatal page faults. It takes a bit more work to determine this. For example, imagine dropping into MacsBug with the following message:

```
Bus Error accessing 00123456 at PC 4080BB8C
4080BB8C     MOVE.W     $0010(A0),D0
```

MacsBug reports that the instruction that caused the bus error is at address $4080BB8C, but this report is not always correct. The dynamic recompiling (DR) emulator found on recent PowerPC computers can cause an imprecise PC address to be reported by MacsBug. However, one piece of information that you can rely on is the address that was being accessed and caused the page fault. In the above example, this is address $00123456. You can find out information about this address using the following MacsBug command:

```
dm $00123456
wh $00123456
```

If this command indicates that the address is in valid memory (ie either the primary address range or one of the file mapped address ranges), the access should have succeeded. The only reason for this bus error is a fatal page fault.

If the "wh" command reports that the address is "not in RAM or ROM", chances are that this bus error is just a normal bus error, ie one caused by dereferencing a bogus pointer.

Another common form of fatal page fault is reported as:

```
Bus Error at 4080BB8C
while writing long work (data = 0000009B) to 00123456
4080BB8C     LINK     A6,#$FFCC
```

The LINK instruction is touching the stack, which may have been paged out. You can check this by looking at the value of SP, which has most probably just crossed a page (4KB) boundary. You can confirm that this is a fatal page fault by looking to see whether paging is safe and by checking the target address 00123456 using "wh" and "dm".

A third form of fatal page fault is reported as:

```
Bus Error accessing 00123456 at PC 00123456
Unable to access that address
```

In this case, the actual instruction fetch has caused a bus error. You can confirm that this is a fatal page fault by looking to see whether paging is safe and by checking the target address using "wh".

Note

These last two examples may also be complicated by the DR emulator. Remember that you can trust the "accessing" address reported but not the PC.

Is This a Double Page Fault?

Not all fatal page faults are double page faults. For example, if you take a page fault while the device driver that's controlled the backing store is busy, the page fault is fatal, even though it isn't a double page fault. See the "Preventing Fatal Page Faults" section for a description of the various reasons why a page fault might be fatal.

Determining whether a fatal page fault is a double page fault is reasonably tricky. One good indicator is whether you're in user or supervisor mode. You can tell this by looking at the S bit in the SR display in MacsBug. If this is a capital "S", you are in supervisor mode; if this is a lower case "s" you are in user mode.

You can't get double page faults from user mode. If you get a fatal page fault and you're in user mode, you know there was some other cause. One good place to start debugging this is to use MacsBug's "drvr" command to see if any of the paging device drivers are busy. Remember that you will suffer a resource constraint fatal page fault if you take a page fault while the paging device driver is busy.

Unfortunately, taking a fatal page fault in supervisor mode still isn't a guarantee that it was a double page fault. The only certain way to determine whether it was a double page fault is to dump memory starting at the ISP and look for a bus error exception stack frame on the interrupt stack. Bus error exception stack frames are relatively easy to recognize because they contain a special word that denotes the frame type. For a bus errror on a 68040, this value is $7008. For a bus error on the 68020, 68030, and the emulated 680x0 processor of a PowerPC-based computer, this value is $B008.

Note

Under rare circumstances is is possible for the 68020 and 68030 processors to generate a frame type of $A008 in response to a bus error.

You can also use other fields in the exception frame to confirm that you have found the correct frame. All of the exception stack frames we're interested in share a common format, as shown below:

```
        SP + $00 - Status Register
        SP + $02 - High Word of PC
        SP + $04 - Low Word of PC
        SP + $06 - Frame Format, Vector Offset (eg $B008)
        ... and so on
```

Once you have found the frame type word on the stack, you can look back six bytes to find the value of the Status Register (SR) immediately prior to the bus error. Common values for SR are $0Ixx or $2ixx, where I is the interrupt level (0 to 7) and xx is "don't care". You can look at this saved SR, see whether its value is sensible, and use that to confirm whether you have found the bus error exception frame.

Once you find the bus error exception stack frame, you can use the information in M68000 Family Programmer's Reference Manual to examine the frame and find more clues about the cause.

Important

On PowerPC-based Mac OS computers, the value for the PC stored in the bus error exception frame is always incorrect. The interaction between the PowerPC processor and the Virtual Memory Manager causes this PC address to always point to an address within VM itself.

What Was the First Page Fault?

Once you know you've taken a double page fault, you can find out information about the first page fault by looking at the bus error exception stack frame. The format of this frame is described in the M68000 Family Programmer's Reference Manual. For example, a bus error exception stack frame on a 68040 looks like:

```
        SP + $00 - Status Register
        SP + $02 - High Word of PC
        SP + $04 - Low Word of PC
        SP + $06 - Frame Format, Vector Offset (contains $7008)
        SP + $08 - High Word of Effective Address
        SP + $0A - Low Word of Effective Address
        ... and so on
```

Once you find the start of the frame, you can dump the long at offset $02 from the start of the frame to determine, subject to the restrictions described in the previous section, the address of the instruction that took the original bus error.

You can also dump the long at offset $08 from the start of the frame to determine the address that the code was trying to access when it bus errored.

Finally, you can dump the word at offset $00 from the start of the frame to determine the value of the SR when the bus error occurred. This can be useful to determine the type of code that was running at the time. For example, interrupt level 4 is used by the Macintosh Serial Communications Controller (SCC) and it's likely that a fatal page fault that happens when the processor is at interrupt level 4 is somehow related to serial code.

Read-Only Memory Exceptions

When VM is enabled, it maps CFM containers into file mapping space. These file mapped address ranges are read-only. Any attempt to write to your own code will cause you to drop into MacsBug with the message:

```
        PowerPC read-only memory exception at 02314BB8 main+00018
```

For example, the following snippet of PowerPC code runs just fine when VM is disabled, but dies under VM:

```
static void Wibble(void)
{
}

void main(void)
{
        char x;

        x = **((char **) Wibble);
        **((char **) Wibble) = x;
}
```

Note

The extra dereference in the above snippet is required because procedure pointers on a CFM architecture are actually pointers to transition vectors.

New System Errors

System 7.5.5 introduced two new system errors related to virtual memory. Both errors are completely fatal for the system, but if you encounter one while debugging, it is useful to know their cause.

`dsVMDeferredFuncTableFull` (112)

> This error is generated when the table of deferred user function table is full. A common way of getting this error is to defer an operation that has already been deferred. Another possibility is to install a Time Manager task which has already been installed.
> If you get this error, you can use the "UsersFns" `'dcmd'` (part of MacsBug 6.5.4a1 and later) to dump out the list of deferred user functions. If you see one entry repeated many times, you should start looking for problems in how you use the system service corresponding to that entry. Specifically, if you find that there the list of deferred user functions is full of Time Manager user functions, check that you are calling `RmvTime` for each time you call `InsTime` (or `InsXTime`). While imbalanced calls to the Time Manager work when VM is disabled, they are not correct and they will cause this system error when VM is enabled.

`dsVMBadBackingStore` (113)

> This error is generated when VM gets an error while reading or writing a backing store. Typically, this indicates a genuine hardware problem. It could be a useful debugging aid if you are developing a paging device driver.

See Technote 1069 - "System 7.5.5" for more details on all of the changes that occurred in the System 7.5.5 release of VM.

MacsBug `'dcmd's`

MacsBug contains two `'dcmd'`s that can be useful when debugging VM problems. The first, "VMDump", dumps the current state of VM on a page-by-page basis. The second, "UserFns", displays a the current list of deferred user functions.

For information about these `'dcmd'`s, type the following commands in MacsBug:

```
help vmdump
help userfns
```

Summary

Virtual memory is not rocket science. While its implementation on the Mac OS is complicated by the constraints of the original design, it's not incomprehensible. Understanding how VM works will help you know why the rules are important, recognize when you need to apply them, and debug problems when they arise.

Further References

- *Inside Macintosh: Memory* provides a description of the VM API calls.
- Technote ME 09 - "Coping With VM and Memory Mappings" Although the information in ME 09 is supplanted by this Technote for application code, ME 09 still contains useful information for traditional device driver (DRVR) writers.
- *Designing PCI Cards and Drivers for Power Macintosh Computers* describes the VM rules that apply to native drivers ('ndrv's)
- *M68000 Family Programmer's Reference Manual* gives detailed information about the 680x0 exception architecture, including the exception vector table and the bus error exception frame.
- *Inside Macintosh: PowerPC System Software* gives a description of the differences between an emulated 680x0 processor and a real one, including the bus error exception frame format of the emulated variety.
- Technote 1063 - *"Inside Macintosh: Processes*: Time Manager Addenda" describes how to mark your Time Manager task "VM immune".
- Technote 1069 - "System 7.5.5" describes how to mark your Time Manager task "VM immune".
- Technote NW 13 - "AppleTalk: The Rest of the Story" describes how to mark your device "VM immune".
- Technote ME 06 - "_StripAddress: The Untold Story"
- *Modern Operating Systems*, by Andrew S. Tanenbaum, Prentice-Hall, 1992, ISBN 0-13-588187-0 gives a good introduction to virtual memory in general.

Acknowledgments

Thanks to Jim Luther, Eric Anderson, Bo3b Johnson, and Jeff Robbins.