Contents

How VM Works

This section describes how VM works, both the theory and the specific implementation under System 7.

Theoretical Background

The basic idea behind VM is that the system creates a logical address space that is larger than the installed physical memory and divides it up into uniformly-sized chunks of memory called pages. Under System 7-style virtual memory, each page is 4KB in size. Each page in the logical address space has a corresponding page on the disk, in a special file known as the backing store. The system then uses the computer's physical memory to give the illusion that the entire logical address space is made up of real memory.

Important
You should never assume the page size is 4K. PCI device drivers should call the Driver Services Library routine `GetLogicalPageSize` to find the page size. Other code should call `Gestalt` with the `gestaltLogicalPageSize` selector to determine the page size.

There are two key features of the processor and its memory management unit (MMU) that you must grasp in order to understand how VM works. The first is the page table. This is a table that maps all logical pages into their corresponding physical pages. When the processor accesses a logical address, the

MMU uses the page table to translate the access into a physical address, which is the address that's actually passed to the computer's memory subsystem.

Note:

Virtual Memory is only available when the processor has a Memory Management Unit (MMU). All 68030, 68040 and PowerPC processors have an MMU. Some 68020 computers have a slot where an optional MMU can be installed. 68000 processors do not support VM.

Important
You should never assume the state of VM based on the processor type. Always use `Gestalt` to determine whether VM is operating, as outlined in *Inside Macintosh: Memory* .

VM also utilitizes the processor's ability to stop executing instructions when a page translation fails. Because the logical address space is bigger than the physical memory of the computer, it stands to reason that not all logical pages have a corresponding physical page. When the processor accesses a logical address that is not in physical memory, the processor stops executing normal code and starts executing the special code to handle this page fault. The page fault handler is responsible for finding a free page of physical memory (writing the previous contents back to the backing store if necessary), reading the contents of the page from the backing store into the physical page, and then changing the logical to physical mapping so that the page appears to be at the right logical address. This process is known as paging.

If the page fault handler cannot successfully handle a page fault, the system crashes. This situation is known as a fatal page fault. The most common type of fatal page fault is the double page fault. Double page faults occur because the Mac OS is incapable of handling a page fault while it is already in the process of handling a page fault. Any such page fault is automatically considered a fatal page fault.

One way to think of VM is that it uses the real memory in the computer as a cache for the entire logical address space. Real memory can be accessed quickly, while memory that's paged out takes a long time to access. Thus, a page fault is analogous to a cache miss. Like all caches, VM relies on locality of reference for speed.

For more information about virtual memory systems in general, consult any operating systems text. I recommend *Modern Operating Systems* by Andrew S. Tanenbaum.

System 7-Style Virtual Memory

System 7-style VM on the Mac OS operates as if the machine was a 680x0 processor. For machines with PowerPC processors, the nanokernel and the 680x0 emulator perpetrate the illusion that the machine has a 680x0 processor. While this section describes the VM system running on 680x0 processors, the same details apply on PowerPC-based computers.

When the 680x0 realizes that there is no page translation for a particular memory access (i.e., it takes a page fault), it raises a bus error exception. In response to a bus error exception, the processor saves the state of the currently executing instruction on the stack, and propagates the error to the operating system by calling the bus error handler. It finds the bus error handler by consulting the exception vector table, whose address is held in a special processor register, Vector Base Register (VBR).

For example, if VBR contains `$000A44F6`, the exception vector table would be:

```
Address 0A44F6 contains 00A48F6 -- ISP for Reset handler
Address 0A44FA contains 00A48F6 -- address of Reset handler
Address 0A44FE contains 00A4E9E -- address of Bus Error handler
Address 0A44F2 contains 00A48F6 -- address of Address Error handler
Address 0A44F6 ... and so on.
```

Note:

> In the above listing, the third entry in the exception vector table is labelled as the "Bus Error Handle". The correct nomenclature is "Access Fault Handler". Bus errors are just one form of access fault that can be generated by 680x0 processors. Other types of access faults, such as write protect faults, are possible, but they are not discussed in this note.

For a full description of the exception vector table, consult M68000 Family Programmer's Reference Manual.

When VM is not enabled, VBR is set to zero, so the bus error handler is at address `$08` in low memory. When VM starts up, it switches VBR to point to its own private vector table. Most of the entries in VM's vector table propagate the exception through to the corresponding vector in the pseudo-vector table that's maintained in low memory. This allows existing programs that modify the low memory vector table to continue to operate under VM.

However, VM's bus error handler does more than just propagate the exception. The VM bus error handler first calculates the logical address that was being accessed to cause a bus error. If this address doesn't fall within any of the ranges being controlled by VM, it propagates the bus error through the low memory bus error pseudo-vector. This means that existing bus error handlers (such as MacsBug and the Slot Manager) can continue to modify this pseudo-vector, even when VM is running.

If the address falls within a range that is under its control, VM must handle the page fault. First, it finds a free page to hold the page that is about to be mapped in. If there are no free pages, it will discard some existing page in favor of the new one. VM discards a page by writing out the page contents to the backing store (if they have been changed) and marking that page "on disk".

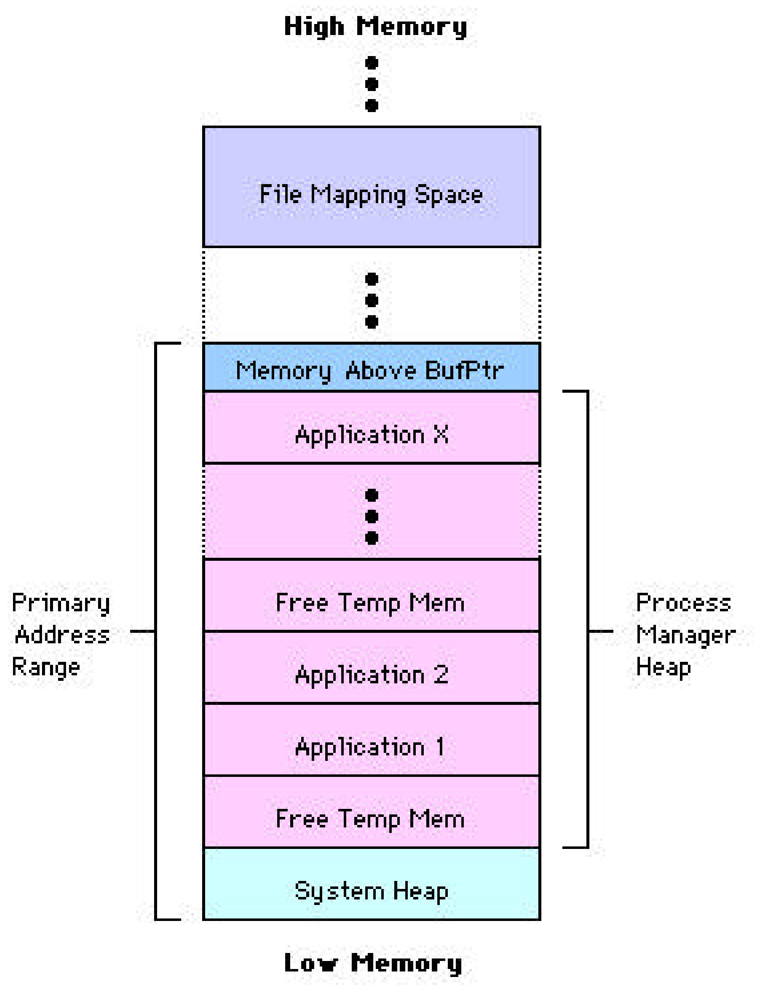
Once it has a free page, VM calculates the offset of the memory access from the base of the address range, and uses that offset to index into the backing store for that range. It then reads the correct page from the backing store into the free page, and modifies the page table to reflect the new mapping from logical to physical memory. It then returns (using the `RTE` instruction) from the bus error handler, which restarts the instruction that caused the page fault.

One can think of the bus error handler as a concrete implementation of the abstract page fault handler described in the previous section.

One thing to note is that, if VM is already handling a page fault when another page fault happens, it simply propagates the second page fault through the bus error pseudo-vector. The code that caused the second page fault will then bus error. This is a double page fault, and it usually results in a fatal system error.

Address Ranges Controlled by VM

So what are the address ranges controlled by VM? Under System 7-style VM, the logical memory map looks something like this:



I've left ROM, I/O and slot space out of this diagram because their locations vary between machines.

VM controls two major classes of address ranges. The first class only has one instance, the primary address range, which makes up the normal memory of the Mac OS. It encompasses the system heap, the Process Manager heap (from which application heaps are allocated), and the memory above `BufPtr`. The backing store for the primary address range is a file called "VM Storage" in the root directory of the volume which you configure in the Memory control panel.

The second class of address ranges occur when Code Fragment Manager (CFM) containers are loaded. Rather than load CFM containers into the primary backing store, VM simply maps them into an area of logical addresses called file mapping space. These file mapped address ranges don't need space in the primary backing store because they are swapped directly to the containers' data fork. This means that loading CFM containers does not take space out of the Process Manager heap. This is why the partition size for CFM applications decreases when you turn VM on.

Originally, CFM file mapping only occurred for PowerPC containers, but Mac OS 7.6 introduced file mapping for CFM-68K containers as well.

File-mapped address ranges differ from the primary address range in that they are read-only. Any attempt to modify memory in a file-mapped address range will cause an access fault exception.

Creating an Address Range

When VM creates an address range -- either the primary address range that's created at system startup time, or the file mapped address ranges that are created when CFM containers are loaded -- it is always associated with a backing store file. This file must have certain attributes:

1. It must be on an HFS volume.
2. The drive controlling the volume, known as the paging device, must not be marked 'ejectable'.
3. The paging device must be read/write (only for the primary address range).

When it creates an address range, VM builds a mapping from logical address in memory to offset in the backing store. To do this, VM talks privately to the HFS file system to map each extent of the file to its corresponding block number on the paging device. This allows VM to read and write the backing store using direct calls to the paging device, instead of going through the File Manager. This speeds things up and keeps the File Manager out of the VM equation.

Holding and Locking

Obviously, there must be some way of controlling whether a chunk of memory can be paged or not. Otherwise imagine what would happen when VM pages its own code out to disk! VM offers a number of routines, documented in chapter 3 of *Inside Macintosh: Memory* , that control whether certain memory can be paged or not.

Note:

The virtual memory chapter of *Inside Macintosh: Memory* makes a lot more sense if you read "hardware interrupt" wherever you see "interrupt".

Note:

System 7.5.5 introduced a new VM API routine, `LockMemoryForOutput`. This call is typically of interest to people writing DMA-capable paging device drivers (or the plug-ins that support them, such as a SCSI SIMs). You can find out more about `LockMemoryForOutput` from Technote 1069 "System 7.5.5".

These calls provide two important memory states. Memory can either be held or locked. When you hold memory, VM will not page it out to disk. When you lock memory, VM will not page it out to disk *and* avoid changing the logical to physical mapping for that memory.

Memory pages that are currently in physical memory are described as being resident. Another way of thinking about holding memory is that it ensures that the memory is resident until you unhold it.

Do not confuse locking memory, in the VM sense, with locking a handle. They are totally different.

Also, do not confuse holding with locking. It's rare for you to need to hold memory but it's *extremely rare* for you to need to lock it. Locking memory is only necessary when you need to access memory using its physical address. Typically the only people who need to do this are people writing Direct Memory Access (DMA) device drivers.

Another good reason to avoid locking memory when you should merely be holding it is that `LockMemory` disables the processor's caches for that range of memory. This is necessary for the sort of operations that `LockMemory` was designed for, namely DMA devices, but it seriously reduces the processor's performance when accessing the locked memory.

User vs Supervisor Mode

Important
Remember that a PowerPC-based Mac OS computer emulates a 680x0 computer sufficiently well that the 680x0 processor mode is still relevant even on a PowerPC computer. Although the PowerPC processor has a native processor mode, the PowerPC computer runs almost entirely with its PowerPC processor in user mode. As far as the Virtual Memory Manager is concerned, the only significant processor mode is the one maintained by the emulated 680x0 processor.

Another aspect of the Virtual Memory Manager that deserves mention is the 680x0 processor mode. When VM is disabled, all code on the Mac OS runs with the in supervisor mode. Turning on VM causes this behavior to change. When VM is turned on, the vast bulk of code is executed in user mode. This is done to prevent double page faults on the stack.

680x0 processors that are capable of VM have two stack pointers. The User Stack Pointer (USP) is used when the processor is in User mode and the Interrupt Stack Pointer (ISP) is used when the processor is in Supervisor mode. [There is in fact a third stack pointer, the Master Stack Pointer (MSP), but this is never used under Mac OS.] When the processor takes an exception, for example a page fault, the processor automatically switches to Supervisor mode, and hence switches on to the ISP.

Switching to the ISP is necessary because the act of taking a page fault exception requires the processor to push the processor state (the exception stack frame) on the stack. If the processor used the USP to push this information, there's a possibility that the user stack might be paged out and the processor might take a page fault in the attempt. This double bus fault is irrecoverable and causes the processor to unconditionally halt.

Switching to the ISP while taking a page fault avoids this problem because the stack that the ISP points to (the interrupt stack) is always held resident.

If your application switches to its own private stack at interrupt time (usually this is done to guarantee a minimum amount of available stack space), it must make sure that the stack is held resident in memory. Otherwise your interrupt time code might cause a page fault while allocating stack space. The processor will switch to the Interrupt Stack Pointer (which it's already on, and which points to a non-resident page) and attempt to create the exception stack frame. The result is a double bus fault, an unrecoverable error condition.

Privileged Instruction Emulation

Because VM runs the 680x0 processor (or the emulated processor under PPC -- remember that VM doesn't really distinguish between them) in User mode, certain privileged instructions are not available to applications. Executing such an instruction causes a 680x0 privilege violation exception.

A number of these instructions are commonly used by code that pre-dates VM. To maintain compatibility, the Virtual Memory Manager traps the privilege violation exception and emulates these instructions in software.

The most obvious example is any instruction that accesses the 680x0 Status Register (SR) register. The SR register is a privileged register on the 68010 or higher. Most people access the SR in order to enable and disable interrupts, and there is no better way. However, if you're accessing the SR in order to get or set condition flags, the 68010 and higher offer an alternative register, the Condition Code Register (CCR), which is not privileged. Unfortunately the CCR is not available on the original 68000, so if your program supports that processor, you have to conditionalize your access to CCR.

The list of 680x0 privileged instructions that are emulated by the Virtual Memory Manager includes:

```
MOVE.W #<data>,SR
ANDI.W #<data>,SR
EORI.W #<data>,SR
ORI.W  #<data>,SR

MOVE SR,Dn
MOVE SR,(An)
MOVE SR,-(An)
MOVE SR,-(SP)
MOVE SR,(d16,An)
MOVE SR,<xxx>.W

MOVE Dn,SR
MOVE (An),SR
MOVE (An)+,SR
MOVE (SP)+,SR
MOVE (d16,An),SR
MOVE <xxx>.W,SR

MOVE (d16,PC),SR

MOVES.W Rn,<xxx>.L
MOVES.W <xxx>.L,Rn

MOVEC CACR,Dn
MOVEC Dn,CACR

FRESTORE (an)+
FSAVE -(an)

RESET

RTE
```

There are also a number of other emulated instructions related to MMU and cache control. These are not documented here because it's unlikely you will be using them.

Note
The instruction to get the VBR register (e.g., `MOVEC VBR,An`) is not emulated by VM.

Privileged instruction execution is one of the places where the VM implementation on PowerPC is different from that on 680x0. When VM is running on a PowerPC-based Mac OS computer, it informs the 680x0 emulator to automatically emulate the privileged instructions that would otherwise be emulated by VM. This minimizes the slowdown associated with executing privileged instructions under VM on a PowerPC system.

Back To Table of Contents

VM Implementation Details

System 7-style VM has five key implementation details that you need to know about. This section covers them one at a time and describes how their solutions affect the parts of VM that are visible to the programmer.

Preventing Fatal Page Faults

The key implementation difficulty with System 7-style VM is preventing fatal page faults. There are three categories of fatal page faults:

1. double page faults -- A double page fault occurs if the system takes a page fault while it's already handling a page fault. This was an original design restriction of System 7-style VM.
2. resource constraint fatal page faults -- A resource constraint page fault occurs if the system takes a page fault when it does not have the resources to handle the fault. An example of a resource constraint fatal page fault is a page fault that happens while the paging device is busy. Because Mac OS device drivers are only capable of handling one request at a time, VM will not be able read the disk if the paging device is already in use. VM specifically guards against this possibility by making page faults fatal while the paging device is busy.
3. true fatal page faults -- A true fatal page fault is one that is fatal for a really good reason, such as a read or write error on the backing device.

While there is no way of avoiding true fatal page faults, VM does its best to prevent the other kinds.

VM takes two different but related approaches to preventing these fatal page faults:

- First, VM stops execution of all code that might cause a page fault while it is handling a page fault (to prevent double page faults) and while the paging device is busy (to prevent resource constraint page faults).
- Second, VM requires that paging devices not cause page faults in the process of handling a read or write request (to prevent resource constraint page faults).

In VM terminology, code that might cause a page fault is called user code and the action of preventing it from running is called disabling user code. Disabling user code is particularly tricky on the Mac OS because application programmers have access to many interrupt level services, like Time Manager and Vertical Retrace Manager. If a hardware interrupt happens while the system is handling a page fault, and this hardware interrupt calls some application code that takes another page fault, you have a double page fault and the system crashes. The exact details of how user code is disabled is the subject of a later section.

Running Old Drivers

System 7 VM was designed in such a way that the vast majority of device driver writers did not have to rewrite their drivers to be compatible with VM. This was done using a combination of techniques:

1. The entire system heap is held. This prevents device drivers causing a page fault when fetching their code or accessing their own internal data structures at hardware interrupt time.
2. The Device Manager routines `_Read`, `_Write`, `_Control` and `_Status` were patched to hold down all parameter blocks passed to device drivers. In addition, the `_Read` and `_Write` patches hold down the memory pointed to by the `ioBuffer` field of `_Read` and `_Write` parameter blocks. The patches also hold down 2 KB of stack space below the stack pointer. This means that device drivers can access the parameter block, the buffer pointed to by the parameter block, and the stack, without causing a page fault.

See the later section "Device Manager `ioCompletion` Routines" for a in-depth description of these patches.

Synchronous SCSI Manager

Another key problem implementing System 7 VM was the nature of the original Mac OS SCSI Manager. Prior to asynchronous SCSI Manager (a.k.a. SCSI Manager 4.3), disk device drivers operated using the following algorithm:

1. Grab the SCSI bus exclusively
2. Do some SCSI operation
3. Release the SCSI bus

The problem here is that the Virtual Memory Manager needs the SCSI bus in order to handle page faults. So if an interrupt occurs while the disk device driver has exclusive access to the SCSI bus, the Virtual Memory Manager is vulnerable to resource constraint fatal page faults.

The solution to this was two-pronged. First, VM does special things (see the previous section) to ensure that device drivers do not cause a page fault. Second, user code is disabled while the SCSI bus is busy, so interrupts cannot cause a page fault.

Asynchronous SCSI Manager

When the asynchronous SCSI Manager 4.3 was introduced, it brought with it a new problem. Under the current Virtual Memory Manager, it's quite common for user code to take a page fault while interrupts are disabled.

A good example of this is the OS Utilities routine `Enqueue`. This routine disables interrupts in order to ensure that the queue is modified atomically. However there's no guarantee that the act of enqueuing won't cause a page fault. If this happens, the Virtual Memory Manager will call the block device driver synchronously to read the data in from the backing store.

Unfortunately the asychronous SCSI Manager relies on interrupts to complete any asynchronous operations. As block device drivers are required to operate asynchronously even if the request is synchronous (see Technote 1067 "Traditional Device Drivers: Sync or Swim"), there's no way for the driver to avoid requiring interrupts enabled to complete its operation.

The asynchronous SCSI Manager gets around this by patching `vSyncWait` to poll the SCSI hardware looking for interrupts that are unnoticed because interrupts are disabled. It's yucky, but it works.

ATA Manager

When ATA (sometimes known as IDE) hard disks were introduced on Mac OS computers, the software for controlling them faced an identical set of problems to those faced by the SCSI software. The first ATA Manager and ATA disk drivers were synchronous, but these were later replaced by asynchronous versions. Thus, the ATA I/O system faces the same challenges running under VM as the SCSI I/O system. In general, the ATA I/O system uses the same mechanism as the SCSI I/O system to avoid getting into trouble with VM. So the above sections also work if you scratch out the word "SCSI" and write in "ATA".

Back To Table of Contents

Disabling User Code

This section describes the techniques used by VM to disable user code to avoid fatal page faults. This is the core of the System 7-style VM implementation.

What is User Code?

To avoid double page faults, the Virtual Memory Manager must disable user code when a page fault is in progress. The definition of user code is "any code that is allowed to cause a page fault". The designers of the Virtual Memory Manager were forced to decide what constitutes user code and what doesn't. They came up with the following list of things that are defined to be user code:

- Normal application-level code
- File Manager `ioCompletion` routines
- Device Manager `ioCompletion` routines
- Time Manager tasks
- Vertical Retrace Manager tasks (VBLs)
- Slot Manager VBLs (Slot VBLs)
- Deferred Tasks
- ADB service routines
- Patches `toPostEvent`
- any code invoked by any of the above

This list was largely determined by the question, "What things should we allow to page fault if we want to avoid breaking too many applications?"

In addition, many newer pieces of system software that use interrupts have been designed to support VM by deferring interrupt-based routines until paging is safe. These includes:

- Sound Manager callbacks
- Open Transport notifiers
- Network driver protocol handlers

The above lists are not complete. In general, it is more sensible to list what is not user code, rather than what is.

What Isn't User Code

The following things are not user code:

- SCSI Manager 4.3 completion routines
- ATA Manager completion routines
- AppleTalk socket listeners
- Slot interrupt handlers (installed using `SIntInstall`)
- Native Interrupt Manager interrupt handlers (as defined in Designing PCI Cards and Drivers for Power Macintosh Computers)
- Any other hardware interrupt handlers, such as those patched into the low memory interrupt vector tables

`DeferUserFn` -- The Guts of the Solution

The Virtual Memory Manager uses the concept of deferred user functions to defer user code while paging is not safe. A deferred user function is an operation that the Virtual Memory Manager traps when paging is not safe. The Virtual Memory Manager remembers the operation in a fixed size internal table of user functions. When paging becomes safe again, it re-enables user code, which runs the deferred user functions, thereby completing all the operations that had been deferred.

This mechanism is also available to third party programmers. The `DeferUserFn` routine, *Inside Macintosh: Memory* p3-33, allows you to defer an operation until paging is safe. You should only use it to transition from non-user code to user code. This means that application programmers rarely need to invoke this function because application programmers are rarely running non-user code. See the Programming Implications section for a list of circumstances under which you might consider using `DeferUserFn`.

Important:

Do not call `DeferUserFn` from code that has already been deferred by VM. Doing is likely to result in a fatal system error.

Do not confuse deferred user functions with deferred tasks as defined by the Deferred Task Manager. A deferred task allows hardware interrupt-level code to schedule a routine to be called when interrupts are re-enabled. This is useful for deferring lengthy calculates that would otherwise keep interrupts disabled for a long period of time. A deferred user function is used to schedule a routine to be called when paging becomes safe. It's a question of using the right tool for the job!

Correct Way To Defer

If you do use `DeferUserFn` to defer processing of a periodic event until paging is safe, you should be careful about how you use it. Remember that the table of deferred user functions is of fixed size. Using up all of the entries results in a fatal system error. This is especially dangerous when you are deferring periodic events, such as a retrace interrupt, timer interrupts, or frequently arriving packets.

For example, imagine a computer whose hard disk has spun down. If the system take a page fault, the computer will be forced to wait for a number of seconds while the hard disk spins up to service the page fault. Page faults will be fatal for this entire time. If you have a high frequency interrupt, such as a packet handler that was using a user function table entry *for each packet* , you might easily overflow the user function table during this time.

In such cases, the correct way to use `DeferUserFn` is as follows:

- Create a global flag that says whether your user function has been installed.
- When you defer an operation, check the flag first. If your user function has not been installed, call `DeferUserFn` to install it and set the flag. If your user function has already been installed, put this operation on a queue where your user function can get to it.
- When your user function runs, first clear the flag and then do all the operations on your queue. It is important that you clear the flag before doing the operations to avoid a race condition with your hardware interrupt handler.

How VM Disables User Code

The mechanism that VM uses to disable user code is dependent on the situation. The following sections describe the various mechanisms used.

Remember that these operations are automatically deferred by VM. You do not need to do anything special to get this automatical deferral.

Normal Application-Level Code

The traditional Mac OS is a co-operatively scheduled system. This means that while one process is blocked waiting for a page fault to be handled, there is no possibility of another process executing by way of a task switch. Thus VM does not need to do anything special to disable general application code while page faults are fatal.

Device Manager `ioCompletion` Routines

VM patches the Device Manager routines `_Read`, `_Write`, `_Control`, and `_Status`. These patches perform the following actions:

1. Hold the parameter block.
2. If the call is `_Read` or `_Write`, hold the buffer pointed to by `ioBuffer` and `ioReqCount`.
3. Hold the top 2048 bytes of stack.
4. If the call is asynchronous, replace the caller's `ioCompletion` routine with the address of the VM's completion routine. Remember the caller's original `ioCompletion` in an internal table.
5. In the asynchronous case, when VM's completion routine is called, it first undoes the holding done in Step 1 - 3. It then puts the original `ioCompletion` address back into the parameter block. Then it determines if paging is safe and if so, it calls the original `ioCompletion` routine. If paging is not safe, execution of the original `ioCompletion` routine is deferred until it is.
6. If the call is being made to a paging device, disable user code, call through to the previous implementation of the trap, then enable user code. Otherwise, just call through to the previous implementation of the trap.
7. If the call was synchronous, undo the holding done in Steps 1 - 3.

These patches ensure that:

- pre-VM device drivers do not cause a fatal page fault when accessing the parameter block, the data pointed to by the parameter block, or the stack.
- Device Manager `ioCompletion` routines (and anything that they might call, most notably the File Manager) are deferred until paging is safe.

Time Manager Tasks

VM patches `_InsTime`, `_PrimeTime` and `_RmvTime` to ensure that neither the Time Manager nor installed timer tasks will cause a fatal page fault. The patches do the following:

- `_InsTime` (which also includes `_InsXTime`)
    1. Hold the `TMTask` record.
    2. Store the address of the `TMTask` record in an internal table.
    3. Call the original `InsTime` code.
- `_PrimeTime`
    1. Store the task routine address from `TMTask.tmAddr` in an internal table and replaces `TMTask.tmAddr` with the address of VM's task routine.
    2. Call the original `_PrimeTime` code.
- When the VM task routine is executed, it first replaces `TMTask.tmAddr` with the original task routine address. Then, if paging is safe, it executes the original task routine. If paging is not safe, execution of the original task routine is deferred until it is.
- `_RmvTime`
    1. Remove the address of the `TMTask` record from the internal table.
    2. Unhold the `TMTask` record passed to `_RmvTime`.
    3. Jump to the original `_RmvTime` code.

Vertical Retrace Manager Tasks (VBLs)

VM prevents VBLs from running by marking the VBL queue as busy. The VBL queue busy flag is used by the Vertical Retrace Manager (VRM) to prevent re-entrancy. When a VBL fires, the VRM looks at the VBL queue to see if it's busy. If it is, it presumes this is a re-entrant VBL and simply quits. Thus VBLs are never executed unless paging is safe.

Slot Manager VBLs (Slot VBLs)

VM patches the VRM routine `DoVBLTask` to:

1. If paging is unsafe, empty the list of VBLs for the specified slot.
2. Call the original trap.
3. Restore the original list of VBLs for that slot.

This approach allows normal slot operations (ie the update of `SlotTicks`) to continue while preventing Slot Manager VBLs from causing fatal page faults.

Note
Do not confuse Slot Manager VBLs, as described in *Inside Macintosh: V* , with Video Services Library (VSL) video interrupts, as described in Designing PCI Cards and Drivers for Power Macintosh Computers. VSL interrupt handlers can be called when paging is unsafe.

Deferred Tasks

The Deferred Task Manager, like the Vertical Retrace Manager, uses the `busy` bit in the VBL queue to prevent re-entrancy. So when VM disables VBLs, it also disables Deferred Tasks.

ADB Service Routines

VM patches the `_ADBOp` to do the following:

1. If all of the addresses are in ROM or the system heap, call through to the old trap.
2. Wait for the previous ADB operation, if any, to complete.
3. Replace the ADB completion routine with VM's own completion routine. Remember the original

completion routine in a global buffer.
4. Hold the ADB data buffer.
5. Call the old trap.
6. When the VM's ADB completion is called, unhold the data buffer and use `DeferUserFn` to schedule another VM routine.
7. When that is called, paging is safe. Restore the original completion routine and call it.

This ensures that the `_ADBOp` completion routine is always called at VM safe time and that the ADB Manager does not touch unheld memory at interrupt time.

PostEvent Patches

VM patches `_PostEvent` for an odd reason, namely that many other programs patch it. `PostEvent` is regularly called at non-deferred hardware interrupt time, and there's a possibility that the other patches on `_PostEvent` could cause a fatal page fault.

When `_PostEvent` is called, VM patch checks whether paging is safe. If it is, it calls directly through to the old trap. If paging isn't safe, VM defers the `_PostEvent` operation until paging is safe.

Classic SCSI Manager

VM patches the following classic SCSI Manager routines:

- `SCSIGet`
    - The VM patch on `SCSIGet` disables user code, to prevent a resource constraint fatal page fault while the SCSI bus is busy.
- Other classic SCSI Manager routines
    - VM patches these routines so that it can re-enable user code when the SCSI bus is free again.

End of this section